

| | | | |
|------------------------------|---|---|--|
| <u>S.No.</u> | | | |
| | 1 Algorithm | Program | |
| | Design time | Implementation time | |
| | Domain knowledge | Programmer | |
| | Any language even English and Maths | Programming language | |
| | Hardware and software independent | Hardware and operating system dependent | |
| | Analyze an algorithm | Testing of programs | |
| | | | |
| | 2 Priori Analysis | Posterior Testing | |
| | Algorithm | Program | |
| | Independent of language | Language dependent | |
| | Hardware independent | Hardware dependent | |
| | Time and space function | watch time and bytes | |
| | | | |
| | 3 Characteristics of algorithm | | |
| | Zero or more inputs | | |
| | Must generate atleast one output | | |
| | Definiteness | | |
| | Finiteness | | |
| | Effectiveness | | |
| | | | |
| | 4 How to analyze an algorithm | | |
| | Time | | |
| | Space | | |
| | Network consumption : Data transfer amount | | |
| | Power consumption | | |
| | CPU registers | | |
| | | | |
| | 5 Frequency Count Method | Used for time snalysis of an algorithm | |
| | Assign 1 unit of time for each statement | | |
| | For any repitition, calculate the frequency of repetition | | |

| | | | |
|--|--|---|--|
| | for(i = 0; i < n; i++) --> condition is checked for n+1 times | 2n + 2 units of time ~ n+1 as we see condition i < n only for now | |
| | any statement within the loop will execute for n times | | |
| | | | |
| | Space complexity depends upon number and kind of variables used | | |
| | | | |
| | | | |
| | 6 Algorithm : sum(A, n) | | |
| | Single for loop - | | |
| | Time complexity: O(N) | | |
| | Space complexity: O(N) | | |
| | | | |
| | 7 Algorithm : Add(A, B, n) | Sum of two square matrices of dimensions nXn | |
| | Two nested for loops - | | |
| | Time complexity: O(N^2) | | |
| | Outer for loop executes for N+1 times | | |
| | Inner for loop executes for N * (N+1) times | | |
| | Any statement within inner for loop executes for (N + 1) * (N + 1) times | | |
| | Space complexity: O(N^2) | | |
| | | | |
| | 8 Algorithm : Multiply(A, B, n) | | |
| | Three nested for loops - | | |
| | Time complexity: O(N^3) | | |
| | Space complexity: O(N^2) | | |
| | | | |
| | 9 Different algorithm conditions | | |
| | For loops | | |
| | for(i = n; i > 0; i--) | n+1 times | |
| | for(i = 0; i < n; i = i + 2) | n/2 times | |

| | | | |
|--|---|---|--|
| | 2 nested for loops where both i and j range from 0 to n | n^2 times | |
| | 2 nested for loops where j ranges from 0 to i | when $i = 0$; j loop repeats 0 times; when $i = 1$; j loop repeats 1 times; and so on...total number of repetitions: $0 + 1 + 2 + 3 + 4 + \dots + n = O(n^2)$ | |
| | $p = 0$; for($i = 1$; $p \leq n$; $i++$){ $p = p + i$; } | $p = k(k+1)/2 \rightarrow$ assuming that the loop exits when p is greater than n $\rightarrow k(k+1) / 2 > n$ | $\sim k^2 > n \rightarrow O(\sqrt{n})$ |
| | for($i = 1$; $i < n$; $i = i * 2$) | will execute for 2^k times | $O(\log n)$ |
| | | Assume $i \geq n$; $i = 2^k \geq n$ | |
| | | $k = \log n$ with base 2 | |
| | for($i = n$; $i \geq 1$; $i = i/2$) | i | |
| | | n | |
| | | $n/2$ | |
| | | $n/2^2$ | |
| | | $n/2^3$ | |
| | | | |
| | | $n/2^k$ | |
| | | Assume $i < 1 \Rightarrow n / 2^k < 1$ | $\sim O(\log n)$ with base 2 |
| | for($i = 0$; $i * i < n$; $i++$) | $i^2 < n$ | |
| | | $i^2 > -n$ | |
| | | $i^2 = n \rightarrow i = \sqrt{n}$ | $\sim O(\sqrt{n})$ |
| | for($i = 0$; $i < n$; $i++$) {.....}for($j = 0$; $j < n$; $j++$){.....} | $O(n)$ | |
| | $p = 0$; for($i = 1$; $i < n$; $i * 2$){.....} for($j = 1$; $j < p$; $j * 2$){.....} | $\log n$ times for upper loop; $\log p$ times for lower loop | $\sim O(\log(\log n))$ |
| | for($i = 0$; $i < n$; $i++$) {.....for($j = 0$; $j < n$; $j * 2$){.....}} | Outer loop repeats n times; inner loop repeats $\log n$ times | $\sim O(n \log n)$ |
| | for($i = 1$; $i < n$; $i = i * 3$) | | $\sim O(\log n)$ with base 3 |
| | | | |
| | While loops | | |
| | while vs. do while | do while will execute for minimum one time | |
| | for and while are almost similar | do while will execute as long as the condition is true; for loop will execute until the condition is false | |

| | | | |
|--|--|--|----------------------|
| | a = 1; | | |
| | while(a < b){ a = a *2;} | 1, 2, 2^2, 2^3.....2^k repetitions | ~O(logb) with base 2 |
| | | assume a > b; 2^k > b ==> k = logb with base 2 | |
| | | | |
| | i = n; while(i > 1) {...i = i/2;} | | ~O(logn) with base 2 |
| | i = 1; k = 1; while(k < n){....k = k + i; i++;} | | |
| | | i k | |
| | | 1 | 1 |
| | | 2 1 + 1 | |
| | | 3 2 + 2 | |
| | | 4 2 + 2 + 3 | |
| | | 5 2 + 2 + 3 + 4 | |
| | | | |
| | m | m(m + 1) /2 | |
| | | | |
| | Assume, k >= n | m(m + 1)/ 2 >= n | ~O(root(n)) |
| | | | |
| | while(m != n) { if(m > n) m = m - n; else n = n - m;} | | ~O(n) |
| | | | |
| | | | |
| | 10 Types of time functions | | |
| | O(1) --- constant | | |
| | O(logn) --- logarithmic | | |
| | O(n) --- linear | | |
| | O(n^2) --- quadratic | | |
| | O(n^3) --- cubic | | |
| | O(2^n) --- exponential | | |
| | | | |
| | 11 Order of complexity | | |
| | 1 < logn < root(n) < n < nlogn < n^2 < n^3 <.....< 2^n < 3^n< n^n | | |

| | | | |
|-----------|--|--|---|
| | | | |
| 12 | Asymptotic Notations | | |
| | Representation of time complexity in simple form which is understandable | | |
| | Big O Notation - works as an upper bound | The function $f(n) = O(g(n))$ iff for all positive constants c and n_0 , such that $f(n) \leq c * g(n)$ for all $n \geq n_0$; here, $f(n) = O(n)$ | e.g. $2n + 3 \leq 10n$; All those functions in time order complexity above n become upper bound; below n become lower bound and n is the average bound |
| | Big Omega Notation - works as a lower bound | The function $f(n) = \Omega(g(n))$ iff for all positive constants c and n_0 , such that $f(n) \geq c * g(n)$ for all $n \geq n_0$; here, $f(n) = \Omega(n)$ | e.g. $2n + 3 \geq 1n$ |
| | Theta Notation - works as an average bound | The function $f(n) = \theta(g(n))$ iff for all positive constants c_1, c_2 and n_0 such that $c_1 * g(n) \leq f(n) \leq c_2 * g(n)$ | e.g. $f(n) 2n + 3; 1n \leq 2n + 3 \leq 5n$ |
| | Most useful is theta notation, then why do we need the other two? | In case we are not able to get the average bound, then we point to its upper or lower bound | |
| | | | |
| 13 | Examples for asymptotic notations | | |
| a | $f(n) = 2n^2 + 3n + 4$ | | |
| | $2n^2 + 3n + 4 \leq 2n^2 + 3n^2 + 4n^2$ i.e. $9n^2$ | $O(n^2)$ | |
| | $2n^2 + 3n + 4 \geq 1n^2$ | $\Omega(n^2)$ | |
| | $1n^2 \leq 2n^2 + 3n + 4 \leq 9n^2$ | $\Theta(n^2)$ | |
| | | | |
| b | $f(n) = n^2 \log n + n$ | | |
| | $n^2 \log n \leq n^2 \log n + n \leq 10n^2 \log n$ | $O(n^2 \log n)$ | |
| | | $\Omega(n^2 \log n)$ | |
| | | $\Theta(n^2 \log n)$ | |
| | | | |
| c | $f(n) = n!$ | | |
| | $1 \leq 1 * 2 * 3 * 4 * \dots * n - 1 * n \leq n * n * n * \dots * n$ | $O(n^n)$ | |
| | | $\Omega(1)$ | |
| | | Cannot find theta for $n!$ | |

| | | | |
|----------|---|---|--|
| | | | |
| d | $f(n) = \log n!$ | | |
| | $1 \leq \log(1 \cdot 2 \cdot 3 \dots \cdot n) \leq \log(n \cdot n \cdot n \dots \cdot n)$ | $O(\log n^n)$ | |
| | | $\Omega(1)$ | |
| | | Cannot find theta for $\log n!$ | |
| | | | |
| | 14 Properties of Asymptotic notations | | |
| | General properties - | | |
| | if $f(n)$ is $O(g(n))$ then $a \cdot f(n)$ is $O(g(n))$ | | |
| | e.g. $f(n) = 2n^2 + 5$ is $O(n^2)$, then $7f(n)$ i.e. $14n^2 + 35$ is also $O(n^2)$ | This would be true for both Ω and θ as well | |
| | | | |
| | Reflexive property - | | |
| | If $f(n)$ is given then $f(n)$ is $O(f(n))$ | | |
| | e.g. $f(n) = n^2$ then $O(n^2)$ | A function is an upper bound of itself | |
| | | Similarly, a function is a lower bound of itself | |
| | | | |
| | Transitive property - | | |
| | If $f(n)$ is $O(g(n))$ and $g(n)$ is $O(h(n))$ then $f(n)$ is $O(h(n))$ | | |
| | e.g. $f(n) = n$; $g(n) = n^2$ and $h(n) = n^3$ | True for all notations | |
| | n is $O(n^2)$ and n^2 is $O(n^3)$ then n is $O(n^3)$ | | |
| | | | |
| | Symmetric property - | | |
| | If $f(n)$ is $\theta(g(n))$ then $g(n)$ is $\theta(f(n))$ | True for only $\theta(n)$ | |
| | e.g. $f(n) = n^2$ $g(n) = n^2$; $f(n) = \theta(n^2)$ and $g(n) = \theta(n^2)$ | | |
| | | | |
| | Transpose symmetric - | True for BigO and Ω notations | |
| | if $f(n) = O(g(n))$ then $g(n)$ is $\Omega(f(n))$ | | |
| | e.g. $f(n) = n$ and $g(n)$ is n^2 then n is $O(n^2)$ and n^2 is $\Omega(n)$ | | |

| | | | |
|--|---|---------------------------------|--|
| | | | |
| | If $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$ then $g(n) \leq f(n) \leq g(n)$ therefore $f(n) = \theta(g(n))$ | | |
| | | | |
| | If $f(n) = O(g(n))$ and $d(n) = O(e(n))$ then $f(n) + d(n) = O(\max(g(n), e(n)))$ | | |
| | e.g. $f(n) = n = O(n)$, $d(n) = n^2 = O(n^2)$ then $f(n) + d(n) = n + n^2 = O(n^2)$ | | |
| | | | |
| | If $f(n) = O(g(n))$ and $d(n) = O(e(n))$ then $f(n) * d(n) = O(g(n) * e(n))$ | | |
| | | | |
| | 15 Comparison of functions | | |
| | First method is substituting values for n and comparing | | |
| | Second method is applying log on both sides | | |
| | | Properties of log - | |
| | Example - | $\log a b = \log a + \log b$ | |
| | $f(n) = n^2 \log n$; $g(n) = n(\log n)^{10}$ | $\log a / b = \log a - \log b$ | |
| | Apply log | $\log a^b = b \log a$ | |
| | $\log(n^2 \log n)$; $\log(n(\log n)^{10})$ | $a^{(\log cb)} = b^{(\log ca)}$ | |
| | $\log(n^2) + \log \log n$; $\log n + \log \log^{10} n$ | $a^b = n$ then $b = \log_a n$ | |
| | $2 \log n + \log \log n$; $\log n + 10 \log \log n$ | | |
| | here; $2 \log n$ is greater than $\log n$ and $\log n$ is a bigger term than $\log \log n$ | | |
| | so, first term is greater than the second one | | |
| | | | |
| | $f(n) = 3n^{(\sqrt{n})}$; $g(n) = 2^{(\sqrt{n} \log_2(n))}$ | | |
| | Applying log | | |
| | $3n^{(\sqrt{n})}$; $(n^{\sqrt{n}}) \log_2(2)$ | | |
| | $3n^{(\sqrt{n})}$; $n^{\sqrt{n}}$ | | |
| | first term is greater than the second one value wise but asymptotically they are equal | | |

| | | | |
|----------|--|---|--|
| | | | |
| | $f(n) = n^{(\log n)}$; $g(n) = 2^{(\sqrt{n})}$ | | |
| | apply log, | | |
| | $\log(n^{\log n})$; $\log(2^{\sqrt{n}})$ | | |
| | $\log n \cdot \log n$; $\sqrt{n} (\log_2 2)$ | | |
| | $\log^2 n$; \sqrt{n} | | |
| | cannot judge, so apply log again | | |
| | $2 \log \log n$; $1/2 \log n$ | | |
| | $\log \log n$ is smaller than $\log n$ | | |
| | thus, second term is greater | | |
| | | | |
| | $f(n) = 2^{(\log n)}$; $g(n) = n^{(\sqrt{n})}$ | | |
| | $\log n \cdot \log_2 2$; $\sqrt{n} \cdot \log n$ | | |
| | $\log n$; $\sqrt{n} \cdot \log n$ | | |
| | second term is greater | | |
| | | | |
| | $f(n) = 2n$; $g(n)$ is $3n$ | | |
| | both are equal asymptotically | | |
| | | | |
| | $f(n) = 2^n$; $g(n) = 2^{(2n)}$ | | |
| | applying log | | |
| | $\log(2^n)$; $\log(2^{2n})$ | | |
| | n ; $2n$ | after applying log, do not cut coefficients | |
| | second function is greater | | |
| | | | |
| | 16 Best, worst and average case analysis | | |
| | Example - | | |
| a | Linear search | | |
| | $A = \{8, 6, 12, 5, 9, 7, 4, 3, 16, 18\}$ key = 7 | | |
| | In linear search, it will start checking for the given key from left hand side | | |

| | | | |
|-----------|---|--|--|
| | total in 6 comparisons, we would get our key | | |
| | Best case - key element is present at first index | | |
| | Best case time - 1 i.e. $B(n) = O(1)$; $\Omega(1)$; $\Theta(1)$ | | |
| | Worst case - key element is present at the last index | | |
| | Worst case time - n i.e. $W(n) = O(n)$; $\Omega(n)$; $\Theta(n)$ | | |
| | Average case = all possible case time / no. of cases | | |
| | average case analysis is very difficult for most of the cases | | |
| | Here, average case time = $1 + 2 + 3 + \dots + n/2 = n(n+1)/2n = n+1/2$ | | |
| | $A(n) = n+1/2$ | | |
| | | | |
| b | Binary search tree | | |
| | height = $\log n$ | | |
| | time taken for a particular key is $\log n$ | | |
| | Best case - element present in the root | | |
| | Best case time - k i.e. $B(n) = O(1)$; $\Omega(1)$; $\Theta(1)$ | | |
| | Worst case - searching for a leaf element - depends upon the height of the tree | | |
| | Worst case time - $\log n$ i.e. $O(\log n)$ | | |
| | min $w(n) = \log n$; max $w(n) = n$ | | |
| | | | |
| 17 | Disjoint sets | | |
| | No common numbers between two sets - intersection is zero | | |
| | Operations - find, union | | |
| | Find - search or check membership | | |
| | Union - Add an edge | | |
| | <i>Kruskal algorithm: If you take an edge and both the vertices belong to the same set, then there is a cycle in the graph</i> | | |
| | Weighted union is used while adding edges and detecting cycle | | |

| | | | |
|-----------|--|--|--|
| | Collapsing find - process of directly linking node to a direct parent of a set is called collapsing find - reduces the time to find | | |
| | | | |
| 18 | Divide and conquer - Strategy 1 | | |
| | Strategy - an approach for solving a problem | | |
| | If a problem cannot be solved, divide it into sub-problems and find a solution for each sub problem, combine the solutions. <i>One point to note is that each sub problem should be similar to the original problem only.</i> | | |
| | Recursive in nature | | |
| | Should have one method to combine the solutions of each sub problem | | |
| | | | |
| 19 | Problems under Divide and Conquer | | |
| | Binary search | | |
| | Finding maximum and minimum | | |
| | MergeSort | | |
| | QuickSort | | |
| | Strassen's matrix multiplication | | |
| | | | |
| 20 | Recurrence relation 1: $T(n) = T(n-1) + 1$ | | |
| | void test(int n) | | |
| | { | | |
| | if(n > 0){ | | |
| | printf("%d",n); | | |
| | test(n-1) | | |
| | } | | |
| | } | | |
| | | | |
| | test(3) | | |
| | 3. test(2) | | |

| | | | |
|----|---|----------|---|
| | 2. test(1) | | |
| | 1 test(0) | | |
| | each print statement takes constant time 1 and there are n+ 1 calls made to the function. we can ignore the last call when it is not printing | | |
| | $f(n) = n + 1$ calls ; $O(n)$ | | |
| | $T(n) = T(n-1) + 1$; if we ignore if condition | | |
| | Let us solve this relation; | | |
| | if we know $T(n-1)$, we can get $T(n)$ | | |
| | $T(n-1) = T(n-2) + 1$ | | |
| | $T(n) = [T(n-2) + 1] + 1$ | | |
| | $T(n) = T(n-3) + 3$ | | |
| |continue for k times | | |
| | $T(n) = T(n-k) + k$ | | |
| | We would stop after k substitutions; now we need to find k | | |
| | Assume $n - k = 0$; therefore $n = k$ | | |
| | $T(n) = T(n-n) + n$ | | |
| | $T(n) = T(0) + n$ | | |
| | $T(n) = n + 1$ i.e. $\theta(n)$ | | |
| | | | |
| 21 | Recurrence relation 2: $T(n) = T(n-1) + n$ (decreasing function) | | |
| | void test(int n) | $T(n)$ | |
| | { | | |
| | if(n > 0) | | 1 |
| | { | | |
| | for(i = 0; i < n; i++) | $n+1$ | |
| | { | | |
| | printf("%d", n); | n | |
| | } | | |
| | test(n-1); | $T(n-1)$ | |

| | | | |
|-----------|---|--|--|
| | } | | |
| | } | | |
| | | $T(n) = T(n-1) + 2n + 2$ i.e. $\theta(n)$ | |
| | we can also write $T(n) = T(n-1) + n$ for $n > 0$ | | |
| | $T(n) = 1$ for $n = 0$ | | |
| | $T(n)$ | n time | |
| | n $T(n-1)$ | n-1 time | |
| | n-1. $T(n-2)$ | n - 2 time | |
| | n-2 $T(n-3)$ | n - 3 time | |
| | | | |
| | $T(2)$ | | |
| | 2 $T(1)$ | 2 units of time | |
| | 1 $T(0)$ | 1 unit of time | |
| | for $T(0)$ it does nothing | 0 unit of time | |
| | time taken - | | |
| | | $0 + 1 + 2 + \dots + n-1 + n$ | |
| | $\theta(n^2)$ | $T(n) = n(n+1)/2$ | |
| | | | |
| | $T(n) = T(n-1) + n$ | | |
| | $T(n-1) = T(n-2) + n-1$ | | |
| | thus, $T(n) = T(n-2) + (n-1) + n$ | **remember, don't add the terms | |
| | $T(n) = T(n-3) + (n-2) + (n-1) + n$ | | |
| | $T(n) = T(n-k) + (n-(k-1)) + (n-(k-2)) \dots + (n-1) + n$ | if we continue for k times | |
| | assume $n - k = 0$; $n = k$ | | |
| | Thus, $T(n) = T(n-n) + (n - n + 1) + (n - n + 2) \dots + (n-1) + n$ | | |
| | $T(n) = T(0) + n(n+1)/2$ | | |
| | $T(n) = 1 + n(n+1)/2$ | $\theta(n^2)$; this extra 1 is owing to the calls | |
| | | | |
| | | | |
| 22 | Recurrence relation 3: $T(n) = T(n-1) + \log n$ | | |
| | void test(int n) | $T(n)$ | |

| | | |
|---|---|--|
| { | | |
| if(n>0) | | |
| { | | |
| for(i = 1; i <n; i = i*2) | | |
| { | | |
| printf("%d", i); | log n times | |
| } | | |
| test(n-1); | T(n-1) | |
| } | | |
| } | | |
| $T(n) = T(n-1) + \log n$ for $n > 0$ | | |
| $T(n) = 1$ for $n = 0$ | | |
| | | |
| Solve using tree method, | | |
| T(n) | | |
| logn T(n-1) | | |
| log(n-1) T(n-2) | | |
| log(n-2) T(n-3) | | |
| | | |
| log2 T(!) | | |
| log 1 T(0) | | |
| | | |
| logn + log(n-1) ++ log2 + log1 | | |
| $\log[n(n-1)(n-2)....2.1] = \log(n!)$ | there is no tight bound for this function but there is an upper bound for it | |
| $O(n \log n)$ | | |
| | | |
| Solving using induction method. | | |
| $T(n) = T(n-1) + \log n$ | | |
| $T(n) = T(n-2) + \log(n-1) + \log(n)$ | | |
| $T(n) = T(n-3) + \log(n-2) + \log(n-1) + \log n$ | | |

| | | | |
|-----------|--|----------------------|---|
| | | | |
| | $T(n) = T(n-k) + \log n + \log(n-1) + \dots \log 1$ | | |
| | Asume $n-k = 0$ | | |
| | $T(n) = T(0) + \log n!$ | | |
| | $T(n) = 1 + \log n!$ | | |
| | $O(n \log n)$ | | |
| | | | |
| 23 | How to get the direct answer for a recurrence relation? | | |
| | $T(n) = T(n-1) + 1$ | $O(n)$ | |
| | $T(n) = T(n-1) + n$ | $O(n^2)$ | |
| | $T(n) = T(n-1) + \log n$ | $O(n \log n)$ | |
| | $T(n) = T(n-1) + n^2$ | $O(n^3)$ | |
| | $T(n) = T(n-2) + 1$ | $O(n/2) \sim O(n)$ | |
| | $T(n) = T(n-100) + n$ | $O(n^2)$ | |
| | $T(n) = 2T(n-1) + 1$ | ??? | |
| | | | |
| 24 | Recurrence relation 4: $T(n) = 2T(n-1) + 1$ | | |
| | Test(int n) | $T(n)$ | |
| | { | | |
| | if(n > 0) | | 1 |
| | { | | |
| | printf("%d", n); | | 1 |
| | test(n-1); | $T(n-1)$ | |
| | test(n-1); | $T(n-1)$ | |
| | } | | |
| | } | | |
| | | $T(n) = 2T(n-1) + 1$ | |
| | $T(n) = 2T(n-1) + 1$ for $n > 0$ | | |
| | $T(n) = 1$ for $n = 0$ | | |
| | | | |

| | | | |
|-----------|--|---|-------|
| | Solve using recursion tree method | | |
| | 1 $T(n-1)$ $T(n-1)$ | | 2 |
| | 1 $T(n-2)$ $T(n-2)$ | 1 $T(n-2)$ $T(n-2)$ | 4 |
| | 1 $T(n-3)$ $T(n-3)$ 1 $T(n-3)$ $T(n-3)$ | 1 $T(n-3)$ $T(n-3)$ 1 $T(n-3)$ $T(n-3)$ | 8 |
| | | | |
| | $T(0)$. $T(0)$ | | 2^k |
| | | | |
| | $1 + 2 + 2^2 + \dots + 2^k = 2^{(k+1)} - 1$ | | |
| | as, $a + ar + ar^2 + \dots + ar^k = a(r^{(k+1)} - 1)/(r - 1)$ | | |
| | | | |
| | Assume $n - k = 0$ | | |
| | thus, $2^{(n+1)} - 1$ | $O(2^n)$ | |
| | | | |
| | Back substitution method | | |
| | $T(n) = 2T(n-1) + 1$ | | |
| | $T(n) = 4T(n-2) + 2 + 1$ | | |
| | $T(n) = 8T(n-3) + 4 + 2 + 1$ | | |
| | | | |
| | $T(n) = 2^k T(n - k) + 2^{(k-1)} + 2^{(k)} + \dots + 2^3 + 2^2 + 1$ | | |
| | Assume $n - k = 0$ | | |
| | $n = k$ | | |
| | $T(n) = 2^n T(0) + 1 + 2 + 2^2 + \dots + 2^{n-1}$ | | |
| | $T(n) = 2^n + 2^n - 1$ i.e. $2^{(n+1)} - 1$ | | |
| | | | |
| 25 | Master theorem for decreasing function | | |
| | $T(n) = T(n-1) + 1$ | $O(n)$ | |
| | $T(n) = T(n-1) + n$ | $O(n^2)$ | |
| | $T(n) = T(n-1) + \log n$ | $O(n \log n)$ | |
| | $T(n) = 2T(n-1) + 1$ | $O(2^n)$ | |
| | $T(n) = 3T(n-1) + 1$ | $O(3^n)$ | |

| | | | |
|-----------|--|--------------|---|
| | $T(n) = 2T(n-1) + n$ | $O(2^{2n})$ | |
| | $T(n) = 2T(n-2) + 1$ | $O(2^{n/2})$ | |
| | | | |
| | $T(n) = aT(n-b) + f(n)$ | | |
| | $a > 0, b > 0$ and $f(n) = O(n^k)$ where $k \geq 0$ | | |
| | if $a = 1, O(n^{k+1})$ or $O(n \cdot f(n))$ | | |
| | | | |
| | if $a > 1, O(n^k \cdot a^{n/b})$ | | |
| | if $a < 1, O(n^k)$ or $O(f(n))$ | | |
| | | | |
| 26 | Dividing functions | | |
| | test(int n) | $T(n)$ | |
| | { | | |
| | if(n > 1) | | |
| | { | | |
| | printf("%d", n); | | 1 |
| | test(n/2) | $T(n/2)$ | |
| | } | | |
| | } | | |
| | | | |
| | $T(n) = T(n/2) + 1$ for $n > 1$ | | |
| | $T(n) = 1$ for $n = 1$ | | |
| | | | |
| | $T(n)$ | | |
| | 1 $T(n/2)$ | | |
| | 1 $T(n/2^2)$ | | |
| | 1 $T(n/2^3)$ | | |
| |continue for k times | | |
| | 1 $T(n/2^k)$ | | |
| | | | |

| | | | |
|-----------|---|-------------------------------|--|
| | assume , $n/2^k = 1$ | | |
| | thus, we have taken k steps overall | | |
| | since, $n/2^k = 1 \Rightarrow k = \log n$ with base 2 | $O(\log n)$ | |
| | | | |
| | Solving by substitution method | | |
| | $T(n) = T(n/2) + 1$ | | |
| | $T(n) = T(n/2^2) + 2$ | | |
| | $T(n) = T(n/2^3) + 3$ | | |
| | | | |
| | $T(n) = T(n/2^k) + k$ | | |
| | assume $n/2^k = 1$ | | |
| | thus, $k = \log n$ with base 2 | | |
| | $T(n) = T(1) + \log n$ | | |
| | $O(\log n)$ | | |
| | | | |
| 27 | Recurrence relation: $T(n) = T(n/2) + n$ | | |
| | $T(n) = T(n/2) + n$ for $n > 1$ | | |
| | $T(n) = 1$ for $n=1$ | | |
| | | | |
| | $T(n)$ | | |
| | $T(n/2) \quad n$ | | |
| | $T(n/2^2) \quad n/2$ | | |
| | $T(n/2^3) \quad n/2^2$ | | |
| | | | |
| | $T(n/2^k). \quad n/2^{(k-1)}$ | | |
| | | | |
| | $T(n) = n + n/2 + n/2^2 + n/2^3..... + n/2^k$ | | |
| | $T(n) = n[1 + 1/2 + 1/2^2 + 1/2^3 +.....1/2^k]$ | | |
| | $T(n) = n \cdot 1 = n$ | | |
| | $O(n)$ | | |

| | | | |
|--|--|----------|--|
| | | | |
| | Using substitution method | | |
| | $T(n) = T(n/2) + n$ | | |
| | | | |
| | $T(n) = T(n/2^2) + n/2 + n$ | | |
| | | | |
| | $T(n) = T(n/2^3) + n/2^2 + n/2 + n$ | | |
| | | | |
| | $T(n) = T(n/2^k) + n/2^{k-1} + \dots + n/2^2 + n/2 + n$ | | |
| | Assume $n/2^k = 1$ | | |
| | $k = \log n$ with base 2 | | |
| | $T(n) = T(1) + n[1/2^{k-1} + \dots + 1/2^2 + \dots + 1]$ | | |
| | $T(n) = 1 + 2n \sim O(n)$ | | |
| | | | |
| | 28 Recurrence Relation: $T(n) = 2T(n/2) + n$ | | |
| | void test(int n) | $T(n)$ | |
| | { | | |
| | if(n > 1) | | |
| | { | | |
| | for(int i = 0; i < n ; i++) | | |
| | { | | |
| | stmt | n | |
| | } | | |
| | test(n/2); | $T(n/2)$ | |
| | test(n/2); | $T(n/2)$ | |
| | | | |
| | $T(n) = 2T(n/2) + n$ for $n > 1$ | | |
| | $T(n) = 1$ for $n = 1$ | | |
| | | | |
| | Solve using recursion tree method, | | |

| | | | |
|--|---|-----------------|--|
| | $T(n)$ | | |
| | $T(n/2). \quad T(n/2) \quad n$ | n | |
| | $T(n/2^2) \quad T(n/2^2) \quad T(n/2^2) \quad T(n/2^2) \quad n/2$ | n | |
| | $T(n/2^3). \quad T(n/2^3). \quad T(n/2^3). \quad T(n/2^3). \quad T(n/2^3). \quad T(n/2^3). \quad T(n/2^3). \quad T(n/2^3).$ | n | |
| | | n | |
| | $T(n/2^k).....$ | | |
| | | n | |
| | assume $n / 2^k = 1$ | | |
| | $k = \log n$ with base 2 | | |
| | $T(n) = nk \sim O(n \log n)$ | | |
| | | | |
| | Using backsubstitution method; | | |
| | $T(n) = 2T(n/2) + n$ | | |
| | $T(n/2) = 2T(n/2^2) + n/2$ | | |
| | $T(n) = 2[2T(n/2^2) + n/2] + n$ | | |
| | $T(n) = 2^2T(n/2^2) + n + n$ | | |
| | $T(n) = 2^3T(n/2^3) + 3n$ | | |
| |continue for k times | | |
| | $T(n) = 2^kT(n/2^k) + kn$ | | |
| | Asume $T(n/2^k) = T(1)$ | | |
| | $k = \log n$ with base 2 | | |
| | Thus, $T(n) = n + n \log n \sim O(n \log n)$ | | |
| | | | |
| | 29 Masters Theorem for dividing functions | | |
| | $T(n) = aT(n/b) + f(n)$ | $\log a$ with b | |
| | $a \geq 1; b > 1; f(n) = \theta(n^k \log^p n)$ | k | |
| | | | |
| | case 1: if $\log a$ with base b $> k$ then $\theta(n^{\log a \text{ with base b}})$ | | |
| | case 2: if $\log a$ with base b $= k$ then | | |

| | | | |
|--|---|--|--|
| | if $p > -1$ $\theta(n^k \log^{(p+1)} n)$ | | |
| | if $p = -1$ $\theta(n^k \log \log n)$ | | |
| | if $p < -1$ then $\theta(n^k)$ | | |
| | case 3: if \log_a with base $b < k$ | | |
| | then, if $p \geq 0$, $\theta(n^k \log^p n)$ | | |
| | if $p < 0$, $\theta(n^k)$ | | |
| | | | |
| | $T(n) = 2T(n/2) + 1$ | | |
| | $a = 2$ | | |
| | $b = 2$ | | |
| | $f(n) = \theta(n^0) * \log n^0$ | | |
| | $k = 0$; $p = 0$ | | |
| | here, \log_a with base $b > k$ | | |
| | $\theta(n^1)$ where \log_a with base b is 1 | | |
| | | | |
| | $T(n) = 4T(n/2) + n$ | | |
| | \log_a with base $b = 2$ | | |
| | $k = 1$ | | |
| | $p = 0$ | | |
| | this is an example of case 1 | | |
| | $\theta(n^2)$ | | |
| | | | |
| | $T(n) = 8T(n/2) + n$ | | |
| | \log_8 with base $2 = 3 > k = 1$ | | |
| | $\theta(n^3)$ | | |
| | | | |
| | $T(n) = 9T(n/3) + 1$ | | |
| | \log_a with base $b = 2 > k$ | | |
| | $\theta(n^2)$ | | |
| | | | |

| | | | |
|--|--------------------------------------|--------|--|
| | $T(n) = 9T(n/3) + n^2$ | | |
| | loga with base $b = 2 = k$ | case 2 | |
| | $\theta(n^2)$ | | |
| | | | |
| | $T(n) = 8T(n/2) + n$ | | |
| | $\theta(n^3)$ | | |
| | | | |
| | $T(n) = 2T(n/2) + n$ | | |
| | loga with base $b = k = 1$; $p = 0$ | | |
| | case 2 | | |
| | $\theta(n \log n)$ | | |
| | | | |
| | $T(n) = 4T(n/2) + n^2$ | | |
| | $\theta(n^2 \log n)$ | | |
| | | | |
| | $T(n) = 4T(n/2) + n^2 \log n$ | | |
| | $\theta(n^2 \log n^2)$ | | |
| | | | |
| | $T(n) = 8T(n/2) + n^3$ | | |
| | $\theta(n^3 \log n)$ | | |
| | | | |
| | $T(n) = 2T(n/2) + n/\log n$ | | |
| | log a with base $b = k = 1$ | | |
| | $p = -1$ | | |
| | $\theta(n \log \log n)$ | | |
| | | | |
| | $T(n) = 2T(n/2) + n/\log n^2$ | | |
| | $p = -2$ | | |
| | $\theta(n)$ | | |
| | | | |

| | | | |
|-----------|------------------------|--|--|
| | $T(n) = 2T(n/2) + n^2$ | | |
| | loga with base $b < k$ | | |
| | $\theta(n^2)$ | | |
| | | | |
| | $T(n) = 2T(n/2) + n^2$ | | |
| | $\theta(n^2 \log n)$ | | |
| | | | |
| | $T(n) = 2T(n/2) + n^3$ | | |
| | loga with base $b < k$ | | |
| | $\theta(n^3)$ | | |
| | | | |
| 30 | $T(n) = 2T(n/2) + 1$ | | |
| | loga with base $b = 1$ | | |
| | $k = 0$ | | |
| | loga with base $b > k$ | | |
| | $\theta(n^1)$ | | |
| | | | |
| | $T(n) = 4T(n/2) + 1$ | | |
| | loga with base $b = 2$ | | |
| | $k = 0$ | | |
| | $\theta(n^2)$ | | |
| | | | |
| | $T(n) = 4T(n/2) + n$ | | |
| | loga with base $b = 2$ | | |
| | $k = 1$ | | |
| | $\theta(n^2)$ | | |
| | | | |
| | $T(n) = 8T(n/2) + n^2$ | | |
| | loga with base $b = 3$ | | |
| | $k = 2$ | | |

| | | | |
|--|---------------------------------|--|--|
| | $\theta(n^3)$ | | |
| | | | |
| | $T(n) = 16T(n/2) + n^2$ | | |
| | log a with base b = 4 | | |
| | k = 2 | | |
| | $\theta(n^4)$ | | |
| | | | |
| | $T(n) = T(n/2) + n$ | | |
| | log a with base b = 0 | | |
| | k = 1 | | |
| | $\theta(n)$ | | |
| | | | |
| | $T(n) = 2T(n/2) + n^2$ | | |
| | log a with base b = 1 | | |
| | k = 2 | | |
| | $\theta(n^2)$ | | |
| | | | |
| | $T(n) = 2T(n/2) + n^2 \log n$ | | |
| | log a with base b = 1 | | |
| | k = 2 | | |
| | $\theta(n^2 \log n)$ | | |
| | | | |
| | $T(n) = 4T(n/2) + n^3 \log^2 n$ | | |
| | log a with base b = 2 | | |
| | k = 3 | | |
| | $\theta(n^3 \log^2 n)$ | | |
| | | | |
| | $T(n) = 2T(n/2) + n^2 / \log n$ | | |
| | log a with base b = 1 | | |
| | k = 2 | | |

| | | | |
|--|---------------------------------|--|--|
| | $\theta(n^2)$ | | |
| | | | |
| | $T(n) = T(n/2) + 1$ | | |
| | log a with base b = 0 | | |
| | k = 0 | | |
| | $\theta(\log n)$ | | |
| | | | |
| | $T(n) = 2T(n/2) + n$ | | |
| | log a with base b = 1 | | |
| | k = 1 | | |
| | p = 0 | | |
| | $\theta(n \log n)$ | | |
| | | | |
| | $T(n) = 2T(n/2) + n \log n$ | | |
| | log a with base b = 1 | | |
| | k = 1 | | |
| | p = 1 | | |
| | $\theta(n \log^2 n)$ | | |
| | | | |
| | $T(n) = 4T(n/2) + n^2$ | | |
| | log a with base b = 2 | | |
| | k = 2; p = 0 | | |
| | $\theta(n^2 \log n)$ | | |
| | | | |
| | $T(n) = 4T(n/2) + (n \log n)^2$ | | |
| | log a with base b = 2 | | |
| | k = 2, p = 2 | | |
| | $\theta(n^2 \log^3 n)$ | | |
| | | | |
| | $T(n) = 2T(n/2) + n/\log n$ | | |

| | | | |
|-----------|---|--|--|
| | log a with base b = 1 | | |
| | k = 1; p = -1 | | |
| | theta(nloglogn) | | |
| | | | |
| | $T(n) = 2T(n/2) + n/\log^2 n$ | | |
| | log a with base b = 1 | | |
| | k = 1; p = -2 | | |
| | theta(n) | | |
| | | | |
| 31 | Root function Recurrence relation | | |
| | $T(n) = T(\sqrt{n}) + 1$ for $n > 2$ | | |
| | $T(n) = 1$ for $n = 2$ | | |
| | | | |
| | $T(n) = T(\sqrt{n}) + 1$ | | |
| | $T(n) = T(n^{1/2}) + 1$equation 1 | | |
| | using substitution | | |
| | $T(n) = T(n^{1/2^2}) + 2$equation 2 | | |
| | $T(n) = T(n^{1/2^3}) + 3$equation 3 | | |
| | $T(n) = T(n^{1/2^k}) + k$equation 4 | | |
| | | | |
| | assume, $n = 2^m$ | | |
| | $T(2^m) = T(2^{m/2^k}) + k$ | | |
| | assume $T(2^{m/2^k}) = T(2)$ | | |
| | thus, $m/2^k = 1$ | | |
| | $m = 2^k$ | | |
| | $k = \log m$ with base 2 | | |
| | substituting value of n | | |
| | $m = \log n$ with base 2 | | |
| | therefore, $k = \log \log n$ with base 2 | | |
| | theta(loglogn with base 2) | | |

| | | | |
|-----------|---|---|--|
| | | | |
| 32 | Binary Search Iterative Method | | |
| | To perform binary search, the prerequisite is that the list must be in sorted order | A = {3, 6, 8, 12, 14, 17, 25, 29, 31, 36, 42, 47, 53, 55, 62} | |
| | we need two index pointers, one is low at the starting point and the other is high at the end point | l = 1, h = 15 (lowest and highest index); mid = 8 | |
| | mid = low + high / 2 and we take the floor value | key value = 42; A[mid] = 29 --> key > A[mid] | |
| | the key value is on the right hand side as key value is greater than A[mid] | | |
| | we will change low to mid + 1 | l = 9, h = 15; mid = 9 + 15 / 2 = 12 | |
| | | A[mid] = 47 > key | |
| | we will change high to mid - 1 as key < A[mid] | | |
| | | h = 11, l = 9, mid = 10; A[mid] = 36 | |
| | | A[mid] < key | |
| | we will change low to mid + 1 | l = 11; h = 11; mid = 11; A[mid] = 42 | |
| | we can return the index as we have found the key value | A[mid] = key | |
| | therefore, binary search looks faster than linear search. It just took 4 comparisons | | |
| | | | |
| | int BinSearch(A, n, key) | | |
| | { | | |
| | l = 1, h = n | | |
| | mid = l + h / 2 - take floor value | | |
| | while(l <= h){ | | |
| | if(key == A[mid]) | | |
| | { return index i.e.element is found} | | |
| | else if(key < A[mid]) | | |
| | {h= mid-1;} | | |
| | else { | | |
| | l = mid + 1;} | | |
| | } | | |

| | | | |
|-----------|---|----------|---|
| | return 0; | | |
| | } | | |
| | | | |
| | Time taken for binary search = $\log n$ | | |
| | min time: $O(1)$ | | |
| | max time: $O(\log n)$ | | |
| | avg time = add time for each element and divide by number of elements | | |
| | | | |
| 33 | Binarysearch Recursive method | | |
| | Algorithm RBinarySearch(l,h,key) | $T(n)$ | |
| | { | | |
| | if(l==h) | | 1 |
| | { | | |
| | if(A[l]== key) | | |
| | { | | |
| | return l; | | |
| | } | | |
| | else | | |
| | { | | |
| | return 0; | | |
| | } | | |
| | else | | |
| | { | | |
| | mid = l + h / 2 //taking floor value | | 1 |
| | if(key == A[mid]) | | 1 |
| | {return mid;} | | |
| | if(key < A[mid]) | | 1 |
| | { | | |
| | return RBinarySearch(l, mid - 1, key) | $T(n/2)$ | |
| | } | | |

| | | | |
|----------|---|---------------------------------|--|
| | else | | |
| | { | | |
| | return RBinarySearch(mid+1, h, key) | $T(n/2)$ | |
| | } | | |
| | } | | |
| | | $T(n) = 1; n = 1$ | |
| | | $T(n) = T(n/2) + 1$ for $n > 1$ | |
| | | | |
| | | $\theta(\log n)$ | |
| | | | |
| | 34 Heaps | | |
| a | Representation of a binary tree using an array | | |
| | T {A, B, C, D, E, F, G} | | |
| | if a node is at index i; | | |
| | its left child is at node $2*i$ | | |
| | its right child is at node $2*i + 1$ | | |
| | its parent is at node $i/2$ | | |
| | if there are missing nodes, we leave a blank in its place in the array | | |
| | | | |
| b | Full binary tree | | |
| | In its height, it has maximum number of nodes and if we wish to add a node, height would increase | | |
| | Max no. of nodes = $2^h - 1$ | | |
| c | Complete binary tree | | |
| | there is no missing element from first element to the last element in array representation of the binary tree | | |
| | Every full binary tree is also a complete binary tree | | |
| | A complete binary tree is a full binary tree until height $h - 1$ | | |
| | Height of a complete binary tree would be minimum i.e. $\log n$ | | |
| d | Heap | | |

| | | | |
|-----------|---|--|--|
| | Heap is a complete binary tree | | |
| | Max Heap: every node has value greater than all its descendants {50, 30, 20, 15, 10, 8, 16} | | |
| | Min Heap: every node has value smaller or equal to than all its descendants {10, 30, 20, 35, 40, 32, 25} | | |
| | | | |
| 35 | Insert operation in a max heap | | |
| | Insert 60 in the above given max heap | | |
| | this value should be inserted in the last free space in the array | | |
| | i.e. left child of the left most leaf node | | |
| | Then, adjust the elements to make it as a heap | | |
| | So, compare and move 60 up the levels and in the array check at $i/2$ indices where initially i would be the last empty index where 60 was inserted | | |
| | Time taken would be equal to the number of swaps | | |
| | this depends upon the height of the tree i.e. $\log n$, hence $O(\log n)$ | | |
| | minimum time is of no swaps $O(1)$; max would be $O(\log n)$ | | |
| | | | |
| 36 | Delete operation in a max heap | | |
| | From the heap, we need to remove the root / top most element only | | |
| | The last element in the complete binary tree would come in its place | | |
| | Adjust the elements to maintain heap order | | |
| | From the root towards the leaf, adjust | | |
| | Compare the children ($2i$ and $2i + 1$) and whichever child is greater than compare with the parent | | |
| | Time taken depends upon the height; max could be $O(\log n)$ | | |
| | Whenever you delete from max heap, you get the next max element and in case of min heap, it would be the next min element | | |
| | | | |

| | | |
|---|-----------------------------|--|
| 37 HeapSort | | |
| For a given set of numbers, create a heap | | |
| Delete all the elements from the heap | | |
| Total N elements we have inserted; each element we assume is moved up to the root; so time taken $O(N\log N)$ | | |
| Then we delete the elements | | |
| Store deleted elements in the array in free space in the end | | |
| Deletion also takes $O(N\log N)$ time | | |
| Thus, heapsort takes $O(N\log N)$ | | |
| | | |
| 38 Heapify | | |
| The process of creating heap but direction is opposite than creating a heap | | |
| $O(N)$ | | |
| | | |
| 39 Priority Queue | | |
| elements will have priority and they would be inserted and deleted as per the priority order | | |
| For min heap, smaller the no. higher the priority | | |
| For max heap, greater the no. higher the priority | | |
| $O(\log N)$ for insertion and/or deletion | | |
| | | |
| 40 TwoWay MergeSort - Iterative method | Algorithm Merge(A, B, m, n) | |
| merging two sorted lists to get a sorted result | {i = 1, j = 1, k = 1; | |
| A = {2, 8, 15, 18} i | while(i <= m && j <= n){ | |
| B = { 5, 9, 12, 17} j | if(A[i] < B[j]) | |
| Compare A(i) with B(j) to get C(k) and move to next location | { | |
| m + n elements are obtained , thus theta(m + n) | C[k++] = A[i++]; | |
| | } | |
| | else { | |
| | C[k++] = B[j++]; | |

| | | | |
|-----------------|---|--------------------------------|--|
| | | } | |
| | | for(; i <=m; i++){ | |
| | | C[k++]=A[i]; | |
| | | } | |
| | | for(;j<= n;j++){ | |
| | | C[k++] = B[j]; | |
| | | } | |
| | | } | |
| | | | |
| 41 | Merging more than two lists | | |
| | M-way merging | | |
| | A = {4, 6, 12} | | |
| | B = {3, 5, 9} | | |
| | C = {8, 10, 16} | | |
| | D = {2, 4, 18} | | |
| | One way is that we merge A and B; C and D and then finally merge the two resulting lists --> so we perform merge three times here | | |
| | Another way is that we first merge A and B; then we merge resulting list with C ; and the resulting list with D | | |
| | Two-way mergesort is an iterative process whereas mergeSort is a recursive process | | |
| | | | |
| | A = {9, 3, 7, 5, 6, 4, 8, 2} - given an array and we have to sort them using 2-way mergesort | | |
| 1st pass | We would consider each element as a sorted list and merge | merged n elements in this pass | |
| | First select two lists 3 and 9; then merge them - 3, 9 | | |
| | Similarly, we select two lists 7 and 5 , merge them - 5 and 7 | | |
| | Another lists we get are {4, 6} and {2, 8} | | |
| | Now, we have 4 lists with two elements each | | |
| 2nd pass | When we merged we kept the resulting 4 lists in another array B; B = {{3, 9}, {5, 7}, {4, 6}, {2, 8}} | merged n elements in this pass | |

| | | | |
|-----------------------------|--|--|---|
| | We merge two lists each | | |
| 3rd pass | $C = \{\{3, 5, 7, 9\}, \{2, 4, 6, 8\}\}$ | merged n elements in this pass | |
| | we merge the above two lists to get a single sorted list | | |
| | $D = \{2, 3, 4, 5, 6, 7, 8, 9\}$ | | |
| | | | |
| | $\log(\text{no of elements}) = \text{no. of passes}$ | | |
| | Time complexity: $O(n \log n)$ | | |
| | | | |
| 42 MergeSort | | | |
| | $A = \{9, 3, 7, 5, 6, 4, 8, 2\}$ | Algorithm MergeSort(l, h){ | $T(n)$ |
| | If there is a single element, we can consider it as a base or small problem {Divide and conquer} | | |
| | | if(l < h){ | |
| | | mid = (l + h) / 2; | 1 |
| | | MergeSort(l, mid); | $T(n/2)$ |
| | | MergeSort(mid + 1, h); | $T(n/2)$ |
| | | Merge(l, mid, h); | n |
| | | } | $T(n) = 2T(n/2) + n$ for $n > 1$ |
| | | } | $T(n) = 1$ for $n = 1$ |
| | time complexity: $\theta(n \log n)$ | | using master's theorem, $a = 2, b = 2, k = 1$ |
| | merging is done in post order traversal | | \log_a with base $b = 1 = k$ |
| | | | thus, it is case 2 |
| | | | $\theta(n \log n)$ |
| 43 Pros of MergeSort | | Cons of MergeSort | |
| | works great for Large size lists | Extra space (not inplace sort) | |
| | suitable for Linked List | no small problem | |
| | supports external sorting | recursive and uses a stack (need $n + \log n$ space) i.e. space complexity: $O(n + \log n)$ where n is the extra space and $\log n$ is the stack space | |
| | stable: the order of duplicates is maintained | | |
| | | insertion sort ($O(n^2)$) | |

| | | | |
|---------------------|---|---|--|
| | | mergesort $O(n \log n)$ | |
| | | for small problems, $n \leq 15$; insertionsort works better --> use insertion sort | |
| | | | |
| 43 QuickSort | | | |
| | students arranging themselves in increasing order of heights | | |
| | 10 80 90 60 30 20 | | |
| | 5 6 3 4 2 1 9 | | |
| | 4 6 7 10 16 12 13 14 | | |
| | | | |
| | A = {10, 16, 8, 12, 15, 6, 3, 9, 5, INFINITY} | partition(l, h){ | |
| | select first element as a pivot | pivot = A[l]; | |
| | pivot = 10 | i = l; j = h; | |
| | we need to find the sorted position for 10 | while(i < j){do | |
| | i starting from pivot and j starting from infinity | { | |
| | i would check for elements greater than 10; j would check for elements smaller than pivot | i++; | |
| | we are using the partitioning procedure | } while(A[i] <= pivot); | |
| | increment i until next value is greater than 10 and decrement j until next value is smaller than pivot; stop and swap | do | |
| | {10, 5, 8, 9, 3, 6, 15, 12, 16} | { | |
| | send pivot element at j position | j--; | |
| | now, we can sort the two lists around the partitioning position by performing quicksort recursively | }while(A[j] > pivot); | |
| | | if(i < j){ | |
| | QuickSort(l, h) | swap(A[i], A[j]); | |
| | { | } | |
| | if(l < h) | swap(A[l], A[j]); | |
| | { | return j; | |
| | j = partition(l, h); | } | |
| | QuickSort(l, j); | | |
| | QuickSort(j+1, h); | | |

| | | | |
|--|---|--|--|
| | } | | |
| | } | | |
| | | | |
| 44 QuickSort Analysis | | | |
| suppose it is partitioning in the middle of 1 and 15th index | | | |
| then, two partitions: [1, 7] ; [9, 15] | | | |
| further partitions: [1, 3]; [5, 7]; [9, 11]; [13, 15] | | | |
| at each level , n elements are being handled | | | |
| and there are logn levels | | | |
| thus time complexity for best case: $O(n \log n)$ | | | |
| median : middle element of a sorted list | | | |
| best case of quicksort is that the partitioning occurs exactly at the middle | | | |
| | | | |
| worstcase: if we have an already sorted list | | | |
| time complexity for worstcase: $O(n^2)$ | | | |
| to handle this, try taking middle element as a pivot | | | |
| 2. select random element as a pivot | | | |
| | | | |
| 45 Strassen's matrix multiplication | | | |
| A = [a11 a12 | | | |
| a21. a22] | | | |
| B = [b11 b12 | | | |
| b21 b22] | | | |
| Cij = Summing up Aik*Bkj | | | |
| | | | |
| for(i = 0; i < n ; i++){ | | | |
| for(j = 0; i < n ; j++){ | | | |
| C[i,j]= 0; | | | |
| for(k=0;k<n;k++){ | | | |
| C[i,j] += A[i, k]*B[k, i]; | | | |

| | | | |
|--|--|--|--|
| | } | | |
| | } | | |
| | } | | |
| | $C_{11} = a_{11} \cdot b_{11} + a_{12} \cdot b_{21}$ | | |
| | $C_{21} = a_{11} \cdot b_{12} + a_{12} \cdot b_{22}$ | $A = [a_{11}]$ | |
| | $C_{21} = a_{21} \cdot b_{11} + a_{22} \cdot b_{21}$ | $B = [b_{11}]$ | |
| | $c_{22} = a_{21} \cdot b_{12} + a_{22} \cdot b_{22}$ | $C = [a_{11} \cdot b_{11}]$ | |
| | for $[2 \times 2]$ matrix, we would use above formula | for $[1 \times 1]$ matrix, use above formula | |
| | | | |
| | we assume that the matrix has dimensions of power of 2 | Algorithm MM(A, B, n) | |
| | | { | |
| | | if(n <= 2 | |
| | 8 times the function is calling itself | { | |
| | $T(n) = 8T(n/2) + n^2$ for $n > 1$ | C = 4 formula stated above; | |
| | $a = 8, b = 2, \log a$ with base $b = 3$ | } | |
| | $k = 2$ | else | |
| | it is case 1 of master's theorem | { | |
| | $\theta(n^3)$ | mid = $n/2$ | |
| | | MM(A ₁₁ , B ₁₁ , $n/2$) + MM(A ₁₂ , B ₂₁ , $n/2$); | |
| | | MM(A ₁₁ , B ₁₂ , $n/2$) + MM(A ₁₂ , B ₂₂ , $n/2$); | |
| | | MM(A ₂₁ , B ₁₁ , $n/2$) + MM(A ₂₂ , B ₂₁ , $n/2$); | |
| | | MM(A ₂₂ , B ₂₂ , $n/2$) + MM(A ₂₁ , B ₁₂ , $n/2$); | |
| | | } | |
| | | } | |
| | | | |
| | Strassen's approach - | | |
| | has given 4 different formulas with 7 multiplications | $P = (A_{11} + A_{22})(B_{11} + B_{22})$ | |
| | $C_{11} = A_{11} \cdot B_{11} + A_{12} \cdot B_{21}$ | $Q = (A_{21} + A_{22}) B_{11}$ | |
| | $C_{21} = A_{11} \cdot B_{12} + A_{12} \cdot B_{22}$ | $R = A_{11}(B_{12} - B_{22})$ | |
| | $C_{21} = A_{21} \cdot B_{11} + A_{22} \cdot B_{21}$ | $S = A_{22}(B_{21} - B_{11})$ | |

| | | | |
|-----------|--|---|--|
| | $C22 = A21 \cdot B12 + A22 \cdot B22$ | $T = (A11 + A12)B22$ | |
| | | $U = (A21 - A11)(B11 + B12)$ | |
| | | $V = (A12 - A22)(B21 + B22)$ | |
| | | | |
| | | $C11 = P + S - T + V$ | |
| | | $C12 = R + T$ | |
| | | $C13 = Q + S$ | |
| | | $C22 = P + R - Q + U$ | |
| | | | |
| | | $T(n) = 7T(n/2) + n^2$ for $n > 2$ | |
| | | $T(n) = 1$ for $n \leq 2$ | |
| | | using master's theorem, | |
| | | $O(n^{\log_7 \text{ with base } 2}) = O(n^{2.81})$ | |
| | | | |
| | Strategies used for solving optimization problems - Greedy Method, Dynamic programming, branch and bound | | |
| 46 | Greedy method | | |
| | Design which we can adopt to solve similar problems | | Greedy method says that each problem should be solved in stages - each stage we give an input, check if the solution is feasible then we pick it up and move to next stage |
| | Solving optimization problems | | Algorithm Greedy(a, n) $a = \{a1, a2, a3, a4, a5\}$; $n = 5$ |
| | Optimization problem: Problems which require either minimum or maximum result | | { |
| | Suppose we have a problem P where we need to travel from source A to destination B, we can have several solutions such as walking on foot, travel by an airplane, ride on a bike, travel by a bus, drive on a car, go by a train and so on..... Now, we notice that we also have some constraints. The solutions that satisfy the conditions given in a problem are called feasible solutions | Minimum cost journey - " Minimization problem "; then feasible solutions giving minimum cost are called optimal solutions . There can be many feasible solutions but only one optimal solution | for $i = 1$ to n do { $x = \text{select}(a)$: |
| | Example: selecting a car to purchase | Example: Hire a person for your company | if feasible(x) then |

| | | | |
|----------------------------|---|--|--------------------------|
| | Method 1: Looking at all the models available in the city | Method 2: Conduct an assessment center to filter people at each stage and get the best person | { |
| | Method 2: Checking for the features of the cars and filtering and selecting based on your preferences - greedy method | So, the person may not be the best but the approach is greedy here as we are using our criteria and constraints to choose the best person | solution = solution + x; |
| | | | |
| 47 Knapsack problem | | | } |
| | n = 7; m = 15 | Bag capacity is 15 kgs and we have been given 7 objects. we have to fill this bag with these objects. Profit is the gain we get by transferring this object. Problem is a container loading problem. Problem is filling the container with the objects as the capacity of container is limited | } |
| Objects | {0 1 2 3 4 5 6 7 } | Optimization and maximization problem | } |
| Profits | {P 10 5. 15 7 6 18 3 } | Constraints : Bag weight limit | |
| Weights | {W 2 3 5 7 1 4 1 } | | |
| Profit by weight | {P/W 5 1.3 3 1 6 4.5 3 } | | |
| | | | |
| 0 ≤ x ≤ 1 | Objects are divisible i.e. we can take just half kg of object 1 and may be 2 kgs of object 2 and so on | | |
| x | () | | |
| | x1 x2 x3 x4 x5 x6 x7 | | |
| Method 1 | Take the thing that have maximum profit | | |
| Method 2 | Take things with smaller weight so that you can put in more things | | |
| Method 3 | Take things that have highest profit by weight | | |
| | Let's use method 3 | | |

| | | | | | | | |
|--------------------|---|----|----|----|----|--|--|
| | First, I include object 5 that has maximum profit by weight. Then I check remaining weight I can put in. We can still put in 14 kgs. Then we select all the quantity of object 1. Remaining weight limit 12 kgs. Add all of object 6. Remaining weight limit 8 kgs. Add all of object 3. Remaining weight limit 3 kgs. Add all of object 7. Remaining weight limit is 2 kgs. Add 2/3 of object 2 as we have only 2 kgs limit remaining. | | | | | | |
| x | (1 2/3 1 0 1 1 1) | | | | | | |
| | Calculate total profit and verify weight | | | | | | |
| | Total weight = $1*2 + 2/3*3 + 1*5 + 0*7 + 1*1 + 1*4 + 1*1 = 15$ | | | | | //Multiplying x elements by Weight w for each object | |
| | Total profits = $1*10 + 2/3*5 + 1*15 + 1*6 + 1*18 + 1*3 = 54.6$ | | | | | //Multiplying x elements by Profit P for each object | |
| | | | | | | | |
| 48 | 0/1 Knapsack problem | | | | | | |
| | Objects are indivisible and fractions are not allowed i.e. either you include the whole thing or you do not include it at all | | | | | | |
| | | | | | | | |
| 49 | Job sequencing with deadlines | | | | | n = 5 (tasks) | |
| Jobs | J1 | J2 | J3 | J4 | J5 | | |
| Profits | 20 | 15 | 10 | 5 | 1 | | |
| Deadlines | 2 | 2 | 1 | 3 | 3 | | |
| | | | | | | | |
| | Assume that there is a machine, on which each job has to be processed and each job takes 1 unit of time (hour) for completion | | | | | | |
| | Set of the jobs which can be completed within their deadlines such that profit is maximized | | | | | Constraints: deadlines must be met | |
| | | | | | | | |
| deadlines | 0-----1-----2-----3 | | | | | maximum 3 slots / jobs | |
| time slots | 9am-----10am-----11am-----12am | | | | | | |
| Jobs chosen | J2 J1 J4 | | | | | | |
| Profits | $15 + 20 + 5 = 40$ | | | | | | |
| Sequence | J1 --> J2 ---> J4 J2 --> J1 --> J4 | | | | | | |

| | | | |
|---------------------|---|--|---------------|
| | | | |
| Job consider | Slot assign | Solution | Profit |
| J1 | [1,2] | J1 | 20 |
| J2 | [0,1][1,2] | J1J2 | 20 + 15 |
| J3 | [0,1][1,2] | J1J2 | 20 + 15 |
| J4 | [0,1][1,2][2,3] | J1J2J4 | 20 + 15 + 5 |
| J5 | [0,1][1,2][2,3] | J1J2J4 | 20 + 15 + 5 |
| | | | |
| 50 | Job sequencing with deadlines another example | n = 7 (jobs) | |
| Jobs | J1 J2 J3 J4 J5 J6 J7 | | |
| Profits | 35 30 25 20 15 12 5 | | |
| Deadlines | 3 4 4 2 3 1 2 | | |
| | | | |
| deadlines | 0.....1.....2.....3.....4 | 4 SLOTS AVAILABLE | |
| Jobs chosen | J4 J3 J1 J2 | | |
| Profits | 20 25 35 30 | 110 | |
| | | | |
| 51 | Optimal Merge Patern | | |
| | A = {3, 8, 12, 20} | | |
| | B = {5, 9, 11, 16} | | |
| | C = {3, 5, 8, 9, 11, 12, 16 20} | How merging works for two sorted lists ; time = $\theta(m + n)$ | |
| | what happens if we have 4 lists? | | |
| List | A B C D | | |
| Sizes | 6 5 2 3 | | |
| | | | |
| Choice 1 | We can merge at a time two lists - first A and B; the merge it with C and finally with D. total cost= 11 + 13 + 16 = 40 | | |

| | | | | | | | |
|---------------------------|---|----|----------|----|----|---|--|
| Choice 2 | Merge A and C; D and E that gives cost 11 and 5 respectively. Merge resulting two lists which will cost 16. Total cost would be $11 + 5 + 16 = 32$ | | | | | | |
| Choice 3 | Merge C and E, resulting list is merged with B and then finally with A; total cost = $5 + 10 + 16 = 31$ | | | | | | |
| | | | | | | | |
| optimal method | Always merge two small sized lists, then combined time would be reduced | | | | | | |
| | | | | | | | |
| Example: | | | | | | | |
| List | x1 | x2 | x3 | x4 | x5 | | |
| Sizes | 20 | 30 | 10 | 5 | 30 | | |
| | | | | | | | |
| Increasing order of sizes | 5 | 10 | 20 | 30 | 30 | | |
| Lists | x4 | x3 | x1 | x2 | x5 | | |
| | | | | | | | |
| | First x4 and x3 are merged, cost = 15; then result is merged with x1; cost = 35; x2 and x5 are merged with cost = 60; the two resulting lists are merged with cost = 95 | | | | | | |
| Total cost | $15 + 35 + 60 + 95 = 205$ | | | | | | |
| | $3*5 + 3*10 + 2*20 + 2*30 + 2*30 = 205$ | | | | | //multiplying distance of each node and size of each node | |
| | | | | | | | |
| 52 Huffman Coding | | | | | | | |
| | Compression technique used to reduce the size of data or message | | | | | | |
| | | | | | | | |
| Message | BCCABBDDEAECBBAEDDCC | | | | | | |
| | Length = 20 | | | | | | |
| | it has to be sent using ASCII codes (8-bit) | | | | | | |
| | A | 65 | 01000001 | | | Size = $8*20 = 160$ bits | |
| | B | 66 | 01000010 | | | | |

| | | | | | | |
|------------------|--|-----|-----|-----|-----|--|
| | C | 67 | | | | |
| | D | 68 | | | | |
| | E | 69 | | | | |
| | Can we use our own codes instead of ASCII codes? | | | | | |
| | Fixed size method | | | | | |
| Character | A | B | C | D | E | |
| Count | 3 | 5 | 6 | 4 | 2 | Total count = 20 |
| Code | 000 | 001 | 010 | 011 | 100 | |
| | | | | | | |
| message | BCCABBDDAECCBBAEDDCC | | | | | |
| bit code | 001010.... | | | | | size = 20*3 = 60 bits |
| | | | | | | 5*8 = 40 bits for ASCII code translations |
| | | | | | | 5*3 = 15 bits --> our assigned codes |
| | | | | | | 40 + 15 = 55 bits |
| | | | | | | message: 60 bits |
| | | | | | | char: 55 bits |
| | | | | | | total message size: 115 bits |
| | | | | | | so, the message size reduced from 160 bits to 115 bits |
| | | | | | | thus, 40% reduction in size with fixed sized code |
| | | | | | | |
| | Huffman coding - variable sized code | | | | | element that appears more / often should have a smaller sized code |
| character | A | B | C | D | E | |
| count | 3 | 5 | 6 | 4 | 2 | |
| code | | | | | | |
| | | | | | | |
| | first, arrange the letters with increasing count / frequency | | | | | |
| character | E | A | D | B | C | |
| count | 2 | 3 | 4 | 5 | 6 | |

| | | | |
|----------------------------------|---|---|--|
| code | 000 001 01 10 11 | Merge two smaller ones, we get 5, then combine with D, we get 9. Combine B and C, we get 11. Finally, combine two resulting lists, we get 20. | |
| bit count | 6 9 8 10 12 | On left side paths, mark as 0 and on right side mark as 1 | |
| total bits for message | 45 bits | Bit count for message can also be obtained from the tree, by counting number of edges for a letter and multiplying by the number of occurrences for that letter in the message i.e. summation of distance and frequency of a letter | |
| ASCII codes for chart | 5*8 = 40 bits | | |
| assigned codes | 12 bits | | |
| total bits for tree/table | 52 bits | | |
| Size of total msg | 52 + 45 = 97 bits | | |
| Message transferred | 001111101101111001011000111110100010100000110 | A tree or a table would be needed along with it | |
| Decoding | BCCD... | | |
| | | | |
| 53 | Minimum Cost Spanning Tree | | |
| | $G = (V, E)$ | | |
| | $V = \{1, 2, 3, 4, 5, 6\}$ | $ V = n = 6$ | |
| | $E = \{(1,2), (2,3), (3,4), (4,5), (5, 6), (6,1)\}$ | $ V - 1 = 5$ edges | |
| | the tree should not have a cycle | | |
| | | | |
| | S is a subset of G, WHERE IN $S = (V', E')$ | $V' = V; E' = V - 1$ | |
| | Number of edges in graph = 6 out of which I have to select 5 edges for spanning tree - thus i can select in 6C5 ways | Suppose we have 7 edges, out of which the seventh edge (3,5) divides the graph into two cycles of less tha 6 vertices, then we can select 5 edges for spanning tree in 7C5 - 2 ways | |

| | | | |
|------------------------|---|----------------|--|
| General formula | $ E C(V -1)$ - no. of cycles | | |
| | | | |
| | Now, if we have a weighted graph, I wish to know the number of possible spanning tree | | |
| | Vertices = 4 | | |
| | Edges = 3 | | |
| | cost = 14 | | |
| | similarly , depending upon the edges we select, cost may vary each time | | |
| | Can I found the minimum cost spanning tree? | | |
| Method 1 | Try all possible spanning trees and get the minimum cost spanning tree | | |
| Method 2 | Prim's algorithm (Greedy method) | | |
| Method 3 | Krskal's algorithm (Greedy method) | | |
| | | | |
| Method 2: | Prim's algorithm | | |
| | Select the minimum cost edge from the graph first | (6,1) ; w = 10 | |
| | Then, following this select minimum cost edge but make sure it is connected to previously chosen vertices | (5, 6); w = 25 | |
| | | (5, 4); w = 22 | |
| | | (4, 3); w = 12 | |
| | | (3, 2); w = 16 | |
| | | (2, 7); w = 14 | |
| | Now, if we add costs of all the chosen edges, total cost = 99 | | |
| | | | |
| | For non connected graphs we cannot find the minimum cost spanning tree or spanning tree | | |
| | | | |
| Method 3 | Kruskal's method | | |
| | Always select smallest cost edge | | |
| | (1,6); w = 10 | | |

| | | | |
|--|--|--|--|
| | (3, 4); w = 12 | | |
| | (2, 7); w = 14 | | |
| | (2, 3); w = 16 | | |
| | (4, 5); w = 22 | | |
| | (5, 6); w = 25 | | |
| | total cost = 99 | | |
| | | | |
| | vertices count : $ V $ | To get a minimum cost edge each time, min heap can be used | |
| | edges count: $ V - 1$ | $\theta(n \log n)$ | |
| | $\theta(V E)$ | | |
| | $\theta(n.e) = \theta(n^2)$ | | |
| | | | |
| | for non-connected graphs, spanning tree cannot be found | | |
| | Kruskal algo may give spanning tree for those non connected components but not for the graph as a whole | | |
| | | | |
| | if in a certain graph, certain edges' weights are missing, then use the given weights of remaining edges to guess the weight | | |
| | | | |
| | 54 Dijkstra algorithm | | |
| | Single source shortest path to all the vertices | | |
| | find the shortest path to a vertex and update it to other vertices. this updation is called relaxation | | |
| | | | |
| | Relaxation | | |
| | $\text{if}(d[u] + c(u,v) < d[v]) \{ d[v] = d[u] + c(u,v) \}$ | | |
| | | | |
| | no of vertices = $ V $ | | |
| | at most no. of vertices relaxing = $ V $ | | |
| | worst case time of Dijkstra algorithm: $\theta(n^2)$ | | |
| | | | |

| | | | | | | | |
|------------------------|--|------|-----------|-----------|-----------|----------|--|
| | Example - starting vertex is 1 | | | | | | |
| selected vertex | 2 | 3 | 4 | 5 | 6 | | |
| | 4 | 50 | 45 | 10 | infinity | infinity | |
| | 5 | 50 | 45 | 10 | 25 | infinity | |
| | 2 | 45 | 45 | 10 | 25 | infinity | |
| | 3 | 45 | 45 | 10 | 25 | infinity | |
| | 6 | 45 | 45 | 10 | 25 | infinity | |
| | Another example - starting vertex is 1 | | | | | | |
| selected vertex | {2, | 3, | 4} | | | | |
| | 2 | {3, | infinity, | 5} | | | |
| | 4 | {3, | infinity, | 5} | | | |
| | 3 | {3, | 7, | 5} | | | |
| | | {3, | 7, | 5} | | | |
| | Another example - starting vertex is 1 | | | | | | |
| selected vertex | {2, | 3, | 4} | | | | |
| | 2 | {3, | infinity, | 5} | | | |
| | 4 | {3, | infinity, | 5} | | | |
| | 3 | {3, | 7, | 5} | | | |
| | | {-3, | 7, | 5} | | | |
| | Dijkstra algorithm might work or might not work in case of an edge having negative weightage | | | | | | |
| 55 | Dynamic programming | | | | | | |
| | Dynamic programming vs greedy method | | | | | | |

| | | | |
|--|--|--|--|
| | In Greedy method, we try to follow a predefined procedure that gives us the best / optimal result. The procedure is already known for optimization. But, in dynamic programming, we try to get all the solutions and then decide the best solution. Mostly dynamic programming questions are solved using recursive procedures. They follow a principle of optimality. In greedy method, decision is taken just once and followed through whereas in dynamic programming, decision is taken at each step | | |
| | Example: | | |
| | <i>Fibonacci series</i> | | |
| | fib(n) = 0 if n = 0 | $T(n) = 2T(n-1) + 1$ {Approximating $T(n-2) \sim T(n-1)$ here} | |
| | fib(n) = 1 if n = 1 | Time taken would be $O(2^n)$ by using Master's theorem | |
| | fib(n) = fib(n-2) + fib(n-1) if n > 1 | Why can't we reduce the function calls to reduce the time taken? | |
| | | For this, we would take a global array and initially fill it with -1 | |
| | int fib(n) { | $F = \{-1, -1, -1, -1, -1\}$ | |
| | if(n <= 1){ | Then, as the function calls f(1), mark it as 1 | |
| | return n;} | Then f(0) is marked as 1 | |
| | return fib(n-2) + fib(n-1); | Then use the stored result to get the rest. | |
| | } | Finally, F would get updated as we solve: $F = \{0, 1, 1, 2, 3, 5\}$ | |
| | | Total 6 calls are made then i.e. n+1 calls i.e. $O(n)$ | |
| | | This is called result of memorization | |
| | From the above example, we can see reduction in number of calls from $O(2^n)$ to $O(n)$ using memorization. It follows top down approach. The same problem can be solved using tabular method (iterative process) as shown below: | | |
| | int fib(int n) { | | |
| | if(n <= 1) { | | |
| | return n;} | | |
| | F[0] = 0; F[1] = 1; | | |

| | | | |
|-------------|---|--|--|
| | for(int i = 2; i <= n; i++){ | | |
| | F[i] = F[i-2] + F[i - 1]; | | |
| | } | | |
| | return F[n]; | | |
| | } | | |
| | F= {0, 1, 1, 2, 3, 5} | | |
| | This is a bottom - up approach i.e. starting from F[0] and moving to F[n] | | |
| 56 | Multistage Graph | | |
| | <i>A multistage graph is a directed weighted graph. The vertices are divided into stages such that the edges are connecting vertices from one stage to next stage only. First stage and last stage will have only one vertex to represent start and end point. This is usually used to represent resource allocation.</i> | | |
| | The objective of the problem is that I have to select a path which gives me minimum cost. | //it is a minimization or optimization problem | |
| | Dynamic programming works on principle of optimality. Principle of optimality says that a problem can be solved in a sequence of decisions. | | |
| | From first stage I have to select one optimal vertex that leads to minimum cost and I have to take this decision at each stage. Thus, I can apply dynamic programming here. | | |
| V | 1 2 3 4 5 6 7 8 9 10 11 12 | | |
| Cost | 16 7 9 18 15 7 5 7 4 2 5 0 | cost(5, 12) = 0 ; here 5 is the stage and 12 is the vertex | |
| d | 2/3 7 6 8 8 10 10 10 12 12 12 12 | cost(4, 9) = 4 | |
| | | cost(4, 10) = 2 | |
| | Formula for multistage graph: | cost(4, 11) = 5 | |
| | cost(lth stage, jth vertex no.) = cost(i, j) = min{C(j, l) + cost(i+1, l)} | cost(3, 6) = min{ C(6, 9) + cost(4, 9) , C(6, 10) + cost(4, 10)} = min{6 + 4, 5 + 2} = 7 | |
| | | Similarly, cost(3, 7) = min{8, 5} = 5 | |
| | Now, we will solve it by going in forward direction and taking decisions based on above data; | cost(3, 8) = min{7, 11} = 7 | |

| | | | |
|-------------|--|---|--|
| | d(1,1) = 2 | Similarly, cost(2, 2) = min{C(2, 6) + cost(3, 6), C(2, 7) + cost(3, 7), C(2, 8) + cost(3, 8)} = min{11, 7, 8} = 7 | |
| | d(2,2) = 7 | cost(2, 3) = min{9, 12} = 9 | |
| | d(3, 7) = 10 | cost(2, 4) = min{18} = 18 | |
| | d(4, 10) = 12 | cost(2, 5) = min{16, 15} = 15 | |
| | Path: 2---> 7---> 10----> 12 | cost(1,1) = min{16, 16, 21, 17} = 16 | |
| | d(1, 1) = 3 | | |
| | d(2, 3) = 6 | | |
| | d(3, 6) = 10 | | |
| | d(4, 10) = 12 | | |
| | Path: 3---> 6---> 10----> 12 | | |
| | So, we have two paths with same cost. | | |
| | | | |
| 57 | Multistage Graph (Program) | | |
| | Cost adjacency Matrix | main(){ | |
| | 0 1 2 3 4 5 6 7 8 | int stages = 4, min; | |
| 0 | 0 0 0 0 0 0 0 0 0 | int n = 8; | |
| 1 | 0 0 2 1 3 0 0 0 0 | int cost[9], d[9], path[9]; | |
| 2 | 0 0 0 0 0 2 3 0 0 | int c[9][9] = {{0,0,0,0,0,0,0,0,0}, {0,0,2,1,3,0,0,0,0}, {0,0,0,0,0,2,3,0,0},} | |
| 3 | 0 0 0 0 0 6 7 0 0 | cost[n] = 0; | |
| 4 | 0 0 0 0 0 6 8 9 0 | for(int i = n-1; i >=1; i--){ | |
| 5 | 0 0 0 0 0 0 0 0 6 | min = 32767; | |
| 6 | 0 0 0 0 0 0 0 0 4 | for(int k = i + 1; k <= n; k++){ | |
| 7 | 0 0 0 0 0 0 0 0 5 | if(C[i][k] != 0 && C[i][k] + C[k] < min){ | |
| 8 | 0 0 0 0 0 0 0 0 0 | min = C[i][k] + C[k] ; | |
| | | d[i] = k; | |
| | 0 1 2 3 4 5 6 7 8 | | |
| cost | -- 9 7 11 12 6 4 5 0 | } | |
| d | -- 2 6 6 5 8 8 8 -- | } | |

| | | | |
|-------------|--|---|--|
| path | -- 1 2 6 8 | cost[i] = min; | |
| | | } | |
| | Path is calculated using the following formula: $p[i] \rightarrow d[p[i-1]]$; $p[2] = d[p[2-1]] = d[1] = 2$ | $p[1] = 1$; $p[\text{stages}] = n$; | |
| | time complexity: $O(n^2)$ | for($i = 2$; $i < \text{stages}$; $i++$){ | |
| | | $p[i] = p[d[i-1]]$; | |
| | | | |
| | 58 All Pairs Shortest Path | | |
| | If I use Dijkstra algorithm on each vertex to find the shortest path of all, the time complexity would be $O(n^3)$. | | |
| A0 = | 1 2 3 4 | | |
| | 1 0 3 INF 7 | | |
| | 2 8 0 2 INF | | |
| | 3 5 INF 0 1 | | |
| | 4 2 INF INF 0 | | |
| | | | |
| | Considering vertex 1 as intermediate vertex | $A0[2,3] \quad A0[2,1] + A0[1,3]$ | |
| A1 = | 1 2 3 4 | 2 < 8 + INF | |
| | 1 0 3 INF 7 | | |
| | 2 8 0 2 15 | $A0[2,4] \quad A0[2,1] + A0[1,4]$ | |
| | 3 5 8 0 1 | INF > 8 + 7 = 15 | |
| | 4 2 8 INF 0 | | |
| | | $A0[3,2] \quad A0[3,1] + A0[1,2]$ | |
| | Considering vertex 2 as intermediate vertex | INF > 5 + 3 = 8 | |
| A2 = | 1 2 3 4 | | |
| | 1 0 3 5 7 | $A[1,3] \quad A[1,2] + A[2,3]$ | |
| | 2 8 0 2 15 | INF < 3 + 2 = 5 | |
| | 3 5 8 0 1 | | |
| | 4 2 5 7 0 | for($k = 1$; $k \leq n$; $k++$){ | |
| | | for($i = 1$; $i \leq n$; $i++$){ | |
| | Considering vertex 3 as intermediate matrix | for($j = 1$; $j \leq n$; $j++$){ | |

| | | | | | | |
|-------------|--|----------|----------|----------|--|--|
| A3 = | 1 | 2 | 3 | 4 | $A[i,j] = \min(A[i,j], A[i,k] + A[k,j])$ | |
| 1 | 0 | 3 | 5 | 6 | } | |
| 2 | 7 | 0 | 2 | 3 | } | |
| 3 | 5 | 8 | 0 | 1 | } | |
| 4 | 2 | 5 | 7 | 0 | | |
| | | | | | | |
| | Considering vertex 4 as intermediate matrix | | | | | |
| A4 = | 1 | 2 | 3 | 4 | | |
| 1 | 0 | 3 | 5 | 6 | | |
| 2 | 5 | 0 | 2 | 3 | | |
| 3 | 3 | 6 | 0 | 1 | | |
| 4 | 2 | 5 | 7 | 0 | | |
| | | | | | | |
| | Formula used to make the above matrices: | | | | | |
| | $A[i,j] = \min\{A[i,j], A[i,k] + A[k, j]\}$ | | | | | |
| | time complexity = $O(n^3)$ | | | | | |
| | | | | | | |
| 59 | Matrix chain multiplication | | | | | |