Algorithms

1) Definition : An algorithm is a finite set of precise instructions for performing a computation, or for solving a problem

2) Vocabulary :

Input : input values from a specified set

Output : solution to the problem

correctness : an algorithm should produce the correct output values for all input values

generality : The algorithm should be applicable for all problems of the desired form, not just for a particular set of input values

finiteness, effectiveness : An algorithm should produce the desired output after a finite (but perhaps large) number of steps for any input

Time complexity : Expressed in terms of the number of operations used by the algorithm when the input has a particular size. (usually expressed as big-oh or big-Θ)

How to present an algorithm!

- → plain English
- → Computer language
- → pseudo code

Pseudo code we will use:

Assignment:

- $i \leftarrow 0$ Assign the value 0 to i

Loops:

- For (Start; End; Step)
  {
  Body
  }
  example: Fa (i=1; $i \leq n$; Step=1)
  executes the loop for all
  i in {1, ... n}
- while (proposition)
  {
  Body
  }
  Example: while ($i \leq n$)

Conditional

If (proposition) then
{
Body 1
}
else if (proposition 2) then
{
Body 2
}
else
{
Body 3
}
endif

3. Searching algorithms

Definition: Searching algorithms solve the problem of locating an element in an ordered list.

This problem occurs in many contexts (think for example spellcheck...)

The simplest way to solve this problem is to test each element of the list → linear search

Pseudo code:

procedure Linear Search (Input: N (integer), $a_1, \dots a_N$ (integers)
x (integer)
Output: Index (integer))

```
Index ← 0
For (i=1; i ≤ N; Step=1)
{
    if (a_i == x) then
        index ← i
        return
    endif
}
return
```

What is the complexity of this algorithm?

It depends!

if $x = a_1$ → 1 comparison) Very different behaviours

if $x = a_N$ → N comparisons)

We define two measures of complexity:

Wast case complexity: largest number of operations needed to solve the given problem of a specified size

Average case complexity: Average number of operations

Wast case complexity for linear search:

When x is in the list. The wast case occurs when x is the last element of the list. The loop is fully executed.

Number of comparisons: 2 per steps, N steps.

Hence the total number of comparison is $2N$, which is $O(N)$

Average case complexity:

if $x = a_1$ the number of comparisons is 2

$x = a_2$ " 4

⋮

$x = a_N$ " $2N$

Average number: $\frac{1}{N}(2 + \cdots 2N) = \frac{2}{N}(1 + \cdots N) = \frac{2}{N}\frac{N(N+1)}{2} = N+1$, i.e $O(N)$

Linear search however is not optimal, as it does not take into account the fact that the data are ordered.

Another approach: Binary search

Example: Let $S = [1, 3, 7, 11, 17, 23, 32, 48, 49, 51, 53, 60]$

and $x = 53$

If we use ~~binary~~ search, we need 22 comparisons.

Binary search proceeds by dichotomy.

$\lfloor \frac{N+1}{2} \rfloor = 6$ $\quad 1, 3, 7, 11, 17, \boxed{23}, 32, 48, 49, 51, 53, 60$

53 is bigger than 23: we only need to look at the right of 23

$\lfloor \frac{N+1}{2} \rfloor = 3$ $\quad 32, 48, \boxed{49}, 51, 53, 60$

53 is bigger than 49: we only need to look at the right of 49

$\lfloor \frac{N+1}{2} \rfloor = 2$ $\quad 51, \boxed{53}, 60$

We are done!

We only need 3 comparisons ... + 3 auxiliary operations to find which number to compare.

Pseudo code for binary search

Procedure Binary Search (Input: N (integer), $a_1, \ldots a_N$ (integers), x (integer); Output: Index (integer))

```
Index ← 0
Left ← 1
Right ← N
while ( Left ≤ Right)
{
        Center ← floor ( (Right + Left) / 2 )
        if ( x == a_center ) then
        {
                index ← center
                return
        }
        else if ( x < a_center ) then
        {
                Right ← Center - 1
        }
        else
        {
                Left ← Center + 1
        }
        endif
}
return
```

What is the worst case complexity of the binary search?

Each step requires 2 comparisons + 1 "operation". For simplicity, we assume that the total cost of one step is "3".

How many steps do we need?

Let us assume $n = 2^k$ (otherwise $2^l \le n < 2^{l+1}$ and we set $k = l+1$). Then $k = \log_2 n$.

At each step, the region in which x is to be found is divided by 2:

$$2^k \to 2^{k-1} \to \dots \to 1$$

Hence, after at most k steps, we know if x is not in the list, or we have located it.

Therefore, in the worst case we need $3 * \log_2(n)$ operations, i.e. the worst case complexity is $O(\log_2 n)$.

Example: to find a name in a phone book containing 250 million entries requires at most 28 steps.

4. Sorting

Ordering elements in a list is a very common problem. There are many, many methods to solve this problem. We will look at one of the simplest, the insertion sort method.

How does it work? Think of arranging cards in increasing order. Take one card after the other and insert in stack of cards already ordered. Repeat until all cards have been ordered.

Example: 2, 4, 1, 3, 8, 5

Steps:

1) Start with 2 assumed to be correctly placed.

[2], 4, 1, 3, 8, 5

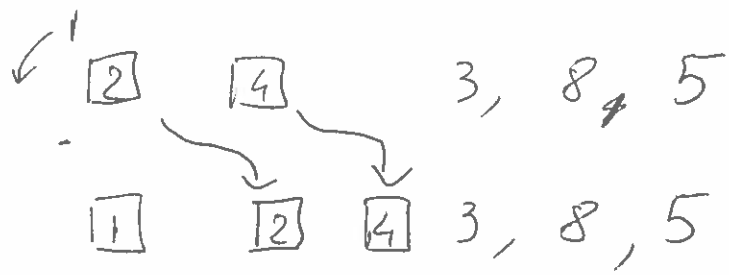
2) Move to second position:

Compare 2 with 4; order is correct so no changes

[2] [4] 1, 3, 8, 5

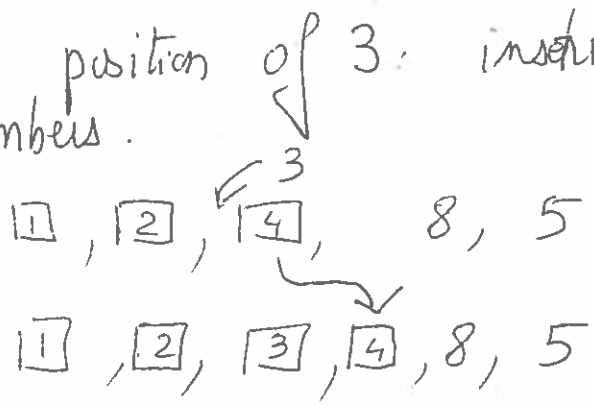
3) Third position

Find position of 1 in ordered list. Insert 1 and shift other numbers



4) Fourth position

Find position of 3. insert and shift other numbers.

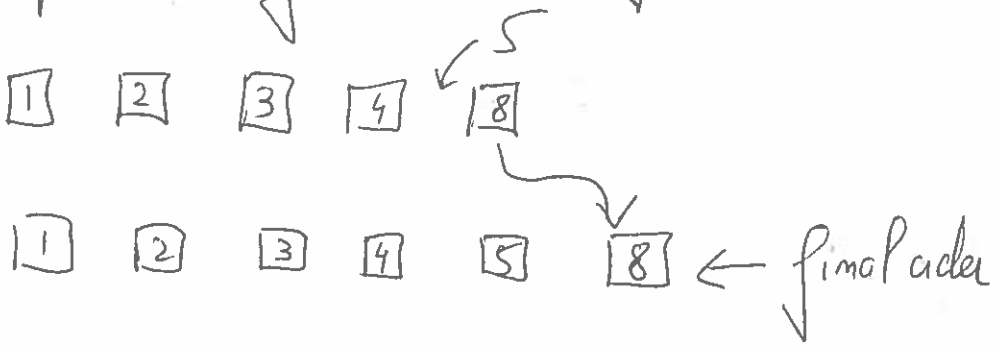


5) Fifth position

Find position of 8. Here, order is correct

6) Sixth position

Find position of 5 and shift other numbers



Pseudo code for Insertion sort

Procedure InsertionSort (N (integer)
$a_1, \ldots a_N$ (integers))

```
#
# Initialise Loop: from 2 to N
#
   For (i = 2; i ≤ N; Step = 1)
   {

#
# Store a local copy of current number a_i
#
        copy ← a_i

#
# Locate position to insert number a_i in the ordered
# numbers 1 ... a_i-1 : use linear search, stop
# when a number greater than a_i is found
#
        IP ← 1 ; J ← 1
        while ((a_i > a_j) ∧ (j < i))
        {
              IP ← IP + 1
              J ← J + 1
        }
#
# Now IP indicates where a_i should be inserted. First,
# move up all numbers from a_IP to a_i-1 by 1:
        For (j = i-1; j ≥ IP; Step = -1)
        {
              a_j+1 ← a_j
        }
#
# Insert a_i at position IP
        a_IP ← copy
#
# end loop
   }
```

Wast case complexity of insertion sort:

$i = 2$	2 comparisons
$i = 3$	4 comparisons
$i = 4$	6 comparisons
⋮	
$i = N$	$2(N-1)$ comparisons

total: $2(1+2+\dots N-1) = N(N-1)$

Insertion sort is of order N^2

5. Understanding complexity of algorithms.

$O(1)$	constant
$O(\log(n))$	Logarithmic
$O(n)$	Linear
$O(n^b)$	Polynomial
$O(b^n)$	Exponential

A problem solvable with an algorithm of wast case complexity that is polynomial is called tractable, otherwise it is intractable.

Types of intractable problems:

- unsolvable
- intractable, but with a solution that can be checked in polynomial time: class NP (non deterministic polynomial)

NP- complete: Class of problems with equivalent complexity. If one of them is proven to be polynomial, all are polynomial.

One example of NP - complete problem:

Given a finite set S of integers, determine whether any non empty subset A of S has its elements that sum to 0.

A supposed answer is very easy to verify, but no one knows a way to solve this problem other than testing every single possible subset.