Chapter 3: Searching and Sorting, Algorithmic Efficiency

Search Algorithms

Problem: Would like an efficient algorithm to find names, numbers etc. in lists/records (could involve DATA TYPES).

Basic Search: Unsorted Lists

Purpose: To find a name in an unsorted list by comparing one at a time until the name is found or the end of the array is reached.

See 'Example 4' in the example programs for a basic program to do this.

Searching through data types requires some different syntax...

Example: Search through data types for a record corresponding to name 'Zelda'

Suppose each name and number is stored in a data type

```
type person
  character(len=30)::name
  integer::number
end type person
```

We declare an array of person(s)

```
type(person)::directory(1000)
```

Then search for 'Zelda'.

```
Do i=1,1000
  if (directory(i)%name=='Zelda') then
  print *,directory(i)%number
  end if
End do
```

This algorithm is fine for an unsorted list of names. However, if we know that the list has been stored in some order (i.e. it is sorted) then there are faster methods.

Binary Search: Sorted Lists

Purpose: To search through an ordered array more quickly. (Note that $<, >=, ==$ etc. can be used with character strings to compare alphabetical order).

Algorithm:

1. Data: The name to be searched for is placed in a alphabetically ordered array 'Namearray' that is filled from data files.

2. Set Low=1 Low, High integers
   Set High=N where N is Size(Namearray)

3. Compare Name with Namearray(Low) and Namearray(High).
   If there is a match, then exit.
   Else Namearray(Low) < Name < Namearray(High).
   Set Mid=(Low+High)/2 !Note integer division, so Mid is an integer $\approx$ N/2.

4. Compare Name with Namearray(Mid)
   if (Name < Namearray(Mid))
   High=Mid
   Mid=(High+Low)/2 !Sets Mid$\approx$N/4
   if (Name == Namearray(Mid)), We are finished.
   if (Name > Namearray(Mid))
   Low=Mid
   Mid=(High+Low)/2 !Sets Mid$\approx$3N/4

5. Repeat Step 4 above until name is found or array is exhausted.

6. Print out results.

Algorithmic Efficiency

Suppose an algorithm handles N pieces of data. Let the time needed to complete the calculation be Z(N).

Z(N) will be a function of both the speed of the computer and the dataset used.
Z(N) may refer either to the worst case scenario or the average over a large number of cases.
Define g(N) to be the 'order' of Z(N)...i.e. Z(N) = $O($ g(N) $)$ or

$$\lim_{N\to\infty} \frac{\mathrm{Z(N)}}{\mathrm{g(N)}} = \text{const} \neq 0.$$

Example:

$$\begin{aligned} \mathrm{Z(N)} &= 3\mathrm{N}^3 + 20\mathrm{N}^2 + 7\mathrm{N} + 3 \\ \text{then} \quad \mathrm{g(N)} &= \mathrm{N}^3 \end{aligned}$$

Often, algorithms are classified using simple functions, e.g.,

polynomial or power	$\mathrm{g(N)} = \mathrm{N}^a$
exponential	$\mathrm{g(N)} = \exp \mathrm{N}a$
logarthmic	$\mathrm{g(N)} = \log_2 \mathrm{N}$
factorial	$\mathrm{g(N)} = N!$

Example: Search Algorithms
Unsorted: Average N/2, Worst Case N.
Sorted (binary search): Average $(1/2)\log_2 \mathrm{N}$, Worst Case: $\log_2 \mathrm{N}$.

The Travelling Salesman Problem

Problem: A salesman must visit N cities (in any order). Which order minimises travel distances?

The 'brute force' approach to this problem is an example of a factorial algorithm (as there are N! possible combinations to check). The brute force method is therefore crippling slow even for fairly modest N.

It is a classic problem in computational science /mathematics to come up with faster algorithms.

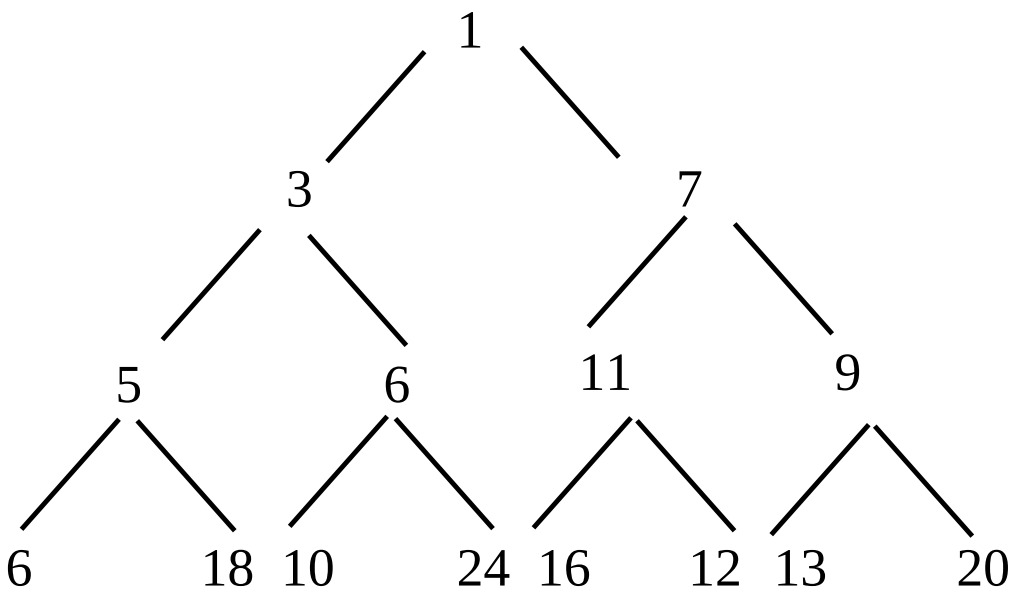

Figure: An example of a partially ordered heap.

Sorting Algorithms

Problem: Algorithm must sort names or numbers in an array OLDARRAY into a particular order.

Sorting Algorithms: Insertion Sort

Algorithm:

1. Create NEWARRAY
2. One at a time, place elements of OLDARRAY at a correct position in NEWARRAY. The simple way of doing this is to start at the top of the new (sorted) array and go downwards until place is found (see e.g. Examples 10, 10a).

This algorithm has efficiency g(N)=$O(\mathrm{N}^2)$.

Sorting Algorithms: Heapsort

Idea: 2 steps

1. Partially sort data into a heap....a partially sorted 'binary tree'.
2. Complete the sort.

In a heap, the top level has one element, second level 2 elements, nth level 2^{n-1} elements. The last level need not be complete.

Each number in the heap is greater than or equal to its immediate 'boss', i.e. the number directly above it in the tree.

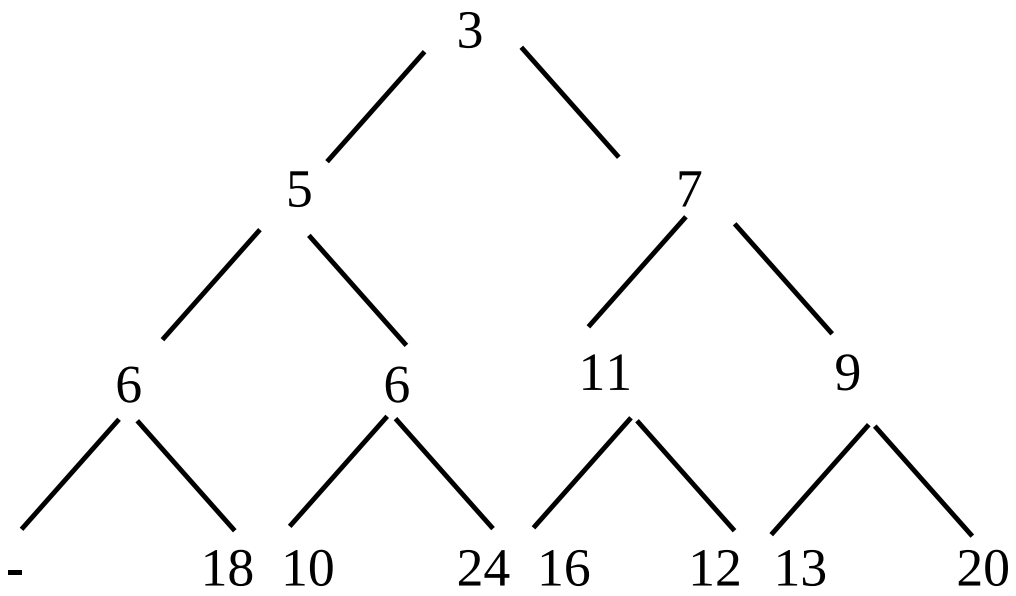

Figure: The partially ordered heap above after one 'retirement' and all the subsequent promotions.

Algorithm:

Store the heap in an array A, A(1)- top element, A(2) A(3) - second row etc.

Step 2 first: To completely order a partially ordered heap

1. Create a new array B. Let B(1)=A(1).
2. Next step is to 'retire' A(1) and promote the lesser of A(2) and A(3) to the top job.
3. Promote one of the underlings of A(2) or A(3) to their vacant job.
4. Keep going until vacancy reaching bottom.
5. Return to 2 above and retire the top number into the next available space in B, until the whole array is sorted and B is filled..

Now Step 1: To create a partially ordered heap from randomly arranged data.

Suppose A is initially filled in a completely random way. The idea here is to create the heap from the bottom row upwards. Note that the bottom row, because they don't have any 'workers' below, already satisfy the 'heap' condition that each 'boss' has a lower value than its two 'workers'.

1. Go through the next-to-bottom row one at a time. Where one or both workers have a lower value than their boss, promote the lower-valued worker and demote the boss. The next-to-bottom and bottom rows now satisfy the 'heap' condition.
2. Look on the two-from-bottom row. Again, when a 'boss' with a higher value than one of its workers is found the two are swapped.
3. This time the demoted boss must be checked in the next-to-bottom row to see if it needs further demotion. This must be done before moving on to check the next boss in the two-from-bottom row.
4. Continue until the bottom three rows all satisfy the 'heap' condition.
5. Move on to the three-from bottom row, and continue, making sure that each boss is demoted all the way down to the bottom row if necessary.

6. Continue until the whole array satisfies the 'heap' condition.

The key point about the HEAPSORT algorithm is that has efficiency $Z(N)=O(N\log_2 N)$, as opposed to $O(N^2)$ for the insertion sort algorithm. This means that for large N it is much, much faster, and so it becomes more than worthwhile to deal with the extra complexity.