Introduction to Algorithms

An algorithm is any well-defined computational procedure that takes some value or set of values as input, and produces some value or set of values as output.

1 Why study algorithms?

As the speed of processors increase, performance is often said to be less important than the other software quality characteristics (e.g. security, extensibility, re-usability, etc.). However, large problem sizes are common place in the area of computational science; which makes performance a very important factor. This is because longer computation time, to name a few, mean slower results, less through research, and higher cost of computation (if buying CPU hours from an external party).

The study of algorithms, therefore, gives us a language to express performance as a function of problem size. A similar approach can also be taken to express resource requirements as a function of problem size. It is important to keep resource requirements in check as it may easily result in poorer performance.

2 Notation

The representation of the problem, the solution, and the algorithm to obtain the solution are greatly influenced by the state of technology. In this day and age, the basic model is the sequential processor with access to random access memory. Simple problems in Mathematics such as "the minimum of a set of integers" requires us to consider two points:

1. How is the set represented?
2. How to find the minimum (depends on how the set is represented)?

To keep things simple[1], let's assume that the integers are stored in a sequential array; and that we know the number of integers to be n. Since *any* one of the integers in the array may be the minimum, the solution is simply to take a look at each one of them and report the minimum. The following algorithm implements this strategy.

```
FIND-MINIMUM (A,n)
if n=0
    then return NIL  /* No minimum in an empty array. */
m ← A[1]
i ← 2
while i≤n
        do if A[i]<m
                then m ← A[i]
           i ← i+1
return m
```

The FIND-MINIMUM algorithm reflects on the state of present day computer technology:

- The computer executes one line at the time.

- There may be branches (if).
- There may be iterations (while, for, repeat/until).
- There may be assignments (m ← A[1]).
- There may be random memory accesses (Using A[i] to access the ith element of A).

3 Analysis

The basic idea is that the more instructions a computer has to process, the longer it will take to complete processing. Thus, the aim of analysing algorithms is to describe how much work a computer has to do. Since the amount of work usually depends on the problem size, we are mostly interested how an algorithm scales with problem size.

For starters, let us examine the FIND-MINIMUM algorithm introduced earlier. Lines 1 to 4 can be completed in some constant amount of time c1 . Due to the while-loop at line 5, lines 5 to 8 take c2 (n-1) time. Lastly, line 9 takes c3 time. Total time taken is therefore the sum of all these:

c1 + c2 (n-1)+ c3 = c1 + c2 n- c2 + c3 Let c4 = c1 - c2 + c3 = c4 + c2 n

Therefore, each element in the input array contributes c2 time to the total runtime.

4 Order of growth

How an algorithm scales in relation to its input problem size is termed order of growth. For example, say we have an algorithm that runs at n3 + n2 +n+4. The following table shows us the significance of each term:

Value of n	n3	n2	n1	4
1	1	1	1	4
10	1000	100	10	4
100	1000000	10000	100	4
1000	1000000000	1000000	1000	4

As can be seen, as n approaches infinity, all the other terms fade in significance next to n3 . Therefore the order of growth of the algorithm is n3 .

So applying this vocabulary to FIND-MINIMUM, whose cost we found to be c4 + c2 n, we say that the order of growth is n.

5 Insertion-Sort Example

A sorting algorithm arranges the elements of a list into a certain order. A simple example is the INSERTION-SORT algorithm. The strategy is to divide the list into a set of sorted elements and a set of unsorted elements. We then take an element from the unsorted set and insert it into the correct position in the sorted set until the unsorted set is empty.

INSERTION-SORT(A)	**Cost**	**Times**

```
1  for j ← 2 to length[A]                  c1   n
2       do key ← A[j]                      c2   n-1
3          i ← j-1                         c3   n-1
4          while i>0 and A[i]>key          c4   Σj=2 n tj
5                 do A[i+1] ← A[i]         c5   Σj=2 n ( tj -1)
6                    i ← i-1               c6   Σj=2 n ( tj -1)
7          A[i+1] ← key                    c7   n-1
```

To compute the running time of this algorithm, we sum the products of the cost and times columns. If the array is in reverse sorted order, the worst case will apply, and so, noting that:

$$\Sigma j=2\ nj= n(n+1)\ 2\ -1$$

the worst case running time is

$$T(n)=(\ c4\ 2 + c5\ 2 + c6\ 2\ )\ n2 +(\ c1 + c2 + c3 + c4\ 2 - c5\ 2 - c6\ 2 + c7\ )n-(\ c2 + c3 + c4 + c7\ )$$

Hence the order of growth is n2 .

6 Design of Algorithms

As with all areas of computing, there are many different approaches to designing an algorithm to solve any given problem. Over the course of this unit, we will look at a few of these approaches, and how to recognise which approach is best for which problem.

Some of the popular approaches include:

- Divide-and-conquer
- Greedy approach
- Dynamic programming
- Linear programming
- Branch-and-bound
- α-β pruning

7 Divide-and-Conquer Algorithms

Divide-and-Conquer is a recursive approach, which involves the following steps:

1. **Divide** the problem into a number of subproblems.
2. **Conquer** the subproblems by solving them recursively. If however the subproblems are small enough, simply solve them.
3. **Combine** the solutions of the subproblems into the solution for the original problem.

Consider the sorting problem. The insertion sort uses an incremental approach, it sorts the first item, then the first two items, then first three items, and so on, until it has sorted all n items of the array.

Another approach is merge sort. It uses the divide-and-conquer approach, as follows:

1. **Divide** the n-element sequence into two subsequences of n/2 elements each.
2. **Conquer** the two subsequences by recursively using the merge sort on each of them.
3. **Merge** the two sorted subsequences into a sorted sequence. See Figure 1.

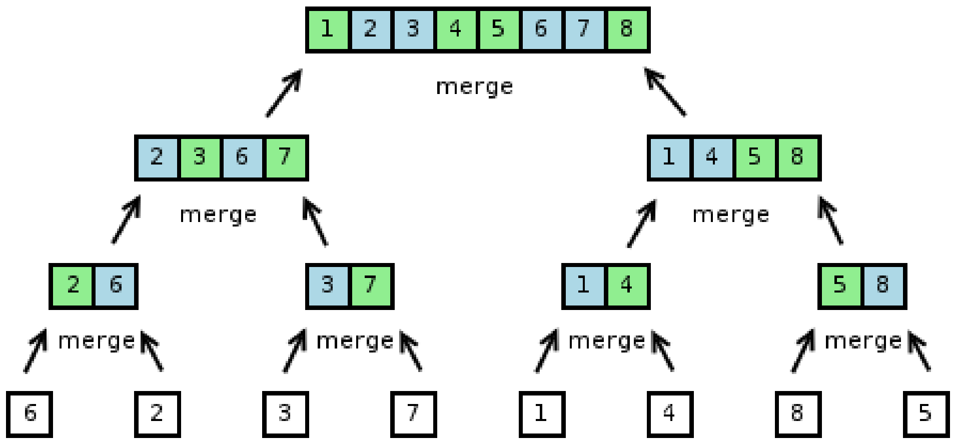

Figure 1: Merging an eight integer sequence.

8 Analysis of Merge Sort

The running time of an algorithm that makes a recursive call to itself is often called a "recursion". We can analyse the running time of merge sort by looking at each of its three steps.

1. **Divide** - Dividing the array requires simply finding the mid-point of the array, this takes constant time, thus D(n)=1.
2. **Conquer** - Here the algorithm makes two calls to itself, each one with an order of input of n/2. Thus C(n)=2T(n/2).
3. **Merge** - The process of merging the two sorted arrays can be done in M(n)=n time.

Combining the above, and remembering that an order of growth of 1 is insignificant next to an order of growth of n, we get the following:

$$T(n)=\begin{cases} c & \text{if } n=1 \\ 2T(\frac{n}{2})+\Theta(n) & \text{if } n>1 \end{cases}$$

Using the above formula, we can build a tree (Figure 2) depicting how much time is spent at each level of the recursion. The summation of all the costs of each level shows that the running time of merge sort is nlgn.

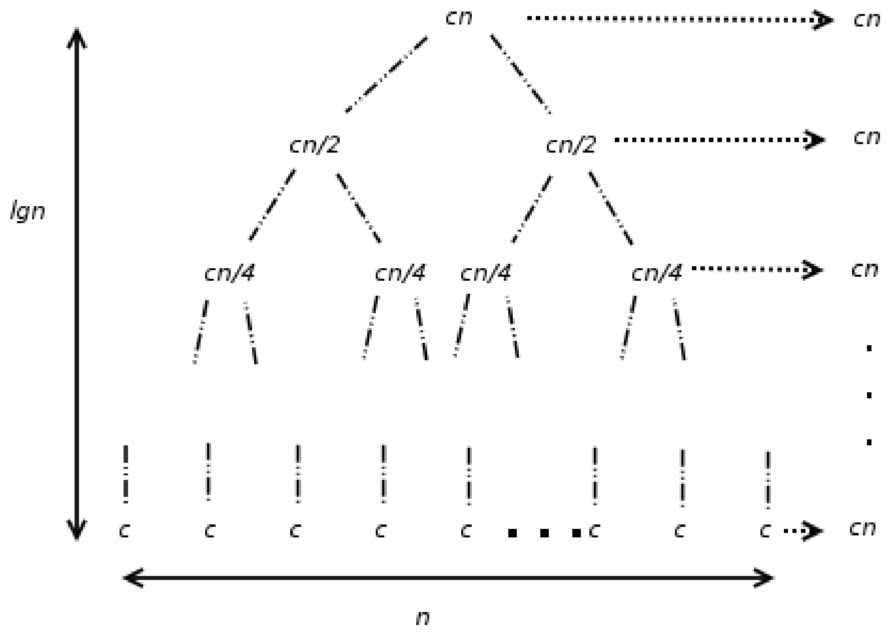

Figure 2: The breakdown of the cost of a Merge Sort. Each level of the tree represents a level of recursion in the algorithm; thus there are lgn levels. The cost of each level is shown on the right of the tree. Intuitively, the cost of the whole algorithm is the sum of all the level costs. In this case, it is nlgn.

Footnotes:

1 Readers who consistently require extracting the minimum/maximum value of a set should take a look at the "heap" data structure covered in these notes.

File translated from $\TeX$ by TTM, version 3.67.
On 31 Mar 2006, 18:12.