ITT304 Algorithm Analysis and Design

Module 1 : Introduction to Algorithms

Anoop S. K. M.
(skmanoop@gecidukki.ac.in)
Department of Information Technology
Govt. Engg. College, Idukki

Acknowledgements

- All the pictures are taken from the Internet using Google search.
- Wikipedia also referred.

Lecture 01

- What are algorithms?
- What is the role of algorithms relative to other technologies used in computers?

What are Algorithms ?

Algorithm

Any well-defined computational procedure that takes some value, or set of values, as input and produces some value, or set of values, as output. An algorithm is thus a sequence of computational steps that transform the input into the output.

Features of a Good Algorithm

According to Scheider and Gersting (1995), features of a good algorithm includes

- **Unambiguous Operations:** Must have specific, outlined steps.
- **Well-Ordered:** The exact order of operations performed should be concretely defined.
- **Feasibility:** All steps of an algorithm should be possible (also known as effectively computable).
- **Input:** Be able to accept a well-defined set of inputs.
- **Output:** Should produce some result as an output, so that its correctness can be reasoned about.
- **Finiteness:** Should terminate after a finite number of instructions
- **Time & Space:**A good algorithm is one that is taking less time and less space

Role of Algorithms

Any branch of Computer Science we take, Algorithms play a key role.

- Computer Networks - Shortest Path Algorithms
- Cryptography - Number Theoretic Algorithms
- Computer Graphics - Geometric Algorithms
- Database Design - Search Algorithms
- Artificial Intelligence - Classification, Regression, Clustering Algorithms
- Search in Web. How ? Google! Page Rank Algorithm!

Lecture # 02

Recap & Goals

Recap...

- Module 1 : Introduction to Algorithms
    - Algorithm : Definition, Features and Role of Algorithms

Today's Goal...

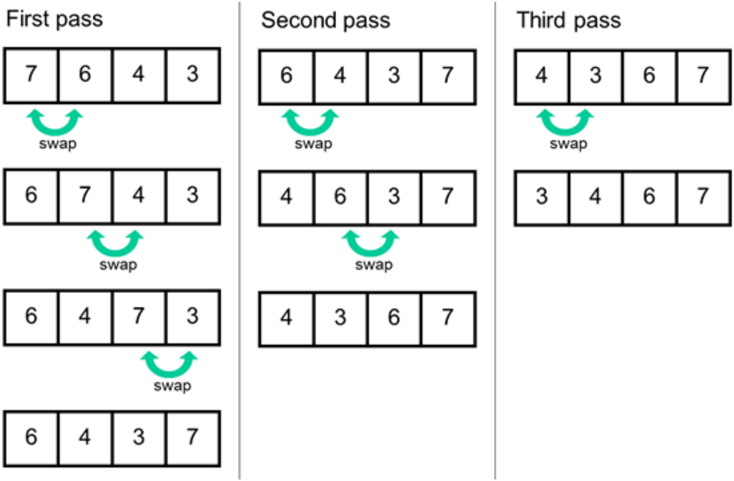
- Algorithm to Pseudocode and to Program

Algorithm, Pseudocode, Program

- Algorithms are generally written in plain English
- Pseudo-codeis written in a format that is similar to the structure of a high-level programming language
- Program write a code in a particular programming language.

Let us see an example : Bubble Sort

Bubble Sort-All passes Example



Bubble Sort : Algorithm & Pseudo-code

Algorithm:

- Step 1 : Accept the contents of the array of size n
- Step 2: Compare the first two elements
  - if first element > second element swap them
- Step 3: continue the Step 2 comparison process, with the second and third element, then third and forth element and so on till the last two elements
- Step 4: Restart from Step 2 the whole process starting with the first two elements until the array is sorted

Pseudo code:

```
for i=0 to n-1
 accept A[i]
for i=1 to n-1
 for j=0 to n-2
  if A[j] > A[j+1] then
   Swap( A[j] and A[j+1])
```

Pseudo code & Program

Pseudo code:

```
for i=0 to n-1
  accept A[i]
for i=1 to n-1
 for j=0 to n-2
   if A[j] > A[j+1] then
     Swap( A[j] and A[j+1])
```

```
int main()
{
 int n,temp;
 scanf("%d",&n);
 int A[n];
 for(int i=0; i<n; i++)
   scanf("%d",&A[i]);
 for(int i=1;i<n;i++){
   for(int j=0;j<n-1;j++){
     if(A[j] > A[j+1]){
       temp = A[j];
       A[j] = A[j+1];
       A[j+1] = temp;
     }
   }
 }
}
```

To Think

Are there any problems for which there is no algorithm ?

Oh Ya! Halting Problem!

Let's discuss one example : Treasure Hunt in Number Line

Lecture # 03

Recap & Goals

Recap...

- Module 1 : Introduction to Algorithms
  - Algorithm : Definition, Features and Role of Algorithms
  - Algorithm to Pseudocode and to Program

Today's Goal...

- Big O

Growth of Functions

Asymptotic Efficiency of Algorithm

We are concerned with how the running time of an algorithm increases with the size of the input, as the size of the input increases without bound.

- Usually, an algorithm that is asymptotically more efficient will be the best choice for all but very small inputs.
- Bubble Sort : $O(n^2)$, Merge Sort : $O(n \log n)$.
- Merge Sort is asymptotically more efficient that Bubble Sort
- Bubble Sort, Insertion Sort, Selection Sort are all asymptotically same!

Big O(O)

If the running time of an algorithm for an input n is given as

$$T(n) = c_0 + c_1 n + c_2 n^2 + \ldots + c_i n^i$$

then we say the running time of the algorithm is $O(n^i)$

Running Time Complexity : Example 1

```
void printFirstElementOfArray(int arr[n])
{
     printf("First element of array = %d",arr[0]);
}
```

Time Complexity is $O(n)$? No!
This is a constant time algorithm, So Time Complexity is $O(1)$

Running Time Complexity : Example 2

```
void printAllElementOfArray(int arr[], int n)
{
     int size=n;
     for (int i = 0; i < size; i++)
     {
         printf("%d\n", arr[i]);
     }
}
```

Time Complexity is $O(n)$, as the number of times printf executes is dependent on the size which is equal to n

Running Time Complexity : Example 3

```
void printAllPossibleOrderedPairs(int arr[], int n)
{
    int size = n;
    for (int i = 0; i < size; i++)
    {
        for (int j = 0; j < size; j++)
        {
            printf("%d = %d\n", arr[i], arr[j]);
        }
    }
}
```

Time Complexity is ? $O(n^2)$, as the number of times printf executes is dependent on the size $\times$size which is equal to n^2

Running Time Complexity : Example 4

```
void printAllItemsTwice(int arr[], int n)
{
    for (int i = 0; i < n; i++)
    {
        printf("%d\n", arr[i]);
    }

    for (int i = 0; i < n; i++)
    {
        printf("%d\n", arr[i]);
    }
}
```

Time Complexity is ? $O(2n)$? It is OK, But we generally avoid the constant term associated and will write it as $O(n)$

Running Time Complexity : Example 5

```
void print1stItemAndFirstHalfThenHi100Times(int arr[], int n)
{
int size = n;
    printf("First element of array = %d\n",arr[0]);
for (int i = 0; i < size/2; i++)
    {
        printf("%d\n", arr[i]);
    }
    for (int i = 0; i < 100; i++)
    {
        printf("Hi\n");
    }
}
```

Time Complexity is ? $O(1 + \frac{n}{2} + 100)$? Like the previous example will write it as $O(n)$, by avoiding the constant terms.

Running Time Complexity : Example 5

```
void print1stItemAndFirstHalfThenHi100Times(int arr[], int n)
{
    int size = n;
    printf("First element of array = %d\n",arr[0]);
    for (int i = 0; i < size/2; i++)
    {
        printf("%d\n", arr[i]);
    }
    for (int i = 0; i < 100; i++)
    {
        printf("Hi\n");
    }
}
```

Time Complexity is ? $O(1 + \frac{n}{2} + 100)$? Like the previous example will write it as $O(n)$, by avoiding the constant terms.

Running Time Complexity : Example 6

$$O(n^3 + 50n^2 + 10000) is \ O(n3)$$
$$O((n + 30) * (n + 5)) is \ O(n^2)$$

Running Time Complexity : Example 7

```
bool arrayContainsElement(int arr[], int size, int element)
{
    for (int i = 0; i < size; i++)
    {
        if (arr[i] == element) return true;
    }
    return false;
}  %\pause
```

- Running Time dependent upon the element that is searched
- Best Case, Worst Case and Average Case
- Best Case $O(1)$
- Average and Worst Case are both ? $O(n)$ Why?

Lecture # 04

Recap & Goals

Recap...

- Module 1 : Introduction to Algorithms
  - Algorithm : Definition, Features and Role of Algorithms
  - Algorithm to Pseudocode and to Program
  - Introduction to Big O

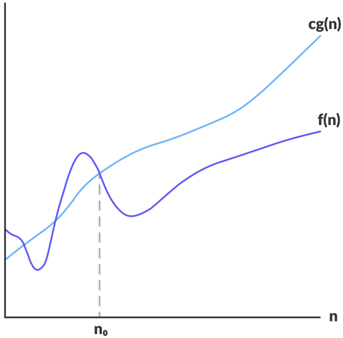
Today's Goal...

- Formal definition : O, Ω, Θ

Big O : Formal Definition

$O(g(n)) =$

$\{f(n) \mid$ there exists positive constants c and n_0 such that
$0 \leq f(n) \leq c * g(n)$ for all $n \geq n_0\}$



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

Big O Example

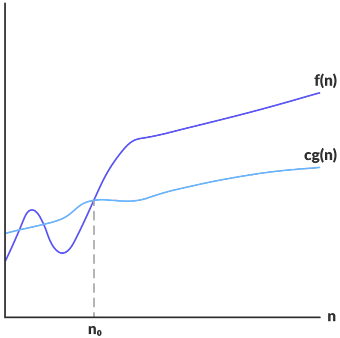
$O(g(n)) =$
$\{f(n) \mid$ there exists positive constants c and n_0 such that
$0 \leq f(n) \leq c * g(n)$ for all $n \geq n_0\}$

Let $f(n) = n^3 + 50n^2 + 100, g(n) = n^3$. Prove that $f(n) = O(n^3)$
Find c, n_0 such that $f(n) \leq c * g(n)$ for all $n \geq n_0$
What about $c = 151, n_0 = 1$
$n^3 + 50n^2 + 100 \leq 151n^3$, Hence $O(n^3 + 50n^2 + 100) is$ $O(n^3)$

Big Omega(Ω): Formal Definition

$\Omega(g(n)) =$

$\{f(n) \mid$ there exists positive constants c and n_0 such that
$0 \leq c * g(n) \leq f(n)$ for all $n \geq n_0\}$



f(n) = Ω(g(n))

Big Omega(Ω) Example

$\Omega(g(n)) =$
$\{f(n) \mid$ there exists positive constants c and n_0 such that
$0 \leq c * g(n) \leq f(n)$ for all $n \geq n_0\}$

If $f(n) = (n^3 + 50n^2 + 100), g(n) = n^3 is$ Prove that $f(n)\Omega(g(n))$
Find c, n_0 such that $f(n) \leq c * g(n)$ for all $n \geq n_0$
What about $c = 1, n_0 = 1$?
$1 * n^3 \leq n^3 + 50n^2 + 100$, for all $n \geq n_0 = 1$,
Hence $(n^3 + 50n^2 + 100) is$ $\Omega(n^3)$

Lecture # 05

Recap & Goals

Recap...

- Module 1 : Introduction to Algorithms
  - Algorithm : Definition, Features and Role of Algorithms
  - Algorithm to Pseudocode and to Program
  - Formal definition O, Ω

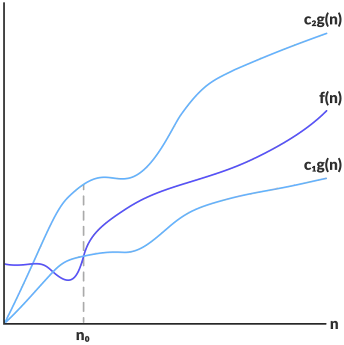
Today's Goal...

- Comparing O, Ω and Θ

Theta(Θ): Formal Definition

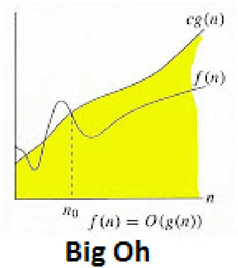
$\Theta(g(n)) =$

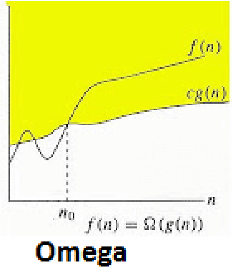
$\{f(n) \mid$ there exists positive constants c_1, c_2 and n_0 such that $0 \leq c_1 * g(n) \leq f(n \leq c_2 * g(n))$ for all $n \geq n_0\}$

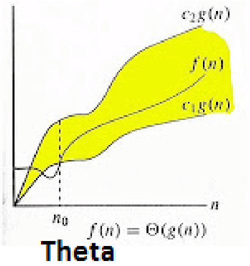


f(n) = Θ(g(n))

O, Ω and Θ







O, Ω and Θ Problem 1

Let $f(n) = \frac{1}{2}n^2 - 3n, g(n) = n^2$

O, Ω and Θ Problem 1

Let $f(n) = \frac{1}{2}n^2 - 3n, g(n) = n^2$
To show that $f(n) = \Theta(g(n))$, we show $\frac{1}{2}n^2 - 3n = \Theta(n^2)$

O, Ω and Θ Problem 1

Let $f(n) = \frac{1}{2}n^2 - 3n, g(n) = n^2$
To show that $f(n) = \Theta(g(n))$, we show $\frac{1}{2}n^2 - 3n = \Theta(n^2)$
To do so, we must determine positive constants c_1, c_2, and n_0 such that,

O, Ω and Θ Problem 1

Let $f(n) = \frac{1}{2}n^2 - 3n, g(n) = n^2$
To show that $f(n) = \Theta(g(n))$, we show $\frac{1}{2}n^2 - 3n = \Theta(n^2)$
To do so, we must determine positive constants c_1, c_2, and n_0 such that,

$$c_1 n^2 \leq \frac{1}{2}n^2 - 3n \leq c_2 n^2$$

O, Ω and Θ Problem 1

Let $f(n) = \frac{1}{2}n^2 - 3n, g(n) = n^2$
To show that $f(n) = \Theta(g(n))$, we show $\frac{1}{2}n^2 - 3n = \Theta(n^2)$
To do so, we must determine positive constants c_1, c_2, and n_0 such that,

$$c_1 n^2 \leq \frac{1}{2}n^2 - 3n \leq c_2 n^2$$

Dividing by n^2 we get

O, Ω and Θ Problem 1

Let $f(n) = \frac{1}{2}n^2 - 3n, g(n) = n^2$
To show that $f(n) = \Theta(g(n))$, we show $\frac{1}{2}n^2 - 3n = \Theta(n^2)$
To do so, we must determine positive constants c_1, c_2, and n_0 such that,

$$c_1 n^2 \leq \frac{1}{2}n^2 - 3n \leq c_2 n^2$$

Dividing by n^2 we get

$$c_1 \leq \frac{1}{2} - \frac{3}{n} \leq c_2$$

O, Ω and Θ Problem 1

Let $f(n) = \frac{1}{2}n^2 - 3n, g(n) = n^2$
To show that $f(n) = \Theta(g(n))$, we show $\frac{1}{2}n^2 - 3n = \Theta(n^2)$
To do so, we must determine positive constants c_1, c_2, and n_0 such that,

$$c_1 n^2 \leq \frac{1}{2}n^2 - 3n \leq c_2 n^2$$

Dividing by n^2 we get

$$c_1 \leq \frac{1}{2} - \frac{3}{n} \leq c_2$$

We can choose c_2 as $\frac{1}{2}$, what about c_1?

O, Ω and Θ Problem 1

Let $f(n) = \frac{1}{2}n^2 - 3n, g(n) = n^2$
To show that $f(n) = \Theta(g(n))$, we show $\frac{1}{2}n^2 - 3n = \Theta(n^2)$
To do so, we must determine positive constants c_1, c_2, and n_0 such that,

$$c_1 n^2 \leq \frac{1}{2}n^2 - 3n \leq c_2 n^2$$

Dividing by n^2 we get

$$c_1 \leq \frac{1}{2} - \frac{3}{n} \leq c_2$$

We can choose c_2 as $\frac{1}{2}$, what about c_1?
We see that for all $n \geq n_0 = 7$, we may choose $c_1 = \frac{1}{14}$.

O, Ω and Θ Problem 1

Let $f(n) = \frac{1}{2}n^2 - 3n, g(n) = n^2$
To show that $f(n) = \Theta(g(n))$, we show $\frac{1}{2}n^2 - 3n = \Theta(n^2)$
To do so, we must determine positive constants c_1, c_2, and n_0 such that,

$$c_1 n^2 \leq \frac{1}{2}n^2 - 3n \leq c_2 n^2$$

Dividing by n^2 we get

$$c_1 \leq \frac{1}{2} - \frac{3}{n} \leq c_2$$

We can choose c_2 as $\frac{1}{2}$, what about c_1?
We see that for all $n \geq n_0 = 7$, we may choose $c_1 = \frac{1}{14}$.
$\forall n \geq 7, \frac{1}{14}n^2 \leq \frac{1}{2}n^2 - 3n \leq \frac{1}{2}n^2$

O, Ω and Θ Problem 2

$$\text{Check if } n^3 = \Theta(n^2)?$$

We need to prove two things

1. $n^3 = \Omega(n^2)$
2. $n^3 = O(n^2)$

To prove $n^3 = \Omega(n^2)$,

- Does $\exists c$(positive constant) such that, $n^3 \geq c * n^2, \forall n \geq n_0$?

O, Ω and Θ Problem 2

$$\text{Check if } n^3 = \Theta(n^2)?$$

We need to prove two things

1. $n^3 = \Omega(n^2)$
2. $n^3 = O(n^2)$

To prove $n^3 = \Omega(n^2)$,

- Does $\exists c$(positive constant) such that, $n^3 \geq c * n^2, \forall n \geq n_0$? Yes! So $n^3 = \Omega(n^2)$

To prove $n^3 = O(n^2)$,

- Does $\exists c$(positive constant) such that $n^3 \leq c * n^2, \forall n \geq n_0$? Dividing by n^2 we get

$$n \leq c$$

O, Ω and Θ Problem 2

$$\text{Check if } n^3 = \Theta(n^2)?$$

We need to prove two things

1. $n^3 = \Omega(n^2)$
2. $n^3 = O(n^2)$

To prove $n^3 = \Omega(n^2)$,

- Does $\exists c$(positive constant) such that, $n^3 \geq c * n^2, \forall n \geq n_0$? Yes! So $n^3 = \Omega(n^2)$

To prove $n^3 = O(n^2)$,

- Does $\exists c$(positive constant) such that $n^3 \leq c * n^2, \forall n \geq n_0$? Dividing by n^2 we get

$$n \leq c$$

We can't find such a constant! So $n^3 \neq O(n^2)$

O, Ω and Θ Tutorial 1

Check if $n^2 = \Theta(n^3)$?

Lecture # 06

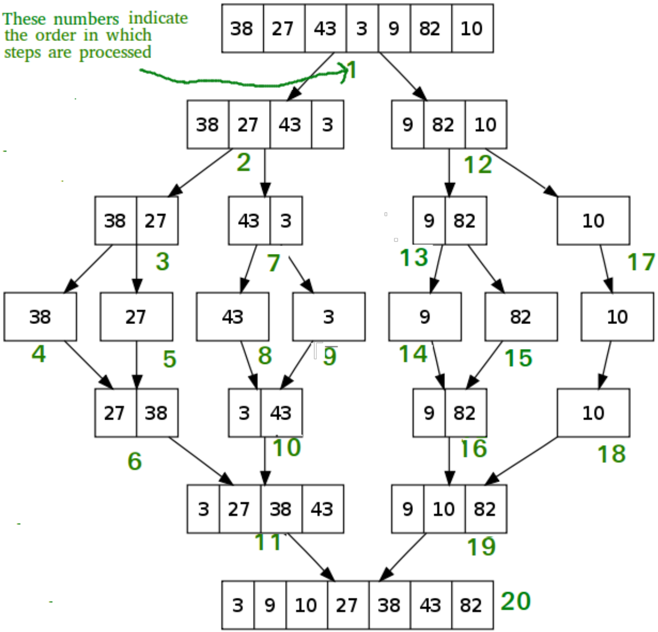
Recap & Goals

Recap...

- Module 1 : Introduction to Algorithms
  - Algorithm : Definition, Features and Role of Algorithms
  - Algorithm to Pseudocode and to Program
  - O, Ω and Θ

Today's Goal...

- Recursive Algorithms : Merge Sort, Quick Sort

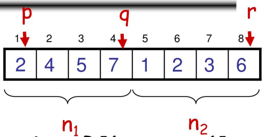

These numbers indicate the order in which steps are processed
38 27 43 3 9 82 10
1
38 27 43 3
9 82 10
2
12
38 27
43 3
9 82
10
3
7
13
17
38
27
43
3
9
82
10
4
5
8
9
14
15
27 38
3 43
9 82
10
6
10
16
18
3 27 38 43
9 10 82
11
19
3 9 10 27 38 43 82
20

Merge Sort

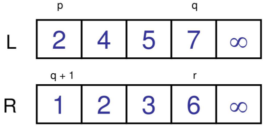
- it is a Divide and Conquer technique!
- divides the input array into two halves
- calls itself for the two halves
- then merges the two sorted halves.
- merge(arr, p, q, r) is a key process that assumes that arr[p..q] and arr[q+1..r] are sorted and merges the two sorted sub-arrays into one

Merge - Pseudocode

Alg.: MERGE(A, p, q, r)




```
Compute n1 and n2
Copy the first n1 elements into
   L[1 . . n1 + 1] and  the next n2 elements into R[1 . . n2 + 1]
L[n1 + 1] ← ∞;     R[n2 + 1] ← ∞
 i ← 1;     j ← 1
 for k ← p to r
      do if L[ i ] ≤ R[ j ]
            then A[k] ← L[ i ]
                 i ←i + 1
            else A[k] ← R[ j ]
                 j ← j + 1
```



18

Merge Sort Algorithm

MERGE-SORT(A, p, r)

if $p < r$

    $q = \lfloor (p + r)/2 \rfloor$

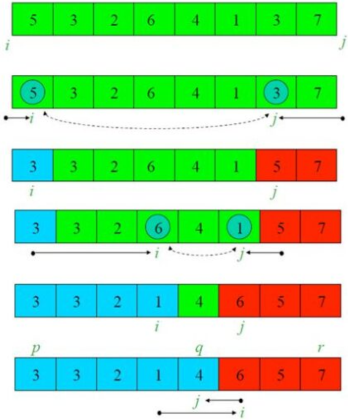
    MERGE-SORT(A, p, q)

    MERGE-SORT$(A, q + 1, r)$

    MERGE(A, p, q, r)

Quick sort- Partition Step

Hoare's Partitioning Algorithm - Ex1 (pivot=5)



Termination: $i = 6; j = 5,$ i.e., $i = j + 1$

Analysis of Algorithms 8

Partition Algorithm- partition(A, lo, hi)

```
pivot = A[lo]
i = lo - 1  // Initialize left index
j = hi + 1  // Initialize right index
while(true){
  do
     i = i + 1;
  while(A[i] < pivot) //Find in left side a value>pivot
  do
     j = j - 1;
  while (A[j] > pivot) //Find in right side a value<pivot
  if i >= j then
     return j
  swap A[i] with A[j]
}
```

Quick Sort Algorithm

```
quicksort( A, low, high)
{
    // base condition
    if (low >= high) {
        return;
    }
    // rearrange elements across pivot
    pivot = partition(a, low, high);

    quicksort(a, low, pivot);

    quicksort(a, pivot + 1, high);
}
```

	0	1	2	3	4	5	6	7	8	9
Partition 0 - 9	4	2	5	14	11	8	15	13	19	10
Partition 0 - 1	2	4	5	14	11	8	15	13	19	10
Partition 3 - 9	2	4	5	8	10	19	13	15	11	14
Partition 5 - 9	2	4	5	8	10	13	11	14	15	19
Partition 5 - 6	2	4	5	8	10	11	13	14	15	19
Partition 8 - 9	2	4	5	8	10	11	13	14	15	19
	2	4	5	8	10	11	13	14	15	19

Lecture # 07

Recap & Goals

Recap...

- Module 1 : Introduction to Algorithms
  - Algorithm : Definition, Features and Role of Algorithms
  - Algorithm to Pseudocode and to Program
  - O, Ω and Θ
  - Recursive Algorithms : Merge Sort, Quick Sort

Today's Goal...

- Recurrence : Merge Sort, Quick Sort

Recurrence

A recurrence is an equation or inequality that describes a function in terms of its value on smaller inputs. Examples

- Factorial : $T(n) = T(n-1) * n$
- Fibonacci : $T(n) = T(n-1) + T(n-2)$
- Binary Search (Worst Case): $T(n) = T(n/2) + 1$

Recurrence of Merge Sort

Time taken to do Merge Sort on n elements is equal to

- Twice the time taken for doing Merge sort on $\frac{n}{2}$ elements
- Time to merge two size $\frac{n}{2}$-sized sorted arrays
- $T(1) = c$, a constant

$$T(n) = \begin{cases} c & \text{if } n = 1. \\ 2T(\frac{n}{2}) + \Theta(n), & \text{otherwise.} \end{cases} \tag{1}$$

Solving Recurrence of Merge Sort

$$
\begin{aligned}
T(n) &= 2T(\frac{n}{2}) + \Theta(n) \\
&\leq 2T(\frac{n}{2}) + cn \\
&\leq 2(2T\frac{n}{4}) + c\frac{n}{2}) + cn \\
&\leq 4T(\frac{n}{4}) + 2cn & &= 2^2T(\frac{n}{2^2}) + 2cn \\
&\leq 8T(\frac{n}{8}) + 3cn & &= 2^3T(\frac{n}{2^3}) + 3cn \\
&\dots \\
&\leq 2^iT(\frac{n}{2^i}) + i \times cn
\end{aligned}
$$

$$T(n) \leq 2^i T(\frac{n}{2^i}) + i \times cn$$

We know $T(1) = c_1$, a constant. We now find value of i, for which $\frac{n}{2^i} = 1$.

$$2^i = n \text{ or } i = \log n$$

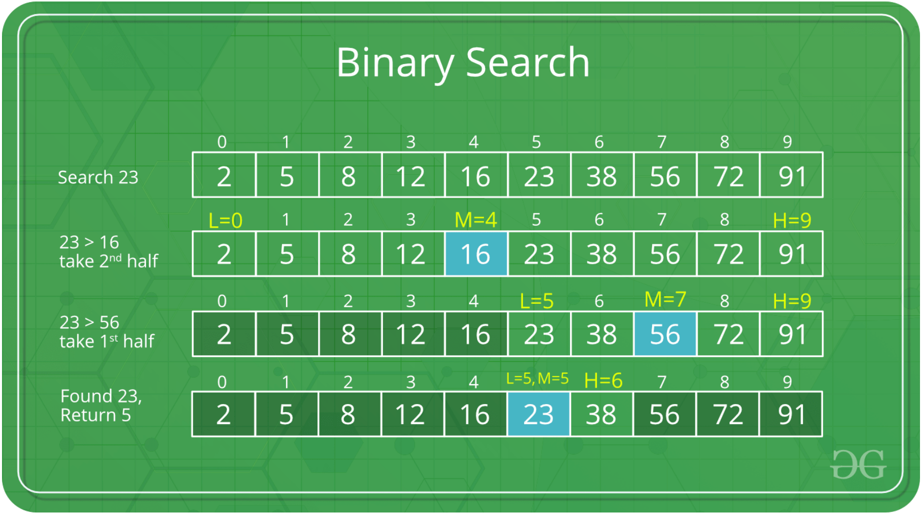
Putting value of i in the above equation we get

$$\begin{aligned}
T(n) &\leq 2^{\log n} T(\frac{n}{2^{\log n}}) + i \times cn \\
&\leq nT(1) + \log n \times cn \\
&\leq n \times c_1 + cn \times \log n \\
&= O(n \log n)
\end{aligned}$$

Searching in Dictionary : Binary Search!

- Search in dictionary for "floccinaucinihilipilification" !
- Do we usually search from page 1 ? No!
- We search at somewhat middle of the book. found hippopotomonstrosesquipedaliophobia!
- So need to search only in the first part of the book.
- What we do is the idea behind - Binary Search!

Binary Search Example



Binary Search

- Check the middle element.
  If found, break.
- else decide which part of the array is relevant and repeat.

Binary Search

- Check the middle element. If found, break.
- else decide which part of the array is relevant and repeat.

```
int n, key;
scanf("%d", &key);
low = 0; high = n-1;
while (low < high) {
   mid = (low + high) / 2;
   if (A[mid] == key) {
      printf("Found at %d", mid);
      break; }
   if (key > A[i]) {
      low = mid+1; }
   else {
      high = mid-1; }
}
```

Recurrence Solution of Binary Search

$$
\begin{aligned}
T(1) &= c_1, \text{a constant} \\
T(n) &= 1 + T(\frac{n}{2}) \\
&\leq c + T(\frac{n}{2}) \\
&\leq c + c + T(\frac{n}{4}) && = 2c + T(\frac{n}{2^2}) \\
&\leq 2c + c + T(\frac{n}{8}) && = 3c + T(\frac{n}{2^3}) \\
&\ldots \\
&\leq i \times c + T(\frac{n}{2^i})
\end{aligned}
$$

We know that the value of i for which $\frac{n}{2^i} = 1$ is $\log n$.

$$T(n) \leq \log n \times c + c_1 = O(\log n)$$

Lecture # 08

Recap & Goals

Recap...

- Module 1 : Introduction to Algorithms
  - Algorithm : Definition, Features and Role of Algorithms
  - Algorithm to Pseudocode and to Program
  - O, Ω and Θ
  - Recursive Algorithms : Merge Sort, Quick Sort

Today's Goal...

- Recurrence : Merge Sort - Worst Case
- Recurrence : Selection Sort

Recurrence Solution of Quick Sort- Worst Case

- Worst case is when the two sub problems are of size $n-1$ and 1
- Happens when the data is in sorted order

$$
\begin{aligned}
T(0) &= c_1, \text{a constant} \\
T(n) &= c * n + T(n-1) \\
&\leq c * n + c * (n-1) + T(n-2) \\
&\leq c(n + (n-1)) + T(n-2) \\
&\leq c(n + n - 1 + n - 2) + T(n-3) \\
&\dots \\
&\leq c(n + n - 1 + n - 2 + \dots n - i + 1) + T(n-i) \\
&\leq c(n + n - 1 + n - 2 + n - 3 + \dots 2 + 1) + T(0) \\
&\leq c(\frac{n(n+1)}{2}) + c_1 \\
&= O(n^2)
\end{aligned}
$$

Recurrence Solution of Selection Sort

- Select the smallest and place it at first position
- Continue the process with the rest of the $n-1$ inputs

$$
\begin{aligned}
T(0) &= c_1, \text{a constant} \\
T(n) &= c*(n-1) + T(n-1) \\
&\leq c*(n-1) + c*(n-2) + T(n-2) \\
&\leq c((n-1)+(n-2)) + T(n-2) \\
&\leq c(n-1+n-2+n-3) + T(n-3) \\
&\dots \\
&\leq c(n-1+n-2+\dots n-i) + T(n-i) \\
&\leq c(n-1+n-2+n-3+\dots 2+1) + T(0) \\
&\leq c(\frac{n(n-1)}{2}) + c_1 \\
&= O(n^2)
\end{aligned}
$$

Lecture # 09

Recap & Goals

Recap...

- Module 1 : Introduction to Algorithms
  - Algorithm : Definition, Features and Role of Algorithms
  - Algorithm to Pseudocode and to Program
  - O, Ω and Θ
  - Recursive Algorithms : Merge Sort, Quick Sort
  - Recurrence : Merge Sort - Worst Case, Selection Sort

Today's Goal...

- Finding Maximum and Minimum

Finding Min & Max : Straight Method

```
FUNCTION MAX-MIN-STRAIGHT (A, low, high)
  min = max = A[low]
  for(i=low+1;i<=high;i++){
    if(A[i] < min) min = A[i];
    if (A[i] > max) max = A[i];
  }
  return (max, min)
```

- Number of comparisons ?

Finding Min & Max : Straight Method

```
FUNCTION MAX-MIN-STRAIGHT (A, low, high)
  min = max = A[low]
  for(i=low+1;i<=high;i++){
    if(A[i] < min) min = A[i];
    if (A[i] > max) max = A[i];
  }
  return (max, min)
```

- Number of comparisons ? $2(n-1)$

Finding Min & Max : Straight Method- improved version

```
FUNCTION MAX-MIN-STRAIGHT (A, low, high)
  min = max = A[low]
  for(i=low+1;i<=high;i++){
    if(A[i] < min){
       min = A[i]);
    }else if (A[i] > max) max = A[i];
  }
  return (max, min)
```

Finding Min & Max : Divide and Conquer Method

```
Function MAXMIN (A, low, high)
    if (high - low + 1 = 2) then
      if (A[low] < A[high]) then
         max = A[high]; min = A[low]
         return((max, min))
      else
         max = A[low]; min = A[high]
         return((max, min))
      end if
   else
      mid = low+high/2
      (max_l , min_l ) = MAXMIN(A, low, mid)
      (max_r , min_r ) =MAXMIN(A, mid + 1, high)
   end if
   Set max to the larger of max_l and max_r ;
   set min to the smaller of min_l and min_r

   return((max, min)).
```

Analysing Divide and Conquer Method of MAXMIN

$$T(n) = \begin{cases} T(\lfloor \frac{n}{2} \rfloor) + T(\lceil \frac{n}{2} \rceil) + 2, & \text{for } n > 2 \\ 1, & \text{for } n = 2 \\ 0, & \text{for } n = 1 \end{cases}$$

Consider $n = 2^k$ for some positive integer$k$, then

$$
\begin{aligned}
T(n) &= 2T(\frac{n}{2}) + 2 \\
&= 2(2T(\frac{n}{4}) + 2) + 2 \\
&= 4T(\frac{n}{4}) + 4 + 2 \\
&= 8T(\frac{n}{8}) + 8 + 4 + 2 \\
&= \dots \\
&= 2^{k-1}T(2) + \Sigma_{1 \le i \le k-1} 2^i \\
&= 2^{k-1} * 1 + 2^k - 2 \\
& \text{we know that the value of } k \text{ is} \log n \\
&= 2^{\log n - 1} + 2^{\log n} - 2 \\
&= \frac{n}{2} + n - 2 \\
&= \frac{3n}{2} - 2
\end{aligned}
$$

Comparing both methods of finding Max & Min

The number of comparisons made by the above two algorithms are

- Straight Method :$2n - 2$
- Divide& Conquer : $\frac{3}{2} * n - 2$
- But Asymptotically both are $O(n)$ algorithms

Lecture # 10

Recap & Goals

Recap...

- Module 1 : Introduction to Algorithms
  - Algorithm : Definition, Features and Role of Algorithms
  - Algorithm to Pseudocode and to Program
  - O, Ω and Θ
  - Recursive Algorithms : Merge Sort, Quick Sort
  - Recurrence : Merge Sort - Worst Case, Selection Sort
- Module 2 : Divide and Conquer

Today's Goal...

- Greedy Algorithms : Kruskal's Minimum Spanning Tree

Lecture # 11

Recap & Goals

Recap...

- Module 1 : Introduction to Algorithms
  - Algorithm : Definition, Features and Role of Algorithms
  - Algorithm to Pseudocode and to Program
  - O, Ω and Θ
- Module 2 : Divide and Conquer
  - Recursive Algorithms : Merge Sort, Quick Sort
  - Recurrence : Merge Sort - Worst Case, Selection Sort
  - Greedy Algorithms : Kruskal's Minimum Spanning Tree

Today's Goal...

- Greedy Algorithms : Prim's Minimum Spanning Tree

Lecture # 12

Recap & Goals

Recap...

- Module 1 : Introduction to Algorithms
  - Algorithm : Definition, Features and Role of Algorithms
  - Algorithm to Pseudocode and to Program
  - O, Ω and Θ
- Module 2 : Divide and Conquer
  - Recursive Algorithms : Merge Sort, Quick Sort
  - Recurrence : Merge Sort - Worst Case, Selection Sort
  - Greedy Algorithms : Kruskal's Minimum Spanning Tree
  - Greedy Algorithms : Prim's Minimum Spanning Tree

Today's Goal...

- Master Theorem : Solving Recurrence
- Strassen's Multiplication

Master Theorem

Theorem

If $T(n) = aT(\lceil \frac{n}{b} \rceil) + O(n^d)$ (for constants $a > 0, b > 1, d \geq 0$), then:

$$T(n) = \begin{cases} O(n^d) & \text{if } d > \log_b a \\ O(n^d \log n) & \text{if } d = \log_b a \\ O(n^{\log_b a}) & \text{if } d < \log_b a \end{cases}$$

Normal Matrix Multiplication

SQUARE-MATRIX-MULTIPLY(A, B)

$n = A.rows$
let C be a new $n \times n$ matrix
for $i = 1$ **to** n
    for $j = 1$ **to** n
        $c_{ij} = 0$
        for $k = 1$ **to** n
            $c_{ij} = c_{ij} + a_{ik} \cdot b_{kj}$
return C

This is an $O(n^3)$ algorithm

Strassen's Multiplication(1969)

p1 = a(f - h)
p2 = (a + b)h
p3 = (c + d)e
p4 = d(g - e)
p5 = (a + d)(e + h)
p6 = (b - d)(g + h)
p7 = (a - c)(e + f)

The A x B can be calculated using above seven multiplications.
Following are values of four sub-matrices of result C

$$\underbrace{\left[\begin{array}{c|c} a & b \\ \hline c & d \end{array}\right]}_{A} \times \underbrace{\left[\begin{array}{c|c} e & f \\ \hline g & h \end{array}\right]}_{B} = \underbrace{\left[\begin{array}{c|c} p5 + p4 - p2 + p6 & p1 + p2 \\ \hline p3 + p4 & p1 + p5 - p3 - p7 \end{array}\right]}_{C}$$

A, B and C are square metrices of size N x N
a, b, c and d are submatrices of A, of size N/2 x N/2
e, f, g and h are submatrices of B, of size N/2 x N/2
p1, p2, p3, p4, p5, p6 and p7 are submatrices of size N/2 x N/2

The algorithm runs in $O(n^{\log 7}) = O(n^{2.8})$