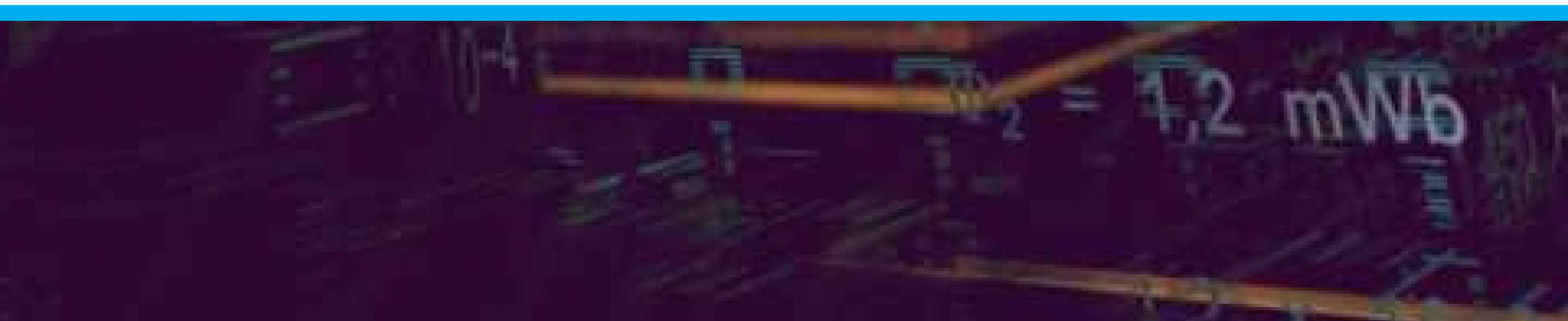
Algorithm Software in Technology Design

Sidney Albert



ALGORITHM SOFTWARE IN TECHNOLOGY DESIGN

ALGORITHM SOFTWARE IN TECHNOLOGY DESIGN

Sidney Albert

BIBLIOTEX
Digital Library

Algorithm Software in Technology Design
by Sidney Albert

Ebook ISBN: 9781984664037

Published by:
Bibliotex
Canada
Website: www.bibliotex.com

Contents

1

Designing of Algorithms

OVERVIEW

At first glance, the algorithms move-until-out and quick-sort have little in common. One processes structures; the other processes lists. One creates a new structure for the generative step; the other splits up a list into three pieces and recurs on two of them. In short, a comparison of the two examples of generative recursion suggests that the design of algorithms is an ad hoc activity and that it is impossible to come up with a general design recipe. A closer look, however, suggests a different picture.

First, even though we speak of algorithms as processes that solve problems, they are still functions that consume and produce data. In other words, we still choose data to represent a problem, and we must definitely understand the nature of our data if we wish to understand the process.

Second, we describe the processes in terms of data, for example, “creating a new structure” or “partitioning a list of numbers.” Third, we always distinguish between input data for which it is trivial to produce a solution and those for which it is not.

Fourth, the generation of problems is the key to the design of algorithms. Although the idea of how to generate a new problem might be independent of a data representation, it must certainly be implemented for whatever form of representation we choose for our problem. Finally, once the generated problems have been solved, the solutions must be combined with other values. Let us examine the six general stages of our structural design recipe in light of our discussion:

Data Analysis and Design

The choice of a data representation for a problem often affects our thinking about the process. Sometimes the description of a process dictates a particular choice of representation. On other occasions, it is possible and worthwhile to explore alternatives. In any case, we must analyze and define our data collections.

Contract, Purpose, Header

We also need a contract, a definition header, and a purpose statement. Since the generative step has no connection to the structure of the data definition, the purpose statement should not only specify what the function does but should also include a comment that explains in general terms how it works.

Function Examples

In our previous design recipes, the function examples merely specified which output the function should produce for some given input. For algorithms, examples should illustrate how the algorithm proceeds for some given input. This helps us to design, and readers to understand, the algorithm.

For functions such as move-until-out the process is trivial and doesn't need more than a few words. For others, including, quick-sort, the process relies on a non-trivial idea for its generative step, and its explanation requires a good example such as the one in figure.

Template

Our discussion suggests a general template for algorithms:

```
(define (generative-recursive-fun problem)
  (cond
    [(trivially-solvable? problem)
    (determine-solution problem)]
    [else
    (combine-solutions
    ... problem ...
    (generative-recursive-fun (generate-problem-1 problem))
    .
    .
    .
    (generative-recursive-fun  (generate-problem-n
problem)))]))
```

DEFINITION

Each function in the template is to remind us that we need to think about the following four questions:

1. What is a trivially solvable problem?
2. What is a corresponding solution?

3. How do we generate new problems that are more easily solvable than the original problem? Is there one new problem that we generate or are there several?
4. Is the solution of the given problem the same as the solution of (one of) the new problems? Or, do we need to combine the solutions to create a solution for the original problem? And, if so, do we need anything from the original problem data?

To define the algorithm, we must express the answers to these four questions in terms of our chosen data representation.

TEST

Once we have a complete function, we must also test it. As before, the goal of testing is to discover bugs and to eliminate them. Remember that testing cannot validate that the function works correctly for all possible inputs.

TERMINATION

Unfortunately, the standard recipe is not good enough for the design of algorithms. Up to now, a function has always produced an output for any legitimate input. That is, the evaluation has always stopped. After all, by the nature of our recipe, each natural recursion consumes an immediate piece of the input, not the input itself. Because data is constructed in a hierarchical manner, this means that the input shrinks at every stage. Hence the function sooner or later consumes an atomic piece of data and stops. With functions based on generative recursion, this is no longer true. The internal recursions don't consume an immediate component of the input but some new piece of data, which is generated from the input. As exercise shows, this step may produce the input over and over again and thus prevent the evaluation from ever producing a result. We say that the program LOOPS or is in an INFINITE LOOP.

In addition, even the slightest mistake in translating the process description into a function definition may cause an infinite loop. The problem is most easily understood with an example. Consider the following definition of smaller-items, one of the two "problem generators" for quick-sort:

```
;; smaller-items: (listof number) number -> (listof number)
;; to create a list with all those numbers on alon
;; that are smaller than or equal to threshold
(define (smaller-items alon threshold)
  (cond
    [(empty? alon) empty]
    [else (if (<= (first alon) threshold)
      (cons (first alon) (smaller-items (rest alon)
threshold))
      (smaller-items (rest alon) threshold))]))
```

Instead of < it employs <= to compare numbers. As a result, this function produces (list 5) when applied to (list 5) and 5.

Worse, if the quick-sort function from figure is combined with this new version of smaller-items, it doesn't produce any output for (list 5):

```
(quick-sort (list 5))
= (append (quick-sort (smaller-items 5 (list 5)))
     (list 5)
     (quick-sort (larger-items 5 (list 5))))
= (append
     (list 5)
     (quick-sort (larger-items 5 (list 5))))
```

The first recursive use demands that quick-sort solve the problem of sorting (list 5)—but that is the exact problem that we started with. Since this is a circular evaluation, (quick-sort (list 5)) never produces a result. More generally, there is no guarantee that the size of the input for a recursive call brings us closer to a solution than the original input. The lesson from this example is that the design of algorithms requires one more step in our design recipe: a TERMINATION ARGUMENT, which explains why the process produces an output for every input and how the function implements this idea; or a warning, which explains when the process may not terminate. For quick-sort, the argument might look like this:

At each step, quick-sort partitions the list into two sublists using smaller-items and larger-items. Each function produces a list that is smaller than the input (the second argument), even if the threshold (the first argument) is an item on the list. Hence each recursive application of quick-sort consumes a strictly shorter list than the given one. Eventually, quick-sort receives and returns empty.

Without such an argument an algorithm must be considered incomplete.

A good termination argument may on occasion also reveal additional termination cases. For example, (smaller-items N (list N)) and (larger-items N(list N)) always produce empty for any N. Therefore we know that quick-sort's answer for (list N) is (list N).

To add this knowledge to quick-sort, we simply add a cond-clause:

```
(define (quick-sort alon)
  (cond
    [(empty? alon) empty]
    [(empty? (rest alon)) alon]
    [else (append
      (quick-sort (smaller-items alon (first alon)))
      (list (first alon))
      (quick-sort (larger-items alon (first alon))))]))
```

The condition (empty? (rest alon)) is one way to ask whether alon contains one item.

EXERCISES

Exercise *1*

Define the function tabulate-div, which accepts a number n and tabulates the list of all of its divisors, starting with 1 and ending in n. A number d is a divisior of a number n if the remainder of dividing n by d is 0, that is, (= (remainder n d) 0) is true. The smallest divisior of any number is 1; the largest one is the number itself.

Structural vs. Generative Recursion

The template for algorithms is so general that it even covers functions based on structural recursion. Consider the version with one termination clause and one generation step:

```
(define (generative-recursive-fun problem)
  (cond
   [(trivially-solvable? problem)
   (determine-solution problem)]
  [else
   (combine-solutions
   problem
   (generative-recursive-fun (generate-problem problem)))]))
```

If we replace trivially-solvable? with empty? and generate-problem with rest, the outline is a template for a list-processing function:

```
(define (generative-recursive-fun problem)
  (cond
   [(empty? problem) (determine-solution problem)]
   [else
     (combine-solutions
     problem
     (generative-recursive-fun (rest problem)))]))
```

MAKING CHOICES

A user cannot distinguish sort and quick-sort. Both consume a list of numbers; both produce a list that consists of the same numbers arranged in ascending order. To an observer, the functions are completely equivalent. This raises the question of which of the two a programmer should provide. More generally, if we can develop a function using structural recursion and an equivalent one using generative recursion, what should we do? To understand this choice better, let's discuss another classical example of generative recursion from mathematics: the problem of finding the greatest common divisor of two positive natural numbers. All such numbers have at least one divisor in common: 1. On occasion, this is also the only common divisor. For example, 2 and 3 have only 1 as common divisor because 2 and 3, respectively, are the only other divisors.

Then again, 6 and 25 are both numbers with several divisors:

1. 6 is evenly divisible by 1, 2, 3, and 6;
2. 25 is evenly divisible by 1, 5, and 25.

Still, the greatest common divisior of 25 and 6 is 1. In contrast, 18 and 24 have many common divisors:

1. 18 is evenly divisible by 1, 2, 3, 6, 9, and 18;
2. 24 is evenly divisible by 1, 2, 3, 4, 6, 8, 12, and 24.

The greatest common divisor is 6.

Following the design recipe, we start with a contract, a purpose statement, and a header:

```
;; gcd: N[>= 1] N[>= 1] -> N
;; to find the greatest common divisior of n and m
(define (gcd n m)
...)
```

The contract specifies the precise inputs: natural numbers that are greater or equal to 1 (not 0).

Now we need to make a decision whether we want to pursue a design based on structural or on generative recursion. Since the answer is by no means obvious, we develop both. For the structural version, we must consider which input the function should process: n, m, or both. A moment's consideration suggests that what we really need is a function that starts with the smaller of the two and outputs the first number smaller or equal to this input that evenly divides both n and m.

Finding the greatest common divisor via structural recursion:

```
;; gcd-structural: N[>= 1] N[>= 1] -> N
;; to find the greatest common divisior of n and m
;; structural recursion using data definition of N[>= 1]
(define (gcd-structural n m)
  (local ((define (first-divisior-<= i)
    (cond
    [(= i 1) 1]
    [else (cond
      [(and (= (remainder n i) 0)
        (= (remainder m i) 0))
      i]
      [else (first-divisior-<= (- i 1))])])))
  (first-divisior-<= (min m n))))
```

We use local to define an appropriate auxiliary function: The conditions “evenly divisible” have been encoded as (= (remainder n i) 0) and (=(remainder m i) 0). The two ensure that i divides n and m without a remainder. Testing gcd-structural with the examples shows that it finds the expected answers. Although the design of gcd-structural is rather straightforward, it is also naive. It simply tests for every number whether it divides both n and m evenly and returns the first such number. For small natural numbers, this process works just fine. Consider the following example, however:

```
  (gcd-structural 101135853 45014640)
```

The result is 177 and to get there gcd-structural had to compare 101135676, that is, 101135853 - 177, numbers. This is a large effort and even reasonably fast computers spend several minutes on this task.

Exercise

Enter the definition of gcd-structural into the Definitions window and evaluate (time (gcd-structural 101135853 45014640)) in the Interactions window. After testing gcd-structural conduct the performance tests in the Full Scheme language (without debugging), which evaluates expressions faster than the lower language levels but with less protection. Add (require-library "core.ss") to the top of the Definitions window. Have some reading handy! Since mathematicians recognized the inefficiency of the "structural algorithm" a long time ago, they studied the problem of finding divisiors in more depth. The essential insight is that for two natural numbers larger and smaller, their greatest common divisor is equal to the greatest common divisior of smaller and the remainder of larger divided into smaller. Here is how we can put this insight into equational form:

```
    (gcd larger smaller)
  = (gcd smaller (remainder larger smaller))
```

Since (remainder larger smaller) is smaller than both larger and smaller, the right-hand side use of gcd consumes smaller first.

Here is how this insight applies to our small example:

1. The given numbers are 18 and 24.
2. According to the mathematicians' insight, they have the same greatest common divisor as 18 and 6.
3. And these two have the same greatest common divisor as 6 and 0.

And here we seem stuck because 0 is nothing expected. But, 0 can be evenly divided by every number, so we have found our answer: 6.

Working through the example not only explains the idea but also suggests how to discover the case with a trivial solution. When the smaller of the two numbers is 0, the result is the larger number. Putting everything together, we get the following definition:

```
  ;; gcd-generative: N[>= 1] N[>=1] -> N
  ;; to find the greatest common divisior of n and m
  ;; generative recursion: (gcd n m) = (gcd n (remainder m n))
if (<= m n)
  (define (gcd-generative n m)
    (local ((define (clever-gcd larger smaller)
      (cond
        [(= smaller 0) larger]
        [else (clever-gcd smaller (remainder larger
smaller))])))
    (clever-gcd (max m n) (min m n))))
```

The local definition introduces the workhorse of the function: clever-gcd, a function based on generative recursion. Its first line discovers the trivially solvable case by comparing smaller to 0 and produces the matching solution. The generative step uses smaller as the new first argument and (remainder larger smaller) as the new second argument to clever-gcd, exploiting the above equation If we now use gcd-generative with our complex example from above: `(gcd-generative 101135853 45014640)` we see that the response is nearly instantaneous. A hand-evaluation shows that clever-gcd recurs only nine times before it produces the solution: 177. In short, generative recursion has helped find us a much faster solution to our problem.

DESIGN PROCESS

For the maximum subarray problem, if you didn't know any better, you'd probably implement a solution that analyzes every possible subarray, and returns the one with the maximum sum:

```
class Array
    def sum
        result = 0
        self.each do |i|
            result+=i
        end
        return result
    end
    #test every sub array - brute force!
    def max_sub_array_order_ncubed
        left_index = 0
        right_index = 0
        max_value = self[left_index..right_index].sum
        for i in (0..self.length)
            for j in (i..self.length)
                this_value = self[i..j].sum
                if (this_value > max_value)
                    max_value = this_value
                    left_index=i
                    right_index=j
                end
            end
        end
        return self[left_index..right_index]
    end
end
```

If you were a bit more clever, you might notice that self[i..j].sum is equal toself[i..(j-1)].sum + self[j] in the innermost loop (the sum method itself), and use an accumulator there as opposed to calculating it each time. That takes you down from n^3 to n^2 time.

But there are (at least) two other ways to solve this problem:

1. A divide and conquer approach that uses recursion and calculates the left and right maximum contiguous subarrays (MCS), along with the MCS that contains the right-most element in the left side and the left-most element in the right side. It compares the three and returns the one with the maximum sum. This gets us to O(n log n) time.
2. An approach I'll call "expanding sliding window." If memory serves me correct, this aptly describes it or was the way a professor of mine described it. In any case, the "expanding sliding window" can do it in one pass (O(n) time), at the cost of a few more variables.

Clearly, these last two approaches aren't nearly as obvious as the first two - so how do you devise them? I'm fairly confident that the only reason I know about them is from a course in algorithms where they were presented to me (and I didn't take the time to work-through and reimplement them for this post). I'm not sure that TDD or just a long ponder would have led me in that direction. (Although, one of the solution submitters claims he TDDed the O(n) solution.)

Three thoughts in designing algorithms I use off the top of my head are:

- There's always brute force, but is there something better?

- Is divide and conquer and option? If so, is it easy enough to implement and understand?
- How can I trade space complexity for gains in time complexity, or vice versa if the situation warrants?

AMORTIZED ANALYSIS

DEFINITION

Amortized analysis means analyzing time-averaged cost for a sequence of operations. Motivation is that traditional worst-case-per-operation analysis can give overly pessimistic bound if the only way of having an expensive operation is to have a lot of cheap ones before it.

Note: this is DIFFERENT from our usual notion of "average case analysis" — we're not making any assumptions about inputs being chosen at random — we're just averaging over time.

The approach is going to be to somehow assign an artificial cost to each operation in the sequence. This artificial cost is called the _amortized cost_ of an operation. The key property required of amortized cost is that the total real cost of the sequence should be bounded by the total of the amortized costs of all the operations. Then, for purposes of analyzing an algorithm that, say, accesses a data structure, it is okay to just use the amortized cost instead of the actual cost of the operation. This will give you correct results.

Note: There is sometimes flexibility in the assignment of amortized costs.

There are going to be three approaches that we call:

The aggregate method The banker's method (tokens in the data structure) The physicist's method (potential functions) (The physicist's method is just a slightly more formal version of the banker's method, as we'll see.).

We'll illustrate these methods through some examples.

*Example*1: a binary counter. Say we want to store a big binary counter in an array A: Start all entries at 0. The operation we are going to implement and Analyse is that of counting.

The algorithm we'll use for incrementing this counter is the usual one. We toggle bit A[0]. If it changed from 0 to 1, then we toggle bit A[1], etc. We stop when a bit changes from 0 to 1. The cost of an increment is the number of bits that change.

A[m]	A[m-1]......	A[3]	A[2]	A[1]	A[0]	cost
0	0	0	0	0	0	1
0	0	0	0	0	1	2
0	0	0	0	1	0	1
0	0	0	0	1	1	3
0	0	0	1	0	0	1
0	0	0	1	0	1	2
0	0	0	1		1	0

The number of bits that change when the increment produces a number n is at most 1+floor(lg n). (That's just the number of bits in the binary representation of n.) Thus, in a sequence of n increments, worst-case cost per increment is bounded by n(1+floor(lg n)) = O(n log n).

But, what is our *amortized* cost per increment?

Answer: 2.

Proof 1 (aggregate method): how often do we flip A[0]?

Answer: every time. how often do we flip A[1]?

Answer: every other time. How often do we flip A[2]?

Answer: every 4th time. Etc. So, total cost spent on flipping A[0] is n, total cost of A[1] is floor(n/2), total cost on A[2] is floor(n/4), etc. So, the total cost is: total cost = n + floor(n/2) + floor(n/4) +... total cost ⇐ n + n/2 + n/4 + n/8 +... ⇐ 2n

So the total cost is 2n, which means the amortized cost of an increment is 2.

Proof 2 (banker's method): Let's us a kind of accounting trick. On every bit that is a 1, let's keep a dollar on that bit. So for example, if the current count is 6, we'd have:

```
                              $     $
array: 0....     0     1      1     0
```

We'll use the convention that whenever we toggle a bit, we must pay a dollar to do that.

Let's say we allocate $2 to do an increment. Let's see how much it costs to do the increment. In general, a bunch of low order bits change from 1 to 0, and then one bit changes from a 0 to a 1, and the process terminates. For each of the bits that changes from 1 to 0, we have a dollar sitting on the bit to pay for toggling that bit. For the bit that changes from a 0 to a 1, we have to pay a dollar to toggle the bit, then put a dollar on that bit (for future use).

Thus, having allocated $2 for the increment always guarantees that we will have enough money to pay for the work, no matter how costly the increment actually is. This completes proof 2 that the amortized cost of the increment is 2.

Example 2: Implementing a FIFO queue with two stacks Say you have a stack data type, and you need to implement a FIFO queue. The stack has the usual POP and PUSH operations, and the cost of each operation is 1.

We can implement a FIFO queue using two stacks as follows. The FIFO has two operations: ENQUEUE and DEQUEUE: *ENQUEUE(x)*: Push x onto stack1.

DEQUEUE(): If stack2 is empty, then we POP the entire contents of stack1 and PUSH it into stack2. Now simply POP from stack2 and return the result.

It's easy to see that this algorithm is correct. Here we'll

just worry about the running time. (I'm going to ignore the cost of checking of stack2 is empty, and only measure the cost in terms of the number of PUSHs and POPs that are done.)

Claim: the amortized cost of ENQUEUE is 3 and DEQUEUE is 1.

Proof 1 (aggregate method): As an element flows through the two-stack data structure, it's PUSHed at most twice and POPPed at most twice. This shows that we can assign an amortized cost of 4 to ENQUEUE and 0 to DEQUEUE.

To get the desired result, note that if an element is not

DEQUEUED it's only PUSHED twice and POPPED once. So the cost of 3 is paid for by the cost of 3 per ENQUEUE. The last POP is paid for by the DEQUEUE (if it happens).

Proof 2 (banker's method): Maintain a collection of tokens on stack1. In fact, keep 2 tokens for each thing in the stack.

When we ENQUEUE, we have three tokens to work with. We use one to push the element onto stack1, and the other two are put on the element for future use. When we have to move stack1 into stack2, we have enough tokens in the stack to pay for the move (one POP and one PUSH for each element).

Finally, the last pop done by the DEQUEUE is paid for by the 1 we allocated for it. QED.

Example 3: doubling array

We'll use an array to implement a stack that allows push and pop operations. We represent the stack as an array, A[] and an integer variable top that points to the top of the stack.

Here is a naive implementation of push and pop:

push(x): top++; A[top] = x;

pop; top—; return A[top+1]

These operations are both constant time (cost=1). The problem is, what happens when the array gets full? To deal with this, we keep another variable L, storing the size of the array, and a variable k keeping count of the number of elements of the array that are in use.

When we are about to overflow the available space (k=L), we allocate an array twice as large, move the data over to the new array, free the old array, and double L. This operation is called "doubling". It will be convenient to Analyse it as thought it was a separate operation on the data strucure, that occurs only when k=L.

If the stack is full, and has size L, and we apply the doubling operation, the cost of the operation is L, because we have to move L items into the new array. (The cost of allocating and freeing the arrays is O(L).)

Starting from an empty stack, doing any sequence n pushes and pops, how much does this cost?

First analysis: Total cost for pushes and pops is n.

Total cost for doubling is at worst $1 + 2 + 4 +\dots n/2 + n < 2n$ So, the total cost is 3n. Therefore we can say that the amortized cost of an operation is 3. *Second analysis*: Again, we use the financial approach. We'll keep money lying around the data structure in the following way.

```
                $ $ $ $
                $ $ $ $
           +—————————————————+
 |x|x|x|x|x|x|x|x|x|x|x|x| | | | |
             +—————————————————+
                    ^
                    |
               midpoint
```

We'll keep $2 on each element stored in the stack that is Beyond The Midpoint of the current array. In the normal case when we do a push (the array is not full), we need $1 to do the work, and then at most $2 to put onto the the new item we just pushed. Thus the cost is $3. (pop is even cheaper.) Now what happens if we have to do a doubling operation?

Before doubling:

```
                   $ $ $ $ $ $ $ $
                   $ $ $ $ $ $ $ $
                     +—————————————————+
|x|x|x|x|x|x|x|x|x|x|x|x|x|x|x|x|
                   —————————————————+
                          ^
                          |
                      midpoint
                   <————————L————————>
After doubling:
          +—————————————————+—————————————
——+   |x|x|x|x|x|x|x|x|x|x|x|x|x|x|x|x| | | | | | | | | | | | | | | | |
          +—————————————————+—————————————
——+
                             ^
                             |
                         midpoint
```

You can see that the money that we had on the items (L) was sufficient to pay for all the work of doubling the array (L). We conclude that if we allocate $3 for each operation, we'll never run out of money. Thus the amortized cost of an operation is 3.

In these examples, the financial model was not really necessary to get the desired results. However, later we'll see examples where this approach is necessary.

THE PHYSICIST'S METHOD

Let's say that instead of distributing our money all over the data structure (as we did above), we keep it all in a piggy bank. What's really important is how much money is in the piggy bank. We'll call the amount of money in this bank the "potential function" (Phi) for the problem.

So for the two problems above, we used the following two potential functions:

Phi(counter) = # of one bits in the counter
Phi(queue) = 2 * the size of stack 1.
[2(k-L/2) if k Þ L/2
Phi(stack) = [
[0 otherwise

(Here L is the current array size, and k is the number of elements currently in the stack.) Using this formalism we can define the amortized cost of an operation. Say the system changes from state S to state S' as a result of doing some operation.

We define the amortized cost of the operation as follows:

amortized cost = actual cost + Delta(Phi) =
= actual cost + Phi(S′) - Phi(S)

This is simply the amount of additional money that we need to maintain our piggy bank and to pay for the work.

For the counter, with the potential function given above:
amortized cost of increment $\Leftarrow 2$

For the stack, with the potential function given above:
amortized cost of pop $\Leftarrow 3$

How is this amortized cost related to actual cost? Let's sum the above definition of amortized cost over all the operations:

Sigma(amortized cost) = Sigma(actual cost) + Phi(final) - Phi(initial) or Sigma(actual cost) = Sigma(amortized cost) + Phi(initial) - Phi(final) If the potential is always non-negative, and starts at zero (as it is in our examples), then Sigma(actual cost) $\Leftarrow$ Sigma(amortized cost) In this more general framework, the potential can be negative, and may not start at 0. So in general we have to worry about the initial and final potentials.

Summary of using potential functions to do amortized analysis:

- Pick a potential function that's going to work (this is art)
- Using your potential function, bound the amortized cost of the operations you're interested in.
- Bound Phi(initial) - Phi(final) In terms of (1), one obvious point is that if the actual cost of an operation is HIGH, and you want the amortized cost to be LOW, then in this case the potential must DECREASE by a lot to pay for it. This is illustrated in both of the examples in this lecture.

We'll see in the next lecture just how essential this formalism is for analyzing splay trees.

DYNAMIC ARRAY

When an array becomes full, just copy the array elements to a larger array

Array$\rightarrow$ pointer to an array

Number → number of items in the array

Size → size of the array

```
AddToTable(x)
   if Number == Size then
       allocate NewArray with size 2*Size
       insert all items from Table to NewArray
       free Array
       Array= NewArray
     Size = 2 * Size
 end if
 insert x into Array
 Number = Number + 1
 end insert
```

In a sequence of operations on a data structure often the worst case can not occur in each operation.

The worst case cost of inserting an element in a dynamic array is N + 1.

N = size of the array before the insertion

But you can't get two worst cases in a row!

Amortized Analysis gives the average performance of each operation in the worst case

Three methods used in amortized analysis:

- Aggregate Method
- Accounting Method
- Potential Method

Example: Incrementing a k-bit binary counter

Let A[0..k-1] be an array of bits representing a number X

A[0] - low order bit, so

$$X = \sum_{J=0}^{K-1} A[J]2^{J}$$

length[A] = k

Start with X = 0

```
Increment (A)
```

```
J = 0
while J < length[A] and A[J] == 1 do
      A[J] = 0
      J = J + 1
   end while
   if J < length[A] then
         A[J] = 1
        end if
     end Increment
```

Count bits flipped.

Worst Case.

Increment flips k bits in worst case.

Sequence of n Increment operations takes O(nk).

AGGREGATE METHOD

T(n) = all work done in worst case in sequence of n operations Amortized cost per operation is T(n)/n

X	**A[4]**	**A[3]**	**A[2]**	**A[1]**	**A[0]**	**Total Cost**
0	0	0	0	0	*0*	0
1	0	0	0	*0*	1	1
2	0	0	0	1	*0*	3
3	0	0	*0*	1	1	4
4	0	0	1	0	*0*	7
5	0	0	1	*0*	1	8
6	0	0	1	1	*0*	10
7	0	*0*	1	1	1	11
8	0	1	0	0	*0*	15
9	0	1	0	*0*	1	16

A[0] flips each time Increment is called n

A[1] flips every other time:

$$\left\lfloor \frac{n}{2} \right\rfloor$$

A[1] flips every fourth time:

$$\left\lfloor \frac{n}{2^2} \right\rfloor$$

A[J] flips every 2^J time:

$$\left\lfloor \frac{n}{2^J} \right\rfloor$$

Total number of flips is:

$$\sum_{J=0}^{\lfloor \lg n \rfloor} \left\lfloor \frac{n}{2^J} \right\rfloor < n \sum_{J=0}^{\infty} \frac{1}{2^J} = 2n$$

So amortized cost of each operation is 2 = O(1)

ACCOUNTING METHOD

Assign an amortized cost to each operation Amortized cost may be more or less than the actual cost. If amortized cost is more than the actual cost of the operation assign the difference to part of the data structure as a credit, No negative credit allowed. Total amortized cost is ⇒ total worst case cost.

Example - binary Counter

Amortized cost of setting bit to 1 2 units

1 unit to pay for setting bit to 1

1 unit stored with bit

Amortized cost of setting bit to 0 0 units

Only a 1-bit is set to 0,

All 1-bits have credit of one unit

This pays for setting bit to 0

Example Continued

```
Increment (A)
  J = 0
  while J < length[A] and A[J] == 1 do
        A[J] = 0                              Cost 0
        J = J + 1
        end while
      if J < length[A] then
           A[J] = 1                           Cost 2
```

```
    end if
  end Increment
```

X	A[4]	A[3]	A[2]	A[1]	A[0]	Amortized cost
0	0	0	0	0	0	0
1	0	0	0	0	1 (1)	2
2	0	0	0	1 (1)	0	4
3	0	0	0	1 (1)	1 (1)	6
4	0	0	1 (1)	0	0	8
5	0	0	1 (1)	0	1 (1)	10
6	0	0	1 (1)	1 (1)	0	12
7	0	0	1 (1)	1 (1)	1 (1)	14
8	0	1 (1)	0	0	0	16
9	0	1 (1)	0	0	1 (1)	18

POTENTIAL METHOD

Assign an amortized cost to each operation: If amortized cost is more than the actual cost of the operation assign the difference the entire data structure as potential energy ck= actual cost of operation k

$\hat{c}_k$ = amortized cost of operation k

Dk = the state of the data structure after applying k'th operation to Dk,

Φ (D_k) = potential associated with Dk

$$\hat{c}_k + \Phi(D_k) - \Phi(D_{k-1})$$

$$\sum_{k=1}^{n} \hat{C}_k = \sum_{k=1}^{n} \left(c_k + \Phi(D_k) - \Phi(D_{k-1})\right)$$

$$= \sum_{k=1}^{n} c_k + \Phi(Dn) - \Phi(D_0)$$

So if $\Phi(D_n) \Rightarrow 0$ then $\Phi(D_0) \Rightarrow 0$ is an upper bound on total cost of the algorithm

Example - binary counter

Potential = number of 1's in the counter

if the k'th operation sets t_k bits to 0 the actual cost is t_k+1

$$\Phi(D_k) - \Phi(D_{k-1}) = 1 - t_k$$

so,

$$\hat{c}_k = c_k + \Phi(D_k) - \Phi(D_{k-1}) = 2$$

How to find $\Phi(D_k)$?

DYNAMIC TABLES

Table → pointer to a table

Number → number of items in the table

Size → size of the table

```
AddToTable(x)
if Number == Size then
allocate NewTable with size 2*Size
insert all items from Table to newTable
free Table
Table = NewTable
Size = 2 * Size
end if
insert x into Table
Number = Number + 1
end insert
```

INSERTS DONE BY N ADDTOTABLE

Amortized Cost per AddToTable 3 inserts Table after moving from size 4 to size 8 Perform 4 AddToTable operations

X	X	X	X								
X	X	X	X	Y(2)					X	X	X
X	Y(2)	Y(2)				X	X	X	X	Y(2)	Y(2)
Y(2)			X	X	X	X	Y(2)	Y(2)	Y(2)	Y(2)	

1/2 the table has 2 credits One credit per item to pay for the move to next size table

X	X	X	X	Y	Y	Y	Y

TABLE EXPANSION AND CONTRACTION

Policy

When table becomes full move to table twice the size When table contracts to 1/4 full, move to table 1/2 size Let load = (Size of table)/ (number of items in the table) Amortized Cost:

Inserting when load ⇒ 1/2

3 units

Inserting when load < 1/2

0 units

Deleting when load > 1/2

0 units

Deleting when load ⇐ 1/2

2 units

X	X	X	X					X	X	X
X	Y(2)					X	X	X	X	
				X	X	X	(1)			
X	X	(1)	(1)							
X	X									

LINKED LIST VS. DYNAMIC ARRAY LIST

Table. Amortized Costs Per Operation over n Operations

	linked List	**Dynamic Array]**
Insertion	1 call to new	lg(n)/n call to new
set two links	3 data moves	
Deletion	two links	3 data moves
after find		
	1 delete	lg(n)/n call to delete

GREEDY ALGORITHM

INTRODUCTION

Greedy algorithms are simple and straightforward. They are shortsighted in their approach in the sense that they take decisions on the basis of information at hand without worrying about the effect these decisions may have in the future. They are easy to invent, easy to implement and most of the time quite efficient. Many problems cannot be solved correctly by greedy approach. Greedy algorithms are used to solve optimization problems

GREEDY APPROACH

Greedy Algorithm works by making the decision that seems most promising at any moment; it never reconsiders this decision, whatever situation may arise later.

As an example consider the problem of "Making Change".

Coins available are:

- Dollars (100 cents)
- Quarters (25 cents)
- Dimes (10 cents)
- Nickels (5 cents)
- Pennies (1 cent)

Problem Make a change of a given amount using the smallest possible number of coins.

Informal Algorithm

- Start with nothing.
- At every stage without passing the given amount.
  - Add the largest to the coins already chosen.

Formal Algorithm

Make change for n units using the least possible number of coins.

```
  MAKE-CHANGE                                          (n)
C ¬ {100, 25, 10, 5, 1}// constant.
```

```
Sol ¬ {};// set that will hold the solution    set.
Sum ¬ 0 sum of item in solution set
WHILE sum not = n
x = largest item in set C such that sum + x £    n
IF no such item THEN
  RETURN "No Solution"
S ¬ S {value of x}
sum ¬ sum + x
RETURN S
```

Example Make a change for 2.89 (289 cents) here n = 2.89 and the solution contains 2 dollars, 3 quarters, 1 dime and 4 pennies. The algorithm is greedy because at every stage it chooses the largest coin without worrying about the consequences. Moreover, it never changes its mind in the sense that once a coin has been included in the solution set, it remains there.

CHARACTERISTICS AND FEATURES

To construct the solution in an optimal way. Algorithm maintains two sets. One contains chosen items and the other contains rejected items.

The greedy algorithm consists of four (4) function:

- A function that checks whether chosen set of items provide a solution.
- A function that checks the feasibility of a set.
- The selection function tells which of the candidates is the most promising.
- An objective function, which does not appear explicitly, gives the value of a solution.

STRUCTURE GREEDY ALGORITHM

- Initially the set of chosen items is empty i.e., solution set.

- At each step
  - item will be added in a solution set by using selection function.
  - IF the set would no longer be feasible
    - a) Reject items under consideration (and is never consider again).
  - ELSE IF set is still feasible THEN
    - a) Add the current item.

DEFINITIONS OF FEASIBILITY

A feasible set (of candidates) is promising if it can be extended to produce not merely a solution, but an optimal solution to the problem. In particular, the empty set is always promising why? (because an optimal solution always exists)

Unlike Dynamic Programmeming, which solves the subproblems bottom-up, a greedy strategy usually progresses in a top-down fashion, making one greedy choice after another, reducing each problem to a smaller one.

Greedy-Choice Property

The "greedy-choice property" and "optimal substructure" are two ingredients in the problem that lend to a greedy strategy.

Greedy-Choice Property

It says that a globally optimal solution can be arrived at by making a locally optimal choice.

SAMPLE PROBLEM

There is a long list of stalls, some of which need to be covered with boards. You can use up to N ($1 \Leftarrow N \Leftarrow 50$)

boards, each of which may cover any number of consecutive stalls. Cover all the necessary stalls, while covering as few total stalls as possible.

THE IDEA

The basic idea behind greedy algorithms is to build large solutions up from smaller ones. Unlike other approaches, however, greedy algorithms keep only the best solution they find as they go along.

Thus, for the sample problem, to build the answer for N = 5, they find the best solution for N = 4, and then alter it to get a solution for N = 5. No other solution for N = 4 is ever considered.

Greedy algorithms are fast, generally linear to quadratic and require little extra memory. Unfortunately, they usually aren't correct. But when they do work, they are often easy to implement and fast enough to execute.

PROBLEMS

There are two basic problems to greedy algorithms.

How to Build

How does one create larger solutions from smaller ones? In general, this is a function of the problem. For the sample problem, the most obvious way to go from four boards to five boards is to pick a board and remove a section, thus creating two boards from one. You should choose to remove the largest section from any board which covers only stalls which don't need covering (so as to minimize the total number of stalls covered).

To remove a section of covered stalls, take the board which spans those stalls, and make into two boards: one of which covers the stalls before the section, one of which covers the stalls after the second.

WORK

The real challenge for the Programmemer lies in the fact that greedy solutions don't always work. Even if they seem to work for the sample input, random input, and all the cases you can think of, if there's a case where it won't work, at least one (if not more!) of the judges' test cases will be of that form. For the sample problem, to see that the greedy algorithm described above works, consider the following:

Assume that the answer doesn't contain the large gap which the algorithm removed, but does contain a gap which is smaller. By combining the two boards at the end of the smaller gap and splitting the board across the larger gap, an answer is obtained which uses as many boards as the original solution but which covers fewer stalls. This new answer is better, so therefore the assumption is wrong and we should always choose to remove the largest gap.

If the answer doesn't contain this particular gap but does contain another gap which is just as large, doing the same transformation yields an answer which uses as many boards and covers as many stalls as the other answer. This new answer is just as good as the original solution but no better, so we may choose either.

Thus, there exists an optimal answer which contains the large gap, so at each step, there is always an optimal answer which is a superset of the current state. Thus, the final answer is optimal.

CONCLUSIONS

If a greedy solution exists, use it. They are easy to code, easy to debug, run quickly, and use little memory, basically defining a good algorithm in contest terms. The only missing element from that list is correctness. If the greedy algorithm finds the correct answer, go for it, but don't get suckered into thinking the greedy solution will work for all problems.

Algorithm

The sequence has three parts: the part which will be 1 when in sorted order, 2 when in sorted order, and 3 when in sorted order. The greedy algorithm swaps as many as possible of the 1's in the 2 part with 2's in the 1 part, as many as possible 1's in the 3 part with 3's in the 1 part, and 2's in the 3 part with 3's in the 2 part.

Once none of these types remains, the remaining elements out of place need to be rotated one way or the other in sets of 3. You can optimally sort these by swapping all the 1's into place and then all the 2's into place.

Analysis: Obviously, a swap can put at most two elements in place, so all the swaps of the first type are optimal. Also, it is clear that they use different types of elements, so there is no "interference" between those types. This means the order does not matter.

Once those swaps have been performed, the best you can do is two swaps for every three elements not in the correct location, which is what the second part will achieve (for example, all the 1's are put in place but no others; then all that remains are 2's in the 3's place and vice-versa, and which can be swapped).

Friendly Coins - A Counterexample [Abridged]

Given the denominations of coins for a newly founded country, the Dairy Republic, and some monetary amount, find the smallest set of coins that sums to that amount. The Dairy Republic is guaranteed to have a 1 cent coin.

Algorithm: Take the largest coin value that isn't more than the goal and iterate on the total minus this value.

(Faulty) Analysis: Obviously, you'd never want to take a smaller coin value, as that would mean you'd have to take more coins to make up the difference, so this algorithm works.

Maybe not: Okay, the algorithm usually works. In fact, for the U.S. coin system {1, 5, 10, 25}, it always yields the optimal set. However, for other sets, like {1, 5, 8, 10} and a goal of 13, this greedy algorithm would take one 10, and then three 1's, for a total of four coins, when the two coin solution {5, 8} also exists.

Topological Sort

Given a collection of objects, along with some ordering constraints, such as "A must be before B," find an order of the objects such that all the ordering constraints hold.

Algorithm: Create a directed graph over the objects, where there is an arc from A to B if "A must be before B." Make a pass through the objects in arbitrary order. Each time you find an object with in-degree of 0, greedily place it on the end of the current ordering, delete all of its out-arcs, and recurse on its (former) children, performing the same check. If this algorithm gets through all the objects without putting every object in the ordering, there is no ordering which satisfies the constraints.

2

Analysis of Algorithms

When analyzing a program in terms of efficiency, we want to look at questions such as, “How long does it take for the program to run?” and “Is there another approach that will get the answer more quickly?” Efficiency will always be less important than correctness; if you don’t care whether a program works correctly, you can make it run very quickly indeed, but no one will think it’s much of an achievement! On the other hand, a program that gives a correct answer after ten thousand years isn’t very useful either, so efficiency is often an important issue.

The term “efficiency” can refer to efficient use of almost any resource, including time, computer memory, disk space, or network bandwidth. In this section, however, we will deal exclusively with time efficiency, and the major question that we want to ask about a program is, how long does it take to perform its task?

It really makes little sense to classify an individual program as being "efficient" or "inefficient." It makes more sense to compare two (correct) programs that perform the same task and ask which one of the two is "more efficient," that is, which one performs the task more quickly. However, even here there are difficulties. The running time of a program is not well-defined.

The run time can be different depending on the number and speed of the processors in the computer on which it is run and, in the case of Java, on the design of the Java Virtual Machine which is used to interpret the program. It can depend on details of the compiler which is used to translate the program from high-level language to machine language. Furthermore, the run time of a program depends on the size of the problem which the program has to solve. It takes a sorting program longer to sort 10000 items than it takes it to sort 100 items. When the run times of two programs are compared, it often happens that Program A solves small problems faster than Program B, while Program B solves large problems faster than Program A, so that it is simply not the case that one program is faster than the other in all cases.

In spite of these difficulties, there is a field of computer science dedicated to analyzing the efficiency of programs. The field is known as analysis of Algorithms. The focus is on algorithms, rather than on programs as such, to avoid having to deal with multiple implementations of the same algorithm written in different languages, compiled with different compilers, and running on different computers. Analysis of Algorithms is a mathematical field that abstracts away from these down-and-dirty details. Still, even though it is a theoretical field, every working programmer should be aware of some of its techniques and results.

One of the main techniques of analysis of algorithms is asymptotic analysis. The term "asymptotic" here means basically "the tendency in the long run." An asymptotic analysis of an algorithm's run time looks at the question of how the run time depends on the size of the problem. The analysis is asymptotic because it only considers what happens to the run time as the size of the problem increases without limit; it is not concerned with what happens for problems of small size or, in fact, for problems of any fixed finite size. Only what happens in the long run, as the problem size increases without limit, is important. Showing that Algorithm A is asymptotically faster than Algorithm B doesn't necessarily mean that Algorithm A will run faster than Algorithm B for problems of size 10 or size 1000 or even size 1000000 — it only means that if you keep increasing the problem size, you will eventually come to a point where Algorithm A is faster than Algorithm B. An asymptotic analysis is only a first approximation, but in practice it often gives important and useful information.

Using this notation, we might say, for example, that an algorithm has a running time that is $O(n^2)$ or $O(n)$ or $O(\log(n))$. These notations are read "Big-Oh of n squared," "Big-Oh of n," and "Big-Oh of log n" (where log is a logarithm function). More generally, we can refer to $O(f(n))$ ("Big-Oh of f of n"), where $f(n)$ is some function that assigns a positive real number to every positive integer n. The "n" in this notation refers to the size of the problem. Before you can even begin an asymptotic analysis, you need some way to measure problem size. Usually, this is not a big issue. For example, if the problem is to sort a list of items, then the problem size can be taken to be the number of items in the list. When the input to an algorithm is an integer, as in the case of an algorithm that checks whether a given positive integer is prime, the usual measure of the size of a problem is the number of bits in the input integer rather than the integer itself. More generally, the number of bits in the input to a problem is often a good measure of the size of the problem.

To say that the running time of an algorithm is O(f(n)) means that for large values of the problem size, n, the running time of the algorithm is no bigger than some constant times f(n). (More rigorously, there is a number C and a positive integer M such that whenever n is greater than M, the run time is less than or equal to C*f(n).) The constant takes into account details such as the speed of the computer on which the algorithm is run; if you use a slower computer, you might have to use a bigger constant in the formula, but changing the constant won't change the basic fact that the run time is O(f(n)).

The constant also makes it unnecessary to say whether we are measuring time in seconds, years, CPU cycles, or any other unit of measure; a change from one unit of measure to another is just multiplication by a constant. Note also that O(f(n)) doesn't depend at all on what happens for small problem sizes, only on what happens in the long run as the problem size increases without limit.

To look at a simple example, consider the problem of adding up all the numbers in an array. The problem size, n, is the length of the array. Using A as the name of the array, the algorithm can be expressed in Java as:

```
total = 0;
for (int i = 0; i < n; i++)
     total = total + A[i];
```

This algorithm performs the same operation, total = total + A[i], n times. The total time spent on this operation is a*n, where a is the time it takes to perform the operation once. Now, this is not the only thing that is done in the algorithm. The value of i is incremented and is compared to n each time through the loop. This adds an additional time of b*n to the run time, for some constant b. Furthermore, i and total both have to be initialized to zero; this adds some constant amount c to the running time.

The exact running time would then be (a+b)*n+c, where the constants a, b, and c depend on factors such as how the code is compiled and what computer it is run on. Using the fact that c is less than or equal to c*n for any positive integer n, we can say that the run time is less than or equal to (a+b+c)*n. That is, the run time is less than or equal to a constant times n. By definition, this means that the run time for this algorithm is O(n).

If this explanation is too mathematical for you, we can just note that for large values of n, the c in the formula (a+b)*n+c is insignificant compared to the other term, (a+b)*n. We say that c is a "lower order term." When doing asymptotic analysis, lower order terms can be discarded. A rough, but correct, asymptotic analysis of the algorithm would go something like this: Each iteration of the for loop takes a certain constant amount of time. There are n iterations of the loop, so the total run time is a constant times n, plus lower order terms (to account for the initialization). Disregarding lower order terms, we see that the run time is O(n).

Note that to say that an algorithm has run time O(f(n)) is to say that its run time is no bigger than some constant times f(n) (for large values of n). O(f(n)) puts an upper limit on the run time. However, the run time could be smaller, even much smaller. For example, if the run time is O(n), it would also be correct to say that the run time is O(n^2) or even O(n^{10}). If the run time is less than a constant times n, then it is certainly less than the same constant times n^2 or n^{10}.

Of course, sometimes it's useful to have a lower limit on the run time. That is, we want to be able to say that the run time is greater than or equal to some constant times f(n) (for large values of n). The notation for this is Ω(f(n)), read "Omega of f of n." "Omega" is the name of a letter in the Greek alphabet, and Ù is the upper case version of that letter. (To be technical, saying that the run time of an algorithm is Ω(f(n)) means that there is a positive number C and a positive integer M such that whenever n is greater than M, the run time is greater than or equal to C*f(n).) O(f(n)) tells you something about the maximum amount of time that you might have to wait for an algorithm to finish; Ω(f(n)) tells you something about the minimum time.

The algorithm for adding up the numbers in an array has a run time that is Ω(n) as well as O(n). When an algorithm has a run time that is both Ω(f(n)) and O(f(n)), its run time is said to be Θ(f(n)), read "Theta of f of n." (Theta is another letter from the Greek alphabet.) To say that the run time of an algorithm is Θ(f(n)) means that for large values of n, the run time is between a*f(n) and b*f(n), where a and b are constants (with b greater than a, and both greater than 0).

Let's look at another example. Consider the algorithm that can be expressed in Java in the following method:

```
/**
 * Sorts the n array elements A[0], A[1], ..., A[n-1] into
increasing order.
 */
public static simpleBubbleSort(int[] A, int n) {
   for (int i = 0; i < n; i++) {
         //Do n passes through the array...
      for (int j = 0; j < n-1; j++) {
         if (A[j] > A[j+1]) {
               //A[j] and A[j+1] are out of order, so swap
them
            int temp = A[j];
            A[j] = A[j+1];
            A[j+1] = temp;
         }
      }
   }
}
```

Here, the parameter n represents the problem size. The outer for loop in the method is executed n times. Each time the outer for loop is executed, the inner for loop is exectued n-1 times, so the if statement is executed n*(n-1) times. This is n^2-n, but since lower order terms are not significant in an asymptotic analysis, it's good enough to say that the if statement is executed about n^2 times. In particular, the test A[j] > A[j+1] is executed about n^2 times, and this fact by itself is enough to say that the run time of the algorithm is $\Omega(n^2)$, that is, the run time is at least some constant times n^2. Furthermore, if we look at other operations — the assignment statements, incrementing i and j, *etc.* — none of them are executed more than n^2 times, so the run time is also $O(n^2)$, that is, the run time is no more than some constant times n^2. Since it is both Ù(n^2) and $O(n^2)$, the run time of the simpleBubbleSort algorithm is $\Theta(n^2)$.

You should be aware that some people use the notation $O(f(n))$ as if it meant $\Theta(f(n))$. That is, when they say that the run time of an algorithm is $O(f(n))$, they mean to say that the run time is about equal to a constant times $f(n)$. For that, they should use $\Theta(f(n))$. Properly speaking, $O(f(n))$ means that the run time is less than a constant times $f(n)$, possibly much less.

So far, the analysis has ignored an imp ortant detail. We have looked at how run time depends on the problem size, but in fact the run time usually depends not just on the size of the problem but on the specific data that has to be processed. For example, the run time of a sorting algorithm can depend on the initial order of the items that are to be sorted, and not just on the number of items.

To account for this dependency, we can consider either the worst case run time analysis or the average case run time analysis of an algorithm. For a worst case run time analysis, we consider all possible problems of size n and look at the longest possible run time for all such problems. For an average case analysis, we consider all possible problems of size n and look at the average of the run times for all such problems. Usually, the average case analysis assumes that all problems of size n are equally likely to be encountered, although this is not always realistic — or even possible in the case where there is an infinite number of different problems of a given size.

In many cases, the average and the worst case run times are the same to within a constant multiple. This means that as far as asymptotic analysis is concerned, they are the same. That is, if the average case run time is $O(f(n))$ or $\Theta(f(n))$, then so is the worst case.

We will not do any rigorous mathematical analysis, but you should be able to follow informal discussion of simple cases such as the examples that we have looked at in this section. Most important, though, you should have a feeling for exactly what it means to say that the running time of an algorithm is $O(f(n))$ or $\Theta(f(n))$ for some common functions $f(n)$. The main point is that these notations do not tell you anything about the actual numerical value of the running time of the algorithm for any particular case. They do not tell you anything at all about the running time for small values of n. What they do tell you is something about the rate of growth of the running time as the size of the problem increases.

Suppose you compare two algorithms that solve the same problem. The run time of one algorithm is $\Theta(n^2)$, while the run time of the second algorithm is $\Theta(n^3)$. What does this tell you? If you want to know which algorithm will be faster for some particular problem of size, say, 100, nothing is certain. As far as you can tell just from the asymptotic analysis, either algorithm could be faster for that particular case — or in any particular case. But what you can say for sure is that if you look at larger and larger problems, you will come to a point where the $\Theta(n^2)$ algorithm is faster than the $\Theta(n^3)$ algorithm. Furthermore, as you continue to increase the problem size, the relative advantage of the $\Theta(n^2)$ algorithm will continue to grow. There will be values of n for which the $\Theta(n^2)$ algorithm is a thousand times faster, a million times faster, a billion times faster, and so on. This is because for any positive constants a and b, the function $a*n^3$ grows faster than the function $b*n^2$ as n gets larger. (Mathematically, the limit of the ratio of $a*n^3$ to $b*n^2$ is infinite as n approaches infinity.)

This means that for “large” problems, a $\Theta(n^2)$ algorithm will definitely be faster than a $\Theta(n^3)$ algorithm. You just don’t know — based on the asymptotic analysis alone — exactly how large “large” has to be.

In practice, in fact, it is likely that the $\Theta(n^2)$ algorithm will be faster even for fairly small values of n, and absent other information you would generally prefer a $\Theta(n^2)$ algorithm to a $\Theta(n^3)$ algorithm.

So, to understand and apply asymptotic analysis, it is essential to have some idea of the rates of growth of some common functions. For the power functions n, n^2, n^3, n^4, ..., the larger the exponent, the greater the rate of growth of the function. Exponential functions such as 2^n and 10^n, where the n is in the exponent, have a growth rate that is faster than that of any power function. In fact, exponential functions grow so quickly that an algorithm whose run time grows exponentially is almost certainly impractical even for relatively modest values of n, because the running time is just too long. Another function that often turns up in asymptotic analysis is the logarithm function, log(n). There are actually many different logarithm functions, but the one that is usually used in computer science is the so-called logarithm to the base two, which is defined by the fact that $\log(2^x) = x$ for any number x. The logarithm function grows very slowly. The growth rate of log(n) is much smaller than the growth rate of n. The growth rate of n*log(n) is a little larger than the growth rate of n, but much smaller than the growth rate of n^2.

POINTS IN ALGORITHMS

- Introduction: Analysis of Selection Sort
- Introduction: Analysis of Merge Sort
- Asymptotic Notation
- Asymptotic Notation Continued
- Heapsort
- Heapsort Continued
- Priority Queues (more heaps)
- Quicksort
- Bounds on Sorting and Linear Time Sorts
- Stable Sorts and Radix Sort
- Begin Dynamic Programming
- More Dynamic Programming
- Begin Greedy Algorithms: Huffman's Algorithm
- Dÿkstra's Algorithm
- Beyond Asymptotic Analysis: Memory Access Time
- B-Trees
- More B-Trees: Insertion and Splitting
- Union/Find
- Warshall's Algorithm, Floyd's Algorithm
- Large Integer Arithmetic
- RSA Public-Key Cryptosystem
- Begin Algorithms and Structural Complexity Theory
- Continue Algorithms and Structural Complexity Theory
- End Algorithms and Structural Complexity Theory

- Generating Permutations and Combinations
- Exam review with sample questions and solutions.

SORTING OF ALGORITHMS

It is not always possible to say that one algorithm is better than another, as relative performance can vary depending on the type of data being sorted. In some situations, most of the data are in the correct order, with only a few items needing to be sorted. In other situations the data are completely mixed up in a random order and in others the data will tend to be in reverse order.

Different algorithms will perform differently according to the data being sorted. Four common algorithms are the exchange or bubble sort, the selection sort, the insertion sort and the quick sort.

The selection sort is a good one to use with students. It is intuitive and very simple to program. It offers quite good performance, its particular strength being the small number of exchanges needed. For a given number of data items, the selection sort always goes through a set number of comparisons and exchanges, so its performance is predictable.

```
  procedure SelectionSort (d: DataArrayType; n: integer) {n
is the number of elements}
    for k = 1 to n-1 do
      begin
      small = k
      for j = k+1 to n do
      if d[ j ] < d[small] then small = j
      {Swap elements k and small}
    Swap(d, k, small)
  end
```

EXCHANGE (BUBBLE) SORT

Element	1	2	3	4	5	6	7	8
Data	27	63	1	72	64	58	14	9
1st pass	27	1	63	64	58	14	9	72
2nd pass	1	27	63	58	14	9	64	72
3rd pass	1	27	58	14	9	63	64	72...

The first two data items (27 and 63) are compared and the smaller one placed on the left hand side. The second and third items (63 and 1) are then compared and the smaller one placed on the left and so on. After all the data has been passed through once, the largest data item (72) will have "bubbled" through to the end of the list. At the end of the second pass, the second largest data item (64) will be in the second last position. For n data items, the process continues for n-1 passes, or until no exchanges are made in a single pass.

INSERTION SORT

Element	1	2	3	4	5	6	7	8
Data	27	63	1	72	64	58	14	9
1st pass	27	63	1	72	64	58	9	14
2nd pass	27	63	1	72	64	9	14	58
3rd pass	27	63	1	72	9	14	58	64...

The insertion sort starts with the last two elements and creates a correctly sorted sub-list, which in the example contains 9 and 14. It then looks at the next element (58) and inserts it into the sub-list in its correct position. It takes the next element (64) and does the same, continuing until the sub-list contains all the data.

SELECTION SORT

The selection sort marks the first element (27). It then goes through the remaining data to find the smallest number (1). It swaps this with the first element and the smallest element is now in its correct position. It then marks the second element (63) and looks through the remaining data for the next smallest number (9). These two numbers are then swapped. This process continues until n-1 passes have been made.

Element	1	2	3	4	5	6	7	8
Data	27	63	1	72	64	58	14	9
1st pass	1	63	27	72	64	58	14	9
2nd pass	1	9	27	72	64	58	14	63
3rd pass	1	9	14	72	64	58	27	63...

QUICK SORT

Element	1	2	3	4	5	6	7	8
Data	27	63	1	72	64	58	14	9
1st pass	1	9	63	72	64	58	14	27
2nd pass	1	9	14	27	64	58	72	63
3rd pass	1	9	14	27	58	63	72	64
4th pass	1	9	14	27	58	63	64	72 sorted!

The quick sort takes the last element (9) and places it such that all the numbers in the left sub-list are smaller and all the numbers in the right sub-list are bigger. It then quick sorts the left sub-list ({1}) and then quick sorts the right sub-list. This is a recursive algorithm, since it is defined in terms of itself. This reduces the complexity of programming it, however it is the least intuitive of the four algorithms.

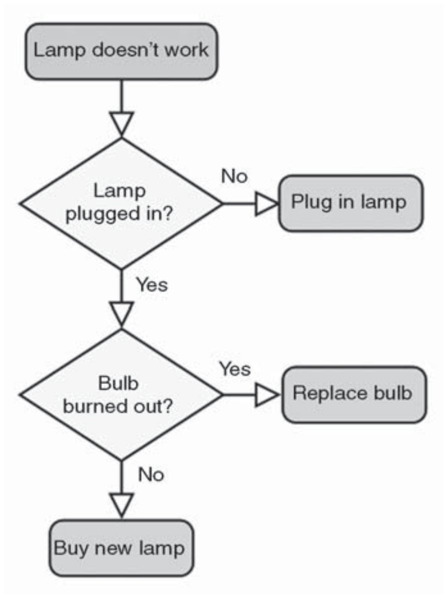
3

Algorithms Computing

In mathematics, computing, and related subjects, an algorithm is an effective method for solving a problem using a finite sequence of instructions. Algorithms are used for calculation, data processing, and many other fields. Each algorithm is a list of well-defined instructions for completing a task. Starting from an initial state, the instructions describe a computation that proceeds through a well-defined series of successive states, eventually terminating in a final ending state.

The transition from one state to the next is not necessarily deterministic; some algorithms, known as randomized algorithms, incorporate randomness.

While there is no generally accepted *formal* definition of "algorithm", an informal definition could be "a process that performs some sequence of operations." For some people, a program is only an algorithm if it stops eventually. For others, a program is only an algorithm if it stops before a given number of calculation steps.



A prototypical example of an algorithm is Euclid's algorithm to determine the maximum common divisor of two integers. We can derive clues to the issues involved and an informal meaning of the word from the following quotation from Boolos & Jeffrey:

No human being can write fast enough, or long enough, or small enough ("smaller and smaller without limit ...you'd be trying to write on molecules, on atoms, on electrons") to list all members of an enumerably infinite set by writing out their names, one after another, in some notation. But humans can do something equally useful, in the case of certain enumerably infinite sets: They can give explicit instructions for determining the *n*th member of the set, for arbitrary finite *n*. Such instructions are to be given quite explicitly, in a form in which they could be followed by a computing machine, or by a human who is capable of carrying out only very elementary operations on symbols.

The term "enumerably infinite" means "countable using integers perhaps extending to infinity." Thus Boolos and Jeffrey are saying that an algorithm *implies* instructions for a process that "creates" output integers from an *arbitrary* "input" integer or integers that, in theory, can be chosen from 0 to infinity.

Thus we might expect an algorithm to be an algebraic equation such as y = m + n — two arbitrary "input variables" mand n that produce an output y. As we see in Algorithm characterizations — the word algorithm implies much more than this, something on the order of (for our addition example):

Precise instructions (in language understood by "the computer") for a "fast, efficient, good" *process* that specifies the "moves" of "the computer" (machine or human, equipped with the necessary internally-contained information and capabilities) to find, decode, and then munch arbitrary input integers/symbols m and n, symbols + and = ... and (reliably, correctly, "effectively") produce, in a "reasonable" time, output-integer y at a specified place and in a specified format.

The concept of *algorithm* is also used to define the notion of decidability. That notion is central for explaining how formal systems come into being starting from a small set of axioms and rules. In logic, the time that an algorithm requires to complete cannot be measured, as it is not apparently related with our customary physical dimension. From such uncertainties, that characterize ongoing work, stems the unavailability of a definition of *algorithm* that suits both concrete (in some sense) and abstract usage of the term.

FORMALIZATION

Algorithms are essential to the way computers process information. Many computer programs contain algorithms that specify the specific instructions a computer should perform (in a specific order) to carry out a specified task, such as calculating employees' paychecks or printing students' report cards. Thus, an algorithm can be considered to be any sequence of operations that can be simulated by a Turing-complete system.

Typically, when an algorithm is associated with processing information, data is read from an input source, written to an output device, and/or stored for further processing. Stored data is regarded as part of the internal state of the entity performing the algorithm. In practice, the state is stored in one or more data structures. For any such computational process, the algorithm must be rigorously defined: specified in the way it applies in all possible circumstances that could arise. That is, any conditional steps must be systematically dealt with, case-by-case; the criteria for each case must be clear (and computable). Because an algorithm is a precise list of precise steps, the order of computation will always be critical to the functioning of the algorithm.

Instructions are usually assumed to be listed explicitly, and are described as starting "from the top" and going "down to the bottom", an idea that is described more formally by *flow of control*. So far, this discussion of the formalization of an algorithm has assumed the premises of imperative programming. This is the most common conception, and it attempts to describe a task in discrete, "mechanical" means. Unique to this conception of formalized algorithms is the assignment, setting the value of a variable. It derives from the intuition of “memory” as a scratchpad. There is an example below of such an assignment.

TERMINATION

Some writers restrict the definition of *algorithm* to procedures that eventually finish. In such a category Kleene places the “*decision procedure* or *decision method* or *algorithm* for the question”. Others, including Kleene, include procedures that could run forever without stopping; such a procedure has been called a “computational method” or “*calculation procedure* or *algorithm* (and hence a *calculation problem*) in relation to a general question which requires for an answer, not yes or no, but the exhibiting of some object”.

Minsky makes the pertinent observation, in regards to determining whether an algorithm will eventually terminate (from a particular starting state): But if the length of the process isn’t known in advance, then “trying” it may not be decisive, because if the process does go on forever — then at no time will we ever be sure of the answer.

As it happens, no other method can do any better, as was shown by Alan Turing with his celebrated result on the undesirability of the so-called halting. There is no algorithmic procedure for determining of arbitrary algorithms whether or not they terminate from given starting states. The analysis of algorithms for their likelihood of termination is called termination analysis.

See the examples of (im-) “proper” subtraction at partial function for more about what can happen when an algorithm fails for certain of its input numbers — *e.g.*, (i) nontermination, (ii) production of "junk" (output in the wrong format to be considered a number) or no number(s) at all (halt ends the computation with no output), (iii) wrong number(s), or (iv) a combination of these. Kleene proposed that the production of "junk" or failure to produce a number is solved by having the algorithm detect these instances and produce *e.g.,* an error message (he suggested "0"), or preferably, force the algorithm into an endless loop. Davis does this to his subtraction algorithm — he fixes his algorithm in a second example so that it is proper subtraction and it terminates Along with the logical outcomes "true" and "false" Kleene also proposes the use of a third logical symbol "u" — undecided — thus an algorithm will always produce *something* when confronted with a "proposition". The problem of wrong answers must be solved with an independent "proof" of the algorithm *e.g.,* using induction:

EXPRESSING ALGORITHMS

Algorithms can be expressed in many kinds of notation, including natural languages, pseudo code, flowcharts, programming languages or control tables (processed by interpreters). Natural language expressions of algorithms tend to be verbose and ambiguous, and are rarely used for complex or technical algorithms. Pseudocode, flowcharts and control tables are structured ways to express algorithms that avoid many of the ambiguities common in natural language statements, while remaining independent of a particular implementation language. Programming languages are primarily intended for expressing algorithms in a form that can be executed by a computer, but are often used as a way to define or document algorithms.

There is a wide variety of representations possible and one can express a given Turing machine program as a sequence of machine tables, as flowcharts, or as a form of rudimentary machine code or assembly code called "sets of quadruples"..

Sometimes it is helpful in the description of an algorithm to supplement small "flow charts" with natural-language and/ or arithmetic expressions written inside "block diagrams" to summarize what the "flow charts" are accomplishing.

Representations of algorithms are generally classed into three accepted levels of Turing machine description:

1. High-level description: "...prose to describe an algorithm, ignoring the implementation details. At this level we do not need to mention how the machine manages its tape or head."
2. Implementation description: "...prose used to define the way the Turing machine uses its head and the way that it stores data on its tape. At this level we do not give details of states or transition function."
3. Formal description: Most detailed, "lowest level", gives the Turing machine's "state table".

COMPUTER ALGORITHMS

In computer systems, an algorithm is basically an instance of logic written in software by software developers to be effective for the intended "target" computer(s), in order for the software on the target machines to *do something*. For instance, if a person is writing software that is supposed to print out a PDF document located at the operating system folder "/My Documents" at computer drive "D:" every Friday, they will write an algorithm that specifies the following actions: "If today's date (computer time) is 'Friday,' open the document at ‘D:/My Documents’ and call the ‘print’ function”. While this simple algorithm does not look into whether the printer has enough paper or whether the document has been moved into a different location, one can make this algorithm more robust and anticipate these problems by rewriting it as a formal CASE statement or as a (carefully crafted) sequence of IF-THEN-ELSE statements. For example the CASE statement might appear as follows (there are other possibilities):

CASE 1: IF today’s date is NOT Friday THEN *exit this CASE instruction* ELSE

CASE 2: IF today’s date is Friday AND the document is located at ‘D:/My Documents’ AND there is paper in the printer THEN print the document (and *exit this CASE instruction*) ELSE

CASE 3: IF today’s date is Friday AND the document is NOT located at ‘D:/My Documents’ THEN display ‘document not found’ error message (and *exit this CASE instruction*) ELSE

CASE 4: IF today’s date is Friday AND the document is located at ‘D:/My Documents’ AND there is NO paper in the printer THEN (i) display ‘out of paper’ error message and (ii) *exit*.

Note that CASE 3 includes two possibilities: (i) the document is NOT located at ‘D:/My Documents’ AND there’s paper in the printer OR (ii) the document is NOT located at ‘D:/My Documents’ AND there’s NO paper in the printer.

The sequence of IF-THEN-ELSE tests might look like this:

TEST 1: IF today’s date is NOT Friday THEN *done* ELSE TEST 2:

TEST 2: IF the document is NOT located at 'D:/My Documents' THEN display 'document not found' error message ELSE TEST 3:

TEST 3: IF there is NO paper in the printer THEN display 'out of paper' error message ELSE print the document.

These examples' logic grants precedence to the instance of "NO document at 'D:/My Documents' ".

Also observe that in a well-crafted CASE statement or sequence of IF-THEN-ELSE statements the number of distinct actions—4 in these examples: do nothing, print the document, display 'document not found', display 'out of paper' — equals the number of cases.

Given unlimited memory, a computational machine with the ability to execute either a set of CASE statements or a sequence of IF-THEN-ELSE statements is Turing complete. Therefore, anything that is computable can be computed by this machine. This form of algorithm is fundamental to computer programming in all its forms.

MASTERS THEOREM

INTRODUCTION

In the analysis of algorithms, the master theorem, which is a specific case of the Akra-Bazzi theorem, provides a cookbook solution in asymptotic terms for recurrence relations of types that occur in practice. It was popularized by the canonical algorithms which introduces and proves Nevertheless, not all recurrence relations can be solved with the use of the master theorem.

Consider a problem that can be solved using recurrence algorithm such as below:

```
procedure T(n: size of problem) defined as:
  if n < k then exit
  Do work of amount f(n)
  T(n/b)
  T(n/b)
  ...repeat for a total of a times...
  T(n/b)
end procedure
```

In above algorithm we are dividing the problem in to number of sub problems recursively, each sub problem being of size n/b. This can be visualized as building a call tree with each node of a tree an instance of one recursive call and its child nodes being instance of next calls. In above example, each node would have a number of child nodes. Each node does amount of work that depends on size of sub problem n passed to that instance of recursive call and given by $f(n)$. For example, if each recursive call is doing sorting then size of work does by each node in the tree would be at least $O(nlog(n))$. Total size of work done by entire tree is sum of work performed by all the nodes in the tree.

Algorithm such as above can be represented as recurrence relationship;

$$T(n) = aT\left(\frac{n}{b}\right) + f(n).$$

This recursive relationship can be successively substituted in to itself and expanded to obtain expression for total amount of work done[

Original Master theorem allows to easily calculate run time of such a recursive algorithm in Big O notation without doing expansion of above recursive relationship. A generalized form of Master Theorem by Akra and Bazzi introduced in 1998 is more usable, simpler and applicable on wide number of cases that occurs in practice.

The master theorem concerns recurrence relations of the form:

$$T(n) = aT\ \frac{n}{b}\ + f(n) \text{ Where a} \geq 1\text{, b>1.}$$

In the application to the analysis of a recursive algorithm, the constants and function take on the following significance:

- *n* is the size of the problem.
- *a* is the number of subproblems in the recursion.
- *n/b* is the size of each subproblem. (Here it is assumed that all subproblems are essentially the same size.)
- *f* (*n*) is the cost of the work done outside the recursive calls, which includes the cost of dividing the problem and the cost of merging the solutions to the subproblems.

GENERIC FORM

If it is true that $f(n) = O\left(n^{\log_b(a)-\varepsilon}\right)$ for some constant ε > 0 (using Big O notation) it follows that:

$$T(n) = \Theta\left(n^{\log_b a}\right).$$

EXAMPLE

$$T(n) = 8T\ \frac{n}{2}\ + 1000n^2$$

As one can see in the formula above, the variables get the following values:

$$a = 8, b = 2, f(n) = 1000n^2, \log_b a = \log_2 8 = 3$$

Now we have to check that the following equation holds:

$$f(n) = O\left(n^{\log_b a - \varepsilon}\right)$$

$$1000n^2 = O\left(n^{3-\varepsilon}\right)$$

If we choose ε = 1, we get:

$$1000n^2 = O\left(n^{3-1}\right) = O\left(n^2\right)$$

Since this equation holds, the first case of the master theorem applies to the given recurrence relation, thus resulting in the conclusion:

$$T(n) = \Theta\left(n^{\log_b a}\right).$$

If we insert the values from above, we finally get:

$$T(n) = \Theta\left(n^3\right).$$

Thus the given recurrence relation *T*(n) was in $\Theta(n^3)$.

CASE 2

Generic Form

If it is true, for some constant $k \geq 0$, that:

$$f(n) = \Theta\left(n^{\log_b a} \log^k n\right)$$

it follows that:

$$T(n) = \Theta\left(n^{\log_b a} \log^{k+1} n\right).$$

Example

$$T(n) = 2T\ \frac{n}{2}\ + 10n$$

As we can see in the formula above the variables get the following values:

$$a = 2, b = 2, k = 0, f(n) = 10n, \log_b a = \log_2 2 = 1$$

Now we have to check that the following equation holds (in this case k=0):

$$f(n) = \Theta\left(n^{\log_b a}\right)$$

If we insert the values from above, we get:

$$10n = \Theta\left(n^1\right) = \Theta(n)$$

Since this equation holds, the second case of the master theorem applies to the given recurrence relation, thus resulting in the conclusion:

$$T(n) = \Theta\left(n^{\log_b a} \log^{k+1} n\right).$$

If we insert the values from above, we finally get:

$$T(n) = \Theta(n \log n).$$

Thus the given recurrence relation *T*(*n*) was in Θ(*n* log *n*).

CASE 3

Generic Form

If it is true that:

$$f(n) = \Omega\left(n^{\log_b (a)+\varepsilon}\right) \text{for some constant } \varepsilon > 0$$

and if it is also true that:

$af \frac{n}{b} \leq cf(n)$ for some constant $c < 1$ and sufficiently large

n

it follows that:

$$T(n) = \Theta\left(f(n)\right)$$

Example

$$T(n) = 2T \frac{n}{2} + n^2$$

As we can see in the formula above the variables get the following values:

$$a = 2,\, b = 2,\, k = 0,\, f\left(n\right) = 10n,\, \log_b a = \log_2 2 = 1$$

Now we have to check that the following equation holds:

$$f\left(n\right) = \Omega\left(n^{\log_b a + \varepsilon}\right)$$

If we insert the values from above, and choose ε = 1, we get:

$$n^2 = \Omega\left(n^{1+1}\right) = \Omega\left(n^2\right)$$

Since this equation holds, we have to check the second condition, namely if it is true that:

$$af \;\; \frac{n}{b} \;\; \le cf\left(n\right)$$

If we insert once more the values from above, we get the number:

$$2 \;\; \frac{n}{2}^{\;\;2} \;\; \le cn^2 \Leftrightarrow \frac{1}{2}n^2 \le cn^2$$

If we choose $c = \frac{1}{2}$, it is true that:

$$\frac{1}{2}n^2 \le \frac{1}{2}n^2 \forall \ge 1$$

So it follows:

$$T\left(n\right) = \Theta\left(f\left(n\right)\right).$$

If we insert once more the necessary values, we get:

$$T\left(n\right) = \Theta\left(n^2\right).$$

Thus the given recurrence relation *T*(*n*) was in $\Theta(n^2)$, that complies with the *f* (*n*) of the original formula. (This result is confirmed by the exact solution of the recurrence relation, which is $T(n) = 2n^2 - n$, assuming $T(1) = 1$.)

COMPARING THE ALGORITHMS

There are two important factors when measuring the performance of a sorting algorithm. The algorithms have to compare the magnitude of different elements and they have to move the different elements around. So counting the number of comparisons and the number of exchanges or moves made by an algorithm offer useful performance measures.

When sorting large record structures, the number of exchanges made may be the principal performance criterion, since exchanging two records will involve a lot of work. When sorting a simple array of integers, then the number of comparisons will be more important.

It has been said that the only thing going for the bubble (exchange) sort is its catchy name. The logic of the algorithm is simple to understand and it is fairly easy to program. It can also be programmed to detect when it has finished sorting. The selection sort, by comparison, always goes through the same amount of work regardless of the data and the quick sort performs particularly badly with ordered data.

However, in general the bubble sort is a very inefficient algorithm. The insertion sort is a little better and whilst it cannot detect that it has finished sorting, the logic of the algorithm means that it comes to a rapid conclusion when dealing with sorted data. The selection sort is a good one to use with students. It is intuitive and very simple to program.

It offers quite good performance, its particular strength being the small number of exchanges needed. For a given number of data items, the selection sort always goes through a set number of comparisons and exchanges, so its performance is predictable.

The first three algorithms all offer O(n^2) performance, that is sorting times increase with the square of the number of elements being sorted.

That means that if you double the number of elements being sorted, then there will be a four-fold increase in the time taken.

Ten times more elements increases the time taken by a factor of 100! This is not a problem with small data sets, but with hundreds or thousands of elements, this becomes very significant. With most large data sets, the quick sort is a vastly superior algorithm (although as you might expect, it is much more complex), as the table below shows.

RANDOM DATA SET: NUMBER OF COMPARISONS MADE

Sort/Elements	50	100	200	300	400	500
Selection Sort	1225	4950	19900	44850	79800	124750
Exchange Sort	1410	5335	20300	45650	79866	126585
Insertion Sort	1391	5399	20473	44449	78779	123715
Quick Sort	399	990	1954	3384	5066	6256

It should be pointed out that the methods above all belong to one family, they are all internal sorting algorithms. This means that they can only be used when the entire data structure to be sorted can be held in the computer's main memory. There will be situations where this is not possible, for example when sorting a very large transaction file which is stored on, say, magnetic tape or disc.

DESCRIPTION

BUBBLE SORT

Exchange two adjacent elements if they are out of order. Repeat until array is sorted. This is a slow algorithm.

```
#include <stdlib.h>
#include <stdio.h>
#define uint32 unsigned int
typedef int (*CMPFUN)(int, int);
void ArraySort(int This[], CMPFUN fun_ptr, uint32 ub)
{
   /* bubble sort */
   uint32 indx;
   uint32 indx2;
   int temp;
   int temp2;
   int flipped;
   if (ub <= 1)
      return;
   indx = 1;
   do
   {
      flipped = 0;
      for (indx2 = ub - 1; indx2 >= indx; —indx2)
      {
         temp = This[indx2];
         temp2 = This[indx2 - 1];
         if ((*fun_ptr)(temp2, temp) > 0)
         {
            This[indx2 - 1] = temp;
            This[indx2] = temp2;
            flipped = 1;
         }
      }
   } while ((++indx < ub) && flipped);
}
#define ARRAY_SIZE 14
int my_array[ARRAY_SIZE];
void fill_array()
```

```
{
   int indx;
   for (indx=0; indx < ARRAY_SIZE; ++indx)
   {
     my_array[indx] = rand();
   }
   /* my_array[ARRAY_SIZE - 1] = ARRAY_SIZE/3; */
}
int cmpfun(int a, int b)
{
   if (a > b)
     return 1;
   else if (a < b)
     return -1;
   else
     return 0;
}
int main()
{
   int indx;
   int indx2;
   for (indx2 = 0; indx2 < 80000; ++indx2)
   {
     fill_array();
     ArraySort(my_array, cmpfun, ARRAY_SIZE);
     for (indx=1; indx < ARRAY_SIZE; ++indx)
     {
       if (my_array[indx - 1] > my_array[indx])
       {
         printf("bad sort\n");
         return(1);
       }
     }
   }
   return(0);
}
```

SELECTION SORT

Find the largest element in the array, and put it in the proper place. Repeat until array is sorted. This is also slow.

```
#include <stdlib.h>
#include <stdio.h>
#define uint32 unsigned int
typedef int (*CMPFUN)(int, int);
void ArraySort(int This[], CMPFUN fun_ptr, uint32 the_len)
{
   /* selection sort */
   uint32 indx;
   uint32 indx2;
   uint32 large_pos;
   int temp;
   int large;
   if (the_len <= 1)
      return;
   for (indx = the_len - 1; indx > 0; —indx)
   {
      /* find the largest number, then put it at the end of
the                  array */
      large = This[0];
      large_pos = 0;
      for (indx2 = 1; indx2 <= indx; ++indx2)
      {
         temp = This[indx2];
         if ((*fun_ptr)(temp,large) > 0)
         {
            large = temp;
            large_pos = indx2;
         }
      }
      This[large_pos] = This[indx];
      This[indx] = large;
   }
}
#define ARRAY_SIZE 14
int my_array[ARRAY_SIZE];
void fill_array()
{
   int indx;
   for (indx=0; indx < ARRAY_SIZE; ++indx)
   {
      my_array[indx] = rand();
```

```
    }
    /* my_array[ARRAY_SIZE - 1] = ARRAY_SIZE/3; */
}
int cmpfun(int a, int b)
{
    if (a > b)
       return 1;
    else if (a < b)
       return -1;
    else
       return 0;
}
    int main()
{
    int indx;
    int indx2;
    for (indx2 = 0; indx2 < 80000; ++indx2)
    {
       fill_array();
       ArraySort(my_array, cmpfun, ARRAY_SIZE);
       for (indx=1; indx < ARRAY_SIZE; ++indx)
       {
          if (my_array[indx - 1] > my_array[indx])
          {
             printf("bad sort\n");
             return(1);
          }
       }
    }
    return(0);
}
```

INSERTION SORT

Scan successive elements for out of order item, then insert the item in the proper place. Sort small array fast, big array very slowly.

```
#include <stdlib.h>
#include <stdio.h>
#define uint32 unsigned int
```

```
  typedef int (*CMPFUN)(int, int);
  void ArraySort(int This[], CMPFUN fun_ptr, uint32 the_len)
  {
     /* insertion sort */
     uint32 indx;
     int cur_val;
     int prev_val;
     if (the_len <= 1)
        return;
     prev_val = This[0];
     for (indx = 1; indx < the_len; ++indx)
     {
        cur_val = This[indx];
        if ((*fun_ptr)(prev_val, cur_val) > 0)
        {
           /* out of order: array[indx-1] > array[indx] */
           uint32 indx2;
           This[indx] = prev_val;/* move up the larger item
first */
           /* find the insertion point for the smaller item */
           for (indx2 = indx - 1; indx2 > 0;)
           {
              int temp_val = This[indx2 - 1];
              if ((*fun_ptr)(temp_val, cur_val) > 0)
              {
                 This[indx2—] = temp_val;
                 /* still out of order, move up 1 slot to make
room */
              }
              else
                 break;
           }
           This[indx2] = cur_val;/* insert the smaller item
right here */
        }
        else
        {
           /* in order, advance to next element */
           prev_val = cur_val;
        }
     }
```

```
}
#define ARRAY_SIZE 14
int my_array[ARRAY_SIZE];
uint32 fill_array()
{
   int indx;
   uint32 checksum = 0;
   for (indx=0; indx < ARRAY_SIZE; ++indx)
   {
      checksum += my_array[indx] = rand();
   }
   return checksum;
}
int cmpfun(int a, int b)
{
   if (a > b)
   return 1;
   else if (a < b)
   return -1;
   else
   return 0;
}
int main()
{
   int indx;
   int indx2;
   uint32 checksum1;
   uint32 checksum2;
   for (indx2 = 0; indx2 < 80000; ++indx2)
   {
      checksum1 = fill_array();
      ArraySort(my_array, cmpfun, ARRAY_SIZE);
      for (indx=1; indx < ARRAY_SIZE; ++indx)
      {
         if (my_array[indx - 1] > my_array[indx])
         {
            printf("bad sort\n");
            return(1);
         }
```

```
        }
        checksum2 = 0;
        for (indx=0; indx < ARRAY_SIZE; ++indx)
        {
          checksum2 += my_array[indx];
        }
        if (checksum1 != checksum2)
        {
          printf("bad checksum %d %d\n", checksum1, checksum2);
        }
      }
      return(0);
  }
```

QUICK SORT

Partition Array into Two Segments

The first segment all elements are less than or equal to the pivot value. The second segment all elements are greater or equal to the pivot value. Sort the two segments recursively. Quicksort is fastest on average, but sometimes unbalanced partitions can lead to very slow sorting.

```
  #include <stdlib.h>
  #include <stdio.h>
  #define INSERTION_SORT_BOUND 16/* boundary point to use
insertion sort */
  #define uint32 unsigned int
  typedef int (*CMPFUN)(int, int);
  /* explain function
   * Description:
   * fixarray::Qsort() is an internal subroutine that implements
quick             sort.
   *
   * Return Value: none
   */
  void Qsort(int This[], CMPFUN fun_ptr, uint32 first, uint32
last)
```

```
  {
     uint32 stack_pointer = 0;
     int first_stack[32];
     int last_stack[32];
     for (;; {
        if (last - first <= INSERTION_SORT_BOUND)
        {
           /* for small sort, use insertion sort */
           uint32 indx;
           int prev_val = This[first];
           int cur_val;
           for (indx = first + 1; indx <= last; ++indx){
              cur_val = This[indx];
              if ((*fun_ptr)(prev_val, cur_val) > 0){
                 /* out of order: array[indx-1] > array[indx]
*/
                 uint32 indx2;
                 This[indx] = prev_val;/* move up the larger
item first */
                 /* find the insertion point for the smaller
item */
                 for (indx2 = indx - 1; indx2 > first;)
                 {
                    int temp_val = This[indx2 - 1];
                    if ((*fun_ptr)(temp_val, cur_val) > 0)
                    {
                       This[indx2—] = temp_val;
                       /* still out of order, move up 1 slot
to make                             room */
                    }
                    else
                    break;
                 }
                 This[indx2] = cur_val;/* insert the smaller
item right                    here */
              }
              else
              {
                 /* in order, advance to next element */
                 prev_val = cur_val;
              }
```

```
            }
         }
         else
         {
            int pivot;
            /* try quick sort */
            {
               int temp;
               uint32 med = (first + last) >> 1;
               /* Choose pivot from first, last, and median
position. */
               /* Sort the three elements. */
               temp = This[first];
               if ((*fun_ptr)(temp, This[last]) > 0)
               {
                  This[first] = This[last]; This[last] = temp;
               }
               temp = This[med];
               if ((*fun_ptr)(This[first], temp) > 0)
               {
                  This[med] = This[first]; This[first] = temp;
               }
               temp = This[last];
               if ((*fun_ptr)(This[med], temp) > 0)
               {
                  This[last] = This[med]; This[med] = temp;
               }
               pivot = This[med];
            }
            {
               uint32 up;
   {
      uint32 down;
      /* First and last element will be loop stopper. */
      /* Split array into two partitions. */
      down = first;
      up = last;
      for (;;)
      {
         do
         {
            ++down;
         } while ((*fun_ptr)(pivot, This[down]) > 0);
         do
         {
            —up;
         } while ((*fun_ptr)(This[up], pivot) > 0);
         if (up > down)
         {
            int temp;
            /* interchange L[down] and L[up] */
            temp = This[down]; This[down]= This[up]; This[up]
= temp;
         }
         else
         break;
         }
      }
      {
         uint32 len1;/* length of first segment */
         uint32 len2;/* length of second segment */
         len1 = up - first + 1;
         len2 = last - up;
         /* stack the partition that is larger */
         if (len1 >= len2)
         {
            first_stack[stack_pointer] = first;
            last_stack[stack_pointer++] = up;
            first = up + 1;
            /* tail recursion elimination of
            * Qsort(This,fun_ptr,up + 1,last)
            */
         }
         else
         {
            first_stack[stack_pointer] = up + 1;
            last_stack[stack_pointer++] = last;
            last = up;
            /* tail recursion elimination of
            * Qsort(This,fun_ptr,first,up)
            */
         }
```

```
   }
   continue;
   }
      /* end of quick sort */
   }
   if (stack_pointer > 0)
   {
      /* Sort segment from stack. */
      first = first_stack[—stack_pointer];
      last = last_stack[stack_pointer];
   }
   else
   break;
   }/* end for */
}
void ArraySort(int This[], CMPFUN fun_ptr, uint32 the_len)
{
   Qsort(This, fun_ptr, 0, the_len - 1);
}
   #define ARRAY_SIZE 250000
   int my_array[ARRAY_SIZE];
   uint32 fill_array()
   {
      int indx;
      uint32 checksum = 0;
      for (indx=0; indx < ARRAY_SIZE; ++indx)
      {
         checksum += my_array[indx] = rand();
      }
      return checksum;
   }
   int cmpfun(int a, int b)
   {
      if (a > b)
         return 1;
      else if (a < b)
         return -1;
      else
         return 0;
      }
      int main()
   {
      int indx;
      uint32 checksum1;
      uint32 checksum2 = 0;
      checksum1 = fill_array();
      ArraySort(my_array, cmpfun, ARRAY_SIZE);
      for (indx=1; indx < ARRAY_SIZE; ++indx)
      {
         if (my_array[indx - 1] > my_array[indx])
         {
            printf("bad sort\n");
         return(1);
      }
   }
      for (indx=0; indx < ARRAY_SIZE; ++indx)
   {
      checksum2 += my_array[indx];
   }
   if (checksum1 != checksum2)
   {
      printf("bad checksum %d %d\n", checksum1, checksum2);
      return(1);
   }
   return(0);
}
```

MERGE SORT

Start from two sorted runs of length 1, merge into a single run of twice the length. Repeat until a single sorted run is left. Mergesort needs N/2 extra buffer. Performance is second place on average, with quite good speed on nearly sorted array. Mergesort is stable in that two elements that are equally ranked in the array will not have their relative positions flipped.

```
#include <stdlib.h>
#include <stdio.h>
#define uint32 unsigned int
typedef int (*CMPFUN)(int, int);
```

```
  #define  INSERTION_SORT_BOUND  8/*  boundary  point  to  use
insertion sort */
  void ArraySort(int This[], CMPFUN fun_ptr, uint32 the_len)
  {
     uint32 span;
     uint32 lb;
     uint32 ub;
     uint32 indx;
     uint32 indx2;
     if (the_len <= 1)
        return;
     span = INSERTION_SORT_BOUND;
     /* insertion sort the first pass */
     {
        int prev_val;
        int cur_val;
        int temp_val;
        for (lb = 0; lb < the_len; lb += span
        {
           if ((ub = lb + span) > the_len) ub = the_len;
           prev_val = This[lb];
           for (indx = lb + 1; indx < ub; ++indx)
           {
              cur_val = This[indx];
              if ((*fun_ptr)(prev_val, cur_val) > 0)
           {
              /* out of order: array[indx-1] > array[indx]
*/
              This[indx] = prev_val;/* move up the larger
item first */
              /* find the insertion point for the smaller
item */
              for (indx2 = indx - 1; indx2 > lb;)
              {
                 temp_val = This[indx2 - 1];
                 if ((*fun_ptr)(temp_val, cur_val) > 0)
                 {
                    This[indx2—] = temp_val;
                    /* still out of order, move up 1 slot
to make                                room */
                 }
```

```
                    else
                        break;
                }
                This[indx2] = cur_val;/* insert the smaller
item right                    here */
            }
            else
            {
            /* in order, advance to next element */
            prev_val = cur_val;
            }
        }
    }
}
/* second pass merge sort */
{
    uint32 median;
    int* aux;
    aux = (int*) malloc(sizeof(int) * the_len/2);
    while (span < the_len)
    {
        /* median is the start of second file */
        for (median = span; median < the_len;)
        {
            indx2 = median - 1;
            if ((*fun_ptr)(This[indx2], This[median]) > 0)
            {
                /* the two files are not yet sorted */
                if ((ub = median + span) > the_len)
                {
                    ub = the_len;
                }
                /*  skip  over  the  already  sorted  largest
elements */
                while ((*fun_ptr)(This[—ub], This[indx2]) >=
0){
                }
                /* copy second file into buffer */
                for (indx = 0; indx2 < ub; ++indx)
                {
                    *(aux + indx) = This[++indx2];
```

```
}
—indx;
indx2 = median - 1;
lb = median - span;
/* merge two files into one */
for (;;)
{
    if ((*fun_ptr)(*(aux + indx), This[indx2])
>= 0)
    {
        This[ub—] = *(aux + indx);
        if (indx > 0) —indx;
        else
        {
            /* second file exhausted */
            for (;;)
            {
                This[ub—] = This[indx2];
                if (indx2 > lb) —indx2;
                else goto mydone;/* done */
            }
        }
    }
    else
    {
        This[ub—] = This[indx2];
        if (indx2 > lb) —indx2;
        else
        {
            /* first file exhausted */
            for (;;)
            {
                This[ub—] = *(aux + indx);
                if (indx > 0) —indx;
                else goto mydone;/* done */
                }
            }
        }
    }
}
mydone:
```

```
                median += span + span;
            }
            span += span;
        }
        free(aux);
    }
}
#define ARRAY_SIZE 250000
int my_array[ARRAY_SIZE];
uint32 fill_array()
{
    int indx;
    uint32 sum = 0;
    for (indx=0; indx < ARRAY_SIZE; ++indx)
    {
        sum += my_array[indx] = rand();
    }
    return sum;
}
int cmpfun(int a, int b)
{
    if (a > b)
        return 1;
    else if (a < b)
        return -1;
    else
        return 0;
    }
    int main()
    {
        int indx;
        uint32 checksum, checksum2;
        checksum = fill_array();
        ArraySort(my_array, cmpfun, ARRAY_SIZE);
        checksum2 = my_array[0];
        for (indx=1; indx < ARRAY_SIZE; ++indx)
        {
            checksum2 += my_array[indx];
            if (my_array[indx - 1] > my_array[indx])
            {
                printf("bad sort\n");
```

```
              return(1);
            }
         }
         if (checksum != checksum2){
            printf("bad checksum %d %d\n", checksum,
checksum2);
            return(1);
      }
      return(0);
  }
```

HEAP SORT

Form a tree with parent of the tree being larger than its children. Remove the parent from the tree successively. On average, Heapsort is third place in speed. Heapsort does not need extra buffer, and performance is not sensitive to initial distributions.

```
  #include <stdlib.h>
  #include <stdio.h>
  #define uint32 unsigned int
  typedef int (*CMPFUN)(int, int);
  void ArraySort(int This[], CMPFUN fun_ptr, uint32 the_len)
  {
    /* heap sort */
    uint32 half;
    uint32 parent;
    if (the_len <= 1)
      return;
    half = the_len >> 1;
    for (parent = half; parent >= 1; —parent)
    {
      int temp;
      int level = 0;
      uint32 child;
      child = parent;
      /* bottom-up downheap */
      /* leaf-search for largest child path */
      while (child <= half)
      {
```

```
    ++level;
    child += child;
    if ((child < the_len) &&
    ((*fun_ptr)(This[child], This[child - 1]) > 0))
    ++child;
  }
    /* bottom-up-search for rotation point */
    temp = This[parent - 1];
    for (;;)
    {
      if (parent == child)
        break;
      if ((*fun_ptr)(temp, This[child - 1]) <= 0)
        break;
      child >>= 1;
    —level;
  }
  /* rotate nodes from parent to rotation point */
  for (;level > 0; —level)
  {
    This[(child >> level) - 1] =
    This[(child >> (level - 1)) - 1];
  }
  This[child - 1] = temp;
}
  —the_len;
  do
  {
    int temp;
    int level = 0;
    uint32 child;
    /* move max element to back of array */
    temp = This[the_len];
    This[the_len] = This[0];
    This[0] = temp;
    child = parent = 1;
    half = the_len >> 1;
    /* bottom-up downheap */
    /* leaf-search for largest child path */
    while (child <= half)
    {
```

```
    ++level;
    child += child;
    if ((child < the_len) &&
    ((*fun_ptr)(This[child], This[child - 1]) > 0))
    ++child;
   }
   /* bottom-up-search for rotation point */
   for (;;)
   {
    if (parent == child)
    break;
    if ((*fun_ptr)(temp, This[child - 1]) <= 0)
    break;
    child >>= 1;
   —level;
   }
   /* rotate nodes from parent to rotation point */
   for (;level > 0; —level)
   {
    This[(child >> level) - 1] =
    This[(child >> (level - 1)) - 1];
   }
   This[child - 1] = temp;
  } while (—the_len >= 1);
}
#define ARRAY_SIZE 250000
int my_array[ARRAY_SIZE];
void fill_array()
{
 int indx;
 for (indx=0; indx < ARRAY_SIZE; ++indx)
 {
  my_array[indx] = rand();
 }
}
int cmpfun(int a, int b)
{
 if (a > b)
  return 1;
 else if (a < b)
  return -1;
```

```
   else
     return 0;
   }
   int main()
   {
     int indx;
     fill_array();
     ArraySort(my_array, cmpfun, ARRAY_SIZE);
     for (indx=1; indx < ARRAY_SIZE; ++indx)
     {
       if (my_array[indx - 1] > my_array[indx])
       {
         printf("bad sort\n");
           return(1);
       }
     }
     return(0);
 }
```

SHELL SORT

Sort every Nth element in an array using insertion sort. Repeat using smaller N values, until N = 1. On average, Shellsort is fourth place in speed. Shellsort may sort some distributions slowly.

```
#include <stdlib.h>
#include <stdio.h>
#define uint32 unsigned int
typedef int (*CMPFUN)(int, int);
/* Calculated from the combinations of 9 * (4^n - 2^n) + 1,
* and 4^n - 3 * 2^n + 1
*/
uint32 hop_array[] =
{
1,
5,
19,
41,
109,
209,
505,
929,
2161,
3905,
8929,
16001,
36289,
64769,
146305,
260609,
587521,
1045505,
2354689,
4188161,
9427969,
16764929,
37730305,
67084289,
150958081,
268386305,
603906049,
1073643521,
2415771649,
0xffffffff };
void ArraySort(int This[], CMPFUN fun_ptr, uint32 the_len)
{
  /* shell sort */
  int level;
  for (level = 0; the_len > hop_array[level]; ++level);
  do
  {
    uint32 dist;
    uint32 indx;
    dist = hop_array[—level];
    for (indx = dist; indx < the_len; ++indx)
    {
      int cur_val;
      uint32 indx2;
      cur_val = This[indx];
      indx2 = indx;
      do
      {
        int early_val;
        early_val = This[indx2 - dist];
        if ((*fun_ptr)(early_val, cur_val) <= 0)
        break;
        This[indx2] = early_val;
        indx2 -= dist;
      } while (indx2 >= dist);
      This[indx2] = cur_val;
    }
  } while (level >= 1);
}
#define ARRAY_SIZE 250000
int my_array[ARRAY_SIZE];
uint32 fill_array()
{
  int indx;
  uint32 checksum = 0;
  for (indx=0; indx < ARRAY_SIZE; ++indx)
  {
    checksum += my_array[indx] = rand();
  }
  return checksum;
}
int cmpfun(int a, int b)
{
  if (a > b)
    return 1;
  else if (a < b)
    return -1;
  else
    return 0;
  }
  int main()
  {
    int indx;
    uint32 sum1;
    uint32 sum2;
    sum1 = fill_array();
    ArraySort(my_array, cmpfun, ARRAY_SIZE);
    for (indx=1; indx < ARRAY_SIZE; ++indx)
```

```
      {
        if (my_array[indx - 1] > my_array[indx])
        {
          printf("bad sort\n");
            return(1);
        }
      }
      for (indx = 0; indx < ARRAY_SIZE; ++indx)
      {
        sum2 += my_array[indx];
      }
      if (sum1 != sum2)
      {
        printf("bad checksum\n");
          return(1);
      }
    return(0);
  }
```

COMBO SORT

Sorting algorithms can be mixed and matched to yield the desired properties. We want fast average performance, good worst case performance, and no large extra storage requirement. We can achieve the goal by starting with the Quicksort (fastest on average). We modify Quicksort by sorting small partitions by using Insertion Sort (best with small partition). If we detect two partitions are badly balanced, we sort the larger partition by Heapsort (good worst case performance). Of course we cannot undo the bad partitions, but we can stop the possible degenerate case from continuing to generate bad partitions.

```
  #include <stdlib.h>
  #include <stdio.h>
  #define uint32 unsigned int
  typedef int (*CMPFUN)(int, int);
  void HelperHeapSort(int This[], CMPFUN fun_ptr, uint32 first,
uint32  the_len)
```

```
  {
   /* heap sort */
   uint32 half;
   uint32 parent;
   if (the_len <= 1)
    return;
   half = the_len >> 1;
   for (parent = half; parent >= 1; —parent)
   {
    int temp;
    int level = 0;
    uint32 child;
    child = parent;
    /* bottom-up downheap */
    /* leaf-search for largest child path */
    while (child <= half)
    {
     ++level;
     child += child;
     if ((child < the_len) &&
     ((*fun_ptr)(This[first + child], This[first + child -
1]) > 0))
     ++child;
    }
    /* bottom-up-search for rotation point */
    temp = This[first + parent - 1];
    for (;;)
    {
     if (parent == child)
      break;
     if ((*fun_ptr)(temp, This[first + child - 1]) <= 0)
      break;
     child >>= 1;
     —level;
    }
     /* rotate nodes from parent to rotation point */
     for (;level > 0; —level)
    {
     This[first + (child >> level) - 1] =
     This[first + (child >> (level - 1)) - 1];
    }
     This[first + child – 1] = temp;
   }
   —the_len;
   do
   {
    int temp;
    int level = 0;
    uint32 child;
    /* move max element to back of array */
    temp = This[first + the_len];
    This[first + the_len] = This[first];
    This[first] = temp;
    child = parent = 1;
    half = the_len >> 1;
    /* bottom–up downheap */
    /* leaf–search for largest child path */
    while (child <= half)
    {
     ++level;
     child += child;
     if ((child < the_len) &&
     ((*fun_ptr)(This[first + child], This[first + child –
1]) > 0))
     ++child;
   }
   /* bottom–up–search for rotation point */
   for (;;)
   {
    if (parent == child)
     break;
    if ((*fun_ptr)(temp, This[first + child – 1]) <= 0)
     break;
    child >>= 1;
    —level;
    }
    /* rotate nodes from parent to rotation point */
    for (;level > 0; —level)
    {
     This[first + (child >> level) – 1] =
     This[first + (child >> (level – 1)) – 1];
    }
```

```
    This[first + child – 1] = temp;
   } while (—the_len >= 1);
  }
  #define INSERTION_SORT_BOUND 16/* boundary point to use
insertion sort */
  /* explain function
   * Description:
   * fixarray::Qsort() is an internal subroutine that implements
quick     sort.
   *
   * Return Value: none
   */
  void Qsort(int This[], CMPFUN fun_ptr, uint32 first, uint32
last)
  {
   uint32 stack_pointer = 0;
   int first_stack[32];
   int last_stack[32];
   for (;;)
   {
    if (last – first <= INSERTION_SORT_BOUND)
    {
     /* for small sort, use insertion sort */
     uint32 indx;
     int prev_val = This[first];
     int cur_val;
     for (indx = first + 1; indx <= last; ++indx)
     {
      cur_val = This[indx];
      if ((*fun_ptr)(prev_val, cur_val) > 0)
      {
       uint32 indx2;
       /* out of order */
       This[indx] = prev_val;
       for (indx2 = indx – 1; indx2 > first; —indx2)
       {
        int temp_val = This[indx2 – 1];
        if ((*fun_ptr)(temp_val, cur_val) > 0)
        {
         This[indx2] = temp_val;
        }
```

```
          else
            break;
        }
        This[indx2] = cur_val;
      }
      else
      {
        /* in order, advance to next element */
        prev_val = cur_val;
      }
    }
  }
  else
  {
    int pivot;
    /* try quick sort */
    {
      int temp;
      uint32 med = (first + last) >> 1;
      /* Choose pivot from first, last, and median position.
*/
      /* Sort the three elements. */
      temp = This[first];
      if ((*fun_ptr)(temp, This[last]) > 0)
      {
        This[first] = This[last]; This[last] = temp;
      }
      temp = This[med];
      if ((*fun_ptr)(This[first], temp) > 0)
      {
        This[med] = This[first]; This[first] = temp;
      }
      temp = This[last];
      if ((*fun_ptr)(This[med], temp) > 0)
      {
        This[last] = This[med]; This[med] = temp;
      }
      pivot = This[med];
    }
    {
      uint32 up;
      {
       uint32 down;
       /* First and last element will be loop stopper. */
       /* Split array into two partitions. */
       down = first;
       up = last;
       for (;;)
       {
        do
        {
         ++down;
        } while ((*fun_ptr)(pivot, This[down]) > 0);
        do
        {
         —up;
        } while ((*fun_ptr)(This[up], pivot) > 0);
        if (up > down)
        {
         int temp;
         /* interchange L[down] and L[up] */
         temp = This[down]; This[down]= This[up]; This[up]
= temp;
        }
        else
         break;
       }
      }
      {
       uint32 len1;/* length of first segment */
       uint32 len2;/* length of second segment */
       len1 = up - first + 1;
       len2 = last - up;
       if (len1 >= len2)
       {
        if ((len1 >> 5) > len2)
        {
         /* badly balanced partitions, heap sort first segment
*/
         HelperHeapSort(This, fun_ptr, first, len1);
        }
        else
        {
         first_stack[stack_pointer] = first;/* stack first
segment */
         last_stack[stack_pointer++] = up;
        }
        first = up + 1;
        /* tail recursion elimination of
        * Qsort(This,fun_ptr,up + 1,last)
        */
       }
       else
       {
        if ((len2 >> 5) > len1)
        {
         /* badly balanced partitions, heap sort second
segment */
         HelperHeapSort(This, fun_ptr, up + 1, len2);
        }
        else
        {
         first_stack[stack_pointer] = up + 1;/* stack second
segment */
         last_stack[stack_pointer++] = last;
        }
        last = up;
        /* tail recursion elimination of
        * Qsort(This,fun_ptr,first,up)
        */
       }
      }
      continue;
     }
     /* end of quick sort */
    }
    if (stack_pointer > 0)
    {
     /* Sort segment from stack. */
     first = first_stack[—stack_pointer];
     last = last_stack[stack_pointer];
     }
     else
    break;
  }/* end for */
}
void ArraySort(int This[], CMPFUN fun_ptr, uint32 the_len)
{
Qsort(This, fun_ptr, 0, the_len - 1);
}
#define ARRAY_SIZE 250000
int my_array[ARRAY_SIZE];
void fill_array()
{
  int indx;
  for (indx=0; indx < ARRAY_SIZE; ++indx)
  {
   my_array[indx] = rand();
  }
}
  int cmpfun(int a, int b)
  {
   if (a > b)
    return 1;
   else if (a < b)
    return -1;
   else
    return 0;
  }
  int main()
  {
  int indx;
  fill_array();
  ArraySort(my_array, cmpfun, ARRAY_SIZE);
  for (indx=1; indx < ARRAY_SIZE; ++indx)
  {
   if (my_array[indx - 1] > my_array[indx])
   {
    printf("bad sort\n");
    return(1);
   }
  }
  return(0);
}
```

4

Augmenting Path Algorithms

The neat part of the Ford-Fulkerson algorithm described above is that it gets the correct result no matter how we solve (correctly!!) the sub-problem of finding an augmenting path.

However, every new path may increase the flow by only 1, hence the number of iterations of the algorithm could be very large if we carelessly choose the augmenting path algorithm to use. The function *max_flow* will look like this, regardless of the actual method we use for finding augmenting paths:

```
int max_flow()
   result = 0
      while (true)
 // the function find_path returns the path            capacity of the
 augmenting path found
 path_capacity = find_path()
 // no augmenting path found
 if (d = 0) exit while
```

```
    else result += path_capacity
  end while
  return result
```

To keep it simple, we will use a 2-dimensional array for storing the capacities of the residual network that we are left with after each step in the algorithm. Initially the residual network is just the original network. We will not store the flows along the edges explicitly, but it's easy to figure out how to find them upon the termination of the algorithm: for each edge x-y in the original network the flow is given by the capacity of the backward edge y-x in the residual network.

Be careful though; if the reversed arc y-x also exists in the original network, this will fail, and it is recommended that the initial capacity of each arc be stored somewhere, and then the flow along the edge is the difference between the initial and the residual capacity.

We now require an implementation for the function *find_path*. The first approach that comes to mind is to use a depth-first search (DFS), as it probably is the easiest to implement. Unfortunately, its performance is very poor on some networks, and normally is less preferred to the ones discussed next.

The next best thing in the matter of simplicity is a breadth-first search (BFS). Recall that this search usually yields the shortest path in an un-weighted graph. Indeed, this also applies here to get the shortest augmenting path from the source to the sink. In the following pseudocode we will basically: find a shortest path from the source to the sink and compute the minimum capacity of an edge (that could be a forward or a backward edge) along the path - the path capacity. Then, for each edge along the path we reduce its capacity and increase the capacity of the reversed edge with the path capacity.

```
     int bfs()
       queue Q
         push source to Q
         mark source as visited
         keep an array from with the                semnification:  from[x]
is the
         previous vertex visited in the shortest
   path from the source to x;
         initialize from with -1 (or any other
sentinel value)
          while Q is not empty
            where = pop from Q
          for each vertex next adjacent to where
          if next is not visited and              capacity[where][next] > 0
            push next to Q
            mark next as visited
            from[next] = where
            if next = sink
              exit while loop
        end for
   end while
   // we compute the path capacity
   where = sink, path_cap = infinity
   while from[where] > -1
      prev = from[where]// the previous vertex
      path_cap = min(path_cap,                    capacity[prev][where])
      where = prev
    end while
      // we update the residual network; if no
path is found the while
      loop will not be entered
      where = sink
         while from[where] > -1
           prev = from[where]
           capacity[prev][where] -= path_capacity
             capacity[where][prev] +=
   path_capacity
```

```
        where = prev
    end while
    // if no path is found, path_cap is infinity
    if path_cap = infinity
        return 0
    else return path_cap
```

As we can see, this is pretty easy to implement. As for its performance, it is guaranteed that this takes at most *N * M/2* steps, where N is the number of vertices and M is the number of edges in the network. This number may seem very large, but it is over-estimated for most networks. For example, in the network we considered 3 augmenting paths are needed which is significantly less than the upper bound of 28. Due to the *O*(*M*) running time of BFS (implemented with adjacency lists) the worst-case running time of the shortest-augmenting path max-flow algorithm is $O(N * M^2)$, but usually the algorithm performs much better than this.

Next we will consider an approach that uses a priority-first search (PFS), that is very similar to the Dijkstra heap method explained here. In this method the augmenting path with a maximum path capacity is preferred. Intuitively this would lead to a faster algorithm, since at each step we increase the flow with the maximum possible amount. However, things are not always so, and the BFS implementation has better running times on some networks.

We assign as a priority to each vertex the minimum capacity of a path (in the residual network) from the source to that vertex. We process vertices in a greedy manner, as in Dijkstra's algorithm, in decreasing order of priorities. When we get to the sink, we are done, since a path with a maximum capacity is found. We would like to implement this with a data structure that allows us to efficiently find the vertex with the highest priority and increase the priority of a vertex (when a new better path is found) - this suggests the use of a heap which has a space complexity proportional to the number of vertices.

In TopCoder matches we may find it faster and easier to implement this with a priority queue or some other data structure that approximates one, even though the space required might grow to being proportional with the number of edges. This is how the following pseudocode is implemented. We also define a structure node that has the members vertex and prioritywith the above significance. Another field from is needed to store the previous vertex on the path.

```
    int pfs()
      priority queue PQ
      push node(source, infinity, -1) to PQ
      keep the array from as in bfs()
      // if no augmenting path is found, path_cap
will remain 0
      path_cap = 0
      while PQ is not empty
          node aux = pop from PQ
          where = aux.vertex, cost = aux.priority
          if we already visited where continue
          from[where] = aux.from
          if where = sink
             path_cap = cost
             exit while loop
            mark where as visited
            for each vertex next adjacent to where
               if capacity[where][next] > 0
                  new_cost = min(cost,
    capacity[where][next])
                  push node(next, new_cost, where) to
            PQ
          end for
end while
// update the residual network
where = sink
while from[where] > -1
   prev = from[where]
   capacity[prev][where] -= path_cap
   capacity[where][prev] += path_cap
   where = prev
end while
   return path_cap
```

The analysis of its performance is pretty complicated, but it may prove worthwhile to remember that with PFS at most $2M1gU$ steps are required, where U is the maximum capacity of an edge in the network.

As with BFS, this number is a lot larger than the actual number of steps for most networks. Combine this with the $O(M\ 1g\ M)$complexity of the search to get the worst-case running time of this algorithm.

Now that we know what these methods are all about, which of them do we choose when we are confronted with a max-flow problem? The PFS approach seems to have a better worst-case performance, but in practice their performance is pretty much the same. So, the method that one is more familiar with may prove more adequate. Personally, I prefer the shortest-path method, as I find it easier to implement during a contest and less error prone.

MAXIMUM FLOW PROBLEM

The maximum flow problem is again structured on a network; but here the arc capacities, or upper bounds, are the only relevant parameters.

The problem is to find the maximum flow possible from some given source node to a given sink node. A network model is in Fig. All arc costs are zero, but the cost on the arc leaving the sink is set to -1. Since the goal of the optimization is to minimize cost, the maximum flow possible is delivered to the sink node.

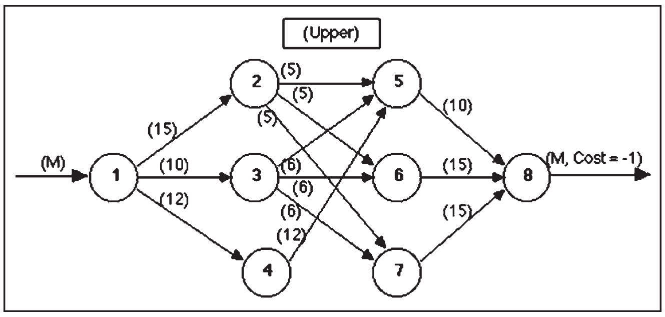


Fig. Network Model for the Maximum Flow Problem.

The solution to the example is in Fig. The maximum flow from node 1 to node 8 is 30 and the flows that yield this flow are shown on the figure. The heavy arcs on the figure are called the minimal cut.

These arcs are the bottlenecks that are restricting the maximum flow. The fact that the sum of the capacities of the arcs on the minimal cut equals the maximum flow is a famous theorem of network theory called the max flow min cut theorem. The arcs on the minimum cut can be identified using sensitivity analysis.

MAX-FLOW/MIN-CUT RELATED PROBLEMS

How to recognize max-flow problems? Often they are hard to detect and usually boil down to maximizing the movement of something from a location to another. We need to look at the constraints when we think we have a working solution based on maximum flow - they should suggest at least an $O(N^3)$ approach. If the number of locations is large, another algorithm (such as dynamic Programmeming or greedy), is more appropriate. The problem description might suggest multiple sources and/or sinks. For example, in the sample statement in the beginning of this article, the company might own more than one factory and multiple distribution Centres. How can we deal with this? We should try to convert this to a network that has a unique source and sink.

In order to accomplish this we will add two "dummy" vertices to our original network - we will refer to them as super-source and super-sink. In addition to this we will add an edge from the super-source to every ordinary source (a factory). As we don't have restrictions on the number of trucks that each factory can send, we should assign to each edge an infinite capacity.

Note that if we had such restrictions, we should have assigned to each edge a capacity equal to the number of trucks each factory could send. Likewise, we add an edge from every ordinary sink (distribution Centres) to the super-sink with infinite capacity.

A maximum flow in this new-built network is the solution to the problem - the sources now become ordinary vertices, and they are subject to the entering-flow equals leaving-flow property. You may want to keep this in your bag of tricks, as it may prove useful to most problems.

What if we are also given the maximum number of trucks that can drive through each of the cities in the country (other than the cities where the factory and the distribution Centre are located)? In other words we have to deal with vertex-capacities too. Intuitively, we should be able to reduce this to maximum-flow, but we must find a way to take the capacities from vertices and put them back on edges, where they belong. Another nice trick comes into play. We will build a network that has two times more vertices than the initial one. For each vertex we will have two nodes: an in-vertex and an out-vertex, and we will direct each edge x-y from the out-vertex of x to the in-vertex of y.

We can assign them the capacities from the problem statement. Additionally we can add an edge for each vertex from the in to the out-vertex.

The capacity this edge will be assigned is obviously the vertex-capacity. Now we just run max-flow on this network and compute the result. Maximum flow problems may appear out of nowhere. Let's take this problem for instance:"You are given the in and out degrees of the vertices of a directed graph.

Your task is to find the edges (assuming that no edge can appear more than once)." First, notice that we can perform this simple test at the beginning.

We can compute the number M of edges by summing the out-degrees or the in-degrees of the vertices. If these numbers are not equal, clearly there is no graph that could be built. This doesn't solve our problem, though.

There are some greedy approaches that come to mind, but none of them work.

We will combine the tricks discussed above to give a max-flow algorithm that solves this problem. First, build a network that has 2 (in/out) vertices for each initial vertex. Now draw an edge from every out vertex to every in vertex. Next, add a super-source and draw an edge from it to every out-vertex. Add a super-sink and draw an edge from every in vertex to it. We now need some capacities for this to be a flow network.

It should be pretty obvious what the intent with this approach is, so we will assign the following capacities: for each edge drawn from the super-source we assign a capacity equal to the out-degree of the vertex it points to.

As there may be only one arc from a vertex to another, we assign a 1 capacity to each of the edges that go from the outs to the ins.

As you can guess, the capacities of the edges that enter the super-sink will be equal to the in-degrees of the vertices. If the maximum flow in this network equals M - the number of edges, we have a solution, and for each edge between the out and in vertices that has a flow along it (which is maximum 1, as the capacity is 1) we can draw an edge between corresponding vertices in our graph.

Note that both x-y and y-x edges may appear in the solution. This is very similar to the maximum matching in a bipartite graph that we will discuss later. An example is given below where the out-degrees are (2, 1, 1, 1) and the in-degrees (1, 2, 1, 1). Some other problems may ask to separate two locations minimally. Some of these problems usually can be reduced to minimum-cut in a network. Two examples will be discussed here, but first let's take the standard min-cut problem and make it sound more like a TopCoder problem. We learned earlier how to find the value of the min-cut and how to find an arbitrary min-cut. In addition to this we will now like to have a minimum-cut with the minimum number of edges.

An idea would be to try to modify the original network in such a way that the minimum cut here is the minimum cut with the minimum edges in the original one.

Notice what happens if we multiply each edge capacity with a constant T. Clearly, the value of the maximum flow is multiplied by T, thus the value of the minimum cut is T times bigger than the original. A minimum cut in the original network is a minimum cut in the modified one as well. Now suppose we add 1 to the capacity of each edge. Is a minimum cut in the original network a minimum cut in this one? The answer is no, as we can see in Figure shown below, if we take T = 2. Why did this happen? Take an arbitrary cut. The value of the cut will be T times the original value of the cut, plus the number of edges in it.

Thus, a non-minimum cut in the first place could become minimum if it contains just a few edges. This is because the constant might not have been chosen properly in the beginning, as is the case in the example above. We can fix this by choosing T large enough to neutralize the difference in the number of edges between cuts in the network.

In the above example T = 4 would be enough, but to generalize, we take T = 10, one more than the number of edges in the original network, and one more than the number of edges that could possibly be in a minimum-cut. It is now true that a minimum-cut in the new network is minimum in the original network as well. However the converse is not true, and it is to our advantage. Notice how the difference between minimum cuts is now made by the number of edges in the cut. So we just find the min-cut in this new network to solve the problem correctly.

Let’s illustrate some more the min-cut pattern: “An undirected graph is given. What is the minimum number of edges that should be removed in order to disconnect the graph?" In other words the problem asks us to remove some edges in order for two nodes to be separated. This should ring a bell - a minimum cut approach might work. So far we have only seen maximum flow in directed graphs, but now we are facing an undirected one.

This should not be a very big problem though, as we can direct the graph by replacing every (undirected) edge x-y with two arcs: x-y and y-x. In this case the value of the min-cut is the number of edges in it, so we assign a 1 capacity to each of them. We are not asked to separate two given vertices, but rather to disconnect optimally any two vertices, so we must take every pair of vertices and treat them as the source and the sink and keep the best one from these minimum-cuts.

An improvement can be made, however. Take one vertex, let's say vertex numbered 1. Because the graph should be disconnected, there must be another vertex unreachable from it. So it suffices to treat vertex 1 as the source and iterate through every other vertex and treat it as the sink.

What if instead of edges we now have to remove a minimum number of vertices to disconnect the graph? Now we are asked for a different min-cut, composed of vertices. We must somehow convert the vertices to edges though. Recall the problem above where we converted vertex-capacities to edge-capacities. The same trick works here. First "un-direct" the graph as in the previous example.

Next double the number of vertices and deal edges the same way: an edge x-y is directed from the out-x vertex to in-y. Then convert the vertex to an edge by adding a 1-capacity arc from the in-vertex to the out-vertex.

Now for each two vertices we must solve the sub-problem of minimally separating them. So, just like before take each pair of vertices and treat the out-vertex of one of them as the source and the in-vertex of the other one as the sink (this is because the only arc leaving the in-vertex is the one that goes to the out-vertex) and take the lowest value of the maximum flow. This time we can't improve in the quadratic number of steps needed, because the first vertex may be in an optimum solution and by always considering it as the source we lose such a case.

5

Expressing Algorithms

Algorithms can be expressed in many kinds of notation, including natural languages, pseudo code, flowcharts, Programmeming languages orcontrol tables (processed by interpreters). Natural language expressions of algorithms tend to be verbose and ambiguous, and are rarely used for complex or technical algorithms.

Pseudocode, flowcharts and control tables are structured ways to express algorithms that avoid many of the ambiguities common in natural language statements, while remaining independent of a particular implementation language. Programmeming languages are primarily intended for expressing algorithms in a form that can be executed by a computer, but are often used as a way to define or document algorithms.

There is a wide variety of representations possible and one can express a given Turing machine Programme as a sequence of machine tables (see more at finite state machine and state transition table), as flowcharts (see more at state diagram), or as a form of rudimentary machine code or assembly code called "sets of quadruples"..

Sometimes it is helpful in the description of an algorithm to supplement small "flow charts" (state diagrams) with natural-language and/or arithmetic expressions written inside "block diagrams" to summarize what the "flow charts" are accomplishing.

Representations of algorithms are generally classed into three accepted levels of Turing machine description:

- *High-level description*: "...prose to describe an algorithm, ignoring the implementation details. At this level we do not need to mention how the machine manages its tape or head."
- *Implementation description*: "...prose used to define the way the Turing machine uses its head and the way that it stores data on its tape. At this level we do not give details of states or transition function."
- *Formal description*: Most detailed, "lowest level", gives the Turing machine's "state table".

COMPUTER ALGORITHMS

In computer systems, an algorithm is basically an instance of logic written in software by software developers to be effective for the intended "target" computer(s), in order for the software on the target machines to *do something*. For instance, if a person is writing software that is supposed to print out a PDF document located at the operating system folder "/My Documents" at computer drive "D:" every Friday, they will write an algorithm that specifies the following actions: "If today's date (computer time) is 'Friday,' open the document at 'D:/My Documents' and call the 'print' function".

While this simple algorithm does not look into whether the printer has enough paper or whether the document has been moved into a different location, one can make this algorithm more robust and anticipate these problems by rewriting it as a formal CASE statement or as a (carefully crafted) sequence of IF-THEN-ELSE statements. For example the CASE statement might appear as follows (there are other possibilities):

- *Case* 1: IF today's date is NOT Friday THEN *exit this CASE instruction* ELSE
- *Case* 2: IF today's date is Friday AND the document is located at 'D:/My Documents' AND there is paper in the printer THEN print the document (and *exit this CASE instruction*) ELSE
- *Case* 3: IF today's date is Friday AND the document is NOT located at 'D:/My Documents' THEN display 'document not found' error message (and *exit this CASE instruction*) ELSE
- *Case* 4: IF today's date is Friday AND the document is located at 'D:/My Documents' AND there is NO paper in the printer THEN, (i) display 'out of paper' error message (ii) *exit*. Note that CASE 3 includes two possibilities: (i) the document is NOT located at 'D:/My Documents' AND there's paper in the printer OR (ii) the document is NOT located at 'D:/My Documents' AND there's NO paper in the printer.

The sequence of IF-THEN-ELSE tests might look like this:

- *Test* 1: IF today's date is NOT Friday THEN *done* ELSE TEST 2:
- *Test* 2: IF the document is NOT located at 'D:/My Documents' THEN display 'document not found' error message ELSE TEST 3:
- *Test* 3: IF there is NO paper in the printer THEN display 'out of paper' error message ELSE print the document.

These examples' logic grants precedence to the instance of "NO document at 'D:/My Documents' ".

DATA TYPES

Since C allows you to define new data types we shall not be able to cover all of the possiblities, only the most important examples. The most important of these are

File: The type which files are classified under

Enum: Enumerated type for abstract data

Void: The "empty" type

Volatile: New ANSI standard type for memory mapped I/O

Const: New ANSI standard type for fixed data

Struct: Groups of variables under a single name

Union: Multi-purpose storage areas for dynamical memory allocation

SPECIAL CONSTANT EXPRESSIONS

Constant expressions are often used without any thought, until a programmer needs to know how to do something special with them. Up to now the distinction between long and short integer types has largely been ignored. Constant values can be declared explicitly as long values, in fact, by placing the letter L after the constant.

```
long int variable = 23L;
variable = 236526598L;
```

Advanced programmers, writing systems software, often find it convenient to work with hexadecimal or octal numbers since these number bases have a special relationship to binary. A constant in one of these types is declared by placing either 0 (zero) or 0x in front of the appropriate value. If *ddd* is a value, then:

```
Octal number 0ddd
Hexadecimal number 0xddd
```

For example:

```
oct_value = 077;/* 77 octal */
hex_value = 0xFFEF;/* FFEF hex */
```

This kind of notation has already been applied to strings and single character constants with the backslash notation, instead of the leading zero character:

```
ch = '\ddd';
ch = '\xdd';
```

The values of character constants, like these, cannot be any greater than 255.

File: In all previous sections, the files stdin, stdout and stderr alone have been used in programs. These special files are always handled implicitly by functions likeprintf() and scanf(): the programmer never gets to know that they are, in fact, files. Programs do not have to use these functions however: standard input/output files can be treated explicitly by general file handling functions just as well. Files are distinguished by filenames and by file pointers. File pointers are variables which pass the location of files to file handling functions; being variables, they have to be declared as being some data type. That type is called FILE and file pointers have to be declared "pointer to FILE".

For example:

```
FILE *fp;
FILE *fp = stdin;
FILE *fopen();
```

File handling functions which return file pointers must also be declared as pointers to files. Notice that, in contrast to all the other reserved words FILE is written in upper case: the reason for this is that FILE is not a simple data type such as char or int, but a *structure* which is only defined by the header file stdio.h and so, strictly speaking, it is not a reserved word itself. We shall return to look more closely at files soon.

Enum: Abstract data are usually the realm of exclusively high level languages such as Pascal. enum is a way of incorporating limited "high level" data facilities into C.

Enum is short for enumerated data. The user defines a type of data which is made up of a fixed set of words, instead of numbers or characters. These words are given substitute integer numbers by the compiler which are used to identify and compare enum type data. For example:

```
enum countries
{
England,
Scotland,
Wales,
Eire,
Norge,
Sverige,
Danmark,
Deutschland
};
main ()
{ enum countries variable;
variable = England;
}
```

Example:

```
/* Enumerated Data */
#include <stdio.h>
   enum countries
{
   England,
   Ireland,
   Scotland,
   Wales,
   Danmark,
   Island,
   Norge,
   Sverige
   };
main () /* Electronic Passport Programme */
{ enum countries birthplace, getinfo();
printf ("Insert electronic passport\n");
birthplace = getinfo();
switch (birthplace)
   {
   case England: printf ("Welcome home!\n");
   break;
   case Danmark:
   case Norge:  printf ("Velkommen til
        England\n");
                break;
   }
}
enum countries getinfo() /* interrogate
              passport */
{
return (England);
}
        /* end */
```

enum makes words into constant integer values for a programmer. Data which are declared enum are not the kind of data which it makes sense to do arithmetic with (even integer arithmetic), so in most cases it should not be necessary to know or even care about what numbers the compiler gives to the words in the list. However, some compilers allow the programmer to force particular values on words. The compiler then tries to give the values successive integer numbers unless the programmer states otherwise.

For instance:

```
enum planets
{
Mercury,
Venus,
Earth = 12,
Mars,
Jupiter,
Saturn,
Uranus,
Neptune,
Pluto
};
```

This would probably yield values Mercury = 0, Venus = 1, Earth = 12, Mars = 13, Jupiter = 14 . etc. If the user tries to force a value which the compiler has already used then the compiler will complain.

The following example programme listing shows two points:

- Enum types can be local or global.
- The labels can be forced to have certain values

MACHINE LEVEL OPERATIONS

BITS AND BYTES

Down in the depths of your computer, below even the operating system are bits of memory. These days we are used to working at such a high level that it is easy to forget them. Bits (or binary digits) are the lowest level software objects in a computer: there is nothing more primitive. For precisely this reason, it is rare for high level languages to even acknowledge the existence of bits, let alone manipulate them. Manipulating bit patterns is usually the preserve of assembly language programmers. C, however, is quite different from most other high level languages in that it allows a programmer full access to bits and even provides high level operators for manipulating them.

BIT PATTERNS

- All computer data, of any type, are bit patterns. The only difference between a string and a floating point variable is the way in which we choose to interpret the patterns of bits in a computer's memory. For the most part, it is quite unnecessary to think of computer data as bit patterns; systems programmers, on the other hand, frequently find that they need to handle bits directly in order to make efficient use of memory when using flags. A flag is a message which is either one thing or the other: in system terms, the flag is said to be 'on' or 'off' or alternatively *set* or *cleared*. The usual place to find flags is in a status register of a CPU (central processor unit) or in a pseudo-register (this is a status register for an imaginary processor, which is held in memory). A status register is a group of bits (a byte perhaps) in which each bit signifies something special. In an ordinary byte of data, bits are grouped together and are interpreted to have a collective meaning; in a status register they are thought of as being independent. Programmers are interested to know about the contents of bits in these registers, perhaps to find out what happened in a programme after some special operation is carried out.

FLAGS, REGISTERS AND MESSAGES

A *register* is a place inside a computer processor chip, where data are worked upon in some way. A *status register* is a register which is used to return information to a programmer about the operations which took place in other registers.

Status registers contain flags which give yes or no answers to questions concerning the other registers. In advanced programming, there may be call for “pseudo registers” in addition to “real” ones. A pseudo register is merely a register which is created by the programmer in computer memory (it does not exist inside a processor). Messages are just like pseudo status registers: they are collections of flags which signal special information between different devices and/or different programs in a computer system.

Messages do not necessarily have fixed locations: they may be passed a parameters. Messages are a very compact way of passing information to low level functions in a programme. Flags, registers, pseudo-registers and messages are all treated as bit patterns.

A programme which makes use of them must therefore be able to assign these objects to C variables for use. A bit pattern would normally be declared as a character or some kind of integer type in C, perhaps with the aid of a typedef statement.

```
typedef char byte;
typedef int bitpattern;
```

```
bitpattern variable;
byte message;
```

The flags or bits in a register/message. have the values 1 or 0, depending upon whether they are on or off (set or cleared). A programme can test for this by using combinations of the operators which C provides.

BIT OPERATORS AND ASSIGNMENTS

C provides the following operators for handling bit patterns:

<< Bit shift left (a specified number or bit positions)

>> Bit shift right(a specified number of bit positions)

| Bitwise Inclusive OR

^ Bitwise Exclusive OR

& Bitwise AND

~ Bitwise one's complement

&= And assign (variable = variable & value)

|= Exclusive OR assign (variable = variable | value)

^= Inclusive OR assign (variable = variable ^ value)

>>= Shift right assign (variable = variable >> value)

<<= Shift left assign (variable = variable << value)

BIT OPERATORS

Bitwise operations are not to be confused with logical operations (&&, ||.) A bit pattern is made up of 0s and 1s and bitwise operators operate individually upon each bit in the operand. Every 0 or 1 undergoes the operations individually. Bitwise operators (AND, OR) can be used in place of logical operators (&&,||), but they are less efficient, because logical operators are designed to reduce the number of comparisons made, in an expression, to the optimum: as soon as the truth or falsity of an expression is known, a logical comparison operator quits. A bitwise operator would continue operating to the last before the final result were known.

Below is a brief summary of the operations which are performed by the above operators on the bits of their operands.

SHIFT OPERATIONS

Imagine a bit pattern as being represented by the following group of boxes. Every box represents a bit; the numbers inside represent their values. The values written over the top are the common integer values which the whole group of bits would have, if they were interpreted collectively as an integer.

```
128    64    32   16  8    4    2  1
________________________________________
| 0  | 0  | 0  | 0  | 0  | 0  | 0  | 1  |    = 1
________________________________________
```

Shift operators move whole bit patterns left or right by shunting them between boxes.

The syntax of this operation is:

```
value << number of positions
value >> number of positions
```

So for example, using the boxed value (1) above:

```
1  <<  1
```

would have the value 2, because the bit pattern would have been moved one place the the left:

```
128    64    32    16    8    4    2    1
_______________________________________
| 0  | 0  | 0  | 0  | 0  | 0  | 1  | 0  | = 2
_______________________________________
```

Similarly:

```
1  <<  4
```

has the value 16 because the original bit pattern is moved by four places:

```
128    64   32   16    8    4    2   1
_________________________________________
| 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | = 16
_________________________________________
```

And:

```
6 << 2 == 12
128    64   32   16    8    4    2   1
_________________________________________
| 0 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | = 6
_________________________________________
```

Shift left 2 places:

```
128    64   32   16    8    4    2   1
_________________________________________
| 0 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | = 12
_________________________________________
```

Notice that every shift left multiplies by 2 and that every shift right would divide by two, integerwise. If a bit reaches the edge of the group of boxes then it falls out and is lost forever.

So:

```
1 >> 1 == 0
2 >> 1 == 1
2 >> 2 == 0
n >> n == 0
```

A common use of shifting is to scan through the bits of a bitpattern one by one in a loop: this is done by using *masks*.

TRUTH TABLES AND MASKING

The operations AND, OR (inclusive OR) and XOR/EOR (exclusive OR) perform comparisons or “masking” operations between two bits. They are binary or dyadic operators. Another operation called COMPLEMENT is a unary operator. The operations performed by these bitwise operators are best summarized by*truth tables*. Truth tables indicate what the results of all possible operations are between two single bits. The same operation is then carried out for all the bits in the variables which are operated upon.

Complement ~

The complement of a number is the *logical opposite* of the number. C provides a "one's complement" operator which simply changes all 1s into 0s and all 0s into 1s.

~1 has the value 0(for each bit)

~0 has the value 1

As a truth table this would be summarized as follows:

~value	==	*result*
0		1
1		0

AND &

This works between two values. e.g. (1 & 0)

value 1	&	*value 2*	==	*result*
0		0		0
0		1		0
1		0		0
1		1		1

Both value 1 AND value 2 have to be 1 in order for the result or be 1.

OR |

This works between two values. e.g. (1 | 0)

value 1	|	*value 2*	==	*result*
0		0		0
0		1		1
1		0		1
1		1		1

The result is 1 if one OR the other OR both of the values is 1.

XOR/EOR ^

Operates on two values. e.g. (1 ^ 0)

value 1	^	*value 2*	==	*result*
0		0		0
0		1		1
1		0		1
1		1		0

The result is 1 if one OR the other (but not both) of the values is 1.

Bit patterns and logic operators are often used to make *masks*. A mask is as a thing which fits over a bit pattern and modifies the result in order perhaps to single out particular bits, usually to cover up part of a bit pattern.

This is particularly pertinent for handling flags, where a programmer wishes to know if one particular flag is set or not set and does not care about the values of the others.

This is done by deliberately inventing a value which only allows the particular flag of interest to have a non-zero value and then ANDing that value with the flag register. For example: in symbolic language:

```
MASK = 00000001
VALUE1 = 10011011
VALUE2 = 10011100
MASK & VALUE1 == 00000001
MASK & VALUE2 == 00000000
```

The zeros in the mask *masks off* the first seven bits and leave only the last one to reveal its true value. Alternatively, masks can be built up by specifying several flags:

```
FLAG1 = 00000001
FLAG2 = 00000010
FLAG3 = 00000100
MESSAGE = FLAG1 | FLAG2 | FLAG3
MESSAGE == 00000111
```

It should be emphasized that these expressions are only written in symbolic language: it is not possible to use binary values in C. The programmer must convert to hexadecimal, octal or denary first.

Example

A simple example helps to show how logical masks and shift operations can be combined. The first programme gets a denary number from the user and converts it into binary. The second programme gets a value from the user in binary and converts it into hexadecimal.

```
/* Bit Manipulation #1 */
   /* Convert denary numbers into binary */
   /* Keep shifting i by one to the left */
   /* and test the highest bit. This does*/
   /* NOT preserve the value of i */
#include <stdio.h>
#define NUMBEROFBITS 8
main ()
{  short i,j,bit,;
   short MASK = 0x80;
printf (“Enter any number less than 128: “);
scanf (“%h”, &i);
if (i > 128)
   {
   printf (“Too big\n”);
   return (0);
   }
printf (“Binary value = “);
for (j = 0; j < NUMBEROFBITS; j++)
   {
   bit = i & MASK;
   printf (“%1d”,bit/MASK);
   i <<= 1;
   }
printf (“\n”);
}
        /* end */
Output:
Enter any number less than 128: 56
Binary value = 00111000
Enter any value less than 128: 3
Binary value = 00000011
```

Example:

```
/* Bit Manipulation #2 */
   /* Convert binary numbers into hex */
#include <stdio.h>
#define NUMBEROFBITS 8
main ()
{  short j,hex = 0;
   short MASK;
```

```
    char binary[NUMBEROFBITS];
printf ("Enter an 8-bit binary number: ");
for (j = 0; j < NUMBEROFBITS; j++)
    {
    binary[j] = getchar();
    }
for (j = 0; j < NUMBEROFBITS; j++)
    {
    hex <<= 1;
    switch (binary[j])
        {
        case '1': MASK = 1;
        break;
        case '0': MASK = 0;
        break;
        default:printf("Not binary\n");
                return(0);
        }
    hex |= MASK;
    }
printf ("Hex value = %1x\n",hex);
}
                /* end */
```

Example:

```
Enter any number less than 128: 56
Binary value = 00111000
Enter any value less than 128: 3
Binary value = 00000011
```

FILES AND DEVICES

Files are places for reading data from or writing data to. This includes disk files and it includes devices such as the printer or the monitor of a computer. C treats all information which enters or leaves a programme as though it were a stream of bytes: a file. The most commonly used file streams are stdin (the keyboard) andstdout (the screen), but more sophisticated programs need to be able to read or write to files which are found on a disk or to the printer etc. An operating system allows a programme to see files in the outside world by providing a number of channels or ‘portals’ (‘inlets’ and ‘outlets’) to work through. In order to examine the contents of a file or to write information to a file, a programme has to *open* one of these portals. The reason for this slightly indirect method of working is that channels/ portals hide operating system dependent details of filing from the programmer. Think of it as a protocol. A programme which writes information does no more than pass that information to one of these portals and the operating system’s filing subsystem does the rest. A programme which reads data simply reads values from its file portal and does not have to worry about how they got there. This is extremely simple to work in practice.

To use a file then, a programme has to go through the following routine:

- Open a file for reading or writing.
- Read or write to the file using file handling functions provided by the standard library.
- Close the file to free the operating system “portal” for use by another programme or file.

A programme opens a file by calling a standard library function and is returned a file pointer, by the operating system, which allows a programme to address that particular file and to distinguish it from all others.

FILES GENERALLY

C provides two levels of file handling; these can be called high level and low level. High level files are all treated as text files. In fact, the data which go into the files are exactly what would be seen on the screen, character by character, except that they are stored in a file instead. This is true whether a file is meant to store characters, integers, floating point types. Any file, which is written to by high level file handling functions, ends up as a text file which could be edited by a text editor.

High level text files are also read back as character files, in the same way that input is acquired from the keyboard. This all means that high level file functions are identical in concept to keyboard/screen input/output. The alternative to these high level functions, is obviously low level functions. These are more efficient, in principle, at filing data as they can store data in large lumps, in raw memory format, without converting to text files first. Low level input/output functions have the disadvantage that they are less 'programmer friendly' than the high level ones, but they are likely to work faster.

FILE POSITIONS

When data are read from a file, the operating system keeps track of the current position of a programme within that file so that it only needs to make a standard library call to 'read the next part of the file' and the operating system obliges by reading some more and advancing its position within the file, until it reaches the end. Each single character which is read causes the position in a file to be advanced by one.

Although the operating system does a great deal of hand holding regarding file positions, a programme can control the way in which that position changes with functions such as ungetc() if need be. In most cases it is not necessary and it should be avoided, since complex movements within a file can cause complex movements of a disk drive mechanism which in turn can lead to wear on disks and the occurrence of errors.

HIGH LEVEL FILE HANDLING FUNCTIONS

Most of the high level input/output functions which deal with files are easily recognizable in that they start with the letter 'f'. Some of these functions will appear strikingly familiar.

For instance:

```
fprintf()
fscanf()
fgets()
fputs()
```

These are all generalized file handling versions of the standard input/output library. They work with generalized files, as opposed to the specific files stdin and stdout which printf() and scanf() use.

The file versions differ only in that they need an extra piece of information: the file pointer to a particular portal. This is passed as an extra parameter to the functions. they process data in an identical way to their standard I/O counterparts. Other filing functions will not look so familiar.

For example:

```
fopen()
fclose()
getc()
ungetc();
putc()
fgetc()
fputc()
feof()
```

OPENING FILES

A file is opened by a call to the library function fopen(): this is available automatically when the library file <stdio.h> is included. There are two stages to opening a file: firstly a file portal must be found so that a programme can access information from a file at all. Secondly the file must be physically located on a disk or as a device or whatever. The fopen() function performs both of these services and, if, in fact, the file it attempts to open does not exist, that file is created anew. The syntax of the fopen() function is:

```
FILE *returnpointer;
returnpointer = fopen("filename","mode");
```

or

```
FILE returnpointer;
char *fname, *mode;
returnpointer = fopen(fname,mode);
```

The filename is a string which provides the name of the file to be opened. Filenames are system dependent so the details of this must be sought from the local operating system manual. The operation mode is also a string, chosen from one of the following:

r Open file for reading

w Open file for writing

a Open file for appending

rw Open file for reading and writing (some systems)

This mode string specifies the way in which the file will be used. Finally, returnpointer is a pointer to a FILE structure which is the whole object of calling this function. If the file (which was named) opened successfully when fopen() was called, returnpointer is a pointer to the file portal. If the file could not be opened, this pointer is set to the value NULL. This should be tested for, because it would not make sense to attempt to write to a file which could not be opened or created, for whatever reason.

A read only file is opened, for example, with some programme code such as:

```
FILE *fp;
if ((fp = fopen ("filename","r")) == NULL)
   {
   printf ("File could not be opened\n");
   error_handler();
   }
```

A question which springs to mind is: what happens if the user has to type in the name of a file while the programme is running? The solution to this problem is quite simple. Recall the function filename() which was written in chapter 20.

```
char *filename()    /* return filename */
{ static char *filenm = "..";
do
   {
   printf ("Enter filename:");
   scanf ("%24s",filenm);
   skipgarb();
   }
while (strlen(filenm) == 0);
return (filenm);
}
```

This function makes file opening simple. The programmer would now write something like:

```
FILE *fp;
char *filename();
if ((fp = fopen (filename(),"r")) == NULL)
                    {
printf ("File could not be opened\n");
    error_handler();
                    }
```

and then the user of the programme would automatically be prompted for a filename. Once a file has been opened, it can be read from or written to using the other library functions (such as fprintf() and fscanf()) and then finally the file has to be closed again.

CLOSING A FILE

A file is closed by calling the function fclose(). fclose() has the syntax:

```
int returncode;
FILE *fp;
returncode = fclose (fp);
```

fp is a pointer to the file which is to be closed and returncode is an integer value which is 0 if the file was closed successfully. fclose() prompts the file manager to finish off its dealings with the named file and to close the portal which the operating system reserved for it. When closing a file, a programme needs to do something like the following:

```
if (fclose(fp) != 0)
   {
   printf ("File did not exist.\n");
   error_handler();
   }
```

fprintf()

This is the highest level function which writes to files. Its name is meant to signify "file-print-formatted" and it is almost identical to its stdout counterpart printf(). The form of the fprintf() statement is as follows:

```
fprintf (fp,"string",variables);
```

where fp is a file pointer, string is a control string which is to be formatted and the variables are those which are to be substituted into the blank fields of the format string. For example, assume that there is an open file, pointed to by fp:

```
int i = 12;
float x = 2.356;
char ch = 's';
fprintf (fp, "%d %f %c", i, x, ch);
```

The conversion specifiers are identical to those for printf(). In fact fprintf() is related to printf() in a very simple way: the following two statements are identical.

```
printf ("Hello world %d", 1);
fprintf (stdout,"Hello world %d", 1);
```

fscanf()

The analogue of scanf() is fscanf() and, as with fprintf(), this function differs from its standard I/O counterpart only in one extra parameter: a file pointer. The form of an fscanf() statement is:

```
FILE *fp;
int n;
n = fscanf (fp,"string",pointers);
```

where n is the number of items matched in the control string and fp is a pointer to the file which is to be read from. For example, assuming that fp is a pointer to an open file:

```
int i = 10;
float x = -2.356;
char ch = 'x';
fscanf (fp, "%d %f %c", &i, &x, &ch);
```

The remarks which were made about scanf() also apply to this function: fscanf() is a 'dangerous' function in that it can easily get out of step with the input data unless the input is properly formatted.

For comparison alone, skipfilegarb() is written below.

```
skipfilegarb(fp)
FILE *fp;
{
while (getc(fp) != '\n')
{
}
}
```

SINGLE CHARACTER I/O

There are commonly four functions/macros which perform single character input/output to or from files. They are analogous to the functions/macros

```
getchar()
putchar()
```

for the standard I/O files and they are called:

```
getc()
ungetc();
putc()
fgetc()
fputc()
getc() and fgetc()
```

The difference between getc() and fgetc() will depend upon a particular system. It might be that getc() is implemented as a macro, whereas fgetc() is implemented as a function or vice versa. One of these alternatives may not be present at all in a library. Check the manual, to be sure! Both getc() and fgetc()fetch a single character from a file:

```
FILE *fp;
char ch;
/* open file */
ch = getc (fp);
ch = fgetc (fp);
```

These functions return a character from the specified file if they operated successfully, otherwise they return EOF to indicate the end of a file or some other error. Apart from this, these functions/macros are quite unremarkable.

```
ungetc()
```

ungetc() is a function which 'un-gets' a character from a file. That is, it reverses the effect of the last get operation. This is not like writing to a file, but it is like stepping back one position within the file. The purpose of this function is to leave the input in the correct place for other functions in a programme when other functions go too far in a file. An example of this would be a programme which looks for a word in a text file and processes that word in some way.

```
while (getc(fp) != '')
   {
   }
```

The programme would skip over spaces until it found a character and then it would know that this was the start of a word. However, having used getc() to read the first character of that word, the position in the file would be the second character in the word! This means that, if another function wanted to read that word from the beginning, the position in the file would not be correct, because the first character would already have been read. The solution is to use ungetc() to move the file position back a character:

```
int returncode;
returncode = ungetc(fp);
```

The returncode is EOF if the operation was unsuccessful.

```
putc() and fputc()
```

These two functions write a single character to the output file, pointed to by fp. As with getc(), one of these may be a macro. The form of these statements is:

```
FILE *fp;
char ch;
int returncode;
returncode = fputc (ch,fp);
returncode = putc (ch,fp);
```

The returncode is the ascii code of the character sent, if the operation was successful, otherwise it is EOF.

```
fgets() and fputs()
```

Just as gets() and puts() fetched and sent strings to standard input/output files stdin and stdout, so fgets() and fputs() send strings to generalized files. The form of an fgets() statement is as follows:

```
char *strbuff,*returnval;
int n;
FILE *fp;
returnval = fgets (strbuff,n,fp);
```

strbuff is a pointer to an input buffer for the string; fp is a pointer to an open file. returnval is a pointer to a string: if there was an error in fgets() this pointer is set to the value NULL, otherwise it is set to the value of "strbuff". No more than (n-1) characters are read by fgets() so the programmer has to be sure to set n equal to the size of the string buffer. (One byte is reserved for the NULL terminator.) The form of an fputs() statement is as follows:

```
char *str;
int returnval;
FILE *fp;
returnval = fputs (str,fp);
```

Where str is the NULL terminated string which is to be sent to the file pointed to by fp. returnval is set to EOF if there was an error in writing to the file.

PRINTER OUTPUT

Any serious application programme will have to be in full control of the output of a programme. For instance, it may need to redirect output to the printer so that data can be made into hard copies. To do this, one of three things must be undertaken:

- Stdout must be redirected so that it sends data to the printer device.
- A new "standard file" must be used (not all C compilers use this method.)
- A new file must be opened in order to write to the printer device

The first method is not generally satisfactory for applications programs, because the standard files stdin and stdout can only easily be redirected from the operating system command line interpreter (when a programme is run by typing its name). Examples of this are:

```
type file > PRN
```

which send a text file to the printer device. The second method is reserved for only a few implementations of C in which another 'standard file' is opened by the local operating system and is available for sending data to the printer stream.

This file might be called "stdprn" or "standard printer file" and data could be written to the printer by switching writing to the file like this:

```
fprintf (stdprn,"string %d.", integer);
```

The final method of writing to the printer is to open a file to the printer, personally. To do this, a programme has to give the "filename" of the printer device.

This could be something like "PRT:" or "PRN" or "LPRT" or whatever. The filename (actually called a pseudo device name) is used to open a file in precisely the same way as any other file is opened: by using a call to fopen(). fopen() then returns a pointer to file (which is effectively "stdprn") and this is used to write data to a computer's printer driver. The programme code to do this should look something like the following:

```
FILE *stdprn;
if ((stdprn = fopen("PRT:","w")) == NULL)
   {
   printf ("Printer busy or disconnected\n");
   error_handler;
   }
```

LOW LEVEL FILING OPERATIONS

Normally a programmer can get away with using the high level input/output functions, but there may be times when C's predilection for handling all high level input/output as text files, becomes a nuisance. A programme can then use a set of low level I/O functions which are provided by the standard library.

These are:

```
open()
close()
creat()
read()
write()
rename()
unlink()/remove()
lseek()
```

These low level routines work on the operating system's end of the file portals. They should be regarded as being advanced features of the language because they are dangerous routines for bug ridden programs.

The data which they deal with is untranslated: that is, no conversion from characters to floating point or integers or any type at all take place. Data are treated as a raw stream of bytes.

Low level functions should not be used on any file at the same time as high level routines, since high level file handling functions often make calls to the low level functions.

Working at the low level, programs can create, delete and rename files but they are restricted to the reading and writing of untranslated data: there are no functions such as fprintf() or fscanf() which make type conversions.

As well as the functions listed above a local operating system will doubtless provide special function calls which enable a programmer to make the most of the facilities offered by the particular operating environment. These will be documented, either in a compiler manual, or in an operating system manual, depending upon the system concerned.

FILE DESCRIPTORS

At the low level, files are not handled using file pointers, but with integers known as file handles or file descriptors. A file handle is essentially the number of a particular file portal in an array. In other words, for all the different terminology, they describe the same thing. For example:

int fd;

Would declare a file *handle* or *descriptor* or *portal* or whatever it is to be called.

open()

Open() is the low level file open function. The form of this function call is:

```
    int fd, mode;
    char *filename;
    fd = open (filename,mode);
```

where filename is a string which holds the name of the file concerned, mode is a value which specifies what the file is to be opened for and fd is either a number used to distinguish the file from others, or -1 if an error occurred.

A programme can give more information to this function than it can to fopen() in order to define exactly what open() will do. The integer *mode* is a message or a pseudo register which passes the necessary information to open(), by using the following flags:

```
O_RDONLY    Read access only
O_WRONLY    Write access only
O_RDWR      Read/Write access
```

and on some compilers:

```
O_CREAT Create the file if it does not exist
O_TRUNC Truncate the file if it does exist
O_APPEND Find the end of the file before each
   write
O_EXCL  Exclude. Force create to fail if the file exists.
```

The macro definitions of these flags will be included in a library file: find out which one and #include it in the programme. The normal procedure is to open a file using one of the first three modes.

For example:

```
#define FAILED -1
main()
{  char *filename();
   int fd;
fd = open(filename(), O_RDONLY);
if (fd == FAILED)
   {
   printf (“File not found\n”);
   error_handler (failed);
   }
}
```

This opens up a read-only file for low level handling, with error checking. Some systems allow a more flexible way of opening files. The four appended modes are values which can be bitwise ORed with one of the first three in order to get more mileage out of open(). The bitwise OR operator is the vertical bar “|”. For example, to emulate the fopen() function a programme could opt to create a file if it did not already exist:

```
fd = open (filename(), O_RDONLY | O_CREAT);
```

open() sets the file position to zero if the file is opened successfully.

Close()

close() releases a file portal for use by other files and brings a file completely up to date with regard to any changes that have been made to it. Like all other filing functions, it returns the value 0 if it performs successfully and the value -1 if it fails. e.g.

```
#define FAILED -1
if (close(fd) == FAILED)
   {
   printf ("ERROR!");
   }
```

Creat()

This function creates a new file and prepares it for access using the low level file handling functions. If a file which already exists is created, its contents are discarded. The form of this function call is:

```
int fd, pmode;
char *filename;
fd = creat(filename,pmode);
```

filename must be a valid filename; pmode is a flag which contains access-privilege mode bits (system specific information about allowed access) and fd is a returned file handle. In the absence of any information about pmode, this parameter can be set to zero. Note that, the action of creating a file opens it too. Thus after a call to creat, you should close the file descriptor.

Read()

This function gets a block of information from a file. The data are loaded directly into memory, as a sequence of bytes. The user must provide a place for them (either by making an array or by using malloc() to reserve space). read() keeps track of file positions automatically, so it actually reads the next block of bytes from the current file position. The following example reads n bytes from a file:

```
int returnvalue, fd, n;
char *buffer;
if ((buffer = malloc(size)) == NULL)
   {
   puts ("Out of memory\n");
   error_handler ();
   }
returnvalue = read (fd,buffer,n);
```

The return value should be checked. Its values are defined as follows:

End of file,

-1 Error occurred

*n*the number of bytes actually read. (If all went well this should be equal to *n*.)

Write()

This function is the opposite of read(). It writes a block of *n* bytes from a contiguous portion of memory to a file which was opened by open(). The form of this function is:

```
int returnvalue, fd, n;
char *buffer;
returnvalue = write (fd,buffer,n);
```

The return value should, again, be checked for errors:

-1   Error

*n*Number of bytes written

Lseek()

Low level file handing functions have their equivalent of fseek() for finding a specific position within a file. This is almost identical to fseek() except that it uses the file handle rather than a file pointer as a parameter and has a different return value. The constants should be declared long int, or simply long.

```
#define FAILED -1L
long int pos,offset,fd;
int mode,returncode;
if ((pos = fseek (fd,offset,mode)) == FAILED)
   {
   printf("Error!\n");
   }
```

pos gives the new file position if successful, and -1 (long) if an attempt was made to read past the end of the file. The values which mode can take are:

0Offset measured relative to the beginning of the file.

1Offset measured relative to the current position.

2Offset measured relative to the end of the file.

Unlink() and Remove()

These functions delete a file from disk storage. Once deleted, files are usually irretrievable. They return -1 if the action failed.

```
#define FAILED -1
int returnvalue;
char *filename;
if (unlink (filename) == FAILED)
   {
   printf ("Can't delete %s\n",filename);
   }
if (remove (filename) == FAILED)
   {
   printf ("Can't delete %s\n",filename);
   }
```

filename is a string containing the name of the file concerned. This function can fail if a file concerned is protected or if it is not found or if it is a device. (It is impossible to delete the printer!)

Rename()

This function renames a file. The programmer specifies two filenames: the old filename and a new file name. As usual, it returns the value -1 if the action fails. An example illustrates the form of the rename() call:

```
#define FAILED -1
char *old,*new;
if (rename(old,new) == FAILED)
   {
   printf ("Can't rename %s as %s\n",old,new);
   }
```

rename() can fail because a file is protected or because it is in use, or because one of the filenames given was not valid.

6

Augmenting Data Structures

There are some Programmeming situations that can be perfectly solved with standard data structures such as a linked lists, hash tables, or binary search trees. Many others require a dash of creativity. Only in rare situations will you need to create an entirely new type of data structure, though.

More often, it will suffice to augment (to modify) an existing data structure by storing additional information in it. You can then Programme new operations for the data structure to support the desired application. Augmenting a data structure is not always straightforward, however, since the added information must be updated and maintained by the ordinary operations on the data structure.

This lecture discusses two data structures that are constructed by augmenting red-black trees (see the previous post on red-black trees).

The first part of the lecture describes a data structure that supports general order-statistic operations on a dynamic set. It's called dynamic order statistics. The notion of order statistics was introduced in lecture six. In lecture six it was shown that any order statistic could be retrieved in O(n) time from an unordered set. In this lecture it is shown how red-black trees can be modified so that any order statistic can be determined in O(lg(n)) time. It presents two algorithms OS-Select(i), which returns i-th smallest item in a dynamic set, and OS-Rank(x), which returns rank (position) of element x in sorted order.

The lecture continues with general methodology of how to augment a data structure.

Augmenting a data structure can be broken into four steps:

- Choosing an underlying data structure.
- Determining additional information to be maintained in the underlying data structure.
- Verifying that the additional information can be maintained for the basic modifying operations (insert, delete, rotate, etc.) on the underlying data structure.
- Developing new operations.

The second part of the lecture applies this methodology to construct a data structure called interval trees. This data structure maintains a dynamic set of elements, with each element x containing an interval. Interval is simply pair of numbers (low, high). For example, a time interval from 3 o'clock to 7 o'clock is a pair (3, 7).

Lecture gives an algorithm called Interval-Search(x), which given a query interval x, quickly finds an interval in the set that overlaps it. Time complexity of this algorithm is O(lg(n)).

ABOUT LUCTURE

The lecture is motivated by two things:

- The implementation of ADTs that extend standard ADTs by one or more additional operations;
- Application in which data "live in" various data structures simultaneously.

Augmentation is a process by which one adds fields to the nodes as necessary.

Augmenting a data structure can be broken into four steps:

- choosing an underlying data structure
- determining additional information to be maintained in the underlying data structure
- verifying that the additional information can be efficiently maintained for the basic modifying operations on the underlying data structure
- developing new operations

DYNAMIC ORDER-STATISTIC TREES

ADT: set S; Dictionary operations (*Insert, Delete, Search*) + two additional operations:

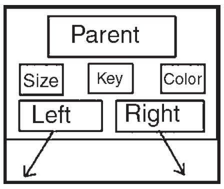
- *Rank(S,x)*: Returns rank of node x (smallest rank is 1)
- *Select(S, i)*: Select ith smallest element from set S

Implementation

An *order-statistic tree* T is simply a red-black tree with additional information stored at each node. Besides the usual red-black tree fields *key*[x], *left*[x], *right*[x],*p*[x] and *Colour*[x] in a node *x*, we have another field *size*[*x*]. This field contains the number of internal nodes in the subtree rooted at *x* (including x itself), that is, the size of the subtree.

$$size[x] = size[left[x]] + size[right[x]] + 1$$

The node of the *order-statistic tree* will now look as follows: figure



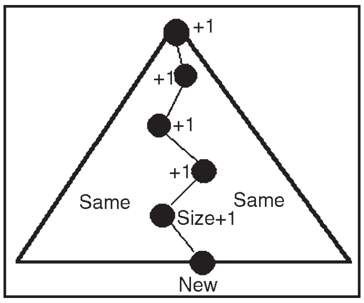
There are couple of things though that we have to worry about:

1. Can *size* be updated in O($\log_2 n$) time per operation?
2. *Rank*(x) and *select*(S, i) to be done in O($\log_2 n$) time.

Operations on Order-statistic trees

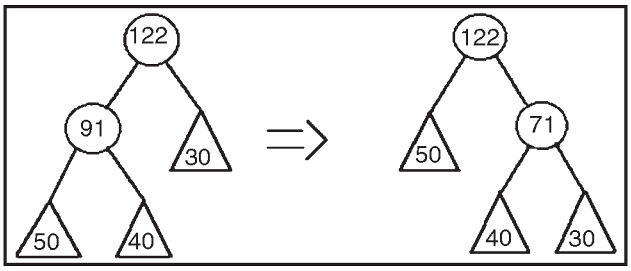
Insert

As we know, insertion into a red-black tree consists of two phases. Phase 1 goes down the path from the root inserting the new node as a child of the existing node. To maintain the subtree sizes we simply increment *size*[*x*] for each node *x* on the path traversed from the root down toward the leaves. The new node added gets size 1. Figure illustrates Phase 1.



In the Phase 2, the only structural changes to the underlying red-black tree are caused by *rotations*, of which there are at most two.

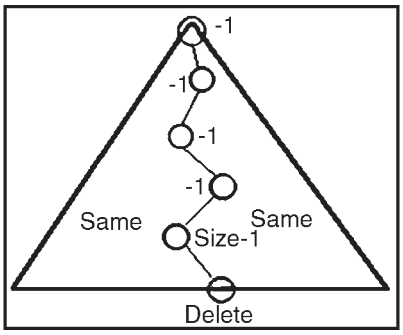
Rotation is a local operation and it invalidates only the two *size* fields in the nodes incident on the link around which the rotation is performed. Figure shows *Left* and *Right*rotations.



Delete

Phase 1 splices out the node we wish to delete. To update subtree sizes we simply traverse a path from node we wish to delete up to the root, decrementing*size* field for each node on the path.

Figure illustrates this process.



The rotations in the Phase 2 are handled in the same manner as for insertion.

Running time: To maintain tree sizes in the Phase 1 of insertion or deletion we have to increment or decrement *size*[*x*] for each node *x* on the path from the root down toward the leaves. Since there are O($\log_2 n$) nodes on the traversed path, the additional cost of maintaining the *size* fields is O($\log_2 n$). Moreover, rotation is a local operation and hence only O(1) additional time is spent updating *size* fields in the Phase 2. Thus, both insertion and deletion take $O(\log_2 n)$ time.

Select

The procedure *Select*(*x*, *i*) returns a pointer to the node containing the ith smallest key in the subtree rooted at *x*.

```
Select(x, i)
r (rank of root) = size[left[x]] + 1
case r = i: return x
case r > i: return Select(left[x], i)
case r < i: return Select(right[x], i)
```

Because each recursive call goes down one level in the order-statistic tree, the total time for *Select* is at worst proportional to the height of the tree. Since the tree is the red-black tree, its height is $O(\log_2 n)$. Thus, the running time of *Select* is $O(\log_2 n)$.

Rank

The procedure *Rank*(*T*, *x*) returns the position of *x* in the linear order determined by an inorder tree walk of *T*. Let *x* is a pointer to the node in the tree.

```
Rank(T, x)
r = size[left[x]] + 1
y = x
while y <> root do
if right[parent[y]] = y
then r += size[left[parent[y]]] + 1
y = parent[y]
return r
```

Since each iteration of the while loop takes O(1) time and *y* goes up one level in the tree with each iteration, the running time of *Rank* is at worst proportional to the height of the tree: $O(\log_2 n)$.

Figure illustrates the *Rank* procedure.

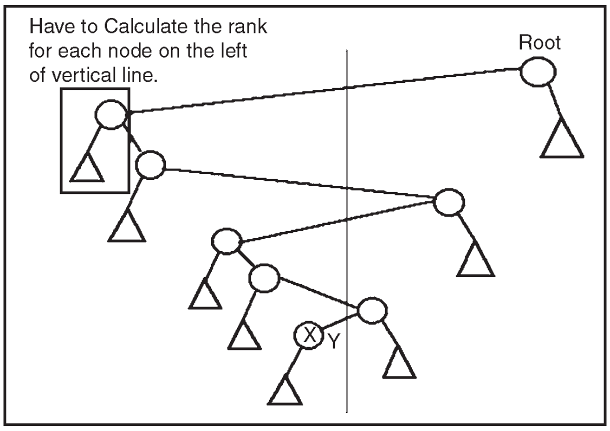


Fig. Link to our Java Applet

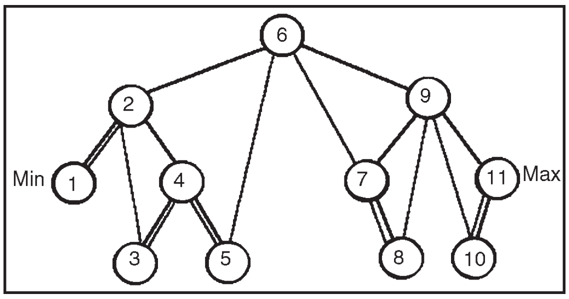
BINARY SEARCH TREES FOR BROWSING ADT

Dictionary + Browsing operations:

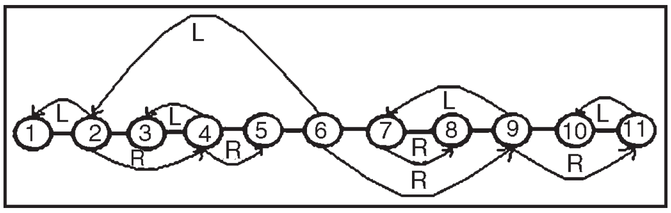
- MIN
- MAX
- Predecessor
- Successor

Implementation

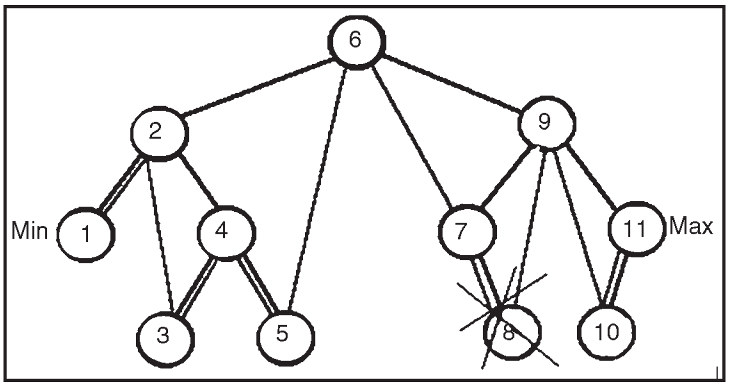
The underlying data structure is a red-black tree where besides usual red-black tree fields each node is augmented with *Min*[*x*], *Max*[*x*], *Predecessor*[*x*] and*Successor*[*x*]. Figure 6 shows the augmented Red-Black tree on the input {1 2 3 4 5 6 7 8 9 10 11}.



The data structure in Figure 6 can be looked at as a binary search tree or a sorted linked list as shown in Figure 7. In fact, according to Prof. Devroye, it is a smooth "marriage" of both data structures.

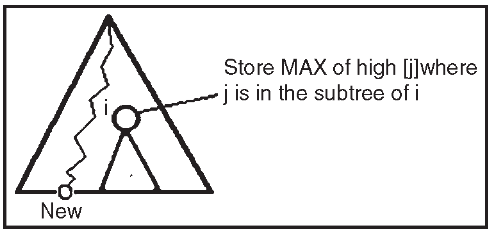


When we insert or delete a node we always have to update *predecessor* and *successor* pointers. In the Figure 8 we have deleted a new node with a *key*value 8. Accordingly we have to update *successor* and *predecessor* pointers. The time it takes to update pointers is equal to O(log$_2$*n*), since we either follow a path up the tree or down the tree. Rotation operations take as usual O(1) time, so the running time for insertion or deletion is O(log$_2$*n*). In figure the updated pointer.



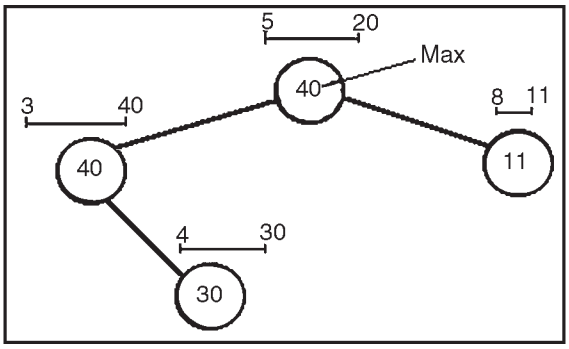
INTERVAL TREES

Interval tree is a binary search tree for intervals which are efficient for the dictionary operations and overlap. *Overlap*(*x*, *i*) for interval *i* and a tree rooted at *x*, returns a pointer to an interval in the collection that overlaps the given interval *i* or returns NIL otherwise. The underlying data structure is a red-black tree in which each node *x* contains an interval *int*[*x*] and the key of *x* is a low endpoint of interval, *low*[*int*[*x*]].*high*[*x*] is the high endpoint of interval. In addition, each node *x* contains a value *max*[*x*] which is the maximum value of any interval endpoint stored in the subtree rooted at *x*.



Operations

- Determine overlap with some interval in the tree: return Yes or No
- Dictionary operations (Insert, Delete, Search)



Given interval *i* and pointer to the root *x* Overlap(x, i) returns NIL if no overlap occured or pointer to the node if there is an overlap.

```
Overlap(x, i)
if x = NIL then return NIL
while x <> NIL and i doesn't overlap int[x] do
if left[x] <> NIL and low[i] Ü max[left[x]]
then x = left[x]
else x = right[x]
```

Running Time

The search for the inteval that overlaps *i* starts with *x* at the root of the tree and proceeds downward. It terminates when either overlapping interval is found or *x* becomes NIL. Since each iteration of the basic loop takes O(1) time, and since the height or the red-black tree is $O(\log_2 n)$ *Overlap*takes $O(\log_2 n)$ time.

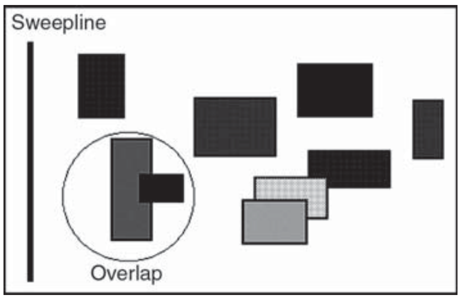
Insertion of a node *x* into a tree consists of *two* phases. During the first phase *x* is inserted as a child of an existing node.

The value of *max*[*x*] can be computed in O(1) time since it depends only on information in the other fields of *x* itself and *x*'s children, but *x*'s children are both NIL. Once *max*[*x*] is computed, the change propagates up the tree. Thus, total time for the first phase is $O(\log_2 n)$. During second phase the only structural changes are caused by rotations. Since only *two* nodes change in rotation, the total time for updating the *max* fields is $O(\log_2 n)$ per rotation. Since the number of rotations during insertion is at most *two*, the total time for insertion is $O(\log_2 n)$.

In the first phase of deletion, changes occur if the deleted node is replaced by its successor, and then again when either the deleted node or its successor is spliced out.

Propagating the updates to *max* caused by these changes costs at most $O(\log_2 n)$ since the changes modify the tree locally. Fixing up the red-black tree during the second phase requires at most *three* rotations, and each rotation requires at most $O(\log_2 n)$ time to propagate the updates to *max*. Thus, like insertion, the total time for deletion is $O(\log_2 n)$.

Problem: Layout of VSLI chips



In this section we present a problem which is motivated by automated chip design. The problem is how can we place many printed circuits on one chip without overlapping?

To make the problem more amenable to analysis, we assume that all n rectangles to be aligned with horizontal axis. We also assume that we are given a configuration of the rectangles, and our task is merely to check whether there is any overlap between rectangles.

In the naive solution we try to remove overlapped rectangles and place them randomly again. It takes time of Θ (n^2). The more efficient algorithm, which takes $O(n\log_2 n)$ time, uses a technique known as *sweeping*.

In sweeping an imaginary vertical sweep line passes through a given set of geometric objects (rectangles), usually from left to right. Sweeping provides a method for ordering geometric objects, usualy by placing them into dynamic data structure.

We sort the rectangles' endpoints by increasing *x*-coordinate and proceed from left to right. We insert *y*-direction intervals into an interval tree when its left endpoint is encountered and we delete it from interval tree when its right endpoint is encountered. Whether two segments first become consecutive in the total order, we check if they overlap.

Sweepline (Turning 2-d into 1-d)

- Sort *x* coordinates of rectangles
- Interval tree for y direction intervals
- for i = 1 to 2n do

If ith point is start then if (a, ß) overlap some interval in interval tree - STOP: Insert((*a*, ß), interval tree) *endpoint* then Delete((*a*, ß), interval tree) *a* is a segment immediately above segment *i* and ß is a segment immediately below segment *i*. Sorting takes O($n\log_2 n$) time using mergesort or heapsort.

Since there are 2*n* points the *for* loop iterates at most 2*n* times. Each iteration takes O($\log_2 n$) time, since each red-black tree operation takes O($\log_2$n) time. To check if there is an overlap takes O(1) time. Thus the total running time is O($\log_2$n).

B-TREES

INTRODUCTION

B-Tree is an indexing technique most commonly used in databases and file systems where pointers to data are placed in a balance tree structure so that all references to any data can be accessed in an equal time frame. It is also a tree data structure which keeps data sorted so that searching, inserting and deleting can be done in logarithmic amortized time.

Compared to a real tree, the B-Tree has roots at the top and leaves and nodes at the bottom. The B-Tree belongs to a group of techniques in computer science known as self-balancing search trees which attempts to automatically keep the number of levels of nodes under the root small at all times. It is the most preferred way to implement sets, associative arrays and other data structures that are used in computer Programmeming languages, relationaldatabase management systems and low level data manipulations.

In the B-Tree, the records are stored in the leaves. This is the location where there is nothing more beyond it. The order of the tree determines the maximum number of children per node. Depth refers to the number of required disk access. A B-Tree can have up to millions and billions of records although it is not all the time that leaves necessarily contain a record but more than half certainly do.

When decision points on the tree, which are called nodes, are on the hard disk instead of on random access memory (RAM), B-Tree is the preferred technique as hard disks could work a thousand times slower compared to RAM because processes on hard disks requires mechanical parts. On RAM, processes are done purely in electronic media.

The nodes in a B-Tree can have a variable number of child nodes within a range pre-defined by the system. When a data is inserted or removed from a node, the number ofchild nodes also changes but the pre-defined ranged should be maintained so internal nodes may either be split or joined.

B-trees do not need frequent re-balancing as both the upper and lower bounds on the number of child nodes are typically fixed. As an example, a 2-3 B-Tree implementation has internal nodes that can only have 2 or 3 child nodes.

To keep the B-tree well balanced, all leaf nodes are required to be of the same depth. The depth only increases very slowly and infrequently.

Searching in a B-Tree structure starts from the root and traversed from top to bottom. Insertion is done by looking for a node where a new leaf or element should be. If there is still room or the maximum legal number of elements is not exceeded, insertion takes place. Otherwise, the leaf node splits into another tow nodes. Deletion in a B-Tree has two strategies. The first involves locating the item to be deleted and immediately doing the action then restructuring the tree. The second involves doing a traversing down the tree and laying out the restructure before deleting.

In a file system, a file may contain any number of B-Trees and each B-Tree must have a unique name composed of any string of characters. Each B-Tree names is saved in the file an item containing the number of the rood node of the B-Tree. Searching, inserting and deleting through the B-Tree starts from the root node.

The B-Tree was created by Rudolf Bayer and Ed McCreight. The B in the B-Tree has ambiguous meaning. Some say it stands for Bayer while others say it stands for Balanced and still there are other who consider it to stand for Boeing where both men were working for Boeing Scientific Research Labs at the time they created the B-tree.

B-Tree structures support various basic dynamic set operations including *Search*, *Predecessor*, *Successor*, *Minimum*, *Maximum*, *Insert*, and *Delete* in time proportional to the height of the tree. Ideally, a tree will be balanced and the height will be *log n* where *n* is the number of nodes in the tree.

To ensure that the height of the tree is as small as possible and therefore provide the best running time, a balanced tree structure like a red-black tree, AVL tree, or b-tree must be used.

When working with large sets of data, it is often not possible or desirable to maintain the entire structure in primary storage (RAM). Instead, a relatively small portion of the data structure is maintained in primary storage, and additional data is read from secondary storage as needed. Unfortunately, a magnetic disk, the most common form of secondary storage, is significantly slower than random access memory (RAM). In fact, the system often spends more time retrieving data than actually processing data.

B-trees are balanced trees that are optimized for situations when part or all of the tree must be maintained in secondary storage such as a magnetic disk. Since disk accesses are expensive (time consuming) operations, a b-tree tries to minimize the number of disk accesses. For example, a b-tree with a height of 2 and a branching factor of 1001 can store over one billion keys but requires at most two disk accesses to search for any node.

STRUCTURE OF B-TREES

Unlike a binary-tree, each node of a b-tree may have a variable number of keys and children. The keys are stored in non-decreasing order. Each key has an associated child that is the root of a subtree containing all nodes with keys less than or equal to the key but greater than the preceeding key. A node also has an additional rightmost child that is the root for a subtree containing all keys greater than any keys in the node.

A b-tree has a minumum number of allowable children for each node known as the *minimization factor*. If *t* is this *minimization factor*, every node must have at least *t - 1* keys. Under certain circumstances, the root node is allowed to violate this property by having fewer than *t - 1* keys. Every node may have at most *2t - 1*keys or, equivalently, *2t* children.

Since each node tends to have a large branching factor (a large number of children), it is typically neccessary to traverse relatively few nodes before locating the desired key. If access to each node requires a disk access, then a b-tree will minimize the number of disk accesses required. The minimzation factor is usually chosen so that the total size of each node corresponds to a multiple of the block size of the underlying storage device. This choice simplifies and optimizes disk access. Consequently, a b-tree is an ideal data structure for situations where all data cannot reside in primary storage and accesses to secondary storage are comparatively expensive (or time consuming).

Height of B-Trees

For *n* greater than or equal to one, the height of an *n*-key b-tree T of height *h* with a minimum degree *t* greater than or equal to 2,

$$H \leq \log_t \frac{n+1}{2}$$

For a proof of the above inequality, refer to Cormen, Leiserson, and Rivest pages 383-384.

The worst case height is *O(log n)*. Since the "branchiness" of a b-tree can be large compared to many other balanced tree structures, the base of the logarithm tends to be large; therefore, the number of nodes visited during a search tends to be smaller than required by other tree structures. Although this does not affect the asymptotic worst case height, b-trees tend to have smaller heights than other trees with the same asymptotic height.

OPERATIONS ON B-TREES

The algorithms for the *search*, *create*, and *insert* operations are shown below. Note that these algorithms are single pass; in other words, they do not traverse back up the tree. Since b-trees strive to minimize disk accesses and the nodes are usually stored on disk, this single-pass approach will reduce the number of node visits and thus the number of disk accesses. Simpler double-pass approaches that move back up the tree to fix violations are possible.

Since all nodes are assumed to be stored in secondary storage (disk) rather than primary storage (memory), all references to a given node be be preceeded by a read operation denoted by *Disk-Read*.

Similarly, once a node is modified and it is no longer needed, it must be written out to secondary storage with a write operation denoted by *Disk-Write*. The algorithms below assume that all nodes referenced in parameters have already had a corresponding *Disk-Read* operation. New nodes are created and assigned storage with the *Allocate-Node* call. The implementation details of the *Disk-Read*, *Disk-Write*, and *Allocate-Node* functions are operating system and implementation dependent.

B-Tree-Search(x, k)

```
   $i \leftarrow 1$
   while $i \Leftarrow n[x]$ and $k > key_i[x]$
       do $i \leftarrow i + 1$
   if $i \Leftarrow n[x]$ and $k = key_i[x]$
           then return (x, i)
if leaf[x]
           then return NIL
           else Disk-Read($c_i[x]$)
                   return B-Tree-Search($c_i[x]$, k)
```

The search operation on a b-tree is analogous to a search on a binary tree. Instead of choosing between a left and a right child as in a binary tree, a b-tree search must make an n-way choice.

The correct child is chosen by performing a linear search of the values in the node. After finding the value greater than or equal to the desired value, the child pointer to the immediate left of that value is followed. If all values are less than the desired value, the rightmost child pointer is followed. Of course, the search can be terminated as soon as the desired node is found. Since the running time of the search operation depends upon the height of the tree, *B-Tree-Search* is $O(\log_t n)$.

B-Tree-Create(T)

```
x ← Allocate-Node()
leaf[x] ← TRUE
n[x] ← 0
Disk-Write(x)
root[T] ← x
```

The *B-Tree-Create* operation creates an empty b-tree by allocating a new root node that has no keys and is a leaf node. Only the root node is permitted to have these properties; all other nodes must meet the criteria outlined previously. The *B-Tree-Create* operation runs in time *O(1)*.

B-Tree-Split-Child(x, i, y)

```
z ¬ Allocate-Node()
leaf[z] ¬ leaf[y]
n[z] ¬ t - 1
for j ¬ 1 to t - 1
      do $key_j[z]$ ¬ $key_{j+t}[y]$
if not leaf[y]
```

```
    then for j ¬ 1 to t
         do $c_j[z]$ ¬ $c_{j+t}[y]$
n[y] ¬ t - 1
for j ¬ n[x] + 1 downto i + 1
         do $c_{j+1}[x]$ ¬ $c_j[x]$
$c_{i+1}$ ¬ z
for j ¬ n[x] downto i
       do $key_{j+1}[x]$ ¬ $key_j[x]$
$key_i[x]$ ¬ $key_t[y]$
n[x] ¬ n[x] + 1
Disk-Write(y)
Disk-Write(z)
Disk-Write(x)
```

If is node becomes "too full," it is necessary to perform a split operation.

The split operation moves the median key of node *x* into its parent *y* where *x* is the i^{th} child of *y*.

A new node, *z*, is allocated, and all keys in *x* right of the median key are moved to *z*. The keys left of the median key remain in the original node *x*. The new node, *z*, becomes the child immediately to the right of the median key that was moved to the parent *y*, and the original node, *x*, becomes the child immediately to the left of the median key that was moved into the parent *y*.

The split operation transforms a full node with *2t - 1* keys into two nodes with *t - 1* keys each. Note that one key is moved into the parent node.

The *B-Tree-Split-Child* algorithm will run in time *O(t)* where *t* is constant.

B-Tree-Insert(T, k)

```
r ← root[T]
if n[r] = 2t - 1
    then s ← Allocate-Node()
```

```
          root[T] ← s
          leaf[s] ← FALSE
          n[s] ← 0
          $c_1$ ← r
          B-Tree-Split-Child(s, 1, r)
          B-Tree-Insert-Nonfull(s, k)
else B-Tree-Insert-Nonfull(r, k)
```

B-Tree-Insert-Nonfull(x, k)

```
 i ¬ n[x]
 if leaf[x]
 then while i Þ 1 and k < $key_i[x]$
 do $key_{i+1}[x]$ ¬ $key_i[x]$
  i ¬ i - 1
 $key_{i+1}[x]$ ¬ k
  n[x] ¬ n[x] + 1
  Disk-Write(x)
 else while i Þ and k < $key_i[x]$
 do i ¬ i - 1
  i ¬ i + 1
  Disk-Read($c_i[x]$)
  if n[$c_i[x]$] = 2t - 1
  then B-Tree-Split-Child(x, i, $c_i[x]$)
  if k > $key_i[x]$
    then i ¬ i + 1
 B-Tree-Insert-Nonfull($c_i[x]$, k)
```

To perform an insertion on a b-tree, the appropriate node for the key must be located using an algorithm similiar to *B-Tree-Search*.

Next, the key must be inserted into the node. If the node is not full prior to the insertion, no special action is required; however, if the node is full, the node must be split to make room for the new key.

Since splitting the node results in moving one key to the parent node, the parent node must not be full or another split operation is required.

This process may repeat all the way up to the root and may require splitting the root node. This approach requires two passes.

The first pass locates the node where the key should be inserted; the second pass performs any required splits on the ancestor nodes.

Since each access to a node may correspond to a costly disk access, it is desirable to avoid the second pass by ensuring that the parent node is never full. To accomplish this, the presented algorithm splits any full nodes encountered while descending the tree. Although this approach may result in unecessary split operations, it guarantees that the parent never needs to be split and eliminates the need for a second pass up the tree.

Since a split runs in linear time, it has little effect on the $O(t \log_t n)$ running time of *B-Tree-Insert*.

Splitting the root node is handled as a special case since a new root must be created to contain the median key of the old root. Observe that a b-tree will grow from the top.

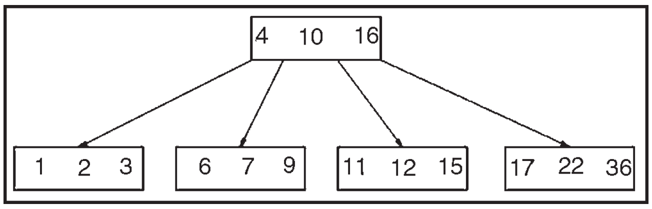
B-Tree-Delete

Deletion of a key from a b-tree is possible; however, special care must be taken to ensure that the properties of a b-tree are maintained.

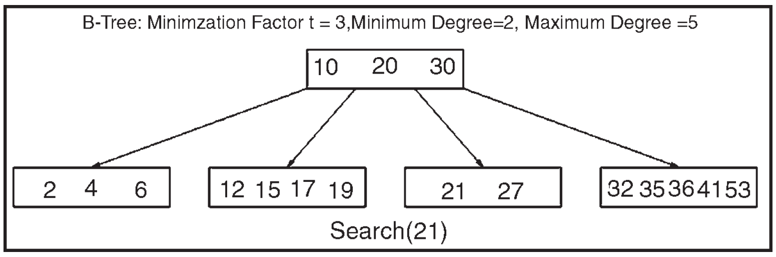
Several cases must be considered. If the deletion reduces the number of keys in a node below the minimum degree of the tree, this violation must be corrected by combining several nodes and possibly reducing the height of the tree. If the key has children, the children must be rearranged.

EXAMPLES

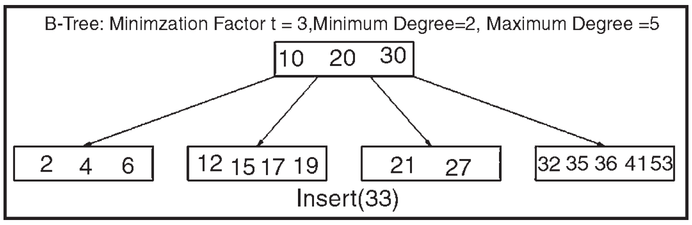
Sample B-Tree



Searching a B-Tree for Key 21


B-Tree: Minimzation Factor t = 3,Minimum Degree=2, Maximum Degree =5
10 20 30
2 4 6
12 15 17 19
21 27
32 35 36 41 53
Search(21)

Inserting Key 33 into a B-Tree (w/ Split)


B-Tree: Minimzation Factor t = 3,Minimum Degree=2, Maximum Degree =5
10 20 30
2 4 6
12 15 17 19
21 27
32 35 36 41 53
Insert(33)

APPLICATIONS

Databases

A database is a collection of data organized in a fashion that facilitates updating, retrieving, and managing the data. The data can consist of anything, including, but not limited to names, addresses, pictures, and numbers. Databases are commonplace and are used everyday. For example, an airline reservation system might maintain a database of available flights, customers, and tickets issued. A teacher might maintain a database of student names and grades.

Because computers excel at quickly and accurately manipulating, storing, and retrieving data, databases are often maintained electronically using a database management system. Database management systems are essential components of many everyday business operations. Database products like Microsoft SQL Server, Sybase Adaptive Server, IBM DB2, and Oracle serve as a foundation for accounting systems, inventory systems, medical recordkeeping sytems, airline reservation systems, and countless other important aspects of modern businesses.

It is not uncommon for a database to contain millions of records requiring many gigabytes of storage. For examples, TELSTRA, an Australian telecommunications company, maintains a customer billing database with 51 billion rows (yes, billion) and 4.2 terabytes of data. In order for a database to be useful and usable, it must support the desired operations, such as retrieval and storage, quickly.

Because databases cannot typically be maintained entirely in memory, b-trees are often used to index the data and to provide fast access.

For example, searching an unindexed and unsorted database containing n key values will have a worst case running time ofO(n); if the same data is indexed with a b-tree, the same search operation will run in O(log n).

To perform a search for a single key on a set of one million keys (1,000,000), a linear search will require at most 1,000,000 comparisons. If the same data is indexed with a b-tree of minimum degree 10, 114 comparisons will be required in the worst case. Clearly, indexing large amounts of data can significantly improve search performance. Although other balanced tree structures can be used, a b-tree also optimizes costly disk accesses that are of concern when dealing with large data sets.

Concurrent Access to B-Trees

Databases typically run in multiuser environments where many users can concurrently perform operations on the database. Unfortunately, this common scenario introduces complications. For example, imagine a database storing bank account balances. Now assume that someone attempts to withdraw $40 from an account containing $60. First, the current balance is checked to ensure sufficent funds.

After funds are disbursed, the balance of the account is reduced. This approach works flawlessly until concurrent transactions are considered. Suppose that another person simultaneously attempts to withdraw $30 from the same account. At the same time the account balance is checked by the first person, the account balance is also retrieved for the second person.

Since neither person is requesting more funds than are currently available, both requests are satisfied for a total of $70. After the first person's transaction, $20 should remain ($60 - $40), so the new balance is recorded as $20. Next, the account balance after the second person's transaction, $30 ($60 - $30), is recorded overwriting the $20 balance. Unfortunately, $70 have been disbursed, but the account balance has only been decreased by $30. Clearly, this

Behaviour is undesirable, and special precautions must be taken. A b-tree suffers from similar problems in a multiuser environment. If two or more processes are manipulating the same tree, it is possible for the tree to become corrupt and result in data loss or errors.

The simplest solution is to serialize access to the data structure. In other words, if another process is using the tree, all other processes must wait. Although this is feasible in many cases, it can place an unecessary and costly limit on performance because many operations actually can be performed concurrently without risk.Locking, introduced by Gray and refined by many others, provides a mechanism for controlling concurrent operations on data structures in order to prevent undesirable side effects and to ensure consistency. For a detailed discussion of this and other concurrency control mechanisms

BINOMIAL

WHAT IS A BINOMIAL EXPERIMENT

A binomial experiment has the following characteristics:

- The experiment involves repeated trials.
- Each trial has only two possible outcomes - a success or a failure.
- The probability that a particular outcome will occur on any given trial is constant.
- All of the trials in the experiment are independent.

A series of coin tosses is a perfect example of a binomial experiment. Suppose we toss a coin three times. Each coin flip represents a trial, so this experiment would have 3 trials.

Each coin flip also has only two possible outcomes - a Head or a Tail. We could call a Head a success; and a Tail, a failure.

The probability of a success on any given coin flip would be constant (i.e., 50%). And finally, the outcome on any coin flip is not affected by previous or succeeding coin flips; so the trials in the experiment are independent.

BINOMIAL DISTRIBUTION

A binomial distribution is a probability distribution. It refers to the probabilities associated with the number of successes in a binomial experiment.

For example, suppose we toss a coin three times and suppose we define Heads as a success. This binomial experiment has four possible outcomes: 0 Heads, 1 Head, 2 Heads, or 3 Heads. The probabilities associated with each possible outcome are an example of a binomial distribution, as shown below.

Outcome,x	**Binomial probability,**	**Cumulative probability,**
P(X = x)		**P(X < x)**
0 Heads	0.125	0.125
1 Head	0.375	0.500
2 Heads	0.375	0.875
3 Heads	0.125	1.000

NUMBER OF TRIALS

The number of trials refers to the number of attempts in a binomial experiment. The number of trials is equal to the number of successes plus the number of failures.

Suppose that we conduct the following binomial experiment. We flip a coin and count the number of Heads. In this experiment, Heads would be classified as success;

tails, as failure. If we flip the coin 3 times, then 3 is the number of trials. If we flip it 20 times, then 20 is the number of trials.

NUMBER OF SUCCESSES

Each trial in a binomial experiment can have one of two outcomes. The experimenter classifies one outcome as a success; and the other, as a failure. The number of successes in a binomial experient is the number of trials that result in an outcome classified as a success.

PROBABILITY OF SUCCESS ON SINGLE TRIAL

In a binomial experiment, the probability of success on any individual trial is constant. For example, the probability of getting Heads on a single coin flip is always 0.50. If "getting Heads" is defined as success, the probability of success on a single trial would be 0.50.

BINOMIAL PROBABILITY

A binomial probability refers to the probability of getting EXACTLY n successes in a specific number of trials. For instance, we might ask: What is the probability of getting EXACTLY 2 Heads in 3 coin tosses. That probability (0.375) would be an example of a binomial probability.

CUMULATIVE BINOMIAL PROBABILITY

Cumulative binomial probability refers to the probability that the value of a binomial random variable falls within a specified range.

The probability of getting AT MOST 2 Heads in 3 coin tosses is an example of a cumulative probability. It is equal to the probability of getting 0 heads (0.125) plus the probability of getting 1 head (0.375) plus the probability of getting 2 heads (0.375).

Thus, the cumulative probability of getting AT MOST 2 Heads in 3 coin tosses is equal to 0.875.

Notation associated with cumulative binomial probability is best explained through illustration. The probability of getting FEWER THAN 2 successes is indicated by $P(X < 2)$; the probability of getting AT MOST 2 successes is indicated by $P(X < 2)$; the probability of getting AT LEAST 2 successes is indicated by $P(X > 2)$; the probability of getting MORE THAN 2 successes is indicated by $P(X > 2)$.

RELATION BETWEEN BINOMIAL AND NORMAL DISTRIBUTIONS

When the number of trials is large and when the probability of success is not extreme (i.e., neither close to 0 nor close to 1), then the normal distribution may be used to very closely approximate results from the binomial distribution.

Note: When the number of trials is greater than 20,000, the Binomial Calculator uses a normal distribution to estimate the cumulative binomial probability.

In most cases, this yields very good results - often accurate to the third decimal place.

BINOMIAL DISTRIBUTION: PROBLEMS

Question

- Suppose you toss a fair coin 12 times. What is the probability of getting exactly 7 Heads.

Solution

We know the following:

- The number of trials is 12.
- The number of success is 7 (since we define getting a Head as success).
- The probability of success (i.e., getting a Head) on any single trial is 0.5.

Therefore, we plug those numbers into the Binomial Calculator and hit the Calculate button. The calculator reports that the binomial probability is 0.193.

That is the probability of getting EXACTLY 7 Heads in 12 coin tosses. (The calculator also reports the cumulative probability - the probability of getting AT MOST 7 heads in 12 coin tosses. The cumulative probability is 0.806.)

Question

- Suppose the probability that a college freshman will graduate is 0.6 Three sisters (triplets) enter college at the same time. What is the probability that at most 2 sisters will graduate?

Solution

We know the following:

- The number of trials is 3 (because we have 3 sisters).
- The number of successes is 2.
- The probability of success for any individual sister is 0.6.

Therefore, we plug those numbers into the Binomial Calculator and hit the Calculate button. The calculator reports that the cumulative binomial probability is 0.784. That is the probability that 2 or fewer sisters will graduate is 0.784. (Note that the calculator also displays the binomial probability - the probability that EXACTLY 2 sisters graduate. The binomial probability is 0.432.)

INSTRUCTIONS

A "Begin" button will appear on the left when the applet is finished loading. This may take a minute or two depending on the speed of your internet connection and computer. Please be patient. If no begin button appears, it is probably because your browser does not support Java 1.1.

SETTING UP CONDITIONS

Press the "Begin" button to start the applet in another window.

This Java applet shows how the binomial distribution can be approximated by the normal distribution. The initial values are for a binomial distribution with the parameters N = 8 and p = 0.5 where N is the number of trials and p is the probability of success on each trial. You can change the values of N and p and see the result (Hit the enter or tab key after changing a value).

You can use this applet to calculate the probability of obtaining a given number of successes. For example, to calculate the probability of exactly 6 successes out of 8 trials with p = 0.50, enter 6 in both the "from" and "to" fields and hit the "Enter" key. The actual binomial probability is 0.1094 and the approximation based on the normal distribution is 0.1059.

Note that the normal approximation computes the area between 5.5 and 6.5 since the probability of getting a value of exactly 6 in a continuous distribution is nil.

Similarly, to approximate the probability of from 0 to 6 successes, you enter 0 in the “from” field and 6 in the “to” field.

The area from below 6.5 is computed.

BINOMIAL PROBABILITIES

- Exact binomial probabilities
- Approximation via the normal distribution
- Approximation via the Poisson Distribution

The logic and computational details of binomial probabilities are described earlier.

This will calculate and/or estimate binomial probabilities for situations of the general”k out of n” type, where k is the number of times a binomial outcome is observed or stipulated to occur, p is the probability that the outcome will occur on any particular occasion,q is the complementary probability (1-p) that the outcome will not occur on any particular occasion, and n is the number of occasions.

For example: In 100 tosses of a coin, with 60 “heads” outcomes observed or stipulated to occur among the 100 tosses,

n = 100	[the number of opportunities for a head to occur]
k = 60	[the stipulated number of heads]
p =.5	[the probability that a head will occur on any particular toss]
q =.5	[the probability that a head will not occur on any particular toss]

Method 1

If n ≤ 1000, exact binomial probabilities will be calculated through repeated applications of the standard binomial formula$_{Q}$

$$p_{(k\,\text{out of}\,n)} = \frac{n!}{k!(n-k)!}\left(p^{k}\right)\left(q^{n-k}\right)$$

In principle, Method 1 is preferable in all cases, since it involves direct calculation of exact binomial probabilities. Its limitation is that it is not computationally feasible with very large samples. The Programmeming on this page is capable of performing the calculation up through n=1000.

Method 2

If np ≥ 5 and nq ≥ 5, binomial probabilities will be estimated by way of the binomial approximation of the normal distribution, according to the formula$_{Q}$

$$z = \frac{(k-m) \pm 5}{\sigma}$$

where:

- M = np [the mean of the binomial sampling distribution]
- σ = sqrt[npq] [the standard deviation of the binomial sampling distribution]

Method 3

If ne”150 and the mean (np) and variance (npq) of the binomial sampling distribution are within 10% of each other, binomial probabilities will be estimated through repeated applications of the Poisson probability function

where:

e = the base of the natural logarithms; and

M = np [the mean of the binomial sampling distribution]

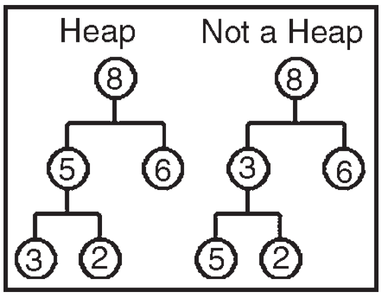
The defining characteristic of a Poisson distribution is that its mean and variance are identical. In a binomial sampling distribution, this condition is approximated as p becomes very small, providing that n is relatively large. The Programmeming on this page permits the Poisson procedure to be performed whenever np and npq are within 10% of each other, providing that ne”150. Do keep in mind, however, that the results of the Poisson procedure are only approximations of the true binomial probabilities, valid only in the degree that the binomial mean and variance are very close.

HEAPS

A heap is a complete tree with an ordering-relation R holding between each node and its descendant. *Examples for R*: smaller-than, bigger-than

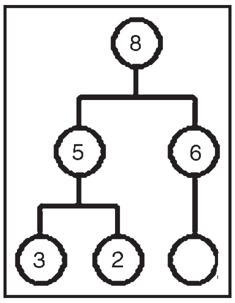
ASSUMPTION

In what follows, *R* is the relation ‘bigger-than’, and the trees have degree 2.

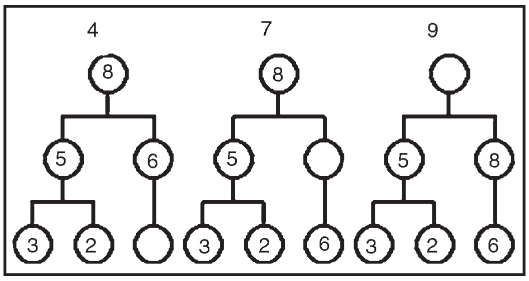


ADDING AN ELEMENT

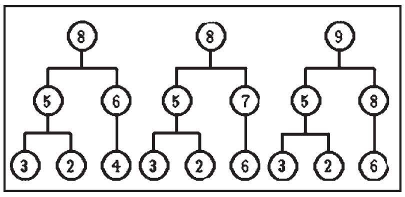
- Add a node to the tree



- Move the elements in the path from the root to the new node one position down, if they are smaller than the new element



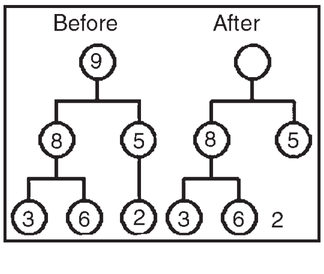
- Insert the new element to the vacant node



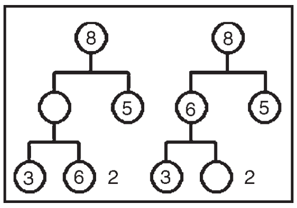
- A complete tree of n nodes has depth $\lceil \log n \rceil \log n$ O(log), hence the time complexity is $O(\log n)$

DELETING AN ELEMENT

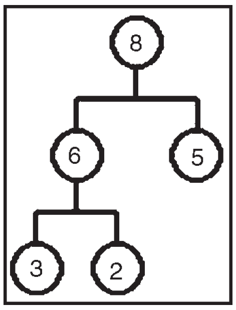
- Delete the value from the root node, and delete the last node while saving its value.



- As long as the saved value is smaller than a child of the vacant node, move up into the vacant node the largest value of the children.



- Insert the saved value into the vacant node



- The time complexity is $O(\log n)$

INITIALIZATION

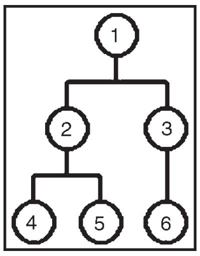
Brute Force

Given a sequence of n values $e_1..., e_n$, repeatedly use the insertion module on the n given values.

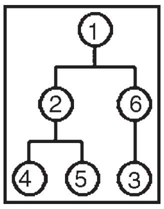
- Level h in a complete tree has at most $2^{h-1} = O(2^n)$ elements
- Levels $1..., h - 1$ have $2^0 + 2^1 +... + 2^{h-2} = O(2^h)$ elements
- Each element requires $O(\log n)$ time. Hence, brute force initialization requires $O(n \log n)$ time.

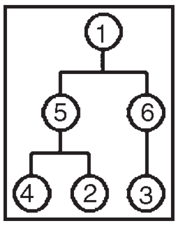
Efficient

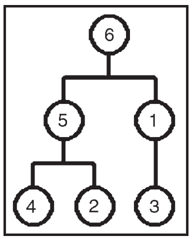
- Insert the n elements $e_1..., e_n$ into a complete tree

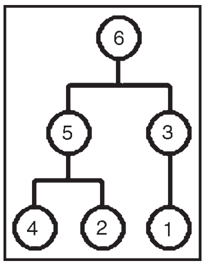


- For each node, starting from the last one and ending at the root, reorganize into a heap the subtree whose root node is given. The reorganization is performed by interchanging the new element with the child of greater value, until the new element is greater than its children.









- The time complexity is $O(0 * (n/2) + 1 * (n/4) + 2 * (n/8) + \ldots + (\log n) * 1) = O(n(0.\ 2^{-1} + 1\ldots\ 2^{-2} + 2.\ 2^{-3} + \ldots + (\log n).2^{-\log n})) = O(n)$
  since the following equalities holds.

$$\sum_{k=1}\infty(k-1)2-k=2\left[\sum_{k=1}\infty(k-1)2-k\right]-$$
$$\left[\sum_{k=1}\infty(k-1)2-k\right]=\left[\sum_{k=1}\infty k2-k\right]-$$
$$\left[\sum_{k=1}\infty(k=1)2-k\right]\sum_{k=1}\infty\left[k-(k-1)\right]2-k=$$
$$\sum_{k=1}\infty 2-k=1$$

APPLICATIONS

Priority Queue

A dynamic set in which elements are deleted according to a given ordering-relation.

Heap Sort

Build a heap from the given set ($O(n)$) time, then repeatedly remove the elements from the heap ($O(n \log n)$).