Lecture Slide - 1

CA31: Analysis and Design of Algorithm

- R. Neapolitan & K. Naimipour: Foundations of Algorithms, Jones & Bartlett
- T.H. Cormen etc.: Introduction to Algorithms, PHI

Algorithms

- The word algorithm comes from the name of a Persian mathematician Abu Ja'far Mohammed ibn-i Musa al Khowarizmi.
- An algorithm is step by step operations to perform specific task. An algorithm performs computation, data processing and automated reasoning tasks.
- In other words – An algorithm is a finite sequence of instructions to accomplishes a particular task.

The Characteristics of an Algorithm

- Input – Zero or more quantities are externally supplied.
- Output – At least one quantity is produced.
- Definiteness – Each instruction is clear and unambiguous.
- Finiteness – The algorithm terminates after finite no of steps.
- Effectiveness – Every instructions must be very basic.

Area of Study of Algorithms

- There are following four distinct areas of study of algorithm:

(i) Algorithm Design – designing an algorithm is an art.

(ii) Algorithm Validation – It is process to check whether it give the correct output for all possible legal inputs.

(iii) Algorithm Analysis – It is task to determine the execution time and memory requirements of the algorithm. These are quantitative measurements.

(iv) Testing – It is a process to find the bug in the program corresponding to an algorithm.

Some non-recursive Algorithms

(i) isPrime – To check whether a positive integer is prime.

(ii) TruthTable – To print truth table of n Boolean variables.

(iii) decimalToBinary – To get a binary number equivalent to given decimal number.

(iv) SelectionSort – To arrange the list of integers in non decreasing order using selection sort algorithm.

(v) SequentialSearch – To search a key value in a given list.

(vi) Horner's Rule – To compute the value of a polynomial for x_0.

Non-recursive Alg: isPrime(n)

- It is used to check whether positive integer n is prime or not.
- If n is not divisible by any number between 2 to int(sqrt(n)) then it is prime otherwise not prime.

```
Algorithm isPrime(n){ if(n == 1) then { return False; }
  m = int(sqrt(n));
  isPrime = True;
  For i = 2 to m do{
  {
    If(n % i == 0) Then{
    {
        isPrime = False;
        break;
     }
  }
  return isPrime;
}
```

Non-recursive Alg: TruthTable(n)

- It is used to generate the truth table of n Boolean variables.
- It uses a Boolean array a[1:n] with initial value F.

```
Algorithm TruthTable(n){
  For i = 1 to n do{ a[i] = F; }
  For i = 0 to $2^n$ - 1 do{
  {
    m = i; j = n;
     While (m≠ 0) do{
       If(m % 2 == 0) Then{ a[j] = F; }
       Else { a[j] = T; }
       m = floor(m/2); j = j – 1;
     }
     Print  a[1:n];
   }
}
```

Non-recursive Alg: decimalToBinary(d)

- It is used to convert a positive decimal integer d into equivalent binary integer.

```
Algorithm decimalToBinary(d){
    If ($d \le 0$) then {Throw Exception(“ d must be positive integer”);
    b = 0; p = 1;
    While($d \ne 0$) do{
        b = b + (d % 2) * p;
        p = 10 * p;
        d = floor(d/2);
    }
    return b;
}
```

Non-recursive Alg: selectionSort(a, n)

- It is used to arrange the elements of list of n integers a[1:n] in increasing order.

```
Algorithm selectionSort(a[1:n], n){
  For i = 1 to n-1 do{
    j = i;
    For k = i+1 to n do {
      If(a[k] < a[j]) then{
        j = k;
      }
    }
    If (i ≠ j) then {
      t = a[i];    a[i] = a[j];    a[j] = t;
    }
  }
}
```

Non-recursive Alg: sequentialSearch(a, n, x)

- It is used to search the key x in the list a[1:n] of n integers using sequential search algorithm.

```
Algorithm sequentialSearch(a[1:n], n, x){
  index = -1
  For i = 1 to n do{
    If(a[i] == x) Then{
      index = i
      break;
    }
  }
  return index;
}
```

Non-recursive Alg: Horner(a, n, x0)

- It is used to evaluate a polynomial of degree n at x_0.
- It requires less number of multiplications in comparison to number of multiplication operations in direct method.
- $A(x) = a_3x^3 + a_2x^2 + a_1x + a_0$. Direct method – 6 mul & 3 add. Horner method $A(x) = ((a_3x + a_2)x+a_1)x + a_0$. – 3 mul & 3 add.

```
Algorithm Horner(a[0:n], n, $x_0$) { // a[0:n] have coefficients of polynomial
  Result = 0;
  For i = n to 1, step -1 do{
    Result = (Result + a[i]) * $x_0$;
  }
  Result = Result + a[0];
  return  Result;
}
```

Recursive Algorithms

An algorithm is said to be recursive if it calls itself. For Example

(i)Factorial – To get the factorial of a positive integer.

(ii)GCD – To get the greatest common divisor of two positive integers.

(iii)Fibonacci – To get nth Fibonacci term.

(iv)Permutation – To print all possible permutations of a finite set.

- A problem may be solved using recursive algorithm iff it is defined as recursive and there is a termination condition.

Recursive Alg: Factorial(n)

- This is defined recursively as: $n! = \begin{cases} n \times (n-1)!, & \text{if } n>1 \\ 1, & \text{if } n=1 \end{cases}$

```
Algorithm factorial(n){
  If (n == 1) Then {
    return 1;
  }Else{
    return   n*factorial(n-1);
  }
}
```

Recursive Alg: GCD(n)

- This is defined recursively as: $\gcd(m,n) = \begin{cases} n, & \text{if } m\%n = 0 \\ \gcd(n, m\%n), & \text{if } m\%n \neq 0 \end{cases}$

```
Algorithm gcd(m, n){
  If (m%n == 0) Then {
    return   n;
  }Else{
    return   gcd(n,  m%n);
  }
}
```

Recursive Alg: Fibonacci(n)

➢ This is defined recursively as: $Fib(n) = \begin{cases} 1, & \text{if } n=1 \text{ OR } n=2 \\ Fib(n-1)+Fib(n-2), & \text{if } n>2 \end{cases}$

```
Algorithm Fib(n){
  If (n==1 OR n==2) Then {
    return   1;
  }Else{
    return    Fib(n-1) + Fib(n-2);
  }
}
```

Recursive Alg: Per(a[], k, n)

- The permutation generator algorithm is a recursive algorithm that prints all possible permutations of a given non empty set.
- This is defined recursively as:

$$Per(a[], k, n) = \begin{cases} prints\ a[1:n], & if\ k = n \\ For\ i = k\ to\ n\{ a[i] \leftrightarrow a[k]; Perm(a, k+1, n); a[i] \leftrightarrow a[k]\}; & if\ k < n \end{cases}$$

```
Algorithm Per(a[1:n], k, n){
  If (k == n) Then {
    Print   a[1:n];
  }Else{
    For i = k to n do {
      t = a[k];   a[k] = a[i];   a[i] = t;
      Perm(a, k+1, n);
      t = a[k];   a[k] = a[i];   a[i] = t;
    }
  }
}
```

Analysis of algorithms

Why study algorithms and their performance?

➢Algorithms help us to understand scalability.

➢Performance often draws the line between what is feasible and what is impossible.

➢Analysis of algorithmic is used to compare a number of algorithms for a given problem.

➢To analyze an algorithm, we basically compute space and time complexity.

➢Speed is fun!

Space Complexity of an algorithms

- The space complexity of an algorithm is the amount of memory needed for its completion.
- The space complexity of an algorithm is sum of following two components:

1. Fixed part (c): Which is independent of input and output characteristics that includes space of the code, simple variables, constant etc.
2. Variable part (Sp): Which depends on instance of the problem is to be solved that includes space needed by reference variables, array, recursion stacks etc.

- The space complexity of an algorithm is S(P) = c + Sp

Space Complexity of an algorithms

Example 1.

```
Algorithm abc(a, b, c) {
    return   a + b + b * c + 5.0;
}
```

➢ The space needed by algorithm abc is independent of instance characteristic; □c = 3, Sp = 0, & S(abc) = 3.

Example 2.

```
Algorithm sum(a[], n) {
    s = 0.0;
    For  i = 1 to n do {
            s = s + a[i];
    }
    return  s;
}
```

➢ The space needed by n, i, and s are 3 words (1 word for each); for a[] it is n words. □c = 3, Sp = n, & S(sum)=n+3.

Space Complexity of an algorithms

Example 3.

```
Algorithm RSum(a[], n) {
    If (n ≤ 0) Then {
    {
            return   0.0;
    }Else {
            return   Rsum(a, n-1) + a[n];
    }
}
```

- ➢ Each call of Rsum requires at least 3 words memory space (includes space for variable n, the return address, and a pointer to array a[]). Since the depth of the recursion is n+1, □S(RSum)= 3(n+1).

- ➢ It show that recursive algorithm takes more memory space in comparison to iterative algorithm for same problem.

Time Complexity

- The time complexity of an algorithm is the amount of computer time needed for completion of the algorithm.
- Since total execution time depends on computer and programming language in which it is implemented, therefore in time complexity we compute the number of steps (instructions) instead of exact amount of execution time.
- To determine total steps, first we determine the number of steps per execution (s/e) of the statements and frequency of each steps.

Time Complexity of an algorithms

Example 1.	Statement	s/e	Frequency	Total Steps
1.	Algorithm sum(a[], n) {	0	-	-
2.	s = 0.0;	1	1	1
3.	For i = 1 to n do {	1	n+1	n+1
4.	s = s + a[i];	1	n	n
5.	}	0	-	-
6.	return s;	1	1	1
7.	}	0	-	-

∴ Total number of steps = 2n + 3

Example 2.	Statement	s/e	Frequency		Total Steps	
			n≤0	n>0	n≤0	n>0
1.	Algorithm Rsum(a[], n) {	0				
2.	If (n ≤ 0) then	1	1	1	1	1
3.	return 0.0;	1	1	0	1	0
4.	Else	-	-	-	-	-
5.	return Rsum(a, n-1)+a[n];	1+x	0	1	0	1+x
6.	}	0	-	-	-	

Time Complexity of an algorithms

Example 3.	Statement	s/e	Frequency	Total Steps
1.	Algorithm selectionSort(a[], n) {	0	-	-
2.	For i = 1 to n-1 do {	1	n	n
3.	j = i;	1	n-1	n-1
4.	For k = i+1 to n do {	1	n*(n+1)/2	n*(n+1)/2
5.	If (a[k] < a[j]) then {	1	n*(n-1)/2	n*(n-1)/2
6.	j = k;	1	n*(n-1)/2	n*(n-1)/2
7.	}	0	-	-
8.	}	0	-	-
9.	If (j ≠ i) then {	1	(n-1)	n-1
10.	t=a[j]; a[j]=a[i]; a[i]=t;	3	(n-1)	3(n-1)
11.	}	0	-	-
12.	}	0	-	-
13.	}	0	-	-

∴ Total number of steps = 6n - 5 + n*(n-1) + n*(n+1)/2.

- If we considers only comparisons of the keys, then T(n) = n*(n-1)/2
- The time complexity of this algorithm is a function of problem size n. It

Running Time

- The running time depends on the input: an already sorted list is easier to sort.
- Parameterize the running time by the size of the input: since short list are easier to sort than long ones.
- Generally, we seek upper bounds on the running time: because everybody likes a guarantee.

Kinds of analyses

Worst-case: (usually)

- W(n) = maximum time of algorithm on any input of size n.

Average-case: (sometimes)

- A(n) = expected time of algorithm over all inputs of size n.
- Need assumption of statistical distribution of inputs.

Best-case:

- Cheat with a slow algorithm that works fast on some input.

Every-case:

- T(n) = time of algorithm for all input values.

Asymptotic Notation

The notation we use to describe the asymptotic running time of an algorithm are defined in terms of functions whose domains are the set of natural numbers

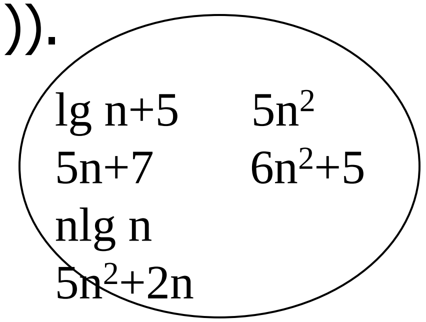
$$\mathbf{N} = \{1, 2, 3, \ldots\}$$

O - Big Ohm (Upper Bound)

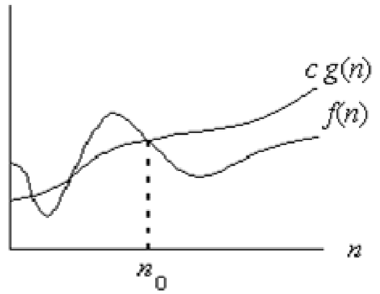
- For an algorithm, instead of computing exact value of T(n), the upper bound of T(n) is more important.
- The asymptotic complexity is a function g(n) that form an upper bound of T(n) for large n, i.e. $T(n) \le c*g(n)$ for $\forall n \ge n_0$ for $\exists c>0, \exists n_0 \in N$.
- In general, just the order of the asymptotic complexity is of the interest i.e. is it linear, quadratic, exponential, factorial functions.
- Let g(n): N □ R be a function, then O(g) is defined as follows:

$$O(g(n)) = \{ f(n) : N \to R \mid \exists c > 0, \exists n_0 \in N, \forall n \ge n_0 : f(n) \le cg(n) \}$$

- This means O(g) is the set of all functions that asymptotically do not grow faster than g.
- If $f(n) \in O(g(n))$, then we say that f(n) is big ohm of g(n) and is also denoted by $f(n) = O(g(n))$.
- $O(n^2)$ contains logarithmic, quadratic, linear functions etc.



$O(n^2$



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

O – Big Ohm: Examples

Example 1. Let $f(n) = 2n^2 + 7n - 10$ and $g(n) = n^2$, then show that $f(n) \in O(g(n))$ or $f(n) = O(g(n))$.

Solution: Since $2n^2 + 7n - 10 \leq 3n^2$, $\forall n \geq 5$; □c=3, and $n_0 = 5$. Therefore, $2n^2 + 7n - 10 \in O(n^2)$. Hence proof.

Example 2. Show that time complexity of the bubble sort is $T(n) = O(n^2)$.

Solution: If we consider only comparisons of keys is to be sorted, then in worst case, $T(n) = n*(n-1)/2 \leq n^2/2$. So, with c=1/2, and $n_0 = 1$, the $T(n) \leq ½*n^2$, $\forall n \geq 1$. Therefore, $T(n) = O(n^2)$. Hence proof.

Example 3. Show that $n^2 + 10n \in O(n^2)$ i.e. $n^2 + 10n = O(n^2)$.

Solution: Since $n^2 + 10n \leq 11n^2$ for $\forall n \geq 1$; □c = 11 and $n_0 = 1$. Therefore, $n^2 + 10n \in O(n^2)$ i.e. $n^2 + 10n = O(n^2)$. Hence proof.

Example 4. Show that $n^2 \in O(n^2 + 10n)$ i.e. $n^2 = O(n^2 + 10n)$.

Solution: Since $n^2 \leq 1*(n^2 + 10n)$ for $\forall n \geq 1$; □c = 1 and $n_0 = 1$. Therefore, $n^2 \in O(n^2+ 10n)$ i.e. $n^2 = O(n^2 + 10n)$. Hence proof.

Example 5. Show that $n \in O(n^2)$ i.e. $n = O(n^2)$.

Solution: Since $n \leq 1*n^2$ for $\forall n \geq 1$; □c = 1 and $n_0 = 1$. Therefore, $n \in O(n^2)$ i.e. $n = O(n^2)$. Hence proof.

Ω - Big Omega (Lower Bound)

- Once an algorithm for a problem is found the question arises whether it is possible to design a faster algorithm. In such a case the lower bound of the time complexity of the existing algorithm is required.
- The asymptotic complexity is a function g(n) that form an lower bound of T(n) for large n, i.e. $T(n) \geq c*g(n)$ for $\forall n \geq n_0$ for $\exists c>0$, $\exists n_0 \in N$.
- Let g(n): N □ R be a function, then Ω(g) is defined as follows:

$$\Omega(g(n)) = \{ f(n) : N \rightarrow R \mid \exists c > 0, \exists n_0 \in N, \forall n \geq n_0 : f(n) \geq cg(n) \}$$

- This means Ω(g) is the set of all functions that asymptotically grow as least as fast as g.
- If $f(n) \in \Omega(g(n))$, then we say that f(n) is big omega of g(n) and is also denoted by $f(n) = \Omega(g(n))$.
- $\Omega(n^2)$ contains quadratic, exponential, factorial etc. functions.

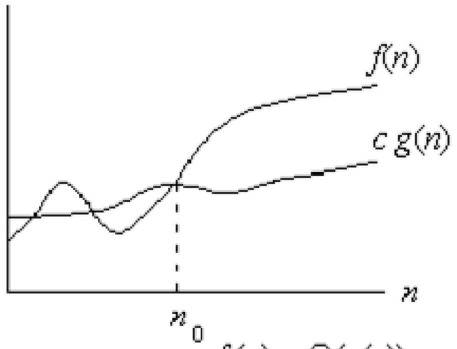
$4n^2$

$4n^3+3n^2$

$5n^2+7$ $\quad$ 2^n+n

$2n^2+7n$ $\quad$ $n!+2n$

$\Omega(n^2)$



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

Ω – Big Omega: Examples

Example 1. Show that $2n^2 + 7n - 10 \in \Omega(n^2)$ or $2n^2 + 7n - 10 = \Omega(n^2)$.

Solution: Since $2n^2 + 7n - 10 \geq 1 * n^2$, $\forall n \geq 2$; □c=1, and $n_0 = 2$. Therefore, $2n^2 + 7n - 10 \in \Omega(n^2)$. Hence proof.

Example 2. Show that time complexity of the bubble sort is $T(n) = \Omega(n)$.

Solution: If we consider only comparisons of keys is to be sorted, then in best case, $T(n) = (n-1) \leq n$. So, with c=1, and $n_0 = 1$, the $T(n) \leq n$, $\forall n \geq 1$. Therefore, $T(n) = \Omega(n)$. Hence proof.

Example 3. Show that $5n^2 \in \Omega(n^2)$ i.e. $5n^2 = \Omega(n^2)$.

Solution: Since $5n^2 \geq 1 * n^2$ for $\forall n \geq 1$; □c = 1 and $n_0 = 1$. Therefore, $5n^2 \in \Omega(n^2)$ i.e. $5n^2 = \Omega(n^2)$. Hence proof.

Example 4. Show that $n^3 \in \Omega(n^2)$ i.e. $n^3 = \Omega(n^2)$.

Solution: Since $n^3 \geq 1*(n^2)$ for $\forall n \geq 1$; □c = 1 and $n_0 = 1$. Therefore, $n^3 \in \Omega(n^2)$ i.e. $n^3 = \Omega(n^2)$. Hence proof.

Example 5. Show that $2^n \in \Omega(n^2)$ i.e. $2^n = \Omega(n^2)$.

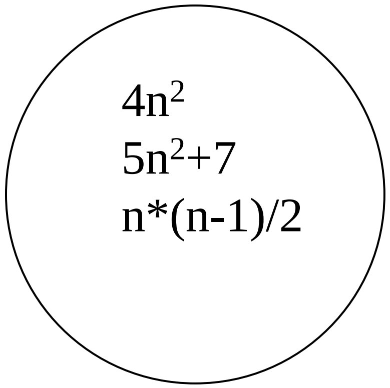
Solution: Since $2^n \geq 1 * n^2$ for $\forall n \geq 4$; □c = 1 and $n_0 = 1$. Therefore, $2^n \in \Omega(n^2)$ i.e. $2^n = \Omega(n^2)$. Hence proof.

Θ - Big Theta (Exact Order)

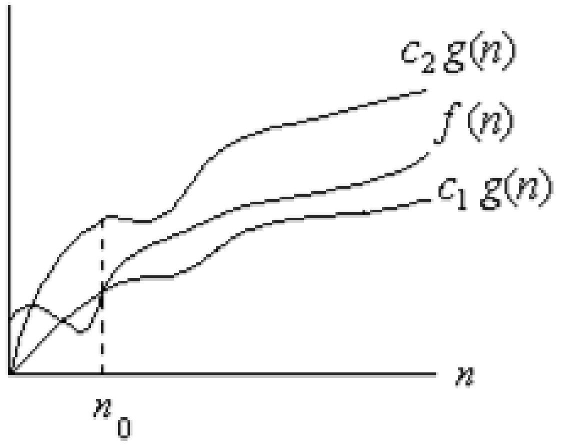
- It is used to express exact order of a function.
- For a given complexity function g(n): Θ(g(n)) = O(g(n)) ∩ Ω(g(n)).
- Let g(n): N □ R be a function, then Θ(g) is defined as follows:

$$\Theta(g(n)) = \{ f(n) : N \rightarrow R \mid \exists c_0, c_1 > 0, \exists n_0 \in N, \forall n \geq n_0 : c_0 g(n) \leq f(n) \leq c_1 g(n) \}$$

- If f(n) ∊ Θ(g(n)), then we say that f(n) is order of g(n) and is also denoted by f(n) = Θ(g(n)).
- $\Theta(n^2)$ contains only quadratic functions.



$\Theta(n^2)$



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

Θ – Big Theta: Examples

Example 1. Show that $2n^2 + 7n - 10 \in \Theta(n^2)$ or $2n^2 + 7n - 10 = \Theta(n^2)$.

Solution: Since $1*n^2 \leq 2n^2 + 7n – 10 \leq 3*n^2$, $\forall n \geq 5$; □c0=1, c1=3, and $n_0 = 5$. Therefore, $2n^2 + 7n – 10 \in \Theta(n^2)$. Hence proof.

Example 2. Show that time complexity of the selection sort is $T(n) = \Theta(n^2)$.

Solution: If we consider only comparisons of keys is to be sorted, then $T(n) = n*(n-1)$, and since $T(n) \in O(n^2)$ and $T(n) \in \Omega(n^2)$ both; □$T(n) \in \Theta(n^2)$ or $T(n) = \Theta(n^2)$. Hence proof.

Example 3. Show that $n \notin \Theta(n^2)$.

Solution: It can be proved by contradiction.

Let $n \in \Theta(n^2)$; □$n \in \Omega(n^2)$. Therefore there exist some positive constant c and $n_0 \in N$ such that for $\forall n \geq n_0$, $n \geq cn^2$.

If we divide both side of this inequality by cn then we gets for $\forall n \geq n_0$,

$1/c \geq n$. But for any $n > 1/c$, this inequality cannot be hold. It means, the inequality $n \geq cn^2$ cannot hold for $\forall n \geq n_0$. So, $n \notin \Omega(n^2)$ and hence $n \notin \Theta(n^2)$. Hence proof.

o – Little oh

- Let g(n): N □ R be a function, then o(g) is defined as follows:

$$o(g(n)) = \{ f(n) : N \rightarrow R \mid \forall c \in R^{+}, \exists n_0 \in N, \forall n \geq n_0 : f(n) \leq cg(n) \}$$

- The function f(n) ∈ o(g(n)), iff $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0.$

Example 1. Show that 3n+2 ∈ o(n^2).

Solution: Since $\lim_{n\to\infty} \frac{3n+2}{n^2} = \lim_{n\to\infty} \frac{3}{2n} = 0.$ So, 3n+2 ∈ o(n^2). Hence proof.

Example 2. Show that 3n+2 ∈ o(n log n).

Solution: Since $\lim_{n\to\infty} \frac{3n+2}{n\log n} = \lim_{n\to\infty} \frac{3}{1+\log n} = 0.$ So, 3n+2 ∈ o(n log n). Hence proof.

Example 3. Show that 2^n+n^2 ∈ o(3^n).

Solution: Since $\lim_{n\to\infty} \frac{2^n+n^2}{3^n} = \lim_{n\to\infty} \left(\frac{2}{3}\right)^n + \lim_{n\to\infty} \frac{n^2}{3^n} = 0 + \lim_{n\to\infty} \frac{2n}{3^n \times \lg 3} = \lim_{n\to\infty} \frac{2}{3^n \times (\lg 3)^2} = 0.$

So, 2^n+n^2 ∈ o(3^n). Hence proof.

Example 4. Show that 3n+2 ∉ o(n).

Solution: Since $\lim_{n\to\infty} \frac{3n+2}{n} = \lim_{n\to\infty} \frac{3}{1} = 3 \neq 0.$ So, 3n+2 ∉ o(n). Hence proof.

ω – Little omega

- Let g(n): N ☐ R be a function, then ω(g) is defined as follows:

$$\omega(g(n)) = \left\{ f(n) : N \to R \mid \forall c \in R^+, \exists n_0 \in N, \forall n \geq n_0 : f(n) \geq cg(n) \right\}$$

- The function f(n) ∈ ω(g(n)), iff $\lim_{n\to\infty} \frac{g(n)}{f(n)} = 0.$

Example 1. Show that $3n^2+2 \in \omega(n)$.

Solution: Since $\lim_{n\to\infty} \frac{n}{3n^2+2} = \lim_{n\to\infty} \frac{1}{6n} = 0.$ So, $3n^2+2 \in \omega(n)$. Hence proof.

Example 2. Show that $3n^2+2 \notin \omega(n^2)$.

Solution: Since $\lim_{n\to\infty} \frac{n^2}{3n^2+2} = \lim_{n\to\infty} \frac{2n}{6n} = \frac{1}{3}.$ So, $3n^2+2 \notin \omega(n^2)$. Hence proof.

Example 3. Show that n log n ∈ ω(n).

Solution: Since $\lim_{n\to\infty} \frac{n}{n\log n} = \lim_{n\to\infty} \frac{1}{\log n} = 0.$ So, n log n ∈ ω(n). Hence proof.

Example 4. Show that $3^n \in \omega(2^n)$.

Solution: Since $\lim_{n\to\infty} \frac{2^n}{3^n} = \lim_{n\to\infty} \left(\frac{2}{3}\right)^n = 0.$ So, $3^n \in \omega(2^n)$. Hence proof.

Properties of Order

1. $f(n) \in O(g(n))$ iff $g(n) \in \Omega(f(n))$.
2. $f(n) \in \Theta(g(n))$ iff $g(n) \in \Theta(f(n))$.
3. If a, b>1, then $\log_a n \in \Theta(\log_b n)$. This implies that all logarithmic complexity functions are in the same complexity category and this category is represented by $\Theta(\lg n)$.
4. If $b > a > 0$, then $a^n \in O(b^n)$. This show that all exponential functions are not in same complexity category.
5. For all a>0, $a^n \in o(n!)$. This implies that n! is worse than any exponential complexity functions.
6. The order of complexity functions are: $\Theta(1)$, $\Theta(\lg n)$, $\Theta(n)$, $\Theta(n \lg n)$, $\Theta(n^2)$, $\Theta(n^j)$, $\Theta(n^k)$, $\Theta(a^n)$, $\Theta(b^n)$, $\Theta(n!)$. Where $k > j > 2$ and $b > a > 1$.
7. If $c \geq 0$, $d > 0$, $f(n) \in O(g(n))$, and $h(n) \in \Theta(g(n))$; then $c*f(n) + d*h(n) \in \Theta(g(n))$

Order Notation Examples & Limit Method to determine order

Example 1. If b > a > 0, then show that $a^n \in o(b^n)$.

Solution: Since $\lim_{n\to\infty} \frac{a^n}{b^n} = \lim_{n\to\infty}\left(\frac{a}{b}\right)^n = 0$. So, $a^n \in o(b^n)$. Hence proof.

Example 2. Using properties of the order, show that $3\lg n + 10n \lg n + 7n^2 \in \Theta(n^2)$.

If b > a > 0, then show that $a^n \in o(b^n)$.

Solution: We know that $n^2 \in \Theta(n^2)$ and $n \lg n \in O(n^2)$. By using property 7 and take c=10, d=7, we have $10n \lg n + 7n^2 \in \Theta(n^2)$. We know that $\lg n \in O(n^2)$, so by taking c=3 and d=1 and applying property 7, we gets $3\lg n + 10n \lg n + 7n^2 \in \Theta(n^2)$. Hence proof.

<u>Limit method to determine order of a function</u>

$$\lim_{n\to\infty}\frac{f(n)}{g(n)} = \begin{cases} c, \text{ where } c > 0 & \Rightarrow f(n) \in \Theta(g(n)) \\ 0 & \Rightarrow f(n) \in o(g(n)) \\ \infty & \Rightarrow f(n) \in \omega(g(n)) \end{cases}$$

Master Theorem

- The master theorem may be used to solve recurrences of the form T(n) = aT(n/b) + f(n) that occurs in divide and conquer algorithm analysis, where a≥1, b>1 are natural numbers and f(n) is asymptotically positive function.

Master Theorem

If T(n) = aT(n/b) + f(n) with a≥1, b>1, then T(n) can be bounded asymptotically as follows:

Case 1. If $f(n) \in O\left(n^{\log_b a-\varepsilon}\right)$ for some constant ε>0, then $T(n) \in \Theta\left(n^{\log_b a}\right)$.

Case 2. If $f(n) \in \Theta\left(n^{\log_b a}\right)$, then $T(n) \in \Theta\left(n^{\log_b a} \lg n\right)$.

Case 3. If $f(n) \in \Omega\left(n^{\log_b a+\varepsilon}\right)$ for some constant ε>0 and it satisfy the regularity condition a*f(n/b)≤c*f(n) for some constant c<1 and all sufficiently large n, then $T(n) \in \Theta(f(n))$.

- Here we compare f(n) with $n^{\log_b a}$. (i) $T(n) \in \Theta\left(n^{\log_b a}\right)$, if $n^{\log_b a}$ is polynomial larger than f(n); (iii) $T(n) \in \Theta(f(n))$, if f(n) is polynomial larger than $n^{\log_b a}$;and (ii) $T(n) \in \Theta\left(n^{\log_b a} \lg n\right)$ if both are polynomial equal. In case (iii) it should also satisfy the regularity condition a*f(n/b)≤c*f(n) for some constant c<1 and all sufficiently large n.

Master Theorem - Examples

Example 1. Solve the recurrence T(n) = 9T(n/3) + n using master theorem.

Solution: Here a = 9, b = 3, f(n)=n. $\because\ n^{\log_b a} = n^{\log_3 9} = n^2$. Since $f(n) \in O\left(n^{\log_b a-\varepsilon}\right)$ with ε=1, so by applying case 1 of the master theorem, we gets T(n) = Θ(n^2).

Example 2. Solve the recurrence T(n) = T(2n/3) + 1 using master theorem.

Solution: Here a = 1, b = 3/2, f(n)=1. $\because\ n^{\log_b a} = n^{\log_{3/2} 1} = n^0 = 1$. Since $f(n) \in \Theta\left(n^{\log_b a}\right)$, so by applying case 2 of the master theorem, we gets T(n) = Θ(lg n).

Example 3. Solve the recurrence T(n) = 3T(n/4) + n lg n using master theorem.

Solution: Here a = 3, b = 4, f(n)=n lg n. $\because\ n^{\log_b a} = n^{\log_4 3} = n^{0.793}$. Since $f(n) \in \Omega\left(n^{\log_b a+\varepsilon}\right)$ with ε=0.3, so case 3 of the master theorem may be apply if we show that it satisfy the regularity condition. For sufficient large n, a*f(n/b) = 3f(n/4) = 3*(n/4)*lg(n/4) < (3/4) * n lg n ≤ c* f(n)) where c = 3/4. So, the regularity condition satisfied and using case 3 of the master theorem we gets T(n) = Θ(n lg n).

Master Theorem - Examples

Example 4. Solve the recurrence T(n) = 2T(n/2) + n lg n using master theorem.

Solution: Here a = 2, b = 2, f(n)=n lg n. $\because n^{\log_b a} = n^{\log_2 2} = n.$ Since $n^{\log_b a}$ is asymptotically smaller than f(n) but it is not polynomially smaller. So, this recurrence fall into the gate of case 2 and case 3 and master theorem cannot be apply.

Example 5. The recurrence T(n) = 7T(n/2) + n^2 is represent the running time of an algorithm A. A competing algorithm A′ has running time T′(n) = aT′(n/4) + n^2. What is the largest integer value for a such that A′ is asymptotically faster than algorithm A.

Solution: For algorithm A, a = 7, b = 2, f(n)=n^2. $\because n^{\log_b a} = n^{\log_2 7} = n^{2.8074}.$

Since $f(n) \in O\left(n^{\log_b a - a}\right)$ with ε=0.8074, so by applying case 1 of the master theorem, we gets T(n) = Θ($n^{2.8074}$). The algorithm A′ will faster than A if T′(n) = Θ(n^2). For this $\because n^{\log_b a} \le f(n) \Rightarrow n^{\log_b a} \le n^2 \Rightarrow \log_4 a \le 2 \Rightarrow a = 16$

Master Theorem - Examples

Example 6. Show that time complexity of binary search algorithm is T(n) = O(lg n).

Solution: The worst case time complexity of binary search algorithm is written as T(n) = T(n/2) + O(1). Here a = 1, b = 2, f(n)= 1. $n^{\log_b a} = n^{\log_2 1} = n^0 = 1$. Since $f(n) \in \Theta\left(n^{\log_b a}\right)$, so by applying case 2 of the master theorem, we gets T(n) = O(lg n). Hence Proof. [Note: Since it is worst case time complexity of the binary search algorithm, so we have represented it in O instead of Θ]

Example 7. Show that time complexity of binary tree traversal (in-order, pre-order, and post-order) algorithms are T(n) = Θ(n).

Solution: The every case time complexity of binary tree traversal algorithms are written as T(n) = 2T(n/2) + O(1). Here a = 2, b = 2, f(n)= 1. $n^{\log_b a} = n^{\log_2 2} = n$. Since $f(n) \in O\left(n^{\log_b a - \varepsilon}\right)$ with ε=1, so by applying case 1 of the master theorem, we gets T(n) = Θ(n). Hence Proof. [Note: Since it is every case time complexity of the binary tree traversal algorithms, so we have represented it in Θ instead of O]

Example 8. Show that time complexity of merge sort algorithm is T(n) = Θ(n).

Solution: The every case time complexity of merge sort algorithm is written as T(n) = 2T(n/2) + O(n). Here a = 2, b = 2, f(n)= n. $n^{\log_b a} = n^{\log_2 2} = n$. Since $f(n) \in \Theta\left(n^{\log_b a}\right)$, so by applying case 2 of the master theorem, we gets T(n) = Θ(n lg n). Hence Proof.

Master Theorem – not applicable

The following equations cannot be solved using the master theorem:

- $T(n) = 2^n T(n/2) + n^n$. Here $a = 2^n$ is number of sub-problems that is not a constant. The number of sub-problems must be fixed. So, this recurrence cannot be solved using master's theorem.
- $T(n) = 2T(n/2) + n/(\log n)$. Here a=2, b=2, and $f(n) = n/(\log n)$. $\because n^{\log_b a} = n^{\log_2 2} = n$. Since $\frac{n^{\log_b a}}{f(n)} = \frac{n}{n/\log n} = \log n$ i.e. $n^{\log_b a}$ is asymptotically larger than f(n) but it is not polynomially larger. So, this recurrence fall into the gate of case 2 and case 3 and master theorem cannot be apply.
- $T(n) = 0.5T(n/2) + n$. Here a < 1 and it is not a whole number. Number of sub-problems cannot be in fraction and master theorem cannot be apply.
- $T(n) = 64T(n/8) – n^2$. The combination time f(n) cannot be negative and master theorem cannot be apply.
- $T(n) = T(n/2) + n(2 – \cos n)$. Here a=1, b=2, and $f(n) = n(2 – \cos n)$. $\because n^{\log_b a} = n^{\log_2 1} = n^0 = 1$. It is Case 3, but regularity violation. Let for c<1, $a*f(n/b) \le c*f(n)$. But $a*f(n/b) = f(n/2) = (n/2)*(2 – \cos(n/2))$. □$(n/2)*(2 – \cos(n/2)) \le c*n(2 – \cos n)$□ $(1/2)*(2 – \cos(n/2)) \le c*(3 – 2\cos^2(n/2))$. Let for large n, $\cos(n/2) = -1$; then $(1/2)*(2 – (-1)) \le c*(3 – 2*(-1)^2)$ □ $3/2 \le c$ □ $c > 1$. Which contradict that c<1, so regularity condition does not hold. So, case 3 of master theorem cannot be apply.

Divide and Conquer Approach

- In this approach, the original problem breaks into several sub-problems that are similar to original problem but smaller in size, solve the sub-problems recursively and then combine their solutions to get the solution of the original problem.
- This approach involves three steps at each level of recursion:

1. Divide the problem into a number of sub-problems.
2. Conquer the sub-problems by solving them recursively. If size of the sub-problems are small enough, then it solve in a straight forward manner.
3. Combine the solutions of the sub-problems to get the solution of the original problem.

- The Control abstraction of the divide and conquer paradigm is:

```
Algorithm DAndC(P) { If (P is small) then { return solution(P); }
                     Else {   divide P into smaller sub-problems P1, P2, ..., Pk;
                              Apply DAndC to each of these sub-problem;
                              return Combine(DAndC(P1), DAndC(p2), ..., DAndC(Pk));
                     }
}
```

Analysis of Divide and Conquer Algorithm

- The time complexity of divide and conquer algorithm can be written in following recurrence equation:

$$T(n) = \begin{cases} \Theta(1), & if\ n \le c \\ aT\left(\frac{n}{b}\right) + D(n) + C(n), & otherwis \end{cases}$$

- If size of a sub-problem is less than equal to c (small enough), then it is solve in straight forward and take constant time i.e. T(n) = Θ(1). Otherwise it divides the problem into a number of sub-problems of size n/b each, solve then recursively and combine their result to get the solution of the original problem. If D(n) is the execution time for dividing the problem into sub-problems, and C(n) is the execution time to combine the solution; then T(n) = aT(n/b) + D(n) + C(n). The sum of D(n) and C(n) may be represented by f(n).

Divide and Conquer: Binary Search

- Let a_i, $1 \le i \le n$, be a sorted list. The binary search is a problem to find the index of an element x, in this list. Since it divide the list into two equal sub-lists and x may be exist in either of these two sub-lists, so it is a divide and conquer algorithm.
- This function should be called with low=0, and high=n-1;

```
Algorithm binarySearch(a[], x, low, high){
If(low ≤ high) Then{
      mid = floor((low + high) / 2);
      If(x == a[mid]) then{ return mid; }
      Else if(x < a[mid]) then { return binarySearch(a, x, low, mid-1); }
      Else {return binarySearch(a, x, mid+1, high); }
   } Else{
      return -1;
   }
}
```

Worst Case Time Analysis of Binary Search

- Here, if size of P is 1, then it is solve in straight forward method by comparing the x with element of the list and it takes constant time i.e. T(n) = Θ(1).
- If size of the P is n for n > 1, then it divides this problem in a sub-problem of size n/2. The time for dividing the problem into sub-problem is constant, so D(n) = Θ(1).
- Here, answer of sub-problem should be the answer of original problem, so combination of the solution does not needed and C(n) = 0.
- The worst case time complexity of binary search algorithm is defined by following recurrence equation:

$$T(n) = \begin{cases} \Theta(1), & if\ n=1 \\ T(n/2)+\Theta(1), & otherwis \end{cases}$$

- It may solve suing master's theorem. Here a=1, b=2, and f(n) 1.

  $\because\ n^{\log_b a} = n^{\log_2 1} = n^0 = 1$. Since $f(n) \in \Theta\left(n^{\log_b a}\right)$, so by applying case 2 of the master theorem, we gets T(n) = O(lg n). [Note: Since it is worst case time complexity of the binary search algorithm, so we have represented it in O instead of Θ]

Divide and Conquer: Merge Sort

- It is another example of divide and conquer approach. It operates as follows:

Divide: divide the list of n elements into two sub-lists of n/2 elements each.

Conquer: sort the two sub-lists recursively using merge sort.

Combine: Merge two sorted sub-lists to get the sorted list of original list.

- The main operation of merge sort is merging of two sorted sub-lists.

```
Algorithm merge(a[low: high], low, mid, high){ i = low; j = mid+1; k = 0;
  b[high – low + 1];
  While(i ≤ mid AND j ≤ high) do{
    If (a[i] ≤ a[j]) then { b[k] = a[i] ; i = i + 1; }
    Else{ b[k] = a[j]; j = j + 1; }
    k = k + 1;
  }
  While(i ≤ mid) do{ b[k] = a[i] ; i = i + 1;  k = k + 1;}
  While(j ≤ high) do{ b[k] = a[j] ; j = j + 1;  k = k + 1;}
  For k = 0 to high – low do { a[low+k] = b[k]; }
}
```

- The merge() operation takes Θ(n) execution time, where n = high-low+1 is the number of elements being merged.

Divide and Conquer: Merge Sort

- The MergeSort() recursive algorithm is used for dividing the problem into sub-problems and combing them.
- It divides the problem P of size n into two sub-problems of size n/2 each and it again call MergeSort() algorithm for solving both sub-problems and merge() algorithm.
- The MergeSort() algorithm is listed as below. It will be calls by passing the argument low=0 and high=n-1.

```
Algorithm MergeSort(a[low: high], low, high){
  If(low < high) then {
    // Divide P into sub-problem
    mid = floor((low + high)/2);
    // Solve sub-problems
    MergeSort(a, low, mid);
    MergeSort(a, mid+1, high);
    // combine the solution
    merge(a, low, mid, high);
  }
}
```

Every Case Time Analysis of Merge Sort

- The running time T(n) of this algorithm is breaks into following components:

Divide: The divide step computes the middle of array, which takes constant execution time i.e. D(n) = Θ(1).

Conquer: Since we recursively solve sub-problems, each of size n/2, which contributes 2T(n/2) to the execution time.

Combine: The merge algorithm is used for this purpose that takes Θ(n) execution time, so C(n) = n.

- Therefore, the time every case time complexity of this algorithm is represented as below.

$$T(n) = \begin{cases} \Theta(1), & if\ n=1 \\ 2T(n/2)+\Theta(n), & if\ n>1 \end{cases}$$

- It may be solve suing master's theorem. Here a = 2, b = 2, f(n)=n.$\because n^{\log_b a} = n^{\log_2 2} = n$. Since $f(n) \in \Theta\left(n^{\log_b a}\right)$, so by applying case 2 of the master theorem, we gets T(n) = Θ(n lg n).

[Note: Since it is every case time complexity of the merge sort, so we have represented it in Θ instead of O]

Divide and Conquer: Quick Sort

- It is another example of divide and conquer strategy.
- It partition the list into two sub-lists using pivot element. All elements smaller than pivot element placed into first sub-list and all elements larger than or equal to pivot element placed into second sub-list. Thereafter, each sub-list sort recursively.
- There are different way to choose pivot element, but for simplicity we have chosen first element as pivot element.
- The main operation in quick sort is partition that is given as below:

```
Algorithm partition(a[low: high], low, high){
  pivotElement = a[low];    j = low;
  For i = low + 1 to high do{
    If(a[i] < pivotElement) then {   j = j + 1;
      If(i ≠ j) then { exchange a[i] ↔ a[j]; }
    }
  }
  pivotPoint = j;
  If(low ≠ pivotPoint) then { exchange a[low] ↔ a[pivotPoint] }
  return pivotPoint;
}
```

Divide and Conquer: Quick Sort

- The quichSort() algorithm partition the list into two sub-lists and sort them recursively.

```
Algorithm quickSort(a[low: high], low, high){
  If(low < high) then { pivotPoint = partition(a, low, high);
    quickSort(a, low, pivotPoint-1);      quickSort(a, pivotPoint+1, high);
  }
}
```

Given list: 15, 22, 13, 27, 12, 10, 20, 25. low=0, high=7, PE=15, j=0; i=1. i=2 □ j=1□ $a_2 \leftrightarrow a_1$, 15, 13, 22, 27, 12, 10, 20, 25. i=3. i=4 □ j=2 □ $a_4 \leftrightarrow a_2$, 15, 13, 12, 27, 22, 10, 20, 25. i=5 □ j=3 □ $a_5 \leftrightarrow a_3$, 15, 13, 12, 10, 22, 27, 20, 25. i=6. i=7.
PP=j=3; $a_0 \leftrightarrow a_3$ □ 10 , 13, 12, 15, 22, 27, 20, 25.
low=0, high=2, PE=10, j=0; i=1. i=2. PP=j=0 □ 10 , 13, 12, 15, 22, 27, 20, 25.
low=1, high=2, PE=13, j=1; i=2□ j=2. PP=j=2; $a_1 \leftrightarrow a_2$ □ 10 , 12, 13, 15, 22, 27, 20, 25.
low=4, high=7, PE=22, j=4; i=5. i=6 □ j=5 □ $a_6 \leftrightarrow a_5$ □ 10 , 12, 13, 15, 22, 20, 27, 25.
i=7. PP=j=5; $a_4 \leftrightarrow a_5$ □ 10 , 12, 13, 15, 20, 22, 27, 25.
low=6, high=7, PE=27, j=6; i=7 □ j=7. PP=j=7; $a_6 \leftrightarrow a_7$ □ 10 , 12, 13, 15, 20, 22, 25, 27.

Performance of Quick Sort

- The time complexity of the quick sort depends on whether the partitioning is balanced or unbalanced.
- If the partitioning is balanced, the algorithm runs asymptotically as fast as merge sort.
- If the partitioning is unbalanced, it can run asymptotically as slow as insertion sort.

Worst Case Time Complexity

- The worst case behaviour for quick sort occurs when partition algorithm produce one sub-problem with n-1 elements and other with zero element. Let us assume that unbalanced partitioning occurs in each recursive call.
- The worst case time complexity of quick sort is defined as

T(n) = T(n-1) + n, where Θ(n) = n is the time complexity of the partition algorithm.

T(n) = T(n-1) + n = T(n-2) + (n-1) + n = T(n-3) + (n-2) + (n-1) + n = ... = T(1) + 2 + 3 + ... + (n-1) + n = 1 + 2 + 3 + ... + (n-1) + n = n*(n+1)/2. □T(n) = O(n^2)

Best Case Time Complexity

- If partition algorithm produce two sub-problems, each of size n/2, then it will best case and time complexity is defined by recurrence relation T(n) = 2*T(n/2) + Θ(n). It mat solve suing master's theorem. Here a = 2, b = 2, f(n)=n. $\because n^{\log_b a} = n^{\log_2 2} = n$ Since $f(n) \in \Theta\left(n^{\log_b a}\right)$ so by applying case 2 of the master theorem, we gets T(n)=Ω(n lg n).

Indicator Random Variable

- It is used to analyze many algorithms.
- It provide a method for conversion between probability and expectation.
- Suppose S is a sample space and A is an event, then indicator random variable associated with event A is denoted by I{A} and defined as $I\{A\} = \begin{cases} 1, & \text{if } A \text{ occurs} \\ 0, & \text{if } A \text{ does not occur} \end{cases}$
- Expected value of occurrence of event A should be expected value of corresponding indicator random variable and defined as E[I{A}] = 1*Pr(A) + 0*Pr(A′).

Example: What is the expected number of heads in tossing of a fair coin.

Solution: Here S = {H, T}. Let Y is a random variable whose value may be H or T each with probability 1/2 i.e. Pr(Y=H) = Pr(Y=T) = ½.

Let X_H is an indicator random variable associated with coming of head in tossing of a coin i.e. X_H = I{Y=H}. Then X_H = 1, if Y=H, and X_H = 0, if Y=T.

The expected number of heads in tossing a coin is simply expected value of indicator random variable X_H.

$E(X_H) = E[I\{Y=H\}] = 1*Pr(Y=H) + 0*Pr(Y=T) = 1*(1/2) + 0*(1/2) = ½$

Thus expected number of head in tossing of a fair coin is ½.

Indicator Random Variable

Example: Let X be a random variable define the number of heads in two flips of coin. Compute $E[X^2]$ and $E^2[X]$.

Solution: Let Y_i is random variable denoting the outcome of the ith flip of coin; and let $X_i = I\{Y_i=H\}$ is indicator random variable that outcome of ith flip of coin is head.

Then, $X_i = 1$ if $Y_i = H$ and $X_i = 0$ if $Y_i = T$.

As X is random variable denoting the number of heads in two flips of a coin.

$\therefore X = \sum_{i=1}^{2} X_i. \quad \therefore E[X] = E\left(\sum_{i=1}^{2} X_i\right) = \sum_{i=1}^{2} E[X_i]$ by linearity of expectation.

But $E[X_i] = \Pr(Y_i = H) = ½$. $\therefore E[X] = \sum_{i=1}^{2} E[X_i] = \sum_{i=1}^{2} \frac{1}{2} = \frac{1}{2} + \frac{1}{2} = 1 \quad \therefore E^2[X] = E[X] \times E[X] = 1 * 1 = 1$

$$E[X^2] = E\left[\left(\sum_{i=1}^{2} X_i\right)^2\right] = E\left[(X_1 + X_2)^2\right]$$

But $(X_1 + X_2)^2 = X_1^2 + X_2^2 + X_1X_2 + X_2X_1 = \sum_{i=1}^{2} X_i^2 + \sum_{i=1}^{2} \sum_{\substack{j=1 \\ j \neq i}}^{2} X_iX_j$

$$\therefore E[X^2] = E\left[\left(\sum_{i=1}^{2} X_i\right)^2\right] = E\left[\sum_{i=1}^{2} X_i^2 + \sum_{i=1}^{2} \sum_{\substack{j=1 \\ j \neq i}}^{2} X_iX_j\right] = \sum_{i=1}^{2} E[X_i^2] + \sum_{i=1}^{2} \sum_{\substack{j=1 \\ j \neq i}}^{2} E[X_iX_j]$$

But $E[X_i^2] = X_i^2 * \Pr(y_i=H) + X_i^2 * \Pr(Y_i=T) = 1^2*(1/2) + 0^2*(1/2) = ½$. And for $i \neq j$, X_i and X_j are independent. □$E[X_i.X_j] = E[X_i]*E[Xj] = ½*1/2=1/4$.

□$E[X^2] = 2 * ½ + 2*1/4 = 1 + ½ = 3/2$.

Average Case Time Complexity of Quick Sort

- In partition algorithm of the quick sort, a pivot element is selected and it is not included in the partitions, thus there should be maximum of n-1 calls of this algorithm and number of comparisons in each call of partition algorithm depends on number of elements in sub-lists.
- Let X is total number of comparisons performed in all calls of the partition algo.
- Let we rename the element of the array A[] as $z_1, z_2, z_3, \ldots, z_n$ where z_i is the ith smallest element. Let set $Z_{ij} = \{z_i, z_{i+1}, z_{i+2}, \ldots, z_j\}$ be the set of elements between z_i and z_j.
- Let X_{ij} is indicator random variable defined as $X_{ij} = I\{z_i \text{ is compared to } z_j\}$
- Since each pair of elements compared at most once $\therefore\ X = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} X_{ij}$

$$\therefore\ E[X] = E\left[\sum_{i=1}^{n-1} \sum_{j=i+1}^{n} X_{ij}\right] = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} E\left[X_{ij}\right] = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \Pr\{z_i \text{ is compared to } z_j\}$$

- If x is pivot element such that $z_i < x < z_j$, then z_i and z_j cannot be compared.
- If z_i is chosen as pivot element before any other element in Z_{ij}, then z_i will compared to each elements in Z_{ij} except itself.
- Similarly if z_j is chosen as pivot element before any other element in Z_{ij}, then z_j will compared to each elements in Z_{ij} except itself.
- Thus z_i and z_j are compared iff the first element to chosen as pivot from Z_{ij} is either z_i or z_j.

Average Case Time Complexity of Quick Sort

- Any element of set Z_{ij} is equally likely to be the first one chosen as pivot.
- Since number of elements in set Z_{ij} is j-i+1, therefore the probability that any given element is the first one chosen as a pivot element is 1/(j-i+1).

□P_r(z_i is compared to z_j) = P_r(z_i is first pivot chosen from Z_{ij}) + P_r(z_j is first pivot chosen from Z_{ij}) = 2/(j-i+1)

$$\therefore E[X] = E\left[\sum_{i=1}^{n-1}\sum_{j=i+1}^{n} X_{ij}\right] = \sum_{i=1}^{n-1}\sum_{j=i+1}^{n} \Pr\{z_i \text{ is compared to } z_j\} = \sum_{i=1}^{n-1}\sum_{j=i+1}^{n} \frac{2}{j-i+1}$$

$$= \left(\frac{2}{2}+\frac{2}{3}+\cdots+\frac{2}{n}\right) + \left(\frac{2}{2}+\frac{2}{3}+\cdots+\frac{2}{n-1}\right) + \left(\frac{2}{2}+\frac{2}{3}+\cdots+\frac{2}{n-2}\right) + \cdots + \left(\frac{2}{2}\right) = (n-1)\frac{2}{2} + (n-2)\frac{2}{3} + (n-3)\frac{2}{4} + \cdots + (n-n+1)\frac{2}{n}$$

$$= 2n\left(\frac{1}{2}+\frac{1}{3}+\cdots+\frac{1}{n}\right) - 2\left(\frac{1}{2}+\frac{2}{3}+\frac{3}{4}+\cdots+\frac{(n-1)}{n}\right) < 2n\lg n$$

- Therefore, the average time complexity of quick sort = Expected number of comparisons = O(n lg n).

Strassen's Matrix Multiplication

Direct Method

- If A and B are two square matrices of order n, then their product AB requires n^3 multiplications and $n^2(n-1) = n^3-n^2$ additions.
- The algorithm is given as below:

```
Algorithm mul(A, B){
   n = order(A);
   For i = 1 to n do{
       For j = 1 to n do {
          c[i][j] = 0;
          For k =  1 to n do {
              c[i][j] = c[i][j] + a[i][k] * b[k][j];
          }
       }
   }
   return C;
}
```

Strassen's Matrix Multiplication

Divide and Conquer Method

- Let n be the order of square matrices is to be multiplied, where n is power of 2; then these matrices are divided into 4 sub-matrices of order n/2 each.
- The resultant matrices are computed by multiplying 8 matrices of order n/2 and then addition of 4 matrices of order n/2 each.

$$\begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix} = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \times \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix} = \begin{bmatrix} A_{11} \times B_{11} + A_{12} \times B_{21} & A_{11} \times B_{12} + A_{12} \times B_{22} \\ A_{21} \times B_{11} + A_{22} \times B_{21} & A_{21} \times B_{12} + A_{22} \times B_{22} \end{bmatrix}$$

- Since two matrices of order n/2 may be added in time cn^2. so, the time complexity of this algorithm for product of matrices A and B is T(n) given as

T(n) = 8T(n/2) + Θ(n2) that may be solved using master's theorem. Here a=8, b=2, f(n) = n^2. $n^{\log_b a} = n^{\log_2 8} = n^3$. Since $f(n) \in O\left(n^{\log_b a-\varepsilon}\right)$ with ε=1, so by applying case 1 of the master theorem, we gets T(n) = $\Theta(n^3)$.

Strassen’s Matrix Multiplication

Divide and Conquer Method – Verification:

- Let order of matrices A and B are 4x4. Then

$$A_{11}=\begin{bmatrix}a_{11}&a_{12}\\a_{21}&a_{22}\end{bmatrix}, A_{12}=\begin{bmatrix}a_{13}&a_{14}\\a_{23}&a_{24}\end{bmatrix}, A_{21}=\begin{bmatrix}a_{31}&a_{32}\\a_{41}&a_{42}\end{bmatrix}, A_{22}=\begin{bmatrix}a_{33}&a_{34}\\a_{43}&a_{44}\end{bmatrix}, B_{11}=\begin{bmatrix}b_{11}&b_{12}\\b_{21}&b_{22}\end{bmatrix}, B_{12}=\begin{bmatrix}b_{13}&b_{14}\\b_{23}&b_{24}\end{bmatrix}, B_{21}=\begin{bmatrix}b_{31}&b_{32}\\b_{41}&b_{42}\end{bmatrix}, B_{22}=\begin{bmatrix}b_{33}&b_{34}\\b_{43}&b_{44}\end{bmatrix}$$

$$A_{11}\times B_{11}=\begin{bmatrix}a_{11}b_{11}+a_{12}b_{21}&a_{11}b_{12}+a_{12}b_{22}\\a_{21}b_{11}+a_{22}b_{21}&a_{21}b_{12}+a_{22}b_{22}\end{bmatrix} \quad A_{12}\times B_{21}=\begin{bmatrix}a_{13}b_{31}+a_{14}b_{41}&a_{13}b_{32}+a_{14}b_{42}\\a_{23}b_{31}+a_{24}b_{41}&a_{23}b_{32}+a_{24}b_{42}\end{bmatrix} \quad C_{11}=A_{11}\times B_{11}+A_{12}\times B_{21}=\begin{bmatrix}a_{11}b_{11}+a_{12}b_{21}+a_{13}b_{31}+a_{14}b_{41}&a_{11}b_{12}+a_{12}b_{22}+a_{13}b_{32}+a_{14}b_{42}\\a_{21}b_{11}+a_{22}b_{21}+a_{23}b_{31}+a_{24}b_{41}&a_{21}b_{12}+a_{22}b_{22}+a_{23}b_{32}+a_{24}b_{42}\end{bmatrix}$$

$$A_{11}\times B_{12}=\begin{bmatrix}a_{11}b_{13}+a_{12}b_{23}&a_{11}b_{14}+a_{12}b_{24}\\a_{21}b_{13}+a_{22}b_{23}&a_{21}b_{14}+a_{22}b_{24}\end{bmatrix} \quad A_{12}\times B_{22}=\begin{bmatrix}a_{13}b_{33}+a_{14}b_{43}&a_{13}b_{34}+a_{14}b_{44}\\a_{23}b_{33}+a_{24}b_{43}&a_{23}b_{34}+a_{24}b_{44}\end{bmatrix} \quad C_{12}=A_{11}\times B_{12}+A_{12}\times B_{22}=\begin{bmatrix}a_{11}b_{13}+a_{12}b_{23}+a_{13}b_{33}+a_{14}b_{43}&a_{11}b_{14}+a_{12}b_{24}+a_{13}b_{34}+a_{14}b_{44}\\a_{21}b_{13}+a_{22}b_{23}+a_{23}b_{33}+a_{24}b_{43}&a_{21}b_{14}+a_{22}b_{24}+a_{23}b_{34}+a_{24}b_{44}\end{bmatrix}$$

$$A_{21}\times B_{11}=\begin{bmatrix}a_{31}b_{11}+a_{32}b_{21}&a_{31}b_{12}+a_{32}b_{22}\\a_{41}b_{11}+a_{42}b_{21}&a_{41}b_{12}+a_{42}b_{22}\end{bmatrix} \quad A_{22}\times B_{21}=\begin{bmatrix}a_{33}b_{31}+a_{34}b_{41}&a_{33}b_{32}+a_{34}b_{42}\\a_{43}b_{31}+a_{44}b_{41}&a_{43}b_{32}+a_{44}b_{42}\end{bmatrix} \quad C_{21}=A_{21}\times B_{11}+A_{22}\times B_{21}=\begin{bmatrix}a_{31}b_{11}+a_{32}b_{21}+a_{33}b_{31}+a_{34}b_{41}&a_{31}b_{12}+a_{32}b_{22}+a_{33}b_{32}+a_{34}b_{42}\\a_{41}b_{11}+a_{42}b_{21}+a_{43}b_{31}+a_{44}b_{41}&a_{41}b_{12}+a_{42}b_{22}+a_{43}b_{32}+a_{44}b_{42}\end{bmatrix}$$

$$A_{21}\times B_{12}=\begin{bmatrix}a_{31}b_{13}+a_{32}b_{23}&a_{31}b_{14}+a_{32}b_{24}\\a_{41}b_{13}+a_{42}b_{23}&a_{41}b_{14}+a_{42}b_{24}\end{bmatrix} \quad A_{22}\times B_{22}=\begin{bmatrix}a_{33}b_{33}+a_{34}b_{43}&a_{33}b_{34}+a_{34}b_{44}\\a_{43}b_{33}+a_{44}b_{43}&a_{43}b_{34}+a_{44}b_{44}\end{bmatrix} \quad C_{22}=A_{21}\times B_{12}+A_{22}\times B_{22}=\begin{bmatrix}a_{31}b_{13}+a_{32}b_{23}+a_{33}b_{33}+a_{34}b_{43}&a_{31}b_{14}+a_{32}b_{24}+a_{33}b_{34}+a_{34}b_{44}\\a_{41}b_{13}+a_{42}b_{23}+a_{43}b_{33}+a_{44}b_{43}&a_{41}b_{14}+a_{42}b_{24}+a_{43}b_{34}+a_{44}b_{44}\end{bmatrix}$$

$$C=\begin{bmatrix}C_{11}&C_{12}\\C_{21}&C_{22}\end{bmatrix}=\begin{bmatrix}a_{11}b_{11}+a_{12}b_{21}+a_{13}b_{31}+a_{14}b_{41}&a_{11}b_{12}+a_{12}b_{22}+a_{13}b_{32}+a_{14}b_{42}&a_{11}b_{13}+a_{12}b_{23}+a_{13}b_{33}+a_{14}b_{43}&a_{11}b_{14}+a_{12}b_{24}+a_{13}b_{34}+a_{14}b_{44}\\a_{21}b_{11}+a_{22}b_{21}+a_{23}b_{31}+a_{24}b_{41}&a_{21}b_{12}+a_{22}b_{22}+a_{23}b_{32}+a_{24}b_{42}&a_{21}b_{13}+a_{22}b_{23}+a_{23}b_{33}+a_{24}b_{43}&a_{21}b_{14}+a_{22}b_{24}+a_{23}b_{34}+a_{24}b_{44}\\a_{31}b_{11}+a_{32}b_{21}+a_{33}b_{31}+a_{34}b_{41}&a_{31}b_{12}+a_{32}b_{22}+a_{33}b_{32}+a_{34}b_{42}&a_{31}b_{13}+a_{32}b_{23}+a_{33}b_{33}+a_{34}b_{43}&a_{31}b_{14}+a_{32}b_{24}+a_{33}b_{34}+a_{34}b_{44}\\a_{41}b_{11}+a_{42}b_{21}+a_{43}b_{31}+a_{44}b_{41}&a_{41}b_{12}+a_{42}b_{22}+a_{43}b_{32}+a_{44}b_{42}&a_{41}b_{13}+a_{42}b_{23}+a_{43}b_{33}+a_{44}b_{43}&a_{41}b_{14}+a_{42}b_{24}+a_{43}b_{34}+a_{44}b_{44}\end{bmatrix}$$

Strassen's Matrix Multiplication

Strassen's Method

- It is also based on divide and conquer approach and published in 1969.
- Like divide and conquer method of matrix multiplication, it also divides the matrices of order n=2k into four matrices of order n/2 each as below and then compute the components C11, C12, C21, and C22 of the resultant matrices using 1-11.

$$\begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix} = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \times \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix}$$

$M_1=(A_{11}+A_{22})(B_{11}+B_{22})$...(1) $M_2=(A_{21}+A_{22})B_{11}$ ...(2) $M_3=A_{11}(B_{12}-B_{22})$ ... (3) $M_4=A_{22}(B_{21}-B_{11})$ ... (4) $M_5=(A_{11}+A_{12})B_{22}$ ... (5) $M_6=(A_{21}-A_{11})(B_{11}+B_{12})$ ... (6) $M_7=(A_{12}-A_{22})(B_{21}+B_{22})$ ... (7)

$C_{11}=M_1+M_4-M_5+M_7$; $C_{12}=M_3+M_5$; $C_{21}=M_2+M_4$; and $C_{22}=M_1+M_3-M_2+M_6$ ... (8-11)

- It requires 7 matrices multiplications of order n/2 and 18 addition/subtractions of matrices of order n/2 each.
- So, if T(n) is the time complexity in terms of multiplications then T(n) = 7T(n/2) + Θ(n2) that may be solved using master's theorem. Here a=7, b=2, f(n) = n^2. $\because n^{\log_b a} = n^{\log_2 7} = n^{2.81}$. Since $f(n) \in O\left(n^{\log_b a-\varepsilon}\right)$ with ε=.81, so by applying case 1 of the master theorem, we gets T(n) = $\Theta(n^{2.81})$.
- It show that it is better than direct or divide and conquer method.

Strassen’s Matrix Multiplication

Strassen’s Method – Verification

$M_1=(A_{11}+A_{22})(B_{11}+B_{22})$…(1) $M_2=(A_{21}+A_{22})B_{11}$ …(2) $M_3=A_{11}(B_{12}-B_{22})$ … (3)
$M_4=A_{22}(B_{21}-B_{11})$ … (4) $M_5=(A_{11}+A_{12})B_{22}$ … (5) $M_6=(A_{21}-A_{11})(B_{11}+B_{12})$ … (6)
$M_7=(A_{12}-A_{22})(B_{21}+B_{22})$ … (7)

$C_{11}=M_1+M_4-M_5+M_7 = (A_{11}+A_{22})(B_{11}+B_{22}) + A_{22}(B_{21}-B_{11}) - (A_{11}+A_{12})B_{22} + (A_{12}-A_{22})(B_{21}+B_{22}) = A_{11}B_{11} + \cancel{A_{11}B_{22}} + \cancel{A_{22}B_{11}} + \cancel{A_{22}B_{22}} + \cancel{A_{22}B_{21}} - \cancel{A_{22}B_{11}} - \cancel{A_{11}B_{22}} - \cancel{A_{12}B_{22}} + A_{12}B_{21} + \cancel{A_{12}B_{22}} - \cancel{A_{22}B_{21}} - \cancel{A_{22}B_{22}} = A_{11}B_{11} + A_{12}B_{21}$

$C_{12}=M_3+M_5=A_{11}(B_{12}-B_{22}) + (A_{11}+A_{12})B_{22} = A_{11}B_{12} - \cancel{A_{11}B_{22}} + \cancel{A_{11}B_{22}} + A_{12}B_{22} = A_{11}B_{12} + A_{12}B_{22}$

$C_{21}=M_2+M_4 = (A_{21}+A_{22})B_{11} + A_{22}(B_{21}-B_{11}) = A_{21}B_{11} + \cancel{A_{22}B_{11}} + A_{22}B_{21} - \cancel{A_{22}B_{11}} = A_{21}B_{11} + A_{22}B_{21}$

$C_{22}=M_1+M_3-M_2+M_6 = M_1+M_3-M_2+M_6 = (A_{11}+A_{22})(B_{11}+B_{22}) + A_{11}(B_{12}-B_{22}) - (A_{21}+A_{22})B_{11}+(A_{21}-A_{11})(B_{11}+B_{12}) = \cancel{A_{11}B_{11}} + \cancel{A_{11}B_{22}} + \cancel{A_{22}B_{11}} + A_{22}B_{22} + \cancel{A_{11}B_{12}} - \cancel{A_{11}B_{22}} - \cancel{A_{21}B_{11}} - \cancel{A_{22}B_{11}} + \cancel{A_{21}B_{11}} + A_{21}B_{12} - \cancel{A_{11}B_{11}} - \cancel{A_{11}B_{12}} = A_{21}B_{12} + A_{22}B_{22}$