How to Analyse Algorithms ?

As an algorithm is executed, it uses the computer's central processing unit to perform operations and its memory to hold the program and data

Analysis of algorithm or **performance evaluation** refers to the task of determining how much computing time or storage an algorithm requires

Two major phases of performance evaluation

- A priori estimates- performance analysis
- A posteriori testing – performance measurements

How to describe algorithms?

Pseudocode conventions

1. Comments begin with //
2. Blocks are indicated by braces **{ }**
3. Variables and their types are not declared- clear from context
4. Assignment statement **variable = expression**
5. Arithmetic operators **<, ≤ , >, ≥, ≠, =** and logical **and, or , not**
6. Loops **while (condition do { statements }**

for variable = value1 to value2 do { statements }

for variable = value1 downto value2 do { statements }

repeat { statements } until condition

7 Branching

if condition then statement1 else statement2

case

{ condition 1: statement 1

…………………………..

else : statement n+!

}

8 Input and output by **read value** and **write value**

9. Procedure

Algorithm Name (parameter list)

{ body

}

Algorithm comparison

A posteriori- implement and compare actual running times on different inputs

A priori- before implementing by counting number of expected steps

Algorithms are written to solve problems
Problem – sorting
Problem instance – sorting the array of size 10 containing values

5 112 23 67 12 45 11 3 25 54

Time and space requirements vary from one problem instance to another

The running time depends on input size

Usual measure for input size is number of inputs ex array size n in case of sorting problem

For some other problems input size can be length of the input such as prime number problem where input is only one number

As input size(n) grows , time taken increases

Order of growth of running time is more important

	Algorithm 1	Algorithm 2	Algorithm 3
Input size	step count	step count	step count
5	10	5	1
10	200	2000	8000
1000	2000	200000	800000
	2n	$n^2/5$	$n^3/125$
Is like	n	n^2	n^3

Space Complexity

The space complexity of an algorithm is the amount of memory it needs to run to completion

Space required by a program = code space (instructions reside in memory + data space(space taken by variables and constants) + stack space

The space required has two components

Fixed component – it is independent of instance characteristics

It includes instruction space, space required by simple variables, aggregate variables and constants

Variable component – It depends on instance characteristics such as size of the input. It includes space needed by reference variables and recursion stack

The space required S(P) of any algorithm P may therefore be written as

$S(P) = C + S_P(\text{instance characteristics}) = C + S_P(n)$

Where C is a constant and n is the input size

```
Algorithm Sum(a,n)
{
    s=0
    for i=1 to n do
        s= s + a[i]
    return s
}
```

Space required= code space + space required by variables

$S_P(n)$ = space required by variables

$S_P(n)$= n (for a)+ 1 (for n) + 1 (for s) + 1 (for i) + 2(for 0 and 1)

$S_P(n) = n + 5$

$S_P(n) \geq n$

```
Algorithm RSum(a,n)  //recursive
 {
      if n ≤  0 then return 0;
    else return RSum(a,n-1) + a[n]
  }
```

Space required= code space + space required by variables + stack space

Recursion Stack Space includes space for formal parameters, local variables and return address = 1 (for n) + 1 (for pointer to a) + 1 (for return address)

The depth of recursion is n+1

$S_P(n) \geq 3(n+1)$

Recursive algorithms have very high space complexity

Time Complexity

The time complexity of an algorithm is the amount of computer time it needs to run to completion

It is the sum of the compile time(fixed component) + run or execution time (depends on instance characteristics)

The execution time of a statement depends on type of operations involved ,values , the type of machine and the type of environment in which the statement is executed

For simplicity, we assume a constant amount of time is required to execute each simple statement of our pseudocode

A loop involves condition checking that will take constant amount of time multiplied by total number of time the loop is executed

Each line will be treated as a program step and execution time will be total step count

Time Complexity of Insertion sort

Algorithm InsertionSort(a,n)	cost	times
{		
for j = 2 to n do	c1	n
rec = A[j]	c2	n-1
// insert rec in proper position	0	
i = j - 1	c3	n-1
while i > 0 and A[i] .key > rec. key do	c4	$\sum_{j=2}^{n} tj$
{		
A[i + 1] = A[i]	c5	$\sum_{j=2}^{n} tj -1$
// decrement		
i = i - 1	c6	$\sum_{j=2}^{n} tj -1$
}		
A[i+ 1] = rec	c7	n-1
}		

Time Complexity of Insertion sort

$$T(n)=c1n+c2(n-1)+c4(n-1)+c5\sum_{j=2}^{n} tj +c6\sum_{j=2}^{n}(tj -1)+c7\sum_{j=2}^{n}(tj -1)+c8(n-1)$$

Even for inputs of a given size n, an algorithm's running time may depend on other instance characteristics as the input values

Best Case

The array is already sorted

For each j, A[i].key is less than or equal to rec.key thus while loop will end immediately   tj=1

T(n)=c1n+c2(n-1)+c4(n-1)+c5**(n-1) +** c8(n-1)

=(c1+c2+c4+c5+c8)n-(c2+c4+c5+c8)

=an +b

linear function of n

Worst Case

The array is sorted in reverse order

For each j, while loop will execute j times

t_j=j $\sum t_j$ = n(n+1)/2 -1 and

$\sum t_j$ **-1** = n(n+1)/2 -1 –(n-1)= n(n-1)/2

T(n)=c1n+c2(n-1)+c4(n-1)+c5**(**n(n+1)/2 -1)+

c6 n(n-1)/2+c7 n(n-1)/2 + c8(n-1)

=((c5+c6+c7)/2) n^2+

(c1+c2+c4- (c5+c6+c7)/2 +c8)n-(c2+c4+c5+c8)

=an^2 +bn +c

Quadratic function of n

The worst case running time will be taken as the running time of the algorithm

- Algorithm will not take longer than worst case
- It acts like an upper bound
- For most algorithms worst case occurs fairly often and is same as average case running time

Asymptotic Notation

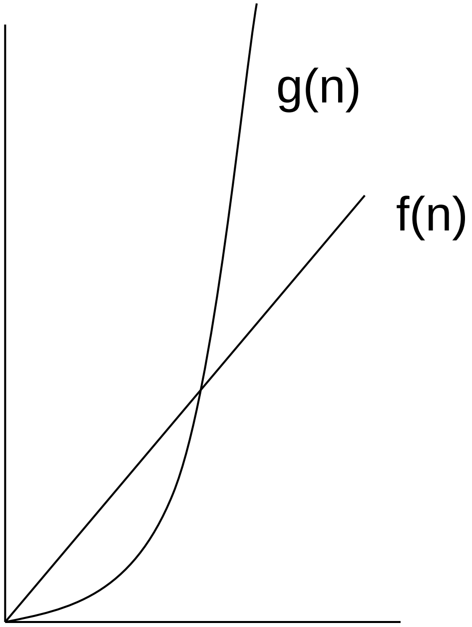
O-notation is used to define an asymptotic upper bound

Defn – A function f(n) is a member of O(g(n)) or we write f(n)= O(g(n)) or say f(n) is of O(g(n))

If there exists a positive constant c and n_0 such that

$0 \leq f(n) \leq c\, g(n)$ for all $n \geq n_0$

0(g(n))= { f(n) : there exists a positive constant c and n_0 such that $0 \leq f(n) \leq c\, g(n)$ for all $n \geq n_0$ }



- $f(n) = n$ Show that $f(n)$ is of $O(n^2)$

$n \le n^2$ for all n

$n \le cn^2$ for all $n \ge n_0$ where c=1 and $n_0 = 1$

- Show that $2n^2 - 3n$ is $O(n^2)$

$2n^2 - 3n \le c\,n^2$

$2 - 3/n \le c$ choose c=2 and $n_0 = 1$

$2n^2 - 3n \le cn^2$ for all $n \ge n_0$ where c=2 and $n_0 = 1$

- Show that $3n^2 + 5n$ is $O(n^2)$

$3n^2 + 5n \le c\,n^2$

$3 + 5/n \le c$ choose c=4 and $n_0 = 5$

$3n^2 + 5n \le cn^2$ for all $n \ge n_0$ where c=4 and $n_0 = 5$

- Is $2^{n+1} = 0(2^n)$?

$2^{n+1} = 2^n \times 2$ for all n

$2^{n+1} \le c\,2^n$ for all $n \ge n_0$ where c = 2 and $n_0 = 1$

- Is $2^{2n} = O(2^n)$?

$2^{2n} = (2^2)^n = 4^n > 2^n$ for all n

2^{2n} is not of $O(2^n)$

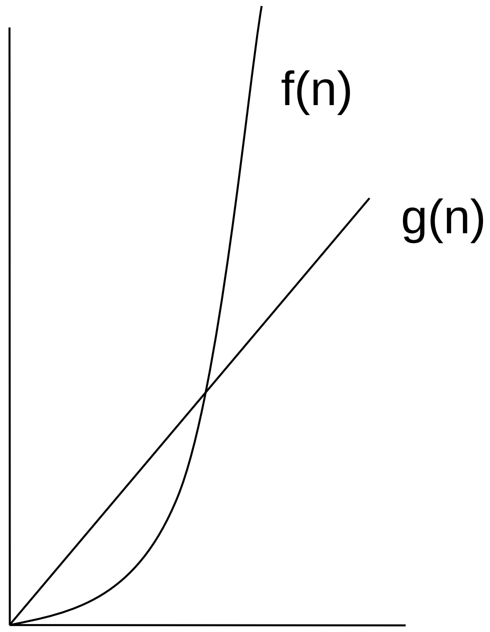
- If $f(n) = a_m n^m + \ldots\ldots + a_1 n + a_0$, then $f(n) = O(n^m)$

$$
\begin{aligned}
f(n) &= a_m n^m + \ldots\ldots + a_1 n + a_0 \\
&\le |a_m| n^m + \ldots\ldots + |a_1| n + |a_0| \\
&\le n^m \sum_{i=0}^{n} |a_i| n^{i-m} \\
&\le n^m \sum_{i=0}^{n} |a_i| \qquad \text{for } n \ge 1
\end{aligned}
$$

For the statement f(n) =O(g(n)) to be informative, g(n) should be as small a function of n one can come up with for which f(n)=O(g(n)))

We never say $3n+2=O(n^2)$ or $3n+2=O(3^n)$ though it is correct but we say $3n+2 = O(n)$

Ω-notation is used to define an asymptotic lower bound



Defn – A function f(n) is a member of Ω(g(n)) or we write f(n)= Ω(g(n)) or say f(n) is of Ω(g(n))

If there exists a positive constant c and n_0 such that

$0 \le c\, g(n) \le f(n)$ for all $n \ge n_0$

$\Omega(g(n)) = \{ f(n)$: there exists a positive constant c and n_0 such that $0 \le c\, g(n) \le f(n)$ for all $n \ge n_0 \}$

O-notation is used to describe an upper bound for the worst case running time of an algorithm

Ω-notation is used to describe a lower bound for the best case running time of an algorithm

Insertion sort is $O(n^2)$ and is $\Omega(n)$

There is separate notation for asymptotically tight bounds

θ-notation is used to define an asymptotic tight bound

Defn – A function f(n) is a member of θ(g(n)) or we write f(n)= θ(g(n)) or say f(n) is of θ(g(n))

If there exists a positive constant c and n_0 such that

$0 \le c1\ g(n) \le f(n) \le c2\ g(n)$ for all $n \ge n_0$

θ(g(n))= { f(n) : there exists a positive constant c and n_0 such that $0 \le c1\ g(n) \le f(n) \le c2\ g(n)$ for all $n \ge n_0$ }

f(n) is equal to g(n) within a constant factor

The function f(n) must be nonnegative

- Show that algorithm to find the maximum in an array of size n is θ(n)

```
Algorithm max(A,n)
{         max=A[1]                 C1   1
               i=2                 C2   1
   while i <= n  do                C3   n
         if A[i] > max             C4   n-1
              max= A[i]            C5   t
   return max
}
```

Best case time is when max lies in first place t=0

$T(n) = c1+c2+c3n+c4(n-1)= \Omega(n)$

Worst case time occurs when array is sorted t=n

$T(n) = c1+c2+c3n+c4(n-1) +c5(n-1)= O(n)$

Max algorithm is θ(n)

- The running time of an algorithm is $\theta(g(n))$ if and only if its worst case running time is $O(g(n))$ and its best case running time is $\Omega(g(n))$

- Let $f(n)$ and $g(n)$ be asymptotically nonnegative functions prove that $\max(f(n), g(n)) = \theta(f(n) + g(n))$

$f(n) \le \max(f(n),g(n))$ and $g(n) \le \max(f(n),g(n))$

$f(n) + g(n) \le 2\max(f(n), g(n))$

If $f(n)$ is the maximum then

$f(n) + g(n) \ge f(n) = \max(f(n),g(n))$

If $g(n)$ is the maximum then

$f(n) + g(n) \ge g(n) = \max(f(n),g(n))$

Thus $\frac{1}{2}(f(n) + g(n)) \le \max(f(n),g(n) \le f(n) + g(n)$

$c_1 f(n) + g(n)) \le \max(f(n),g(n) \le c_2(f(n) + g(n))$

for all $n > 1$ where $c_1 = \frac{1}{2}$ and $c_2 = 1$

o-notation is used to define an asymptotic upper bound which is not asymptotically tight

Defn – A function f(n) is a member of o(g(n)) or we write f(n)= o(g(n)) or say f(n) is of (Little “oh”) o(g(n))

If for any positive constant $c > 0$, there exist a constant $n_0 > 0$ such that

$0 \leq f(n) < c\, g(n)$ for all $n \geq n_0$

$2n=O(n^2)$ $2n=o(n^2)$ $2n^2 =O(n^2)$ but $2n^2 \neq o(n^2)$

$$\lim_{n \to \infty} f(n) / g(n) = 0$$

ω-notation is used to define an asymptotic lower bound which is not asymptotically tight

Defn – A function $f(n)$ is a member of $\omega(g(n))$ or we write $f(n)= \omega(g(n))$ or say $f(n)$ is of (little omega) $\omega(g(n))$

If for any positive constant $c > 0$, there exist a constant $n0 > 0$ such that

$0 \le c\, g(n) < f(n)$ for all $n \ge n0$

$\lim_{n \to \infty} f(n) / g(n) = \infty$ or $\lim_{n \to \infty} g(n)/f(n) = 0$

There is anology between asymptotic notation and usual comparisons

$O \approx \le$ upper bound $o \approx <$ strict upper bound

$\Omega \approx \ge$ lower bound $\omega \approx >$ strict lower bound

$\theta \approx =$ tight bound

$\theta(n)$ = any linear function $\theta(n^2)$= any quadratic function

Asymptotic notation satisfy following properties

Transitivity

All asymptotic notations O, o, Ω, θ , ω are transitive

Reflexivity O, Ω, θ are reflexive

Symmetry θ is symmetric

Transpose symmetry

f(n) = O(g(n)) if and only if g(n) = Ω (f(n)

f(n) = o(g(n)) if and only if g(n) = ω (f(n)

Trichotomy is not true i.e for any two functions f(n) and g(n) neither f(n) = O(g(n)) nor f(n) = Ω (g(n) holds

- Consider the following functions of n

$f_1(n) = n^2$

$f_2(n) = n^2 + 1000n$

$f_3(n) =$ n if n is odd

n^3 if n is even

$f_4(n) = n$ if $n \leq 100$

n^3 if $n > 100$

Find i and j such that $f_i(n)$ is $O(f_j(n))$

and or $f_i(n)$ is $\Omega(f_j(n))$

Note that $f_2(n)$ is neither $O(f_3(n))$ nor Ω $(f(_3(n))$

The above example shows that Trichotomy is not true for asymptotic notation

The rates of growth of polynomials and exponentials can be related by the following .

For all constants a and b , $a > 1$

$n^b = O(a^n)$

Any positive exponential function grows faster than any polynomial

$\log_b n = (\log n)b = o(n^a)$ for any constant $a > 0$

Any positive polynomial function grows faster than any polylogarithmic function

Using Stirling's approximation following can be prooved

$n! = o(n^n)$ - n^n grows faster than factorial

$n! = O(2^n)$ - factorial grows faster than exponential

- Order the following functions by growth rate and justify

a) $2n$ b) $\sqrt{n}$ c) $\log n$ d) $\log(\log n)$ e) $\log_2 n$

f) $n / \log n$ g) $\sqrt{n} \log n$ h) $(1/3)^n$ i) $(3/2)^n$

j) 17 k) $n!$ l) n^4 m) n^n

17 is bounded above by constant always irrespective of n it is O(1)

$(1/3)^n \to 0$ as n increases , it is bounded above by 1/3

And bounded below by 0 it is O(1)

log (logn) grows slower than logn

log n i.e $\log_e n$ grows slower than $\log_2 n$

$\sqrt{n}$ grows slower than 2n

$\sqrt{n}$ grows slower than $\sqrt{n} \log n$

$$\lim \sqrt{n} \log n/2n = \lim \log n/\sqrt{n} = \lim (1/n)/(1/2\sqrt{n})$$

$$= \lim 1/\sqrt{n} = 0$$

$\sqrt{n} \log_2 n$ grows slower than 2n

2n grows slower than $(3/2)^n$, $(3/2)^n$ grows slower than n!

n! grows slower than n^n

Given any problem we look for best complexity algorithm

Best algorithm would be of constant order

Irrespective of input size it always takes same time

The algorithm is of O(1)

searching problem

Given an array containing n elements we are looking for an integer x

There are two instance characteristics on which the complexity of algorithm will depend

n-number of integers and

x –the number to be searched

1 Sequential search

```
 Algorithm SeqSearch(A, n, x)
 {           i=1

while i <= n

        {    if( A[i] =x ) then  return i

                  i=i+1         }

          return -1

}
```

Best case – searching for first element - $O(1)$

Worst case- searching for element not present – $O(n)$

Worst case occurs more often-

average running time of sequential search is $O(n)$

2 Sequential search on sorted file

```
Algorithm SeqSearchOnSorted(A, n, x)
{           i=1

  while i <= n and A[i] <x

            i=i+1

      if ( A[i] =x) then return i

         return -1  }
```

Best case – searching for first element - O(1)

Worst case- searching for number greater than or equal to last element– O(n)

The time for other cases varies between 1 to n – almost all are equally likely

Average running time O(n/2)

3 Binary Search on sorted file

```
Algorithm  BinarySearch(A, m,n, x)
  {  if ( m=n) then
        { if( A[m]=x then
              return i
             else  return -1
             }
    else
{ mid=(m+n)/2
  if A[mid]=x  then return mid
   else if( A[mid] < x ) then
              return BinarySearch(A, mid+1,n,x)
           else return BinarySearch(A, m,mid-1,x)
} }
```

Best case when n=1 for any x – O(1)

Binary search suddenly reduces the file to half the size so next we are searching in a file of half the size. This repeated division quickly reduces the file size to 1

The time taken by the algorithm can be expressed by a recurrence relation

$$T(n) = \begin{cases} a & n=1 \\ T(n/2)+b & n>1 \end{cases}$$

$= T(n/2) + b$

$= T(n/4) + b + b$

$= T(n/4) + 2b$

$= T(n/8) + 3b$

$= T(n/2^k) + kb \qquad n=2^k$

$= T(1) + kb$

$= a + b \log_2 n$

$= O(\log_2 n)$

4 Ternary search- It is a modification on binary search- The file is divided into three parts and with two comparisons we will able to decide which one third part one should take up for further search

With two comparisons file size is reduced to one third and recurrence relation for running time is

$$T(n) = \begin{cases} a & n=1 \\ T(n/3) + 2b & n > 1 \end{cases}$$

$$= T(n/3) + 2b$$

$$= T(n/9) + 4b$$

$$= T(n/3^k) + 2kb \qquad n=3^k$$

$$= a + 2b \log_3 n$$

$$= O(2 \log_3 n) = O(\log_3 n)$$

$$2 \log_3 n = 2 \log_2 n / \log_2 3 = 2 \log_3 2 \log_2 n$$

$$= \log_3 4 \log_2 n$$

$$> \log_2 n$$

5 Hash search- The file is kept in hashed order

The record is kept at the place given by the hash function- number of records n is fixed

Given x- use hash function to compute address- go to the position and record is found

For any x- time is constant- time taken for computing address by hashing function- $O(1)$

Sequential search $O(n)$

Ternary Search $O(2 \log_3 n)$

Binary Search $O(\log_2 n)$

Hash search $O(1)$

Sorting problem- to sort an array of n records

1. Insertion sort Best case $O(n)$

Worst case $O(n^2)$

2. Heap sort

Heap data structure is an array that can be viewed as an almost complete binary tree

The binary tree is completely filled in all levels except possibly the lowest, which is filled from left up to a point

Heaps satisfy heap property

For Max heap every i other than the root , $A[parent[i]] \geq A[i]$ (parent of ith node is at [i/2]).

For Min heap every i other than the root , $A[parent[i]] \leq A[i]$

Shift down and Shift up are the routines used to maintain heap property

If the left and right subtrees of ith node are heaps but ith node is violating heap property then shift down pushes A[i] to correct position in the tree so that A[i] becomes a heap

Algorithm ShiftDown(A,i, n)

```
{ left=2 * i
right = 2*i+1
If left < n and A[left> A[i] then
  largest = left                // choose the largest as left
else largest = i        // or parent is the largest
If right < n and A[right> A[largest] then
  largest = right  //choose the largest as right
If largest ≠ i  then // if any child is greater
 exchange (A[i]) and A[largest])// swap
   shiftdown(A , largest) // recursive call for the affected node
}
```

Since a heap of n elements is almost complete binary tree, its height is θ(log n). The no of recursive calls depends on height hence shiftdown is O(logn)

To build a heap for a given array, we can use shiftdown(heapify) procedure in a bottom up manner.

The order should be such that the process guarantees that the children are already heaps.

Thus we start with last possible parent which is obviously at position n/2

Algorithm BuildHeap(A.n)

```
{ for i= n/2 downto 1
        shiftdown(A,i,n) }
```

The running time of this algorithm is O(nlogn)

Heapsort starts with building a heap.

The maximum element in first position is then swapped with the element in last position.

Since this disturbs the heap property, shiftdown is called on the array from which last element is removed as it is in correct position.

This is repeated till all elements are in correct positions

Algorithm Heapsort(A, n)

```
{           BuildHeap(A, n)
     for i= n downto 2
            { exchange( A[1], A[i])
             shiftdown( A, I, n-1)}
}
```

Heapsort takes time nlogn + (n-1) logn as there are n-1 calls to shiftdown which is of order logn

Heapsort is O(nlogn)

Priority Queue – It is a queue in which elements are inserted in any order but leave the queue in order of priority(the element with maximum value will leave first).

The queue need to be maintained with maximum element at the front and in sorted order

Delete operation removes front element – O(1)

Insert operation need to insert the element so that array is sorted and in worst case may require n comparisons-O(n)

Deletion is faster but insertion is slow. Deletion is not as fast as it looks as elements in an array will have to be shifted up(or use a circular queue)

Priority queue can be implemented as a heap

Deletion will remove the highest priority element at A[1]. This creates a hole at top which is filled with last element. This disturbs heap property at top and can be set right by using shift down

Deletion is O(logn)

Insertion can insert in the last position as that is the only place available. This disturbs heap property and can be set right using a shift up operation which is also of logn

Insertion is O(logn)

Priority queue implemented as a heap is a very efficient data structure in which both insertion and deletion are of O(logn)

```
Algorithm shiftUp( A,n)
{ i=n
  value = A[i]
While i >1 and A[i/2] < value do
       {
            A[i] = A[i/2]
                 i= i/2
         }
 A[i] =value
}
```

The running time of shiftUp is O(logn), since the path traced from the new leaf at n to the root is of length O(logn)

3. Counting sort- linear order sorting algorithm

It assumes that input elements are in the range 1 to k for an integer k where k is much less than n or k=O(n) (elements are repeated)

An array count of size k is used to store how many times each element occurs. This information is used to put elements in proper position. This requires an array b of same size as A to hold the sorted output.

Algorithm CountingSort(A,n)

```
{ for i=1 to k do
  count[i]=0                    // O(k)
for j= 1 to n do
   count(A[j])  =count (A[j]) + 1  // O(n)
```

```
for i=2 to k do
   c[i]=c[i]+c[i-1]              // O(k)
for j= n downto 1
   B[c[A[j]] =A[j]   // c[A[j]] gives the position where
              // A[j]  is to be  inserted in B
   c[A[j]]=c[A[j]]-1              // O(n)
```

Counting sort is O(n) . It has a very high space complexity

Two properties are important for sorting algorithms

1. In–place- same array is used to store sorted output
2. Stable-numbers with same value appear in same relative order as in input

Counting sort is stable but is not an In-place algorithm

Heap sort is not stable but is an In-place algorithm

Insertion sort is both stable and an in place algorithm

Radix sort

Each element in the array to be sorted has d digits where digit 1 is the lowest order digit while d^{th} digit is the highest order digit

Algorithm RadixSort(A,n,d)

{

 for i = 1 to d do

Sort (A, n, i)

// sort array A on digit i using any stable efficient sorting

  // algorithm

}

Since each digit is in the range 0 to 9 , counting sort is the obvious choice

Each sorting pass takes θ(n) time and there are d passes so the total time for radix sort is θ(dn) where d is constant

RadixSort runs in linear time i.e θ(n) but not In-place

Higher Order Algorithms

Matrix multiplication algorithm is $O(n^3)$

```
Algorithm Matrixmult( A, B, C, n)
{
    for i=1 to n do
    {  for j=1 to n do
          {   sum=0
                   for k=1 to n do
                      sum= sum+ A[i, k]* B[k, j]
               C[i, j] = sum
           }
     }
}
```

Consider the algorithm to generate n th Fibonacci

It can written both as a recursive algorithm as also an iterative algorithm

```
Algorithm RecurFibonacci (n)
{
        if n=1 return 0;
        if n=2 return 1;
        return RecurFibonacci(n-1) + RecurFibonacci(n-2)
}
```

The running time of this algorithm T(n) can be expressed as a recurrence relation

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

$$\begin{aligned} T(n) = T(n-1) + T(n-2) &\le 2\ T(n-1) \\ &\le 2\ (\ 2\ T(n-2)) = 2^2\ T(n-2) \\ &\le\ 2^k\ T(n-k) \\ &\le\ 2^k\ T(1) = 2^k\ c \qquad \text{when } n-k = 2 \\ &= 2^{(n+2)}\ c \qquad k=n+2 \\ &= O(2^n) \end{aligned}$$

Fibonacci algorithm is of exponential order

Recursive algorithms have very high space complexity

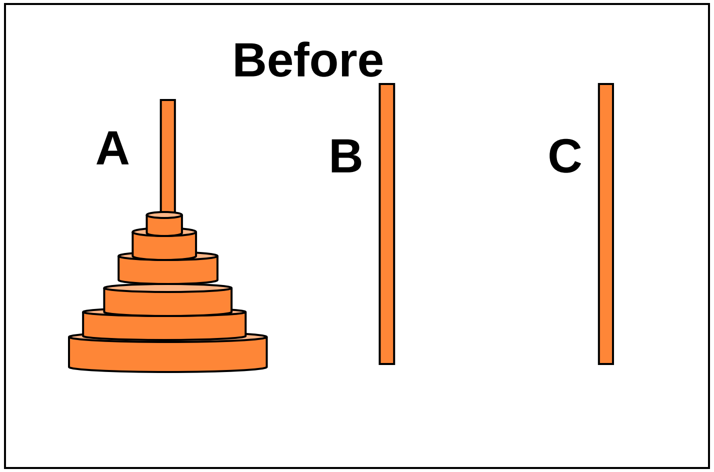
```
Algorithm IterativeFibonacci (n)
{
        if n=1 return 0;
        if n=2 return 1;
        f1=0
        f2 =1
  for i=3 to n do
       { f3 = f2+f1
         f1=f2
         f2=f3 }
 return f3
        }
```

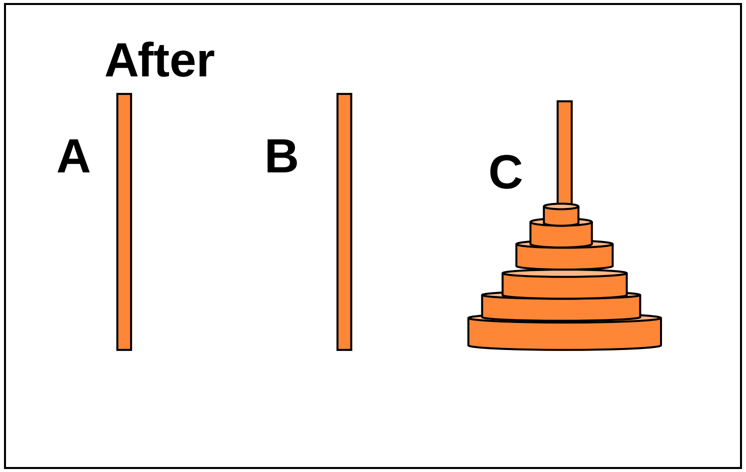
Each of the steps taking constant time are executed in worst case n-2 times

The running time of this algorithm is O(n-2)=O(n)

Note that n is not the input size but k=log n is the input size and in terms of k the algorithm is $O(2^k)$

Tower of Hanoi

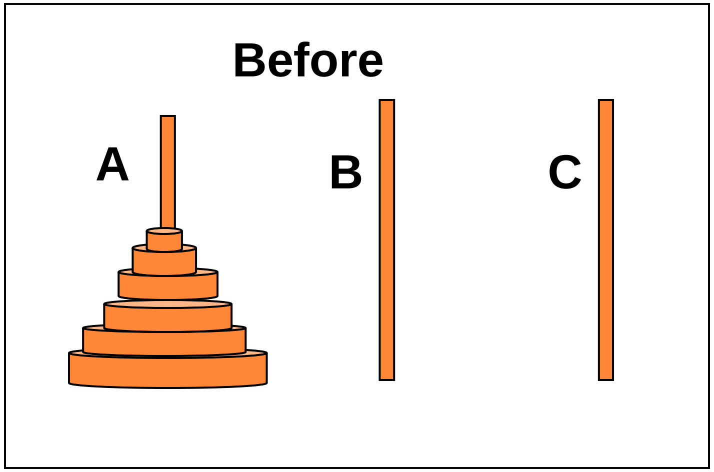


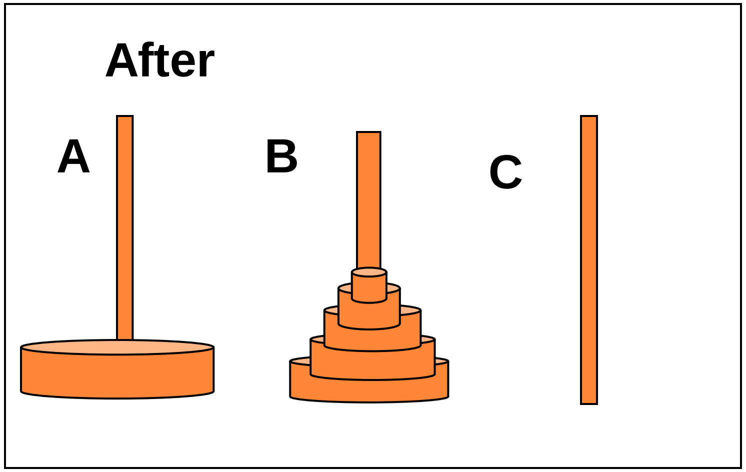


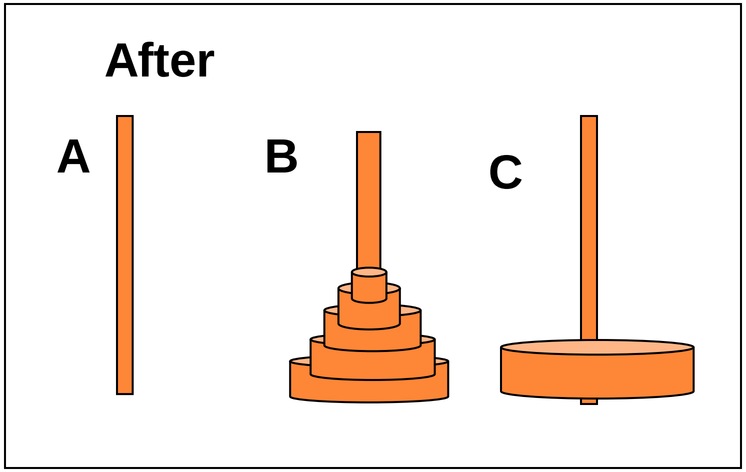
The disks of decreasing size are stacked on the tower in decreasing order of size bottom to top. The disks are to be moved from tower A to tower C using tower B for intermediate storage. The disks are heavy, they can be moved only one at a time. At no time can a disk be on top of a smaller disk

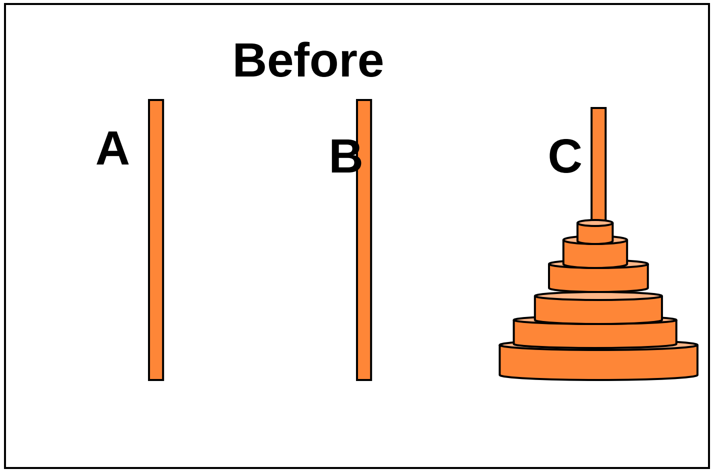
A very simple solution using recursion can be given as follows . First move n-1 disks to tower B using C. Move the bottom disk to C. Now shift n-1 disks from B to C using A

Tower of Hanoi









Algorithm TowerOfHanoi(n, A, B, C)
{// Moving n disks from A to C using B
    If n ≥ 1 then
        {
            TowerOfHanoi(n-1, A, C, B)
        // physically move top disk from tower A to C
 write(“Move top disk from”, A, “ to top of tower”,C)
            TowerOfHanoi(n-1,B, A, C)
        }
}

The recurrence relation for running time

$$T(n) = \begin{cases} a & \text{if } n < 1 \\ 2T(n-1) + c & \text{if } n \geq 1 \end{cases}$$

$$T(n) = 2T(n-1) + c$$
$$= 2\ 2T(n-2) + (2 +1)c$$
$$= 2^kT(n-k) +(2^k+\ldots.+2+1)c \qquad n-k=1$$
$$= 2^{(n-1)}a + c(2^n-1) = O(2^n)$$

Permutation problem- Given a set n 1 of elements , the problem is to print all possible permutations of this set

If the set is { a, b, c} The permutations are

{ {a,b,c},{a,c,b},{b,a,c}, {b,c,a}, {c,b,a}, {c,a,b}}

In case of four elements, the permutations area followed by all permutations of {b,c,d}

b followed by all permutations of {a,c,d}

c followed by all permutations of {a,b,d}

d followed by all permutations of {a,b,c}

The algorithm can be written recursively as if we know how to solve it for n-1 elements ,we can solve for n elements

Algorithm Perm (A,k,n)

{// A is the array containing n elements

If k=n then writeA(A) // writeA writes all n elements A
else

```
for i=k to n do
        { swap ( A[k], A[i]) // swap A[k] with A[i]
        Perm(A,k+1,n)
         swap(A[i],A[k]);
        } }
```

The recurrence relation for running time is

$$T(n) = 1 \quad \text{if } n=1$$

$$(n-1)(T(n-1) + b)$$

$$T(n) = n-1T(n-1) + b$$

$$= n-1((n-2)(T(n-2) + b) + b)$$

$$= (n-1)(n-2)\ldots.T(n-k) + b((n-1)+(n-1)(n-2)+ \ldots\ldots.+(n-1)(n-2)\ldots(n-k+1))$$

$$= (n-1)(n-2)\ldots.1 + b(((n-1)+..+(n-1)(n-2)..1) \quad n=k$$

$$\leq n-1! + b\, n((n-1)!) = O(n!)$$

Since there are n! permutations and it has to write all the permutations any algorithm to generate permutations is $\Omega(n!)$

THANK YOU!!!