CSE 125
Discrete Mathematics

Nazia Sultana Chowdhury
nazia.nishat1971@gmail.com

Binary Search

- Pre-condition: Sorted Array.

Binary Search

3	5	6	10	13	17	19

n = 7, number = 5

Case 1 :- if the target is equal to array [mid]

Case 2 :- if the target is les than array [mid]

Case 3 :- if the target is greater than array [mid]

→ If doen't ~~fullfill~~ fulfill, "not found".

```
binarySearch(int arr[],int l,int r,int num)
{

    if(r>=l)
    {
        int mid = l+(r-l)/2;

        if(num == arr[mid])
        {
            return mid;
        }
        else if(num<arr[mid])
        {
            return binarySearch(arr,l,mid-1,num);
        }
        else
        {
            return binarySearch(arr,mid+1,r,num);
        }
    }

    return -1;
}
```

Divide and Conquer Algorithms

- A divide-and-conquer algorithm recursively breaks down a problem into two or more sub-problems of the same or related type, until these become simple enough to be solved directly.
- The solutions to the sub-problems are then combined to give a solution to the original problem.

Quick Sort Algorithm

- Quicksort is a sorting algorithm based on the divide and conquer approach.
- Sort function by most of the language libraries are implementations of Quick Sort only.

```
quickSort(int arr[],int low,int high)
{
    if(low<high)
    {
        int pi = partition_arr(arr,low,high);
        quickSort(arr,low,pi-1);
        quickSort(arr,pi+1,high);
    }
}
```

```
int partition_arr(int arr[],int low,int high)
{
    int pivot = arr[high];

    int i=low-1;

    for(int j=low; j<high; j++)
    {
        if(arr[j]<= pivot)
        {
            i++;
            swap(arr[j],arr[i]);
        }
    }

    swap(arr[i+1],arr[high]);
    return (i+1);
}
```

Merge Sort Algorithm

- Merge Sort is a Divide and Conquer algorithm.
- It divides the input array into two halves, calls itself for the two halves, and then merges the two sorted halves.

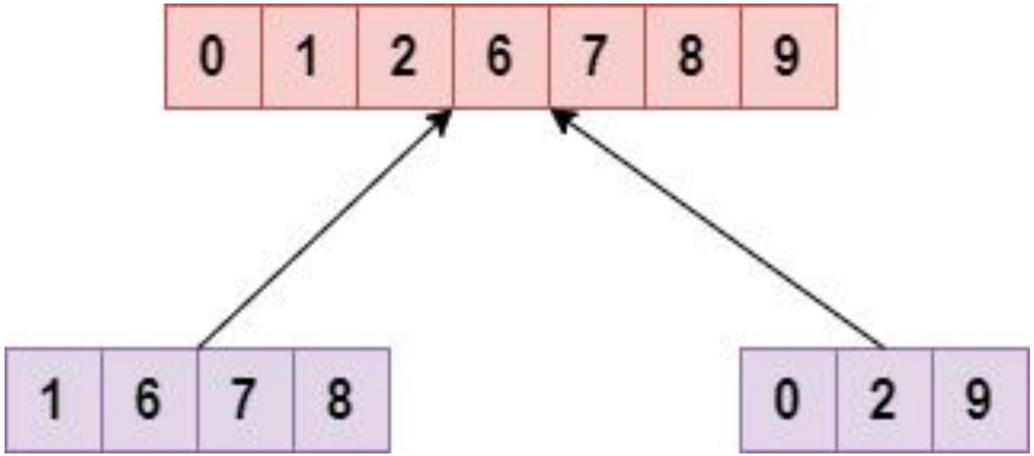


Fig: Merging Sorted Array

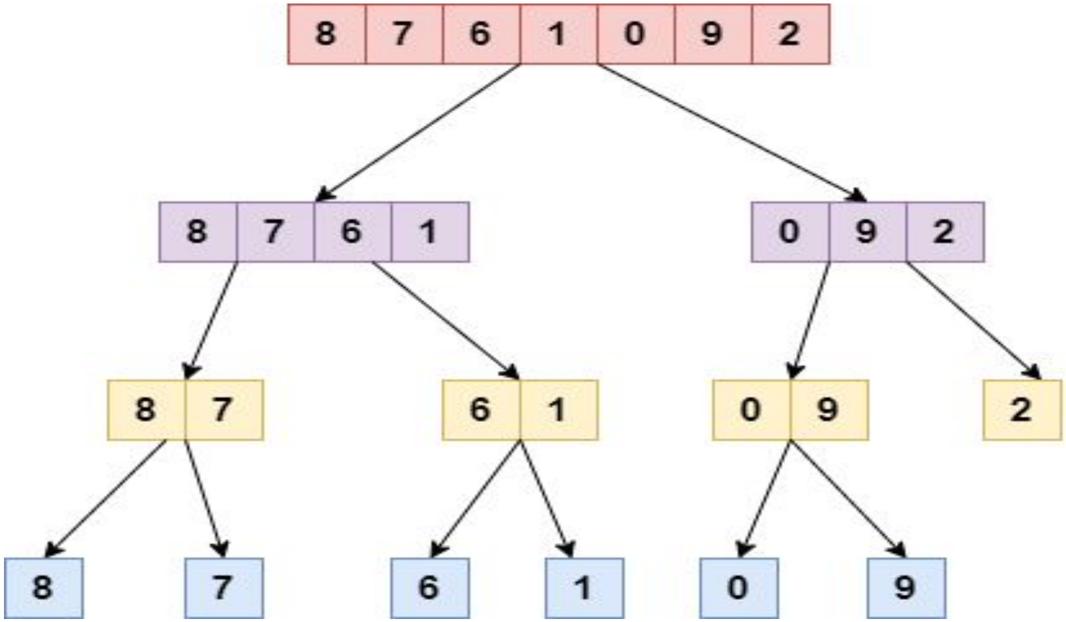


Fig: Merge Sort Process

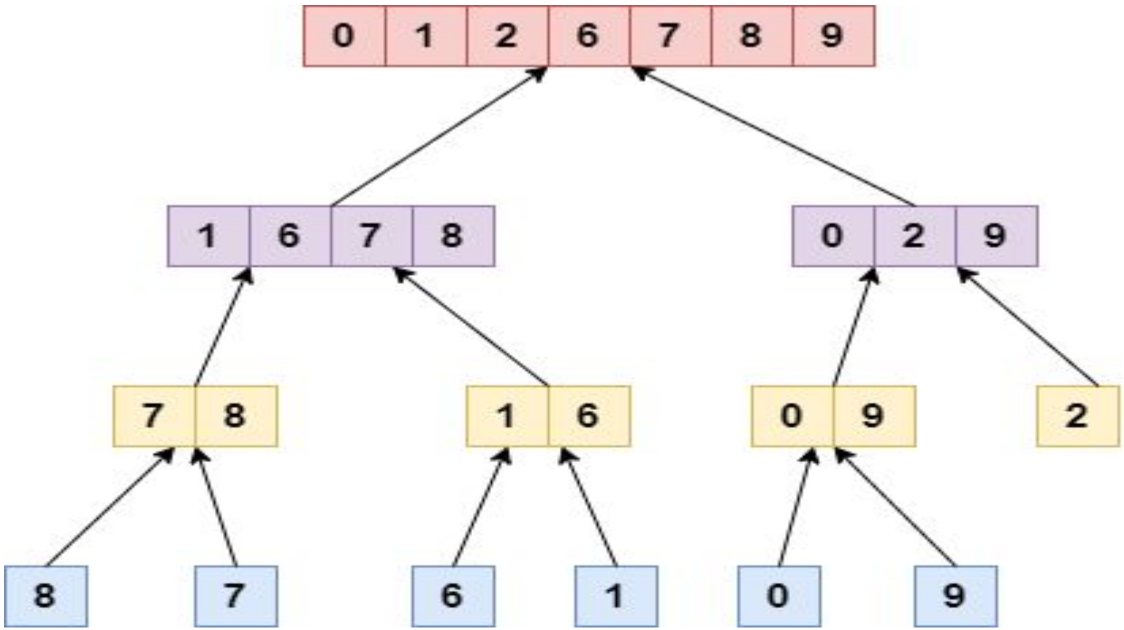


Fig: Merge Sort Process

```
mergeSort(int a[],int l,int r)
{
    if(l<r)
    {
        int m=(l+r)/2;
        mergeSort(a,l,m);
        mergeSort(a,m+1,r);
        _merge(a,l,m,r);
    }
}
```

```
_merge(int arr[],int l,int m,int r)
{
    int leftLen = m-l+1;
    int rightLen = r-m;

    int left[leftLen],right[rightLen];
    for(int i=0; i<leftLen; i++)
    {
        left[i] = arr[i+l];
    }
    for(int i=0; i<rightLen; i++)
    {
        right[i] = arr[i+m+1];
    }
    int i=0,j=0,k=l;
```

```
while(i<leftLen && j<rightLen)
{
    if(left[i] < right[j])
    {
        arr[k] = left[i];
        i++;
    }
    else
    {
        arr[k] = right[j];
        j++;
    }
    k++;
}
```

```
    while(i<leftLen)
    {
        arr[k++] = left[i];
        i++;
    }
    while(j<rightLen)
    {
        arr[k++] = right[j];
        j++;
    }
}
```

The Growth of Functions

- Determining how fast an algorithm can solve a problem as the size of the input grows.
- Comparing the efficiency of two different algorithms for solving the same problem.

Describing Growth of Functions

- Big-O Notation
- Big-Omega
- Big-Theta Notation

Big-O Notation

Let f and g be functions from the set of integers or the set of real numbers to the set of real numbers.

- $f(x)$ is $O(g(x))$ if there are constants C and k such that

  $|f(x)| \leq C|g(x)|$ whenever $x > k$.

Big-O Notation

- Describes the long-term growth rate of functions.
- Doesn't care about constants.
- Gives an upper bound.

$f(n)$, in terms of $O(g(n))$?

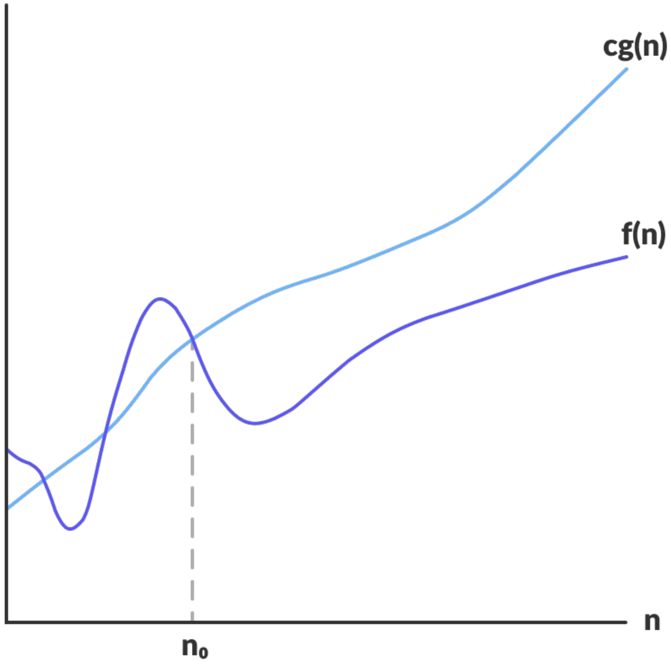
Here, $f(n) = n^2 + 2n$

$$f(n) \leq C\, g(n)$$

$\Rightarrow n^2 + 2n \leq$ ______

$\Rightarrow n^2 + 2n \leq C \cdot n^2$

$\Rightarrow n^2 + 2n \leq 3n^2$, for $C = 3$ and $n \geq 1$



f(n) = O(g(n))

Big-Omega

- Let f and g be functions from the set of integers or the set of real numbers to the set of real numbers.
- $f(x)$ is $\Omega(g(x))$ if there are positive constants C and k such that $|f(x)| \geq C|g(x)|$ whenever $x > k$.
- Lower Bound.

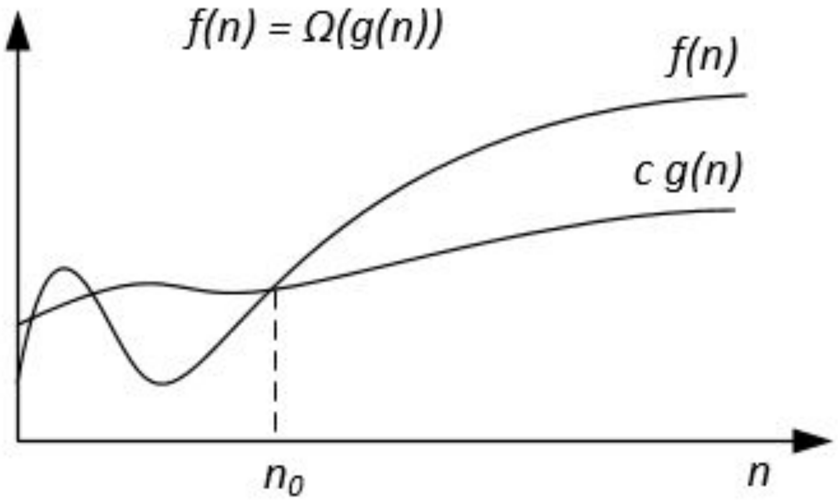
$f(n)$ in terms of $\Omega(g(n))$?

$$f(n) \geqslant c\, g(n)$$

$$\Rightarrow n^2 + 2n \geqslant \_\_\_\_\_\_$$

$$\Rightarrow n^2 + 2n \geqslant c\, n^2$$

$$\Rightarrow n^2 + 2n \geqslant n^2, \text{ for } c = 1 \text{ and } n \geqslant 1$$


f(n) = Ω(g(n))
f(n)
c g(n)
n0
n

Big-Theta Notation

Let f and g be functions from the set of integers or the set of real numbers to the set of real numbers.

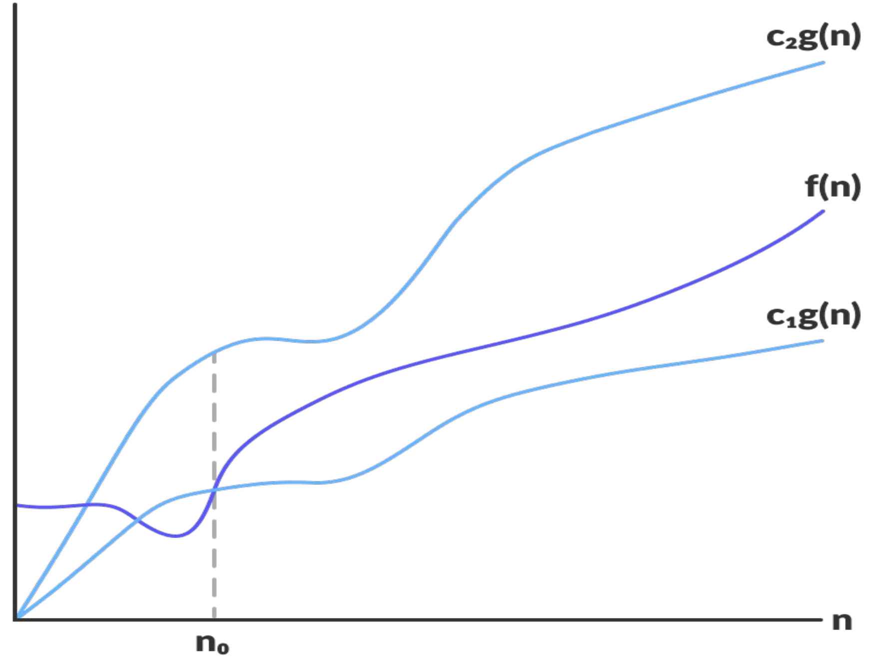
$f(x)$ is $\Theta(g(x))$ if

- $f(x)$ is $O(g(x))$ and
- $f(x)$ is $\Omega(g(x))$.

Big-Theta

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

$$\Rightarrow \quad n^2 - \cancel{2} \le n^2 + 2n \le 3n^2$$



$f(n) = \Theta(g(n))$

Time Complexity

- Estimates how much time the algorithm will use for some input.
- The idea is to represent the efficiency as a function whose parameter is the size of the input.
- By calculating the time complexity, we can find out whether the algorithm is fast enough without implementing it.

```
for (int i = 1; i <= n; i++) {
    // code
}
```

Time Complexity: **O(n)**

```
for (int i = 1; i <= n; i++) {
    for (int j = 1; j <= n; j++) {
        // code
    }
}
```

Time Complexity: $\mathbf{O(n^2)}$

```
for (int i = 1; i <= 3*n; i++) {
    // code
}
```

```
for (int i = 1; i <= n+5; i++) {
    // code
}
```

```
for (int i = 1; i <= n; i += 2) {
    // code
}
```

Time Complexity: **O(n)**

```
for (int i = 1; i <= n; i++) {
    for (int j = i+1; j <= n; j++) {
        // code
    }
}
```

Time Complexity: $\mathbf{O(n^2)}$

```
for (int i = 1; i <= n; i++) {
    // code
}
for (int i = 1; i <= n; i++) {
    for (int j = 1; j <= n; j++) {
        // code
    }
}
for (int i = 1; i <= n; i++) {
    // code
}
```

Time Complexity: $\mathbf{O(n^2)}$

```
for (int i = 1; i <= n; i++) {
    for (int j = 1; j <= m; j++) {
        // code
    }
}
```

Time Complexity: **O(nm)**

input size	required time complexity
$n \leq 10$	$O(n!)$
$n \leq 20$	$O(2^n)$
$n \leq 500$	$O(n^3)$
$n \leq 5000$	$O(n^2)$
$n \leq 10^6$	$O(n \log n)$ or $O(n)$
n is large	$O(1)$ or $O(\log n)$

Some useful estimates assuming a time limit of one second.