# Asymptotic Analysis

All about algorithm performance

# Why performance analysis?

- An Algorithm should take care of features, like user friendliness, modularity, security, maintainability, etc.
- Performance is like currency through which we can buy all the above things.
- To summarize, performance == scale. Imagine a text editor that can load 1000 pages, but can spell check 1 page per minute OR an image editor that takes 1 hour to rotate your image 90 degrees left OR … you get it. If a software feature can not cope with the scale of tasks users need to perform – it is as good as dead.

# Which is better?

- Given two algorithms for a task, how do we find out which one is better?
- Okay. Implement both the algorithms and run the two programs on your computer and see which one takes less time.
- Will the evaluation be absolute?

# Problem with run time

- If the performance is based on run time, we see the following issue:

1) It might be possible that for some inputs, first algorithm performs better than the second. And for some inputs second performs better.
2) It might also be possible that for some inputs, first algorithm perform better on one machine and the second works better on other machine for some other inputs.

# Asymptotic Analysis

- Consider the following function:

$F(x) = 1/(x-2)$

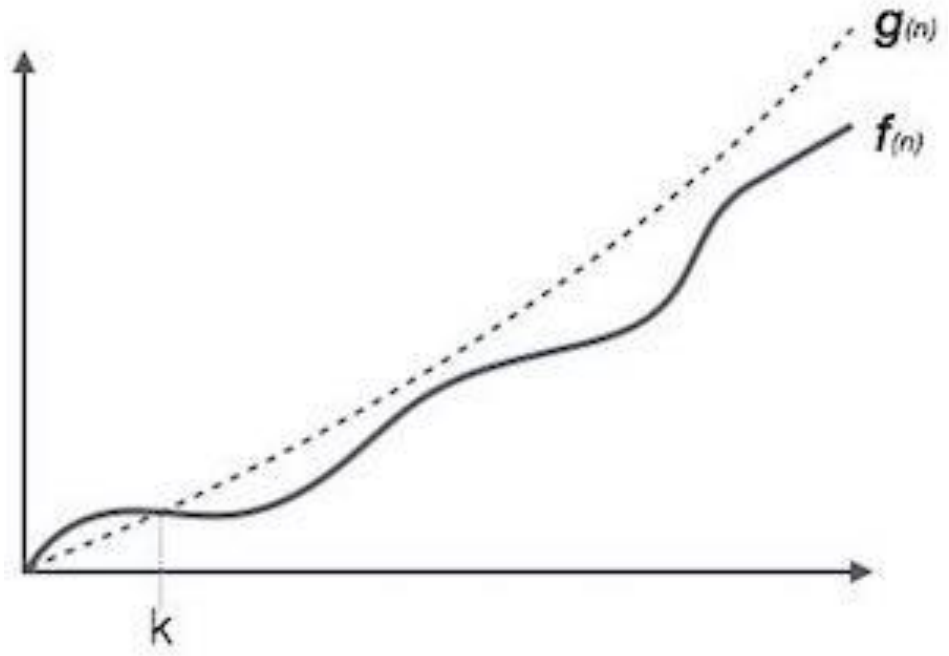
Desmos | Graphing Calculator

# How is it better?

- In Asymptotic Analysis, we evaluate the performance of an algorithm in terms of input size (we don’t measure the actual running time).
- We calculate, how the time (or space) taken by an algorithm increases with the input size.

# Big O

- O($f$(n)) = { $g$(n) : there exists c > 0 and $n_0$ such that $f$(n) ≤ c.$g$(n) for all n > $n_0$. }



# Example

Let $f(n) = 5.5n^2 - 7n$

We need to verify whether $f(n)$ is $O(n^2)$

Let c be a constant such that $5.5n^2 - 7n \leq c.n^2$

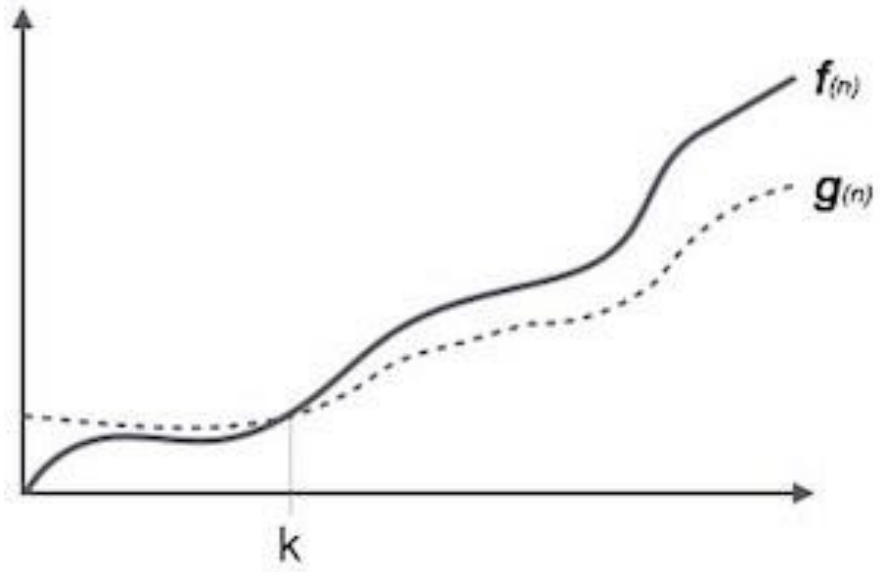
or $n \geq \frac{7}{c - 5.5}$

Fix c=9 to get $n \geq 2$

So, we can say that for c=9 and $n_0$=2, f(n) is $O(n^2)$

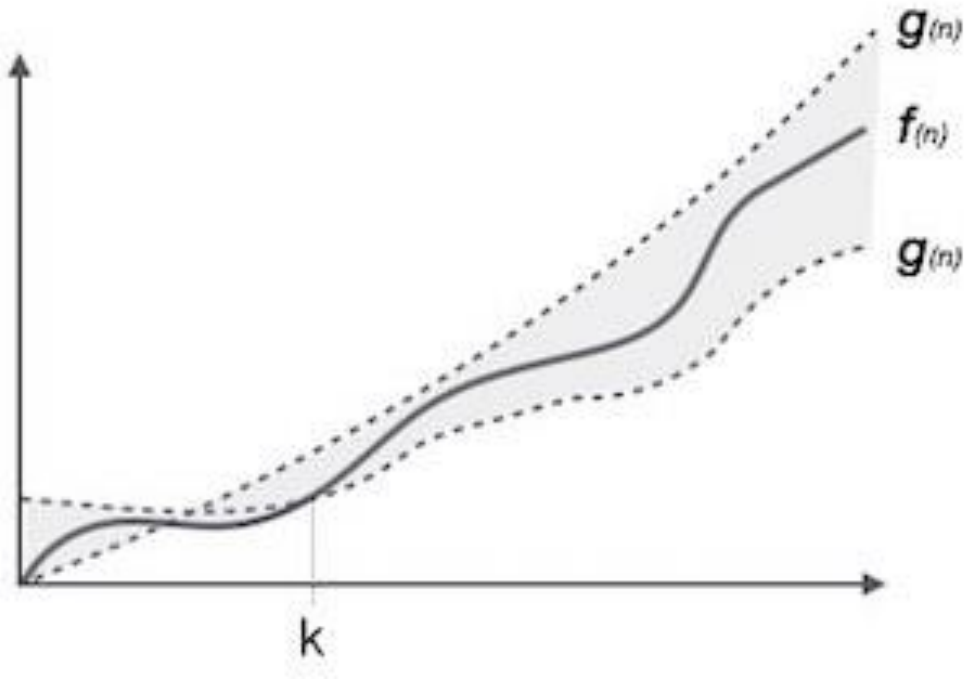
# Omega

- $\Omega(f(n)) \geq \{ g(n)$ : there exists $c > 0$ and $n_0$ such that $g(n) \leq c.f(n)$ for all $n > n_0$. $\}$



# Theta

- $\theta(f(n))$ = { $g(n)$ if and only if $g(n) = O(f(n))$ and $g(n) = \Omega(f(n))$ for all $n > n_0$. }



# Is it always better?

- Asymptotic Analysis is not perfect, but that's the best way available for analyzing algorithms.
- For example, say there are two sorting algorithms that take 1000nLogn and 2nLogn time respectively on a machine. Both of these algorithms are asymptotically same (order of growth is nLogn). So, With Asymptotic Analysis, we can't judge which one is better as we ignore constants in Asymptotic Analysis.

- Also, in Asymptotic analysis, we always talk about input sizes larger than a constant value. It might be possible that those large inputs are never given to your software and an algorithm which is asymptotically slower, always performs better for your particular situation. So, you may end up choosing an algorithm that is Asymptotically slower but faster for your software.

# Question

```
int a = 0, b = 0;
for (i = 0; i < N; i++) {
    a = a + rand();
}
for (j = 0; j < M; j++) {
    b = b + rand();
}
```

# Question

```
int a = 0;
for (i = 0; i < N; i++) {
    for (j = N; j > i; j--) {
        a = a + i + j;
    }
}
```

# Question

```
int i, j, k = 0;
for (i = n / 2; i <= n; i++) {
    for (j = 2; j <= n; j = j * 2) {
        k = k + n / 2;
    }
}
```

# Question

```
int a = 0, i = N;
while (i > 0) {
    a += i;
    i /= 2;
}
```

# Question

```
void fun(int n)
{
  for(int i=0;i*i<n;i++)
    cout<<"GeeksforGeeks";
}
```

# Solution

The loop will stop when i * i >=n

i.e., i*i=n

$i*i=n \Rightarrow k^2 = n$

$k = \sqrt{n}$

Hence, the time complexity is $O(\sqrt{n})$.

# Question

```
void fun(int n)
{
    if (n < 5)
        cout << "GeeksforGeeks";
    else {
        for (int i = 0; i < n; i++) {
            cout << i;
        }
    }
}
```

# Question

```
void fun(int a, int b)
{
    while (a != b) {
        if (a > b)
            a = a - b;
        else
            b = b - a;
    }
}
```

# Linear Search

```
#include <stdio.h>
int search(int arr[], int size, int x) {
    int i=0;
    for (i=0; i<size; i++)
        if (arr[i] == x)
            return i;
    return -1;
}

int main() {
    int arr[] = {2,3,1,5};
    int size = sizeof(arr) / sizeof(arr[0]);
    int find = 1;
    printf("Position of %d is %d\n", find, search(arr,size,find));
    return 0;
}
```

# Worst Case

The worst case will take place if:

1. The element to be search is in the last index
2. The element to be search is not present in the list

In both cases, the maximum number of comparisons take place in Linear Search which is equal to N comparisons.

Hence, the Worst Case Time Complexity of Linear Search is O(N).

# Best Case

The Best Case will take place if: The element to be search is on the first index.

The number of comparisons in this case is 1. Therefore, Best Case Time Complexity of Linear Search is O(1).

# Average Case

Let there be N distinct numbers: a1, a2, ..., a(N-1), aN.

We need to find element P.

There are two cases:

Case 1: The element P can be in N distinct indexes from 0 to N-1.

Case 2: There will be a case when the element P is not present in the list.

There are N case 1 and 1 case 2. So, there are N+1 distinct cases to consider in total.

# Average Case: Contd

If element P is in index K, then Linear Search will do K+1 comparisons.

Number of comparisons for all cases in case 1 = Comparisons if element is in index 0 + Comparisons if element is in index 1 + ... + Comparisons if element is in index N-1
= 1 + 2 + ... + N
= N * (N+1) / 2 comparisons

If element P is not in the list, then Linear Search will do N comparisons.

Number of comparisons for all cases in case 2 = N

Therefore, total number of comparisons for all N+1 cases = N * (N+1) / 2 + N

= N * ((N+1)/2 + 1)

Average number of comparisons = ( N * ((N+1)/2 + 1) ) / (N+1)

= N/2 + N/(N+1)

# Space Complexity

As the amount of extra data in Linear Search is fixed, the Space Complexity is O(1).

# Binary Search

```
int binarySearch(int[] A, int x) {
    int low = 0, high = A.length - 1;
    while (low <= high) {
        int mid = (low + high) / 2;
        if (x == A[mid]) {
            return mid;
        }
        else if (x < A[mid]) {
            high = mid - 1;
        }
        else {
            low = mid + 1;
        }
    }
    return -1;
}
```

# Worst Case

The worst case of Binary Search occurs when:
The element is to search is in the first index or last index

In this case, the total number of comparisons required is logN comparisons.

Therefore, Worst Case Time Complexity of Binary Search is O(logN).

# Best Case

The best case of Binary Search occurs when: The element to be search is in the middle of the list

In this case, the element is found in the first step itself and this involves 1 comparison.

Therefore, Best Case Time Complexity of Binary Search is O(1).

# Average Case

Let there be N distinct numbers: a1, a2, ..., a(N-1), aN. We need to find element P.

There are two cases:

Case 1: The element P can be in N distinct indexes from 0 to N-1.

Case 2: There will be a case when the element P is not present in the list.

There are N case 1 and 1 case 2. So, there are N+1 distinct cases to consider in total.

# Average Case Contd.

The element at index N/2 can be found in 1 comparison as Binary Search starts from middle.

Similarly, in the 2nd comparisons, elements at index N/4 and 3N/4 are compared based on the result of 1st comparison.

On this line, in the 3rd comparison, elements at index N/8, 3N/8, 5N/8, 7N/8 are compared based on the result of 2nd comparison.

Based on this, we know that:

- Elements requiring 1 comparison: 1
- Elements requiring 2 comparisons: 2
- Elements requiring 3 comparisons: 4

Therefore, Elements requiring I comparisons: 2^(I-1)

# Average Case Contd.

The maximum number of comparisons = Number of times N is divided by 2 so that result is 1 = Comparisons to reach 1st element = logN comparisons

I can vary from 0 to logN

Total number of comparisons = 1 * (Elements requiring 1 comparison) + 2 * (Elements requiring 2 comparisons) + ... + logN * (Elements requiring logN comparisons)

Total number of comparisons = 1 * (1) + 2 * (2) + 3 * (4) + ... + logN * (2^(logN-1))

= 1 + 4 + 12 + 32 + ... = 2^logN * (logN - 1) + 1

= N * (logN - 1) + 1

Total number of cases = N+1

Therefore, average number of comparisons = ( N * (logN - 1) + 1 ) / (N+1)

The dominant term is N * logN / (N+1) which is approximately logN. Therefore, Average Case Time Complexity of Binary Search is O(logN).

# Selection Sort

```
void selectionSort(int numbers[], int array_size) {
   int i, j, T, min, count;
   for (i = 0; i < array_size; i++) {
      min = i;
      for (j = i + 1; j < array_size; j++) {
         if (numbers[j] < numbers[min]) {
            min = j; }
      }
      T = numbers[min];
      numbers[min] = numbers[i];
      numbers[i] = T;
    }
}
```

# Worst Case

The worst case is the case when the array is already sorted (with one swap) but the smallest element is the last element.

For example, if the sorted number as a1, a2, ..., aN, then:

a2, a3, ..., aN, a1

will be the worst case for our particular implementation of Selection Sort.

# Worst Case Contd.

Hence, the worst case has:

- N * (N+1) / 2 comparisons
- N swaps

Hence, the time complexity is O(N^2).

# Best Case

The best case is the case when the array is already sorted.

For example, if the sorted number as a1, a2, ..., aN, then:

a1, a2, a3, ..., aN

will be the best case for our particular implementation of Selection Sort.

# Best Case Contd.

Hence, the best case has:

- N * (N+1) / 2 comparisons
- 0 swaps

Note only the number of swaps has changed.
Hence, the time complexity is O(N^2).

# Average Case

Number of comparisons = N * (N+1) / 2

Therefore, the time complexity will be O(N^2).

# Bubble Sort

```
void bubbleSort (int S[ ],  int length) {
      bool isSorted = false;
      while(!isSorted)
      {
            isSorted = true;
            for(int i = 0; i<length; i++)
            {
               if(S[i] > S[i+1])
               {
                        int temp = S[i];
                        S[i] = S[i+1];
                        S[i+1] = temp;
                        isSorted = false;
                }
            }
            length--;
      }
}
```

# Worst Case

- No of comparisons: $\frac{1}{2}(n^2 - n)$
- $O(N^2)$

# Best Case

- If the numbers are already sorted in ascending order, the algorithm will determine in the first iteration that no number pairs need to be swapped and will then terminate immediately.
- The algorithm must perform n-1 comparisons; therefore:
- Complexity: Omega(N)

# Average Case

- The average time complexity of Bubble Sort case is: *Theta($n^2$)*

# Insertion Sort

```
void insertionSort(int array[], int length)
{
   int i, j, value;
   for(i = 1; i < length; i++)
    {
        value = a[i];
        for (j = i - 1; j >= 0 && a[ j ] > value; j--)
        {
            a[j + 1] = a[ j ];
        }
        a[j + 1] = value;
    }
}
```

# Worst Case

- The worst-case time complexity of Insertion Sort is: $O(n^2)$

# Best Case

- The best-case time complexity of Insertion Sort is: *Omega(n)*

# Average Case

- The average-case time complexity of Insertion Sort is: Theta$(n^2)$

# Till Now

| Algorithm | Worst Case | Average Case | Best Case | Space Complexity |
|---|---|---|---|---|
| Linear Search | **$O(N)$** | **$\Theta(N)$** | **$\Omega(1)$** | **$O(1)$** |
| Binary Search | **$O(\log N)$** | **$\Theta(\log N)$** | **$\Omega(1)$** | **$O(1)$** |
| Selection Sort | **$O(N^2)$** | **$\Theta(N^2)$** | **$\Omega(N^2)$** | **$O(1)$** |
| Bubble Sort | **$O(N^2)$** | **$\Theta(N^2)$** | **$\Omega(N)$** | **$O(1)$** |
| Insertion Sort | **$O(N^2)$** | **$\Theta(N^2)$** | **$\Omega(N)$** | **$O(1)$** |

# Recurrence Relation

- $T(n)=1$ for $n=0$
- $T(n)=1+T(n-1)$ for $n>0$

# Recurrence Relation

- $T(n)=1+T(n/2)$ for $n>0$
- $T(n) = 1$ for $n=1$

# Recurrence Relation

- $a_n = 2\,a_{n-1} - 1$ and $a_1 = 3$, Solve the recurrence relation.

# Recurrence Relation

- $a_n = 5\ a_{n-1} - 1$ and $a_1 = 3$, Solve the recurrence relation.

The above recurrence relation can be written as:

T(n) = 5 T(n-1) – 1. Now solve as per the usual method.

# Characteristic Root Technique

- Suppose we want to solve a recurrence relation expressed as a combination of the two previous terms, such as $a_n = a_{n-1} + 6a_{n-2}$.
- The characteristic equation in this case is:
  $x^2 - x - 6 = 0$

If the roots of the equation are distinct, then

$a_n = ar_1^n + br_2^n$

where a and b are constants defined by initial condition.

Why?

http://discrete.openmathbooks.org/dmoi2/sec_recurrence.html

# Characteristic Root Technique

If the roots of the equation are same, then

$a_n = ar^n + bnr^n$

where a and b are constants defined by initial condition.

# Characteristic Root Technique

- $a_n = 6 * a_{n-1} - 9 * a_{n-2}$
- $a_0 = 1$ and $a_1 = 4$.

# Exercise

- Solve the recurrence relation $a_n=a_{n-1}+2^n$ with $a_0=5$.

# Master's Theorem

- Dividing Functions $T(n) = aT(n/b) + f(n)$
- Decreasing Functions $T(n) = aT(n-b) + f(n)$

Why master theorem works?
https://www.scaler.com/topics/data-structures/masters-theorem/

# Dividing Functions

- T(n) = aT(n/b) + f(n) where f(n) = Theta(n^k*log(^p)n)

Example:

T(n) = 2T(n/2) + n^2

T(n) = T(n/4) + nlogn

# Cases

- Case 1: if $\log_b a > k$, then:

T(n) = Theta(n^ $\log_b a$)

- Case 2: if $\log_b a = k$, then:

If p>-1, then T(n) = Theta(n^k $\log^{(p+1)} n$)

If p=-1, then T(n) = Theta(n^k loglogn)

- Case 3: if $\log_b a < k$, then:

If p>=0, then T(n) = Theta(n^k $\log^p n$)

If p<0, then T(n) = Theta(n^k)

# Decreasing Function

- T(n) = aT(n-b) + f(n) where f(n) = Theta(n^k)

Example

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

T(n) = 2*T(n-1) + n^2

# Cases

- If a<1, then T(n) = Theta(n^k)
- If a=1, then T(n) = Theta(n^(k+1))
- If a>1, then T(n) = Theta(n^(a/b) * f(n))

# Merge Sort

```
void merge(int*,int*,int,int,int);
void mergesort(int *a, int*b, int low, int high) {
  int pivot;
  if(low<high) {
     pivot=(low+high)/2;
     mergesort(a,b,low,pivot);
     mergesort(a,b,pivot+1,high);
     merge(a,b,low,pivot,high);
   }
}
  int main() {
  int a[] = {12,10,43,23,78,45,56,98,41,90,24};
  int num; num = sizeof(a)/sizeof(int);
  int b[num];
  mergesort(a,b,0,num-1);
  for(int i=0; i<num; i++) cout<<a[i]<<" "; cout<<endl; }
```

```
void merge(int *a, int *b, int low, int pivot,
  int high) {
  int h,i,j,k; h=low; i=low; j=pivot+1;
  while((h<=pivot)&&(j<=high)) {
    if(a[h]<=a[j]) { b[i]=a[h]; h++; }
    else { b[i]=a[j]; j++; }
    i++; }
    if(h>pivot) {
      for(k=j; k<=high; k++) {
        b[i]=a[k]; i++; } }
    else {
      for(k=h; k<=pivot; k++) {
        b[i]=a[k]; i++; } }
    for(k=low; k<=high; k++) a[k]=b[k]; }
```

# Complexity Analysis

- Worst Case Time Complexity: **O(n*log n)**
- Best Case Time Complexity: Omega**(n*log n)**
- Average Time Complexity: Theta**(n*log n)**
- Space Complexity: Depends on implementation. If array then O(N) else if linked list O(1).

# Till Now

| Algorithm | Worst Case | Average Case | Best Case | Space Complexity |
|---|---|---|---|---|
| Linear Search | **$O(N)$** | **$\Theta(N)$** | **$\Omega(1)$** | **$O(1)$** |
| Binary Search | **$O(\log N)$** | **$\Theta(\log N)$** | **$\Omega(1)$** | **$O(1)$** |
| Selection Sort | **$O(N^2)$** | **$\Theta(N^2)$** | **$\Omega(N^2)$** | **$O(1)$** |
| Bubble Sort | **$O(N^2)$** | **$\Theta(N^2)$** | **$\Omega(N)$** | **$O(1)$** |
| Insertion Sort | **$O(N^2)$** | **$\Theta(N^2)$** | **$\Omega(N)$** | **$O(1)$** |
| Merge Sort | **$O(N \log N)$** | **$\Theta(N \log N)$** | **$\Omega(N \log N)$** | **$O(N)/O(1)$** |

# Quick Sort

```
void quickSort(int a[], int first, int last);
int pivot(int a[], int first, int last);
void swap(int& a, int& b);
Void main()
{
  int test[] = { 7, -13, 1, 3, 10, 5, 2, 4 };
  int N = sizeof(test)/sizeof(int);
  quickSort(test, 0, N-1);
}
void swap(int& a, int& b) {
  int temp = a;
  a = b;
  b = temp;
}
```

```
void quickSort( int a[], int first, int last) {
  int pivotElement;
  if(first < last)  {
    pivotElement = pivot(a, first, last);
    quickSort(a, first, pivotElement-1);
    quickSort(a, pivotElement+1, last); }
}
int pivot(int a[], int first, int last){
  int p = first;
  int pivotElement = a[first];
  for(int i = first+1 ; i <= last ; i++){
    if(a[i] <= pivotElement){
      p++;
      swap(a[i], a[p]);
    }
  }
  swap(a[p], a[first]);
  return p;
}
```

# Complexity Analysis

- Worst Case Time Complexity: **$O(n^2)$**
- Best Case Time Complexity: Omega**(n*log n)**
- Average Time Complexity: Theta**(n*log n)**
- Space Complexity: O(N)

# Till Now

| Algorithm | Worst Case | Average Case | Best Case | Space Complexity |
|---|---|---|---|---|
| Linear Search | **$O(N)$** | **$\Theta(N)$** | **$\Omega(1)$** | **$O(1)$** |
| Binary Search | **$O(\log N)$** | **$\Theta(\log N)$** | **$\Omega(1)$** | **$O(1)$** |
| Selection Sort | **$O(N^2)$** | **$\Theta(N^2)$** | **$\Omega(N^2)$** | **$O(1)$** |
| Bubble Sort | **$O(N^2)$** | **$\Theta(N^2)$** | **$\Omega(N)$** | **$O(1)$** |
| Insertion Sort | **$O(N^2)$** | **$\Theta(N^2)$** | **$\Omega(N)$** | **$O(1)$** |
| Merge Sort | **$O(N \log N)$** | **$\Theta(N \log N)$** | **$\Omega(N \log N)$** | **$O(N)/O(1)$** |
| Quick Sort | **$O(N^2)$** | **$\Theta(N \log N)$** | **$\Omega(N \log N)$** | **$O(N)$** |

# Heap Sort

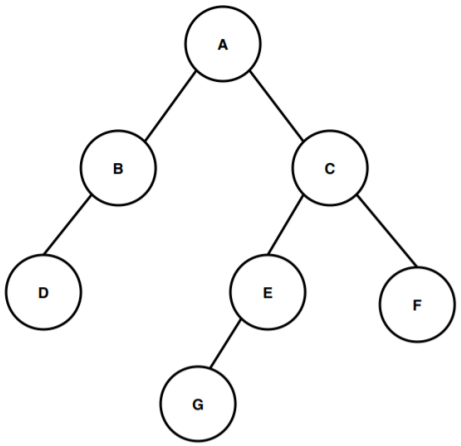
```
void heapify(int arr[], int n, int i) {
        int largest = i;
        int left = 2 * i + 1;
        int right = 2 * i + 2;
        if (left < n && arr[left] > arr[largest])
                largest = left;
        if (right < n && arr[right] > arr[largest])
                largest = right;
        if (largest != i) {
                swap(&arr[i], &arr[largest]);
                heapify(arr, n, largest);
        }}
```

# Contd.

```
void heapSort(int arr[], int n) {
        for (int i = n / 2 - 1; i >= 0; i--)
                heapify(arr, n, i);
        for (int i = n - 1; i >= 0; i--) {
                swap(&arr[0], &arr[i]);
                heapify(arr, i, 0);
        }
}
```

# Height of a Tree

- The height of a binary tree is the height of the root node in the whole binary tree.
- In other words, the height of a binary tree is equal to the largest number of edges from the root to the most distant leaf node.



# Time Complexity

The time complexity of heapsort consists of three parts:

1. Complexity of inserting a new node
2. Complexity of removing the max valued node from heap
3. Complexity of creating a heap

# New Node Insertion

- When we insert a new value in the heap when making the heap, the max number of steps we would need to take comes out to be O(log(n)).
- As we use binary trees, we know that the max height of such a structure is always O(log(n)).
- When we insert a new value in the heap, we will swap it with a value greater than it, to maintain the max-heap property. The number of such swaps would be O(log(n)).

# Removal of Max Node

- When we remove the max valued node from the heap, to add to the end of the list, the max number of steps required would also be O(log(n)).
- Since we swap the max valued node till it comes down to the bottom-most level, the max number of steps we'd need to take is the same as when inserting a new node, which is O(log(n)).

# Heap Creation

- When we create a heap, not all nodes will move down O(log(n)) times.
- The nodes at the bottom-most level (given by n/2) won't move down at all. The nodes at the second last level (n/4) would move down 1 time, as there is only one level below remaining to move down. The nodes at the third last level would move down 2 times, and so on.
- So if we multiply the number of moves we take for all nodes, mathematically, it would turn out like a geometric series, as explained below-
- (n/2 * 0) + (n/4 * 1) + (n/8 * 2) + (n/16 * 3) + ...h

Here h represents the height of the max-heap structure.

# Analysis

- Mathematically, we see that-
- The first remove of a node takes log(n) time
- The second remove takes log(n-1) time
- The third remove takes log(n-2) time
- and so on till the last node, which will take log(1) time
- So summing up all the terms, we get-

  log(n) + log(n-1) + log(n-2) + ....log(1)
  as log(x) + log(y) = log(x * y), we get
  $=\log(n*(n-1)*(n-2)*\ldots*2*1)$
  $=\log(n!)$
  Upon further simplification (using Stirling's approximation), log(n!) turns out to be
  $=n*\log(n)-n+O(\log(n))$

# Complexity Analysis

- Worst Case Time Complexity: **O((n*log n)**
- Best Case Time Complexity: Omega**(n)**
- Average Time Complexity: Theta**(n*log n)**
- Space Complexity: O(1)

# Till Now

| Algorithm | Worst Case | Average Case | Best Case | Space Complexity |
|---|---|---|---|---|
| Linear Search | **$O(N)$** | **$\Theta(N)$** | **$\Omega(1)$** | **$O(1)$** |
| Binary Search | **$O(\log N)$** | **$\Theta(\log N)$** | **$\Omega(1)$** | **$O(1)$** |
| Selection Sort | **$O(N^2)$** | **$\Theta(N^2)$** | **$\Omega(N^2)$** | **$O(1)$** |
| Bubble Sort | **$O(N^2)$** | **$\Theta(N^2)$** | **$\Omega(N)$** | **$O(1)$** |
| Insertion Sort | **$O(N^2)$** | **$\Theta(N^2)$** | **$\Omega(N)$** | **$O(1)$** |
| Merge Sort | **$O(N \log N)$** | **$\Theta(N \log N)$** | **$\Omega(N \log N)$** | **$O(N)/O(1)$** |
| Quick Sort | **$O(N^2)$** | **$\Theta(N \log N)$** | **$\Omega(N \log N)$** | **$O(N)$** |
| Heap Sort | **$O(N \log N)$** | **$\Theta(N \log N)$** | **$\Omega(N)$** | **$O(1)$** |

# Problem Solving Strategies

- Divide and conquer
- Backtracking
- Greedy
- Dynamic Programming
- Branch and Bound

Source:
https://www.enjoyalgorithms.com/blog/problem-solving-approaches-in-data-structures-and-algorithms

# Incremental Approach

- Our daily problem-solving activities: build partial step-by-step solution using loops.

1. Input-centric: process one input at each iteration.
2. Output-centric: add one output at each iteration.
3. Iterative-improvement: start with approximation solution and improve upon it.

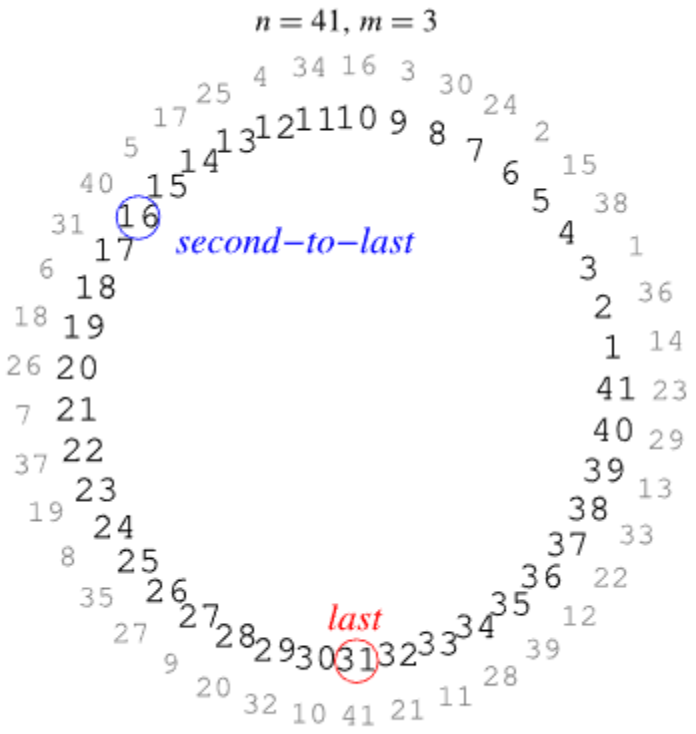
Example: Insertion Sort, Finding max and min in an array, Valid mountain array, Find equilibrium index of an array, Dutch national flag problem, Sort an array in a waveform.

# Decrease and Conquer Approach

- This is based on solution to a given problem via its one sub-problem solution.
- Such as approach leads naturally to a recursive algorithm, which reduces the problem to a sequence of smaller input sizes.
- This is continued till it reaches recursion's base case.

Example: Euclid algorithm of finding GCD, Binary Search, Josephus problem.

# Josephus Problem




Given a group of n men arranged in a circle under the edict that every mth man will be executed going around the circle until only one remains, find the position L(n,m) in which you should stand in order to be the last survivor (Ball and Coxeter 1987).

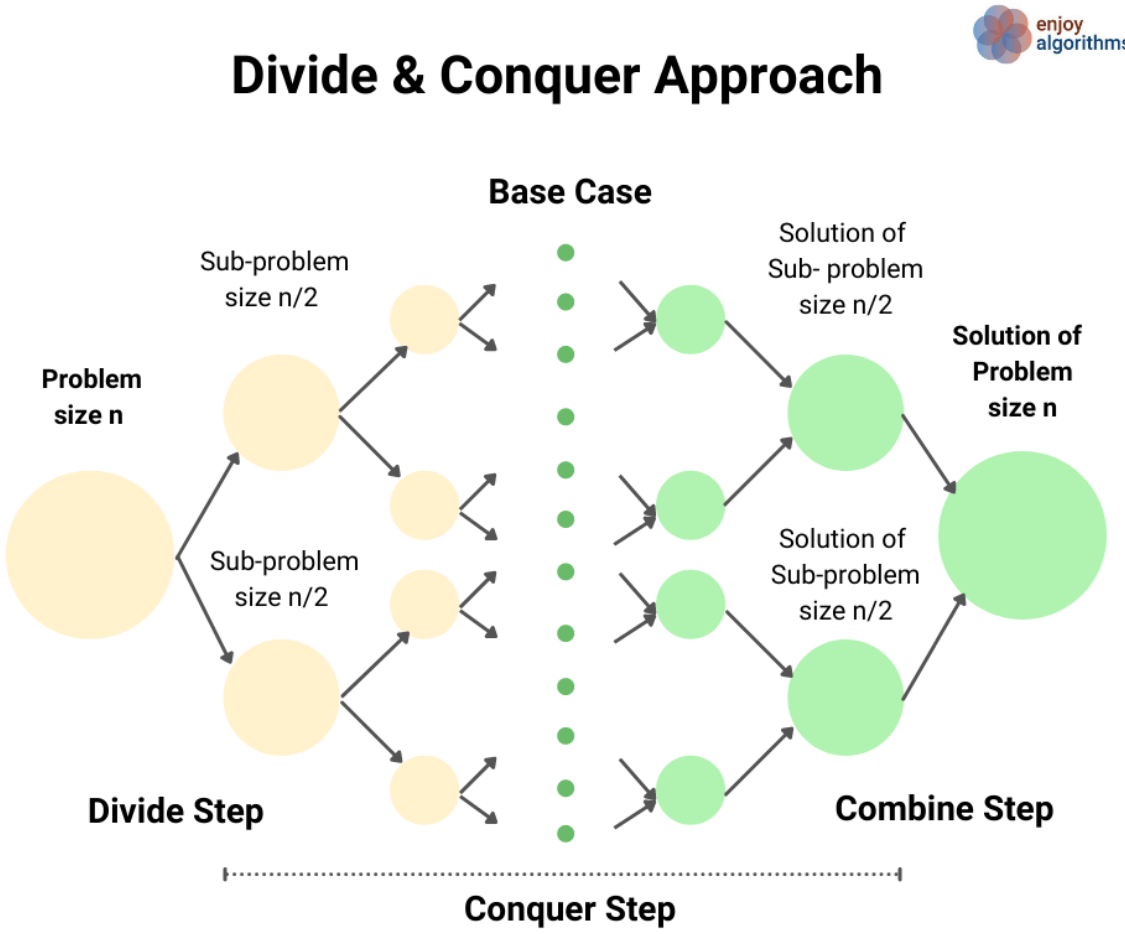
# Solution

```
#include <stdio.h>
int josephus(int n, int k)
{
   if (n == 1)
      return 1;
   else
      /* The position returned by josephus(n - 1, k) is
         adjusted because the recursive call josephus(n -
         1, k) considers the original position
         k%n + 1 as position 1 */
      return (josephus(n - 1, k) + k - 1) % n + 1;
}

int main()
{
   int n = 14;
   int k = 2;
   printf("The chosen place is %d", josephus(n, k));
   return 0;
}
```

# Divide and Conquer

- Merge Sort
- Quick Sort
- Median of Two Sorted Arrays



# Median of Sorted Arrays

- There are two sorted arrays A[] and B[] of size n each, write a program to find the median of the array obtained after merging both the arrays i.e. Merged array of size 2n.

Example

A[] = [1, 3, 6]  B[] = [2, 8, 12]

Output: 4.5

# Solution

- Here idea is to compare the medians of both sorted arrays and recursively reduce the search space by half.
- Let m1 = A[n/2] and m2 = B[n/2], then
- Case 1 if m1 == m2
- Case 2 if m1 < m2
- Case 3 if m1 > m2

# Case 1: m1 == m2

- ar1[] = {1, 12, 15, 26, 38}
- ar2[] = {2, 13, 15, 30, 45}
- combined list = {1, 2, 12, 13, 15, 15, 26, 30, 38, 45}

# Case 2: m1 < m2

- ar1[] = {1, 12, 15, 26, 38}
- ar2[] = {2, 13, 17, 30, 45}
- combined list = {1, 2, 12, 13, 15, 17, 26, 30, 38, 45}
- We need to consider {15, 26, 38}, values greater than 15 and {2, 13, 17}, values less then 17.

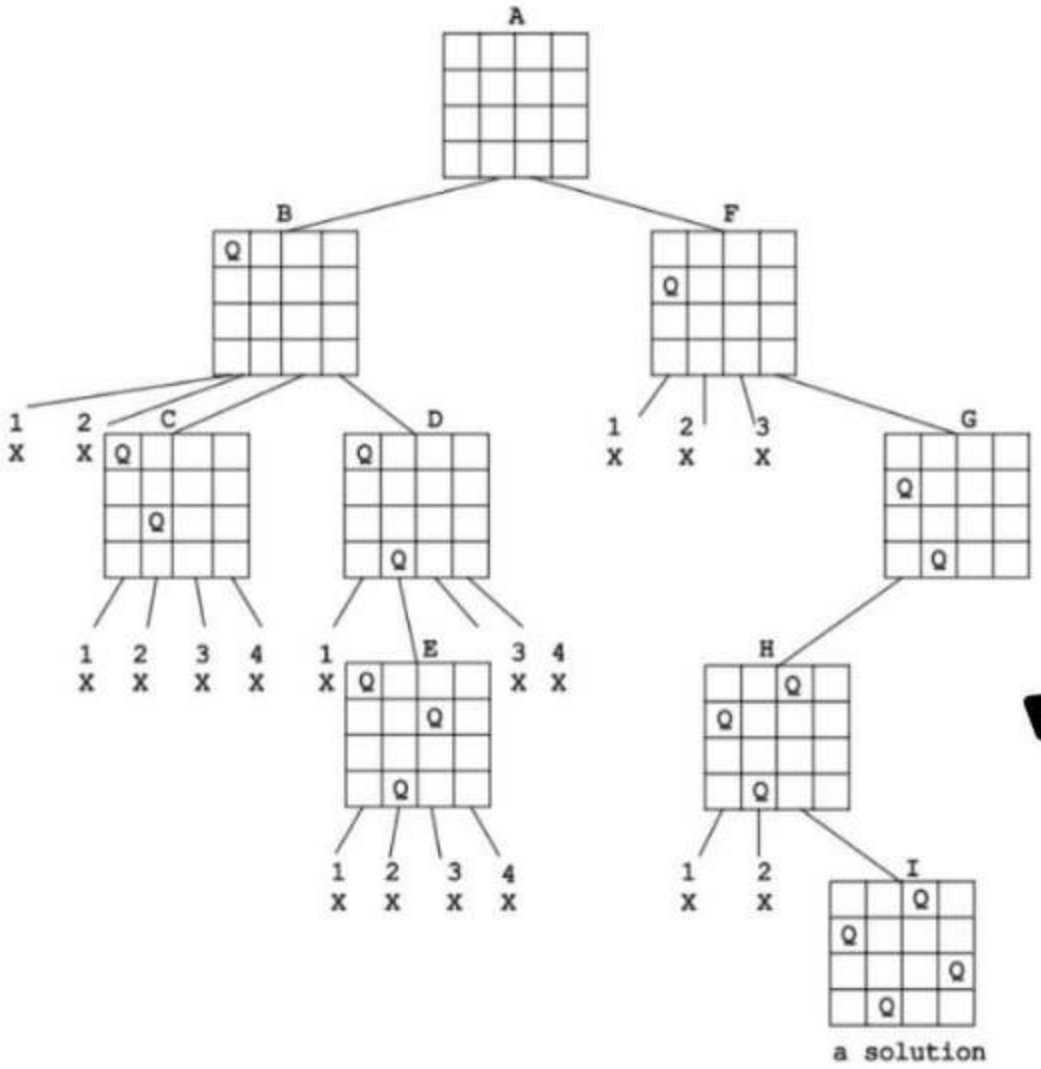
# Case 3: m1 > m2

- Similar to previous case.

# Backtracking

- Backtracking is an improvement over the approach of exhaustive search.
- It is a method for generating a solution by avoiding unnecessary possibilities of the solutions.

# N-Queen Problem




## n-Queens Problem

Place **n** queens on an **n × n** chessboard so that no two queens attack each other by being in the same column, row, or diagonal.

Backtracking solution of 4-queens problem

# Greedy

- This solves as optimization problem by expanding a partially constructed solution until a complete solution is reached.
- The greedy choice is the best alternative available at each step.

# 0-1 Knapsack Problem

- Given weights and values of n items, put these items in a knapsack of capacity W to get the maximum total value in the knapsack.

## 0-1 Knapsack Problem

value[] = {60, 100, 120};
weight[] = {10, 20, 30};
W = 50;

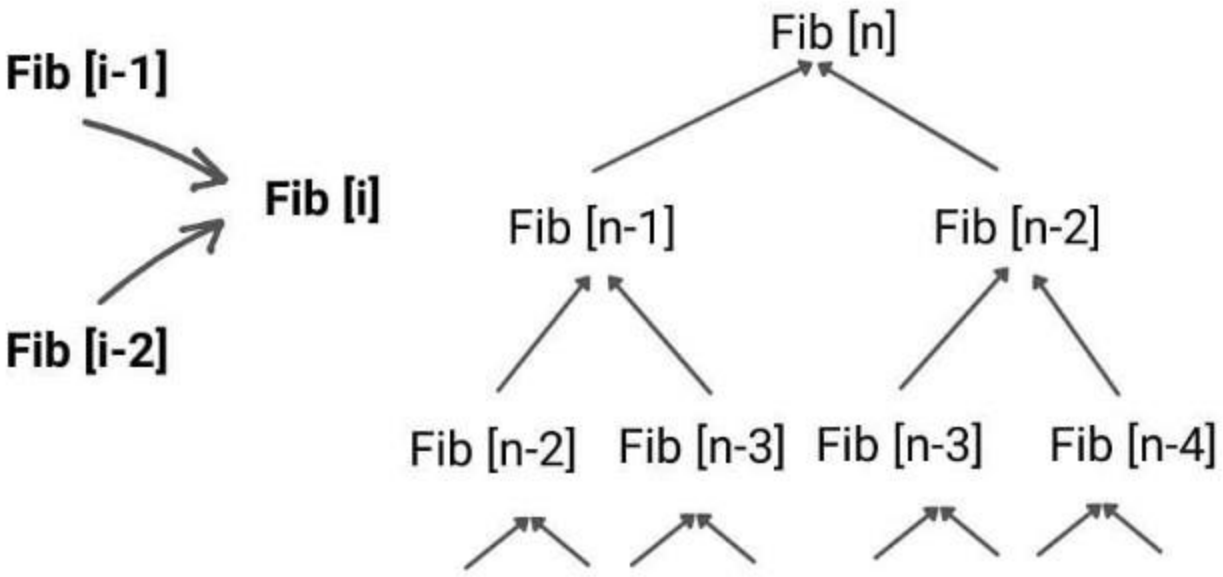
Solution: 220

Weight = 10; Value = 60;
Weight = 20; Value = 100;
Weight = 30; Value = 120;
Weight = (20+10); Value = (100+60);
Weight = (30+10); Value = (120+60);
Weight = (30+20); Value = (120+100);
Weight = (30+20+10) > 50

# Dynamic Programming

- For solving problems with overlapping or repeated subproblems.
- Here rather than solving overlapping subproblems repeatedly, we solve each smaller subproblems only once and store the result in memory.

# Fibonacci Sequence




A DP solution avoids recomputation, solves each sub-subproblem just once, and saves its answer in a table.

Dynamic Programming applies when the subproblems are dependent and repeated during the recursive calls.