**Topics :**

1. [Analysis of Algorithms](http://www.geeksforgeeks.org/fundamentals-of-algorithms/#AnalysisofAlgorithms)
2. [Searching and Sorting](http://www.geeksforgeeks.org/fundamentals-of-algorithms/#SearchingandSorting)
3. [Greedy Algorithms](http://www.geeksforgeeks.org/fundamentals-of-algorithms/#GreedyAlgorithms)
4. [Dynamic Programming](http://www.geeksforgeeks.org/fundamentals-of-algorithms/#DynamicProgramming)
5. [Pattern Searching](http://www.geeksforgeeks.org/fundamentals-of-algorithms/#PatternSearching)
6. [Other String Algorithms](http://www.geeksforgeeks.org/fundamentals-of-algorithms/#OtherStringAlgorithms)
7. [Backtracking](http://www.geeksforgeeks.org/fundamentals-of-algorithms/#Backtracking)
8. [Divide and Conquer](http://www.geeksforgeeks.org/fundamentals-of-algorithms/#DivideandConquer)
9. [Geometric Algorithms](http://www.geeksforgeeks.org/fundamentals-of-algorithms/#GeometricAlgorithms)
10. [Mathematical Algorithms](http://www.geeksforgeeks.org/fundamentals-of-algorithms/#MathematicalAlgorithms)
11. [Bit Algorithms](http://www.geeksforgeeks.org/fundamentals-of-algorithms/#BitAlgorithms)
12. [Graph Algorithms](http://www.geeksforgeeks.org/fundamentals-of-algorithms/#GraphAlgorithms)
13. [Randomized Algorithms](http://www.geeksforgeeks.org/fundamentals-of-algorithms/#RandomizedAlgorithms)
14. [Branch and Bound](http://www.geeksforgeeks.org/fundamentals-of-algorithms/#BranchandBound)
15. [Quizzes on Algorithms](http://www.geeksforgeeks.org/fundamentals-of-algorithms/#QuizzesonAlgorithms)
16. [Misc](http://www.geeksforgeeks.org/fundamentals-of-algorithms/#Misc)

**Analysis of Algorithms**:

1. [Asymptotic Analysis](http://www.geeksforgeeks.org/analysis-of-algorithms-set-1-asymptotic-analysis/)
2. [Worst, Average and Best Cases](http://www.geeksforgeeks.org/analysis-of-algorithms-set-2-asymptotic-analysis/)
3. [Asymptotic Notations](http://www.geeksforgeeks.org/analysis-of-algorithms-set-3asymptotic-notations/)
4. [Analysis of Loops](http://www.geeksforgeeks.org/analysis-algorithms-set-4-analysis-programs/)
5. [Solving Recurrences](http://www.geeksforgeeks.org/analysis-algorithm-set-4-master-method-solving-recurrences/)
6. [Amortized Analysis](http://www.geeksforgeeks.org/analysis-algorithm-set-5-amortized-analysis-introduction/)
7. [What does ‘Space Complexity’ mean?](http://www.geeksforgeeks.org/g-fact-86/)
8. [Pseudo-polynomial Algorithms](http://www.geeksforgeeks.org/pseudo-polynomial-in-algorithms/)
9. [NP-Completeness Introduction](http://www.geeksforgeeks.org/np-completeness-set-1/)
10. [Polynomial Time Approximation Scheme](http://www.geeksforgeeks.org/polynomial-time-approximation-scheme/)
11. [A Time Complexity Question](http://www.geeksforgeeks.org/a-time-complexity-question/)
12. [Time Complexity of building a heap](http://www.geeksforgeeks.org/g-fact-85/)
13. [Time Complexity where loop variable is incremented by 1, 2, 3, 4 ..](http://www.geeksforgeeks.org/time-complexity-where-loop-variable-is-incremented-by-1-2-3-4/)
14. [Time Complexity of Loop with Powers](http://www.geeksforgeeks.org/time-complexity-of-loop-with-powers/)
15. [Performance of loops (A caching question)](http://geeksquiz.com/performance-of-loops-a-caching-question/)

[**Recent Articles on Analysis of Algorithms**](http://www.geeksforgeeks.org/category/algorithm/analysis/)  
[Quiz on Analysis of Algorithms](http://geeksquiz.com/algorithms/analysis-of-algorithms/)  
[Quiz on Recurrences](http://geeksquiz.com/algorithms/analysis-of-algorithms-recurrences/)

**Searching and Sorting**:

1. [Linear Search](http://quiz.geeksforgeeks.org/linear-search/), [Binary Search](http://geeksquiz.com/binary-search/), [Jump Search](http://www.geeksforgeeks.org/jump-search/), [Interpolation Search](http://www.geeksforgeeks.org/interpolation-search/)
2. [Selection Sort](http://geeksquiz.com/selection-sort/), [Bubble Sort](http://geeksquiz.com/bubble-sort/), [Insertion Sort](http://geeksquiz.com/insertion-sort/), [Merge Sort](http://geeksquiz.com/merge-sort/), [Heap Sort](http://geeksquiz.com/heap-sort/), [QuickSort](http://geeksquiz.com/quick-sort/), [Radix Sort](http://www.geeksforgeeks.org/radix-sort/), [Counting Sort](http://www.geeksforgeeks.org/counting-sort/), [Bucket Sort](http://www.geeksforgeeks.org/bucket-sort-2/), [ShellSort](http://geeksquiz.com/shellsort/), [Comb Sort](http://www.geeksforgeeks.org/comb-sort/), [Pigeonhole Sort](http://www.geeksforgeeks.org/pigeonhole-sort/)
3. [Interpolation search vs Binary search](http://www.geeksforgeeks.org/g-fact-84/)
4. [Stability in sorting algorithms](http://www.geeksforgeeks.org/stability-in-sorting-algorithms/)
5. [When does the worst case of Quicksort occur?](http://www.geeksforgeeks.org/when-does-the-worst-case-of-quicksort-occur/)
6. [Lower bound for comparison based sorting algorithms](http://www.geeksforgeeks.org/lower-bound-on-comparison-based-sorting-algorithms/)
7. [Which sorting algorithm makes minimum number of memory writes?](http://www.geeksforgeeks.org/which-sorting-algorithm-makes-minimum-number-of-writes/)
8. [Find the Minimum length Unsorted Subarray, sorting which makes the complete array sorted](http://www.geeksforgeeks.org/minimum-length-unsorted-subarray-sorting-which-makes-the-complete-array-sorted/)
9. [Merge Sort for Linked Lists](http://www.geeksforgeeks.org/merge-sort-for-linked-list/)
10. [Sort a nearly sorted (or K sorted) array](http://www.geeksforgeeks.org/nearly-sorted-algorithm/)
11. [Iterative Quick Sort](http://www.geeksforgeeks.org/iterative-quick-sort/)
12. [QuickSort on Singly Linked List](http://www.geeksforgeeks.org/quicksort-on-singly-linked-list/)
13. [QuickSort on Doubly Linked List](http://www.geeksforgeeks.org/quicksort-for-linked-list/)
14. [Find k closest elements to a given value](http://www.geeksforgeeks.org/find-k-closest-elements-given-value/)
15. [Sort n numbers in range from 0 to n^2 – 1 in linear time](http://www.geeksforgeeks.org/sort-n-numbers-range-0-n2-1-linear-time/)
16. [A Problem in Many Binary Search Implementations](http://www.geeksforgeeks.org/problem-binary-search-implementations/)
17. [Search in an almost sorted array](http://www.geeksforgeeks.org/search-almost-sorted-array/)
18. [Sort an array in wave form](http://www.geeksforgeeks.org/sort-array-wave-form-2/)
19. [Why is Binary Search preferred over Ternary Search?](http://www.geeksforgeeks.org/binary-search-preferred-ternary-search/)
20. [K’th Smallest/Largest Element in Unsorted Array](http://www.geeksforgeeks.org/kth-smallestlargest-element-unsorted-array-set-2-expected-linear-time/)
21. [K’th Smallest/Largest Element in Unsorted Array in Expected Linear Time](http://www.geeksforgeeks.org/kth-smallestlargest-element-unsorted-array-set-2-expected-linear-time/)
22. [K’th Smallest/Largest Element in Unsorted Array in Worst Case Linear Time](http://www.geeksforgeeks.org/kth-smallestlargest-element-unsorted-array-set-3-worst-case-linear-time/)
23. [Find the closest pair from two sorted arrays](http://www.geeksforgeeks.org/given-two-sorted-arrays-number-x-find-pair-whose-sum-closest-x/)
24. [Find common elements in three sorted arrays](http://www.geeksforgeeks.org/find-common-elements-three-sorted-arrays/)
25. [Given a sorted array and a number x, find the pair in array whose sum is closest to x](http://geeksquiz.com/given-sorted-array-number-x-find-pair-array-whose-sum-closest-x/)
26. [Count 1’s in a sorted binary array](http://geeksquiz.com/count-1s-sorted-binary-array/)
27. [Binary Insertion Sort](http://geeksquiz.com/binary-insertion-sort/)
28. [Insertion Sort for Singly Linked List](http://geeksquiz.com/insertion-sort-for-singly-linked-list/)
29. [Why Quick Sort preferred for Arrays and Merge Sort for Linked Lists?](http://www.geeksforgeeks.org/why-quick-sort-preferred-for-arrays-and-merge-sort-for-linked-lists/)
30. [Merge Sort for Doubly Linked List](http://www.geeksforgeeks.org/merge-sort-for-doubly-linked-list/)

[**Recent Articles on Searching**](http://www.geeksforgeeks.org/category/searching/)  
[**Recent Articles on Sorting**](http://www.geeksforgeeks.org/category/sorting/)  
[Quiz on Searching](http://geeksquiz.com/algorithms/searching/)  
[Quiz on Sorting](http://geeksquiz.com/algorithms/searching-and-sorting/)  
[Coding Practice on Searching](http://www.practice.geeksforgeeks.org/tag-page.php?tag=searching&isCmp=0)  
[Coding Practice on Sorting](http://www.practice.geeksforgeeks.org/tag-page.php?tag=sorting&isCmp=0)

**Greedy Algorithms**:

1. [Activity Selection Problem](http://www.geeksforgeeks.org/greedy-algorithms-set-1-activity-selection-problem/)
2. [Kruskal’s Minimum Spanning Tree Algorithm](http://www.geeksforgeeks.org/greedy-algorithms-set-2-kruskals-minimum-spanning-tree-mst/)
3. [Huffman Coding](http://www.geeksforgeeks.org/greedy-algorithms-set-3-huffman-coding/)
4. [Efficient Huffman Coding for Sorted Input](http://www.geeksforgeeks.org/greedy-algorithms-set-3-huffman-coding-set-2/)
5. [Prim’s Minimum Spanning Tree Algorithm](http://www.geeksforgeeks.org/greedy-algorithms-set-5-prims-minimum-spanning-tree-mst-2/)
6. [Prim’s MST for Adjacency List Representation](http://www.geeksforgeeks.org/greedy-algorithms-set-5-prims-mst-for-adjacency-list-representation/)
7. [Dijkstra’s Shortest Path Algorithm](http://www.geeksforgeeks.org/greedy-algorithms-set-6-dijkstras-shortest-path-algorithm/)
8. [Dijkstra’s Algorithm for Adjacency List Representation](http://www.geeksforgeeks.org/greedy-algorithms-set-7-dijkstras-algorithm-for-adjacency-list-representation/)
9. [Job Sequencing Problem](http://www.geeksforgeeks.org/job-sequencing-problem-set-1-greedy-algorithm/)
10. [Quiz on Greedy Algorithms](http://geeksquiz.com/algorithms/greedy-algorithms/)
11. [Greedy Algorithm to find Minimum number of Coins](http://geeksquiz.com/greedy-algorithm-to-find-minimum-number-of-coins/)
12. [K Centers Problem](http://www.geeksforgeeks.org/k-centers-problem-set-1-greedy-approximate-algorithm/)

[**Recent Articles on Greedy Algorithms**](http://www.geeksforgeeks.org/category/algorithm/greedy/)  
[Quiz on Greedy Algorithms](http://geeksquiz.com/algorithms/greedy-algorithms/)  
[Coding Practice on Greedy Algorithms](http://www.practice.geeksforgeeks.org/tag-page.php?tag=greedy&isCmp=0)

**Dynamic Programming**:

1. [Overlapping Subproblems Property](http://www.geeksforgeeks.org/dynamic-programming-set-1/)
2. [Optimal Substructure Property](http://www.geeksforgeeks.org/dynamic-programming-set-2-optimal-substructure-property/)
3. [Longest Increasing Subsequence](http://www.geeksforgeeks.org/dynamic-programming-set-3-longest-increasing-subsequence/)
4. [Longest Common Subsequence](http://www.geeksforgeeks.org/dynamic-programming-set-4-longest-common-subsequence/)
5. [Edit Distance](http://www.geeksforgeeks.org/dynamic-programming-set-5-edit-distance/)
6. [Min Cost Path](http://www.geeksforgeeks.org/dynamic-programming-set-6-min-cost-path/)
7. [Coin Change](http://www.geeksforgeeks.org/dynamic-programming-set-7-coin-change/)
8. [Matrix Chain Multiplication](http://www.geeksforgeeks.org/dynamic-programming-set-8-matrix-chain-multiplication/)
9. [Binomial Coefficient](http://www.geeksforgeeks.org/dynamic-programming-set-9-binomial-coefficient/)
10. [0-1 Knapsack Problem](http://www.geeksforgeeks.org/dynamic-programming-set-10-0-1-knapsack-problem/)
11. [Egg Dropping Puzzle](http://www.geeksforgeeks.org/dynamic-programming-set-11-egg-dropping-puzzle/)
12. [Longest Palindromic Subsequence](http://www.geeksforgeeks.org/dynamic-programming-set-12-longest-palindromic-subsequence/)
13. [Cutting a Rod](http://www.geeksforgeeks.org/dynamic-programming-set-13-cutting-a-rod/)
14. [Maximum Sum Increasing Subsequence](http://www.geeksforgeeks.org/dynamic-programming-set-14-maximum-sum-increasing-subsequence/)
15. [Longest Bitonic Subsequence](http://www.geeksforgeeks.org/dynamic-programming-set-15-longest-bitonic-subsequence/)
16. [Floyd Warshall Algorithm](http://www.geeksforgeeks.org/dynamic-programming-set-16-floyd-warshall-algorithm/)
17. [Palindrome Partitioning](http://www.geeksforgeeks.org/dynamic-programming-set-17-palindrome-partitioning/)
18. [Partition problem](http://www.geeksforgeeks.org/dynamic-programming-set-18-partition-problem/)
19. [Word Wrap Problem](http://www.geeksforgeeks.org/dynamic-programming-set-18-word-wrap/)
20. [Maximum Length Chain of Pairs](http://www.geeksforgeeks.org/dynamic-programming-set-20-maximum-length-chain-of-pairs/)
21. [Variations of LIS](http://www.geeksforgeeks.org/dynamic-programming-set-14-variations-of-lis/)
22. [Box Stacking Problem](http://www.geeksforgeeks.org/dynamic-programming-set-21-box-stacking-problem/)
23. [Program for Fibonacci numbers](http://www.geeksforgeeks.org/program-for-nth-fibonacci-number/)
24. [Minimum number of jumps to reach end](http://www.geeksforgeeks.org/minimum-number-of-jumps-to-reach-end-of-a-given-array/)
25. [Maximum size square sub-matrix with all 1s](http://www.geeksforgeeks.org/maximum-size-sub-matrix-with-all-1s-in-a-binary-matrix/)
26. [Ugly Numbers](http://www.geeksforgeeks.org/ugly-numbers/)
27. [Largest Sum Contiguous Subarray](http://www.geeksforgeeks.org/largest-sum-contiguous-subarray/)
28. [Longest Palindromic Substring](http://www.geeksforgeeks.org/longest-palindrome-substring-set-1/)
29. [Bellman–Ford Algorithm for Shortest Paths](http://www.geeksforgeeks.org/dynamic-programming-set-23-bellman-ford-algorithm/)
30. [Optimal Binary Search Tree](http://www.geeksforgeeks.org/dynamic-programming-set-24-optimal-binary-search-tree/)
31. [Largest Independent Set Problem](http://www.geeksforgeeks.org/largest-independent-set-problem/)
32. [Subset Sum Problem](http://www.geeksforgeeks.org/dynamic-programming-subset-sum-problem/)
33. [Maximum sum rectangle in a 2D matrix](http://www.geeksforgeeks.org/dynamic-programming-set-27-max-sum-rectangle-in-a-2d-matrix/)
34. [Count number of binary strings without consecutive 1?s](http://www.geeksforgeeks.org/count-number-binary-strings-without-consecutive-1s/)
35. [Boolean Parenthesization Problem](http://www.geeksforgeeks.org/dynamic-programming-set-37-boolean-parenthesization-problem/)
36. [Count ways to reach the n’th stair](http://www.geeksforgeeks.org/count-ways-reach-nth-stair/)
37. [Minimum Cost Polygon Triangulation](http://www.geeksforgeeks.org/minimum-cost-polygon-triangulation/)
38. [Mobile Numeric Keypad Problem](http://www.geeksforgeeks.org/mobile-numeric-keypad-problem/)
39. [Count of n digit numbers whose sum of digits equals to given sum](http://www.geeksforgeeks.org/count-of-n-digit-numbers-whose-sum-of-digits-equals-to-given-sum/)
40. [Minimum Initial Points to Reach Destination](http://www.geeksforgeeks.org/minimum-positive-points-to-reach-destination/)
41. [Total number of non-decreasing numbers with n digits](http://www.geeksforgeeks.org/total-number-of-non-decreasing-numbers-with-n-digits/)
42. [Find length of the longest consecutive path from a given starting character](http://www.geeksforgeeks.org/find-length-of-the-longest-consecutive-path-in-a-character-matrix/)
43. [Tiling Problem](http://www.geeksforgeeks.org/tiling-problem/)
44. [Minimum number of squares whose sum equals to given number n](http://www.geeksforgeeks.org/minimum-number-of-squares-whose-sum-equals-to-given-number-n/)
45. [Find minimum number of coins that make a given value](http://www.geeksforgeeks.org/find-minimum-number-of-coins-that-make-a-change/)
46. [Collect maximum points in a grid using two traversals](http://www.geeksforgeeks.org/collect-maximum-points-in-a-grid-using-two-traversals/)
47. [Shortest Common Supersequence](http://www.geeksforgeeks.org/shortest-common-supersequence/)
48. [Compute sum of digits in all numbers from 1 to n](http://www.geeksforgeeks.org/count-sum-of-digits-in-numbers-from-1-to-n/)
49. [Count possible ways to construct buildings](http://www.geeksforgeeks.org/count-possible-ways-to-construct-buildings/)
50. [Maximum profit by buying and selling a share at most twice](http://www.geeksforgeeks.org/maximum-profit-by-buying-and-selling-a-share-at-most-twice/)
51. [How to print maximum number of A’s using given four keys](http://www.geeksforgeeks.org/how-to-print-maximum-number-of-a-using-given-four-keys/)
52. [Find the minimum cost to reach destination using a train](http://www.geeksforgeeks.org/find-the-minimum-cost-to-reach-a-destination-where-every-station-is-connected-in-one-direction/)
53. [Vertex Cover Problem | Set 2 (Dynamic Programming Solution for Tree)](http://www.geeksforgeeks.org/vertex-cover-problem-set-2-dynamic-programming-solution-tree/)
54. [Count number of ways to reach a given score in a game](http://www.geeksforgeeks.org/count-number-ways-reach-given-score-game/)
55. [Weighted Job Scheduling](http://www.geeksforgeeks.org/weighted-job-scheduling/)
56. [Longest Even Length Substring such that Sum of First and Second Half is same](http://www.geeksforgeeks.org/longest-even-length-substring-sum-first-second-half/)

[**Recent Articles on Dynamic Programming**](http://www.geeksforgeeks.org/category/algorithm/dynamic-programming/)  
[Quiz on Dynamic Programming](http://geeksquiz.com/algorithms/dynamic-programming/)  
[Coding Practice on Dynamic Programing](http://www.practice.geeksforgeeks.org/tag-page.php?tag=dp&isCmp=0)

**Pattern Searching:**

1. [Naive Pattern Searching](http://www.geeksforgeeks.org/searching-for-patterns-set-1-naive-pattern-searching/)
2. [KMP Algorithm](http://www.geeksforgeeks.org/searching-for-patterns-set-2-kmp-algorithm/)
3. [Rabin-Karp Algorithm](http://www.geeksforgeeks.org/searching-for-patterns-set-3-rabin-karp-algorithm/)
4. [A Naive Pattern Searching Question](http://www.geeksforgeeks.org/pattern-searching-set-4-a-naive-string-matching-algo-question/)
5. [Finite Automata](http://www.geeksforgeeks.org/searching-for-patterns-set-5-finite-automata/)
6. [Efficient Construction of Finite Automata](http://www.geeksforgeeks.org/pattern-searching-set-5-efficient-constructtion-of-finite-automata/)
7. [Boyer Moore Algorithm – Bad Character Heuristic](http://www.geeksforgeeks.org/pattern-searching-set-7-boyer-moore-algorithm-bad-character-heuristic/)
8. [Suffix Array](http://www.geeksforgeeks.org/suffix-array-set-1-introduction/)
9. [Anagram Substring Search (Or Search for all permutations)](http://www.geeksforgeeks.org/anagram-substring-search-search-permutations/)
10. [Pattern Searching using a Trie of all Suffixes](http://www.geeksforgeeks.org/pattern-searching-using-trie-suffixes/)
11. [Aho-Corasick Algorithm for Pattern Searching](http://www.geeksforgeeks.org/aho-corasick-algorithm-pattern-searching/)
12. [kasai’s Algorithm for Construction of LCP array from Suffix Array](http://www.geeksforgeeks.org/%c2%ad%c2%adkasais-algorithm-for-construction-of-lcp-array-from-suffix-array/)
13. [Z algorithm (Linear time pattern searching Algorithm)](http://www.geeksforgeeks.org/z-algorithm-linear-time-pattern-searching-algorithm/)

[**Recent Articles on Pattern Searching**](http://www.geeksforgeeks.org/category/algorithm/pattern-searching/)

**Other String Algorithms:**

1. [Manacher’s Algorithm – Linear Time Longest Palindromic Substring – Part 1](http://www.geeksforgeeks.org/manachers-algorithm-linear-time-longest-palindromic-substring-part-1/), [Part 2](http://www.geeksforgeeks.org/manachers-algorithm-linear-time-longest-palindromic-substring-part-2/), [Part 3](http://www.geeksforgeeks.org/manachers-algorithm-linear-time-longest-palindromic-substring-part-3-2/), [Part 4](http://www.geeksforgeeks.org/manachers-algorithm-linear-time-longest-palindromic-substring-part-4/)
2. [Longest Even Length Substring such that Sum of First and Second Half is same](http://www.geeksforgeeks.org/longest-even-length-substring-sum-first-second-half/)
3. [Print all possible strings that can be made by placing spaces](http://www.geeksforgeeks.org/print-possible-strings-can-made-placing-spaces/)

[**Recent Articles on Strings**](http://www.geeksforgeeks.org/category/c-strings/)  
[Coding practice on Strings](http://www.practice.geeksforgeeks.org/tag-page.php?tag=string&isCmp=0)

**Backtracking**:

1. [Print all permutations of a given string](http://www.geeksforgeeks.org/write-a-c-program-to-print-all-permutations-of-a-given-string/)
2. [The Knight’s tour problem](http://www.geeksforgeeks.org/backtracking-set-1-the-knights-tour-problem/)
3. [Rat in a Maze](http://www.geeksforgeeks.org/backttracking-set-2-rat-in-a-maze/)
4. [N Queen Problem](http://www.geeksforgeeks.org/backtracking-set-3-n-queen-problem/)
5. [Subset Sum](http://www.geeksforgeeks.org/backttracking-set-4-subset-sum/)
6. [m Coloring Problem](http://www.geeksforgeeks.org/backttracking-set-5-m-coloring-problem/)
7. [Hamiltonian Cycle](http://www.geeksforgeeks.org/backtracking-set-7-hamiltonian-cycle/)
8. [Sudoku](http://www.geeksforgeeks.org/backtracking-set-7-suduku/)
9. [Tug of War](http://www.geeksforgeeks.org/tug-of-war/)
10. [Solving Cryptarithmetic Puzzles](http://www.geeksforgeeks.org/backtracking-set-8-solving-cryptarithmetic-puzzles/)

[**Recent Articles on Backtracking**](http://www.geeksforgeeks.org/category/algorithm/backtracking/)  
[Coding Practice on Backtracking](http://www.practice.geeksforgeeks.org/tag-page.php?tag=backtracking&isCmp=0)

**Divide and Conquer**:

1. [Introduction](http://www.geeksforgeeks.org/divide-and-conquer-set-1-find-closest-pair-of-points/)
2. [Write your own pow(x, n) to calculate x\*n](http://www.geeksforgeeks.org/write-a-c-program-to-calculate-powxn/)
3. [Median of two sorted arrays](http://www.geeksforgeeks.org/median-of-two-sorted-arrays/)
4. [Count Inversions](http://www.geeksforgeeks.org/counting-inversions/)
5. [Closest Pair of Points](http://www.geeksforgeeks.org/closest-pair-of-points/)
6. [Strassen’s Matrix Multiplication](http://www.geeksforgeeks.org/strassens-matrix-multiplication/)

[**Recent Articles on Divide and Conquer**](http://www.geeksforgeeks.org/category/algorithm/divide-and-conquer/)  
[Quiz on Divide and Conquer](http://geeksquiz.com/algorithms/divide-and-conquer/)  
[Coding practice on Divide and Conquer](http://www.practice.geeksforgeeks.org/tag-page.php?tag=divide-and-conquer&isCmp=0)

**Geometric Algorithms:**

1. [Closest Pair of Points | O(nlogn) Implementation](http://www.geeksforgeeks.org/closest-pair-of-points-onlogn-implementation/)
2. [How to check if two given line segments intersect?](http://www.geeksforgeeks.org/check-if-two-given-line-segments-intersect/)
3. [How to check if a given point lies inside or outside a polygon?](http://www.geeksforgeeks.org/how-to-check-if-a-given-point-lies-inside-a-polygon/)
4. [Convex Hull | Set 1 (Jarvis’s Algorithm or Wrapping)](http://www.geeksforgeeks.org/convex-hull-set-1-jarviss-algorithm-or-wrapping/)
5. [Convex Hull | Set 2 (Graham Scan)](http://www.geeksforgeeks.org/convex-hull-set-2-graham-scan/)
6. [Given n line segments, find if any two segments intersect](http://www.geeksforgeeks.org/given-a-set-of-line-segments-find-if-any-two-segments-intersect/)
7. [Check whether a given point lies inside a triangle or not](http://www.geeksforgeeks.org/check-whether-a-given-point-lies-inside-a-triangle-or-not/)
8. [How to check if given four points form a square](http://geeksquiz.com/check-given-four-points-form-square/)

[**Recent Articles on Geometric Algorithms**](http://www.geeksforgeeks.org/category/algorithm/geometric/)  
[Coding Practice on Geometric Algorithms](http://www.practice.geeksforgeeks.org/tag-page.php?tag=geometry&isCmp=0)

**Mathematical Algorithms:**

1. [Write an Efficient Method to Check if a Number is Multiple of 3](http://www.geeksforgeeks.org/write-an-efficient-method-to-check-if-a-number-is-multiple-of-3/)
2. [Efficient way to multiply with 7](http://www.geeksforgeeks.org/efficient-way-to-multiply-with-7/)
3. [Write a C program to print all permutations of a given string](http://www.geeksforgeeks.org/write-a-c-program-to-print-all-permutations-of-a-given-string/)
4. [Lucky Numbers](http://www.geeksforgeeks.org/lucky-numbers/)
5. [Write a program to add two numbers in base 14](http://www.geeksforgeeks.org/write-a-program-to-add-two-numbers-in-base-14/)
6. [Babylonian method for square root](http://www.geeksforgeeks.org/square-root-of-a-perfect-square/)
7. [Multiply two integers without using multiplication, division and bitwise operators, and no loops](http://www.geeksforgeeks.org/multiply-two-numbers-without-using-multiply-division-bitwise-operators-and-no-loops/)
8. [Print all combinations of points that can compose a given number](http://www.geeksforgeeks.org/print-all-combinations-of-points-that-can-compose-a-given-number/)
9. [Write you own Power without using multiplication(\*) and division(/) operators](http://www.geeksforgeeks.org/write-you-own-power-without-using-multiplication-and-division/)
10. [Program for Fibonacci numbers](http://www.geeksforgeeks.org/program-for-nth-fibonacci-number/)
11. [Average of a stream of numbers](http://www.geeksforgeeks.org/average-of-a-stream-of-numbers/)
12. [Count numbers that don’t contain 3](http://www.geeksforgeeks.org/count-numbers-that-dont-contain-3/)
13. [MagicSquare](http://www.geeksforgeeks.org/magic-square/)
14. [Sieve of Eratosthenes](http://www.geeksforgeeks.org/sieve-of-eratosthenes/)
15. [Find day of the week for a given date](http://www.geeksforgeeks.org/find-day-of-the-week-for-a-given-date/)
16. [DFA based division](http://www.geeksforgeeks.org/dfa-based-division/)
17. [Generate integer from 1 to 7 with equal probability](http://www.geeksforgeeks.org/generate-integer-from-1-to-7-with-equal-probability/)
18. [Given a number, find the next smallest palindrome](http://www.geeksforgeeks.org/given-a-number-find-next-smallest-palindrome-larger-than-this-number/)
19. [Make a fair coin from a biased coin](http://www.geeksforgeeks.org/print-0-and-1-with-50-probability/)
20. [Check divisibility by 7](http://www.geeksforgeeks.org/divisibility-by-7/)
21. [Find the largest multiple of 3](http://www.geeksforgeeks.org/find-the-largest-number-multiple-of-3/)
22. [Lexicographic rank of a string](http://www.geeksforgeeks.org/lexicographic-rank-of-a-string/)
23. [Print all permutations in sorted (lexicographic) order](http://www.geeksforgeeks.org/lexicographic-permutations-of-string/)
24. [Shuffle a given array](http://www.geeksforgeeks.org/shuffle-a-given-array/)
25. [Space and time efficient Binomial Coefficient](http://www.geeksforgeeks.org/space-and-time-efficient-binomial-coefficient/)
26. [Reservoir Sampling](http://www.geeksforgeeks.org/reservoir-sampling/)
27. [Pascal’s Triangle](http://www.geeksforgeeks.org/pascal-triangle/)
28. [Select a random number from stream, with O(1) space](http://www.geeksforgeeks.org/select-a-random-number-from-stream-with-o1-space/)
29. [Find the largest multiple of 2, 3 and 5](http://www.geeksforgeeks.org/find-the-largest-multiple-of-2-3-and-5/)
30. [Efficient program to calculate e^x](http://www.geeksforgeeks.org/program-to-efficiently-calculate-ex/)
31. [Measure one litre using two vessels and infinite water supply](http://www.geeksforgeeks.org/measure-1-litre-from-two-vessels-infinite-water-supply/)
32. [Efficient program to print all prime factors of a given number](http://www.geeksforgeeks.org/print-all-prime-factors-of-a-given-number/)
33. [Print all possible combinations of r elements in a given array of size n](http://www.geeksforgeeks.org/print-all-possible-combinations-of-r-elements-in-a-given-array-of-size-n/)
34. [Random number generator in arbitrary probability distribution fashion](http://www.geeksforgeeks.org/random-number-generator-in-arbitrary-probability-distribution-fashion/)
35. [How to check if a given number is Fibonacci number?](http://www.geeksforgeeks.org/check-number-fibonacci-number/)
36. [Russian Peasant Multiplication](http://www.geeksforgeeks.org/fast-multiplication-method-without-using-multiplication-operator-russian-peasants-algorithm/)
37. [Count all possible groups of size 2 or 3 that have sum as multiple of 3](http://www.geeksforgeeks.org/count-possible-groups-size-2-3-sum-multiple-3/)
38. [Tower of Hanoi](http://geeksquiz.com/c-program-for-tower-of-hanoi/)
39. [Horner’s Method for Polynomial Evaluation](http://www.geeksforgeeks.org/horners-method-polynomial-evaluation/)
40. [Count trailing zeroes in factorial of a number](http://www.geeksforgeeks.org/count-trailing-zeroes-factorial-number/)
41. [Program for nth Catalan Number](http://www.geeksforgeeks.org/program-nth-catalan-number/)
42. [Generate one of 3 numbers according to given probabilities](http://www.geeksforgeeks.org/write-a-function-to-generate-3-numbers-according-to-given-probabilities/)
43. [Find Excel column name from a given column number](http://www.geeksforgeeks.org/find-excel-column-name-given-number/)
44. [Find next greater number with same set of digits](http://www.geeksforgeeks.org/find-next-greater-number-set-digits/)
45. [Count Possible Decodings of a given Digit Sequence](http://www.geeksforgeeks.org/count-possible-decodings-given-digit-sequence/)
46. [Calculate the angle between hour hand and minute hand](http://www.geeksforgeeks.org/calculate-angle-hour-hand-minute-hand/)
47. [Count number of binary strings without consecutive 1?s](http://www.geeksforgeeks.org/count-number-binary-strings-without-consecutive-1s/)
48. [Find the smallest number whose digits multiply to a given number n](http://www.geeksforgeeks.org/find-smallest-number-whose-digits-multiply-given-number-n/)
49. [Draw a circle without floating point arithmetic](http://geeksquiz.com/draw-circle-without-floating-point-arithmetic/)
50. [How to check if an instance of 8 puzzle is solvable?](http://www.geeksforgeeks.org/check-instance-8-puzzle-solvable/)
51. [Birthday Paradox](http://www.geeksforgeeks.org/birthday-paradox/)
52. [Multiply two polynomials](http://www.geeksforgeeks.org/multiply-two-polynomials-2/)
53. [Count Distinct Non-Negative Integer Pairs (x, y) that Satisfy the Inequality x\*x + y\*y < n](http://www.geeksforgeeks.org/count-distinct-non-negative-pairs-x-y-satisfy-inequality-xx-yy-n-2/)
54. [Count ways to reach the n’th stair](http://www.geeksforgeeks.org/count-ways-reach-nth-stair/)
55. [Replace all ‘0’ with ‘5’ in an input Integer](http://geeksquiz.com/replace-0-5-input-integer/)
56. [Program to add two polynomials](http://geeksquiz.com/program-add-two-polynomials/)
57. [Print first k digits of 1/n where n is a positive integer](http://geeksquiz.com/print-first-k-digits-1n-n-positive-integer/)
58. [Given a number as a string, find the number of contiguous subsequences which recursively add up to 9](http://geeksquiz.com/given-number-find-number-contiguous-subsequences-recursively-add-9/)

[**Recent Articles on Mathematical Algorithms**](http://www.geeksforgeeks.org/category/algorithm/mathematical/)  
[Coding Practice on Mathematical Algorithms](http://www.practice.geeksforgeeks.org/tag-page.php?tag=maths&isCmp=0)

**Bit Algorithms:**

1. [Find the element that appears once](http://www.geeksforgeeks.org/find-the-element-that-appears-once/)
2. [Detect opposite signs](http://www.geeksforgeeks.org/detect-if-two-integers-have-opposite-signs/)
3. [Set bits in all numbers from 1 to n](http://www.geeksforgeeks.org/count-total-set-bits-in-all-numbers-from-1-to-n/)
4. [Swap bits](http://www.geeksforgeeks.org/swap-bits-in-a-given-number/)
5. [Add two numbers](http://www.geeksforgeeks.org/add-two-numbers-without-using-arithmetic-operators/)
6. [Smallest of three](http://www.geeksforgeeks.org/smallest-of-three-integers-without-comparison-operators/)
7. [A Boolean Array Puzzle](http://www.geeksforgeeks.org/a-boolean-array-puzzle/)
8. [Set bits in an (big) array](http://www.geeksforgeeks.org/program-to-count-number-of-set-bits-in-an-big-array/)
9. [Next higher number with same number of set bits](http://www.geeksforgeeks.org/next-higher-number-with-same-number-of-set-bits/)
10. [Optimization Technique (Modulus)](http://www.geeksforgeeks.org/optimization-techniques-set-1-modulus/)
11. [Add 1 to a number](http://www.geeksforgeeks.org/add-1-to-a-given-number/)
12. [Multiply with 3.5](http://www.geeksforgeeks.org/multiply-an-integer-with-3-5/)
13. [Turn off the rightmost set bit](http://www.geeksforgeeks.org/turn-off-the-rightmost-set-bit/)
14. [Check for Power of 4](http://www.geeksforgeeks.org/find-whether-a-given-number-is-a-power-of-4-or-not/)
15. [Absolute value (abs) without branching](http://www.geeksforgeeks.org/compute-the-integer-absolute-value-abs-without-branching/)
16. [Modulus division by a power-of-2-number](http://www.geeksforgeeks.org/compute-modulus-division-by-a-power-of-2-number/)
17. [Minimum or Maximum of two integers](http://www.geeksforgeeks.org/compute-the-minimum-or-maximum-max-of-two-integers-without-branching/)
18. [Rotate bits](http://www.geeksforgeeks.org/rotate-bits-of-an-integer/)
19. [Find the two non-repeating elements in an array](http://www.geeksforgeeks.org/find-two-non-repeating-elements-in-an-array-of-repeating-elements/)
20. [Number Occurring Odd Number of Times](http://www.geeksforgeeks.org/find-the-number-occurring-odd-number-of-times/)
21. [Check for Integer Overflow](http://www.geeksforgeeks.org/check-for-integer-overflow/)
22. [Little and Big Endian](http://www.geeksforgeeks.org/little-and-big-endian-mystery/)
23. [Reverse Bits of a Number](http://www.geeksforgeeks.org/write-an-efficient-c-program-to-reverse-bits-of-a-number/)
24. [Count set bits in an integer](http://www.geeksforgeeks.org/count-set-bits-in-an-integer/)
25. [Number of bits to be flipped to convert A to B](http://www.geeksforgeeks.org/count-number-of-bits-to-be-flipped-to-convert-a-to-b/)
26. [Next Power of 2](http://www.geeksforgeeks.org/next-power-of-2/)
27. [Check if a Number is Multiple of 3](http://www.geeksforgeeks.org/write-an-efficient-method-to-check-if-a-number-is-multiple-of-3/)
28. [Find parity](http://www.geeksforgeeks.org/write-a-c-program-to-find-the-parity-of-an-unsigned-integer/)
29. [Multiply with 7](http://www.geeksforgeeks.org/efficient-way-to-multiply-with-7/)
30. [Find whether a no is power of two](http://www.geeksforgeeks.org/write-one-line-c-function-to-find-whether-a-no-is-power-of-two/)
31. [Position of rightmost set bit](http://www.geeksforgeeks.org/position-of-rightmost-set-bit/)
32. [Binary representation of a given number](http://www.geeksforgeeks.org/binary-representation-of-a-given-number/)
33. [Swap all odd and even bits](http://www.geeksforgeeks.org/swap-all-odd-and-even-bits/)
34. [Find position of the only set bit](http://www.geeksforgeeks.org/find-position-of-the-only-set-bit/)
35. [Karatsuba algorithm for fast multiplication](http://www.geeksforgeeks.org/divide-and-conquer-set-2-karatsuba-algorithm-for-fast-multiplication/)
36. [How to swap two numbers without using a temporary variable?](http://www.geeksforgeeks.org/swap-two-numbers-without-using-temporary-variable/)
37. [Check if a number is multiple of 9 using bitwise operators](http://www.geeksforgeeks.org/divisibility-9-using-bitwise-operators/)
38. [Swap two nibbles in a byte](http://www.geeksforgeeks.org/swap-two-nibbles-byte/)
39. [How to turn off a particular bit in a number?](http://www.geeksforgeeks.org/how-to-turn-off-a-particular-bit-in-a-number/)
40. [Check if binary representation of a number is palindrome](http://www.geeksforgeeks.org/check-binary-representation-number-palindrome/)

[**Recent Articles on Bit Algorithms**](http://www.geeksforgeeks.org/category/bit-magic/)  
[Quiz on Bit Algorithms](http://geeksquiz.com/algorithms/bit-algorithms/)  
[Coding Practice on Bit Algorithms](http://www.practice.geeksforgeeks.org/tag-page.php?tag=bit&isCmp=0)

**Graph Algorithms:**

***Introduction, DFS and BFS:***

1. [Graph and its representations](http://www.geeksforgeeks.org/graph-and-its-representations/)
2. [Breadth First Traversal for a Graph](http://www.geeksforgeeks.org/breadth-first-traversal-for-a-graph/)
3. [Depth First Traversal for a Graph](http://www.geeksforgeeks.org/depth-first-traversal-for-a-graph/)
4. [Applications of Depth First Search](http://www.geeksforgeeks.org/applications-of-depth-first-search/)
5. [Detect Cycle in a Directed Graph](http://www.geeksforgeeks.org/detect-cycle-in-a-graph/)
6. [Detect Cycle in a an Undirected Graph](http://www.geeksforgeeks.org/union-find/)
7. [Detect cycle in an undirected graph](http://www.geeksforgeeks.org/detect-cycle-undirected-graph/)
8. [Longest Path in a Directed Acyclic Graph](http://www.geeksforgeeks.org/find-longest-path-directed-acyclic-graph/)
9. [Topological Sorting](http://www.geeksforgeeks.org/topological-sorting/)
10. [Check whether a given graph is Bipartite or not](http://www.geeksforgeeks.org/bipartite-graph/)
11. [Snake and Ladder Problem](http://www.geeksforgeeks.org/snake-ladder-problem-2/)
12. [Biconnected Components](http://www.geeksforgeeks.org/biconnected-components/)
13. [Check if a given graph is tree or not](http://geeksquiz.com/check-given-graph-tree/)

***Minimum Spanning Tree:***

1. [Prim’s Minimum Spanning Tree (MST))](http://www.geeksforgeeks.org/greedy-algorithms-set-5-prims-minimum-spanning-tree-mst-2/)
2. [Applications of Minimum Spanning Tree Problem](http://www.geeksforgeeks.org/applications-of-minimum-spanning-tree/)
3. [Prim’s MST for Adjacency List Representation](http://www.geeksforgeeks.org/greedy-algorithms-set-5-prims-mst-for-adjacency-list-representation/)
4. [Kruskal’s Minimum Spanning Tree Algorithm](http://www.geeksforgeeks.org/greedy-algorithms-set-2-kruskals-minimum-spanning-tree-mst/)
5. [Boruvka’s algorithm for Minimum Spanning Tree](http://www.geeksforgeeks.org/greedy-algorithms-set-9-boruvkas-algorithm/)

***Shortest Paths:***

1. [Dijkstra’s shortest path algorithm](http://www.geeksforgeeks.org/greedy-algorithms-set-6-dijkstras-shortest-path-algorithm/)
2. [Dijkstra’s Algorithm for Adjacency List Representation](http://www.geeksforgeeks.org/greedy-algorithms-set-7-dijkstras-algorithm-for-adjacency-list-representation/)
3. [Bellman–Ford Algorithm](http://www.geeksforgeeks.org/dynamic-programming-set-23-bellman-ford-algorithm/)
4. [Floyd Warshall Algorithm](http://www.geeksforgeeks.org/dynamic-programming-set-16-floyd-warshall-algorithm/)
5. [Johnson’s algorithm for All-pairs shortest paths](http://www.geeksforgeeks.org/johnsons-algorithm/)
6. [Shortest Path in Directed Acyclic Graph](http://www.geeksforgeeks.org/shortest-path-for-directed-acyclic-graphs/)
7. [Some interesting shortest path questions](http://www.geeksforgeeks.org/interesting-shortest-path-questions-set-1/)
8. [Shortest path with exactly k edges in a directed and weighted graph](http://www.geeksforgeeks.org/shortest-path-exactly-k-edges-directed-weighted-graph/)

***Connectivity:***

1. [Find if there is a path between two vertices in a directed graph](http://www.geeksforgeeks.org/find-if-there-is-a-path-between-two-vertices-in-a-given-graph/)
2. [Connectivity in a directed graph](http://www.geeksforgeeks.org/connectivity-in-a-directed-graph/)
3. [Articulation Points (or Cut Vertices) in a Graph](http://www.geeksforgeeks.org/articulation-points-or-cut-vertices-in-a-graph/)
4. [Biconnected graph](http://www.geeksforgeeks.org/biconnectivity-in-a-graph/)
5. [Bridges in a graph](http://www.geeksforgeeks.org/bridge-in-a-graph/)
6. [Eulerian path and circuit](http://www.geeksforgeeks.org/eulerian-path-and-circuit/)
7. [Fleury’s Algorithm for printing Eulerian Path or Circuit](http://www.geeksforgeeks.org/fleurys-algorithm-for-printing-eulerian-path/)
8. [Strongly Connected Components](http://www.geeksforgeeks.org/strongly-connected-components/)
9. [Transitive closure of a graph](http://www.geeksforgeeks.org/transitive-closure-of-a-graph/)
10. [Find the number of islands](http://www.geeksforgeeks.org/find-number-of-islands/)
11. [Count all possible walks from a source to a destination with exactly k edges](http://www.geeksforgeeks.org/count-possible-paths-source-destination-exactly-k-edges/)
12. [Euler Circuit in a Directed Graph](http://www.geeksforgeeks.org/euler-circuit-directed-graph/)
13. [Biconnected Components](http://www.geeksforgeeks.org/biconnected-components/)
14. [Tarjan’s Algorithm to find Strongly Connected Components](http://www.geeksforgeeks.org/tarjan-algorithm-find-strongly-connected-components/)

***Hard Problems:***

1. [Graph Coloring (Introduction and Applications)](http://www.geeksforgeeks.org/graph-coloring-applications/)
2. [Greedy Algorithm for Graph Coloring](http://www.geeksforgeeks.org/graph-coloring-set-2-greedy-algorithm/)
3. [Travelling Salesman Problem (Naive and Dynamic Programming)](http://www.geeksforgeeks.org/travelling-salesman-problem-set-1/)
4. [Travelling Salesman Problem (Approximate using MST)](http://www.geeksforgeeks.org/travelling-salesman-problem-set-2-approximate-using-mst/)
5. [Hamiltonian Cycle](http://www.geeksforgeeks.org/backtracking-set-7-hamiltonian-cycle/)
6. [Vertex Cover Problem (Introduction and Approximate Algorithm)](http://www.geeksforgeeks.org/vertex-cover-problem-set-1-introduction-approximate-algorithm-2/)
7. [K Centers Problem (Greedy Approximate Algorithm)](http://www.geeksforgeeks.org/k-centers-problem-set-1-greedy-approximate-algorithm/)

***Maximum Flow:***

1. [Ford-Fulkerson Algorithm for Maximum Flow Problem](http://www.geeksforgeeks.org/ford-fulkerson-algorithm-for-maximum-flow-problem/)
2. [Find maximum number of edge disjoint paths between two vertices](http://www.geeksforgeeks.org/find-edge-disjoint-paths-two-vertices/)
3. [Find minimum s-t cut in a flow network](http://www.geeksforgeeks.org/minimum-cut-in-a-directed-graph/)
4. [Maximum Bipartite Matching](http://www.geeksforgeeks.org/maximum-bipartite-matching/)
5. [Channel Assignment Problem](http://www.geeksforgeeks.org/channel-assignment-problem/)

**Misc:**

1. [Find if the strings can be chained to form a circle](http://www.geeksforgeeks.org/given-array-strings-find-strings-can-chained-form-circle/)
2. [Given a sorted dictionary of an alien language, find order of characters](http://www.geeksforgeeks.org/given-sorted-dictionary-find-precedence-characters/)
3. [Karger’s algorithm for Minimum Cut](http://www.geeksforgeeks.org/kargers-algorithm-for-minimum-cut-set-1-introduction-and-implementation/)
4. [Karger’s algorithm for Minimum Cut | Set 2 (Analysis and Applications)](http://www.geeksforgeeks.org/kargers-algorithm-for-minimum-cut-set-2-analysis-and-applications/)
5. [Hopcroft–Karp Algorithm for Maximum Matching | Set 1 (Introduction)](http://www.geeksforgeeks.org/hopcroft-karp-algorithm-for-maximum-matching-set-1-introduction/)
6. [Hopcroft–Karp Algorithm for Maximum Matching | Set 2 (Implementation)](http://www.geeksforgeeks.org/hopcroft-karp-algorithm-for-maximum-matching-set-1-introduction/)
7. [Length of shortest chain to reach a target word](http://www.geeksforgeeks.org/length-of-shortest-chain-to-reach-a-target-word/)
8. [Find same contacts in a list of contacts](http://www.geeksforgeeks.org/find-same-contacts-in-a-list-of-contacts/)

[**All Algorithms on Graph**](http://www.geeksforgeeks.org/graph-data-structure-and-algorithms/)  
[Quiz on Graph](http://geeksquiz.com/data-structure/graph/)  
[Quiz on Graph Traversals](http://geeksquiz.com/algorithms/graph-traversals/)  
[Quiz on Graph Shortest Paths](http://geeksquiz.com/algorithms/graph-shortest-paths/)  
[Quiz on Graph Minimum Spanning Tree](http://geeksquiz.com/algorithms/graph-minimum-spanning-tree/)  
[Coding Practice on Graph](http://www.practice.geeksforgeeks.org/tag-page.php?tag=graph&isCmp=0)

**Randomized Algorithms:**

1. [Linearity of Expectation](http://www.geeksforgeeks.org/linearity-of-expectation/)
2. [Expected Number of Trials until Success](http://www.geeksforgeeks.org/expected-number-of-trials-before-success/)
3. [Randomized Algorithms | Set 0 (Mathematical Background)](http://www.geeksforgeeks.org/randomized-algorithms-set-0-mathematical-background/)
4. [Randomized Algorithms | Set 1 (Introduction and Analysis)](http://www.geeksforgeeks.org/randomized-algorithms-set-1-introduction-and-analysis/)
5. [Randomized Algorithms | Set 2 (Classification and Applications)](http://www.geeksforgeeks.org/randomized-algorithms-set-2-classification-and-applications/)
6. [Randomized Algorithms | Set 3 (1/2 Approximate Median)](http://www.geeksforgeeks.org/randomized-algorithms-set-3-12-approximate-median/)
7. [Karger’s algorithm for Minimum Cut](http://www.geeksforgeeks.org/kargers-algorithm-for-minimum-cut-set-1-introduction-and-implementation/)
8. [K’th Smallest/Largest Element in Unsorted Array | Set 2 (Expected Linear Time)](http://www.geeksforgeeks.org/kth-smallestlargest-element-unsorted-array-set-2-expected-linear-time/)
9. [Reservoir Sampling](http://www.geeksforgeeks.org/reservoir-sampling/)
10. [Shuffle a given array](http://www.geeksforgeeks.org/shuffle-a-given-array/)
11. [Select a Random Node from a Singly Linked List](http://www.geeksforgeeks.org/select-a-random-node-from-a-singly-linked-list/)

[**Recent Articles on Randomized Algorithms**](http://www.geeksforgeeks.org/category/algorithm/randomized/)

**Branch and Bound**:

1. [Branch and Bound | Set 1 (Introduction with 0/1 Knapsack)](http://www.geeksforgeeks.org/branch-and-bound-set-1-introduction-with-01-knapsack/)
2. [Branch and Bound | Set 2 (Implementation of 0/1 Knapsack)](http://www.geeksforgeeks.org/branch-and-bound-set-2-implementation-of-01-knapsack/)
3. [Branch and Bound | Set 3 (8 puzzle Problem)](http://www.geeksforgeeks.org/branch-bound-set-3-8-puzzle-problem/)
4. [Branch And Bound | Set 4 (Job Assignment Problem)](http://www.geeksforgeeks.org/branch-bound-set-4-job-assignment-problem/)
5. [Branch and Bound | Set 5 (N Queen Problem)](http://www.geeksforgeeks.org/branch-and-bound-set-4-n-queen-problem/)
6. [Branch And Bound | Set 6 (Traveling Salesman Problem)](http://www.geeksforgeeks.org/branch-bound-set-5-traveling-salesman-problem/)

[**Recent Articles on Branch and Bound**](http://www.geeksforgeeks.org/category/algorithm/branch-and-bound/)

**Quizzes on Algorithms:**

1. [Analysis of Algorithms](http://geeksquiz.com/algorithms/analysis-of-algorithms/)
2. [Sorting](http://geeksquiz.com/algorithms/searching-and-sorting/)
3. [Divide and Conquer](http://geeksquiz.com/algorithms/divide-and-conquer/)
4. [Greedy Algorithms](http://geeksquiz.com/algorithms/greedy-algorithms/)
5. [Dynamic Programming](http://geeksquiz.com/algorithms/dynamic-programming/)
6. [Backtracking](http://geeksquiz.com/algorithms/backtracking/)
7. [Misc](http://geeksquiz.com/algorithms/misc-2/)
8. [NP Complete](http://geeksquiz.com/algorithms/np-complete/)
9. [Searching](http://geeksquiz.com/algorithms/searching/)
10. [Analysis of Algorithms (Recurrences)](http://geeksquiz.com/algorithms/analysis-of-algorithms-recurrences/)
11. [Recursion](http://geeksquiz.com/algorithms/recursion/)
12. [Bit Algorithms](http://geeksquiz.com/algorithms/bit-algorithms/)
13. [Graph Traversals](http://geeksquiz.com/algorithms/graph-traversals/)
14. [Graph Shortest Paths](http://geeksquiz.com/algorithms/graph-shortest-paths/)
15. [Graph Minimum Spanning Tree](http://geeksquiz.com/algorithms/graph-minimum-spanning-tree/)

**Misc:**

1. [Commonly Asked Algorithm Interview Questions | Set 1](http://geeksquiz.com/commonly-asked-algorithm-interview-questions-set-1/)
2. [Given a matrix of ‘O’ and ‘X’, find the largest subsquare surrounded by ‘X’](http://www.geeksforgeeks.org/given-matrix-o-x-find-largest-subsquare-surrounded-x/)
3. [Nuts & Bolts Problem (Lock & Key problem)](http://www.geeksforgeeks.org/nuts-bolts-problem-lock-key-problem/)
4. [Flood fill Algorithm – how to implement fill() in paint?](http://www.geeksforgeeks.org/flood-fill-algorithm-implement-fill-paint/)
5. [Given n appointments, find all conflicting appointments](http://www.geeksforgeeks.org/given-n-appointments-find-conflicting-appointments/)
6. [Check a given sentence for a given set of simple grammer rules](http://www.geeksforgeeks.org/check-given-sentence-given-set-simple-grammer-rules/)
7. [Find Index of 0 to be replaced with 1 to get longest continuous sequence of 1s in a binary array](http://www.geeksforgeeks.org/find-index-0-replaced-1-get-longest-continuous-sequence-1s-binary-array/)
8. [How to check if two given sets are disjoint?](http://www.geeksforgeeks.org/check-two-given-sets-disjoint/)
9. [Minimum Number of Platforms Required for a Railway/Bus Station](http://www.geeksforgeeks.org/minimum-number-platforms-required-railwaybus-station/)
10. [Length of the largest subarray with contiguous elements | Set 1](http://www.geeksforgeeks.org/length-largest-subarray-contiguous-elements-set-1/)
11. [Length of the largest subarray with contiguous elements | Set 2](http://www.geeksforgeeks.org/length-largest-subarray-contiguous-elements-set-2/)
12. [Print all increasing sequences of length k from first n natural numbers](http://www.geeksforgeeks.org/print-increasing-sequences-length-k-first-n-natural-numbers/)
13. [Given two strings, find if first string is a subsequence of second](http://www.geeksforgeeks.org/given-two-strings-find-first-string-subsequence-second/)
14. [Snake and Ladder Problem](http://www.geeksforgeeks.org/snake-ladder-problem-2/)
15. [Write a function that returns 2 for input 1 and returns 1 for 2](http://geeksquiz.com/write-function-returns-2-input-1-returns-1-2/)
16. [Connect n ropes with minimum cost](http://www.geeksforgeeks.org/connect-n-ropes-minimum-cost/)
17. [Find the number of valid parentheses expressions of given length](http://geeksquiz.com/find-number-valid-parentheses-expressions-given-length/)
18. [Longest Monotonically Increasing Subsequence Size (N log N): Simple implementation](http://geeksquiz.com/longest-monotonically-increasing-subsequence-size-n-log-n-simple-implementation/)
19. [Generate all binary permutations such that there are more 1’s than 0’s at every point in all permutations](http://geeksquiz.com/generate-binary-permutations-1s-0s-every-point-permutations/)
20. [Lexicographically minimum string rotation](http://geeksquiz.com/lexicographically-minimum-string-rotation/)
21. [Construct an array from its pair-sum array](http://geeksquiz.com/construct-array-pair-sum-array/)
22. [Program to evaluate simple expressions](http://geeksquiz.com/program-evaluate-simple-expressions/)
23. [Check if characters of a given string can be rearranged to form a palindrome](http://geeksquiz.com/check-characters-given-string-can-rearranged-form-palindrome/)
24. [Print all pairs of anagrams in a given array of strings](http://geeksquiz.com/print-pairs-anagrams-given-array-strings/)

Please see [Data Structures and Advanced Data Structures](http://www.geeksforgeeks.org/data-structures/) for Graph, Binary Tree, BST and Linked List based algorithms.

We will be adding more categories and posts to this page soon.

You can create a new Algorithm topic and discuss it with other geeks using[**Geeksforgeeks Q&A**](http://qa.geeksforgeeks.org/index.php?qa=ask) page. See already discussed [**Algorithm questions on forum**](http://qa.geeksforgeeks.org/index.php?qa=tag&qa_1=algorithms).

Analysis of Algorithms | Set 1 (Asymptotic Analysis)

***Why performance analysis?***  
There are many important things that should be taken care of, like user friendliness, modularity, security, maintainability, etc. Why to worry about performance?   
The answer to this is simple, we can have all the above things only if we have performance. So performance is like currency through which we can buy all the above things. Another reason for studying performance is – speed is fun!

***Given two algorithms for a task, how do we find out which one is better?***  
One naive way of doing this is – implement both the algorithms and run the two programs on your computer for different inputs and see which one takes less time. There are many problems with this approach for analysis of algorithms.  
1) It might be possible that for some inputs, first algorithm performs better than the second. And for some inputs second performs better.  
2) It might also be possible that for some inputs, first algorithm perform better on one machine and the second works better on other machine for some other inputs.

[Asymptotic Analysis](http://en.wikipedia.org/wiki/Asymptotic_analysis) is the big idea that handles above issues in analyzing algorithms. In Asymptotic Analysis, we evaluate the performance of an algorithm in terms of input size (we don’t measure the actual running time). We calculate, how does the time (or space) taken by an algorithm increases with the input size.  
For example, let us consider the search problem (searching a given item) in a sorted array. One way to search is Linear Search (order of growth is linear) and other way is Binary Search (order of growth is logarithmic). To understand how Asymptotic Analysis solves the above mentioned problems in analyzing algorithms, let us say we run the Linear Search on a fast computer and Binary Search on a slow computer. For small values of input array size n, the fast computer may take less time. But, after certain value of input array size, the Binary Search will definitely start taking less time compared to the Linear Search even though the Binary Search is being run on a slow machine. The reason is the order of growth of Binary Search with respect to input size logarithmic while the order of growth of Linear Search is linear. So the machine dependent constants can always be ignored after certain values of input size.

***Does Asymptotic Analysis always work?***  
Asymptotic Analysis is not perfect, but that’s the best way available for analyzing algorithms. For example, say there are two sorting algorithms that take 1000nLogn and 2nLogn time respectively on a machine. Both of these algorithms are asymptotically same (order of growth is nLogn). So, With Asymptotic Analysis, we can’t judge which one is better as we ignore constants in Asymptotic Analysis.  
Also, in Asymptotic analysis, we always talk about input sizes larger than a constant value. It might be possible that those large inputs are never given to your software and an algorithm which is asymptotically slower, always performs better for your particular situation. So, you may end up choosing an algorithm that is Asymptotically slower but faster for your software.

Next – [Analysis of Algorithms | Set 2 (Worst, Average and Best Cases)](http://www.geeksforgeeks.org/analysis-of-algorithms-set-2-asymptotic-analysis/)

**References:**  
[MIT’s Video lecture 1 on Introduction to Algorithms](http://www.youtube.com/watch?v=JPyuH4qXLZ0).

Analysis of Algorithms | Set 2 (Worst, Average and Best Cases)

In the [previous post](http://www.geeksforgeeks.org/archives/11064), we discussed how Asymptotic analysis overcomes the problems of naive way of analyzing algorithms. In this post, we will take an example of Linear Search and analyze it using Asymptotic analysis.

We can have three cases to analyze an algorithm:  
1) Worst Case  
2) Average Case  
3) Best Case

Let us consider the following implementation of Linear Search.

|  |
| --- |
| #include <stdio.h>    // Linearly search x in arr[].  If x is present then return the index,  // otherwise return -1  int search(int arr[], int n, int x)  {      int i;      for (i=0; i<n; i++)      {         if (arr[i] == x)           return i;      }      return -1;  }    /\* Driver program to test above functions\*/  int main()  {      int arr[] = {1, 10, 30, 15};      int x = 30;      int n = sizeof(arr)/sizeof(arr[0]);      printf("%d is present at index %d", x, search(arr, n, x));        getchar();      return 0;  } |

Run on IDE

**Worst Case Analysis (Usually Done)**  
In the worst case analysis, we calculate upper bound on running time of an algorithm. We must know the case that causes maximum number of operations to be executed. For Linear Search, the worst case happens when the element to be searched (x in the above code) is not present in the array. When x is not present, the search() functions compares it with all the elements of arr[] one by one. Therefore, the worst case time complexity of linear search would be Θ(n).

**Average Case Analysis (Sometimes done)**  
In average case analysis, we take all possible inputs and calculate computing time for all of the inputs. Sum all the calculated values and divide the sum by total number of inputs. We must know (or predict) distribution of cases. For the linear search problem, let us assume that all cases are [uniformly distributed](http://en.wikipedia.org/wiki/Uniform_distribution_%28discrete%29) (including the case of x not being present in array). So we sum all the cases and divide the sum by (n+1). Following is the value of average case time complexity.

Average Case Time = [analysis1](http://d1hyf4ir1gqw6c.cloudfront.net/wp-content/uploads/analysis1.png)

= [analysis2](http://d1hyf4ir1gqw6c.cloudfront.net/wp-content/uploads/analysis2.png)

= Θ(n)

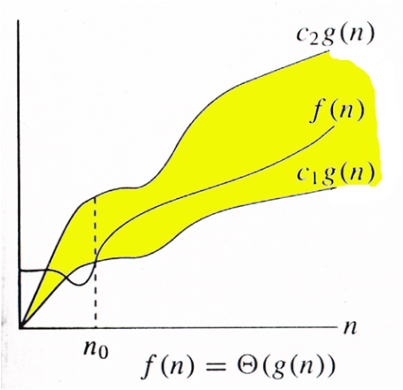
**Best Case Analysis (Bogus)**  
In the best case analysis, we calculate lower bound on running time of an algorithm. We must know the case that causes minimum number of operations to be executed. In the linear search problem, the best case occurs when x is present at the first location. The number of operations in the best case is constant (not dependent on n). So time complexity in the best case would be Θ(1)  
Most of the times, we do worst case analysis to analyze algorithms. In the worst analysis, we guarantee an upper bound on the running time of an algorithm which is good information.  
The average case analysis is not easy to do in most of the practical cases and it is rarely done. In the average case analysis, we must know (or predict) the mathematical distribution of all possible inputs.  
The Best Case analysis is bogus. Guaranteeing a lower bound on an algorithm doesn’t provide any information as in the worst case, an algorithm may take years to run.

For some algorithms, all the cases are asymptotically same, i.e., there are no worst and best cases. For example,[Merge Sort](http://en.wikipedia.org/wiki/Merge_sort). Merge Sort does Θ(nLogn) operations in all cases. Most of the other sorting algorithms have worst and best cases. For example, in the typical implementation of Quick Sort (where pivot is chosen as a corner element), the worst occurs when the input array is already sorted and the best occur when the pivot elements always divide array in two halves. For insertion sort, the worst case occurs when the array is reverse sorted and the best case occurs when the array is sorted in the same order as output.

Next – [Analysis of Algorithms | Set 3 (Asymptotic Notations)](http://www.geeksforgeeks.org/analysis-of-algorithms-set-3asymptotic-notations/)

# Analysis of Algorithms | Set 3 (Asymptotic Notations)

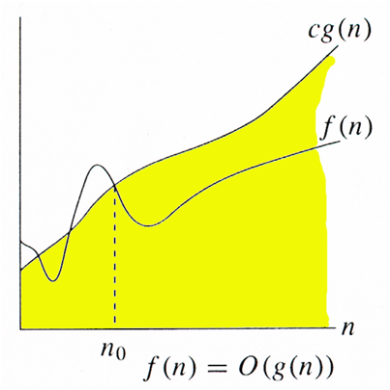
We have discussed [Asymptotic Analysis](http://www.geeksforgeeks.org/analysis-of-algorithms-set-1-asymptotic-analysis/), and[Worst, Average and Best Cases of Algorithms](http://www.geeksforgeeks.org/analysis-of-algorithms-set-2-asymptotic-analysis/). The main idea of asymptotic analysis is to have a measure of efficiency of algorithms that doesn’t depend on machine specific constants, and doesn’t require algorithms to be implemented and time taken by programs to be compared. Asymptotic notations are mathematical tools to represent time complexity of algorithms for asymptotic analysis. The following 3 asymptotic notations are mostly used to represent time complexity of algorithms.

[](http://d1hyf4ir1gqw6c.cloudfront.net/wp-content/uploads/thetanotation.png)**1) Θ Notation:** The theta notation bounds a functions from above and below, so it defines exact asymptotic behavior.  
A simple way to get Theta notation of an expression is to drop low order terms and ignore leading constants. For example, consider the following expression.  
3n3 + 6n2 + 6000 = Θ(n3)  
Dropping lower order terms is always fine because there will always be a n0 after which Θ(n3) has higher values than Θn2) irrespective of the constants involved.  
For a given function g(n), we denote Θ(g(n)) is following set of functions.

Θ(g(n)) = {f(n): there exist positive constants c1, c2 and n0 such

that 0 <= c1\*g(n) <= f(n) <= c2\*g(n) for all n >= n0}

The above definition means, if f(n) is theta of g(n), then the value f(n) is always between c1\*g(n) and c2\*g(n) for large values of n (n >= n0). The definition of theta also requires that f(n) must be non-negative for values of n greater than n0.

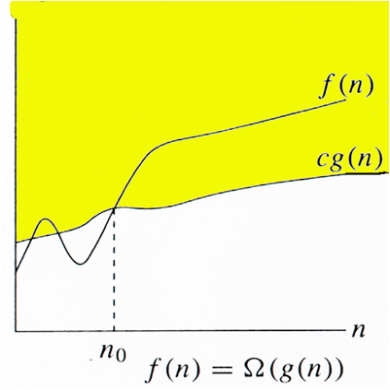
[](http://d1hyf4ir1gqw6c.cloudfront.net/wp-content/uploads/BigO.png)**2) Big O Notation:** The Big O notation defines an upper bound of an algorithm, it bounds a function only from above. For example, consider the case of Insertion Sort. It takes linear time in best case and quadratic time in worst case. We can safely say that the time complexity of Insertion sort is O(n^2). Note that O(n^2) also covers linear time.  
If we use Θ notation to represent time complexity of Insertion sort, we have to use two statements for best and worst cases:  
1. The worst case time complexity of Insertion Sort is Θ(n^2).  
2. The best case time complexity of Insertion Sort is Θ(n).

The Big O notation is useful when we only have upper bound on time complexity of an algorithm. Many times we easily find an upper bound by simply looking at the algorithm.

O(g(n)) = { f(n): there exist positive constants c and

n0 such that 0 <= f(n) <= cg(n) for

all n >= n0}

[](http://d1hyf4ir1gqw6c.cloudfront.net/wp-content/uploads/BigOmega.png)**3) Ω Notation:** Just as Big O notation provides an asymptotic upper bound on a function, Ω notation provides an asymptotic lower bound.

Ω Notation< can be useful when we have lower bound on time complexity of an algorithm. As discussed in the previous post, the [best case performance of an algorithm is generally not useful](http://www.geeksforgeeks.org/analysis-of-algorithms-set-2-asymptotic-analysis/), the Omega notation is the least used notation among all three.

For a given function g(n), we denote by Ω(g(n)) the set of functions.

Ω (g(n)) = {f(n): there exist positive constants c and

n0 such that 0 <= cg(n) <= f(n) for

all n >= n0}.

Let us consider the same Insertion sort example here. The time complexity of Insertion Sort can be written as Ω(n), but it is not a very useful information about insertion sort, as we are generally interested in worst case and sometimes in average case.

**Exercise:**  
Which of the following statements is/are valid?  
**1.** Time Complexity of QuickSort is Θ(n^2)  
**2.** Time Complexity of QuickSort is O(n^2)  
**3.** For any two functions f(n) and g(n), we have f(n) = Θ(g(n)) if and only if f(n) = O(g(n)) and f(n) = Ω(g(n)).  
**4.**Time complexity of all computer algorithms can be written as Ω(1)

Next –[Analysis of Algorithms | Set 4 (Analysis of Loops)](http://www.geeksforgeeks.org/analysis-of-algorithms-set-4-analysis-of-loops/)

# Analysis of Algorithms | Set 4 (Analysis of Loops)

We have discussed [Asymptotic Analysis](http://www.geeksforgeeks.org/analysis-of-algorithms-set-1-asymptotic-analysis/),  [Worst, Average and Best Cases](http://www.geeksforgeeks.org/analysis-of-algorithms-set-2-asymptotic-analysis/) and [Asymptotic Notations](http://www.geeksforgeeks.org/analysis-of-algorithms-set-3asymptotic-notations/) in previous posts. In this post, analysis of iterative programs with simple examples is discussed.

**1) O(1):**Time complexity of a function (or set of statements) is considered as O(1) if it doesn’t contain loop, recursion and call to any other non-constant time function.

// set of non-recursive and non-loop statements

For example [swap() function](http://geeksquiz.com/c-program-swap-two-numbers/) has O(1) time complexity.  
A loop or recursion that runs a constant number of times is also considered as O(1). For example the following loop is O(1).

// Here c is a constant

for (int i = 1; i <= c; i++) {

// some O(1) expressions

}

**2) O(n):** Time Complexity of a loop is considered as O(n) if the loop variables is incremented / decremented by a constant amount. For example following functions have O(n) time complexity.

// Here c is a positive integer constant

for (int i = 1; i <= n; i += c) {

// some O(1) expressions

}

for (int i = n; i > 0; i -= c) {

// some O(1) expressions

}

**3) O(nc)**: Time complexity of nested loops is equal to the number of times the innermost statement is executed. For example the following sample loops have O(n2) time complexity

for (int i = 1; i <=n; i += c) {

for (int j = 1; j <=n; j += c) {

// some O(1) expressions

}

}

for (int i = n; i > 0; i += c) {

for (int j = i+1; j <=n; j += c) {

// some O(1) expressions

}

For example [Selection sort](http://geeksquiz.com/selection-sort/) and [Insertion Sort](http://geeksquiz.com/insertion-sort/) have O(n2) time complexity.  
**4) O(Logn)** Time Complexity of a loop is considered as O(Logn) if the loop variables is divided / multiplied by a constant amount.

for (int i = 1; i <=n; i \*= c) {

// some O(1) expressions

}

for (int i = n; i > 0; i /= c) {

// some O(1) expressions

}

For example [Binary Search(refer iterative implementation)](http://geeksquiz.com/binary-search/) has O(Logn) time complexity.  
**5) O(LogLogn)** Time Complexity of a loop is considered as O(LogLogn) if the loop variables is reduced / increased exponentially by a constant amount.

// Here c is a constant greater than 1

for (int i = 2; i <=n; i = pow(i, c)) {

// some O(1) expressions

}

//Here fun is sqrt or cuberoot or any other constant root

for (int i = n; i > 0; i = fun(i)) {

// some O(1) expressions

}

See [this](http://www.geeksforgeeks.org/time-complexity-loop-loop-variable-expands-shrinks-exponentially/)for more explanation.  
**How to combine time complexities of consecutive loops?**  
When there are consecutive loops, we calculate time complexity as sum of time complexities of individual loops.

for (int i = 1; i <=m; i += c) {

// some O(1) expressions

}

for (int i = 1; i <=n; i += c) {

// some O(1) expressions

}

Time complexity of above code is O(m) + O(n) which is O(m+n)

If m == n, the time complexity becomes O(2n) which is O(n).

**How to calculate time complexity when there are many if, else statements inside loops?**  
As discussed [here](http://www.geeksforgeeks.org/analysis-of-algorithms-set-2-asymptotic-analysis/), worst case time complexity is the most useful among best, average and worst. Therefore we need to consider worst case. We evaluate the situation when values in if-else conditions cause maximum number of statements to be executed.  
For example consider the [linear search function](http://www.geeksforgeeks.org/analysis-of-algorithms-set-2-asymptotic-analysis/) where we consider the case when element is present at the end or not present at all.  
When the code is too complex to consider all if-else cases, we can get an upper bound by ignoring if else and other complex control statements.  
**How to calculate time complexity of recursive functions?**  
Time complexity of a recursive function can be written as a mathematical recurrence relation. To calculate time complexity, we must know how to solve recurrences. We will soon be discussing recurrence solving techniques as a separate post.

[Quiz on Analysis of Algorithms](http://geeksquiz.com/algorithms/analysis-of-algorithms/)

Next – [Analysis of Algorithm | Set 4 (Solving Recurrences)](http://www.geeksforgeeks.org/analysis-algorithm-set-4-master-method-solving-recurrences/)

# Analysis of Algorithm | Set 4 (Solving Recurrences)

In the previous post, we discussed [analysis of loops](http://www.geeksforgeeks.org/analysis-of-algorithms-set-4-analysis-of-loops/). Many algorithms are recursive in nature. When we analyze them, we get a recurrence relation for time complexity. We get running time on an input of size n as a function of n and the running time on inputs of smaller sizes. For example in [Merge Sort](http://geeksquiz.com/merge-sort/), to sort a given array, we divide it in two halves and recursively repeat the process for the two halves. Finally we merge the results. Time complexity of Merge Sort can be written as T(n) = 2T(n/2) + cn. There are many other algorithms like Binary Search, Tower of Hanoi, etc.

There are mainly three ways for solving recurrences.

**1) Substitution Method**: We make a guess for the solution and then we use mathematical induction to prove the the guess is correct or incorrect.

For example consider the recurrence T(n) = 2T(n/2) + n

We guess the solution as T(n) = O(nLogn). Now we use induction

to prove our guess.

We need to prove that T(n) <= cnLogn. We can assume that it is true

for values smaller than n.

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

<= cn/2Log(n/2) + n

= cnLogn - cnLog2 + n

= cnLogn - cn + n

<= cnLogn

**2) Recurrence Tree Method:** In this method, we draw a recurrence tree and calculate the time taken by every level of tree. Finally, we sum the work done at all levels. To draw the recurrence tree, we start from the given recurrence and keep drawing till we find a pattern among levels. The pattern is typically a arithmetic or geometric series.

For example consider the recurrence relation

T(n) = T(n/4) + T(n/2) + cn2

cn2

/ \

T(n/4) T(n/2)

If we further break down the expression T(n/4) and T(n/2),

we get following recursion tree.

cn2

/ \

c(n2)/16 c(n2)/4

/ \ / \

T(n/16) T(n/8) T(n/8) T(n/4)

Breaking down further gives us following

cn2

/ \

c(n2)/16 c(n2)/4

/ \ / \

c(n2)/256 c(n2)/64 c(n2)/64 c(n2)/16

/ \ / \ / \ / \

To know the value of T(n), we need to calculate sum of tree

nodes level by level. If we sum the above tree level by level,

we get the following series

T(n) = c(n^2 + 5(n^2)/16 + 25(n^2)/256) + ....

The above series is geometrical progression with ratio 5/16.

To get an upper bound, we can sum the infinite series.

We get the sum as (n2)/(1 - 5/16) which is O(n2)

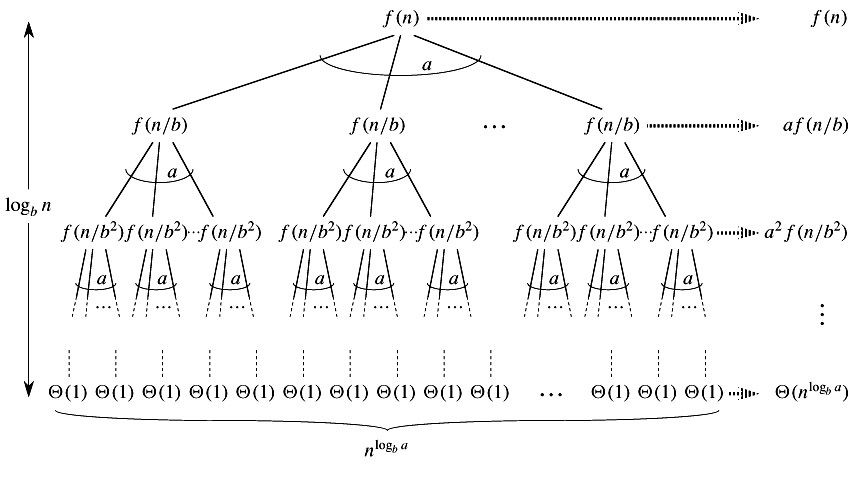
**3) Master Method:**  
Master Method is a direct way to get the solution. The master method works only for following type of recurrences or for recurrences that can be transformed to following type.

T(n) = aT(n/b) + f(n) where a >= 1 and b > 1

There are following three cases:  
**1.** If f(n) = Θ(nc) where c < Logba then T(n) = Θ(nLogba)

**2.** If f(n) = Θ(nc) where c = Logba then T(n) = Θ(ncLog n)

**3.**If f(n) = Θ(nc) where c > Logba then T(n) = Θ(f(n))

**How does this work?**  
Master method is mainly derived from recurrence tree method. If we draw recurrence tree of T(n) = aT(n/b) + f(n), we can see that the work done at root is f(n) and work done at all leaves is Θ(nc) where c is Logba. And the height of recurrence tree is Logbn  
[](http://d1hyf4ir1gqw6c.cloudfront.net/wp-content/uploads/Master-Theorem.jpg)  
In recurrence tree method, we calculate total work done. If the work done at leaves is polynomially more, then leaves are the dominant part, and our result becomes the work done at leaves (Case 1). If work done at leaves and root is asymptotically same, then our result becomes height multiplied by work done at any level (Case 2). If work done at root is asymptotically more, then our result becomes work done at root (Case 3).

**Examples of some standard algorithms whose time complexity can be evaluated using Master Method**  
[Merge Sort](http://geeksquiz.com/merge-sort/): T(n) = 2T(n/2) + Θ(n). It falls in case 2 as c is 1 and Logba] is also 1. So the solution is Θ(n Logn)

[Binary Search](http://geeksquiz.com/binary-search/): T(n) = T(n/2) + Θ(1). It also falls in case 2 as c is 0 and Logba is also 0. So the solution is Θ(Logn)

**Notes:**  
**1)** It is not necessary that a recurrence of the form T(n) = aT(n/b) + f(n) can be solved using Master Theorem. The given three cases have some gaps between them. For example, the recurrence T(n) = 2T(n/2) + n/Logn cannot be solved using master method.

**2)** Case 2 can be extended for f(n) = Θ(ncLogkn)  
If f(n) = Θ(ncLogkn) for some constant k >= 0 and c = Logba, then T(n) = Θ(ncLogk+1n)

[Practice Problems and Solutions on Master Theorem.](http://www.csd.uwo.ca/~moreno/CS424/Ressources/master.pdf)

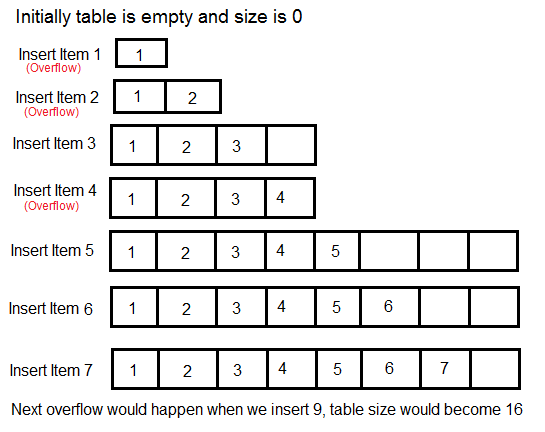
Next – [Analysis of Algorithm | Set 5 (Amortized Analysis Introduction)](http://www.geeksforgeeks.org/analysis-algorithm-set-5-amortized-analysis-introduction/)

**References:**  
<http://en.wikipedia.org/wiki/Master_theorem>  
[MIT Video Lecture on Asymptotic Notation | Recurrences | Substitution, Master Method](http://www.youtube.com/watch?v=whjt_N9uYFI)  
[Introduction to Algorithms 3rd Edition by Clifford Stein, Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest](http://www.flipkart.com/introduction-algorithms-3/p/itmczynzhyhxv2gs?pid=9788120340077&affid=sandeepgfg)

# Analysis of Algorithm | Set 5 (Amortized Analysis Introduction)

[Amortized Analysis](http://en.wikipedia.org/wiki/Amortized_analysis) is used for algorithms where an occasional operation is very slow, but most of the other operations are faster. In Amortized Analysis, we analyze a sequence of operations and guarantee a worst case average time which is lower than the worst case time of a particular expensive operation.  
The example data structures whose operations are analyzed using Amortized Analysis are Hash Tables, Disjoint Sets and Splay Trees.

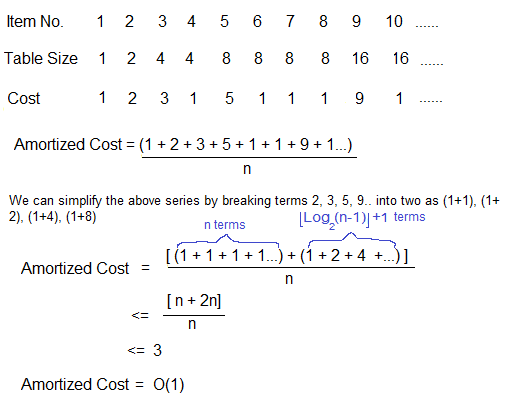
Let us consider an example of a simple hash table insertions. How do we decide table size? There is a trade-off between space and time, if we make hash-table size big, search time becomes fast, but space required becomes high.

[](http://d1hyf4ir1gqw6c.cloudfront.net/wp-content/uploads/Dynamic-Table.png)

The solution to this trade-off problem is to use [Dynamic Table (or Arrays)](http://en.wikipedia.org/wiki/Dynamic_array). The idea is to increase size of table whenever it becomes full. Following are the steps to follow when table becomes full.  
1) Allocate memory for a larger table of size, typically twice the old table.  
2) Copy the contents of old table to new table.  
3) Free the old table.

If the table has space available, we simply insert new item in available space.

**What is the time complexity of n insertions using the above scheme?**  
If we use simple analysis, the worst case cost of an insertion is O(n). Therefore, worst case cost of n inserts is n \* O(n) which is O(n2). This analysis gives an upper bound, but not a tight upper bound for n insertions as all insertions don’t take Θ(n) time.

[](http://d1hyf4ir1gqw6c.cloudfront.net/wp-content/uploads/AmortizedAnalysis.png)

So using Amortized Analysis, we could prove that the Dynamic Table scheme has O(1) insertion time which is a great result used in hashing. Also, the concept of dynamic table is used in [vectors in C++,](http://www.cplusplus.com/reference/vector/vector/) [ArrayList in Java](http://docs.oracle.com/javase/7/docs/api/java/util/ArrayList.html).

Following are few important notes.  
**1)**Amortized cost of a sequence of operations can be seen as expenses of a salaried person. The average monthly expense of the person is less than or equal to the salary, but the person can spend more money in a particular month by buying a car or something. In other months, he or she saves money for the expensive month.

**2)** The above Amortized Analysis done for Dynamic Array example is called ***Aggregate Method***. There are two more powerful ways to do Amortized analysis called [***Accounting Method***](http://en.wikipedia.org/wiki/Accounting_method)and [***Potential Method***](http://en.wikipedia.org/wiki/Potential_method). We will be discussing the other two methods in separate posts.

**3)**The amortized analysis doesn’t involve probability. There is also another different notion of average case running time where algorithms use randomization to make them faster and expected running time is faster than the worst case running time. These algorithms are analyzed using Randomized Analysis. Examples of these algorithms are Randomized Quick Sort, Quick Select and Hashing. We will soon be covering Randomized analysis in a different post.

**Sources:**  
[Berkeley Lecture 35: Amortized Analysis](https://www.youtube.com/watch?v=UYcWpldlX-o) [MIT Lecture 13: Amortized Algorithms, Table Doubling, Potential Method](https://www.youtube.com/watch?v=b733mo4CxAQ)  
<http://www.cs.cornell.edu/courses/cs3110/2011sp/lectures/lec20-amortized/amortized.htm>

# What does ‘Space Complexity’ mean?

**Space Complexity:**  
The term Space Complexity is misused for Auxiliary Space at many places. Following are the correct definitions of Auxiliary Space and Space Complexity.

Auxiliary Space is the extra space or temporary space used by an algorithm.

Space Complexityof an algorithm is total space taken by the algorithm with respect to the input size. Space complexity includes both Auxiliary space and space used by input.

For example, if we want to compare standard sorting algorithms on the basis of space, then Auxiliary Space would be a better criteria than Space Complexity. Merge Sort uses O(n) auxiliary space, Insertion sort and Heap Sort use O(1) auxiliary space. Space complexity of all these sorting algorithms is O(n) though.

# Pseudo-polynomial Algorithms

**What is Pseudo-polynomial?**  
An algorithm whose worst case time complexity depends on numeric value of input (not number of inputs) is called Pseudo-polynomial algorithm.  
For example, consider the problem of counting frequencies of all elements in an array of positive numbers. A pseudo-polynomial time solution for this is to first find the maximum value, then iterate from 1 to maximum value and for each value, find its frequency in array. This solution requires time according to maximum value in input array, therefore pseudo-polynomial. On the other hand, an algorithm whose time complexity is only based on number of elements in array (not value) is considered as polynomial time algorithm.

**Pseudo-polynomial and NP-Completeness**  
Some NP-Complete problems have Pseudo Polynomial time solutions. For example, Dynamic Programming Solutions of [0-1 Knapsack](http://www.geeksforgeeks.org/dynamic-programming-set-10-0-1-knapsack-problem/), [Subset-Sum](http://www.geeksforgeeks.org/dynamic-programming-subset-sum-problem/) and [Partition](http://www.geeksforgeeks.org/dynamic-programming-set-18-partition-problem/) problems are Pseudo-Polynomial. NP complete problems that can be solved using a pseudo-polynomial time algorithms are called weakly NP-complete.

**Reference:**  
<https://en.wikipedia.org/wiki/Pseudo-polynomial_time>

This article is contributed by **Dheeraj Gupta**. If you like GeeksforGeeks and would like to contribute, you can also write an article and mail your article to contribute@geeksforgeeks.org. See your article appearing on the GeeksforGeeks main page and help other Geeks.

# NP-Completeness | Set 1 (Introduction)

We have been writing about efficient algorithms to solve complex problems, like [shortest path](http://www.geeksforgeeks.org/greedy-algorithms-set-6-dijkstras-shortest-path-algorithm/), [Euler graph](http://www.geeksforgeeks.org/eulerian-path-and-circuit/), [minimum spanning tree](http://www.geeksforgeeks.org/greedy-algorithms-set-2-kruskals-minimum-spanning-tree-mst/), etc. Those were all success stories of algorithm designers. In this post, failure stories of computer science are discussed.

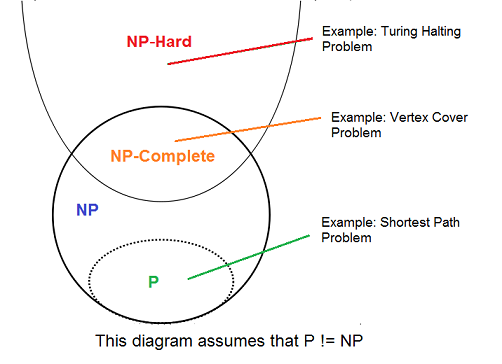
**Can all computational problems be solved by a computer?** There are computational problems that can not be solved by algorithms even with unlimited time. For example Turing Halting problem (Given a program and an input, whether the program will eventually halt when run with that input, or will run forever). Alan Turing proved that general algorithm to solve the halting problem for all possible program-input pairs cannot exist. A key part of the proof is, Turing machine was used as a mathematical definition of a computer and program (Source [Halting Problem](http://en.wikipedia.org/wiki/Halting_problem)).  
Status of NP Complete problems is another failure story, NP complete problems are problems whose status is unknown. No polynomial time algorithm has yet been discovered for any NP complete problem, nor has anybody yet been able to prove that no polynomial-time algorithm exist for any of them. The interesting part is, if any one of the NP complete problems can be solved in polynomial time, then all of them can be solved.

**What are NP, P, NP-complete and NP-Hard problems?**  
P is set of problems that can be solved by a deterministic Turing machine in **P**olynomial time.

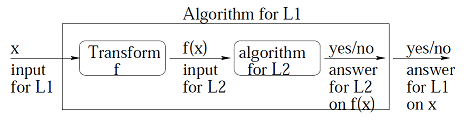
NP is set of decision problems that can be solved by a **N**on-deterministic Turing Machine in **P**olynomial time. P is subset of NP (any problem that can be solved by deterministic machine in polynomial time can also be solved by non-deterministic machine in polynomial time).  
Informally, NP is set of decision problems which can be solved by a polynomial time via a “Lucky Algorithm”, a magical algorithm that always makes a right guess among the given set of choices (Source [Ref 1](http://www.youtube.com/watch?v=moPtwq_cVH8)).

NP-complete problems are the hardest problems in NP set.  A decision problem L is NP-complete if:  
**1)** L is in NP (Any given solution for NP-complete problems can be verified quickly, but there is no efficient known solution).  
**2)**Every problem in NP is reducible to L in polynomial time (Reduction is defined below).

A problem is NP-Hard if it follows property 2 mentioned above, doesn’t need to follow property 1. Therefore, NP-Complete set is also a subset of NP-Hard set.

[](http://d1hyf4ir1gqw6c.cloudfront.net/wp-content/uploads/NP-CompleteSet.png)

**Decision vs Optimization Problems**  
NP-completeness applies to the realm of decision problems.  It was set up this way because it’s easier to compare the difficulty of decision problems than that of optimization problems.   In reality, though, being able to solve a decision problem in polynomial time will often permit us to solve the corresponding optimization problem in polynomial time (using a polynomial number of calls to the decision problem). So, discussing the difficulty of decision problems is often really equivalent to discussing the difficulty of optimization problems. (Source [Ref 2](http://uic.edu.hk/~taochen/teaching/comp3040/week13/l17.pdf)).  
For example, consider the [vertex cover problem](http://en.wikipedia.org/wiki/Vertex_cover) (Given a graph, find out the minimum sized vertex set that covers all edges). It is an optimization problem. Corresponding decision problem is, given undirected graph G and k, is there a vertex cover of size k?

**What is Reduction?**  
Let L1 and L2be two decision problems. Suppose algorithm A2solves L2. That is, if y is an input for L2 then algorithm A2will answer Yes or No depending upon whether y belongs toL2 or not.  
The idea is to find a transformation from L1 to L2 so that the algorithm A2 can be part of an algorithm A1 to solve L1.  
[](http://d1hyf4ir1gqw6c.cloudfront.net/wp-content/uploads/reduction.png)  
Learning reduction in general is very important. For example, if we have library functions to solve certain problem and if we can reduce a new problem to one of the solved problems, we save a lot of time. Consider the example of a problem where we have to find minimum product path in a given directed graph where product of path is multiplication of weights of edges along the path. If we have code for Dijkstra’s algorithm to find shortest path, we can take log of all weights and use Dijkstra’s algorithm to find the minimum product path rather than writing a fresh code for this new problem.

**How to prove that a given problem is NP complete?**  
From the definition of NP-complete, it appears impossible to prove that a problem L is NP-Complete.  By definition, it requires us to that show every problem in NP is polynomial time reducible to L.   Fortunately, there is an alternate way to prove it.   The idea is to take a known NP-Complete problem and reduce it to L.  If polynomial time reduction is possible, we can prove that L is NP-Complete by transitivity of reduction (If a NP-Complete problem is reducible to L in polynomial time, then all problems are reducible to L in polynomial time).

**What was the first problem proved as NP-Complete?**  
There must be some first NP-Complete problem proved by definition of NP-Complete problems.  [SAT (Boolean satisfiability problem)](http://en.wikipedia.org/wiki/Boolean_satisfiability_problem)is the first NP-Complete problem proved by Cook (See CLRS book for proof).

It is always useful to know about NP-Completeness even for engineers. Suppose you are asked to write an efficient algorithm to solve an extremely important problem for your company. After a lot of thinking, you can only come up exponential time approach which is impractical. If you don’t know about NP-Completeness, you can only say that I could not come with an efficient algorithm. If you know about NP-Completeness and prove that the problem as NP-complete, you can proudly say that the polynomial time solution is unlikely to exist. If there is a polynomial time solution possible, then that solution solves a big problem of computer science many scientists have been trying for years.

We will soon be discussing more NP-Complete problems and their proof for NP-Completeness.

**References:**  
[MIT Video Lecture on Computational Complexity](http://www.youtube.com/watch?v=moPtwq_cVH8)  
[Introduction to Algorithms 3rd Edition by Clifford Stein, Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest](http://www.flipkart.com/introduction-algorithms-3/p/itmczynzhyhxv2gs?pid=9788120340077&affid=sandeepgfg)  
<http://www.ics.uci.edu/~eppstein/161/960312.html>

Polynomial Time Approximation Scheme

It is a very well know fact that there is no known polynomial time solution for [NP Complete problems](http://www.geeksforgeeks.org/np-completeness-set-1/) and these problems occur a lot in real world (See [this](http://www.geeksforgeeks.org/graph-coloring-applications/), [this](http://www.geeksforgeeks.org/k-centers-problem-set-1-greedy-approximate-algorithm/) and [this](http://www.geeksforgeeks.org/set-cover-problem-set-1-greedy-approximate-algorithm/) for example). So there must be a way to handle them. We have seen algorithms to these problems which are p approximate (For example [2 approximate for Travelling Salesman](http://www.geeksforgeeks.org/travelling-salesman-problem-set-2-approximate-using-mst/)). Can we do better?

**P**olynomial **T**ime **A**pproximation **S**cheme (PTAS) is a type of approximate algorithms that provide user to control over accuracy which is a desirable feature. These algorithms take an additional parameter ε > 0 and provide a solution that is (1 + ε) approximate for minimization and (1 – ε) for maximization. For example consider a minimization problem, if ε is 0.5, then the solution provided by the PTAS algorithm is 1.5 approximate. The running time of PTAS must be polynomial in terms of n, however it can be exponential in terms of ε

In PTAS algorithms, the exponent of the polynomial can increase dramatically as ε reduces, for example if the runtime is O(n(1/ε)!) which is a problem. There is a stricter scheme, **F**ully **P**olynomial **T**ime **A**pproximation **S**cheme (FPTAS). In FPTAS, algorithm need to polynomial in both the problem size n and 1/ε.

**Example (0-1 knapsack problem):**  
      We know that [0-1 knapsack](http://www.geeksforgeeks.org/dynamic-programming-set-10-0-1-knapsack-problem/) is NP Complete. There is a DP based [pseudo polynomial solution](http://www.geeksforgeeks.org/pseudo-polynomial-in-algorithms/) for this. But if input values are high, then the solution becomes infeasible and there is a need of approximate solution. One approximate solution is to use Greedy Approach (compute value per kg for all items and put the highest value per kg first if it is smaller than W), but Greedy approach is not PTAS, so we don’t have control over accuracy.

Below is a FPTAS solution for 0-1 Knapsack problem:  
Input:  
**W** (Capacity of Knapsack)  
**val[0..n-1]** (Values of Items)  
**wt[0..n-1]** (Weights of Items)

1. Find the maximum valued item, i.e., find maximum value in val[]. Let this maximum value be maxVal.
2. Compute adjustment factor k for all values

k = (maxVal \* ε) / n

1. Adjust all values, i.e., create a new array val'[] that values divided by k. Do following for every value val[i].

val'[i] = floor(val[i] / k)

1. Run [DP based solution](http://www.geeksforgeeks.org/dynamic-programming-set-10-0-1-knapsack-problem/) for reduced values, i,e, val'[0..n-1] and all other parameter same.

The above solution works in polynomial time in terms of both n and ε. The solution provided by this FPTAS is (1 – ε) approximate. The idea is to rounds off some of the least significant digits of values then they will be bounded by a polynomial and 1/ε.

Example:

val[] = {12, 16, 4, 8}

wt[] = {3, 4, 5, 2}

W = 10

ε = 0.5

maxVal = 20 [maximum value in val[]]

Adjustment factor, k = (16 \* 0.5)/4 = 2.0

Now we apply DP based solution on below modified

instance of problem.

val'[] = {6, 8, 2. 4} [ val'[i] = floor(val[i]/k) ]

wt[] = {3, 4, 5, 2}

W = 10

**How is the solution (1 – ε) \* OPT?**  
Here **OPT** is the optimal value. Let **S** be the set produced by above FPTAS algorithm and total value of S be val(S). We need to show that

val(S) >= (1 - ε)\*OPT

Let **O**be the set produced by optimal solution (the solution with total value OPT), i.e., val(O) = OPT.

val(O) - k\*val'(O) <= n\*k

[Because val'[i] = floor(val[i]/k) ]

After the dynamic programming step, we get a set that is optimal for the scaled instance  
and therefore must be at least as good as choosing the set S with the smaller profits. From that, we can conclude,

val'(S) >= k . val'(O)

>= val(O) - nk

>= OPT - ε \* maxVal

>= OPT - ε \* OPT [OPT >= maxVal]

>= (1 - ε) \* OPT

**Sources:**  
<http://math.mit.edu/~goemans/18434S06/knapsack-katherine.pdf>  
<https://en.wikipedia.org/wiki/Polynomial-time_approximation_scheme>

This article is contributed by Dheeraj Gupta. If you like GeeksforGeeks and would like to contribute, you can also write an article and mail your article to contribute@geeksforgeeks.org. See your article appearing on the GeeksforGeeks main page and help other Geeks.

A Time Complexity Question

What is the time complexity of following function fun()? Assume that log(x) returns log value in base 2.

|  |
| --- |
| void fun()  {     int i, j;     for (i=1; i<=n; i++)        for (j=1; j<=log(i); j++)           printf("GeeksforGeeks");  } |

Time Complexity of the above function can be written as Θ(log 1) + Θ(log 2) + Θ(log 3) + . . . . + Θ(log n) which is Θ (log n!)

Order of growth of ‘log n!’ and ‘n log n’ is same for large values of n, i.e., Θ (log n!) = Θ(n log n). So time complexity of fun() is Θ(n log n).

The expression Θ(log n!) = Θ(n log n) can be easily derived from following [Stirling’s approximation (or Stirling’s formula)](http://en.wikipedia.org/wiki/Stirling%27s_approximation).

log n! = n log n - n + O(log(n))

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

Sources:  
<http://en.wikipedia.org/wiki/Stirling%27s_approximation>

# Time Complexity of building a heap

Consider the following algorithm for building a Heap of an input array A.

BUILD-HEAP(A)

heapsize := size(A);

for i := floor(heapsize/2) downto 1

do HEAPIFY(A, i);

end for

END

What is the worst case time complexity of the above algo?  
Although the worst case complexity looks like O(nLogn), upper bound of time complexity is O(n). See following links for the proof of time complexity.

http://www.cse.iitk.ac.in/users/sbaswana/Courses/ESO211/heap.pdf/  
<http://www.cs.sfu.ca/CourseCentral/307/petra/2009/SLN_2.pdf>

Time Complexity where loop variable is incremented by 1, 2, 3, 4 ..

What is the time complexity of below code?

|  |
| --- |
| void fun(int n)  {     int j = 1, i = 0;     while (i < n)     {         // Some O(1) task         i = i + j;         j++;     }  } |

The loop variable ‘i’ is incremented by 1, 2, 3, 4, … until i becomes greater than or equal to n.

The value of i is x(x+1)/2 after x iterations. So if loop runs x times, then x(x+1)/2 < n. Therefore time complexity can be written as Θ(√n).

# Time Complexity of Loop with Powers

What is the time complexity of below function?

void fun(int n, int k)

{

for (int i=1; i<=n; i++)

{

int p = pow(i, k);

for (int j=1; j<=p; j++)

{

// Some O(1) work

}

}

}

Time complexity of above function can be written as 1k + 2k + 3k + … n1k.

Let us try few examples:

k=1

Sum = 1 + 2 + 3 ... n

= n(n+1)/2

= n2 + n/2

k=2

Sum = 12 + 22 + 32 + ... n12.

= n(n+1)(2n+1)/6

= n3/3 + n2/2 + n/6

k=3

Sum = 13 + 23 + 33 + ... n13.

= n2(n+1)2/4

= n4/4 + n3/2 + n2/4

In general, asymptotic value can be written as **(nk+1)/(k+1) + Θ(nk)**

Note that, in asymptotic notations like Θ we can always ignore lower order terms. So the time complexity is **Θ(nk+1 / (k+1))**

# Performance of loops (A caching question)

Consider below two C language functions to compute sum of elements in a 2D array. Ignoring the compiler optimizations, which of the two is better implementation of sum?

// Function 1

int fun1(int arr[R][C])

{

int sum = 0;

for (int i=0; i<R; i++)

for (int j=0; j<C; j++)

sum += arr[i][j];

}

// Function 2

int fun2(int arr[R][C])

{

int sum = 0;

for (int j=0; j<C; j++)

for (int i=0; i<R; i++)

sum += arr[i][j];

}

In C/C++, elements are stored in Row-Major order. So the first implementation has better spatial locality (nearby memory locations are referenced in successive iterations). Therefore, first implementation should always be preferred for iterating multidimensional arrays.

**Searching and Sorting**:

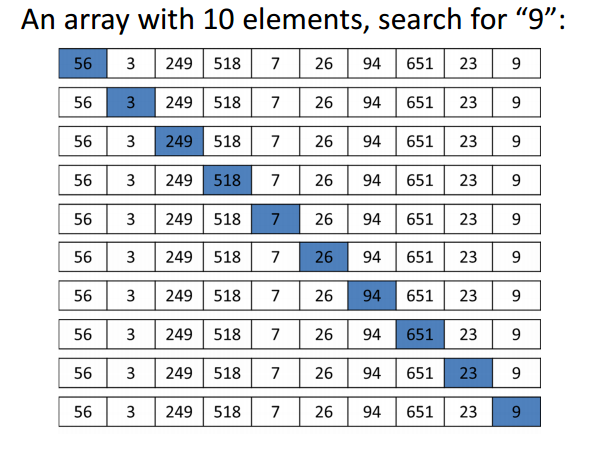
1. [Linear Search](http://quiz.geeksforgeeks.org/linear-search/), [Binary Search](http://geeksquiz.com/binary-search/), [Jump Search](http://www.geeksforgeeks.org/jump-search/), [Interpolation Search](http://www.geeksforgeeks.org/interpolation-search/)

Linear Search

**Problem:** Given an array arr[] of n elements, write a function to search a given element x in arr[].

A simple approach is to do **linear search**, i.e

* Start from the leftmost element of arr[] and one by one compare x with each element of arr[]
* If x matches with an element, return the index.
* If x doesn’t match with any of elements, return -1.

**Example:**[](http://quiz.geeksforgeeks.org/wp-content/uploads/2016/10/linear-search1.png)

* C/C++
* Python
* Java

|  |
| --- |
| // Linearly search x in arr[].  If x is present then return its  // location,  otherwise return -1  int search(int arr[], int n, int x)  {      int i;      for (i=0; i<n; i++)          if (arr[i] == x)           return i;      return -1;  } |

Run on IDE

The time complexity of above algorithm is O(n).

Linear search is rarely used practically because other search algorithms such as the binary search algorithm and hash tables allow significantly faster searching comparison to Linear search.

Also See – [Binary Search](http://quiz.geeksforgeeks.org/binary-search/)

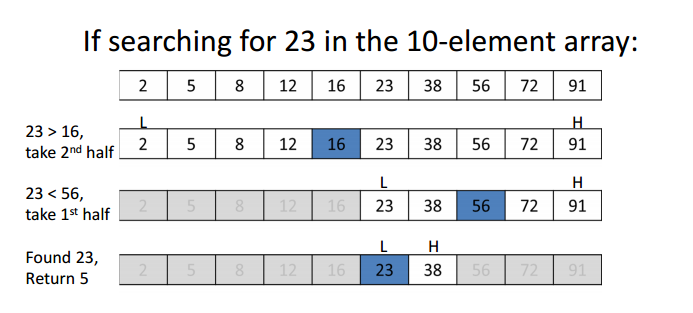
Image Source : <http://www.nyckidd.com/bob/Linear%20Search%20and%20Binary%20Search_WorkingCopy.pdf>

# Binary Search

Given a sorted array arr[] of n elements, write a function to search a given element x in arr[].

A simple approach is to do [**linear search**](http://quiz.geeksforgeeks.org/linear-search/)**.**The time complexity of above algorithm is O(n). Another approach to perform the same task is using Binary Search.

**Binary Search:** Search a sorted array by repeatedly dividing the search interval in half. Begin with an interval covering the whole array. If the value of the search key is less than the item in the middle of the interval, narrow the interval to the lower half. Otherwise narrow it to the upper half. Repeatedly check until the value is found or the interval is empty.

Example:[](http://quiz.geeksforgeeks.org/wp-content/uploads/2014/01/binary-search1.png)  
Image Source : <http://www.nyckidd.com/bob/Linear%20Search%20and%20Binary%20Search_WorkingCopy.pdf>

The idea of binary search is to use the information that the array is sorted and reduce the time complexity to O(Logn).

## [We strongly recommend that you click here and practice it, before moving on to the solution.](http://www.practice.geeksforgeeks.org/problem-page.php?pid=700238)

We basically ignore half of the elements just after one comparison.

1. Compare x with the middle element.
2. If x matches with middle element, we return the mid index.
3. Else If x is greater than the mid element, then x can only lie in right half subarray after the mid element. So we recur for right half.
4. Else (x is smaller) recur for the left half.

**Recursive**implementation of Binary Search

* C/C++
* Python
* Java

|  |
| --- |
| #include <stdio.h>    // A recursive binary search function. It returns location of x in  // given array arr[l..r] is present, otherwise -1  int binarySearch(int arr[], int l, int r, int x)  {     if (r >= l)     {          int mid = l + (r - l)/2;            // If the element is present at the middle itself          if (arr[mid] == x)  return mid;            // If element is smaller than mid, then it can only be present          // in left subarray          if (arr[mid] > x) return binarySearch(arr, l, mid-1, x);            // Else the element can only be present in right subarray          return binarySearch(arr, mid+1, r, x);     }       // We reach here when element is not present in array     return -1;  }    int main(void)  {     int arr[] = {2, 3, 4, 10, 40};     int n = sizeof(arr)/ sizeof(arr[0]);     int x = 10;     int result = binarySearch(arr, 0, n-1, x);     (result == -1)? printf("Element is not present in array")                   : printf("Element is present at index %d", result);     return 0;  } |

Run on IDE

Output:

Element is present at index 3

**Iterative**implementation of Binary Search

* C/C++
* Python
* Java

|  |
| --- |
| #include <stdio.h>    // A iterative binary search function. It returns location of x in  // given array arr[l..r] if present, otherwise -1  int binarySearch(int arr[], int l, int r, int x)  {    while (l <= r)    {      int m = l + (r-l)/2;        // Check if x is present at mid      if (arr[m] == x)          return m;        // If x greater, ignore left half      if (arr[m] < x)          l = m + 1;        // If x is smaller, ignore right half      else           r = m - 1;    }      // if we reach here, then element was not present    return -1;  }    int main(void)  {     int arr[] = {2, 3, 4, 10, 40};     int n = sizeof(arr)/ sizeof(arr[0]);     int x = 10;     int result = binarySearch(arr, 0, n-1, x);     (result == -1)? printf("Element is not present in array")                   : printf("Element is present at index %d", result);     return 0;  } |

Run on IDE

Output:

Element is present at index 3

**Time Complexity:**  
The time complexity of Binary Search can be written as

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

The above recurrence can be solved either using Recurrence T ree method or Master method. It falls in case II of Master Method and solution of the recurrence is .

**Auxiliary Space:** O(1) in case of iterative implementation. In case of recursive implementation, O(Logn) recursion call stack space.

Jump Search

Like [Binary Search](http://geeksquiz.com/binary-search/), Jump Search is a searching algorithm for sorted arrays. The basic idea is to check fewer elements (than [linear search](http://www.geeksforgeeks.org/analysis-of-algorithms-set-2-asymptotic-analysis/)) by jumping ahead by fixed steps or skipping some elements in place of searching all elements.

For example, suppose we have an array arr[] of size n and block (to be jumped) size m. Then we search at the indexes arr[0], arr[m], arr[2m]…..arr[km] and so on. Once we find the interval (arr[km] < x < arr[(k+1)m]), we perform a linear search operation from the index km to find the element x.

Let’s consider the following array: (0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610). Length of the array is 16. Jump search will find the value of 55 with the following steps assuming that the block size to be jumped is 4.  
STEP 1: Jump from index 0 to index 4;  
STEP 2: Jump from index 4 to index 8;  
STEP 3: Jump from index 8 to index 16;  
STEP 4: Since the element at index 16 is greater than 55 we will jump back a step to come to index 9.  
STEP 5: Perform linear search from index 9 to get the element 55.

**What is the optimal block size to be skipped?**  
In the worst case, we have to do n/m jumps and if the last checked value is greater than the element to be searched for, we perform m-1 comparisons more for linear search. Therefore the total number of comparisons in the worst case will be ((n/m) + m-1). The value of the function ((n/m) + m-1) will be minimum when m = √n. Therefore, the best step size is m = **√n.**  
 **Java Implementation :**

|  |
| --- |
| // Java program to implement Jump Search.  public class JumpSearch  {      public static int jumpSearch(int[] arr, int x)      {          int n = arr.length;            // Finding block size to be jumped          int step = (int)Math.floor(Math.sqrt(n));            // Finding the block where element is          // present (if it is present)          int prev = 0;          while (arr[Math.min(step, n)-1] < x)          {              prev = step;              step += (int)Math.floor(Math.sqrt(n));              if (prev >= n)                  return -1;          }            // Doing a linear search for x in block          // beginning with prev.          while (arr[prev] < x)          {              prev++;                // If we reached next block or end of              // array, element is not present.              if (prev == Math.min(step, n))                  return -1;          }            // If element is found          if (arr[prev] == x)              return prev;            return -1;      }        // Driver program to test function      public static void main(String [ ] args)      {          int arr[] = { 0, 1, 1, 2, 3, 5, 8, 13, 21,                       34, 55, 89, 144, 233, 377, 610};          int x = 55;            // Find the index of 'x' using Jump Search          int index = jumpSearch(arr, x);            // Print the index where 'x' is located          System.out.println("\nNumber " + x +                              " is at index " + index);      }  } |

Run on IDE

Output:

Number 55 is at index 10

Time Complexity : O(√n)  
Auxiliary Space : O(1)

**Important points:**

* Works only sorted arrays.
* The optimal size of a block to be jumped is O(√ n). This makes the time complexity of Jump Search O(√ n).
* The time complexity of Jump Search is between Linear Search ( ( O(n) ) and Binary Search ( O (Log n) ).
* Binary Search is better than Jump Search, but Jump search has an advantage that we traverse back only once (Binary Search may require up to O(Log n) jumps, consider a situation where the element to be search is the smallest element or smaller than the smallest). So in a systems where jumping back is costly, we use Jump Search.

**References:**  
<https://en.wikipedia.org/wiki/Jump_search>

This article is contributed by **Harsh Agarwal**. If you like GeeksforGeeks and would like to contribute, you can also write an article using [contribute.geeksforgeeks.org](http://www.contribute.geeksforgeeks.org/) or mail your article to contribute@geeksforgeeks.org. See your article appearing on the GeeksforGeeks main page and help other Geeks.Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

Interpolation Search

Given a sorted array of n uniformly distributed values arr[], write a function to search for a particular element x in the array.

Linear Search finds the element in O(n) time, [Jump Search](http://www.geeksforgeeks.org/jump-search/) takes O(√ n) time and [Binary Search](http://quiz.geeksforgeeks.org/binary-search/) take O(Log n) time.  
The Interpolation Search is an improvement over [Binary Search](http://quiz.geeksforgeeks.org/binary-search/) for instances, where the values in a sorted array are uniformly distributed. Binary Search always goes to middle element to check. On the other hand interpolation search may go to different locations according the value of key being searched. For example if the value of key is closer to the last element, interpolation search is likely to start search toward the end side.

To find the position to be searched, it uses following formula.

// The idea of formula is to return higher value of **pos**

// when element to be searched is closer to **arr[hi]**. And

// smaller value when closer to **arr[lo]**

pos = lo + [ (x-arr[lo])\*(hi-lo) / (arr[hi]-arr[Lo]) ]

arr[] ==> Array where elements need to be searched

x ==> Element to be searched

lo ==> Starting index in arr[]

hi ==> Ending index in arr[]

**Algorithm**  
Rest of the Interpolation algorithm is same except the above partition logic.

**Step1:** In a loop, calculate the value of “pos” using the probe position formula.  
**Step2:** If it is a match, return the index of the item, and exit.  
**Step3:** If the item is less than arr[pos], calculate the probe position of the left sub-array. Otherwise calculate the same in the right sub-array.  
**Step4:** Repeat until a match is found or the sub-array reduces to zero.

Below is C implementation of algorithm.

|  |
| --- |
| // C program to implement interpolation search  #include<stdio.h>    // If x is present in arr[0..n-1], then returns  // index of it, else returns -1.  int interpolationSearch(int arr[], int n, int x)  {      // Find indexes of two corners      int lo = 0, hi = (n - 1);        // Since array is sorted, an element present      // in array must be in range defined by corner      while (lo <= hi && x >= arr[lo] && x <= arr[hi])      {          // Probing the position with keeping          // uniform distribution in mind.          int pos = lo + (((double)(hi-lo) /                (arr[hi]-arr[lo]))\*(x - arr[lo]));            // Condition of target found          if (arr[pos] == x)              return pos;            // If x is larger, x is in upper part          if (arr[pos] < x)              lo = pos + 1;            // If x is smaller, x is in lower part          else              hi = pos - 1;      }      return -1;  }    // Driver Code  int main()  {      // Array of items on which search will      // be conducted.      int arr[] =  {10, 12, 13, 16, 18, 19, 20, 21, 22, 23,                    24, 33, 35, 42, 47};      int n = sizeof(arr)/sizeof(arr[0]);        int x = 18; // Element to be searched      int index = interpolationSearch(arr, n, x);        // If element was found      if (index != -1)          printf("Element found at index %d", index);      else          printf("Element not found.");      return 0;  } |

Output :

Element found at index 4

Time Complexity : If elements are uniformly distributed, then **O (log log n))**. In worst case it can take upto O(n).  
Auxiliary Space : O(1)

This article is contributed by **Aayu sachdev**. If you like GeeksforGeeks and would like to contribute, you can also write an article using [contribute.geeksforgeeks.org](http://www.contribute.geeksforgeeks.org/) or mail your article to contribute@geeksforgeeks.org. See your article appearing on the GeeksforGeeks main page and help other Geeks.

2.[Selection Sort](http://geeksquiz.com/selection-sort/), [Bubble Sort](http://geeksquiz.com/bubble-sort/), [Insertion Sort](http://geeksquiz.com/insertion-sort/), [Merge Sort](http://geeksquiz.com/merge-sort/), [Heap Sort](http://geeksquiz.com/heap-sort/), [QuickSort](http://geeksquiz.com/quick-sort/), [Radix Sort](http://www.geeksforgeeks.org/radix-sort/), [Counting Sort](http://www.geeksforgeeks.org/counting-sort/), [Bucket Sort](http://www.geeksforgeeks.org/bucket-sort-2/), [ShellSort](http://geeksquiz.com/shellsort/), [Comb Sort](http://www.geeksforgeeks.org/comb-sort/), [Pigeonhole Sort](http://www.geeksforgeeks.org/pigeonhole-sort/)

# Selection Sort

The selection sort algorithm sorts an array by repeatedly finding the minimum element (considering ascending order) from unsorted part and putting it at the beginning. The algorithm maintains two subarrays in a given array.

1) The subarray which is already sorted.  
2) Remaining subarray which is unsorted.

In every iteration of selection sort, the minimum element (considering ascending order) from the unsorted subarray is picked and moved to the sorted subarray.

Following example explains the above steps:

arr[] = 64 25 12 22 11

// Find the minimum element in arr[0...4]

// and place it at beginning

**11** 25 12 22 64

// Find the minimum element in arr[1...4]

// and place it at beginning of arr[1...4]

11 **12** 25 22 64

// Find the minimum element in arr[2...4]

// and place it at beginning of arr[2...4]

11 12 **22** 25 64

// Find the minimum element in arr[3...4]

// and place it at beginning of arr[3...4]

11 12 22 **25** 64

## [We strongly recommend that you click here and practice it, before moving on to the solution.](http://www.practice.geeksforgeeks.org/probfunc-page.php?pid=700147)

* C/C++
* Python
* Java

|  |
| --- |
| // C program for implementation of selection sort  #include <stdio.h>    void swap(int \*xp, int \*yp)  {      int temp = \*xp;      \*xp = \*yp;      \*yp = temp;  }    void selectionSort(int arr[], int n)  {      int i, j, min\_idx;        // One by one move boundary of unsorted subarray      for (i = 0; i < n-1; i++)      {          // Find the minimum element in unsorted array          min\_idx = i;          for (j = i+1; j < n; j++)            if (arr[j] < arr[min\_idx])              min\_idx = j;            // Swap the found minimum element with the first element          swap(&arr[min\_idx], &arr[i]);      }  }    /\* Function to print an array \*/  void printArray(int arr[], int size)  {      int i;      for (i=0; i < size; i++)          printf("%d ", arr[i]);      printf("\n");  }    // Driver program to test above functions  int main()  {      int arr[] = {64, 25, 12, 22, 11};      int n = sizeof(arr)/sizeof(arr[0]);      selectionSort(arr, n);      printf("Sorted array: \n");      printArray(arr, n);      return 0;  } |

Run on IDE

Output:

Sorted array:

11 12 22 25 64

**Time Complexity:** O(n2) as there are two nested loops.

**Auxiliary Space:** O(1)

The good thing about selection sort is it never makes more than O(n) swaps and can be useful when memory write is a costly operation.

Bubble Sort

Bubble Sort is the simplest sorting algorithm that works by repeatedly swapping the adjacent elements if they are in wrong order.

**Example:**  
**First Pass:**  
( **5** **1** 4 2 8 ) –> ( **1** **5** 4 2 8 ), Here, algorithm compares the first two elements, and swaps since 5 > 1.  
( 1 **5** **4** 2 8 ) –>  ( 1 **4** **5** 2 8 ), Swap since 5 > 4  
( 1 4 **5** **2** 8 ) –>  ( 1 4 **2** **5** 8 ), Swap since 5 > 2  
( 1 4 2 **5** **8** ) –> ( 1 4 2 **5** **8** ), Now, since these elements are already in order (8 > 5), algorithm does not swap them.

**Second Pass:**  
( **1** **4** 2 5 8 ) –> ( **1** **4** 2 5 8 )  
( 1 **4** **2** 5 8 ) –> ( 1 **2** **4** 5 8 ), Swap since 4 > 2  
( 1 2 **4** **5** 8 ) –> ( 1 2 **4** **5** 8 )  
( 1 2 4 **5** **8** ) –>  ( 1 2 4 **5** **8** )  
Now, the array is already sorted, but our algorithm does not know if it is completed. The algorithm needs one **whole** pass without **any** swap to know it is sorted.

**Third Pass:**  
( **1** **2** 4 5 8 ) –> ( **1** **2** 4 5 8 )  
( 1 **2** **4** 5 8 ) –> ( 1 **2** **4** 5 8 )  
( 1 2 **4** **5** 8 ) –> ( 1 2 **4** **5** 8 )  
( 1 2 4 **5** **8** ) –> ( 1 2 4 **5** **8** )

Following are C/C++, Python and Java implementations of Bubble Sort.

* C/C++
* Java
* Python

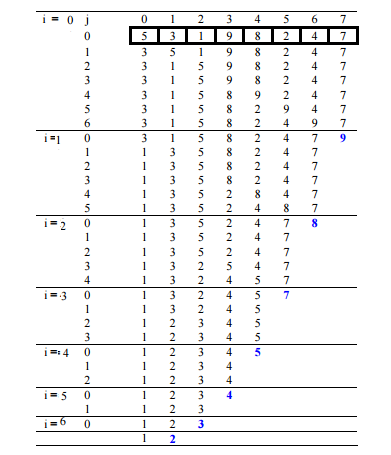
|  |
| --- |
| // C program for implementation of Bubble sort  #include <stdio.h>    void swap(int \*xp, int \*yp)  {      int temp = \*xp;      \*xp = \*yp;      \*yp = temp;  }    // A function to implement bubble sort  void bubbleSort(int arr[], int n)  {     int i, j;     for (i = 0; i < n-1; i++)           // Last i elements are already in place         for (j = 0; j < n-i-1; j++)             if (arr[j] > arr[j+1])                swap(&arr[j], &arr[j+1]);  }    /\* Function to print an array \*/  void printArray(int arr[], int size)  {      for (int i=0; i < size; i++)          printf("%d ", arr[i]);      printf("\n");  }    // Driver program to test above functions  int main()  {      int arr[] = {64, 34, 25, 12, 22, 11, 90};      int n = sizeof(arr)/sizeof(arr[0]);      bubbleSort(arr, n);      printf("Sorted array: \n");      printArray(arr, n);      return 0;  } |

Run on IDE

Output:

Sorted array:

11 12 22 25 34 64 90

**Illustration :**  
[](http://quiz.geeksforgeeks.org/wp-content/uploads/2014/02/bubble-sort1.png)

**Optimized Implementation:**  
The above function always runs O(n^2) time even if the array is sorted. It can be optimized by stopping the algorithm if inner loop didn’t cause any swap.

|  |
| --- |
| // Optimized implementation of Bubble sort  #include <stdio.h>    void swap(int \*xp, int \*yp)  {      int temp = \*xp;      \*xp = \*yp;      \*yp = temp;  }    // An optimized version of Bubble Sort  void bubbleSort(int arr[], int n)  {     int i, j;     bool swapped;     for (i = 0; i < n-1; i++)     {       swapped = false;       for (j = 0; j < n-i-1; j++)       {          if (arr[j] > arr[j+1])          {             swap(&arr[j], &arr[j+1]);             swapped = true;          }       }         // IF no two elements were swapped by inner loop, then break       if (swapped == false)          break;     }  }    /\* Function to print an array \*/  void printArray(int arr[], int size)  {      int i;      for (i=0; i < size; i++)          printf("%d ", arr[i]);      printf("\n");  }    // Driver program to test above functions  int main()  {      int arr[] = {64, 34, 25, 12, 22, 11, 90};      int n = sizeof(arr)/sizeof(arr[0]);      bubbleSort(arr, n);      printf("Sorted array: \n");      printArray(arr, n);      return 0;  } |

Run on IDE

Output:

Sorted array:

11 12 22 25 34 64 90

**Worst and Average Case Time Complexity:**O(n\*n). Worst case occurs when array is reverse sorted.

**Best Case Time Complexity:** O(n). Best case occurs when array is already sorted.

**Auxiliary Space:** O(1)

**Boundary Cases:** Bubble sort takes minimum time (Order of n) when elements are already sorted.

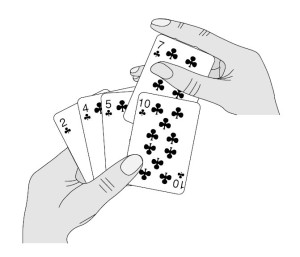
**Sorting In Place:**Yes

**Stable:** Yes

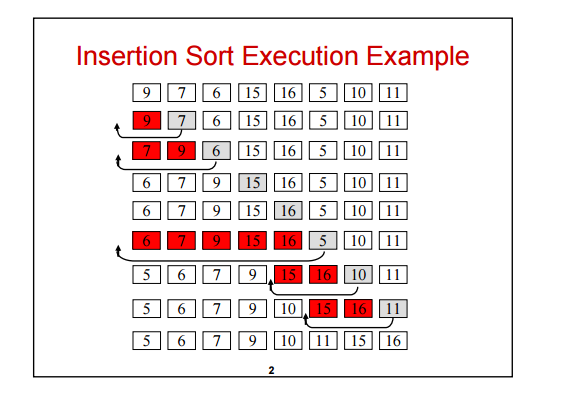
Due to its simplicity, bubble sort is often used to introduce the concept of a sorting algorithm.  
In computer graphics it is popular for its capability to detect a very small error (like swap of just two elements) in almost-sorted arrays and fix it with just linear complexity (2n). For example, it is used in a polygon filling algorithm, where bounding lines are sorted by their x coordinate at a specific scan line (a line parallel to x axis) and with incrementing y their order changes (two elements are swapped) only at intersections of two lines (Source: [Wikipedia](http://en.wikipedia.org/wiki/Bubble_sort#In_practice))

# Insertion Sort

Insertion sort is a simple sorting algorithm that works the way we sort playing cards in our hands.

[](http://quiz.geeksforgeeks.org/wp-content/uploads/2013/03/Insertion-Sort.jpg)

**Algorithm**  
// Sort an arr[] of size n  
insertionSort(arr, n)  
Loop from i = 1 to n-1.  
……a) Pick element arr[i] and insert it into sorted sequence arr[0…i-1]

**Example:**  
[](http://quiz.geeksforgeeks.org/wp-content/uploads/2013/03/insertion-sort.png)  
**Another Example:**  
**12**, 11, 13, 5, 6

Let us loop for i = 1 (second element of the array) to 5 (Size of input array)

i = 1. Since 11 is smaller than 12, move 12 and insert 11 before 12  
**11, 12**, 13, 5, 6

i = 2. 13 will remain at its position as all elements in A[0..I-1] are smaller than 13  
**11, 12, 13**, 5, 6

i = 3. 5 will move to the beginning and all other elements from 11 to 13 will move one position ahead of their current position.  
**5, 11, 12, 13**, 6

i = 4. 6 will move to position after 5, and elements from 11 to 13 will move one position ahead of their current position.  
**5, 6, 11, 12, 13**

## [We strongly recommend that you click here and practice it, before moving on to the solution.](http://www.practice.geeksforgeeks.org/probfunc-page.php?pid=700148)

* C/C++
* Python
* Java

|  |
| --- |
| // C program for insertion sort  #include <stdio.h>  #include <math.h>    /\* Function to sort an array using insertion sort\*/  void insertionSort(int arr[], int n)  {     int i, key, j;     for (i = 1; i < n; i++)     {         key = arr[i];         j = i-1;           /\* Move elements of arr[0..i-1], that are            greater than key, to one position ahead            of their current position \*/         while (j >= 0 && arr[j] > key)         {             arr[j+1] = arr[j];             j = j-1;         }         arr[j+1] = key;     }  }    // A utility function ot print an array of size n  void printArray(int arr[], int n)  {     int i;     for (i=0; i < n; i++)         printf("%d ", arr[i]);     printf("\n");  }        /\* Driver program to test insertion sort \*/  int main()  {      int arr[] = {12, 11, 13, 5, 6};      int n = sizeof(arr)/sizeof(arr[0]);        insertionSort(arr, n);      printArray(arr, n);        return 0;  } |

Run on IDE

Output:

5 6 11 12 13

**Time Complexity:** O(n\*n)

**Auxiliary Space:**O(1)

**Boundary Cases**: Insertion sort takes maximum time to sort if elements are sorted in reverse order. And it takes minimum time (Order of n) when elements are already sorted.

**Algorithmic Paradigm:** Incremental Approach

**Sorting In Place:** Yes

**Stable:** Yes

**Online:** Yes

**Uses:** Insertion sort is used when number of elements is small. It can also be useful when input array is almost sorted, only few elements are misplaced in complete big array.

**What is Binary Insertion Sort?**  
We can use binary search to reduce the number of comparisons in normal insertion sort. Binary Insertion Sort find use binary search to find the proper location to insert the selected item at each iteration. In normal insertion, sort it takes O(i) (at ith iteration) in worst case. we can reduce it to O(logi) by using binary search. The algorithm as a whole still has a running worst case running time of O(n2) because of the series of swaps required for each insertion. Refer [this](http://quiz.geeksforgeeks.org/binary-insertion-sort/) for implementation.

**How to implement Insertion Sort for Linked List?**  
Below is simple insertion sort algorithm for linked list.

1) Create an empty sorted (or result) list

2) Traverse the given list, do following for every node.

......a) Insert current node in sorted way in sorted or result list.

3) Change head of given linked list to head of sorted (or result) list.

# Merge Sort

Like [QuickSort](http://quiz.geeksforgeeks.org/quick-sort/), Merge Sort is a [Divide and Conquer](http://www.geeksforgeeks.org/divide-and-conquer-set-1-find-closest-pair-of-points/) algorithm. It divides input array in two halves, calls itself for the two halves and then merges the two sorted halves. **The merge() function** is used for merging two halves. The merge(arr, l, m, r) is key process that assumes that arr[l..m] and arr[m+1..r] are sorted and merges the two sorted sub-arrays into one. See following C implementation for details.

**MergeSort(arr[], l, r)**

If r > l

**1.** Find the middle point to divide the array into two halves:

middle m = (l+r)/2

**2.** Call mergeSort for first half:

Call mergeSort(arr, l, m)

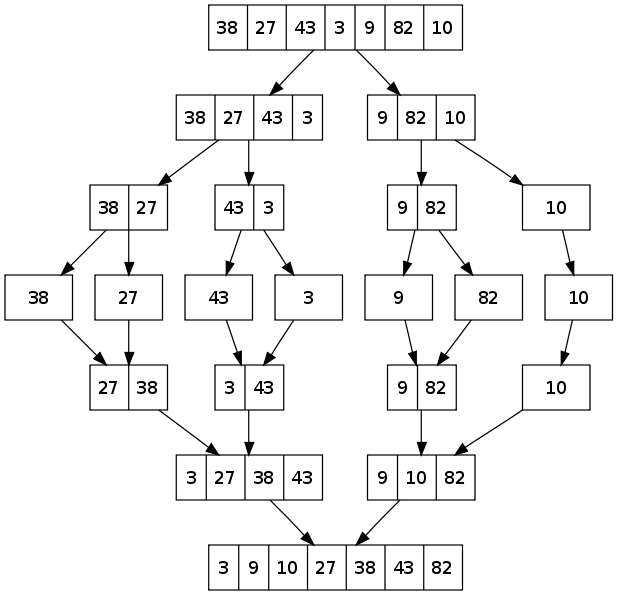
**3.** Call mergeSort for second half:

Call mergeSort(arr, m+1, r)

**4.** Merge the two halves sorted in step 2 and 3:

Call merge(arr, l, m, r)

The following diagram from [wikipedia](http://en.wikipedia.org/wiki/File:Merge_sort_algorithm_diagram.svg) shows the complete merge sort process for an example array {38, 27, 43, 3, 9, 82, 10}. If we take a closer look at the diagram, we can see that the array is recursively divided in two halves till the size becomes 1. Once the size becomes 1, the merge processes comes into action and starts merging arrays back till the complete array is merged.

[](http://quiz.geeksforgeeks.org/wp-content/uploads/2013/03/Merge-Sort.png)

## [We strongly recommend that you click here and practice it, before moving on to the solution.](http://www.practice.geeksforgeeks.org/probfunc-page.php?pid=700150)

* C/C++
* Java
* Python

|  |
| --- |
| /\* C program for Merge Sort \*/  #include<stdlib.h>  #include<stdio.h>    // Merges two subarrays of arr[].  // First subarray is arr[l..m]  // Second subarray is arr[m+1..r]  void merge(int arr[], int l, int m, int r)  {      int i, j, k;      int n1 = m - l + 1;      int n2 =  r - m;        /\* create temp arrays \*/      int L[n1], R[n2];        /\* Copy data to temp arrays L[] and R[] \*/      for (i = 0; i < n1; i++)          L[i] = arr[l + i];      for (j = 0; j < n2; j++)          R[j] = arr[m + 1+ j];        /\* Merge the temp arrays back into arr[l..r]\*/      i = 0; // Initial index of first subarray      j = 0; // Initial index of second subarray      k = l; // Initial index of merged subarray      while (i < n1 && j < n2)      {          if (L[i] <= R[j])          {              arr[k] = L[i];              i++;          }          else          {              arr[k] = R[j];              j++;          }          k++;      }        /\* Copy the remaining elements of L[], if there         are any \*/      while (i < n1)      {          arr[k] = L[i];          i++;          k++;      }        /\* Copy the remaining elements of R[], if there         are any \*/      while (j < n2)      {          arr[k] = R[j];          j++;          k++;      }  }    /\* l is for left index and r is right index of the     sub-array of arr to be sorted \*/  void mergeSort(int arr[], int l, int r)  {      if (l < r)      {          // Same as (l+r)/2, but avoids overflow for          // large l and h          int m = l+(r-l)/2;            // Sort first and second halves          mergeSort(arr, l, m);          mergeSort(arr, m+1, r);            merge(arr, l, m, r);      }  }    /\* UTILITY FUNCTIONS \*/  /\* Function to print an array \*/  void printArray(int A[], int size)  {      int i;      for (i=0; i < size; i++)          printf("%d ", A[i]);      printf("\n");  }    /\* Driver program to test above functions \*/  int main()  {      int arr[] = {12, 11, 13, 5, 6, 7};      int arr\_size = sizeof(arr)/sizeof(arr[0]);        printf("Given array is \n");      printArray(arr, arr\_size);        mergeSort(arr, 0, arr\_size - 1);        printf("\nSorted array is \n");      printArray(arr, arr\_size);      return 0;  } |

Run on IDE

Output:

Given array is

12 11 13 5 6 7

Sorted array is

5 6 7 11 12 13

**Time Complexity:** Sorting arrays on different machines. Merge Sort is a recursive algorithm and time complexity can be expressed as following recurrence relation.  
T(n) = 2T(n/2) +   
The above recurrence can be solved either using Recurrence Tree method or Master method. It falls in case II of Master Method and solution of the recurrence is .  
Time complexity of Merge Sort is  in all 3 cases (worst, average and best) as merge sort always divides the array in two halves and take linear time to merge two halves.

**Auxiliary Space:** O(n)

**Algorithmic Paradigm:**Divide and Conquer

**Sorting In Place:** No in a typical implementation

**Stable:** Yes

**Applications of Merge Sort**

1. [Merge Sort is useful for sorting linked lists in O(nLogn) time](http://www.geeksforgeeks.org/merge-sort-for-linked-list/).In case of linked lists the case is different mainly due to difference in memory allocation of arrays and linked lists. Unlike arrays, linked list nodes may not be adjacent in memory. Unlike array, in linked list, we can insert items in the middle in O(1) extra space and O(1) time. Therefore merge operation of merge sort can be implemented without extra space for linked lists.

In arrays, we can do random access as elements are continuous in memory. Let us say we have an integer (4-byte) array A and let the address of A[0] be x then to access A[i], we can directly access the memory at (x + i\*4). Unlike arrays, we can not do random access in linked list. Quick Sort requires a lot of this kind of access. In linked list to access i’th index, we have to travel each and every node from the head to i’th node as we don’t have continuous block of memory. Therefore, the overhead increases for quick sort. Merge sort accesses data sequentially and the need of random access is low.

1. [Inversion Count Problem](http://www.geeksforgeeks.org/counting-inversions/)
2. Used in [External Sorting](http://en.wikipedia.org/wiki/External_sorting)

Heap Sort

Heap sort is a comparison based sorting technique based on Binary Heap data structure. It is similar to selection sort where we first find the maximum element and place the maximum element at the end. We repeat the same process for remaining element.

**What is**[**Binary Heap**](http://geeksquiz.com/binary-heap/)**?**  
Let us first define a Complete Binary Tree. A complete binary tree is a binary tree in which every level, except possibly the last, is completely filled, and all nodes are as far left as possible (Source [Wikipedia](http://en.wikipedia.org/wiki/Binary_tree#Types_of_binary_trees))

A [Binary Heap](http://geeksquiz.com/binary-heap/) is a Complete Binary Tree where items are stored in a special order such that value in a parent node is greater(or smaller) than the values in its two children nodes. The former is called as max heap and the latter is called min heap. The heap can be represented by binary tree or array.

**Why array based representation for Binary Heap?**  
Since a Binary Heap is a Complete Binary Tree, it can be easily represented as array and array based representation is space efficient. If the parent node is stored at index I, the left child can be calculated by 2 \* I + 1 and right child by 2 \* I + 2 (assuming the indexing starts at 0).

**Heap Sort Algorithm for sorting in increasing order:**  
**1.** Build a max heap from the input data.  
**2.** At this point, the largest item is stored at the root of the heap. Replace it with the last item of the heap followed by reducing the size of heap by 1. Finally, heapify the root of tree.  
**3.** Repeat above steps while size of heap is greater than 1.

**How to build the heap?**  
Heapify procedure can be applied to a node only if its children nodes are heapified. So the heapification must be performed in the bottom up order.

Lets understand with the help of an example:

Input data: 4, 10, 3, 5, 1

4(0)

/ \

10(1) 3(2)

/ \

5(3) 1(4)

The numbers in bracket represent the indices in the array

representation of data.

Applying heapify procedure to index 1:

4(0)

/ \

10(1) 3(2)

/ \

5(3) 1(4)

Applying heapify procedure to index 0:

10(0)

/ \

5(1) 3(2)

/ \

4(3) 1(4)

The heapify procedure calls itself recursively to build heap

in top down manner.

* C++
* Java
* Python

|  |
| --- |
| // C++ program for implementation of Heap Sort  #include <iostream>  using namespace std;    // To heapify a subtree rooted with node i which is  // an index in arr[]. n is size of heap  void heapify(int arr[], int n, int i)  {      int largest = i;  // Initialize largest as root      int l = 2\*i + 1;  // left = 2\*i + 1      int r = 2\*i + 2;  // right = 2\*i + 2        // If left child is larger than root      if (l < n && arr[l] > arr[largest])          largest = l;        // If right child is larger than largest so far      if (r < n && arr[r] > arr[largest])          largest = r;        // If largest is not root      if (largest != i)      {          swap(arr[i], arr[largest]);            // Recursively heapify the affected sub-tree          heapify(arr, n, largest);      }  }    // main function to do heap sort  void heapSort(int arr[], int n)  {      // Build heap (rearrange array)      for (int i = n / 2 - 1; i >= 0; i--)          heapify(arr, n, i);        // One by one extract an element from heap      for (int i=n-1; i>=0; i--)      {          // Move current root to end          swap(arr[0], arr[i]);            // call max heapify on the reduced heap          heapify(arr, i, 0);      }  }    /\* A utility function to print array of size n \*/  void printArray(int arr[], int n)  {      for (int i=0; i<n; ++i)          cout << arr[i] << " ";      cout << "\n";  }    // Driver program  int main()  {      int arr[] = {12, 11, 13, 5, 6, 7};      int n = sizeof(arr)/sizeof(arr[0]);        heapSort(arr, n);        cout << "Sorted array is \n";      printArray(arr, n);  } |

Run on IDE

Output:

Sorted array is

5 6 7 11 12 13

[Here](http://code.geeksforgeeks.org/rFO7Lm) is previous C code for reference.

**Notes:**  
Heap sort is an in-place algorithm.  
Its typical implementation is not stable, but can be made stable (See [this](http://www.geeksforgeeks.org/stability-in-sorting-algorithms/))

**Time Complexity:**Time complexity of heapify is O(Logn). Time complexity of createAndBuildHeap() is O(n) and overall time complexity of Heap Sort is O(nLogn).

**Applications of HeapSort**  
**1.** [Sort a nearly sorted (or K sorted) array](http://www.geeksforgeeks.org/nearly-sorted-algorithm/)  
**2.**[k largest(or smallest) elements in an array](http://www.geeksforgeeks.org/k-largestor-smallest-elements-in-an-array/)

Heap sort algorithm has limited uses because Quicksort and Mergesort are better in practice. Nevertheless, the Heap data structure itself is enormously used. See [Applications of Heap Data Structure](http://www.geeksforgeeks.org/applications-of-heap-data-structure/)

# QuickSort

Like [Merge Sort](http://quiz.geeksforgeeks.org/merge-sort/), QuickSort is a Divide and Conquer algorithm. It picks an element as pivot and partitions the given array around the picked pivot. There are many different versions of quickSort that pick pivot in different ways.

1. Always pick first element as pivot.
2. Always pick last element as pivot (implemented below)
3. Pick a random element as pivot.
4. Pick median as pivot.

The key process in quickSort is partition(). Target of partitions is, given an array and an element x of array as pivot, put x at its correct position in sorted array and put all smaller elements (smaller than x) before x, and put all greater elements (greater than x) after x. All this should be done in linear time.

**Pseudo Code for recursive QuickSort function :**

/\* low --> Starting index, high --> Ending index \*/

quickSort(arr[], low, high)

{

if (low < high)

{

/\* pi is partitioning index, arr[p] is now

at right place \*/

pi = partition(arr, low, high);

quickSort(arr, low, pi - 1); // Before pi

quickSort(arr, pi + 1, high); // After pi

}

}

[](http://quiz.geeksforgeeks.org/wp-content/uploads/2014/01/QuickSort2.png)

**Partition Algorithm**  
There can be many ways to do partition, following pseudo code adopts the method given in CLRS book. The logic is simple, we start from the leftmost element and keep track of index of smaller (or equal to) elements as i. While traversing, if we find a smaller element, we swap current element with arr[i]. Otherwise we ignore current element.

/\* low --> Starting index, high --> Ending index \*/

quickSort(arr[], low, high)

{

if (low < high)

{

/\* pi is partitioning index, arr[p] is now

at right place \*/

pi = partition(arr, low, high);

quickSort(arr, low, pi - 1); // Before pi

quickSort(arr, pi + 1, high); // After pi

}

}

**Pseudo code for partition()**

/\* This function takes last element as pivot, places

the pivot element at its correct position in sorted

array, and places all smaller (smaller than pivot)

to left of pivot and all greater elements to right

of pivot \*/

partition (arr[], low, high)

{

// pivot (Element to be placed at right position)

pivot = arr[high];

i = (low - 1) // Index of smaller element

for (j = low; j <= high- 1; j++)

{

// If current element is smaller than or

// equal to pivot

if (arr[j] <= pivot)

{

i++; // increment index of smaller element

swap arr[i] and arr[j]

}

}

swap arr[i + 1] and arr[high])

return (i + 1)

}

**Illustration of partition() :**

arr[] = {10, 80, 30, 90, 40, 50, 70}

Indexes: 0 1 2 3 4 5 6

low = 0, high = 6, pivot = arr[h] = 70

Initialize index of smaller element, **i = -1**

Traverse elements from j = low to high-1

**j = 0** : Since arr[j] <= pivot, do i++ and swap(arr[i], arr[j])

**i = 0**

arr[] = {10, 80, 30, 90, 40, 50, 70} // No change as i and j

// are same

**j = 1** : Since arr[j] > pivot, do nothing

// No change in i and arr[]

**j = 2** : Since arr[j] <= pivot, do i++ and swap(arr[i], arr[j])

**i = 1**

arr[] = {10, 30, 80, 90, 40, 50, 70} // We swap 80 and 30

**j = 3** : Since arr[j] > pivot, do nothing

// No change in i and arr[]

**j = 4** : Since arr[j] <= pivot, do i++ and swap(arr[i], arr[j])

**i = 2**

arr[] = {10, 30, 40, 90, 80, 50, 70} // 80 and 40 Swapped

**j = 5** : Since arr[j] <= pivot, do i++ and swap arr[i] with arr[j]

**i = 3**

arr[] = {10, 30, 40, 50, 80, 90, 70} // 90 and 50 Swapped

We come out of loop because j is now equal to high-1.

**Finally we place pivot at correct position by swapping**

**arr[i+1] and arr[high] (or pivot)**

arr[] = {10, 30, 40, 50, 70, 90, 80} // 80 and 70 Swapped

Now 70 is at its correct place. All elements smaller than

70 are before it and all elements greater than 70 are after

it.

## [We strongly recommend that you click here and practice it, before moving on to the solution.](http://www.practice.geeksforgeeks.org/probfunc-page.php?pid=700151)

**Implementation:**  
Following are C++, Java and Python implementations of QuickSort.

* C/C++
* Java
* Python

|  |
| --- |
| /\* C implementation QuickSort \*/  #include<stdio.h>    // A utility function to swap two elements  void swap(int\* a, int\* b)  {      int t = \*a;      \*a = \*b;      \*b = t;  }    /\* This function takes last element as pivot, places     the pivot element at its correct position in sorted      array, and places all smaller (smaller than pivot)     to left of pivot and all greater elements to right     of pivot \*/  int partition (int arr[], int low, int high)  {      int pivot = arr[high];    // pivot      int i = (low - 1);  // Index of smaller element        for (int j = low; j <= high- 1; j++)      {          // If current element is smaller than or          // equal to pivot          if (arr[j] <= pivot)          {              i++;    // increment index of smaller element              swap(&arr[i], &arr[j]);          }      }      swap(&arr[i + 1], &arr[high]);      return (i + 1);  }    /\* The main function that implements QuickSort   arr[] --> Array to be sorted,    low  --> Starting index,    high  --> Ending index \*/  void quickSort(int arr[], int low, int high)  {      if (low < high)      {          /\* pi is partitioning index, arr[p] is now             at right place \*/          int pi = partition(arr, low, high);            // Separately sort elements before          // partition and after partition          quickSort(arr, low, pi - 1);          quickSort(arr, pi + 1, high);      }  }    /\* Function to print an array \*/  void printArray(int arr[], int size)  {      int i;      for (i=0; i < size; i++)          printf("%d ", arr[i]);      printf("\n");  }    // Driver program to test above functions  int main()  {      int arr[] = {10, 7, 8, 9, 1, 5};      int n = sizeof(arr)/sizeof(arr[0]);      quickSort(arr, 0, n-1);      printf("Sorted array: \n");      printArray(arr, n);      return 0;  } |

Run on IDE

Output:

Sorted array:

1 5 7 8 9 10

**Analysis of QuickSort**  
Time taken by QuickSort in general can be written as following.

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

The first two terms are for two recursive calls, the last term is for the partition process. k is the number of elements which are smaller than pivot.  
The time taken by QuickSort depends upon the input array and partition strategy. Following are three cases.

***Worst Case:*** The worst case occurs when the partition process always picks greatest or smallest element as pivot. If we consider above partition strategy where last element is always picked as pivot, the worst case would occur when the array is already sorted in increasing or decreasing order. Following is recurrence for worst case.

T(n) = T(0) + T(n-1) + (n)

which is equivalent to

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

The solution of above recurrence is (n2).

***Best Case:*** The best case occurs when the partition process always picks the middle element as pivot. Following is recurrence for best case.

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

The solution of above recurrence is (nLogn). It can be solved using case 2 of [Master Theorem](http://en.wikipedia.org/wiki/Master_theorem).

***Average Case:***  
To do average case analysis, we need to [consider all possible permutation of array and calculate time taken by every permutation which doesn't look easy](http://www.geeksforgeeks.org/analysis-of-algorithms-set-2-asymptotic-analysis/).  
We can get an idea of average case by considering the case when partition puts O(n/9) elements in one set and O(9n/10) elements in other set. Following is recurrence for this case.

T(n) = T(n/9) + T(9n/10) + (n)

Solution of above recurrence is also O(nLogn)

Although the worst case time complexity of QuickSort is O(n2) which is more than many other sorting algorithms like [Merge Sort](http://quiz.geeksforgeeks.org/merge-sort/) and [Heap Sort](http://quiz.geeksforgeeks.org/heap-sort/), QuickSort is faster in practice, because its inner loop can be efficiently implemented on most architectures, and in most real-world data. QuickSort can be implemented in different ways by changing the choice of pivot, so that the worst case rarely occurs for a given type of data. However, merge sort is generally considered better when data is huge and stored in external storage.

**What is 3-Way QuickSort?**  
In simple QuickSort algorithm, we select an element as pivot, partition the array around pivot and recur for subarrays on left and right of pivot.  
Consider an array which has many redundant elements. For example, {1, 4, 2, 4, 2, 4, 1, 2, 4, 1, 2, 2, 2, 2, 4, 1, 4, 4, 4}. If 4 is picked as pivot in Simple QuickSort, we fix only one 4 and recursively process remaining occurrences. In 3 Way QuickSort, an array arr[l..r] is divided in 3 parts:  
a) arr[l..i] elements less than pivot.  
b) arr[i+1..j-1] elements equal to pivot.  
c) arr[j..r] elements greater than pivot.  
See [this](http://www.geeksforgeeks.org/3-way-quicksort/) for implementation.

**How to implement QuickSort for Linked Lists?**  
[QuickSort on Singly Linked List](http://www.geeksforgeeks.org/quicksort-on-singly-linked-list/)  
[QuickSort on Doubly Linked List](http://www.geeksforgeeks.org/quicksort-for-linked-list/)

**Can we implement QuickSort Iteratively?**  
Yes, please refer [Iterative Quick Sort](http://www.geeksforgeeks.org/iterative-quick-sort/).

**Why Quick Sort is preferred over MergeSort for sorting Arrays**  
Quick Sort in its general form is an in-place sort (i.e. it doesn’t require any extra storage) whereas merge sort requires O(N) extra storage, N denoting the array size which may be quite expensive. Allocating and de-allocating the extra space used for merge sort increases the running time of the algorithm. Comparing average complexity we find that both type of sorts have O(NlogN) average complexity but the constants differ. For arrays, merge sort loses due to the use of extra O(N) storage space.

Most practical implementations of Quick Sort use randomized version. The randomized version has expected time complexity of O(nLogn). The worst case is possible in randomized version also, but worst case doesn’t occur for a particular pattern (like sorted array) and randomized Quick Sort works well in practice.

Quick Sort is also a cache friendly sorting algorithm as it has good locality of reference when used for arrays.

Quick Sort is also tail recursive, therefore tail call optimizations is done.

**Why MergeSort is preferred over QuickSort for Linked Lists?**  
In case of linked lists the case is different mainly due to difference in memory allocation of arrays and linked lists. Unlike arrays, linked list nodes may not be adjacent in memory. Unlike array, in linked list, we can insert items in the middle in O(1) extra space and O(1) time. Therefore merge operation of merge sort can be implemented without extra space for linked lists.

In arrays, we can do random access as elements are continuous in memory. Let us say we have an integer (4-byte) array A and let the address of A[0] be x then to access A[i], we can directly access the memory at (x + i\*4). Unlike arrays, we can not do random access in linked list. Quick Sort requires a lot of this kind of access. In linked list to access i’th index, we have to travel each and every node from the head to i’th node as we don’t have continuous block of memory. Therefore, the overhead increases for quick sort. Merge sort accesses data sequentially and the need of random access is low.

Radix Sort

The [lower bound for Comparison based sorting algorithm](http://www.geeksforgeeks.org/lower-bound-on-comparison-based-sorting-algorithms/) (Merge Sort, Heap Sort, Quick-Sort .. etc) is Ω(nLogn), i.e., they cannot do better than nLogn.

[Counting sort](http://www.geeksforgeeks.org/counting-sort/) is a linear time sorting algorithm that sort in O(n+k) time when elements are in range from 1 to k.

***What if the elements are in range from 1 to n2?***  
We can’t use counting sort because counting sort will take O(n2) which is worse than comparison based sorting algorithms. Can we sort such an array in linear time?  
[Radix Sort](http://en.wikipedia.org/wiki/Radix_sort) is the answer. The idea of Radix Sort is to do digit by digit sort starting from least significant digit to most significant digit. Radix sort uses counting sort as a subroutine to sort.

***The Radix Sort Algorithm***  
**1)** Do following for each digit i where i varies from least significant digit to the most significant digit.  
………….**a)** Sort input array using counting sort (or any stable sort) according to the i’th digit.

**Example:**  
Original, unsorted list:

170, 45, 75, 90, 802, 24, 2, 66

Sorting by least significant digit (1s place) gives: [\*Notice that we keep 802 before 2, because 802 occurred before 2 in the original list, and similarly for pairs 170 & 90 and 45 & 75.]

170, 90, 802, 2, 24, 45, 75, 66

Sorting by next digit (10s place) gives: [\*Notice that 802 again comes before 2 as 802 comes before 2 in the previous list.]

802, 2, 24, 45, 66, 170, 75, 90

Sorting by most significant digit (100s place) gives:

2, 24, 45, 66, 75, 90, 170, 802

***What is the running time of Radix Sort?***  
Let there be d digits in input integers. Radix Sort takes O(d\*(n+b)) time where b is the base for representing numbers, for example, for decimal system, b is 10. What is the value of d? If k is the maximum possible value, then d would be O(logb(k)). So overall time complexity is O((n+b) \* logb(k)). Which looks more than the time complexity of comparison based sorting algorithms for a large k. Let us first limit k. Let k <= nc where c is a constant. In that case, the complexity becomes O(nLogb(n)). But it still doesn’t beat comparison based sorting algorithms.  
What if we make value of b larger?. What should be the value of b to make the time complexity linear? If we set b as n, we get the time complexity as O(n). In other words, we can sort an array of integers with range from 1 to nc if the numbers are represented in base n (or every digit takes log2(n) bits).

***Is Radix Sort preferable to Comparison based sorting algorithms like Quick-Sort?***  
If we have log2n bits for every digit, the running time of Radix appears to be better than Quick Sort for a wide range of input numbers. The constant factors hidden in asymptotic notation are higher for Radix Sort and Quick-Sort uses hardware caches more effectively. Also, Radix sort uses counting sort as a subroutine and counting sort takes extra space to sort numbers.

**Implementation of Radix Sort**  
Following is a simple C++ implementation of Radix Sort. For simplicity, the value of d is assumed to be 10. We recommend you to see [Counting Sort](http://www.geeksforgeeks.org/counting-sort/) for details of countSort() function in below code.

* C/C++
* Java
* Python

|  |
| --- |
| // C++ implementation of Radix Sort  #include<iostream>  using namespace std;    // A utility function to get maximum value in arr[]  int getMax(int arr[], int n)  {      int mx = arr[0];      for (int i = 1; i < n; i++)          if (arr[i] > mx)              mx = arr[i];      return mx;  }    // A function to do counting sort of arr[] according to  // the digit represented by exp.  void countSort(int arr[], int n, int exp)  {      int output[n]; // output array      int i, count[10] = {0};        // Store count of occurrences in count[]      for (i = 0; i < n; i++)          count[ (arr[i]/exp)%10 ]++;        // Change count[i] so that count[i] now contains actual      //  position of this digit in output[]      for (i = 1; i < 10; i++)          count[i] += count[i - 1];        // Build the output array      for (i = n - 1; i >= 0; i--)      {          output[count[ (arr[i]/exp)%10 ] - 1] = arr[i];          count[ (arr[i]/exp)%10 ]--;      }        // Copy the output array to arr[], so that arr[] now      // contains sorted numbers according to current digit      for (i = 0; i < n; i++)          arr[i] = output[i];  }    // The main function to that sorts arr[] of size n using  // Radix Sort  void radixsort(int arr[], int n)  {      // Find the maximum number to know number of digits      int m = getMax(arr, n);        // Do counting sort for every digit. Note that instead      // of passing digit number, exp is passed. exp is 10^i      // where i is current digit number      for (int exp = 1; m/exp > 0; exp \*= 10)          countSort(arr, n, exp);  }    // A utility function to print an array  void print(int arr[], int n)  {      for (int i = 0; i < n; i++)          cout << arr[i] << " ";  }    // Driver program to test above functions  int main()  {      int arr[] = {170, 45, 75, 90, 802, 24, 2, 66};      int n = sizeof(arr)/sizeof(arr[0]);      radixsort(arr, n);      print(arr, n);      return 0;  } |

Output:

2 24 45 66 75 90 170 802

Counting Sort

[Counting sort](http://en.wikipedia.org/wiki/Counting_sort) is a sorting technique based on keys between a specific range. It works by counting the number of objects having distinct key values (kind of hashing). Then doing some arithmetic to calculate the position of each object in the output sequence.

Let us understand it with the help of an example.

For simplicity, consider the data in the range 0 to 9.

Input data: 1, 4, 1, 2, 7, 5, 2

1) Take a count array to store the count of each unique object.

Index: 0 1 2 3 4 5 6 7 8 9

Count: 0 2 2 0 1 1 0 1 0 0

2) Modify the count array such that each element at each index

stores the sum of previous counts.

Index: 0 1 2 3 4 5 6 7 8 9

Count: 0 2 4 4 5 6 6 7 7 7

The modified count array indicates the position of each object in

the output sequence.

3) Output each object from the input sequence followed by

decreasing its count by 1.

Process the input data: 1, 4, 1, 2, 7, 5, 2. Position of 1 is 2.

Put data 1 at index 2 in output. Decrease count by 1 to place

next data 1 at an index 1 smaller than this index.

Following is C implementation of counting sort.

* C/C++
* Java
* Python

|  |
| --- |
| // C Program for counting sort  #include <stdio.h>  #include <string.h>  #define RANGE 255    // The main function that sort the given string arr[] in  // alphabatical order  void countSort(char arr[])  {      // The output character array that will have sorted arr      char output[strlen(arr)];        // Create a count array to store count of inidividul      // characters and initialize count array as 0      int count[RANGE + 1], i;      memset(count, 0, sizeof(count));        // Store count of each character      for(i = 0; arr[i]; ++i)          ++count[arr[i]];        // Change count[i] so that count[i] now contains actual      // position of this character in output array      for (i = 1; i <= RANGE; ++i)          count[i] += count[i-1];        // Build the output character array      for (i = 0; arr[i]; ++i)      {          output[count[arr[i]]-1] = arr[i];          --count[arr[i]];      }        // Copy the output array to arr, so that arr now      // contains sorted characters      for (i = 0; arr[i]; ++i)          arr[i] = output[i];  }    // Driver program to test above function  int main()  {      char arr[] = "geeksforgeeks";//"applepp";        countSort(arr);        printf("Sorted character array is %s\n", arr);      return 0;  } |

Run on IDE

Output:

Sorted character array is eeeefggkkorss

**Time Complexity:** O(n+k) where n is the number of elements in input array and k is the range of input.  
**Auxiliary Space:** O(n+k)

**Points to be noted:**  
**1.** Counting sort is efficient if the range of input data is not significantly greater than the number of objects to be sorted. Consider the situation where the input sequence is between range 1 to 10K and the data is 10, 5, 10K, 5K.  
**2.** It is not a comparison based sorting. It running time complexity is O(n) with space proportional to the range of data.  
**3.** It is often used as a sub-routine to another sorting algorithm like radix sort.  
**4.** Counting sort uses a partial hashing to count the occurrence of the data object in O(1).  
**5.** Counting sort can be extended to work for negative inputs also.

**Exercise:**  
**1.** Modify above code to sort the input data in the range from M to N.  
**2.** Modify above code to sort negative input data.  
**3.** Is counting sort stable and online?  
**4.**Thoughts on parallelizing the counting sort algorithm.

Bucket Sort

Bucket sort is mainly useful when input is uniformly distributed over a range. For example, consider the following problem.   
*Sort a large set of floating point numbers which are in range from 0.0 to 1.0 and are uniformly distributed across the range. How do we sort the numbers efficiently?*

A simple way is to apply a comparison based sorting algorithm. The [lower bound for Comparison based sorting algorithm](http://www.geeksforgeeks.org/lower-bound-on-comparison-based-sorting-algorithms/) (Merge Sort, Heap Sort, Quick-Sort .. etc) is Ω(n Log n), i.e., they cannot do better than nLogn.  
Can we sort the array in linear time? [Counting sort](http://www.geeksforgeeks.org/counting-sort/) can not be applied here as we use keys as index in counting sort. Here keys are floating point numbers.   
The idea is to use bucket sort. Following is bucket algorithm.

**bucketSort(arr[], n)**

1) Create n empty buckets (Or lists).

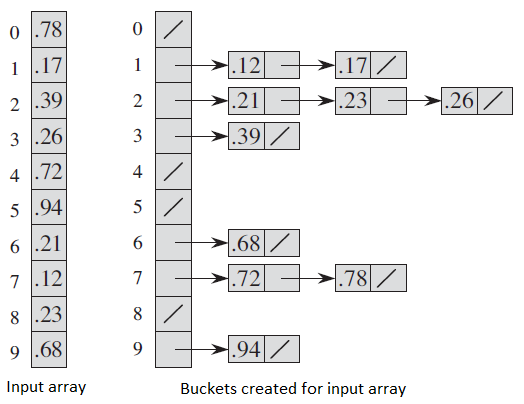
2) Do following for every array element arr[i].

.......a) Insert arr[i] into bucket[n\*array[i]]

3) Sort individual buckets using insertion sort.

4) Concatenate all sorted buckets.

Following diagram (taken from [CLRS book](http://www.flipkart.com/introduction-algorithms-3rd/p/itmdvd93bzvrnc7b?pid=9788120340077&affid=sandeepgfg)) demonstrates working of bucket sort.

[](http://d1hyf4ir1gqw6c.cloudfront.net/wp-content/uploads/BucketSort.png)

**Time Complexity:** If we assume that insertion in a bucket takes O(1) time then steps 1 and 2 of the above algorithm clearly take O(n) time. The O(1) is easily possible if we use a linked list to represent a bucket (In the following code, C++ vector is used for simplicity). Step 4 also takes O(n) time as there will be n items in all buckets.  
The main step to analyze is step 3. This step also takes O(n) time on average if all numbers are uniformly distributed (please refer [CLRS book](http://www.flipkart.com/introduction-algorithms-3rd/p/itmdvd93bzvrnc7b?pid=9788120340077&affid=sandeepgfg) for more details)

Following is C++ implementation of the above algorithm.

|  |
| --- |
| // C++ program to sort an array using bucket sort  #include <iostream>  #include <algorithm>  #include <vector>  using namespace std;    // Function to sort arr[] of size n using bucket sort  void bucketSort(float arr[], int n)  {      // 1) Create n empty buckets      vector<float> b[n];        // 2) Put array elements in different buckets      for (int i=0; i<n; i++)      {         int bi = n\*arr[i]; // Index in bucket         b[bi].push\_back(arr[i]);      }        // 3) Sort individual buckets      for (int i=0; i<n; i++)         sort(b[i].begin(), b[i].end());        // 4) Concatenate all buckets into arr[]      int index = 0;      for (int i = 0; i < n; i++)          for (int j = 0; j < b[i].size(); j++)            arr[index++] = b[i][j];  }    /\* Driver program to test above funtion \*/  int main()  {      float arr[] = {0.897, 0.565, 0.656, 0.1234, 0.665, 0.3434};      int n = sizeof(arr)/sizeof(arr[0]);      bucketSort(arr, n);        cout << "Sorted array is \n";      for (int i=0; i<n; i++)         cout << arr[i] << " ";      return 0;  } |

Run on IDE

Output:

Sorted array is

0.1234 0.3434 0.565 0.656 0.665 0.897

**References:**  
[Introduction to Algorithms 3rd Edition by Clifford Stein, Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest](http://www.flipkart.com/introduction-algorithms-3rd/p/itmczynzhyhxv2gs?pid=9788120340077&affid=sandeepgfg)  
<http://en.wikipedia.org/wiki/Bucket_sort>

ShellSort

[ShellSort](http://en.wikipedia.org/wiki/Shellsort)is mainly a variation of [Insertion Sort](http://quiz.geeksforgeeks.org/insertion-sort/). In insertion sort, we move elements only one position ahead. When an element has to be moved far ahead, many movements are involved. The idea of shellSort is to allow exchange of far items. In shellSort, we make the array h-sorted for a large value of h. We keep reducing the value of h until it becomes 1. An array is said to be h-sorted if all sublists of every h’th element is sorted.

Following is C++ implementation of ShellSort.

* C++
* Java
* Python

|  |
| --- |
| // C++ implementation of Shell Sort  #include  <iostream>  using namespace std;    /\* function to sort arr using shellSort \*/  int shellSort(int arr[], int n)  {      // Start with a big gap, then reduce the gap      for (int gap = n/2; gap > 0; gap /= 2)      {          // Do a gapped insertion sort for this gap size.          // The first gap elements a[0..gap-1] are already in gapped order          // keep adding one more element until the entire array is          // gap sorted          for (int i = gap; i < n; i += 1)          {              // add a[i] to the elements that have been gap sorted              // save a[i] in temp and make a hole at position i              int temp = arr[i];                // shift earlier gap-sorted elements up until the correct              // location for a[i] is found              int j;              for (j = i; j >= gap && arr[j - gap] > temp; j -= gap)                  arr[j] = arr[j - gap];                //  put temp (the original a[i]) in its correct location              arr[j] = temp;          }      }      return 0;  }    void printArray(int arr[], int n)  {      for (int i=0; i<n; i++)          cout << arr[i] << " ";  }    int main()  {      int arr[] = {12, 34, 54, 2, 3}, i;      int n = sizeof(arr)/sizeof(arr[0]);        cout << "Array before sorting: \n";      printArray(arr, n);        shellSort(arr, n);        cout << "\nArray after sorting: \n";      printArray(arr, n);        return 0;  } |

Run on IDE

Output:

Array before sorting:

12 34 54 2 3

Array after sorting:

2 3 12 34 54

**Time Complexity:** Time complexity of above implementation of shellsort is O(n2). In the above implementation gap is reduce by half in every iteration. There are many other ways to reduce gap which lead to better time complexity. See [this](http://en.wikipedia.org/wiki/Shellsort#Gap_sequences)for more details.

**References:**  
<https://www.youtube.com/watch?v=pGhazjsFW28>  
<http://en.wikipedia.org/wiki/Shellsort>

# Comb Sort

Comb Sort is mainly an improvement over Bubble Sort. Bubble sort always compares adjacent values. So all [inversions](http://www.geeksforgeeks.org/counting-inversions/) are removed one by one. Comb Sort improves on Bubble Sort by using gap of size more than 1. The gap starts with a large value and shrinks by a factor of 1.3 in every iteration until it reaches the value 1. Thus Comb Sort removes more than one [inversion counts](http://www.geeksforgeeks.org/counting-inversions/) with one swap and performs better than Bublle Sort.

The shrink factor has been empirically found to be 1.3 (by testing Combsort on over 200,000 random lists) [Source: [Wiki](https://en.wikipedia.org/wiki/Comb_sort)]

Although, it works better than Bubble Sort on average, worst case remains O(n2).

Below is C++ implementation.

* C++
* Java
* Python

|  |
| --- |
| // C++ implementation of Comb Sort  #include<bits/stdc++.h>  using namespace std;    // To find gap between elements  int getNextGap(int gap)  {      // Shrink gap by Shrink factor      gap = (gap\*10)/13;        if (gap < 1)          return 1;      return gap;  }    // Function to sort a[0..n-1] using Comb Sort  void combSort(int a[], int n)  {      // Initialize gap      int gap = n;        // Initialize swapped as true to make sure that      // loop runs      bool swapped = true;        // Keep running while gap is more than 1 and last      // iteration caused a swap      while (gap != 1 || swapped == true)      {          // Find next gap          gap = getNextGap(gap);            // Initialize swapped as false so that we can          // check if swap happened or not          swapped = false;            // Compare all elements with current gap          for (int i=0; i<n-gap; i++)          {              if (a[i] > a[i+gap])              {                  swap(a[i], a[i+gap]);                  swapped = true;              }          }      }  }    // Driver program  int main()  {      int a[] = {8, 4, 1, 56, 3, -44, 23, -6, 28, 0};      int n = sizeof(a)/sizeof(a[0]);        combSort(a, n);        printf("Sorted array: \n");      for (int i=0; i<n; i++)          printf("%d ", a[i]);        return 0;  } |

Run on IDE

Output :

Sorted array:

-44 -6 0 1 3 4 8 23 28 56

**Illustration:**

Let the array elements be

8, 4, 1, 56, 3, -44, 23, -6, 28, 0

Initially gap value = 10  
After shrinking gap value => 10/1.3 = **7**;

**8** 4 1 56 3 -44 23 **-6** 28 0

-6 4 **1** 56 3 -44 23 8 28 **0**

-6 4 0 56 3 -44 23 8 28 1

New gap value => 7/1.3 = **5**;

-44 4 0 **56** 3 -6 23 8 **28** 1

-44 4 0 28 **3** -6 23 8 **56** 1

-44 4 0 28 1 -6 23 8 56 3

New gap value => 5/1.3 = **3**;

-44 1 **0** 28 4 **-6** 23 8 56 3

-44 1 -6 **28** 4 0 **23** 8 56 3

-44 1 -6 23 4 0 **28** 8 56 **3**

-44 1 -6 23 4 0 3 8 56 28

New gap value => 3/1.3 = **2**;

-44 1 -6 0 **4** 23 **3** 8 56 28

-44 1 -6 0 3 **23** 4 **8** 56 28

-44 1 -6 0 3 8 4 23 56 28

New gap value => 2/1.3 = **1**;

-44 -6 **1 0** 3 8 4 23 56 28

-44 -6 0 1 3 **8 4** 23 56 28

-44 -6 0 1 3 4 8 23 **56 28**

-44 -6 0 1 3 4 8 23 28 56

no more swaps required (Array sorted)

**Time Complexity :**Worst case complexity of this algorithm is O(n2) and the Best Case complexity is O(n).

**Auxiliary Space :**O(1).

## [Quiz on Comb Sort](http://geeksquiz.com/quiz-combsort/)

This article is contributed by **Rahul Agrawal**. If you like GeeksforGeeks and would like to contribute, you can also write an article and mail your article to contribute@geeksforgeeks.org. See your article appearing on the GeeksforGeeks main page and help other Geeks.

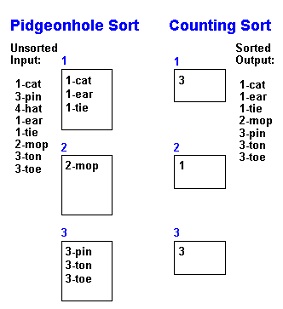
Pigeonhole Sort

[Pigeonhole sorting](https://en.wikipedia.org/wiki/Pigeonhole_sort) is a sorting algorithm that is suitable for sorting lists of elements where the number of elements and the number of possible key values are approximately the same.  
It requires O(*n* + *Range*) time where n is number of elements in input array and ‘Range’ is number of possible values in array.

**Working of Algorithm :**

1. Find minimum and maximum values in array. Let the minimum and maximum values be ‘min’ and ‘max’ respectively. Also find range as ‘max-min-1’.
2. Set up an array of initially empty “pigeonholes” the same size as of the range.
3. Visit each element of the array and then put each element in its pigeonhole. An element arr[i] is put in hole at index arr[i] – min.
4. Start the loop all over the pigeonhole array in order and put the elements from non- empty holes back into the original array.

**Comparison with Counting Sort :**  
It is similar to [counting sort](http://www.geeksforgeeks.org/counting-sort/), but differs in that it “moves items twice: once to the bucket array and again to the final destination “.

[](http://d1hyf4ir1gqw6c.cloudfront.net/wp-content/uploads/ps.jpg)

Below is C++ implementation of Pegionhole Sort.

|  |
| --- |
| /\* C program to implement Pegionhole Sort \*/  #include <bits/stdc++.h>  using namespace std;    /\* Sorts the array using pigeonhole algorithm \*/  void pigeonholeSort(int arr[], int n)  {      // Find minimum and maximum values in arr[]      int min = arr[0], max = arr[0];      for (int i = 1; i < n; i++)      {          if (arr[i] < min)              min = arr[i];          if (arr[i] > max)              max = arr[i];      }      int range = max - min + 1; // Find range        // Create an array of vectors. Size of array      // range. Each vector represents a hole that      // is going to contain matching elements.      vector<int> holes[range];        // Traverse through input array and put every      // element in its respective hole      for (int i = 0; i < n; i++)          holes[arr[i]-min].push\_back(arr[i]);        // Traverse through all holes one by one. For      // every hole, take its elements and put in      // array.      int index = 0;  // index in sorted array      for (int i = 0; i < range; i++)      {         vector<int>::iterator it;         for (it = holes[i].begin(); it != holes[i].end(); ++it)              arr[index++]  = \*it;      }  }    // Driver program to test the above function  int main()  {      int arr[] = {8, 3, 2, 7, 4, 6, 8};      int n = sizeof(arr)/sizeof(arr[0]);        pigeonholeSort(arr, n);        printf("Sorted order is : ");      for (int i = 0; i < n; i++)          printf("%d ", arr[i]);        return 0;  } |

Run on IDE

Output:

Sorted order is : 2 3 4 6 7 8 8

Pigeonhole sort has limited use as requirements are rarely met. For arrays where range is much larger than *n*, bucket sort is a generalization that is more efficient in space and time.

**References:**  
<https://en.wikipedia.org/wiki/Pigeonhole_sort>

This article is contributed **Ayush Govil**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

3.[Interpolation search vs Binary search](http://www.geeksforgeeks.org/g-fact-84/)

# Interpolation search vs Binary search

[Interpolation search](http://en.wikipedia.org/wiki/Interpolation_search) works better than Binary Search for a sorted and uniformly distributed array.

On average the interpolation search makes about log(log(n)) comparisons (if the elements are uniformly distributed), where n is the number of elements to be searched. In the worst case (for instance where the numerical values of the keys increase exponentially) it can make up to O(n) comparisons.

# Stability in sorting algorithms

A sorting algorithm is said to be stable if two objects with equal keys appear in the same order in sorted output as they appear in the input unsorted array. Some sorting algorithms are stable by nature like Insertion sort, Merge Sort, Bubble Sort, etc. And some sorting algorithms are not, like Heap Sort, Quick Sort, etc.

However, any given sorting algo which is not stable can be modified to be stable. There can be sorting algo specific ways to make it stable, but in general, any comparison based sorting algorithm which is not stable by nature can be modified to be stable by changing the key comparison operation so that the comparison of two keys considers position as a factor for objects with equal keys.

References:  
<http://www.math.uic.edu/~leon/cs-mcs401-s08/handouts/stability.pdf>  
<http://en.wikipedia.org/wiki/Sorting_algorithm#Stability>

# When does the worst case of Quicksort occur?

The answer depends on strategy for choosing pivot. In early versions of Quick Sort where leftmost (or rightmost) element is chosen as pivot, the worst occurs in following cases.

1) Array is already sorted in same order.  
2) Array is already sorted in reverse order.  
3) All elements are same (special case of case 1 and 2)

Since these cases are very common use cases, the problem was easily solved by choosing either a random index for the pivot, choosing the middle index of the partition or (especially for longer partitions) choosing the median of the first, middle and last element of the partition for the pivot. With these modifications, the worst case of Quick sort has less chances to occur, but worst case can still occur if the input array is such that the maximum (or minimum) element is always chosen as pivot.

References:  
<http://en.wikipedia.org/wiki/Quicksort>

# Lower bound for comparison based sorting algorithms

The problem of sorting can be viewed as following.

**Input:** A sequence of n numbers <a1, a2, . . . , an>.  
**Output:** A permutation (reordering) <a‘1, a‘2, . . . , a‘n> of the input sequence such that a‘1 <= a‘2 ….. <= a‘n.

A sorting algorithm is comparison based if it uses comparison operators to find the order between two numbers.  Comparison sorts can be viewed abstractly in terms of decision trees. A decision tree is a[full binary tree](http://en.wikipedia.org/wiki/Binary_tree#Types_of_binary_trees) that represents the comparisons between elements that are performed by a particular sorting algorithm operating on an input of a given size. The execution of the sorting algorithm corresponds to tracing a path from the root of the decision tree to a leaf. At each internal node, a comparison ai <= aj is made. The left subtree then dictates subsequent comparisons for ai <= aj, and the right subtree dictates subsequent comparisons for ai > aj. When we come to a leaf, the sorting algorithm has established the ordering. So we can say following about the decison tree.

**1)**Each of the n! permutations on n elements must appear as one of the leaves of the decision tree for the sorting algorithm to sort properly.

**2)**Let x be the maximum number of comparisons in a sorting algorithm. The maximum height of the decison tree would be x. A tree with maximum height x has at most 2^x leaves.

After combining the above two facts, we get following relation.

n!  <= 2^x

Taking Log on both sides.

log2(n!) <= x

Since log2(n!) = Θ(nLogn), we can say

x = Ω(nLog2n)

Therefore, any comparison based sorting algorithm must make at least nLog2n comparisons to sort the input array, and Heapsort and merge sort are asymptotically optimal comparison sorts.

**References:**  
[Introduction to Algorithms, by Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest and Clifford Stein](http://mitpress.mit.edu/algorithms/)

# Which sorting algorithm makes minimum number of memory writes?

Minimizing the number of writes is useful when making writes to some huge data set is very expensive, such as with [EEPROMs](http://en.wikipedia.org/wiki/EEPROM) or [Flash memory](http://en.wikipedia.org/wiki/Flash_memory), where each write reduces the lifespan of the memory.

Among the sorting algorithms that we generally study in our data structure and algorithm courses,  [Selection Sort](http://en.wikipedia.org/wiki/Selection_sort) makes least number of writes (it makes O(n) swaps).  But, [Cycle Sort](http://en.wikipedia.org/wiki/Cycle_sort) almost always makes less number of writes compared to Selection Sort.  In Cycle Sort, each value is either written zero times, if it’s already in its correct position, or written one time to its correct position. This matches the minimal number of overwrites required for a completed in-place sort.

Sources:  
<http://en.wikipedia.org/wiki/Cycle_sort>  
<http://en.wikipedia.org/wiki/Selection_sort>

# Find the Minimum length Unsorted Subarray, sorting which makes the complete array sorted

Given an unsorted array arr[0..n-1] of size n, find the minimum length subarray arr[s..e] such that sorting this subarray makes the whole array sorted.  
 **Examples:**  
1) If the input array is [10, 12, 20, 30, 25, 40, 32, 31, 35, 50, 60], your program should be able to find that the subarray lies between the indexes 3 and 8.

2) If the input array is [0, 1, 15, 25, 6, 7, 30, 40, 50], your program should be able to find that the subarray lies between the indexes 2 and 5.

## [We strongly recommend that you click here and practice it, before moving on to the solution.](http://www.practice.geeksforgeeks.org/problem-page.php?pid=573)

**Solution:**  
**1) Find the candidate unsorted subarray**  
a) Scan from left to right and find the first element which is greater than the next element. Let sbe the index of such an element. In the above example 1, sis 3 (index of 30).  
b) Scan from right to left and find the first element (first in right to left order) which is smaller than the next element (next in right to left order). Let ebe the index of such an element. In the above example 1, e is 7 (index of 31).

**2) Check whether sorting the candidate unsorted subarray makes the complete array sorted or not. If not, then include more elements in the subarray.**  
a) Find the minimum and maximum values in arr[s..e]. Let minimum and maximum values be minand max. minand maxfor [30, 25, 40, 32, 31] are 25 and 40 respectively.  
b) Find the first element (if there is any) in arr[0..s-1] which is greater than min, change sto index of this element. There is no such element in above example 1.  
c) Find the last element (if there is any) in arr[e+1..n-1]which is smaller than max, change eto index of this element. In the above example 1, e is changed to 8 (index of 35)

**3) Print sand e.**

**Implementation:**

|  |
| --- |
| #include<stdio.h>    void printUnsorted(int arr[], int n)  {    int s = 0, e = n-1, i, max, min;      // step 1(a) of above algo    for (s = 0; s < n-1; s++)    {      if (arr[s] > arr[s+1])        break;    }    if (s == n-1)    {      printf("The complete array is sorted");      return;    }      // step 1(b) of above algo    for(e = n - 1; e > 0; e--)    {      if(arr[e] < arr[e-1])        break;    }      // step 2(a) of above algo    max = arr[s]; min = arr[s];    for(i = s + 1; i <= e; i++)    {      if(arr[i] > max)        max = arr[i];      if(arr[i] < min)        min = arr[i];    }      // step 2(b) of above algo    for( i = 0; i < s; i++)    {      if(arr[i] > min)      {        s = i;        break;      }    }      // step 2(c) of above algo    for( i = n -1; i >= e+1; i--)    {      if(arr[i] < max)      {        e = i;        break;      }    }      // step 3 of above algo    printf(" The unsorted subarray which makes the given array "           " sorted lies between the indees %d and %d", s, e);    return;  }    int main()  {    int arr[] = {10, 12, 20, 30, 25, 40, 32, 31, 35, 50, 60};    int arr\_size = sizeof(arr)/sizeof(arr[0]);    printUnsorted(arr, arr\_size);    getchar();    return 0;  } |

**Time Complexity:**O(n)

Merge Sort for Linked Lists

[Merge sort](http://en.wikipedia.org/wiki/Merge_sort) is often preferred for sorting a linked list. The slow random-access performance of a linked list makes some other algorithms (such as quicksort) perform poorly, and others (such as heapsort) completely impossible.

Let head be the first node of the linked list to be sorted and headRef be the pointer to head. Note that we need a reference to head in MergeSort() as the below implementation changes next links to sort the linked lists (not data at the nodes), so head node has to be changed if the data at original head is not the smallest value in linked list.

MergeSort(headRef)

1) If head is NULL or there is only one element in the Linked List

then return.

2) Else divide the linked list into two halves.

FrontBackSplit(head, &a, &b); /\* a and b are two halves \*/

3) Sort the two halves a and b.

MergeSort(a);

MergeSort(b);

4) Merge the sorted a and b (using SortedMerge() discussed [here](http://geeksforgeeks.org/?p=3622))

and update the head pointer using headRef.

\*headRef = SortedMerge(a, b);

|  |
| --- |
| #include<stdio.h>  #include<stdlib.h>    /\* Link list node \*/  struct node  {      int data;      struct node\* next;  };    /\* function prototypes \*/  struct node\* SortedMerge(struct node\* a, struct node\* b);  void FrontBackSplit(struct node\* source,            struct node\*\* frontRef, struct node\*\* backRef);    /\* sorts the linked list by changing next pointers (not data) \*/  void MergeSort(struct node\*\* headRef)  {    struct node\* head = \*headRef;    struct node\* a;    struct node\* b;      /\* Base case -- length 0 or 1 \*/    if ((head == NULL) || (head->next == NULL))    {      return;    }      /\* Split head into 'a' and 'b' sublists \*/    FrontBackSplit(head, &a, &b);      /\* Recursively sort the sublists \*/    MergeSort(&a);    MergeSort(&b);      /\* answer = merge the two sorted lists together \*/    \*headRef = SortedMerge(a, b);  }    /\* See <http://geeksforgeeks.org/?p=3622> for details of this     function \*/  struct node\* SortedMerge(struct node\* a, struct node\* b)  {    struct node\* result = NULL;      /\* Base cases \*/    if (a == NULL)       return(b);    else if (b==NULL)       return(a);      /\* Pick either a or b, and recur \*/    if (a->data <= b->data)    {       result = a;       result->next = SortedMerge(a->next, b);    }    else    {       result = b;       result->next = SortedMerge(a, b->next);    }    return(result);  }    /\* UTILITY FUNCTIONS \*/  /\* Split the nodes of the given list into front and back halves,       and return the two lists using the reference parameters.       If the length is odd, the extra node should go in the front list.       Uses the fast/slow pointer strategy.  \*/  void FrontBackSplit(struct node\* source,            struct node\*\* frontRef, struct node\*\* backRef)  {    struct node\* fast;    struct node\* slow;    if (source==NULL || source->next==NULL)    {      /\* length < 2 cases \*/      \*frontRef = source;      \*backRef = NULL;    }    else    {      slow = source;      fast = source->next;        /\* Advance 'fast' two nodes, and advance 'slow' one node \*/      while (fast != NULL)      {        fast = fast->next;        if (fast != NULL)        {          slow = slow->next;          fast = fast->next;        }      }        /\* 'slow' is before the midpoint in the list, so split it in two        at that point. \*/      \*frontRef = source;      \*backRef = slow->next;      slow->next = NULL;    }  }    /\* Function to print nodes in a given linked list \*/  void printList(struct node \*node)  {    while(node!=NULL)    {     printf("%d ", node->data);     node = node->next;    }  }    /\* Function to insert a node at the beginging of the linked list \*/  void push(struct node\*\* head\_ref, int new\_data)  {    /\* allocate node \*/    struct node\* new\_node =              (struct node\*) malloc(sizeof(struct node));      /\* put in the data  \*/    new\_node->data  = new\_data;      /\* link the old list off the new node \*/    new\_node->next = (\*head\_ref);      /\* move the head to point to the new node \*/    (\*head\_ref)    = new\_node;  }    /\* Drier program to test above functions\*/  int main()  {    /\* Start with the empty list \*/    struct node\* res = NULL;    struct node\* a = NULL;      /\* Let us create a unsorted linked lists to test the functions     Created lists shall be a: 2->3->20->5->10->15 \*/    push(&a, 15);    push(&a, 10);    push(&a, 5);    push(&a, 20);    push(&a, 3);    push(&a, 2);      /\* Sort the above created Linked List \*/    MergeSort(&a);      printf("\n Sorted Linked List is: \n");    printList(a);      getchar();    return 0;  } |

Run on IDE

Time Complexity: O(nLogn)

Sources:  
<http://en.wikipedia.org/wiki/Merge_sort>  
<http://cslibrary.stanford.edu/105/LinkedListProblems.pdf>

Sort a nearly sorted (or K sorted) array

Given an array of n elements, where each element is at most k away from its target position, devise an algorithm that sorts in O(n log k) time.   
For example, let us consider k is 2, an element at index 7 in the sorted array, can be at indexes 5, 6, 7, 8, 9 in the given array.

Source: [Nearly sorted algorithm](http://geeksforgeeks.org/forum/topic/nearly-sorted-algorithm-on-log-k)

We can **use Insertion Sort** to sort the elements efficiently. Following is the C code for standard Insertion Sort.

|  |
| --- |
| /\* Function to sort an array using insertion sort\*/  void insertionSort(int A[], int size)  {     int i, key, j;     for (i = 1; i < size; i++)     {         key = A[i];         j = i-1;           /\* Move elements of A[0..i-1], that are greater than key, to one            position ahead of their current position.            This loop will run at most k times \*/         while (j >= 0 && A[j] > key)         {             A[j+1] = A[j];             j = j-1;         }         A[j+1] = key;     }  } |

Run on IDE

The inner loop will run at most k times. To move every element to its correct place, at most k elements need to be moved. So overall *complexity will be O(nk)*

We can sort such arrays**more efficiently with the help of Heap data structure**. Following is the detailed process that uses Heap.  
1) Create a Min Heap of size k+1 with first k+1 elements. This will take O(k) time (See [this GFact](http://www.geeksforgeeks.org/archives/12580))  
2) One by one remove min element from heap, put it in result array, and add a new element to heap from remaining elements.

Removing an element and adding a new element to min heap will take Logk time. So overall complexity will be O(k) + O((n-k)\*logK)

|  |
| --- |
| #include<iostream>  using namespace std;    // Prototype of a utility function to swap two integers  void swap(int \*x, int \*y);    // A class for Min Heap  class MinHeap  {      int \*harr; // pointer to array of elements in heap      int heap\_size; // size of min heap  public:      // Constructor      MinHeap(int a[], int size);        // to heapify a subtree with root at given index      void MinHeapify(int );        // to get index of left child of node at index i      int left(int i) { return (2\*i + 1); }        // to get index of right child of node at index i      int right(int i) { return (2\*i + 2); }        // to remove min (or root), add a new value x, and return old root      int replaceMin(int x);        // to extract the root which is the minimum element      int extractMin();  };    // Given an array of size n, where every element is k away from its target  // position, sorts the array in O(nLogk) time.  int sortK(int arr[], int n, int k)  {      // Create a Min Heap of first (k+1) elements from      // input array      int \*harr = new int[k+1];      for (int i = 0; i<=k && i<n; i++) // i < n condition is needed when k > n          harr[i] = arr[i];      MinHeap hp(harr, k+1);        // i is index for remaining elements in arr[] and ti      // is target index of for cuurent minimum element in      // Min Heapm 'hp'.      for(int i = k+1, ti = 0; ti < n; i++, ti++)      {          // If there are remaining elements, then place          // root of heap at target index and add arr[i]          // to Min Heap          if (i < n)              arr[ti] = hp.replaceMin(arr[i]);            // Otherwise place root at its target index and          // reduce heap size          else              arr[ti] = hp.extractMin();      }  }    // FOLLOWING ARE IMPLEMENTATIONS OF STANDARD MIN HEAP METHODS FROM CORMEN BOOK  // Constructor: Builds a heap from a given array a[] of given size  MinHeap::MinHeap(int a[], int size)  {      heap\_size = size;      harr = a;  // store address of array      int i = (heap\_size - 1)/2;      while (i >= 0)      {          MinHeapify(i);          i--;      }  }    // Method to remove minimum element (or root) from min heap  int MinHeap::extractMin()  {      int root = harr[0];      if (heap\_size > 1)      {          harr[0] = harr[heap\_size-1];          heap\_size--;          MinHeapify(0);      }      return root;  }    // Method to change root with given value x, and return the old root  int MinHeap::replaceMin(int x)  {      int root = harr[0];      harr[0] = x;      if (root < x)          MinHeapify(0);      return root;  }    // A recursive method to heapify a subtree with root at given index  // This method assumes that the subtrees are already heapified  void MinHeap::MinHeapify(int i)  {      int l = left(i);      int r = right(i);      int smallest = i;      if (l < heap\_size && harr[l] < harr[i])          smallest = l;      if (r < heap\_size && harr[r] < harr[smallest])          smallest = r;      if (smallest != i)      {          swap(&harr[i], &harr[smallest]);          MinHeapify(smallest);      }  }    // A utility function to swap two elements  void swap(int \*x, int \*y)  {      int temp = \*x;      \*x = \*y;      \*y = temp;  }    // A utility function to print array elements  void printArray(int arr[], int size)  {     for (int i=0; i < size; i++)         cout << arr[i] << " ";     cout << endl;  }    // Driver program to test above functions  int main()  {      int k = 3;      int arr[] = {2, 6, 3, 12, 56, 8};      int n = sizeof(arr)/sizeof(arr[0]);      sortK(arr, n, k);        cout << "Following is sorted array\n";      printArray (arr, n);        return 0;  } |

Run on IDE

Output:

Following is sorted array

2 3 6 8 12 56

The Min Heap based method takes O(nLogk) time and uses O(k) auxiliary space.

We can also **use a Balanced Binary Search Tree** instead of Heap to store K+1 elements. The [insert](http://www.geeksforgeeks.org/archives/17679)and [delete](http://www.geeksforgeeks.org/archives/18009)operations on Balanced BST also take O(Logk) time. So Balanced BST based method will also take O(nLogk) time, but the Heap bassed method seems to be more efficient as the minimum element will always be at root. Also, Heap doesn’t need extra space for left and right pointers.

Iterative Quick Sort

Following is a typical recursive implementation of [Quick Sort](http://geeksquiz.com/quick-sort/) that uses last element as pivot.

* C++
* Python
* Java

/\* A typical recursive C/C++ implementation of QuickSort \*/

/\* This function takes last element as pivot, places

the pivot element at its correct position in sorted

array, and places all smaller (smaller than pivot)

to left of pivot and all greater elements to right

of pivot \*/

int partition (int arr[], int l, int h)

{

int x = arr[h];

int i = (l - 1);

for (int j = l; j <= h- 1; j++)

{

if (arr[j] <= x)

{

i++;

swap (&arr[i], &arr[j]);

}

}

swap (&arr[i + 1], &arr[h]);

return (i + 1);

}

/\* A[] --> Array to be sorted,

l --> Starting index,

h --> Ending index \*/

void quickSort(int A[], int l, int h)

{

if (l < h)

{

/\* Partitioning index \*/

int p = partition(A, l, h);

quickSort(A, l, p - 1);

quickSort(A, p + 1, h);

}

}

The above implementation can be optimized in many ways

1) The above implementation uses last index as pivot. This causes worst-case behavior on already sorted arrays, which is a commonly occurring case. The problem can be solved by choosing either a random index for the pivot, or choosing the middle index of the partition or choosing the median of the first, middle and last element of the partition for the pivot. (See [this](http://www.geeksforgeeks.org/archives/10069)for details)

2) To reduce the recursion depth, recur first for the smaller half of the array, and use a tail call to recurse into the other.

3) Insertion sort works better for small subarrays. Insertion sort can be used for invocations on such small arrays (i.e. where the length is less than a threshold t determined experimentally). For example, [this](http://code.google.com/p/dexandroid/source/browse/trunk/bionic/libc/stdlib/qsort.c?r=2) library implementation of qsort uses insertion sort below size 7.

Despite above optimizations, the function remains recursive and uses [function call stack](http://en.wikipedia.org/wiki/Call_stack) to store intermediate values of l and h. The function call stack stores other bookkeeping information together with parameters. Also, function calls involve overheads like storing activation record of the caller function and then resuming execution.

The above function can be easily converted to iterative version with the help of an auxiliary stack. Following is an iterative implementation of the above recursive code.

* C/C++
* Python
* Java

// An iterative implementation of quick sort

#include <stdio.h>

// A utility function to swap two elements

void swap ( int\* a, int\* b )

{

int t = \*a;

\*a = \*b;

\*b = t;

}

/\* This function is same in both iterative and recursive\*/

int partition (int arr[], int l, int h)

{

int x = arr[h];

int i = (l - 1);

for (int j = l; j <= h- 1; j++)

{

if (arr[j] <= x)

{

i++;

swap (&arr[i], &arr[j]);

}

}

swap (&arr[i + 1], &arr[h]);

return (i + 1);

}

/\* A[] --> Array to be sorted,

l --> Starting index,

h --> Ending index \*/

void quickSortIterative (int arr[], int l, int h)

{

// Create an auxiliary stack

int stack[ h - l + 1 ];

// initialize top of stack

int top = -1;

// push initial values of l and h to stack

stack[ ++top ] = l;

stack[ ++top ] = h;

// Keep popping from stack while is not empty

while ( top >= 0 )

{

// Pop h and l

h = stack[ top-- ];

l = stack[ top-- ];

// Set pivot element at its correct position

// in sorted array

int p = partition( arr, l, h );

// If there are elements on left side of pivot,

// then push left side to stack

if ( p-1 > l )

{

stack[ ++top ] = l;

stack[ ++top ] = p - 1;

}

// If there are elements on right side of pivot,

// then push right side to stack

if ( p+1 < h )

{

stack[ ++top ] = p + 1;

stack[ ++top ] = h;

}

}

}

// A utility function to print contents of arr

void printArr( int arr[], int n )

{

int i;

for ( i = 0; i < n; ++i )

printf( "%d ", arr[i] );

}

// Driver program to test above functions

int main()

{

int arr[] = {4, 3, 5, 2, 1, 3, 2, 3};

int n = sizeof( arr ) / sizeof( \*arr );

quickSortIterative( arr, 0, n - 1 );

printArr( arr, n );

return 0;

}

Output:

1 2 2 3 3 3 4 5

The above mentioned optimizations for recursive quick sort can also be applied to iterative version.

1) Partition process is same in both recursive and iterative. The same techniques to choose optimal pivot can also be applied to iterative version.

2) To reduce the stack size, first push the indexes of smaller half.

3) Use insertion sort when the size reduces below a experimentally calculated threshold.

**References:**  
<http://en.wikipedia.org/wiki/Quicksort>

QuickSort on Singly Linked List

[QuickSort on Doubly Linked List](http://www.geeksforgeeks.org/quicksort-for-linked-list/) is discussed [here](http://www.geeksforgeeks.org/quicksort-for-linked-list/). QuickSort on Singly linked list was given as an exercise. Following is C++ implementation for same. The important things about implementation are, it changes pointers rather swapping data and time complexity is same as the implementation for Doubly Linked List.  
In **partition()**, we consider last element as pivot. We traverse through the current list and if a node has value greater than pivot, we move it after tail. If the node has smaller value, we keep it at its current position.  
In **QuickSortRecur()**, we first call partition() which places pivot at correct position and returns pivot. After pivot is placed at correct position, we find tail node of left side (list before pivot) and recur for left list. Finally, we recur for right list.

|  |
| --- |
| // C++ program for Quick Sort on Singly Linled List  #include <iostream>  #include <cstdio>  using namespace std;    /\* a node of the singly linked list \*/  struct node  {      int data;      struct node \*next;  };    /\* A utility function to insert a node at the beginning of linked list \*/  void push(struct node\*\* head\_ref, int new\_data)  {      /\* allocate node \*/      struct node\* new\_node = new node;        /\* put in the data  \*/      new\_node->data  = new\_data;        /\* link the old list off the new node \*/      new\_node->next = (\*head\_ref);        /\* move the head to point to the new node \*/      (\*head\_ref)    = new\_node;  }    /\* A utility function to print linked list \*/  void printList(struct node \*node)  {      while (node != NULL)      {          printf("%d  ", node->data);          node = node->next;      }      printf("\n");  }    // Returns the last node of the list  struct node \*getTail(struct node \*cur)  {      while (cur != NULL && cur->next != NULL)          cur = cur->next;      return cur;  }    // Partitions the list taking the last element as the pivot  struct node \*partition(struct node \*head, struct node \*end,                         struct node \*\*newHead, struct node \*\*newEnd)  {      struct node \*pivot = end;      struct node \*prev = NULL, \*cur = head, \*tail = pivot;        // During partition, both the head and end of the list might change      // which is updated in the newHead and newEnd variables      while (cur != pivot)      {          if (cur->data < pivot->data)          {              // First node that has a value less than the pivot - becomes              // the new head              if ((\*newHead) == NULL)                  (\*newHead) = cur;                prev = cur;              cur = cur->next;          }          else // If cur node is greater than pivot          {              // Move cur node to next of tail, and change tail              if (prev)                  prev->next = cur->next;              struct node \*tmp = cur->next;              cur->next = NULL;              tail->next = cur;              tail = cur;              cur = tmp;          }      }        // If the pivot data is the smallest element in the current list,      // pivot becomes the head      if ((\*newHead) == NULL)          (\*newHead) = pivot;        // Update newEnd to the current last node      (\*newEnd) = tail;        // Return the pivot node      return pivot;  }      //here the sorting happens exclusive of the end node  struct node \*quickSortRecur(struct node \*head, struct node \*end)  {      // base condition      if (!head || head == end)          return head;        node \*newHead = NULL, \*newEnd = NULL;        // Partition the list, newHead and newEnd will be updated      // by the partition function      struct node \*pivot = partition(head, end, &newHead, &newEnd);        // If pivot is the smallest element - no need to recur for      // the left part.      if (newHead != pivot)      {          // Set the node before the pivot node as NULL          struct node \*tmp = newHead;          while (tmp->next != pivot)              tmp = tmp->next;          tmp->next = NULL;            // Recur for the list before pivot          newHead = quickSortRecur(newHead, tmp);            // Change next of last node of the left half to pivot          tmp = getTail(newHead);          tmp->next =  pivot;      }        // Recur for the list after the pivot element      pivot->next = quickSortRecur(pivot->next, newEnd);        return newHead;  }    // The main function for quick sort. This is a wrapper over recursive  // function quickSortRecur()  void quickSort(struct node \*\*headRef)  {      (\*headRef) = quickSortRecur(\*headRef, getTail(\*headRef));      return;  }    // Driver program to test above functions  int main()  {      struct node \*a = NULL;      push(&a, 5);      push(&a, 20);      push(&a, 4);      push(&a, 3);      push(&a, 30);        cout << "Linked List before sorting \n";      printList(a);        quickSort(&a);        cout << "Linked List after sorting \n";      printList(a);        return 0;  } |

Run on IDE

Output:

Linked List before sorting

30 3 4 20 5

Linked List after sorting

3 4 5 20 30

This article is contributed by [**Balasubramanian.N**](http://in.linkedin.com/pub/balasubramanian-nagasundaram/3a/361/97b). Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

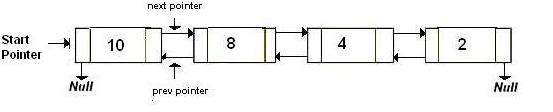
QuickSort on Doubly Linked List

Following is a typical recursive implementation of [QuickSort](http://en.wikipedia.org/wiki/Quicksort) for arrays. The implementation uses last element as pivot.

|  |
| --- |
| /\* A typical recursive implementation of Quicksort for array\*/    /\* This function takes last element as pivot, places the pivot element at its     correct position in sorted array, and places all smaller (smaller than     pivot) to left of pivot and all greater elements to right of pivot \*/  int partition (int arr[], int l, int h)  {      int x = arr[h];      int i = (l - 1);        for (int j = l; j <= h- 1; j++)      {          if (arr[j] <= x)          {              i++;              swap (&arr[i], &arr[j]);          }      }      swap (&arr[i + 1], &arr[h]);      return (i + 1);  }    /\* A[] --> Array to be sorted, l  --> Starting index, h  --> Ending index \*/  void quickSort(int A[], int l, int h)  {      if (l < h)      {          int p = partition(A, l, h); /\* Partitioning index \*/          quickSort(A, l, p - 1);          quickSort(A, p + 1, h);      }  } |

Run on IDE

**Can we use same algorithm for Linked List?**  
Following is C++ implementation for doubly linked list. The idea is simple, we first find out pointer to last node. Once we have pointer to last node, we can recursively sort the linked list using pointers to first and last nodes of linked list, similar to the above recursive function where we pass indexes of first and last array elements. The partition function for linked list is also similar to partition for arrays. Instead of returning index of the pivot element, it returns pointer to the pivot element. In the following implementation, quickSort() is just a wrapper function, the main recursive function is \_quickSort() which is similar to quickSort() for array implementation.



* C++
* Java

|  |
| --- |
| // A C++ program to sort a linked list using Quicksort  #include <iostream>  #include <stdio.h>  using namespace std;    /\* a node of the doubly linked list \*/  struct node  {      int data;      struct node \*next;      struct node \*prev;  };    /\* A utility function to swap two elements \*/  void swap ( int\* a, int\* b )  {   int t = \*a;      \*a = \*b;       \*b = t;   }    // A utility function to find last node of linked list  struct node \*lastNode(node \*root)  {      while (root && root->next)          root = root->next;      return root;  }    /\* Considers last element as pivot, places the pivot element at its     correct position in sorted array, and places all smaller (smaller than     pivot) to left of pivot and all greater elements to right of pivot \*/  node\* partition(node \*l, node \*h)  {      // set pivot as h element      int x  = h->data;        // similar to i = l-1 for array implementation      node \*i = l->prev;        // Similar to "for (int j = l; j <= h- 1; j++)"      for (node \*j = l; j != h; j = j->next)      {          if (j->data <= x)          {              // Similar to i++ for array              i = (i == NULL)? l : i->next;                swap(&(i->data), &(j->data));          }      }      i = (i == NULL)? l : i->next; // Similar to i++      swap(&(i->data), &(h->data));      return i;  }    /\* A recursive implementation of quicksort for linked list \*/  void \_quickSort(struct node\* l, struct node \*h)  {      if (h != NULL && l != h && l != h->next)      {          struct node \*p = partition(l, h);          \_quickSort(l, p->prev);          \_quickSort(p->next, h);      }  }    // The main function to sort a linked list. It mainly calls \_quickSort()  void quickSort(struct node \*head)  {      // Find last node      struct node \*h = lastNode(head);        // Call the recursive QuickSort      \_quickSort(head, h);  }    // A utility function to print contents of arr  void printList(struct node \*head)  {      while (head)      {          cout << head->data << "  ";          head = head->next;      }      cout << endl;  }    /\* Function to insert a node at the beginging of the Doubly Linked List \*/  void push(struct node\*\* head\_ref, int new\_data)  {      struct node\* new\_node = new node;     /\* allocate node \*/      new\_node->data  = new\_data;        /\* since we are adding at the begining, prev is always NULL \*/      new\_node->prev = NULL;        /\* link the old list off the new node \*/      new\_node->next = (\*head\_ref);        /\* change prev of head node to new node \*/      if ((\*head\_ref) !=  NULL)  (\*head\_ref)->prev = new\_node ;        /\* move the head to point to the new node \*/      (\*head\_ref)    = new\_node;  }    /\* Driver program to test above function \*/  int main()  {      struct node \*a = NULL;      push(&a, 5);      push(&a, 20);      push(&a, 4);      push(&a, 3);      push(&a, 30);        cout << "Linked List before sorting \n";      printList(a);        quickSort(a);        cout << "Linked List after sorting \n";      printList(a);        return 0;  } |

Run on IDE

Output :

Linked List before sorting

30 3 4 20 5

Linked List after sorting

3 4 5 20 30

**Time Complexity:**Time complexity of the above implementation is same as time complexity of QuickSort() for arrays. It takes O(n^2) time in worst case and O(nLogn) in average and best cases. The worst case occurs when the linked list is already sorted.

Can we implement random quick sort for linked list?  
Quicksort can be implemented for Linked List only when we can pick a fixed point as pivot (like last element in above implementation). Random QuickSort cannot be efficiently implemented for Linked Lists by picking random pivot.

**Exercise:**  
The above implementation is for doubly linked list. Modify it for singly linked list. Note that we don’t have prev pointer in singly linked list.  
Refer [QuickSort on Singly Linked List](http://www.geeksforgeeks.org/quicksort-on-singly-linked-list/)for solution.

Find k closest elements to a given value

Given a sorted array arr[] and a value X, find the k closest elements to X in arr[].   
Examples:

Input: K = 4, X = 35

arr[] = {12, 16, 22, 30, 35, 39, 42,

45, 48, 50, 53, 55, 56}

Output: 30 39 42 45

Note that if the element is present in array, then it should not be in output, only the other closest elements are required.

In the following solutions, it is assumed that all elements of array are distinct.

A **simple solution**is to do linear search for k closest elements.  
1) Start from the first element and search for the crossover point (The point before which elements are smaller than or equal to X and after which elements are greater). This step takes O(n) time.  
2) Once we find the crossover point, we can compare elements on both sides of crossover point to print k closest elements. This step takes O(k) time.

The time complexity of the above solution is O(n).

An **Optimized Solution** is to find k elements in O(Logn + k) time. The idea is to use [Binary Search](http://geeksquiz.com/binary-search/) to find the crossover point. Once we find index of crossover point, we can print k closest elements in O(k) time.

* C/C++
* Java

|  |
| --- |
| #include<stdio.h>    /\* Function to find the cross over point (the point before     which elements are smaller than or equal to x and after     which greater than x)\*/  int findCrossOver(int arr[], int low, int high, int x)  {    // Base cases    if (arr[high] <= x) // x is greater than all      return high;    if (arr[low] > x)  // x is smaller than all      return low;      // Find the middle point    int mid = (low + high)/2;  /\* low + (high - low)/2 \*/      /\* If x is same as middle element, then return mid \*/    if (arr[mid] <= x && arr[mid+1] > x)      return mid;      /\* If x is greater than arr[mid], then either arr[mid + 1]      is ceiling of x or ceiling lies in arr[mid+1...high] \*/    if(arr[mid] < x)        return findCrossOver(arr, mid+1, high, x);      return findCrossOver(arr, low, mid - 1, x);  }    // This function prints k closest elements to x in arr[].  // n is the number of elements in arr[]  void printKclosest(int arr[], int x, int k, int n)  {      // Find the crossover point      int l = findCrossOver(arr, 0, n-1, x);      int r = l+1;   // Right index to search      int count = 0; // To keep track of count of elements already printed        // If x is present in arr[], then reduce left index      // Assumption: all elements in arr[] are distinct      if (arr[l] == x) l--;        // Compare elements on left and right of crossover      // point to find the k closest elements      while (l >= 0 && r < n && count < k)      {          if (x - arr[l] < arr[r] - x)              printf("%d ", arr[l--]);          else              printf("%d ", arr[r++]);          count++;      }        // If there are no more elements on right side, then      // print left elements      while (count < k && l >= 0)          printf("%d ", arr[l--]), count++;        // If there are no more elements on left side, then      // print right elements      while (count < k && r < n)          printf("%d ", arr[r++]), count++;  }    /\* Driver program to check above functions \*/  int main()  {     int arr[] ={12, 16, 22, 30, 35, 39, 42,                 45, 48, 50, 53, 55, 56};     int n = sizeof(arr)/sizeof(arr[0]);     int x = 35, k = 4;     printKclosest(arr, x, 4, n);     return 0;  } |

Run on IDE

Output:

39 30 42 45

The time complexity of this method is O(Logn + k).

**Exercise:** Extend the optimized solution to work for duplicates also, i.e., to work for arrays where elements don’t have to be distinct.

Sort n numbers in range from 0 to n^2 – 1 in linear time

Given an array of numbers of size n. It is also given that the array elements are in range from 0 to n2 – 1. Sort the given array in linear time.

Examples:

Since there are 5 elements, the elements can be from 0 to 24.

Input: arr[] = {0, 23, 14, 12, 9}

Output: arr[] = {0, 9, 12, 14, 23}

Since there are 3 elements, the elements can be from 0 to 8.

Input: arr[] = {7, 0, 2}

Output: arr[] = {0, 2, 7}

***We strongly recommend to minimize the browser and try this yourself first.***

**Solution:** If we use [Counting Sort](http://www.geeksforgeeks.org/counting-sort/), it would take O(n^2) time as the given range is of size n^2. Using any comparison based sorting like[Merge Sort](http://geeksquiz.com/merge-sort/), [Heap Sort](http://geeksquiz.com/heap-sort/), .. etc would take O(nLogn) time.  
Now question arises how to do this in 0(n)? Firstly, is it possible? Can we use data given in question? n numbers in range from 0 to n2 – 1?  
The idea is to use [Radix Sort](http://www.geeksforgeeks.org/radix-sort/). Following is standard Radix Sort algorithm.

1) Do following for each digit i where i varies from least

significant digit to the most significant digit.

…………..a) Sort input array using counting sort (or any stable

sort) according to the i’th digit

Let there be d digits in input integers. Radix Sort takes O(d\*(n+b)) time where b is the base for representing numbers, for example, for decimal system, b is 10. Since n2-1 is the maximum possible value, the value of d would be O(logb(n)). So overall time complexity is O((n+b)\*O(logb(n)). Which looks more than the time complexity of comparison based sorting algorithms for a large k. The idea is to change base b. If we set b as n, the value of O(logb(n)) becomes O(1) and overall time complexity becomes O(n).

arr[] = {0, 10, 13, 12, 7}

Let us consider the elements in base 5. For example 13 in

base 5 is 23, and 7 in base 5 is 12.

arr[] = {00(0), 20(10), 23(13), 22(12), 12(7)}

After first iteration (Sorting according to the last digit in

base 5), we get.

arr[] = {00(0), 20(10), 12(7), 22(12), 23(13)}

After second iteration, we get

arr[] = {00(0), 12(7), 20(10), 22(12), 23(13)}

Following is C++ implementation to sort an array of size n where elements are in range from 0 to n2 – 1.

* C++
* Java

|  |
| --- |
| #include<iostream>  using namespace std;    // A function to do counting sort of arr[] according to  // the digit represented by exp.  int countSort(int arr[], int n, int exp)  {        int output[n]; // output array      int i, count[n] ;      for (int i=0; i < n; i++)         count[i] = 0;        // Store count of occurrences in count[]      for (i = 0; i < n; i++)          count[ (arr[i]/exp)%n ]++;        // Change count[i] so that count[i] now contains actual      // position of this digit in output[]      for (i = 1; i < n; i++)          count[i] += count[i - 1];        // Build the output array      for (i = n - 1; i >= 0; i--)      {          output[count[ (arr[i]/exp)%n] - 1] = arr[i];          count[(arr[i]/exp)%n]--;      }        // Copy the output array to arr[], so that arr[] now      // contains sorted numbers according to curent digit      for (i = 0; i < n; i++)          arr[i] = output[i];  }      // The main function to that sorts arr[] of size n using Radix Sort  void sort(int arr[], int n)  {      // Do counting sort for first digit in base n. Note that      // instead of passing digit number, exp (n^0 = 0) is passed.      countSort(arr, n, 1);        // Do counting sort for second digit in base n. Note that      // instead of passing digit number, exp (n^1 = n) is passed.      countSort(arr, n, n);  }    // A utility function to print an array  void printArr(int arr[], int n)  {      for (int i = 0; i < n; i++)          cout << arr[i] << " ";  }    // Driver program to test above functions  int main()  {      // Since array size is 7, elements should be from 0 to 48      int arr[] = {40, 12, 45, 32, 33, 1, 22};      int n = sizeof(arr)/sizeof(arr[0]);      cout << "Given array is \n";      printArr(arr, n);        sort(arr, n);        cout << "\nSorted array is \n";      printArr(arr, n);      return 0;  } |

Run on IDE

Output:

Given array is

40 12 45 32 33 1 22

Sorted array is

1 12 22 32 33 40 45

**How to sort if range is from 1 to n2?**  
If range is from 1 to n n2, the above process can not be directly applied, it must be changed. Consider n = 100 and range from 1 to 10000. Since the base is 100, a digit must be from 0 to 99 and there should be 2 digits in the numbers. But the number 10000 has more than 2 digits. So to sort numbers in a range from 1 to n2, we can use following process.  
1) Subtract all numbers by 1.  
2) Since the range is now 0 to n2, do counting sort twice as done in the above implementation.  
3) After the elements are sorted, add 1 to all numbers to obtain the original numbers.

**How to sort if range is from 0 to n^3 -1?**  
Since there can be 3 digits in base n, we need to call counting sort 3 times.

A Problem in Many Binary Search Implementations

Consider the following C implementation of [Binary Search](http://geeksquiz.com/binary-search/) function, is there anything wrong in this?

|  |
| --- |
| // A iterative binary search function. It returns location of x in  // given array arr[l..r] if present, otherwise -1  int binarySearch(int arr[], int l, int r, int x)  {      while (l <= r)      {          // find index of middle element          int m = (l+r)/2;            // Check if x is present at mid          if (arr[m] == x) return m;            // If x greater, ignore left half          if (arr[m] < x) l = m + 1;            // If x is smaller, ignore right half          else r = m - 1;      }        // if we reach here, then element was not present      return -1;  } |

Run on IDE

The above looks fine except one subtle thing, the expression “m = (l+r)/2”. It fails for large values of l and r. Specifically, it fails if the sum of low and high is greater than the maximum positive int value (231 – 1). The sum overflows to a negative value, and the value stays negative when divided by two. In C this causes an array index out of bounds with unpredictable results.

**What is the way to resolve this problem?**  
Following is one way:

int mid = low + ((high - low) / 2);

Probably faster, and arguably as clear is (works only in Java, refer [this](http://www.geeksforgeeks.org/bitwise-shift-operators-in-java/)):

int mid = (low + high) >>> 1;

In C and C++ (where you don’t have the >>> operator), you can do this:

mid = ((unsigned int)low + (unsigned int)high)) >> 1

The similar problem appears in [Merge Sort](http://geeksquiz.com/merge-sort/) as well.

The above content is taken from [google reasearch blog](http://googleresearch.blogspot.in/2006/06/extra-extra-read-all-about-it-nearly.html).

Please refer [this](http://locklessinc.com/articles/binary_search/)as well, it points out that the above solutions may not always work.

The above problem occurs when array length is 230 or greater and the search repeatedly moves to second half of the array. This much size of array is not likely to appear most of the time. For example, when we try the below program with 32 bit [Code Blocks](http://www.codeblocks.org/) compiler, we get compiler error.

|  |
| --- |
| int main()  {      int arr[1<<30];      return 0;  } |

Run on IDE

Output:

error: size of array 'arr' is too large

Even when we try boolean array, the program compiles fine, but crashes when run in Windows 7.0 and [Code Blocks](http://www.codeblocks.org/) 32 bit compiler

|  |
| --- |
| #include <stdbool.h>  int main()  {      bool arr[1<<30];      return 0;  } |

Run on IDE

Output: No compiler error, but crashes at run time.

**Sources:**  
<http://googleresearch.blogspot.in/2006/06/extra-extra-read-all-about-it-nearly.html>  
<http://locklessinc.com/articles/binary_search/>

Search in an almost sorted array

Given an array which is sorted, but after sorting some elements are moved to either of the adjacent positions, i.e., arr[i] may be present at arr[i+1] or arr[i-1]. Write an efficient function to search an element in this array. Basically the element arr[i] can only be swapped with either arr[i+1] or arr[i-1].

For example consider the array {2, 3, 10, 4, 40}, 4 is moved to next position and 10 is moved to previous position.

Example:

Input: arr[] = {10, 3, 40, 20, 50, 80, 70}, key = 40

Output: 2

Output is index of 40 in given array

Input: arr[] = {10, 3, 40, 20, 50, 80, 70}, key = 90

Output: -1

-1 is returned to indicate element is not present

A simple solution is to linearly search the given key in given array. Time complexity of this solution is O(n). We cab modify [binary search](http://geeksquiz.com/binary-search/) to do it in O(Logn) time.

The idea is to compare the key with middle 3 elements, if present then return the index. If not present, then compare the key with middle element to decide whether to go in left half or right half. Comparing with middle element is enough as all the elements after mid+2 must be greater than element mid and all elements before mid-2 must be smaller than mid element.

Following is C++ implementation of this approach.

* C++
* Java

|  |
| --- |
| // C++ program to find an element in an almost sorted  // array  #include <stdio.h>    // A recursive binary search based function. It returns  // index of x in given array arr[l..r] is present,  // otherwise -1  int binarySearch(int arr[], int l, int r, int x)  {     if (r >= l)     {          int mid = l + (r - l)/2;            // If the element is present at one of the middle          // 3 positions          if (arr[mid] == x)  return mid;          if (mid > l && arr[mid-1] == x) return (mid - 1);          if (mid < r && arr[mid+1] == x) return (mid + 1);            // If element is smaller than mid, then it can only          // be present in left subarray          if (arr[mid] > x) return binarySearch(arr, l, mid-2, x);            // Else the element can only be present in right subarray          return binarySearch(arr, mid+2, r, x);     }       // We reach here when element is not present in array     return -1;  }    // Driver program to test above function  int main(void)  {     int arr[] = {3, 2, 10, 4, 40};     int n = sizeof(arr)/ sizeof(arr[0]);     int x = 4;     int result = binarySearch(arr, 0, n-1, x);     (result == -1)? printf("Element is not present in array")                   : printf("Element is present at index %d", result);     return 0;  } |

Output:

Element is present at index 3

Time complexity of the above function is O(Logn).

# Sort an array in wave form

Given an unsorted array of integers, sort the array into a wave like array. An array ‘arr[0..n-1]’ is sorted in wave form if arr[0] >= arr[1] <= arr[2] >= arr[3] <= arr[4] >= …..

Examples:

Input: arr[] = {10, 5, 6, 3, 2, 20, 100, 80}

Output: arr[] = {10, 5, 6, 2, 20, 3, 100, 80} OR

{20, 5, 10, 2, 80, 6, 100, 3} OR

any other array that is in wave form

Input: arr[] = {20, 10, 8, 6, 4, 2}

Output: arr[] = {20, 8, 10, 4, 6, 2} OR

{10, 8, 20, 2, 6, 4} OR

any other array that is in wave form

Input: arr[] = {2, 4, 6, 8, 10, 20}

Output: arr[] = {4, 2, 8, 6, 20, 10} OR

any other array that is in wave form

Input: arr[] = {3, 6, 5, 10, 7, 20}

Output: arr[] = {6, 3, 10, 5, 20, 7} OR

any other array that is in wave form

## [We strongly recommend that you click here and practice it, before moving on to the solution.](http://www.practice.geeksforgeeks.org/problem-page.php?pid=125)

A **Simple Solution** is to use sorting. First sort the input array, then swap all adjacent elements.

For example, let the input array be {3, 6, 5, 10, 7, 20}. After sorting, we get {3, 5, 6, 7, 10, 20}. After swapping adjacent elements, we get {5, 3, 7, 6, 20, 10}.

Below are implementations of this simple approach.

* C++
* Python
* Java

|  |
| --- |
| // A C++ program to sort an array in wave form using  // a sorting function  #include<iostream>  #include<algorithm>  using namespace std;    // A utility method to swap two numbers.  void swap(int \*x, int \*y)  {      int temp = \*x;      \*x = \*y;      \*y = temp;  }    // This function sorts arr[0..n-1] in wave form, i.e.,  // arr[0] >= arr[1] <= arr[2] >= arr[3] <= arr[4] >= arr[5]..  void sortInWave(int arr[], int n)  {      // Sort the input array      sort(arr, arr+n);        // Swap adjacent elements      for (int i=0; i<n-1; i += 2)          swap(&arr[i], &arr[i+1]);  }    // Driver program to test above function  int main()  {      int arr[] = {10, 90, 49, 2, 1, 5, 23};      int n = sizeof(arr)/sizeof(arr[0]);      sortInWave(arr, n);      for (int i=0; i<n; i++)         cout << arr[i] << " ";      return 0;  } |

Run on IDE

Output:

2 1 10 5 49 23 90

The time complexity of the above solution is O(nLogn) if a O(nLogn) sorting algorithm like [Merge Sort](http://geeksquiz.com/merge-sort/), [Heap Sort](http://geeksquiz.com/heap-sort/), .. etc is used.

This can be done in **O(n) time by doing a single traversal** of given array. The idea is based on the fact that if we make sure that all even positioned (at index 0, 2, 4, ..) elements are greater than their adjacent odd elements, we don’t need to worry about odd positioned element. Following are simple steps.  
1) Traverse all even positioned elements of input array, and do following.  
….a) If current element is smaller than previous odd element, swap previous and current.  
….b) If current element is smaller than next odd element, swap next and current.

Below are implementations of above simple algorithm.

* C++
* Python
* Java

|  |
| --- |
| // A O(n) program to sort an input array in wave form  #include<iostream>  using namespace std;    // A utility method to swap two numbers.  void swap(int \*x, int \*y)  {      int temp = \*x;      \*x = \*y;      \*y = temp;  }    // This function sorts arr[0..n-1] in wave form, i.e., arr[0] >=  // arr[1] <= arr[2] >= arr[3] <= arr[4] >= arr[5] ....  void sortInWave(int arr[], int n)  {      // Traverse all even elements      for (int i = 0; i < n; i+=2)      {          // If current even element is smaller than previous          if (i>0 && arr[i-1] > arr[i] )              swap(&arr[i], &arr[i-1]);            // If current even element is smaller than next          if (i<n-1 && arr[i] < arr[i+1] )              swap(&arr[i], &arr[i + 1]);      }  }    // Driver program to test above function  int main()  {      int arr[] = {10, 90, 49, 2, 1, 5, 23};      int n = sizeof(arr)/sizeof(arr[0]);      sortInWave(arr, n);      for (int i=0; i<n; i++)         cout << arr[i] << " ";      return 0;  } |

Run on IDE

Output:

90 10 49 1 5 2 23

Why is Binary Search preferred over Ternary Search?

The following is a simple recursive **Binary Search** function in C++ taken from [here](http://geeksquiz.com/binary-search/).

|  |
| --- |
| // A recursive binary search function. It returns location of x in  // given array arr[l..r] is present, otherwise -1  int binarySearch(int arr[], int l, int r, int x)  {     if (r >= l)     {          int mid = l + (r - l)/2;            // If the element is present at the middle itself          if (arr[mid] == x)  return mid;            // If element is smaller than mid, then it can only be present          // in left subarray          if (arr[mid] > x) return binarySearch(arr, l, mid-1, x);            // Else the element can only be present in right subarray          return binarySearch(arr, mid+1, r, x);     }       // We reach here when element is not present in array     return -1;  } |

Run on IDE

The following is a simple recursive **Ternary Search** function in C++.

|  |
| --- |
| // A recursive ternary search function. It returns location of x in  // given array arr[l..r] is present, otherwise -1  int ternarySearch(int arr[], int l, int r, int x)  {     if (r >= l)     {          int mid1 = l + (r - l)/3;          int mid2 = mid1 + (r - l)/3;            // If x is present at the mid1          if (arr[mid1] == x)  return mid1;            // If x is present at the mid2          if (arr[mid2] == x)  return mid2;            // If x is present in left one-third          if (arr[mid1] > x) return ternarySearch(arr, l, mid1-1, x);            // If x is present in right one-third          if (arr[mid2] < x) return ternarySearch(arr, mid2+1, r, x);            // If x is present in middle one-third          return ternarySearch(arr, mid1+1, mid2-1, x);     }     // We reach here when element is not present in array     return -1;  } |

Run on IDE

**Which of the above two does less comparisons in worst case?**  
From the first look, it seems the ternary search does less number of comparisons as it makes Log3n recursive calls, but binary search makes Log2n recursive calls. Let us take a closer look.  
The following is recursive formula for counting comparisons in worst case of Binary Search.

T(n) = T(n/2) + 2, T(1) = 1

The following is recursive formula for counting comparisons in worst case of Ternary Search.

T(n) = T(n/3) + 4, T(1) = 1

In binary search, there are 2Log2n + 1 comparisons in worst case. In ternary search, there are 4Log3n + 1 comparisons in worst case.

Time Complexity for Binary search = 2clog2n + O(1)

Time Complexity for Ternary search = 4clog3n + O(1)

Therefore, the comparison of Ternary and Binary Searches boils down the comparison of expressions 2Log3n and Log2n . The value of 2Log3n can be written as (2 / Log23) \* Log2n . Since the value of (2 / Log23) is more than one, Ternary Search does more comparisons than Binary Search in worst case.

**Exercise:**  
Why Merge Sort divides input array in two halves, why not in three or more parts?

K’th Smallest/Largest Element in Unsorted Array | Set 2 (Expected Linear Time)

We recommend to read following post as a prerequisite of this post.

[K’th Smallest/Largest Element in Unsorted Array | Set 1](http://www.geeksforgeeks.org/kth-smallestlargest-element-unsorted-array/)

Given an array and a number k where k is smaller than size of array, we need to find the k’th smallest element in the given array. It is given that ll array elements are distinct.

Examples:

Input: arr[] = {7, 10, 4, 3, 20, 15}

k = 3

Output: 7

Input: arr[] = {7, 10, 4, 3, 20, 15}

k = 4

Output: 10

We have discussed three different solutions [here](http://www.geeksforgeeks.org/kth-smallestlargest-element-unsorted-array/).

In this post method 4 is discussed which is mainly an extension of method 3 (QuickSelect) discussed in the [previous](http://www.geeksforgeeks.org/kth-smallestlargest-element-unsorted-array/)post. The idea is to randomly pick a pivot element. To implement randomized partition, we use a random function, [rand()](http://www.cplusplus.com/reference/cstdlib/rand/) to generate index between l and r, swap the element at randomly generated index with the last element, and finally call the standard partition process which uses last element as pivot.

Following is implementation of above Randomized QuickSelect.

* C/C++
* Java

|  |
| --- |
| // C++ implementation of randomized quickSelect  #include<iostream>  #include<climits>  #include<cstdlib>  using namespace std;    int randomPartition(int arr[], int l, int r);    // This function returns k'th smallest element in arr[l..r] using  // QuickSort based method. ASSUMPTION: ELEMENTS IN ARR[] ARE DISTINCT  int kthSmallest(int arr[], int l, int r, int k)  {      // If k is smaller than number of elements in array      if (k > 0 && k <= r - l + 1)      {          // Partition the array around a random element and          // get position of pivot element in sorted array          int pos = randomPartition(arr, l, r);            // If position is same as k          if (pos-l == k-1)              return arr[pos];          if (pos-l > k-1)  // If position is more, recur for left subarray              return kthSmallest(arr, l, pos-1, k);            // Else recur for right subarray          return kthSmallest(arr, pos+1, r, k-pos+l-1);      }        // If k is more than number of elements in array      return INT\_MAX;  }    void swap(int \*a, int \*b)  {      int temp = \*a;      \*a = \*b;      \*b = temp;  }    // Standard partition process of QuickSort().  It considers the last  // element as pivot and moves all smaller element to left of it and  // greater elements to right. This function is used by randomPartition()  int partition(int arr[], int l, int r)  {      int x = arr[r], i = l;      for (int j = l; j <= r - 1; j++)      {          if (arr[j] <= x)          {              swap(&arr[i], &arr[j]);              i++;          }      }      swap(&arr[i], &arr[r]);      return i;  }    // Picks a random pivot element between l and r and partitions  // arr[l..r] arount the randomly picked element using partition()  int randomPartition(int arr[], int l, int r)  {      int n = r-l+1;      int pivot = rand() % n;      swap(&arr[l + pivot], &arr[r]);      return partition(arr, l, r);  }    // Driver program to test above methods  int main()  {      int arr[] = {12, 3, 5, 7, 4, 19, 26};      int n = sizeof(arr)/sizeof(arr[0]), k = 3;      cout << "K'th smallest element is " << kthSmallest(arr, 0, n-1, k);      return 0;  } |

Run on IDE

Output:

K'th smallest element is 5

**Time Complexity:**  
The worst case time complexity of the above solution is still O(n2). In worst case, the randomized function may always pick a corner element. The expected time complexity of above randomized QuickSelect is Θ(n), see [CLRS book](http://www.flipkart.com/introduction-algorithms-english-3rd/p/itmdwxyrafdburzg?pid=9788120340077&affid=sandeepgfg) or [MIT video lecture](http://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-046j-introduction-to-algorithms-sma-5503-fall-2005/video-lectures/lecture-6-order-statistics-median/) for proof. The assumption in the analysis is, random number generator is equally likely to generate any number in the input range.

**Sources:**  
[MIT Video Lecture on Order Statistics, Median](http://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-046j-introduction-to-algorithms-sma-5503-fall-2005/video-lectures/lecture-6-order-statistics-median/)  
[Introduction to Algorithms by Clifford Stein, Thomas H. Cormen, Charles E. Leiserson, Ronald L.](http://www.flipkart.com/introduction-algorithms-8120340078/p/itmczynzhyhxv2gs?pid=9788120340077&affid=sandeepgfg)

K’th Smallest/Largest Element in Unsorted Array | Set 3 (Worst Case Linear Time)

We recommend to read following posts as a prerequisite of this post.

[K’th Smallest/Largest Element in Unsorted Array | Set 1](http://www.geeksforgeeks.org/kth-smallestlargest-element-unsorted-array/)  
[K’th Smallest/Largest Element in Unsorted Array | Set 2 (Expected Linear Time)](http://www.geeksforgeeks.org/kth-smallestlargest-element-unsorted-array-set-2-expected-linear-time/)

Given an array and a number k where k is smaller than size of array, we need to find the k’th smallest element in the given array. It is given that ll array elements are distinct.

Examples:

Input: arr[] = {7, 10, 4, 3, 20, 15}

k = 3

Output: 7

Input: arr[] = {7, 10, 4, 3, 20, 15}

k = 4

Output: 10

In [previous post](http://www.geeksforgeeks.org/kth-smallestlargest-element-unsorted-array-set-2-expected-linear-time/), we discussed an expected linear time algorithm. In this post, a worst case linear time method is discussed. *The idea in this new method is similar to quickSelect(), we get worst case linear time by selecting a pivot that divides array in a balanced way (there are not very few elements on one side and many on other side)*. After the array is divided in a balanced way, we apply the same steps as used in quickSelect() to decide whether to go left or right of pivot.

Following is complete algorithm.

**kthSmallest(arr[0..n-1], k)**

**1)** Divide arr[] into ⌈n/5rceil; groups where size of each group is 5

except possibly the last group which may have less than 5 elements.

**2)** Sort the above created ⌈n/5⌉ groups and find median

of all groups. Create an auxiliary array 'median[]' and store medians

of all ⌈n/5⌉ groups in this median array.

// Recursively call this method to find median of median[0..⌈n/5⌉-1]

**3)** medOfMed = kthSmallest(median[0..⌈n/5⌉-1], ⌈n/10⌉)

**4)** Partition arr[] around medOfMed and obtain its position.

pos = partition(arr, n, medOfMed)

**5)** If pos == k return medOfMed

**6)** If pos < k return kthSmallest(arr[l..pos-1], k)

**7)** If poa > k return kthSmallest(arr[pos+1..r], k-pos+l-1)

In above algorithm, last 3 steps are same as algorithm in [previous post](http://www.geeksforgeeks.org/kth-smallestlargest-element-unsorted-array-set-2-expected-linear-time/). The first four steps are used to obtain a good point for partitioning the array (to make sure that there are not too many elements either side of pivot).

Following is C++ implementation of above algorithm.

|  |
| --- |
| // C++ implementation of worst case linear time algorithm  // to find k'th smallest element  #include<iostream>  #include<algorithm>  #include<climits>  using namespace std;    int partition(int arr[], int l, int r, int k);    // A simple function to find median of arr[].  This is called  // only for an array of size 5 in this program.  int findMedian(int arr[], int n)  {      sort(arr, arr+n);  // Sort the array      return arr[n/2];   // Return middle element  }    // Returns k'th smallest element in arr[l..r] in worst case  // linear time. ASSUMPTION: ALL ELEMENTS IN ARR[] ARE DISTINCT  int kthSmallest(int arr[], int l, int r, int k)  {      // If k is smaller than number of elements in array      if (k > 0 && k <= r - l + 1)      {          int n = r-l+1; // Number of elements in arr[l..r]            // Divide arr[] in groups of size 5, calculate median          // of every group and store it in median[] array.          int i, median[(n+4)/5]; // There will be floor((n+4)/5) groups;          for (i=0; i<n/5; i++)              median[i] = findMedian(arr+l+i\*5, 5);          if (i\*5 < n) //For last group with less than 5 elements          {              median[i] = findMedian(arr+l+i\*5, n%5);              i++;          }            // Find median of all medians using recursive call.          // If median[] has only one element, then no need          // of recursive call          int medOfMed = (i == 1)? median[i-1]:                                   kthSmallest(median, 0, i-1, i/2);            // Partition the array around a random element and          // get position of pivot element in sorted array          int pos = partition(arr, l, r, medOfMed);            // If position is same as k          if (pos-l == k-1)              return arr[pos];          if (pos-l > k-1)  // If position is more, recur for left              return kthSmallest(arr, l, pos-1, k);            // Else recur for right subarray          return kthSmallest(arr, pos+1, r, k-pos+l-1);      }        // If k is more than number of elements in array      return INT\_MAX;  }    void swap(int \*a, int \*b)  {      int temp = \*a;      \*a = \*b;      \*b = temp;  }    // It searches for x in arr[l..r], and partitions the array  // around x.  int partition(int arr[], int l, int r, int x)  {      // Search for x in arr[l..r] and move it to end      int i;      for (i=l; i<r; i++)          if (arr[i] == x)             break;      swap(&arr[i], &arr[r]);        // Standard partition algorithm      i = l;      for (int j = l; j <= r - 1; j++)      {          if (arr[j] <= x)          {              swap(&arr[i], &arr[j]);              i++;          }      }      swap(&arr[i], &arr[r]);      return i;  }    // Driver program to test above methods  int main()  {      int arr[] = {12, 3, 5, 7, 4, 19, 26};      int n = sizeof(arr)/sizeof(arr[0]), k = 3;      cout << "K'th smallest element is "           << kthSmallest(arr, 0, n-1, k);      return 0;  } |

Run on IDE

Output:

K'th smallest element is 5

**Time Complexity:**  
The worst case time complexity of the above algorithm is O(n). Let us analyze all steps.

The steps 1) and 2) take O(n) time as finding median of an array of size 5 takes O(1) time and there are n/5 arrays of size 5.  
The step 3) takes T(n/5) time. The step 4 is standard partition and takes O(n) time.  
The interesting steps are 6) and 7). At most, one of them is executed. These are recursive steps. What is the worst case size of these recursive calls. The answer is maximum number of elements greater than medOfMed (obtained in step 3) or maximum number of elements smaller than medOfMed.  
*How many elements are greater than medOfMed and how many are smaller?*  
At least half of the medians found in step 2 are greater than or equal to medOfMed. Thus, at least half of the n/5 groups contribute 3 elements that are greater than medOfMed, except for the one group that has fewer than 5 elements. Therefore, the number of elements greater than medOfMed is at least.  
[a](http://d1hyf4ir1gqw6c.cloudfront.net/wp-content/uploads/a.png)

Similarly, the number of elements that are less than medOfMed is at least 3n/10 – 6. In the worst case, the function recurs for at most n – (3n/10 – 6) which is 7n/10 + 6 elements.

Note that 7n/10 + 6 < n for n > 20 and that any input of 80 or fewer elements requires O(1) time. We can therefore obtain the recurrence  
[b](http://d1hyf4ir1gqw6c.cloudfront.net/wp-content/uploads/b.png)

We show that the running time is linear by substitution. Assume that T(n) cn for some constant c and all n > 80. Substituting this inductive hypothesis into the right-hand side of the recurrence yields

T(n) <= cn/5 + c(7n/10 + 6) + O(n)

<= cn/5 + c + 7cn/10 + 6c + O(n)

<= 9cn/10 + 7c + O(n)

<= cn,

since we can pick c large enough so that c(n/10 - 7) is larger than the function described by the O(n) term for all n > 80. The worst-case running time of is therefore linear (Source: <http://staff.ustc.edu.cn/~csli/graduate/algorithms/book6/chap10.htm> ).

Note that the above algorithm is linear in worst case, but the constants are very high for this algorithm. Therefore, this algorithm doesn't work well in practical situations, [randomized quickSelect](http://www.geeksforgeeks.org/kth-smallestlargest-element-unsorted-array-set-2-expected-linear-time/) works much better and preferred.

**Sources:**  
[MIT Video Lecture on Order Statistics, Median](http://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-046j-introduction-to-algorithms-sma-5503-fall-2005/video-lectures/lecture-6-order-statistics-median/)  
[Introduction to Algorithms by Clifford Stein, Thomas H. Cormen, Charles E. Leiserson, Ronald L.](http://www.flipkart.com/introduction-algorithms-8120340078/p/itmczynzhyhxv2gs?pid=9788120340077&affid=sandeepgfg)  
<http://staff.ustc.edu.cn/~csli/graduate/algorithms/book6/chap10.htm>

Find the closest pair from two sorted arrays

Given two sorted arrays and a number x, find the pair whose sum is closest to x and **the pair has an element from each array**.

We are given two arrays ar1[0…m-1] and ar2[0..n-1] and a number x, we need to find the pair ar1[i] + ar2[j] such that absolute value of (ar1[i] + ar2[j] – x) is minimum.

Example:

Input: ar1[] = {1, 4, 5, 7};

ar2[] = {10, 20, 30, 40};

x = 32

Output: 1 and 30

Input: ar1[] = {1, 4, 5, 7};

ar2[] = {10, 20, 30, 40};

x = 50

Output: 7 and 40

**We strongly recommend to minimize your browser and try this yourself first.**

A **Simple Solution** is to run two loops. The outer loop considers every element of first array and inner loop checks for the pair in second array. We keep track of minimum difference between ar1[i] + ar2[j] and x.

We can do it **in O(n) time**using following steps.  
1) Merge given two arrays into an auxiliary array of size m+n using [merge process of merge sort](http://geeksquiz.com/merge-sort/). While merging keep another boolean array of size m+n to indicate whether the current element in merged array is from ar1[] or ar2[].

2) Consider the merged array and use the [linear time algorithm to find the pair with sum closest to x](http://geeksquiz.com/given-sorted-array-number-x-find-pair-array-whose-sum-closest-x/). One extra thing we need to consider only those pairs which have one element from ar1[] and other from ar2[], we use the boolean array for this purpose.

**Can we do it in a single pass and O(1) extra space?**  
The idea is to start from left side of one array and right side of another array, and use the algorithm same as step 2 of above approach. Following is detailed algorithm.

1) Initialize a variable diff as infinite (Diff is used to store the

difference between pair and x). We need to find the minimum diff.

2) Initialize two index variables l and r in the given sorted array.

(a) Initialize first to the leftmost index in ar1: l = 0

(b) Initialize second the rightmost index in ar2: r = n-1

3) Loop while l < m and r >= 0

(a) If abs(ar1[l] + ar2[r] - sum) < diff then

update diff and result

(b) Else if(ar1[l] + ar2[r] < sum ) then l++

(c) Else r--

4) Print the result.

Following is C++ implementation of this approach.

* C++
* Java

|  |
| --- |
| // C++ program to find the pair from two sorted arays such  // that the sum of pair is closest to a given number x  #include <iostream>  #include <climits>  #include <cstdlib>  using namespace std;    // ar1[0..m-1] and ar2[0..n-1] are two given sorted arrays  // and x is given number. This function prints the pair  from  // both arrays such that the sum of the pair is closest to x.  void printClosest(int ar1[], int ar2[], int m, int n, int x)  {      // Initialize the diff between pair sum and x.      int diff = INT\_MAX;        // res\_l and res\_r are result indexes from ar1[] and ar2[]      // respectively      int res\_l, res\_r;        // Start from left side of ar1[] and right side of ar2[]      int l = 0, r = n-1;      while (l<m && r>=0)      {         // If this pair is closer to x than the previously         // found closest, then update res\_l, res\_r and diff         if (abs(ar1[l] + ar2[r] - x) < diff)         {             res\_l = l;             res\_r = r;             diff = abs(ar1[l] + ar2[r] - x);         }           // If sum of this pair is more than x, move to smaller         // side         if (ar1[l] + ar2[r] > x)             r--;         else  // move to the greater side             l++;      }        // Print the result      cout << "The closest pair is [" << ar1[res\_l] << ", "           << ar2[res\_r] << "] \n";  }    // Driver program to test above functions  int main()  {      int ar1[] = {1, 4, 5, 7};      int ar2[] = {10, 20, 30, 40};      int m = sizeof(ar1)/sizeof(ar1[0]);      int n = sizeof(ar2)/sizeof(ar2[0]);      int x = 38;      printClosest(ar1, ar2, m, n, x);      return 0;  } |

Run on IDE

Output:

The closest pair is [7, 30]

Find common elements in three sorted arrays

Given three arrays sorted in non-decreasing order, print all common elements in these arrays.

Examples:

ar1[] = {1, 5, 10, 20, 40, 80}

ar2[] = {6, 7, 20, 80, 100}

ar3[] = {3, 4, 15, 20, 30, 70, 80, 120}

Output: 20, 80

ar1[] = {1, 5, 5}

ar2[] = {3, 4, 5, 5, 10}

ar3[] = {5, 5, 10, 20}

Output: 5, 5

A simple solution is to first find [intersection of two arrays](http://www.geeksforgeeks.org/union-and-intersection-of-two-sorted-arrays-2/)and store the intersection in a temporary array, then find the intersection of third array and temporary array. Time complexity of this solution is O(n1 + n2 + n3) where n1, n2 and n3 are sizes of ar1[], ar2[] and ar3[] respectively.

The above solution requires extra space and two loops, we can find the common elements using a single loop and without extra space. The idea is similar to [intersection of two arrays](http://www.geeksforgeeks.org/union-and-intersection-of-two-sorted-arrays-2/). Like two arrays loop, we run a loop and traverse three arrays.  
Let the current element traversed in ar1[] be x, in ar2[] be y and in ar3[] be z. We can have following cases inside the loop.  
1) If x, y and z are same, we can simply print any of them as common element and move ahead in all three arrays.  
2) Else If x < y, we can move ahead in ar1[] as x cannot be a common element 3) Else If y < z, we can move ahead in ar2[] as y cannot be a common element 4) Else (We reach here when x > y and y > z), we can simply move ahead in ar3[] as z cannot be a common element.

Following are implementations of the above idea.

* C++
* Python
* Java

|  |
| --- |
| // C++ program to print common elements in three arrays  #include <iostream>  using namespace std;    // This function prints common elements in ar1  void findCommon(int ar1[], int ar2[], int ar3[], int n1, int n2, int n3)  {      // Initialize starting indexes for ar1[], ar2[] and ar3[]      int i = 0, j = 0, k = 0;        // Iterate through three arrays while all arrays have elements      while (i < n1 && j < n2 && k < n3)      {           // If x = y and y = z, print any of them and move ahead           // in all arrays           if (ar1[i] == ar2[j] && ar2[j] == ar3[k])           {   cout << ar1[i] << " ";   i++; j++; k++; }             // x < y           else if (ar1[i] < ar2[j])               i++;             // y < z           else if (ar2[j] < ar3[k])               j++;             // We reach here when x > y and z < y, i.e., z is smallest           else               k++;      }  }    // Driver program to test above function  int main()  {      int ar1[] = {1, 5, 10, 20, 40, 80};      int ar2[] = {6, 7, 20, 80, 100};      int ar3[] = {3, 4, 15, 20, 30, 70, 80, 120};      int n1 = sizeof(ar1)/sizeof(ar1[0]);      int n2 = sizeof(ar2)/sizeof(ar2[0]);      int n3 = sizeof(ar3)/sizeof(ar3[0]);        cout << "Common Elements are ";      findCommon(ar1, ar2, ar3, n1, n2, n3);      return 0;  } |

Run on IDE

Output:

Common Elements are 20 80

Time complexity of the above solution is O(n1 + n2 + n3). In worst case, the largest sized array may have all small elements and middle sized array has all middle elements.

Given a sorted array and a number x, find the pair in array whose sum is closest to x

Given a sorted array and a number x, find a pair in array whose sum is closest to x.

Examples:

Input: arr[] = {10, 22, 28, 29, 30, 40}, x = 54

Output: 22 and 30

Input: arr[] = {1, 3, 4, 7, 10}, x = 15

Output: 4 and 10

A simple solution is to consider every pair and keep track of closest pair (absolute difference between pair sum and x is minimum). Finally print the closest pair. Time complexity of this solution is O(n2)

An efficient solution can find the pair in O(n) time. The idea is similar to method 2 of [this](http://www.geeksforgeeks.org/write-a-c-program-that-given-a-set-a-of-n-numbers-and-another-number-x-determines-whether-or-not-there-exist-two-elements-in-s-whose-sum-is-exactly-x/)post. Following is detailed algorithm.

1) Initialize a variable diff as infinite (Diff is used to store the

difference between pair and x). We need to find the minimum diff.

2) Initialize two index variables l and r in the given sorted array.

(a) Initialize first to the leftmost index: l = 0

(b) Initialize second the rightmost index: r = n-1

3) Loop while l < r.

(a) If abs(arr[l] + arr[r] - sum) < diff then

update diff and result

(b) Else if(arr[l] + arr[r] < sum ) then l++

(c) Else r--

Following is C++ implementation of above algorithm.

* C++
* Java

|  |
| --- |
| // Simple C++ program to find the pair with sum closest to a given no.  #include <iostream>  #include <climits>  #include <cstdlib>  using namespace std;    // Prints the pair with sum closest to x  void printClosest(int arr[], int n, int x)  {      int res\_l, res\_r;  // To store indexes of result pair        // Initialize left and right indexes and difference between      // pair sum and x      int l = 0, r = n-1, diff = INT\_MAX;        // While there are elements between l and r      while (r > l)      {         // Check if this pair is closer than the closest pair so far         if (abs(arr[l] + arr[r] - x) < diff)         {             res\_l = l;             res\_r = r;             diff = abs(arr[l] + arr[r] - x);         }           // If this pair has more sum, move to smaller values.         if (arr[l] + arr[r] > x)             r--;         else // Move to larger values             l++;      }        cout <<" The closest pair is " << arr[res\_l] << " and " << arr[res\_r];  }    // Driver program to test above functions  int main()  {      int arr[] =  {10, 22, 28, 29, 30, 40}, x = 54;      int n = sizeof(arr)/sizeof(arr[0]);      printClosest(arr, n, x);      return 0;  } |

Run on IDE

Output:

The closest pair is 22 and 30

This article is contributed by **Harsh**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

Count 1’s in a sorted binary array

Given a binary array sorted in non-increasing order, count the number of 1’s in it.

Examples:

Input: arr[] = {1, 1, 0, 0, 0, 0, 0}

Output: 2

Input: arr[] = {1, 1, 1, 1, 1, 1, 1}

Output: 7

Input: arr[] = {0, 0, 0, 0, 0, 0, 0}

Output: 0

A simple solution is to linearly traverse the array. The time complexity of the simple solution is O(n). We can use [Binary Search](http://quiz.geeksforgeeks.org/binary-search/)to find count in O(Logn) time. The idea is to look for last occurrence of 1 using Binary Search. Once we find the index last occurrence, we return index + 1 as count.

The following is C++ implementation of above idea.

* C++
* Python
* Java

|  |
| --- |
| // C++ program to count one's in a boolean array  #include <iostream>  using namespace std;    /\* Returns counts of 1's in arr[low..high].  The array is     assumed to be sorted in non-increasing order \*/  int countOnes(bool arr[], int low, int high)  {    if (high >= low)    {      // get the middle index      int mid = low + (high - low)/2;        // check if the element at middle index is last 1      if ( (mid == high || arr[mid+1] == 0) && (arr[mid] == 1))        return mid+1;        // If element is not last 1, recur for right side      if (arr[mid] == 1)        return countOnes(arr, (mid + 1), high);        // else recur for left side      return countOnes(arr, low, (mid -1));    }    return 0;  }    /\* Driver program to test above functions \*/  int main()  {     bool arr[] = {1, 1, 1, 1, 0, 0, 0};     int n = sizeof(arr)/sizeof(arr[0]);     cout << "Count of 1's in given array is " << countOnes(arr, 0, n-1);     return 0;  } |

Run on IDE

Output:

Count of 1's in given array is 4

Time complexity of the above solution is O(Logn)

Binary Insertion Sort

We can use binary search to reduce the number of comparisons in [normal insertion sort](http://quiz.geeksforgeeks.org/insertion-sort/). Binary Insertion Sort find use binary search to find the proper location to insert the selected item at each iteration.   
In normal insertion, sort it takes O(i) (at ith iteration) in worst case. we can reduce it to O(logi) by using [binary search](http://quiz.geeksforgeeks.org/binary-search/).

|  |
| --- |
| // C program for implementation of binary insertion sort  #include <stdio.h>    // A binary search based function to find the position  // where item should be inserted in a[low..high]  int binarySearch(int a[], int item, int low, int high)  {      if (high <= low)          return (item > a[low])?  (low + 1): low;        int mid = (low + high)/2;        if(item == a[mid])          return mid+1;        if(item > a[mid])          return binarySearch(a, item, mid+1, high);      return binarySearch(a, item, low, mid-1);  }    // Function to sort an array a[] of size 'n'  void insertionSort(int a[], int n)  {      int i, loc, j, k, selected;        for (i = 1; i < n; ++i)      {          j = i - 1;          selected = a[i];            // find location where selected sould be inseretd          loc = binarySearch(a, selected, 0, j);            // Move all elements after location to create space          while (j >= loc)          {              a[j+1] = a[j];              j--;          }          a[j+1] = selected;      }  }    // Driver program to test above function  int main()  {      int a[] = {37, 23, 0, 17, 12, 72, 31,                46, 100, 88, 54};      int n = sizeof(a)/sizeof(a[0]), i;        insertionSort(a, n);        printf("Sorted array: \n");      for (i = 0; i < n; i++)          printf("%d ",a[i]);        return 0;  } |

Run on IDE

Output:

Sorted array:

0 12 17 23 31 37 46 54 72 88 100

**Time Complexity:** The algorithm as a whole still has a running worst case running time of O(n2) because of the series of swaps required for each insertion.

Insertion Sort for Singly Linked List

We have discussed [Insertion Sort for arrays](http://quiz.geeksforgeeks.org/insertion-sort/). In this article same for linked list is discussed.

Below is simple insertion sort algorithm for linked list.

1) Create an empty sorted (or result) list

2) Traverse the given list, do following for every node.

......a) Insert current node in sorted way in sorted or result list.

3) Change head of given linked list to head of sorted (or result) list.

The main step is (2.a) which has been covered in below post.  
[Sorted Insert for Singly Linked List](http://www.geeksforgeeks.org/given-a-linked-list-which-is-sorted-how-will-you-insert-in-sorted-way/)

Below is C implementation of above algorithm

|  |
| --- |
| /\* C program for insertion sort on a linked list \*/  #include<stdio.h>  #include<stdlib.h>    /\* Link list node \*/  struct node  {      int data;      struct node\* next;  };    // Function to insert a given node in a sorted linked list  void sortedInsert(struct node\*\*, struct node\*);    // function to sort a singly linked list using insertion sort  void insertionSort(struct node \*\*head\_ref)  {      // Initialize sorted linked list      struct node \*sorted = NULL;        // Traverse the given linked list and insert every      // node to sorted      struct node \*current = \*head\_ref;      while (current != NULL)      {          // Store next for next iteration          struct node \*next = current->next;            // insert current in sorted linked list          sortedInsert(&sorted, current);            // Update current          current = next;      }        // Update head\_ref to point to sorted linked list      \*head\_ref = sorted;  }      /\* function to insert a new\_node in a list. Note that this    function expects a pointer to head\_ref as this can modify the    head of the input linked list (similar to push())\*/  void sortedInsert(struct node\*\* head\_ref, struct node\* new\_node)  {      struct node\* current;      /\* Special case for the head end \*/      if (\*head\_ref == NULL || (\*head\_ref)->data >= new\_node->data)      {          new\_node->next = \*head\_ref;          \*head\_ref = new\_node;      }      else      {          /\* Locate the node before the point of insertion \*/          current = \*head\_ref;          while (current->next!=NULL &&                 current->next->data < new\_node->data)          {              current = current->next;          }          new\_node->next = current->next;          current->next = new\_node;      }  }    /\* BELOW FUNCTIONS ARE JUST UTILITY TO TEST sortedInsert \*/    /\* Function to print linked list \*/  void printList(struct node \*head)  {      struct node \*temp = head;      while(temp != NULL)      {          printf("%d  ", temp->data);          temp = temp->next;      }  }    /\* A utility function to insert a node at the beginning of linked list \*/  void push(struct node\*\* head\_ref, int new\_data)  {      /\* allocate node \*/      struct node\* new\_node = new node;        /\* put in the data  \*/      new\_node->data  = new\_data;        /\* link the old list off the new node \*/      new\_node->next = (\*head\_ref);        /\* move the head to point to the new node \*/      (\*head\_ref)    = new\_node;  }      // Driver program to test above functions  int main()  {      struct node \*a = NULL;      push(&a, 5);      push(&a, 20);      push(&a, 4);      push(&a, 3);      push(&a, 30);        printf("Linked List before sorting \n");      printList(a);        insertionSort(&a);        printf("\nLinked List after sorting \n");      printList(a);        return 0;  } |

Run on IDE

Linked List before sorting

30 3 4 20 5

Linked List after sorting

3 4 5 20 30

# Why Quick Sort preferred for Arrays and Merge Sort for Linked Lists?

**Why is**[**Quick Sort**](http://geeksquiz.com/quick-sort/)**preferred for arrays?**  
Below are recursive and iterative implementations of Quick Sort and Merge Sort for arrays.

[Recursive Quick Sort for array.](http://geeksquiz.com/quick-sort/)  
[Iterative Quick Sort for arrays.](http://www.geeksforgeeks.org/iterative-quick-sort/)  
[Recursive Merge Sort for arrays](http://geeksquiz.com/merge-sort/)  
[Iterative Merge Sort for arrays](http://www.geeksforgeeks.org/iterative-merge-sort/)

Quick Sort in its general form is an in-place sort (i.e. it doesn’t require any extra storage) whereas merge sort requires O(N) extra storage, N denoting the array size which may be quite expensive. Allocating and de-allocating the extra space used for merge sort increases the running time of the algorithm. Comparing average complexity we find that both type of sorts have O(NlogN) average complexity but the constants differ. For arrays, merge sort loses due to the use of extra O(N) storage space.

Most practical implementations of Quick Sort use randomized version. The randomized version has expected time complexity of O(nLogn). The worst case is possible in randomized version also, but worst case doesn’t occur for a particular pattern (like sorted array) and randomized Quick Sort works well in practice.

Quick Sort is also a cache friendly sorting algorithm as it has good [locality of reference](http://en.wikipedia.org/wiki/Locality_of_reference) when used for arrays.

Quick Sort is also [tail recursive](http://www.geeksforgeeks.org/tail-recursion/), therefore tail call optimizations is done.

**Why is**[**Merge Sort**](http://geeksquiz.com/merge-sort/)**preferred for Linked Lists?**  
Below are implementations of Quicksort and Mergesort for singly and doubly linked lists.

[Quick Sort for Doubly Linked List](http://www.geeksforgeeks.org/quicksort-for-linked-list)  
[Quick Sort for Singly Linked List](http://www.geeksforgeeks.org/quicksort-on-singly-linked-list/)  
[Merge Sort for Singly Linked List](http://www.geeksforgeeks.org/merge-sort-for-linked-list/)  
[Merge Sort for Doubly Linked List](http://www.geeksforgeeks.org/merge-sort-for-doubly-linked-list/)

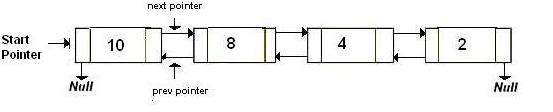
In case of [linked lists](http://geeksquiz.com/linked-list-set-1-introduction/) the case is different mainly due to difference in memory allocation of arrays and linked lists. Unlike arrays, linked list nodes may not be adjacent in memory. Unlike array, in [linked list](http://geeksquiz.com/linked-list-set-1-introduction/), we can insert items in the middle in O(1) extra space and O(1) time. Therefore merge operation of merge sort can be implemented without extra space for linked lists.

In arrays, we can do random access as elements are continuous in memory. Let us say we have an integer (4-byte) array A and let the address of A[0] be x then to access A[i], we can directly access the memory at (x + i\*4). Unlike arrays, we can not do random access in linked list. Quick Sort requires a lot of this kind of access. In linked list to access i’th index, we have to travel each and every node from the head to i’th node as we don’t have continuous block of memory. Therefore, the overhead increases for quick sort. Merge sort accesses data sequentially and the need of random access is low.

Merge Sort for Doubly Linked List

Given a doubly linked list, write a function to sort the doubly linked list in increasing order using merge sort.

For example, the following doubly linked list should be changed to 2<->4<->8<->10



**We strongly recommend to minimize your browser and try this yourself first.**  
[Merge sort for singly linked list](http://www.geeksforgeeks.org/merge-sort-for-linked-list/) is already discussed. The important change here is to modify the previous pointers also when merging two lists.

Below is the implementation of merge sort for doubly linked list.

* C
* Java
* Python

|  |
| --- |
| // C program for merge sort on doubly linked list  #include<stdio.h>  #include<stdlib.h>  struct node  {      int data;      struct node \*next, \*prev;  };    struct node \*split(struct node \*head);    // Function to merge two linked lists  struct node \*merge(struct node \*first, struct node \*second)  {      // If first linked list is empty      if (!first)          return second;        // If second linked list is empty      if (!second)          return first;        // Pick the smaller value      if (first->data < second->data)      {          first->next = merge(first->next,second);          first->next->prev = first;          first->prev = NULL;          return first;      }      else      {          second->next = merge(first,second->next);          second->next->prev = second;          second->prev = NULL;          return second;      }  }    // Function to do merge sort  struct node \*mergeSort(struct node \*head)  {      if (!head || !head->next)          return head;      struct node \*second = split(head);        // Recur for left and right halves      head = mergeSort(head);      second = mergeSort(second);        // Merge the two sorted halves      return merge(head,second);  }    // A utility function to insert a new node at the  // beginning of doubly linked list  void insert(struct node \*\*head, int data)  {      struct node \*temp =          (struct node \*)malloc(sizeof(struct node));      temp->data = data;      temp->next = temp->prev = NULL;      if (!(\*head))          (\*head) = temp;      else      {          temp->next = \*head;          (\*head)->prev = temp;          (\*head) = temp;      }  }    // A utility function to print a doubly linked list in  // both forward and backward directions  void print(struct node \*head)  {      struct node \*temp = head;      printf("Forward Traversal using next poitner\n");      while (head)      {          printf("%d ",head->data);          temp = head;          head = head->next;      }      printf("\nBackward Traversal using prev pointer\n");      while (temp)      {          printf("%d ", temp->data);          temp = temp->prev;      }  }    // Utility function to swap two integers  void swap(int \*A, int \*B)  {      int temp = \*A;      \*A = \*B;      \*B = temp;  }    // Split a doubly linked list (DLL) into 2 DLLs of  // half sizes  struct node \*split(struct node \*head)  {      struct node \*fast = head,\*slow = head;      while (fast->next && fast->next->next)      {          fast = fast->next->next;          slow = slow->next;      }      struct node \*temp = slow->next;      slow->next = NULL;      return temp;  }    // Driver program  int main(void)  {      struct node \*head = NULL;      insert(&head,5);      insert(&head,20);      insert(&head,4);      insert(&head,3);      insert(&head,30);      insert(&head,10);      head = mergeSort(head);      printf("\n\nLinked List after sorting\n");      print(head);      return 0;  } |

Run on IDE

Output:

Linked List after sorting

Forward Traversal using next pointer

3 4 5 10 20 30

Backward Traversal using prev pointer

30 20 10 5 4 3

Thanks to Goku for providing above implementation in a comment [here](http://www.geeksforgeeks.org/quicksort-for-linked-list/).

**Time Complexity:**Time complexity of the above implementation is same as time complexity of [MergeSort for arrays](http://geeksquiz.com/merge-sort/). It takes Θ(nLogn) time.

**Greedy Algorithms**:

1. [Activity Selection Problem](http://www.geeksforgeeks.org/greedy-algorithms-set-1-activity-selection-problem/)
2. [Kruskal’s Minimum Spanning Tree Algorithm](http://www.geeksforgeeks.org/greedy-algorithms-set-2-kruskals-minimum-spanning-tree-mst/)
3. [Huffman Coding](http://www.geeksforgeeks.org/greedy-algorithms-set-3-huffman-coding/)
4. [Efficient Huffman Coding for Sorted Input](http://www.geeksforgeeks.org/greedy-algorithms-set-3-huffman-coding-set-2/)
5. [Prim’s Minimum Spanning Tree Algorithm](http://www.geeksforgeeks.org/greedy-algorithms-set-5-prims-minimum-spanning-tree-mst-2/)
6. [Prim’s MST for Adjacency List Representation](http://www.geeksforgeeks.org/greedy-algorithms-set-5-prims-mst-for-adjacency-list-representation/)
7. [Dijkstra’s Shortest Path Algorithm](http://www.geeksforgeeks.org/greedy-algorithms-set-6-dijkstras-shortest-path-algorithm/)
8. [Dijkstra’s Algorithm for Adjacency List Representation](http://www.geeksforgeeks.org/greedy-algorithms-set-7-dijkstras-algorithm-for-adjacency-list-representation/)
9. [Job Sequencing Problem](http://www.geeksforgeeks.org/job-sequencing-problem-set-1-greedy-algorithm/)
10. [Quiz on Greedy Algorithms](http://geeksquiz.com/algorithms/greedy-algorithms/)
11. [Greedy Algorithm to find Minimum number of Coins](http://geeksquiz.com/greedy-algorithm-to-find-minimum-number-of-coins/)
12. [K Centers Problem](http://www.geeksforgeeks.org/k-centers-problem-set-1-greedy-approximate-algorithm/)

# Greedy Algorithms | Set 1 (Activity Selection Problem)

Greedy is an algorithmic paradigm that builds up a solution piece by piece, always choosing the next piece that offers the most obvious and immediate benefit. Greedy algorithms are used for optimization problems. An optimization problem can be solved using Greedy if the problem has the following property:At every step, we can make a choice that looks best at the moment, and we get the optimal solution of the complete problem.  
If a Greedy Algorithm can solve a problem, then it generally becomes the best method to solve that problem as the Greedy algorithms are in general more efficient than other techniques like Dynamic Programming. But Greedy algorithms cannot always be applied. For example, Fractional Knapsack problem (See [this](http://www.cs.binghamton.edu/~dima/cs333/knapsack.ppt)) can be solved using Greedy, but [0-1 Knapsack](http://www.geeksforgeeks.org/archives/18430) cannot be solved using Greedy.

Following are some standard algorithms that are Greedy algorithms.  
**1)**[**Kruskal’s Minimum Spanning Tree (MST)**](http://en.wikipedia.org/wiki/Kruskal%27s_algorithm)**:** In Kruskal’s algorithm, we create a MST by picking edges one by one. The Greedy Choice is to pick the smallest weight edge that doesn’t cause a cycle in the MST constructed so far.  
**2)**[**Prim’s Minimum Spanning Tree**](http://en.wikipedia.org/wiki/Prim%27s_algorithm)**:** In Prim’s algorithm also, we create a MST by picking edges one by one. We maintain two sets: set of the vertices already included in MST and the set of the vertices not yet included. The Greedy Choice is to pick the smallest weight edge that connects the two sets.  
**3)**[**Dijkstra’s Shortest Path**](http://en.wikipedia.org/wiki/Dijkstra%27s_algorithm)**:**The Dijkstra’s algorithm is very similar to Prim’s algorithm. The shortest path tree is built up, edge by edge. We maintain two sets: set of the vertices already included in the tree and the set of the vertices not yet included. The Greedy Choice is to pick the edge that connects the two sets and is on the smallest weight path from source to the set that contains not yet included vertices.  
**4)**[**Huffman Coding**](http://en.wikipedia.org/wiki/Huffman_coding)**:** Huffman Coding is a loss-less compression technique. It assigns variable length bit codes to different characters. The Greedy Choice is to assign least bit length code to the most frequent character.

The greedy algorithms are sometimes also used to get an approximation for Hard optimization problems. For example, [Traveling Salesman Problem](http://en.wikipedia.org/wiki/Travelling_salesman_problem) is a NP Hard problem. A Greedy choice for this problem is to pick the nearest unvisited city from the current city at every step. This solutions doesn’t always produce the best optimal solution, but can be used to get an approximate optimal solution.

Let us consider the [Activity Selection problem](http://en.wikipedia.org/wiki/Activity_selection_problem) as our first example of Greedy algorithms. Following is the problem statement.  
You are given n activities with their start and finish times. Select the maximum number of activities that can be performed by a single person, assuming that a person can only work on a single activity at a time.  
Example:

Consider the following 6 activities.

start[] = {1, 3, 0, 5, 8, 5};

finish[] = {2, 4, 6, 7, 9, 9};

The maximum set of activities that can be executed

by a single person is {0, 1, 3, 4}

## [We strongly recommend that you click here and practice it, before moving on to the solution.](http://www.practice.geeksforgeeks.org/problem-page.php?pid=296)

The greedy choice is to always pick the next activity whose finish time is least among the remaining activities and the start time is more than or equal to the finish time of previously selected activity. We can sort the activities according to their finishing time so that we always consider the next activity as minimum finishing time activity.

1) Sort the activities according to their finishing time  
2) Select the first activity from the sorted array and print it.  
3) Do following for remaining activities in the sorted array.  
…….a) If the start time of this activity is greater than the finish time of previously selected activity then select this activity and print it.

In the following C implementation, it is assumed that the activities are already sorted according to their finish time.

* C++
* Java
* Python

|  |
| --- |
| #include<stdio.h>    // Prints a maximum set of activities that can be done by a single  // person, one at a time.  //  n   -->  Total number of activities  //  s[] -->  An array that contains start time of all activities  //  f[] -->  An array that contains finish time of all activities  void printMaxActivities(int s[], int f[], int n)  {      int i, j;        printf ("Following activities are selected \n");        // The first activity always gets selected      i = 0;      printf("%d ", i);        // Consider rest of the activities      for (j = 1; j < n; j++)      {        // If this activity has start time greater than or        // equal to the finish time of previously selected        // activity, then select it        if (s[j] >= f[i])        {            printf ("%d ", j);            i = j;        }      }  }    // driver program to test above function  int main()  {      int s[] =  {1, 3, 0, 5, 8, 5};      int f[] =  {2, 4, 6, 7, 9, 9};      int n = sizeof(s)/sizeof(s[0]);      printMaxActivities(s, f, n);      getchar();      return 0;  } |

Run on IDE

Output:

Following activities are selected

0 1 3 4

**How does Greedy Choice work for Activities sorted according to finish time?**  
Let the give set of activities be S = {1, 2, 3, ..n} and activities be sorted by finish time. The greedy choice is to always pick activity 1. How come the activity 1 always provides one of the optimal solutions. We can prove it by showing that if there is another solution B with first activity other than 1, then there is also a solution A of same size with activity 1 as first activity. Let the first activity selected by B be k, then there always exist A = {B – {k}} U {1}.(Note that the activities in B are independent and k has smallest finishing time among all. Since k is not 1, finish(k) >= finish(1)).

**References:**  
[Introduction to Algorithms by Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein](http://mitpress.mit.edu/algorithms/)  
<http://en.wikipedia.org/wiki/Greedy_algorithm>

# Greedy Algorithms | Set 2 (Kruskal’s Minimum Spanning Tree Algorithm)

What is Minimum Spanning Tree?  
Given a connected and undirected graph, a spanning tree of that graph is a subgraph that is a tree and connects all the vertices together. A single graph can have many different spanning trees. A minimum spanning tree (MST) or minimum weight spanning tree for a weighted, connected and undirected graph is a spanning tree with weight less than or equal to the weight of every other spanning tree. The weight of a spanning tree is the sum of weights given to each edge of the spanning tree.

How many edges does a minimum spanning tree has?  
A minimum spanning tree has (V – 1) edges where V is the number of vertices in the given graph.

What are the applications of Minimum Spanning Tree?  
See [this](http://www.geeksforgeeks.org/archives/11110)for applications of MST.

Below are the steps for finding MST using Kruskal’s algorithm

**1.** Sort all the edges in non-decreasing order of their weight.

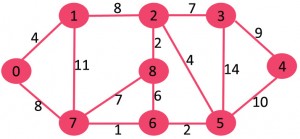
**2.** Pick the smallest edge. Check if it forms a cycle with the spanning tree

formed so far. If cycle is not formed, include this edge. Else, discard it.

**3.** Repeat step#2 until there are (V-1) edges in the spanning tree.

The step#2 uses [Union-Find algorithm](http://www.geeksforgeeks.org/archives/26350) to detect cycle. So we recommend to read following post as a prerequisite.  
[Union-Find Algorithm | Set 1 (Detect Cycle in a Graph)](http://www.geeksforgeeks.org/union-find/)  
[Union-Find Algorithm | Set 2 (Union By Rank and Path Compression)](http://www.geeksforgeeks.org/union-find-algorithm-set-2-union-by-rank/)

The algorithm is a Greedy Algorithm. The Greedy Choice is to pick the smallest weight edge that does not cause a cycle in the MST constructed so far. Let us understand it with an example: Consider the below input graph.

[](http://d1hyf4ir1gqw6c.cloudfront.net/wp-content/uploads/Fig-0.jpg)

The graph contains 9 vertices and 14 edges. So, the minimum spanning tree formed will be having (9 – 1) = 8 edges.

After sorting:

Weight Src Dest

1 7 6

2 8 2

2 6 5

4 0 1

4 2 5

6 8 6

7 2 3

7 7 8

8 0 7

8 1 2

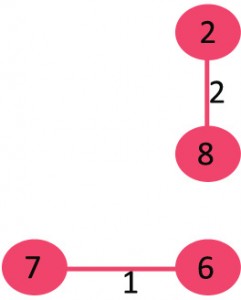
9 3 4

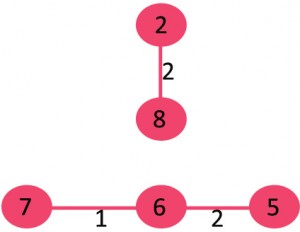
10 5 4

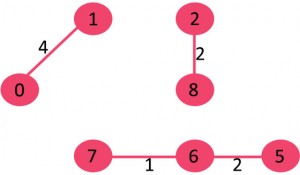
11 1 7

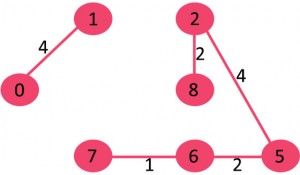
14 3 5

Now pick all edges one by one from sorted list of edges  
**1.** Pick edge 7-6: No cycle is formed, include it.  
[](http://d1hyf4ir1gqw6c.cloudfront.net/wp-content/uploads/Fig-1.jpg)

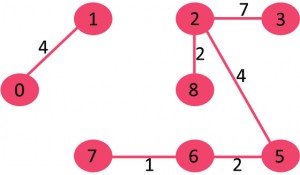
**2.** Pick edge 8-2: No cycle is formed, include it.  
[](http://d1hyf4ir1gqw6c.cloudfront.net/wp-content/uploads/Fig-2.jpg)

**3.** Pick edge 6-5: No cycle is formed, include it.  
[](http://d1hyf4ir1gqw6c.cloudfront.net/wp-content/uploads/Fig-3.jpg)

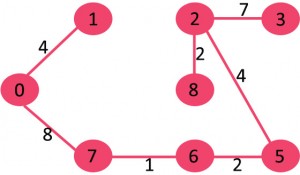
**4.** Pick edge 0-1: No cycle is formed, include it.  
[](http://d1hyf4ir1gqw6c.cloudfront.net/wp-content/uploads/Fig-4.jpg)

**5.** Pick edge 2-5: No cycle is formed, include it.  
[](http://d1hyf4ir1gqw6c.cloudfront.net/wp-content/uploads/Fig-5.jpg)

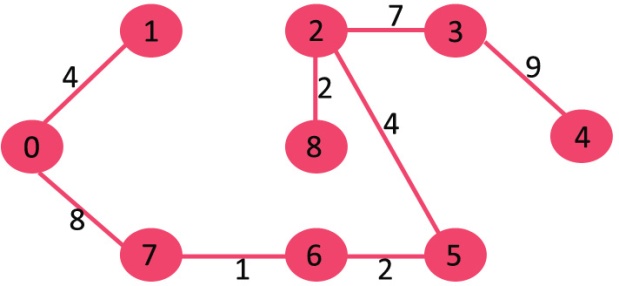
**6.**Pick edge 8-6:Since including this edge results in cycle, discard it.

**7.** Pick edge 2-3: No cycle is formed, include it.  
[](http://d1hyf4ir1gqw6c.cloudfront.net/wp-content/uploads/Fig-6.jpg)

**8.** Pick edge 7-8: Since including this edge results in cycle, discard it.

**9.** Pick edge 0-7: No cycle is formed, include it.  
[](http://d1hyf4ir1gqw6c.cloudfront.net/wp-content/uploads/Fig-7.jpg)

**10.** Pick edge 1-2:Since including this edge results in cycle, discard it.

**11.** Pick edge 3-4: No cycle is formed, include it.  
[](http://d1hyf4ir1gqw6c.cloudfront.net/wp-content/uploads/fig8new.jpeg)

Since the number of edges included equals (V – 1), the algorithm stops here.

**We strongly recommend you to minimize your browser and try this yourself first.**

* C/C++
* Java
* Python

|  |
| --- |
| // C++ program for Kruskal's algorithm to find Minimum Spanning Tree  // of a given connected, undirected and weighted graph  #include <stdio.h>  #include <stdlib.h>  #include <string.h>    // a structure to represent a weighted edge in graph  struct Edge  {      int src, dest, weight;  };    // a structure to represent a connected, undirected and weighted graph  struct Graph  {      // V-> Number of vertices, E-> Number of edges      int V, E;        // graph is represented as an array of edges. Since the graph is      // undirected, the edge from src to dest is also edge from dest      // to src. Both are counted as 1 edge here.      struct Edge\* edge;  };    // Creates a graph with V vertices and E edges  struct Graph\* createGraph(int V, int E)  {      struct Graph\* graph = (struct Graph\*) malloc( sizeof(struct Graph) );      graph->V = V;      graph->E = E;        graph->edge = (struct Edge\*) malloc( graph->E \* sizeof( struct Edge ) );        return graph;  }    // A structure to represent a subset for union-find  struct subset  {      int parent;      int rank;  };    // A utility function to find set of an element i  // (uses path compression technique)  int find(struct subset subsets[], int i)  {      // find root and make root as parent of i (path compression)      if (subsets[i].parent != i)          subsets[i].parent = find(subsets, subsets[i].parent);        return subsets[i].parent;  }    // A function that does union of two sets of x and y  // (uses union by rank)  void Union(struct subset subsets[], int x, int y)  {      int xroot = find(subsets, x);      int yroot = find(subsets, y);        // Attach smaller rank tree under root of high rank tree      // (Union by Rank)      if (subsets[xroot].rank < subsets[yroot].rank)          subsets[xroot].parent = yroot;      else if (subsets[xroot].rank > subsets[yroot].rank)          subsets[yroot].parent = xroot;        // If ranks are same, then make one as root and increment      // its rank by one      else      {          subsets[yroot].parent = xroot;          subsets[xroot].rank++;      }  }    // Compare two edges according to their weights.  // Used in qsort() for sorting an array of edges  int myComp(const void\* a, const void\* b)  {      struct Edge\* a1 = (struct Edge\*)a;      struct Edge\* b1 = (struct Edge\*)b;      return a1->weight > b1->weight;  }    // The main function to construct MST using Kruskal's algorithm  void KruskalMST(struct Graph\* graph)  {      int V = graph->V;      struct Edge result[V];  // Tnis will store the resultant MST      int e = 0;  // An index variable, used for result[]      int i = 0;  // An index variable, used for sorted edges        // Step 1:  Sort all the edges in non-decreasing order of their weight      // If we are not allowed to change the given graph, we can create a copy of      // array of edges      qsort(graph->edge, graph->E, sizeof(graph->edge[0]), myComp);        // Allocate memory for creating V ssubsets      struct subset \*subsets =          (struct subset\*) malloc( V \* sizeof(struct subset) );        // Create V subsets with single elements      for (int v = 0; v < V; ++v)      {          subsets[v].parent = v;          subsets[v].rank = 0;      }        // Number of edges to be taken is equal to V-1      while (e < V - 1)      {          // Step 2: Pick the smallest edge. And increment the index          // for next iteration          struct Edge next\_edge = graph->edge[i++];            int x = find(subsets, next\_edge.src);          int y = find(subsets, next\_edge.dest);            // If including this edge does't cause cycle, include it          // in result and increment the index of result for next edge          if (x != y)          {              result[e++] = next\_edge;              Union(subsets, x, y);          }          // Else discard the next\_edge      }        // print the contents of result[] to display the built MST      printf("Following are the edges in the constructed MST\n");      for (i = 0; i < e; ++i)          printf("%d -- %d == %d\n", result[i].src, result[i].dest,                                                     result[i].weight);      return;  }    // Driver program to test above functions  int main()  {      /\* Let us create following weighted graph               10          0--------1          |  \     |         6|   5\   |15          |      \ |          2--------3              4       \*/      int V = 4;  // Number of vertices in graph      int E = 5;  // Number of edges in graph      struct Graph\* graph = createGraph(V, E);          // add edge 0-1      graph->edge[0].src = 0;      graph->edge[0].dest = 1;      graph->edge[0].weight = 10;        // add edge 0-2      graph->edge[1].src = 0;      graph->edge[1].dest = 2;      graph->edge[1].weight = 6;        // add edge 0-3      graph->edge[2].src = 0;      graph->edge[2].dest = 3;      graph->edge[2].weight = 5;        // add edge 1-3      graph->edge[3].src = 1;      graph->edge[3].dest = 3;      graph->edge[3].weight = 15;        // add edge 2-3      graph->edge[4].src = 2;      graph->edge[4].dest = 3;      graph->edge[4].weight = 4;        KruskalMST(graph);        return 0;  } |

Run on IDE

Following are the edges in the constructed MST

2 -- 3 == 4

0 -- 3 == 5

0 -- 1 == 10

**Time Complexity:** O(ElogE) or O(ElogV). Sorting of edges takes O(ELogE) time. After sorting, we iterate through all edges and apply find-union algorithm. The find and union operations can take atmost O(LogV) time. So overall complexity is O(ELogE + ELogV) time. The value of E can be atmost O(V2), so O(LogV) are O(LogE) same. Therefore, overall time complexity is O(ElogE) or O(ElogV)

References:  
<http://www.ics.uci.edu/~eppstein/161/960206.html>  
<http://en.wikipedia.org/wiki/Minimum_spanning_tree>

# Greedy Algorithms | Set 3 (Huffman Coding)

Huffman coding is a lossless data compression algorithm. The idea is to assign variable-legth codes to input characters, lengths of the assigned codes are based on the frequencies of corresponding characters. The most frequent character gets the smallest code and the least frequent character gets the largest code.  
The variable-length codes assigned to input characters are [Prefix Codes](http://en.wikipedia.org/wiki/Prefix_code), means the codes (bit sequences) are assigned in such a way that the code assigned to one character is not prefix of code assigned to any other character. This is how Huffman Coding makes sure that there is no ambiguity when decoding the generated bit stream.  
Let us understand prefix codes with a counter example. Let there be four characters a, b, c and d, and their corresponding variable length codes be 00, 01, 0 and 1. This coding leads to ambiguity because code assigned to c is prefix of codes assigned to a and b. If the compressed bit stream is 0001, the de-compressed output may be “cccd” or “ccb” or “acd” or “ab”.

See [this](http://en.wikipedia.org/wiki/Huffman_coding#Applications)for applications of Huffman Coding.

There are mainly two major parts in Huffman Coding  
**1)** Build a Huffman Tree from input characters.  
**2)** Traverse the Huffman Tree and assign codes to characters.

**Steps to build Huffman Tree**  
Input is array of unique characters along with their frequency of occurrences and output is Huffman Tree.

**1.** Create a leaf node for each unique character and build a min heap of all leaf nodes (Min Heap is used as a priority queue. The value of frequency field is used to compare two nodes in min heap. Initially, the least frequent character is at root)

**2.** Extract two nodes with the minimum frequency from the min heap.

**3.** Create a new internal node with frequency equal to the sum of the two nodes frequencies. Make the first extracted node as its left child and the other extracted node as its right child. Add this node to the min heap.

**4.** Repeat steps#2 and #3 until the heap contains only one node. The remaining node is the root node and the tree is complete.

Let us understand the algorithm with an example:

character Frequency

a 5

b 9

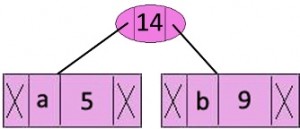
c 12

d 13

e 16

f 45

**Step 1.** Build a min heap that contains 6 nodes where each node represents root of a tree with single node.

**Step 2** Extract two minimum frequency nodes from min heap. Add a new internal node with frequency 5 + 9 = 14.  
[](http://d1hyf4ir1gqw6c.cloudfront.net/wp-content/uploads/fig-1.jpeg)  
Now min heap contains 5 nodes where 4 nodes are roots of trees with single element each, and one heap node is root of tree with 3 elements

character Frequency

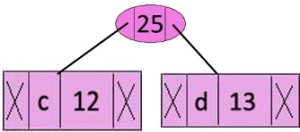
c 12

d 13

Internal Node 14

e 16

f 45

**Step 3:** Extract two minimum frequency nodes from heap. Add a new internal node with frequency 12 + 13 = 25  
[](http://d1hyf4ir1gqw6c.cloudfront.net/wp-content/uploads/fig-2.jpg)  
Now min heap contains 4 nodes where 2 nodes are roots of trees with single element each, and two heap nodes are root of tree with more than one nodes.

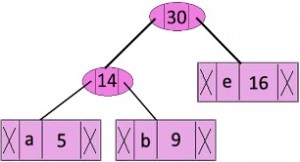
character Frequency

Internal Node 14

e 16

Internal Node 25

f 45

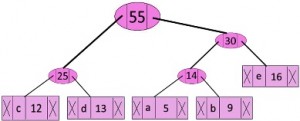
**Step 4:** Extract two minimum frequency nodes. Add a new internal node with frequency 14 + 16 = 30  
[](http://d1hyf4ir1gqw6c.cloudfront.net/wp-content/uploads/fig-3.jpg)  
Now min heap contains 3 nodes.

character Frequency

Internal Node 25

Internal Node 30

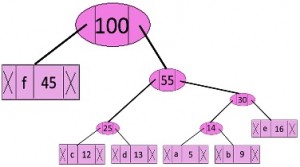
f 45

**Step 5:** Extract two minimum frequency nodes. Add a new internal node with frequency 25 + 30 = 55  
[](http://d1hyf4ir1gqw6c.cloudfront.net/wp-content/uploads/fig-4.jpg)  
Now min heap contains 2 nodes.

character Frequency

f 45

Internal Node 55

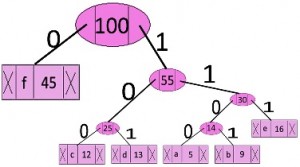
**Step 6:** Extract two minimum frequency nodes. Add a new internal node with frequency 45 + 55 = 100  
[](http://d1hyf4ir1gqw6c.cloudfront.net/wp-content/uploads/fig-5.jpg)  
Now min heap contains only one node.

character Frequency

Internal Node 100

Since the heap contains only one node, the algorithm stops here.

**Steps to print codes from Huffman Tree:**  
Traverse the tree formed starting from the root. Maintain an auxiliary array. While moving to the left child, write 0 to the array. While moving to the right child, write 1 to the array. Print the array when a leaf node is encountered.

[](http://d1hyf4ir1gqw6c.cloudfront.net/wp-content/uploads/fig-6.jpg)  
The codes are as follows:

character code-word

f 0

c 100

d 101

a 1100

b 1101

e 111

* C
* C++ using STL

|  |
| --- |
| // C program for Huffman Coding  #include <stdio.h>  #include <stdlib.h>    // This constant can be avoided by explicitly calculating height of Huffman Tree  #define MAX\_TREE\_HT 100    // A Huffman tree node  struct MinHeapNode  {      char data;  // One of the input characters      unsigned freq;  // Frequency of the character      struct MinHeapNode \*left, \*right; // Left and right child of this node  };    // A Min Heap:  Collection of min heap (or Hufmman tree) nodes  struct MinHeap  {      unsigned size;    // Current size of min heap      unsigned capacity;   // capacity of min heap      struct MinHeapNode \*\*array;  // Attay of minheap node pointers  };    // A utility function allocate a new min heap node with given character  // and frequency of the character  struct MinHeapNode\* newNode(char data, unsigned freq)  {      struct MinHeapNode\* temp =            (struct MinHeapNode\*) malloc(sizeof(struct MinHeapNode));      temp->left = temp->right = NULL;      temp->data = data;      temp->freq = freq;      return temp;  }    // A utility function to create a min heap of given capacity  struct MinHeap\* createMinHeap(unsigned capacity)  {      struct MinHeap\* minHeap =           (struct MinHeap\*) malloc(sizeof(struct MinHeap));      minHeap->size = 0;  // current size is 0      minHeap->capacity = capacity;      minHeap->array =       (struct MinHeapNode\*\*)malloc(minHeap->capacity \* sizeof(struct MinHeapNode\*));      return minHeap;  }    // A utility function to swap two min heap nodes  void swapMinHeapNode(struct MinHeapNode\*\* a, struct MinHeapNode\*\* b)  {      struct MinHeapNode\* t = \*a;      \*a = \*b;      \*b = t;  }    // The standard minHeapify function.  void minHeapify(struct MinHeap\* minHeap, int idx)  {      int smallest = idx;      int left = 2 \* idx + 1;      int right = 2 \* idx + 2;        if (left < minHeap->size &&          minHeap->array[left]->freq < minHeap->array[smallest]->freq)        smallest = left;        if (right < minHeap->size &&          minHeap->array[right]->freq < minHeap->array[smallest]->freq)        smallest = right;        if (smallest != idx)      {          swapMinHeapNode(&minHeap->array[smallest], &minHeap->array[idx]);          minHeapify(minHeap, smallest);      }  }    // A utility function to check if size of heap is 1 or not  int isSizeOne(struct MinHeap\* minHeap)  {      return (minHeap->size == 1);  }    // A standard function to extract minimum value node from heap  struct MinHeapNode\* extractMin(struct MinHeap\* minHeap)  {      struct MinHeapNode\* temp = minHeap->array[0];      minHeap->array[0] = minHeap->array[minHeap->size - 1];      --minHeap->size;      minHeapify(minHeap, 0);      return temp;  }    // A utility function to insert a new node to Min Heap  void insertMinHeap(struct MinHeap\* minHeap, struct MinHeapNode\* minHeapNode)  {      ++minHeap->size;      int i = minHeap->size - 1;      while (i && minHeapNode->freq < minHeap->array[(i - 1)/2]->freq)      {          minHeap->array[i] = minHeap->array[(i - 1)/2];          i = (i - 1)/2;      }      minHeap->array[i] = minHeapNode;  }    // A standard funvtion to build min heap  void buildMinHeap(struct MinHeap\* minHeap)  {      int n = minHeap->size - 1;      int i;      for (i = (n - 1) / 2; i >= 0; --i)          minHeapify(minHeap, i);  }    // A utility function to print an array of size n  void printArr(int arr[], int n)  {      int i;      for (i = 0; i < n; ++i)          printf("%d", arr[i]);      printf("\n");  }    // Utility function to check if this node is leaf  int isLeaf(struct MinHeapNode\* root)  {      return !(root->left) && !(root->right) ;  }    // Creates a min heap of capacity equal to size and inserts all character of  // data[] in min heap. Initially size of min heap is equal to capacity  struct MinHeap\* createAndBuildMinHeap(char data[], int freq[], int size)  {      struct MinHeap\* minHeap = createMinHeap(size);      for (int i = 0; i < size; ++i)          minHeap->array[i] = newNode(data[i], freq[i]);      minHeap->size = size;      buildMinHeap(minHeap);      return minHeap;  }    // The main function that builds Huffman tree  struct MinHeapNode\* buildHuffmanTree(char data[], int freq[], int size)  {      struct MinHeapNode \*left, \*right, \*top;        // Step 1: Create a min heap of capacity equal to size.  Initially, there are      // modes equal to size.      struct MinHeap\* minHeap = createAndBuildMinHeap(data, freq, size);        // Iterate while size of heap doesn't become 1      while (!isSizeOne(minHeap))      {          // Step 2: Extract the two minimum freq items from min heap          left = extractMin(minHeap);          right = extractMin(minHeap);            // Step 3:  Create a new internal node with frequency equal to the          // sum of the two nodes frequencies. Make the two extracted node as          // left and right children of this new node. Add this node to the min heap          // '$' is a special value for internal nodes, not used          top = newNode('$', left->freq + right->freq);          top->left = left;          top->right = right;          insertMinHeap(minHeap, top);      }        // Step 4: The remaining node is the root node and the tree is complete.      return extractMin(minHeap);  }    // Prints huffman codes from the root of Huffman Tree.  It uses arr[] to  // store codes  void printCodes(struct MinHeapNode\* root, int arr[], int top)  {      // Assign 0 to left edge and recur      if (root->left)      {          arr[top] = 0;          printCodes(root->left, arr, top + 1);      }        // Assign 1 to right edge and recur      if (root->right)      {          arr[top] = 1;          printCodes(root->right, arr, top + 1);      }        // If this is a leaf node, then it contains one of the input      // characters, print the character and its code from arr[]      if (isLeaf(root))      {          printf("%c: ", root->data);          printArr(arr, top);      }  }    // The main function that builds a Huffman Tree and print codes by traversing  // the built Huffman Tree  void HuffmanCodes(char data[], int freq[], int size)  {     //  Construct Huffman Tree     struct MinHeapNode\* root = buildHuffmanTree(data, freq, size);       // Print Huffman codes using the Huffman tree built above     int arr[MAX\_TREE\_HT], top = 0;     printCodes(root, arr, top);  }    // Driver program to test above functions  int main()  {      char arr[] = {'a', 'b', 'c', 'd', 'e', 'f'};      int freq[] = {5, 9, 12, 13, 16, 45};      int size = sizeof(arr)/sizeof(arr[0]);      HuffmanCodes(arr, freq, size);      return 0;  } |

Run on IDE

f: 0

c: 100

d: 101

a: 1100

b: 1101

e: 111

**Time complexity:** O(nlogn) where n is the number of unique characters. If there are n nodes, extractMin() is called 2\*(n – 1) times. extractMin() takes O(logn) time as it calles minHeapify(). So, overall complexity is O(nlogn).

If the input array is sorted, there exists a linear time algorithm. We will soon be discussing in our next post.

***Reference:***  
<http://en.wikipedia.org/wiki/Huffman_coding>

# Greedy Algorithms | Set 4 (Efficient Huffman Coding for Sorted Input)

We recommend to read following post as a prerequisite for this.

[Greedy Algorithms | Set 3 (Huffman Coding)](http://www.geeksforgeeks.org/archives/26851)

Time complexity of the algorithm discussed in above post is O(nLogn). If we know that the given array is sorted (by non-decreasing order of frequency), we can generate Huffman codes in O(n) time. Following is a O(n) algorithm for sorted input.

**1.** Create two empty queues.

**2.** Create a leaf node for each unique character and Enqueue it to the first queue in non-decreasing order of frequency. Initially second queue is empty.

**3.** Dequeue two nodes with the minimum frequency by examining the front of both queues. Repeat following steps two times  
…..**a)** If second queue is empty, dequeue from first queue.  
…..**b)** If first queue is empty, dequeue from second queue.  
…..**c)** Else, compare the front of two queues and dequeue the minimum.

**4.** Create a new internal node with frequency equal to the sum of the two nodes frequencies. Make the first Dequeued node as its left child and the second Dequeued node as right child. Enqueue this node to second queue.

**5.** Repeat steps#3 and #4 until there is more than one node in the queues. The remaining node is the root node and the tree is complete.

|  |
| --- |
| // C Program for Efficient Huffman Coding for Sorted input  #include <stdio.h>  #include <stdlib.h>    // This constant can be avoided by explicitly calculating height of Huffman Tree  #define MAX\_TREE\_HT 100    // A node of huffman tree  struct QueueNode  {      char data;      unsigned freq;      struct QueueNode \*left, \*right;  };    // Structure for Queue: collection of Huffman Tree nodes (or QueueNodes)  struct Queue  {      int front, rear;      int capacity;      struct QueueNode \*\*array;  };    // A utility function to create a new Queuenode  struct QueueNode\* newNode(char data, unsigned freq)  {      struct QueueNode\* temp =         (struct QueueNode\*) malloc(sizeof(struct QueueNode));      temp->left = temp->right = NULL;      temp->data = data;      temp->freq = freq;      return temp;  }    // A utility function to create a Queue of given capacity  struct Queue\* createQueue(int capacity)  {      struct Queue\* queue = (struct Queue\*) malloc(sizeof(struct Queue));      queue->front = queue->rear = -1;      queue->capacity = capacity;      queue->array =        (struct QueueNode\*\*) malloc(queue->capacity \* sizeof(struct QueueNode\*));      return queue;  }    // A utility function to check if size of given queue is 1  int isSizeOne(struct Queue\* queue)  {      return queue->front == queue->rear && queue->front != -1;  }    // A utility function to check if given queue is empty  int isEmpty(struct Queue\* queue)  {      return queue->front == -1;  }    // A utility function to check if given queue is full  int isFull(struct Queue\* queue)  {      return queue->rear == queue->capacity - 1;  }    // A utility function to add an item to queue  void enQueue(struct Queue\* queue, struct QueueNode\* item)  {      if (isFull(queue))          return;      queue->array[++queue->rear] = item;      if (queue->front == -1)          ++queue->front;  }    // A utility function to remove an item from queue  struct QueueNode\* deQueue(struct Queue\* queue)  {      if (isEmpty(queue))          return NULL;      struct QueueNode\* temp = queue->array[queue->front];      if (queue->front == queue->rear)  // If there is only one item in queue          queue->front = queue->rear = -1;      else          ++queue->front;      return temp;  }    // A utility function to get from of queue  struct QueueNode\* getFront(struct Queue\* queue)  {      if (isEmpty(queue))          return NULL;      return queue->array[queue->front];  }    /\* A function to get minimum item from two queues \*/  struct QueueNode\* findMin(struct Queue\* firstQueue, struct Queue\* secondQueue)  {      // Step 3.a: If second queue is empty, dequeue from first queue      if (isEmpty(firstQueue))          return deQueue(secondQueue);        // Step 3.b: If first queue is empty, dequeue from second queue      if (isEmpty(secondQueue))          return deQueue(firstQueue);        // Step 3.c:  Else, compare the front of two queues and dequeue minimum      if (getFront(firstQueue)->freq < getFront(secondQueue)->freq)          return deQueue(firstQueue);        return deQueue(secondQueue);  }    // Utility function to check if this node is leaf  int isLeaf(struct QueueNode\* root)  {      return !(root->left) && !(root->right) ;  }    // A utility function to print an array of size n  void printArr(int arr[], int n)  {      int i;      for (i = 0; i < n; ++i)          printf("%d", arr[i]);      printf("\n");  }    // The main function that builds Huffman tree  struct QueueNode\* buildHuffmanTree(char data[], int freq[], int size)  {      struct QueueNode \*left, \*right, \*top;        // Step 1: Create two empty queues      struct Queue\* firstQueue  = createQueue(size);      struct Queue\* secondQueue = createQueue(size);        // Step 2:Create a leaf node for each unique character and Enqueue it to      // the first queue in non-decreasing order of frequency. Initially second      // queue is empty      for (int i = 0; i < size; ++i)          enQueue(firstQueue, newNode(data[i], freq[i]));        // Run while Queues contain more than one node. Finally, first queue will      // be empty and second queue will contain only one node      while (!(isEmpty(firstQueue) && isSizeOne(secondQueue)))      {          // Step 3: Dequeue two nodes with the minimum frequency by examining          // the front of both queues          left = findMin(firstQueue, secondQueue);          right = findMin(firstQueue, secondQueue);            // Step 4: Create a new internal node with frequency equal to the sum          // of the two nodes frequencies. Enqueue this node to second queue.          top = newNode('$' , left->freq + right->freq);          top->left = left;          top->right = right;          enQueue(secondQueue, top);      }        return deQueue(secondQueue);  }    // Prints huffman codes from the root of Huffman Tree.  It uses arr[] to  // store codes  void printCodes(struct QueueNode\* root, int arr[], int top)  {      // Assign 0 to left edge and recur      if (root->left)      {          arr[top] = 0;          printCodes(root->left, arr, top + 1);      }        // Assign 1 to right edge and recur      if (root->right)      {          arr[top] = 1;          printCodes(root->right, arr, top + 1);      }        // If this is a leaf node, then it contains one of the input      // characters, print the character and its code from arr[]      if (isLeaf(root))      {          printf("%c: ", root->data);          printArr(arr, top);      }  }    // The main function that builds a Huffman Tree and print codes by traversing  // the built Huffman Tree  void HuffmanCodes(char data[], int freq[], int size)  {     //  Construct Huffman Tree     struct QueueNode\* root = buildHuffmanTree(data, freq, size);       // Print Huffman codes using the Huffman tree built above     int arr[MAX\_TREE\_HT], top = 0;     printCodes(root, arr, top);  }    // Driver program to test above functions  int main()  {      char arr[] = {'a', 'b', 'c', 'd', 'e', 'f'};      int freq[] = {5, 9, 12, 13, 16, 45};      int size = sizeof(arr)/sizeof(arr[0]);      HuffmanCodes(arr, freq, size);      return 0;  } |

Run on IDE

Output:

f: 0

c: 100

d: 101

a: 1100

b: 1101

e: 111

**Time complexity:** O(n)

If the input is not sorted, it need to be sorted first before it can be processed by the above algorithm. Sorting can be done using heap-sort or merge-sort both of which run in Theta(nlogn). So, the overall time complexity becomes O(nlogn) for unsorted input.

**Reference:**  
<http://en.wikipedia.org/wiki/Huffman_coding>

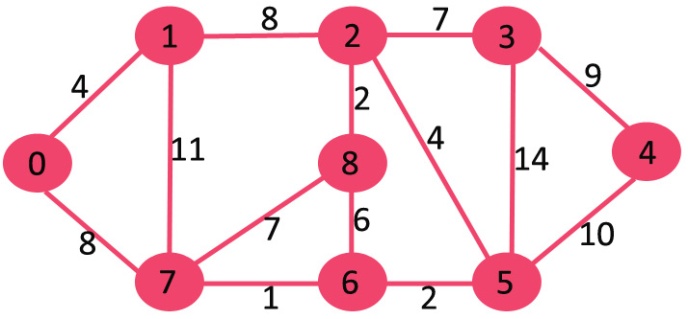
# Greedy Algorithms | Set 5 (Prim’s Minimum Spanning Tree (MST))

We have discussed [Kruskal’s algorithm for Minimum Spanning Tree](http://www.geeksforgeeks.org/archives/26604). Like Kruskal’s algorithm, Prim’s algorithm is also a [Greedy algorithm](http://www.geeksforgeeks.org/archives/18528). It starts with an empty spanning tree. The idea is to maintain two sets of vertices. The first set contains the vertices already included in the MST, the other set contains the vertices not yet included. At every step, it considers all the edges that connect the two sets, and picks the minimum weight edge from these edges. After picking the edge, it moves the other endpoint of the edge to the set containing MST.  
A group of edges that connects two set of vertices in a graph is called [cut in graph theory](http://en.wikipedia.org/wiki/Cut_%28graph_theory%29). So, at every step of Prim’s algorithm, we find a cut (of two sets, one contains the vertices already included in MST and other contains rest of the verices), pick the minimum weight edge from the cut and include this vertex to MST Set (the set that contains already included vertices).

**How does Prim’s Algorithm Work?** The idea behind Prim’s algorithm is simple, a spanning tree means all vertices must be connected. So the two disjoint subsets (discussed above) of vertices must be connected to make a Spanning Tree. And they must be connected with the minimum weight edge to make it a Minimum Spanning Tree.

**Algorithm**  
**1)** Create a set mstSet that keeps track of vertices already included in MST.  
**2)** Assign a key value to all vertices in the input graph. Initialize all key values as INFINITE. Assign key value as 0 for the first vertex so that it is picked first.  
**3)** While mstSet doesn’t include all vertices  
….**a)** Pick a vertex u which is not there in mstSetand has minimum key value.  
….**b)** Include uto mstSet.  
….**c)** Update key value of all adjacent vertices of u. To update the key values, iterate through all adjacent vertices. For every adjacent vertex v, if weight of edge u-v is less than the previous key value of v, update the key value as weight of u-v

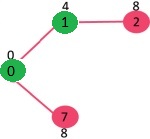
The idea of using key values is to pick the minimum weight edge from [cut](http://en.wikipedia.org/wiki/Cut_(graph_theory)). The key values are used only for vertices which are not yet included in MST, the key value for these vertices indicate the minimum weight edges connecting them to the set of vertices included in MST.

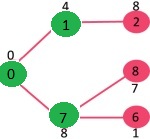
Let us understand with the following example:  
[](http://d1hyf4ir1gqw6c.cloudfront.net/wp-content/uploads/Fig-11.jpg)

The set mstSetis initially empty and keys assigned to vertices are {0, INF, INF, INF, INF, INF, INF, INF} where INF indicates infinite. Now pick the vertex with minimum key value. The vertex 0 is picked, include it in mstSet. So mstSetbecomes {0}. After including to mstSet, update key values of adjacent vertices. Adjacent vertices of 0 are 1 and 7. The key values of 1 and 7 are updated as 4 and 8. Following subgraph shows vertices and their key values, only the vertices with finite key values are shown. The vertices included in MST are shown in green color.

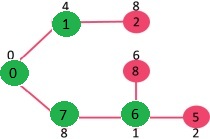
[](http://d1hyf4ir1gqw6c.cloudfront.net/wp-content/uploads/MST1.jpg)

Pick the vertex with minimum key value and not already included in MST (not in mstSET). The vertex 1 is picked and added to mstSet. So mstSet now becomes {0, 1}. Update the key values of adjacent vertices of 1. The key value of vertex 2 becomes 8.

[](http://d1hyf4ir1gqw6c.cloudfront.net/wp-content/uploads/MST2.jpg)

Pick the vertex with minimum key value and not already included in MST (not in mstSET). We can either pick vertex 7 or vertex 2, let vertex 7 is picked. So mstSet now becomes {0, 1, 7}. Update the key values of adjacent vertices of 7. The key value of vertex 6 and 8 becomes finite (7 and 1 respectively).  
[](http://d1hyf4ir1gqw6c.cloudfront.net/wp-content/uploads/MST3.jpg)

Pick the vertex with minimum key value and not already included in MST (not in mstSET). Vertex 6 is picked. So mstSet now becomes {0, 1, 7, 6}. Update the key values of adjacent vertices of 6. The key value of vertex 5 and 8 are updated.

[](http://d1hyf4ir1gqw6c.cloudfront.net/wp-content/uploads/MST4.jpg)

We repeat the above steps until mstSetincludes all vertices of given graph. Finally, we get the following graph.

[](http://d1hyf4ir1gqw6c.cloudfront.net/wp-content/uploads/MST5.jpg)

## [We strongly recommend that you click here and practice it, before moving on to the solution.](http://www.practice.geeksforgeeks.org/problem-page.php?pid=700343)

**How to implement the above algorithm?**  
We use a boolean array mstSet[] to represent the set of vertices included in MST. If a value mstSet[v] is true, then vertex v is included in MST, otherwise not. Array key[] is used to store key values of all vertices. Another array parent[] to store indexes of parent nodes in MST. The parent array is the output array which is used to show the constructed MST.

* C/C++
* Java

|  |
| --- |
| // A C / C++ program for Prim's Minimum Spanning Tree (MST) algorithm.  // The program is for adjacency matrix representation of the graph    #include <stdio.h>  #include <limits.h>    // Number of vertices in the graph  #define V 5    // A utility function to find the vertex with minimum key value, from  // the set of vertices not yet included in MST  int minKey(int key[], bool mstSet[])  {     // Initialize min value     int min = INT\_MAX, min\_index;       for (int v = 0; v < V; v++)       if (mstSet[v] == false && key[v] < min)           min = key[v], min\_index = v;       return min\_index;  }    // A utility function to print the constructed MST stored in parent[]  int printMST(int parent[], int n, int graph[V][V])  {     printf("Edge   Weight\n");     for (int i = 1; i < V; i++)        printf("%d - %d    %d \n", parent[i], i, graph[i][parent[i]]);  }    // Function to construct and print MST for a graph represented using adjacency  // matrix representation  void primMST(int graph[V][V])  {       int parent[V]; // Array to store constructed MST       int key[V];   // Key values used to pick minimum weight edge in cut       bool mstSet[V];  // To represent set of vertices not yet included in MST         // Initialize all keys as INFINITE       for (int i = 0; i < V; i++)          key[i] = INT\_MAX, mstSet[i] = false;         // Always include first 1st vertex in MST.       key[0] = 0;     // Make key 0 so that this vertex is picked as first vertex       parent[0] = -1; // First node is always root of MST         // The MST will have V vertices       for (int count = 0; count < V-1; count++)       {          // Pick the minimum key vertex from the set of vertices          // not yet included in MST          int u = minKey(key, mstSet);            // Add the picked vertex to the MST Set          mstSet[u] = true;            // Update key value and parent index of the adjacent vertices of          // the picked vertex. Consider only those vertices which are not yet          // included in MST          for (int v = 0; v < V; v++)               // graph[u][v] is non zero only for adjacent vertices of m             // mstSet[v] is false for vertices not yet included in MST             // Update the key only if graph[u][v] is smaller than key[v]            if (graph[u][v] && mstSet[v] == false && graph[u][v] <  key[v])               parent[v]  = u, key[v] = graph[u][v];       }         // print the constructed MST       printMST(parent, V, graph);  }      // driver program to test above function  int main()  {     /\* Let us create the following graph            2    3        (0)--(1)--(2)         |   / \   |        6| 8/   \5 |7         | /     \ |        (3)-------(4)              9          \*/     int graph[V][V] = {{0, 2, 0, 6, 0},                        {2, 0, 3, 8, 5},                        {0, 3, 0, 0, 7},                        {6, 8, 0, 0, 9},                        {0, 5, 7, 9, 0},                       };        // Print the solution      primMST(graph);        return 0;  } |

Run on IDE

Output:

Edge Weight

0 - 1 2

1 - 2 3

0 - 3 6

1 - 4 5

Time Complexity of the above program is O(V^2). If the input [graph is represented using adjacency list](http://www.geeksforgeeks.org/archives/27134), then the time complexity of Prim’s algorithm can be reduced to O(E log V) with the help of binary heap. Please see [Prim’s MST for Adjacency List Representation](http://www.geeksforgeeks.org/greedy-algorithms-set-5-prims-mst-for-adjacency-list-representation/) for more details.

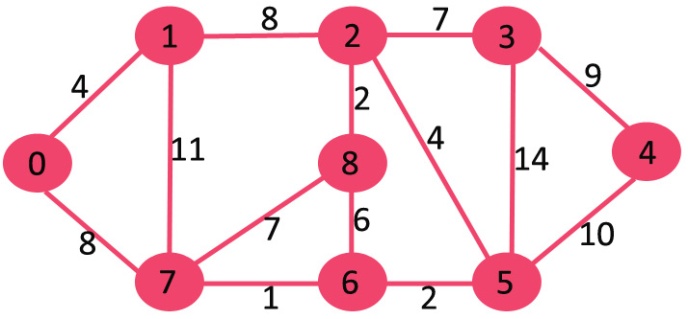
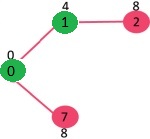
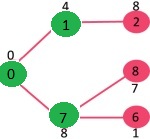
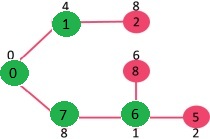
# Greedy Algorithms | Set 6 (Prim’s MST for Adjacency List Representation)

We recommend to read following two posts as a prerequisite of this post.

**1.** [Greedy Algorithms | Set 5 (Prim’s Minimum Spanning Tree (MST))](http://www.geeksforgeeks.org/archives/27455)  
**2.** [Graph and its representations](http://www.geeksforgeeks.org/archives/27134)

We have discussed [Prim’s algorithm and its implementation for adjacency matrix representation of graphs](http://www.geeksforgeeks.org/archives/27455). The time complexity for the matrix representation is O(V^2). In this post, O(ELogV) algorithm for adjacency list representation is discussed.  
As discussed in the previous post, in Prim’s algorithm, two sets are maintained, one set contains list of vertices already included in MST, other set contains vertices not yet included. With adjacency list representation, all vertices of a graph can be traversed in O(V+E) time using [BFS](http://www.geeksforgeeks.org/archives/18382). The idea is to traverse all vertices of graph using [BFS](http://www.geeksforgeeks.org/archives/18382)and use a Min Heap to store the vertices not yet included in MST. Min Heap is used as a priority queue to get the minimum weight edge from the [cut](http://en.wikipedia.org/wiki/Cut_%28graph_theory%29). Min Heap is used as time complexity of operations like extracting minimum element and decreasing key value is O(LogV) in Min Heap.

Following are the detailed steps.  
**1)**Create a Min Heap of size V where V is the number of vertices in the given graph. Every node of min heap contains vertex number and key value of the vertex.  
**2)** Initialize Min Heap with first vertex as root (the key value assigned to first vertex is 0). The key value assigned to all other vertices is INF (infinite).  
**3)**While Min Heap is not empty, do following  
…..**a)** Extract the min value node from Min Heap. Let the extracted vertex be u.  
…..**b)** For every adjacent vertex v of u, check if v is in Min Heap (not yet included in MST). If v is in Min Heap and its key value is more than weight of u-v, then update the key value of v as weight of u-v.

Let us understand the above algorithm with the following example:  
[](http://d1hyf4ir1gqw6c.cloudfront.net/wp-content/uploads/Fig-11.jpg)  
Initially, key value of first vertex is 0 and INF (infinite) for all other vertices. So vertex 0 is extracted from Min Heap and key values of vertices adjacent to 0 (1 and 7) are updated. Min Heap contains all vertices except vertex 0.  
The vertices in green color are the vertices included in MST.  
[](http://d1hyf4ir1gqw6c.cloudfront.net/wp-content/uploads/MST1.jpg)  
Since key value of vertex 1 is minimum among all nodes in Min Heap, it is extracted from Min Heap and key values of vertices adjacent to 1 are updated (Key is updated if the a vertex is not in Min Heap and previous key value is greater than the weight of edge from 1 to the adjacent). Min Heap contains all vertices except vertex 0 and 1.  
[](http://d1hyf4ir1gqw6c.cloudfront.net/wp-content/uploads/MST2.jpg)  
Since key value of vertex 7 is minimum among all nodes in Min Heap, it is extracted from Min Heap and key values of vertices adjacent to 7 are updated (Key is updated if the a vertex is not in Min Heap and previous key value is greater than the weight of edge from 7 to the adjacent). Min Heap contains all vertices except vertex 0, 1 and 7.  
[](http://d1hyf4ir1gqw6c.cloudfront.net/wp-content/uploads/MST3.jpg)  
Since key value of vertex 6 is minimum among all nodes in Min Heap, it is extracted from Min Heap and key values of vertices adjacent to 6 are updated (Key is updated if the a vertex is not in Min Heap and previous key value is greater than the weight of edge from 6 to the adjacent). Min Heap contains all vertices except vertex 0, 1, 7 and 6.  
[](http://d1hyf4ir1gqw6c.cloudfront.net/wp-content/uploads/MST4.jpg)  
The above steps are repeated for rest of the nodes in Min Heap till Min Heap becomes empty  
[](http://d1hyf4ir1gqw6c.cloudfront.net/wp-content/uploads/MST5.jpg)

|  |
| --- |
| // C / C++ program for Prim's MST for adjacency list representation of graph    #include <stdio.h>  #include <stdlib.h>  #include <limits.h>    // A structure to represent a node in adjacency list  struct AdjListNode  {      int dest;      int weight;      struct AdjListNode\* next;  };    // A structure to represent an adjacency liat  struct AdjList  {      struct AdjListNode \*head;  // pointer to head node of list  };    // A structure to represent a graph. A graph is an array of adjacency lists.  // Size of array will be V (number of vertices in graph)  struct Graph  {      int V;      struct AdjList\* array;  };    // A utility function to create a new adjacency list node  struct AdjListNode\* newAdjListNode(int dest, int weight)  {      struct AdjListNode\* newNode =              (struct AdjListNode\*) malloc(sizeof(struct AdjListNode));      newNode->dest = dest;      newNode->weight = weight;      newNode->next = NULL;      return newNode;  }    // A utility function that creates a graph of V vertices  struct Graph\* createGraph(int V)  {      struct Graph\* graph = (struct Graph\*) malloc(sizeof(struct Graph));      graph->V = V;        // Create an array of adjacency lists.  Size of array will be V      graph->array = (struct AdjList\*) malloc(V \* sizeof(struct AdjList));         // Initialize each adjacency list as empty by making head as NULL      for (int i = 0; i < V; ++i)          graph->array[i].head = NULL;        return graph;  }    // Adds an edge to an undirected graph  void addEdge(struct Graph\* graph, int src, int dest, int weight)  {      // Add an edge from src to dest.  A new node is added to the adjacency      // list of src.  The node is added at the begining      struct AdjListNode\* newNode = newAdjListNode(dest, weight);      newNode->next = graph->array[src].head;      graph->array[src].head = newNode;        // Since graph is undirected, add an edge from dest to src also      newNode = newAdjListNode(src, weight);      newNode->next = graph->array[dest].head;      graph->array[dest].head = newNode;  }    // Structure to represent a min heap node  struct MinHeapNode  {      int  v;      int key;  };    // Structure to represent a min heap  struct MinHeap  {      int size;      // Number of heap nodes present currently      int capacity;  // Capacity of min heap      int \*pos;     // This is needed for decreaseKey()      struct MinHeapNode \*\*array;  };    // A utility function to create a new Min Heap Node  struct MinHeapNode\* newMinHeapNode(int v, int key)  {      struct MinHeapNode\* minHeapNode =             (struct MinHeapNode\*) malloc(sizeof(struct MinHeapNode));      minHeapNode->v = v;      minHeapNode->key = key;      return minHeapNode;  }    // A utilit function to create a Min Heap  struct MinHeap\* createMinHeap(int capacity)  {      struct MinHeap\* minHeap =           (struct MinHeap\*) malloc(sizeof(struct MinHeap));      minHeap->pos = (int \*)malloc(capacity \* sizeof(int));      minHeap->size = 0;      minHeap->capacity = capacity;      minHeap->array =           (struct MinHeapNode\*\*) malloc(capacity \* sizeof(struct MinHeapNode\*));      return minHeap;  }    // A utility function to swap two nodes of min heap. Needed for min heapify  void swapMinHeapNode(struct MinHeapNode\*\* a, struct MinHeapNode\*\* b)  {      struct MinHeapNode\* t = \*a;      \*a = \*b;      \*b = t;  }    // A standard function to heapify at given idx  // This function also updates position of nodes when they are swapped.  // Position is needed for decreaseKey()  void minHeapify(struct MinHeap\* minHeap, int idx)  {      int smallest, left, right;      smallest = idx;      left = 2 \* idx + 1;      right = 2 \* idx + 2;        if (left < minHeap->size &&          minHeap->array[left]->key < minHeap->array[smallest]->key )        smallest = left;        if (right < minHeap->size &&          minHeap->array[right]->key < minHeap->array[smallest]->key )        smallest = right;        if (smallest != idx)      {          // The nodes to be swapped in min heap          MinHeapNode \*smallestNode = minHeap->array[smallest];          MinHeapNode \*idxNode = minHeap->array[idx];            // Swap positions          minHeap->pos[smallestNode->v] = idx;          minHeap->pos[idxNode->v] = smallest;            // Swap nodes          swapMinHeapNode(&minHeap->array[smallest], &minHeap->array[idx]);            minHeapify(minHeap, smallest);      }  }    // A utility function to check if the given minHeap is ampty or not  int isEmpty(struct MinHeap\* minHeap)  {      return minHeap->size == 0;  }    // Standard function to extract minimum node from heap  struct MinHeapNode\* extractMin(struct MinHeap\* minHeap)  {      if (isEmpty(minHeap))          return NULL;        // Store the root node      struct MinHeapNode\* root = minHeap->array[0];        // Replace root node with last node      struct MinHeapNode\* lastNode = minHeap->array[minHeap->size - 1];      minHeap->array[0] = lastNode;        // Update position of last node      minHeap->pos[root->v] = minHeap->size-1;      minHeap->pos[lastNode->v] = 0;        // Reduce heap size and heapify root      --minHeap->size;      minHeapify(minHeap, 0);        return root;  }    // Function to decreasy key value of a given vertex v. This function  // uses pos[] of min heap to get the current index of node in min heap  void decreaseKey(struct MinHeap\* minHeap, int v, int key)  {      // Get the index of v in  heap array      int i = minHeap->pos[v];        // Get the node and update its key value      minHeap->array[i]->key = key;        // Travel up while the complete tree is not hepified.      // This is a O(Logn) loop      while (i && minHeap->array[i]->key < minHeap->array[(i - 1) / 2]->key)      {          // Swap this node with its parent          minHeap->pos[minHeap->array[i]->v] = (i-1)/2;          minHeap->pos[minHeap->array[(i-1)/2]->v] = i;          swapMinHeapNode(&minHeap->array[i],  &minHeap->array[(i - 1) / 2]);            // move to parent index          i = (i - 1) / 2;      }  }    // A utility function to check if a given vertex  // 'v' is in min heap or not  bool isInMinHeap(struct MinHeap \*minHeap, int v)  {     if (minHeap->pos[v] < minHeap->size)       return true;     return false;  }    // A utility function used to print the constructed MST  void printArr(int arr[], int n)  {      for (int i = 1; i < n; ++i)          printf("%d - %d\n", arr[i], i);  }    // The main function that constructs Minimum Spanning Tree (MST)  // using Prim's algorithm  void PrimMST(struct Graph\* graph)  {      int V = graph->V;// Get the number of vertices in graph      int parent[V];   // Array to store constructed MST      int key[V];      // Key values used to pick minimum weight edge in cut        // minHeap represents set E      struct MinHeap\* minHeap = createMinHeap(V);        // Initialize min heap with all vertices. Key value of      // all vertices (except 0th vertex) is initially infinite      for (int v = 1; v < V; ++v)      {          parent[v] = -1;          key[v] = INT\_MAX;          minHeap->array[v] = newMinHeapNode(v, key[v]);          minHeap->pos[v] = v;      }        // Make key value of 0th vertex as 0 so that it      // is extracted first      key[0] = 0;      minHeap->array[0] = newMinHeapNode(0, key[0]);      minHeap->pos[0]   = 0;        // Initially size of min heap is equal to V      minHeap->size = V;        // In the followin loop, min heap contains all nodes      // not yet added to MST.      while (!isEmpty(minHeap))      {          // Extract the vertex with minimum key value          struct MinHeapNode\* minHeapNode = extractMin(minHeap);          int u = minHeapNode->v; // Store the extracted vertex number            // Traverse through all adjacent vertices of u (the extracted          // vertex) and update their key values          struct AdjListNode\* pCrawl = graph->array[u].head;          while (pCrawl != NULL)          {              int v = pCrawl->dest;                // If v is not yet included in MST and weight of u-v is              // less than key value of v, then update key value and              // parent of v              if (isInMinHeap(minHeap, v) && pCrawl->weight < key[v])              {                  key[v] = pCrawl->weight;                  parent[v] = u;                  decreaseKey(minHeap, v, key[v]);              }              pCrawl = pCrawl->next;          }      }        // print edges of MST      printArr(parent, V);  }    // Driver program to test above functions  int main()  {      // Let us create the graph given in above fugure      int V = 9;      struct Graph\* graph = createGraph(V);      addEdge(graph, 0, 1, 4);      addEdge(graph, 0, 7, 8);      addEdge(graph, 1, 2, 8);      addEdge(graph, 1, 7, 11);      addEdge(graph, 2, 3, 7);      addEdge(graph, 2, 8, 2);      addEdge(graph, 2, 5, 4);      addEdge(graph, 3, 4, 9);      addEdge(graph, 3, 5, 14);      addEdge(graph, 4, 5, 10);      addEdge(graph, 5, 6, 2);      addEdge(graph, 6, 7, 1);      addEdge(graph, 6, 8, 6);      addEdge(graph, 7, 8, 7);        PrimMST(graph);        return 0;  } |

Run on IDE

Output:

0 - 1

5 - 2

2 - 3

3 - 4

6 - 5

7 - 6

0 - 7

2 - 8

**Time Complexity:** The time complexity of the above code/algorithm looks O(V^2) as there are two nested while loops. If we take a closer look, we can observe that the statements in inner loop are executed O(V+E) times (similar to BFS). The inner loop has decreaseKey() operation which takes O(LogV) time. So overall time complexity is O(E+V)\*O(LogV) which is O((E+V)\*LogV) = O(ELogV) (For a connected graph, V = O(E))

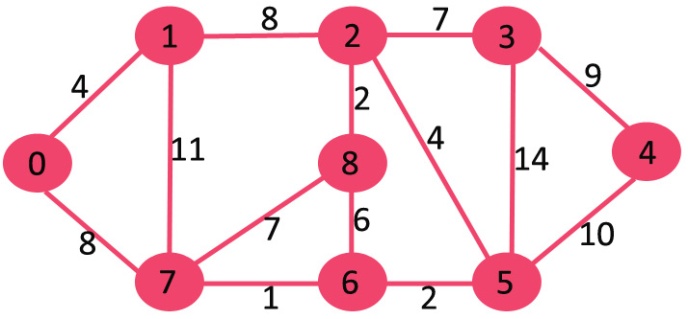
**References:**  
[Introduction to Algorithms by Clifford Stein, Thomas H. Cormen, Charles E. Leiserson, Ronald L.](http://www.flipkart.com/introduction-algorithms-8120340078/p/itmczynzhyhxv2gs?pid=9788120340077&affid=sandeepgfg)

# Greedy Algorithms | Set 7 (Dijkstra’s shortest path algorithm)

Given a graph and a source vertex in graph, find shortest paths from source to all vertices in the given graph.

Dijkstra’s algorithm is very similar to [Prim’s algorithm for minimum spanning tree](http://www.geeksforgeeks.org/archives/27455). Like Prim’s MST, we generate aSPT (shortest path tree) with given source as root. We maintain two sets, one set contains vertices included in shortest path tree, other set includes vertices not yet included in shortest path tree. At every step of the algorithm, we find a vertex which is in the other set (set of not yet included) and has minimum distance from source.

Below are the detailed steps used in Dijkstra’s algorithm to find the shortest path from a single source vertex to all other vertices in the given graph.  
Algorithm  
**1)** Create a set sptSet (shortest path tree set) that keeps track of vertices included in shortest path tree, i.e., whose minimum distance from source is calculated and finalized. Initially, this set is empty.  
**2)** Assign a distance value to all vertices in the input graph. Initialize all distance values as INFINITE. Assign distance value as 0 for the source vertex so that it is picked first.  
**3)** While sptSet doesn’t include all vertices  
….**a)** Pick a vertex u which is not there in sptSetand has minimum distance value.  
….**b)** Include u to sptSet.  
….**c)** Update distance value of all adjacent vertices of u. To update the distance values, iterate through all adjacent vertices. For every adjacent vertex v, if sum of distance value of u (from source) and weight of edge u-v, is less than the distance value of v, then update the distance value of v.

Let us understand with the following example:  
[](http://d1hyf4ir1gqw6c.cloudfront.net/wp-content/uploads/Fig-11.jpg)

The set sptSetis initially empty and distances assigned to vertices are {0, INF, INF, INF, INF, INF, INF, INF} where INF indicates infinite. Now pick the vertex with minimum distance value. The vertex 0 is picked, include it in sptSet. So sptSetbecomes {0}. After including 0 to sptSet, update distance values of its adjacent vertices. Adjacent vertices of 0 are 1 and 7. The distance values of 1 and 7 are updated as 4 and 8. Following subgraph shows vertices and their distance values, only the vertices with finite distance values are shown. The vertices included in SPT are shown in green color.

[](http://d1hyf4ir1gqw6c.cloudfront.net/wp-content/uploads/MST1.jpg)

Pick the vertex with minimum distance value and not already included in SPT (not in sptSET). The vertex 1 is picked and added to sptSet. So sptSet now becomes {0, 1}. Update the distance values of adjacent vertices of 1. The distance value of vertex 2 becomes 12.

[](http://d1hyf4ir1gqw6c.cloudfront.net/wp-content/uploads/DIJ2.jpg)

Pick the vertex with minimum distance value and not already included in SPT (not in sptSET). Vertex 7 is picked. So sptSet now becomes {0, 1, 7}. Update the distance values of adjacent vertices of 7. The distance value of vertex 6 and 8 becomes finite (15 and 9 respectively).  
[](http://d1hyf4ir1gqw6c.cloudfront.net/wp-content/uploads/DIJ3.jpg)

Pick the vertex with minimum distance value and not already included in SPT (not in sptSET). Vertex 6 is picked. So sptSet now becomes {0, 1, 7, 6}. Update the distance values of adjacent vertices of 6. The distance value of vertex 5 and 8 are updated.

[](http://d1hyf4ir1gqw6c.cloudfront.net/wp-content/uploads/DIJ4.jpg)

We repeat the above steps until sptSetdoesn’t include all vertices of given graph. Finally, we get the following Shortest Path Tree (SPT).

[](http://d1hyf4ir1gqw6c.cloudfront.net/wp-content/uploads/DIJ5.jpg)

**How to implement the above algorithm?**

## [We strongly recommend that you click here and practice it, before moving on to the solution.](http://www.practice.geeksforgeeks.org/problem-page.php?pid=700334)

We use a boolean array sptSet[] to represent the set of vertices included in SPT. If a value sptSet[v] is true, then vertex v is included in SPT, otherwise not. Array dist[] is used to store shortest distance values of all vertices.

* C/C++
* Java

|  |
| --- |
| // A C / C++ program for Dijkstra's single source shortest path algorithm.  // The program is for adjacency matrix representation of the graph    #include <stdio.h>  #include <limits.h>    // Number of vertices in the graph  #define V 9    // A utility function to find the vertex with minimum distance value, from  // the set of vertices not yet included in shortest path tree  int minDistance(int dist[], bool sptSet[])  {     // Initialize min value     int min = INT\_MAX, min\_index;       for (int v = 0; v < V; v++)       if (sptSet[v] == false && dist[v] <= min)           min = dist[v], min\_index = v;       return min\_index;  }    // A utility function to print the constructed distance array  int printSolution(int dist[], int n)  {     printf("Vertex   Distance from Source\n");     for (int i = 0; i < V; i++)        printf("%d \t\t %d\n", i, dist[i]);  }    // Funtion that implements Dijkstra's single source shortest path algorithm  // for a graph represented using adjacency matrix representation  void dijkstra(int graph[V][V], int src)  {       int dist[V];     // The output array.  dist[i] will hold the shortest                        // distance from src to i         bool sptSet[V]; // sptSet[i] will true if vertex i is included in shortest                       // path tree or shortest distance from src to i is finalized         // Initialize all distances as INFINITE and stpSet[] as false       for (int i = 0; i < V; i++)          dist[i] = INT\_MAX, sptSet[i] = false;         // Distance of source vertex from itself is always 0       dist[src] = 0;         // Find shortest path for all vertices       for (int count = 0; count < V-1; count++)       {         // Pick the minimum distance vertex from the set of vertices not         // yet processed. u is always equal to src in first iteration.         int u = minDistance(dist, sptSet);           // Mark the picked vertex as processed         sptSet[u] = true;           // Update dist value of the adjacent vertices of the picked vertex.         for (int v = 0; v < V; v++)             // Update dist[v] only if is not in sptSet, there is an edge from           // u to v, and total weight of path from src to  v through u is           // smaller than current value of dist[v]           if (!sptSet[v] && graph[u][v] && dist[u] != INT\_MAX                                         && dist[u]+graph[u][v] < dist[v])              dist[v] = dist[u] + graph[u][v];       }         // print the constructed distance array       printSolution(dist, V);  }    // driver program to test above function  int main()  {     /\* Let us create the example graph discussed above \*/     int graph[V][V] = {{0, 4, 0, 0, 0, 0, 0, 8, 0},                        {4, 0, 8, 0, 0, 0, 0, 11, 0},                        {0, 8, 0, 7, 0, 4, 0, 0, 2},                        {0, 0, 7, 0, 9, 14, 0, 0, 0},                        {0, 0, 0, 9, 0, 10, 0, 0, 0},                        {0, 0, 4, 14, 10, 0, 2, 0, 0},                        {0, 0, 0, 0, 0, 2, 0, 1, 6},                        {8, 11, 0, 0, 0, 0, 1, 0, 7},                        {0, 0, 2, 0, 0, 0, 6, 7, 0}                       };        dijkstra(graph, 0);        return 0;  } |

Run on IDE

Output:

Vertex Distance from Source

0 0

1 4

2 12

3 19

4 21

5 11

6 9

7 8

8 14

**Notes:**  
**1)** The code calculates shortest distance, but doesn’t calculate the path information. We can create a parent array, update the parent array when distance is updated (like [prim’s implementation](http://www.geeksforgeeks.org/archives/27455)) and use it show the shortest path from source to different vertices.

**2)** The code is for undirected graph, same dijekstra function can be used for directed graphs also.

**3)** The code finds shortest distances from source to all vertices. If we are interested only in shortest distance from source to a single target, we can break the for loop when the picked minimum distance vertex is equal to target (Step 3.a of algorithm).

**4)** Time Complexity of the implementation is O(V^2). If the input [graph is represented using adjacency list](http://www.geeksforgeeks.org/archives/27134), it can be reduced to O(E log V) with the help of binary heap. Please see  
[Dijkstra’s Algorithm for Adjacency List Representation](http://www.geeksforgeeks.org/greedy-algorithms-set-7-dijkstras-algorithm-for-adjacency-list-representation/) for more details.

**5)** Dijkstra’s algorithm doesn’t work for graphs with negative weight edges. For graphs with negative weight edges, [Bellman–Ford algorithm](http://en.wikipedia.org/wiki/Bellman-Ford_algorithm) can be used, we will soon be discussing it as a separate post.

[Dijkstra’s Algorithm for Adjacency List Representation](http://www.geeksforgeeks.org/greedy-algorithms-set-7-dijkstras-algorithm-for-adjacency-list-representation/)

[Printing Paths in Dijkstra’s Shortest Path Algorithm](http://www.geeksforgeeks.org/printing-paths-dijkstras-shortest-path-algorithm/)

[Dijkstra’s shortest path algorithm using set in STL](http://www.geeksforgeeks.org/dijkstras-shortest-path-algorithm-using-set-in-stl/)

# Greedy Algorithms | Set 8 (Dijkstra’s Algorithm for Adjacency List Representation)

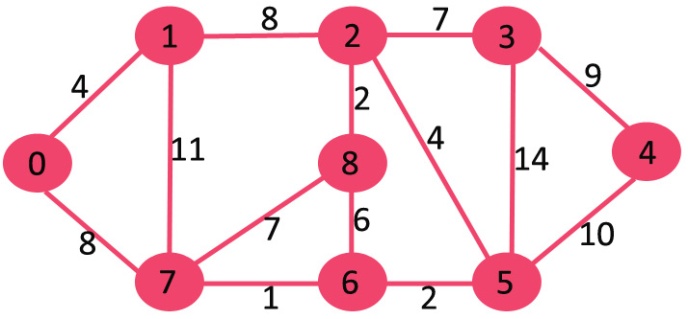
We recommend to read following two posts as a prerequisite of this post.

**1.** [Greedy Algorithms | Set 7 (Dijkstra’s shortest path algorithm)](http://www.geeksforgeeks.org/archives/27697)  
**2.** [Graph and its representations](http://www.geeksforgeeks.org/archives/27134)

We have discussed [Dijkstra’s algorithm and its implementation for adjacency matrix representation of graphs](http://www.geeksforgeeks.org/archives/27697). The time complexity for the matrix representation is O(V^2). In this post, O(ELogV) algorithm for adjacency list representation is discussed.

As discussed in the previous post, in Dijkstra’s algorithm, two sets are maintained, one set contains list of vertices already included in SPT (Shortest Path Tree), other set contains vertices not yet included. With adjacency list representation, all vertices of a graph can be traversed in O(V+E) time using [BFS](http://www.geeksforgeeks.org/archives/18382). The idea is to traverse all vertices of graph using [BFS](http://www.geeksforgeeks.org/archives/18382)and use a Min Heap to store the vertices not yet included in SPT (or the vertices for which shortest distance is not finalized yet).  Min Heap is used as a priority queue to get the minimum distance vertex from set of not yet included vertices. Time complexity of operations like extract-min and decrease-key value is O(LogV) for Min Heap.

Following are the detailed steps.  
**1)**Create a Min Heap of size V where V is the number of vertices in the given graph. Every node of min heap contains vertex number and distance value of the vertex.  
**2)** Initialize Min Heap with source vertex as root (the distance value assigned to source vertex is 0). The distance value assigned to all other vertices is INF (infinite).  
**3)**While Min Heap is not empty, do following  
…..**a)** Extract the vertex with minimum distance value node from Min Heap. Let the extracted vertex be u.  
…..**b)** For every adjacent vertex v of u, check if v is in Min Heap. If v is in Min Heap and distance value is more than weight of u-v plus distance value of u, then update the distance value of v.

Let us understand with the following example. Let the given source vertex be 0  
[](http://d1hyf4ir1gqw6c.cloudfront.net/wp-content/uploads/Fig-11.jpg)

Initially, distance value of source vertex is 0 and INF (infinite) for all other vertices. So source vertex is extracted from Min Heap and distance values of vertices adjacent to 0 (1 and 7) are updated. Min Heap contains all vertices except vertex 0.  
The vertices in green color are the vertices for which minimum distances are finalized and are not in Min Heap

[](http://d1hyf4ir1gqw6c.cloudfront.net/wp-content/uploads/MST1.jpg)

Since distance value of vertex 1 is minimum among all nodes in Min Heap, it is extracted from Min Heap and distance values of vertices adjacent to 1 are updated (distance is updated if the a vertex is not in Min Heap and distance through 1 is shorter than the previous distance). Min Heap contains all vertices except vertex 0 and 1.

[](http://d1hyf4ir1gqw6c.cloudfront.net/wp-content/uploads/DIJ2.jpg)

Pick the vertex with minimum distance value from min heap. Vertex 7 is picked. So min heap now contains all vertices except 0, 1 and 7. Update the distance values of adjacent vertices of 7. The distance value of vertex 6 and 8 becomes finite (15 and 9 respectively).  
[](http://d1hyf4ir1gqw6c.cloudfront.net/wp-content/uploads/DIJ3.jpg)

Pick the vertex with minimum distance from min heap. Vertex 6 is picked. So min heap now contains all vertices except 0, 1, 7 and 6. Update the distance values of adjacent vertices of 6. The distance value of vertex 5 and 8 are updated.

[](http://d1hyf4ir1gqw6c.cloudfront.net/wp-content/uploads/DIJ4.jpg)

Above steps are repeated till min heap doesn’t become empty. Finally, we get the following shortest path tree.

[](http://d1hyf4ir1gqw6c.cloudfront.net/wp-content/uploads/DIJ5.jpg)

|  |
| --- |
| // C / C++ program for Dijkstra's shortest path algorithm for adjacency  // list representation of graph    #include <stdio.h>  #include <stdlib.h>  #include <limits.h>    // A structure to represent a node in adjacency list  struct AdjListNode  {      int dest;      int weight;      struct AdjListNode\* next;  };    // A structure to represent an adjacency liat  struct AdjList  {      struct AdjListNode \*head;  // pointer to head node of list  };    // A structure to represent a graph. A graph is an array of adjacency lists.  // Size of array will be V (number of vertices in graph)  struct Graph  {      int V;      struct AdjList\* array;  };    // A utility function to create a new adjacency list node  struct AdjListNode\* newAdjListNode(int dest, int weight)  {      struct AdjListNode\* newNode =              (struct AdjListNode\*) malloc(sizeof(struct AdjListNode));      newNode->dest = dest;      newNode->weight = weight;      newNode->next = NULL;      return newNode;  }    // A utility function that creates a graph of V vertices  struct Graph\* createGraph(int V)  {      struct Graph\* graph = (struct Graph\*) malloc(sizeof(struct Graph));      graph->V = V;        // Create an array of adjacency lists.  Size of array will be V      graph->array = (struct AdjList\*) malloc(V \* sizeof(struct AdjList));         // Initialize each adjacency list as empty by making head as NULL      for (int i = 0; i < V; ++i)          graph->array[i].head = NULL;        return graph;  }    // Adds an edge to an undirected graph  void addEdge(struct Graph\* graph, int src, int dest, int weight)  {      // Add an edge from src to dest.  A new node is added to the adjacency      // list of src.  The node is added at the begining      struct AdjListNode\* newNode = newAdjListNode(dest, weight);      newNode->next = graph->array[src].head;      graph->array[src].head = newNode;        // Since graph is undirected, add an edge from dest to src also      newNode = newAdjListNode(src, weight);      newNode->next = graph->array[dest].head;      graph->array[dest].head = newNode;  }    // Structure to represent a min heap node  struct MinHeapNode  {      int  v;      int dist;  };    // Structure to represent a min heap  struct MinHeap  {      int size;      // Number of heap nodes present currently      int capacity;  // Capacity of min heap      int \*pos;     // This is needed for decreaseKey()      struct MinHeapNode \*\*array;  };    // A utility function to create a new Min Heap Node  struct MinHeapNode\* newMinHeapNode(int v, int dist)  {      struct MinHeapNode\* minHeapNode =             (struct MinHeapNode\*) malloc(sizeof(struct MinHeapNode));      minHeapNode->v = v;      minHeapNode->dist = dist;      return minHeapNode;  }    // A utility function to create a Min Heap  struct MinHeap\* createMinHeap(int capacity)  {      struct MinHeap\* minHeap =           (struct MinHeap\*) malloc(sizeof(struct MinHeap));      minHeap->pos = (int \*)malloc(capacity \* sizeof(int));      minHeap->size = 0;      minHeap->capacity = capacity;      minHeap->array =           (struct MinHeapNode\*\*) malloc(capacity \* sizeof(struct MinHeapNode\*));      return minHeap;  }    // A utility function to swap two nodes of min heap. Needed for min heapify  void swapMinHeapNode(struct MinHeapNode\*\* a, struct MinHeapNode\*\* b)  {      struct MinHeapNode\* t = \*a;      \*a = \*b;      \*b = t;  }    // A standard function to heapify at given idx  // This function also updates position of nodes when they are swapped.  // Position is needed for decreaseKey()  void minHeapify(struct MinHeap\* minHeap, int idx)  {      int smallest, left, right;      smallest = idx;      left = 2 \* idx + 1;      right = 2 \* idx + 2;        if (left < minHeap->size &&          minHeap->array[left]->dist < minHeap->array[smallest]->dist )        smallest = left;        if (right < minHeap->size &&          minHeap->array[right]->dist < minHeap->array[smallest]->dist )        smallest = right;        if (smallest != idx)      {          // The nodes to be swapped in min heap          MinHeapNode \*smallestNode = minHeap->array[smallest];          MinHeapNode \*idxNode = minHeap->array[idx];            // Swap positions          minHeap->pos[smallestNode->v] = idx;          minHeap->pos[idxNode->v] = smallest;            // Swap nodes          swapMinHeapNode(&minHeap->array[smallest], &minHeap->array[idx]);            minHeapify(minHeap, smallest);      }  }    // A utility function to check if the given minHeap is ampty or not  int isEmpty(struct MinHeap\* minHeap)  {      return minHeap->size == 0;  }    // Standard function to extract minimum node from heap  struct MinHeapNode\* extractMin(struct MinHeap\* minHeap)  {      if (isEmpty(minHeap))          return NULL;        // Store the root node      struct MinHeapNode\* root = minHeap->array[0];        // Replace root node with last node      struct MinHeapNode\* lastNode = minHeap->array[minHeap->size - 1];      minHeap->array[0] = lastNode;        // Update position of last node      minHeap->pos[root->v] = minHeap->size-1;      minHeap->pos[lastNode->v] = 0;        // Reduce heap size and heapify root      --minHeap->size;      minHeapify(minHeap, 0);        return root;  }    // Function to decreasy dist value of a given vertex v. This function  // uses pos[] of min heap to get the current index of node in min heap  void decreaseKey(struct MinHeap\* minHeap, int v, int dist)  {      // Get the index of v in  heap array      int i = minHeap->pos[v];        // Get the node and update its dist value      minHeap->array[i]->dist = dist;        // Travel up while the complete tree is not hepified.      // This is a O(Logn) loop      while (i && minHeap->array[i]->dist < minHeap->array[(i - 1) / 2]->dist)      {          // Swap this node with its parent          minHeap->pos[minHeap->array[i]->v] = (i-1)/2;          minHeap->pos[minHeap->array[(i-1)/2]->v] = i;          swapMinHeapNode(&minHeap->array[i],  &minHeap->array[(i - 1) / 2]);            // move to parent index          i = (i - 1) / 2;      }  }    // A utility function to check if a given vertex  // 'v' is in min heap or not  bool isInMinHeap(struct MinHeap \*minHeap, int v)  {     if (minHeap->pos[v] < minHeap->size)       return true;     return false;  }    // A utility function used to print the solution  void printArr(int dist[], int n)  {      printf("Vertex   Distance from Source\n");      for (int i = 0; i < n; ++i)          printf("%d \t\t %d\n", i, dist[i]);  }    // The main function that calulates distances of shortest paths from src to all  // vertices. It is a O(ELogV) function  void dijkstra(struct Graph\* graph, int src)  {      int V = graph->V;// Get the number of vertices in graph      int dist[V];      // dist values used to pick minimum weight edge in cut        // minHeap represents set E      struct MinHeap\* minHeap = createMinHeap(V);        // Initialize min heap with all vertices. dist value of all vertices      for (int v = 0; v < V; ++v)      {          dist[v] = INT\_MAX;          minHeap->array[v] = newMinHeapNode(v, dist[v]);          minHeap->pos[v] = v;      }        // Make dist value of src vertex as 0 so that it is extracted first      minHeap->array[src] = newMinHeapNode(src, dist[src]);      minHeap->pos[src]   = src;      dist[src] = 0;      decreaseKey(minHeap, src, dist[src]);        // Initially size of min heap is equal to V      minHeap->size = V;        // In the followin loop, min heap contains all nodes      // whose shortest distance is not yet finalized.      while (!isEmpty(minHeap))      {          // Extract the vertex with minimum distance value          struct MinHeapNode\* minHeapNode = extractMin(minHeap);          int u = minHeapNode->v; // Store the extracted vertex number            // Traverse through all adjacent vertices of u (the extracted          // vertex) and update their distance values          struct AdjListNode\* pCrawl = graph->array[u].head;          while (pCrawl != NULL)          {              int v = pCrawl->dest;                // If shortest distance to v is not finalized yet, and distance to v              // through u is less than its previously calculated distance              if (isInMinHeap(minHeap, v) && dist[u] != INT\_MAX &&                                            pCrawl->weight + dist[u] < dist[v])              {                  dist[v] = dist[u] + pCrawl->weight;                    // update distance value in min heap also                  decreaseKey(minHeap, v, dist[v]);              }              pCrawl = pCrawl->next;          }      }        // print the calculated shortest distances      printArr(dist, V);  }      // Driver program to test above functions  int main()  {      // create the graph given in above fugure      int V = 9;      struct Graph\* graph = createGraph(V);      addEdge(graph, 0, 1, 4);      addEdge(graph, 0, 7, 8);      addEdge(graph, 1, 2, 8);      addEdge(graph, 1, 7, 11);      addEdge(graph, 2, 3, 7);      addEdge(graph, 2, 8, 2);      addEdge(graph, 2, 5, 4);      addEdge(graph, 3, 4, 9);      addEdge(graph, 3, 5, 14);      addEdge(graph, 4, 5, 10);      addEdge(graph, 5, 6, 2);      addEdge(graph, 6, 7, 1);      addEdge(graph, 6, 8, 6);      addEdge(graph, 7, 8, 7);        dijkstra(graph, 0);        return 0;  } |

Run on IDE

Output:

Vertex Distance from Source

0 0

1 4

2 12

3 19

4 21

5 11

6 9

7 8

8 14

**Time Complexity:** The time complexity of the above code/algorithm looks O(V^2) as there are two nested while loops. If we take a closer look, we can observe that the statements in inner loop are executed O(V+E) times (similar to BFS). The inner loop has decreaseKey() operation which takes O(LogV) time. So overall time complexity is O(E+V)\*O(LogV) which is O((E+V)\*LogV) = O(ELogV)  
Note that the above code uses Binary Heap for Priority Queue implementation. Time complexity can be reduced to O(E + VLogV) using Fibonacci Heap. The reason is, Fibonacci Heap takes O(1) time for decrease-key operation while Binary Heap takes O(Logn) time.

**Notes:**  
**1)** The code calculates shortest distance, but doesn’t calculate the path information. We can create a parent array, update the parent array when distance is updated (like [prim’s implementation](http://www.geeksforgeeks.org/archives/27580)) and use it show the shortest path from source to different vertices.

**2)** The code is for undirected graph, same dijekstra function can be used for directed graphs also.

**3)** The code finds shortest distances from source to all vertices. If we are interested only in shortest distance from source to a single target, we can break the for loop when the picked minimum distance vertex is equal to target (Step 3.a of algorithm).

**4)** Dijkstra’s algorithm doesn’t work for graphs with negative weight edges. For graphs with negative weight edges, [Bellman–Ford algorithm](http://en.wikipedia.org/wiki/Bellman-Ford_algorithm) can be used, we will soon be discussing it as a separate post.

[Printing Paths in Dijkstra’s Shortest Path Algorithm](http://www.geeksforgeeks.org/printing-paths-dijkstras-shortest-path-algorithm/)  
[Dijkstra’s shortest path algorithm using set in STL](http://www.geeksforgeeks.org/dijkstras-shortest-path-algorithm-using-set-in-stl/)

**References:**  
[Introduction to Algorithms by Clifford Stein, Thomas H. Cormen, Charles E. Leiserson, Ronald L.](http://www.flipkart.com/introduction-algorithms-8120340078/p/itmczynzhyhxv2gs?pid=9788120340077&affid=sandeepgfg)[Algorithms by Sanjoy Dasgupta, Christos Papadimitriou, Umesh Vazirani](http://www.flipkart.com/algorithms-0070636613/p/itmczynvb7p2zacz?pid=9780070636613&affid=sandeepgfg)

# Job Sequencing Problem | Set 1 (Greedy Algorithm)

Given an array of jobs where every job has a deadline and associated profit if the job is finished before the deadline. It is also given that every job takes single unit of time, so the minimum possible deadline for any job is 1. How to maximize total profit if only one job can be scheduled at a time.

**Examples:**

Input: Four Jobs with following deadlines and profits

JobID Deadline Profit

a 4 20

b 1 10

c 1 40

d 1 30

Output: Following is maximum profit sequence of jobs

c, a

Input: Five Jobs with following deadlines and profits

JobID Deadline Profit

a 2 100

b 1 19

c 2 27

d 1 25

e 3 15

Output: Following is maximum profit sequence of jobs

c, a, e

**We strongly recommend to minimize your browser and try this yourself first.**

A **Simple Solution** is to generate all subsets of given set of jobs and check individual subset for feasibility of jobs in that subset. Keep track of maximum profit among all feasible subsets. The time complexity of this solution is exponential.

This is a standard [Greedy Algorithm](http://www.geeksforgeeks.org/greedy-algorithms-set-1-activity-selection-problem/)problem. Following is algorithm.

1) Sort all jobs in decreasing order of profit.

2) Initialize the result sequence as first job in sorted jobs.

3) Do following for remaining n-1 jobs

.......a) If the current job can fit in the current result sequence

without missing the deadline, add current job to the result.

Else ignore the current job.

The Following is C++ implementation of above algorithm.

|  |
| --- |
| // Program to find the maximum profit job sequence from a given array  // of jobs with deadlines and profits  #include<iostream>  #include<algorithm>  using namespace std;    // A structure to represent a job  struct Job  {     char id;      // Job Id     int dead;    // Deadline of job     int profit;  // Profit if job is over before or on deadline  };    // This function is used for sorting all jobs according to profit  bool comparison(Job a, Job b)  {       return (a.profit > b.profit);  }    // Returns minimum number of platforms reqquired  void printJobScheduling(Job arr[], int n)  {      // Sort all jobs according to decreasing order of prfit      sort(arr, arr+n, comparison);        int result[n]; // To store result (Sequence of jobs)      bool slot[n];  // To keep track of free time slots        // Initialize all slots to be free      for (int i=0; i<n; i++)          slot[i] = false;        // Iterate through all given jobs      for (int i=0; i<n; i++)      {         // Find a free slot for this job (Note that we start         // from the last possible slot)         for (int j=min(n, arr[i].dead)-1; j>=0; j--)         {            // Free slot found            if (slot[j]==false)            {               result[j] = i;  // Add this job to result               slot[j] = true; // Make this slot occupied               break;            }         }      }        // Print the result      for (int i=0; i<n; i++)         if (slot[i])           cout << arr[result[i]].id << " ";  }    // Driver program to test methods  int main()  {      Job arr[] = { {'a', 2, 100}, {'b', 1, 19}, {'c', 2, 27},                     {'d', 1, 25}, {'e', 3, 15}};      int n = sizeof(arr)/sizeof(arr[0]);      cout << "Following is maximum profit sequence of jobs\n";      printJobScheduling(arr, n);      return 0;  } |

Run on IDE

Output:

Following is maximum profit sequence of jobs

c a e

**Time Complexity** of the above solution is O(n2). It can be optimized using Disjoint Set Data Structure. Please refer below post for details.

[Job Sequencing Problem | Set 2 (Using Disjoint Set)](http://www.geeksforgeeks.org/job-sequencing-using-disjoint-set-union/)

**Sources:**  
<http://ocw.mit.edu/courses/civil-and-environmental-engineering/1-204-computer-algorithms-in-systems-engineering-spring-2010/lecture-notes/MIT1_204S10_lec10.pdf>

# Greedy Algorithm to find Minimum number of Coins

Given a value V, if we want to make change for V Rs, and we have infinite supply of each of the denominations in Indian currency, i.e., we have infinite supply of { 1, 2, 5, 10, 20, 50, 100, 500, 1000} valued coins/notes, what is the minimum number of coins and/or notes needed to make the change?

Examples:

Input: V = 70

Output: 2

We need a 50 Rs note and a 20 Rs note.

Input: V = 121

Output: 3

We need a 100 Rs note, a 20 Rs note and a

1 Rs coin.

**We strongly recommend you to minimize your browser and try this yourself first.**  
The idea is simple Greedy Algorithm. Start from largest possible denomination and keep adding denominations while remaining value is greater than 0. Below is complete algorithm.

1) Initialize result as empty.

2) find the largest denomination that is

smaller than V.

3) Add found denomination to result. Subtract

value of found denomination from V.

4) If V becomes 0, then print result.

Else repeat steps 2 and 3 for new value of V

Below is C++ implementation of above algorithm.

|  |
| --- |
| // C++ program to find minimum number of denominations  #include <bits/stdc++.h>  using namespace std;    // All denominations of Indian Currency  int deno[] = {1, 2, 5, 10, 20, 50, 100, 500, 1000};  int n = sizeof(deno)/sizeof(deno[0]);    // Driver program  void findMin(int V)  {      // Initialize result      vector<int> ans;        // Traverse through all denomination      for (int i=n-1; i>=0; i--)      {          // Find denominations          while (V >= deno[i])          {             V -= deno[i];             ans.push\_back(deno[i]);          }      }        // Print result      for (int i = 0; i < ans.size(); i++)             cout << ans[i] << "  ";  }    // Driver program  int main()  {     int n = 93;     cout << "Following is minimal number of change for " << n << " is ";     findMin(n);     return 0;  } |

Run on IDE

Output:

Following is minimal number of change for 93 is 50 20 20 2 1

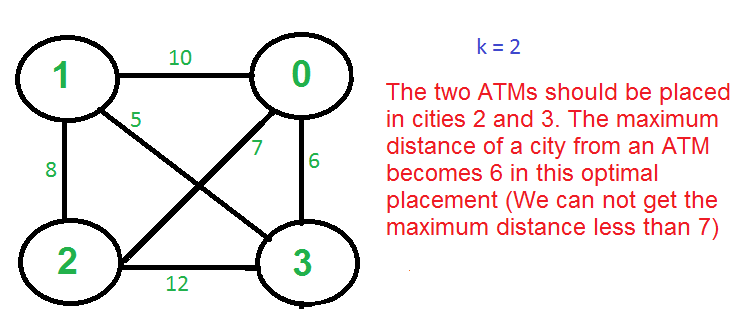
Note that above approach may not work for all denominations. For example, it doesn’t work for denominations {9, 6, 5, 1} and V = 11. The above approach would print 9, 1 and 1. But we can use 2 denominations 5 and 6.  
For general input, we use below dynamic programming approach.

[Find minimum number of coins that make a given value](http://www.geeksforgeeks.org/find-minimum-number-of-coins-that-make-a-change/)

# K Centers Problem | Set 1 (Greedy Approximate Algorithm)

Given n cities and distances between every pair of cities, select k cities to place warehouses (or ATMs or Cloud Server) such that the maximum distance of a city to a warehouse (or ATM or Cloud Server) is minimized.

For example consider the following four cities, 0, 1, 2 and 3 and distances between them, how do place 2 ATMs among these 4 cities so that the maximum distance of a city to an ATM is minimized.

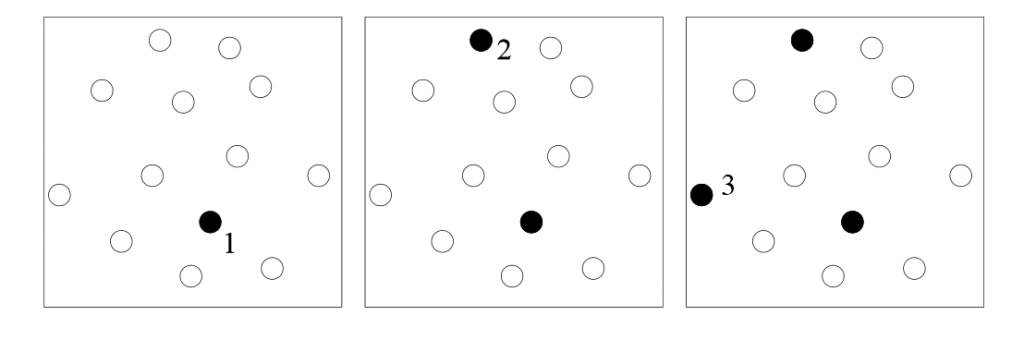
[](http://www.geeksforgeeks.org/k-centers-problem-set-1-greedy-approximate-algorithm/kcenters1/)

There is no polynomial time solution available for this problem as the problem is a known NP-Hard problem. There is a polynomial time Greedy approximate algorithm, the greedy algorithm provides a solution which is never worse that twice the optimal solution. The greedy solution works only if the distances between cities follow [Triangular Inequality](http://en.wikipedia.org/wiki/Triangle_inequality) (Distance between two points is always smaller than sum of distances through a third point).

**The 2-Approximate Greedy Algorithm:**  
1) Choose the first center arbitrarily.

2) Choose remaining k-1 centers using the following criteria.  
       Let c1, c2, c3, … ci be the already chosen centers. Choose  
       (i+1)’th center by picking the city which is farthest from already  
       selected centers, i.e, the point p which has following value as maximum  
                 Min[dist(p, c1), dist(p, c2), dist(p, c3), …. dist(p, ci)]

The following diagram taken from [here](http://algo2.iti.kit.edu/vanstee/courses/kcenter.pdf)illustrates above algorithm.

[](http://www.geeksforgeeks.org/k-centers-problem-set-1-greedy-approximate-algorithm/greedyalgo/)

**Example (k = 3 in the above shown Graph)**  
a) Let the first arbitrarily picked vertex be 0.

b) The next vertex is 1 because 1 is the farthest vertex from 0.

c) Remaining cities are 2 and 3. Calculate their distances from already selected centers (0 and 1). The greedy algorithm basically calculates following values.

        Minimum of all distanced from 2 to already considered centers  
        Min[dist(2, 0), dist(2, 1)] = Min[7, 8] = 7

        Minimum of all distanced from 3 to already considered centers  
        Min[dist(3, 0), dist(3, 1)] = Min[6, 5] = 5

        After computing the above values, the city 2 is picked as the value corresponding to 2 is maximum.

Note that the greedy algorithm doesn’t give best solution for k = 2 as this is just an approximate algorithm with bound as twice of optimal.

**Proof that the above greedy algorithm is 2 approximate.**  
Let OPT be the maximum distance of a city from a center in the Optimal solution. We need to show that the maximum distance obtained from Greedy algorithm is 2\*OPT.

The proof can be done using contradiction.

a) Assume that the distance from the furthest point to all centers is > 2·OPT.

b) This means that distances between all centers are also > 2·OPT.

c) We have k + 1 points with distances > 2·OPT between every pair.

d) Each point has a center of the optimal solution with distance <= OPT to it.

e) There exists a pair of points with the same center X in the optimal solution (pigeonhole principle: k optimal centers, k+1 points)

f) The distance between them is at most 2·OPT (triangle inequality) which is a contradiction.

**Source:**  
<http://algo2.iti.kit.edu/vanstee/courses/kcenter.pdf>

**Dynamic Programming**:

1. [Overlapping Subproblems Property](http://www.geeksforgeeks.org/dynamic-programming-set-1/)
2. [Optimal Substructure Property](http://www.geeksforgeeks.org/dynamic-programming-set-2-optimal-substructure-property/)
3. [Longest Increasing Subsequence](http://www.geeksforgeeks.org/dynamic-programming-set-3-longest-increasing-subsequence/)
4. [Longest Common Subsequence](http://www.geeksforgeeks.org/dynamic-programming-set-4-longest-common-subsequence/)
5. [Edit Distance](http://www.geeksforgeeks.org/dynamic-programming-set-5-edit-distance/)
6. [Min Cost Path](http://www.geeksforgeeks.org/dynamic-programming-set-6-min-cost-path/)
7. [Coin Change](http://www.geeksforgeeks.org/dynamic-programming-set-7-coin-change/)
8. [Matrix Chain Multiplication](http://www.geeksforgeeks.org/dynamic-programming-set-8-matrix-chain-multiplication/)
9. [Binomial Coefficient](http://www.geeksforgeeks.org/dynamic-programming-set-9-binomial-coefficient/)
10. [0-1 Knapsack Problem](http://www.geeksforgeeks.org/dynamic-programming-set-10-0-1-knapsack-problem/)
11. [Egg Dropping Puzzle](http://www.geeksforgeeks.org/dynamic-programming-set-11-egg-dropping-puzzle/)
12. [Longest Palindromic Subsequence](http://www.geeksforgeeks.org/dynamic-programming-set-12-longest-palindromic-subsequence/)
13. [Cutting a Rod](http://www.geeksforgeeks.org/dynamic-programming-set-13-cutting-a-rod/)
14. [Maximum Sum Increasing Subsequence](http://www.geeksforgeeks.org/dynamic-programming-set-14-maximum-sum-increasing-subsequence/)
15. [Longest Bitonic Subsequence](http://www.geeksforgeeks.org/dynamic-programming-set-15-longest-bitonic-subsequence/)
16. [Floyd Warshall Algorithm](http://www.geeksforgeeks.org/dynamic-programming-set-16-floyd-warshall-algorithm/)
17. [Palindrome Partitioning](http://www.geeksforgeeks.org/dynamic-programming-set-17-palindrome-partitioning/)
18. [Partition problem](http://www.geeksforgeeks.org/dynamic-programming-set-18-partition-problem/)
19. [Word Wrap Problem](http://www.geeksforgeeks.org/dynamic-programming-set-18-word-wrap/)
20. [Maximum Length Chain of Pairs](http://www.geeksforgeeks.org/dynamic-programming-set-20-maximum-length-chain-of-pairs/)
21. [Variations of LIS](http://www.geeksforgeeks.org/dynamic-programming-set-14-variations-of-lis/)
22. [Box Stacking Problem](http://www.geeksforgeeks.org/dynamic-programming-set-21-box-stacking-problem/)
23. [Program for Fibonacci numbers](http://www.geeksforgeeks.org/program-for-nth-fibonacci-number/)
24. [Minimum number of jumps to reach end](http://www.geeksforgeeks.org/minimum-number-of-jumps-to-reach-end-of-a-given-array/)
25. [Maximum size square sub-matrix with all 1s](http://www.geeksforgeeks.org/maximum-size-sub-matrix-with-all-1s-in-a-binary-matrix/)
26. [Ugly Numbers](http://www.geeksforgeeks.org/ugly-numbers/)
27. [Largest Sum Contiguous Subarray](http://www.geeksforgeeks.org/largest-sum-contiguous-subarray/)
28. [Longest Palindromic Substring](http://www.geeksforgeeks.org/longest-palindrome-substring-set-1/)
29. [Bellman–Ford Algorithm for Shortest Paths](http://www.geeksforgeeks.org/dynamic-programming-set-23-bellman-ford-algorithm/)
30. [Optimal Binary Search Tree](http://www.geeksforgeeks.org/dynamic-programming-set-24-optimal-binary-search-tree/)
31. [Largest Independent Set Problem](http://www.geeksforgeeks.org/largest-independent-set-problem/)
32. [Subset Sum Problem](http://www.geeksforgeeks.org/dynamic-programming-subset-sum-problem/)
33. [Maximum sum rectangle in a 2D matrix](http://www.geeksforgeeks.org/dynamic-programming-set-27-max-sum-rectangle-in-a-2d-matrix/)
34. [Count number of binary strings without consecutive 1?s](http://www.geeksforgeeks.org/count-number-binary-strings-without-consecutive-1s/)
35. [Boolean Parenthesization Problem](http://www.geeksforgeeks.org/dynamic-programming-set-37-boolean-parenthesization-problem/)
36. [Count ways to reach the n’th stair](http://www.geeksforgeeks.org/count-ways-reach-nth-stair/)
37. [Minimum Cost Polygon Triangulation](http://www.geeksforgeeks.org/minimum-cost-polygon-triangulation/)
38. [Mobile Numeric Keypad Problem](http://www.geeksforgeeks.org/mobile-numeric-keypad-problem/)
39. [Count of n digit numbers whose sum of digits equals to given sum](http://www.geeksforgeeks.org/count-of-n-digit-numbers-whose-sum-of-digits-equals-to-given-sum/)
40. [Minimum Initial Points to Reach Destination](http://www.geeksforgeeks.org/minimum-positive-points-to-reach-destination/)
41. [Total number of non-decreasing numbers with n digits](http://www.geeksforgeeks.org/total-number-of-non-decreasing-numbers-with-n-digits/)
42. [Find length of the longest consecutive path from a given starting character](http://www.geeksforgeeks.org/find-length-of-the-longest-consecutive-path-in-a-character-matrix/)
43. [Tiling Problem](http://www.geeksforgeeks.org/tiling-problem/)
44. [Minimum number of squares whose sum equals to given number n](http://www.geeksforgeeks.org/minimum-number-of-squares-whose-sum-equals-to-given-number-n/)
45. [Find minimum number of coins that make a given value](http://www.geeksforgeeks.org/find-minimum-number-of-coins-that-make-a-change/)
46. [Collect maximum points in a grid using two traversals](http://www.geeksforgeeks.org/collect-maximum-points-in-a-grid-using-two-traversals/)
47. [Shortest Common Supersequence](http://www.geeksforgeeks.org/shortest-common-supersequence/)
48. [Compute sum of digits in all numbers from 1 to n](http://www.geeksforgeeks.org/count-sum-of-digits-in-numbers-from-1-to-n/)
49. [Count possible ways to construct buildings](http://www.geeksforgeeks.org/count-possible-ways-to-construct-buildings/)
50. [Maximum profit by buying and selling a share at most twice](http://www.geeksforgeeks.org/maximum-profit-by-buying-and-selling-a-share-at-most-twice/)
51. [How to print maximum number of A’s using given four keys](http://www.geeksforgeeks.org/how-to-print-maximum-number-of-a-using-given-four-keys/)
52. [Find the minimum cost to reach destination using a train](http://www.geeksforgeeks.org/find-the-minimum-cost-to-reach-a-destination-where-every-station-is-connected-in-one-direction/)
53. [Vertex Cover Problem | Set 2 (Dynamic Programming Solution for Tree)](http://www.geeksforgeeks.org/vertex-cover-problem-set-2-dynamic-programming-solution-tree/)
54. [Count number of ways to reach a given score in a game](http://www.geeksforgeeks.org/count-number-ways-reach-given-score-game/)
55. [Weighted Job Scheduling](http://www.geeksforgeeks.org/weighted-job-scheduling/)
56. [Longest Even Length Substring such that Sum of First and Second Half is same](http://www.geeksforgeeks.org/longest-even-length-substring-sum-first-second-half/)

# Dynamic Programming | Set 1 (Overlapping Subproblems Property)

Dynamic Programming is an algorithmic paradigm that solves a given complex problem by breaking it into subproblems and stores the results of subproblems to avoid computing the same results again. Following are the two main properties of a problem that suggest that the given problem can be solved using Dynamic programming.

In this post, we will discuss first property (Overlapping Subproblems) in detail. The second property of Dynamic programming is discussed in next post i.e. [Set 2](http://www.geeksforgeeks.org/dynamic-programming-set-2-optimal-substructure-property/).

1) Overlapping Subproblems  
2) Optimal Substructure

**1) Overlapping Subproblems:**  
Like Divide and Conquer, Dynamic Programming combines solutions to sub-problems. Dynamic Programming is mainly used when solutions of same subproblems are needed again and again. In dynamic programming, computed solutions to subproblems are stored in a table so that these don’t have to recomputed. So Dynamic Programming is not useful when there are no common (overlapping) subproblems because there is no point storing the solutions if they are not needed again. For example, [Binary Search](http://en.wikipedia.org/wiki/Binary_search_algorithm) doesn’t have common subproblems. If we take example of following recursive program for Fibonacci Numbers, there are many subproblems which are solved again and again.

|  |
| --- |
| /\* simple recursive program for Fibonacci numbers \*/  int fib(int n)  {     if ( n <= 1 )        return n;     return fib(n-1) + fib(n-2);  } |

Run on IDE

Recursion tree for execution of fib(5)

fib(5)

/ \

fib(4) fib(3)

/ \ / \

fib(3) fib(2) fib(2) fib(1)

/ \ / \ / \

fib(2) fib(1) fib(1) fib(0) fib(1) fib(0)

/ \

fib(1) fib(0)

We can see that the function f(3) is being called 2 times. If we would have stored the value of f(3), then instead of computing it again, we could have reused the old stored value. There are following two different ways to store the values so that these values can be reused:  
a) Memoization (Top Down)  
b) Tabulation (Bottom Up)

**a) Memoization (Top Down):**The memoized program for a problem is similar to the recursive version with a small modification that it looks into a lookup table before computing solutions. We initialize a lookup array with all initial values as NIL. Whenever we need solution to a subproblem, we first look into the lookup table. If the precomputed value is there then we return that value, otherwise we calculate the value and put the result in lookup table so that it can be reused later.

Following is the memoized version for nth Fibonacci Number.

* C/C++
* Python

|  |
| --- |
| /\* C/C++ program for Memoized version for nth Fibonacci number \*/  #include<stdio.h>  #define NIL -1  #define MAX 100    int lookup[MAX];    /\* Function to initialize NIL values in lookup table \*/  void \_initialize()  {    int i;    for (i = 0; i < MAX; i++)      lookup[i] = NIL;  }    /\* function for nth Fibonacci number \*/  int fib(int n)  {     if (lookup[n] == NIL)     {        if (n <= 1)           lookup[n] = n;        else           lookup[n] = fib(n-1) + fib(n-2);     }       return lookup[n];  }    int main ()  {    int n = 40;    \_initialize();    printf("Fibonacci number is %d ", fib(n));    return 0;  } |

Run on IDE

**b) Tabulation (Bottom Up):**The tabulated program for a given problem builds a table in bottom up fashion and returns the last entry from table. For example, for the same Fibonacci number, we first calculate fib(0) then fib(1) then fib(2) then fib(3) and so on. So literally, we are building the solutions of subproblems bottom-up.

Following is the tabulated version for nth Fibonacci Number.

* C/C++
* Python

|  |
| --- |
| /\* C program for Tabulated version \*/  #include<stdio.h>  int fib(int n)  {    int f[n+1];    int i;    f[0] = 0;   f[1] = 1;    for (i = 2; i <= n; i++)        f[i] = f[i-1] + f[i-2];      return f[n];  }    int main ()  {    int n = 9;    printf("Fibonacci number is %d ", fib(n));    return 0;  } |

Run on IDE

Output:

Fibonacci number is 34

Both Tabulated and Memoized store the solutions of subproblems. In Memoized version, table is filled on demand while in Tabulated version, starting from the first entry, all entries are filled one by one. Unlike the Tabulated version, all entries of the lookup table are not necessarily filled in Memoized version. For example,[Memoized solution](https://www.ics.uci.edu/~eppstein/161/960229.html)of the [LCS problem](http://en.wikipedia.org/wiki/Longest_common_subsequence_problem)doesn’t necessarily fill all entries.

To see the optimization achieved by Memoized and Tabulated solutions over the basic Recursive solution, see the time taken by following runs for calculating 40th Fibonacci number:

[Recursive solution](http://code.geeksforgeeks.org/vHt6ly)  
[Memoized solution](http://code.geeksforgeeks.org/Z94jYR)  
[Tabulated solution](http://code.geeksforgeeks.org/12C5bP)

Time taken by Recursion method is much more than the two Dynamic Programming techniques mentioned above – Memoization and Tabulation!

Also see method 2 of [Ugly Number post](http://geeksforgeeks.org/?p=753) for one more simple example where we have overlapping subproblems and we store the results of subproblems.

We will be covering Optimal Substructure Property and some more example problems in future posts on Dynamic Programming.

Try following questions as an exercise of this post.  
1) Write a Memoized solution for LCS problem. Note that the Tabular solution is given in the CLRS book.  
2) How would you choose between Memoization and Tabulation?

# Dynamic Programming | Set 2 (Optimal Substructure Property)

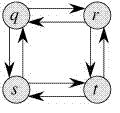
As we discussed in [Set 1](http://geeksforgeeks.org/?p=12635), following are the two main properties of a problem that suggest that the given problem can be solved using Dynamic programming:  
1) Overlapping Subproblems  
2) Optimal Substructure

We have already discussed Overlapping Subproblem property in the [Set 1](http://geeksforgeeks.org/?p=12635). Let us discuss Optimal Substructure property here.

**2) Optimal Substructure:**A given problems has Optimal Substructure Property if optimal solution of the given problem can be obtained by using optimal solutions of its subproblems.

For example, the Shortest Path problem has following optimal substructure property:  
If a node x lies in the shortest path from a source node u to destination node v then the shortest path from u to v is combination of shortest path from u to x and shortest path from x to v. The standard All Pair Shortest Path algorithms like [Floyd–Warshall](http://en.wikipedia.org/wiki/Floyd%E2%80%93Warshall_algorithm) and [Bellman–Ford](http://en.wikipedia.org/wiki/Bellman%E2%80%93Ford_algorithm)are typical examples of Dynamic Programming.

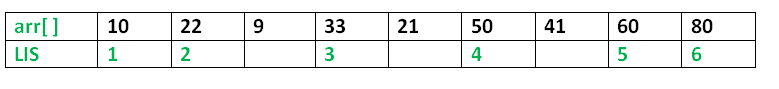
On the other hand, the Longest Path problem doesn’t have the Optimal Substructure property. Here by Longest Path we mean longest simple path (path without cycle) between two nodes. Consider the following unweighted graph given in the [CLRS book](http://mitpress.mit.edu/catalog/item/default.asp?ttype=2&tid=11866). There are two longest paths from q to t: q→r→t and q→s→t. Unlike shortest paths, these longest paths do not have the optimal substructure property. For example, the longest path q→r→t is not a combination of longest path from q to r and longest path from r to t, because the longest path from q to r is q→s→t→r and the longest path from r to t is r→q→s→t.

[](http://geeksforgeeks.org/wp-content/uploads/LongestPath.gif)

We will be covering some example problems in future posts on [Dynamic Programming](http://www.geeksforgeeks.org/fundamentals-of-algorithms/#DynamicProgramming).

# Dynamic Programming | Set 3 (Longest Increasing Subsequence)

We have discussed Overlapping Subproblems and Optimal Substructure properties in [Set 1](http://geeksforgeeks.org/?p=12635) and [Set 2](http://geeksforgeeks.org/?p=12819) respectively.

Let us discuss Longest Increasing Subsequence (LIS) problem as an example problem that can be solved using Dynamic Programming.  
The Longest Increasing Subsequence (LIS) problem is to find the length of the longest subsequence of a given sequence such that all elements of the subsequence are sorted in increasing order. For example, the length of LIS for {10, 22, 9, 33, 21, 50, 41, 60, 80} is 6 and LIS is {10, 22, 33, 50, 60, 80}.  


More Examples:

Input : arr[] = {3, 10, 2, 1, 20}

Output : Length of LIS = 3

The longest increasing subsequence is 3, 10, 20

Input : arr[] = {3, 2}

Output : Length of LIS = 1

The longest increasing subsequences are {3} and {2}

Input : arr[] = {50, 3, 10, 7, 40, 80}

Output : Length of LIS = 4

The longest increasing subsequence is {3, 7, 40, 80}

## [We strongly recommend that you click here and practice it, before moving on to the solution.](http://www.practice.geeksforgeeks.org/problem-page.php?pid=134)

**Optimal Substructure:**  
Let arr[0..n-1] be the input array and L(i) be the length of the LIS ending at index i such that arr[i] is the last element of the LIS.  
Then, L(i) can be recursively written as:  
L(i) = 1 + max( L(j) ) where 0 < j < i and arr[j] < arr[i]; or  
L(i) = 1, if no such j exists.  
To find the LIS for a given array, we need to return max(L(i)) where 0 < i < n.  
Thus, we see the LIS problem satisfies the optimal substructure property as the main problem can be solved using solutions to subproblems.

Following is a simple recursive implementation of the LIS problem. It follows the recursive structure discussed above.

* C/C++
* Java
* Python

|  |
| --- |
| // A naive C/C++ based recursive implementation of LIS problem  #include<stdio.h>  #include<stdlib.h>    // Recursive implementation for calculating the LIS  int \_lis(int arr[], int n, int \*max\_lis\_length)  {      // Base case      if (n == 1)          return 1;        int current\_lis\_length = 1;      for (int i=0; i<n-1; i++)      {          // Recursively calculate the length of the LIS          // ending at arr[i]          int subproblem\_lis\_length = \_lis(arr, i, max\_lis\_length);            // Check if appending arr[n-1] to the LIS          // ending at arr[i] gives us an LIS ending at          // arr[n-1] which is longer than the previously          // calculated LIS ending at arr[n-1]          if (arr[i] < arr[n-1] &&              current\_lis\_length < (1+subproblem\_lis\_length))              current\_lis\_length = 1+subproblem\_lis\_length;      }        // Check if currently calculated LIS ending at      // arr[n-1] is longer than the previously calculated      // LIS and update max\_lis\_length accordingly      if (\*max\_lis\_length < current\_lis\_length)          \*max\_lis\_length = current\_lis\_length;        return current\_lis\_length;  }    // The wrapper function for \_lis()  int lis(int arr[], int n)  {      int max\_lis\_length = 1; // stores the final LIS        // max\_lis\_length is passed as a reference below      // so that it can maintain its value      // between the recursive calls      \_lis( arr, n, &max\_lis\_length );        return max\_lis\_length;  }    // Driver program to test the functions above  int main()  {      int arr[] = {10, 22, 9, 33, 21, 50, 41, 60};      int n = sizeof(arr)/sizeof(arr[0]);      printf("Length of LIS is %d\n", lis( arr, n ));      return 0;  } |

Run on IDE

Output:

Length of LIS is 5

**Overlapping Subproblems:**  
Considering the above implementation, following is recursion tree for an array of size 4. lis(n) gives us the length of LIS for arr[].

lis(4)

/ | \

lis(3) lis(2) lis(1)

/ \ /

lis(2) lis(1) lis(1)

/

lis(1)

We can see that there are many subproblems which are solved again and again. So this problem has Overlapping Substructure property and recomputation of same subproblems can be avoided by either using Memoization or Tabulation. Following is a tabluated implementation for the LIS problem.

* C/C++
* Java
* Python

|  |
| --- |
| /\* Dynamic Programming C/C++ implementation of LIS problem \*/  #include<stdio.h>  #include<stdlib.h>    /\* lis() returns the length of the longest increasing    subsequence in arr[] of size n \*/  int lis( int arr[], int n )  {      int \*lis, i, j, max = 0;      lis = (int\*) malloc ( sizeof( int ) \* n );        /\* Initialize LIS values for all indexes \*/      for (i = 0; i < n; i++ )          lis[i] = 1;        /\* Compute optimized LIS values in bottom up manner \*/      for (i = 1; i < n; i++ )          for (j = 0; j < i; j++ )              if ( arr[i] > arr[j] && lis[i] < lis[j] + 1)                  lis[i] = lis[j] + 1;        /\* Pick maximum of all LIS values \*/      for (i = 0; i < n; i++ )          if (max < lis[i])              max = lis[i];        /\* Free memory to avoid memory leak \*/      free(lis);        return max;  }    /\* Driver program to test above function \*/  int main()  {      int arr[] = { 10, 22, 9, 33, 21, 50, 41, 60 };      int n = sizeof(arr)/sizeof(arr[0]);      printf("Length of lis is %d\n", lis( arr, n ) );      return 0;  } |

Run on IDE

Output:

Length of lis is 5

Note that the time complexity of the above Dynamic Programming (DP) solution is O(n^2) and there is a O(nLogn) solution for the LIS problem. We have not discussed the O(n Log n) solution here as the purpose of this post is to explain Dynamic Programming with a simple example. See below post for O(n Log n) solution.

[Longest Increasing Subsequence Size (N log N)](http://www.geeksforgeeks.org/longest-monotonically-increasing-subsequence-size-n-log-n/)

# Dynamic Programming | Set 4 (Longest Common Subsequence)

We have discussed Overlapping Subproblems and Optimal Substructure properties in [Set 1](http://geeksforgeeks.org/?p=12635) and [Set 2](http://geeksforgeeks.org/?p=12819) respectively. We also discussed one example problem in [Set 3](http://geeksforgeeks.org/?p=12832). Let us discuss Longest Common Subsequence (LCS) problem as one more example problem that can be solved using Dynamic Programming.

LCS Problem Statement: Given two sequences, find the length of longest subsequence present in both of them. A subsequence is a sequence that appears in the same relative order, but not necessarily contiguous. For example, “abc”, “abg”, “bdf”, “aeg”, ‘”acefg”, .. etc are subsequences of “abcdefg”. So a string of length n has 2^n different possible subsequences.

It is a classic computer science problem, the basis of [diff](http://en.wikipedia.org/wiki/Diff)(a file comparison program that outputs the differences between two files), and has applications in bioinformatics.

**Examples:**  
LCS for input Sequences “ABCDGH” and “AEDFHR” is “ADH” of length 3.  
LCS for input Sequences “AGGTAB” and “GXTXAYB” is “GTAB” of length 4.

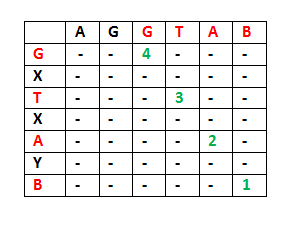
## [We strongly recommend that you click here and practice it, before moving on to the solution.](http://www.practice.geeksforgeeks.org/problem-page.php?pid=152)

The naive solution for this problem is to generate all subsequences of both given sequences and find the longest matching subsequence. This solution is exponential in term of time complexity. Let us see how this problem possesses both important properties of a Dynamic Programming (DP) Problem.

**1) Optimal Substructure:**  
Let the input sequences be X[0..m-1] and Y[0..n-1] of lengths m and n respectively. And let L(X[0..m-1], Y[0..n-1]) be the length of LCS of the two sequences X and Y. Following is the recursive definition of L(X[0..m-1], Y[0..n-1]).

If last characters of both sequences match (or X[m-1] == Y[n-1]) then  
L(X[0..m-1], Y[0..n-1]) = 1 + L(X[0..m-2], Y[0..n-2])

If last characters of both sequences do not match (or X[m-1] != Y[n-1]) then  
L(X[0..m-1], Y[0..n-1]) = MAX ( L(X[0..m-2], Y[0..n-1]), L(X[0..m-1], Y[0..n-2])

Examples:  
1) Consider the input strings “AGGTAB” and “GXTXAYB”. Last characters match for the strings. So length of LCS can be written as:  
L(“AGGTAB”, “GXTXAYB”) = 1 + L(“AGGTA”, “GXTXAY”)  
  
2) Consider the input strings “ABCDGH” and “AEDFHR. Last characters do not match for the strings. So length of LCS can be written as:  
L(“ABCDGH”, “AEDFHR”) = MAX ( L(“ABCDG”, “AEDFH**R**”), L(“ABCDG**H**”, “AEDFH”) )

So the LCS problem has optimal substructure property as the main problem can be solved using solutions to subproblems.

**2) Overlapping Subproblems:**  
Following is simple recursive implementation of the LCS problem. The implementation simply follows the recursive structure mentioned above.

* C/C++
* Python

|  |
| --- |
| /\* A Naive recursive implementation of LCS problem \*/  #include<bits/stdc++.h>    int max(int a, int b);    /\* Returns length of LCS for X[0..m-1], Y[0..n-1] \*/  int lcs( char \*X, char \*Y, int m, int n )  {     if (m == 0 || n == 0)       return 0;     if (X[m-1] == Y[n-1])       return 1 + lcs(X, Y, m-1, n-1);     else       return max(lcs(X, Y, m, n-1), lcs(X, Y, m-1, n));  }    /\* Utility function to get max of 2 integers \*/  int max(int a, int b)  {      return (a > b)? a : b;  }    /\* Driver program to test above function \*/  int main()  {    char X[] = "AGGTAB";    char Y[] = "GXTXAYB";      int m = strlen(X);    int n = strlen(Y);      printf("Length of LCS is %d\n", lcs( X, Y, m, n ) );      return 0;  } |

Run on IDE

Output:

Length of LCS is 4

Time complexity of the above naive recursive approach is O(2^n) in worst case and worst case happens when all characters of X and Y mismatch i.e., length of LCS is 0.  
Considering the above implementation, following is a partial recursion tree for input strings “AXYT” and “AYZX”

lcs("AXYT", "AYZX")

/ \

lcs("AXY", "AYZX") lcs("AXYT", "AYZ")

/ \ / \

lcs("AX", "AYZX") lcs("AXY", "AYZ") lcs("AXY", "AYZ") lcs("AXYT", "AY")

In the above partial recursion tree, lcs(“AXY”, “AYZ”) is being solved twice. If we draw the complete recursion tree, then we can see that there are many subproblems which are solved again and again. So this problem has Overlapping Substructure property and recomputation of same subproblems can be avoided by either using Memoization or Tabulation. Following is a tabulated implementation for the LCS problem.

* C/C++
* Python

|  |
| --- |
| /\* Dynamic Programming C/C++ implementation of LCS problem \*/  #include<bits/stdc++.h>    int max(int a, int b);    /\* Returns length of LCS for X[0..m-1], Y[0..n-1] \*/  int lcs( char \*X, char \*Y, int m, int n )  {     int L[m+1][n+1];     int i, j;       /\* Following steps build L[m+1][n+1] in bottom up fashion. Note        that L[i][j] contains length of LCS of X[0..i-1] and Y[0..j-1] \*/     for (i=0; i<=m; i++)     {       for (j=0; j<=n; j++)       {         if (i == 0 || j == 0)           L[i][j] = 0;           else if (X[i-1] == Y[j-1])           L[i][j] = L[i-1][j-1] + 1;           else           L[i][j] = max(L[i-1][j], L[i][j-1]);       }     }       /\* L[m][n] contains length of LCS for X[0..n-1] and Y[0..m-1] \*/     return L[m][n];  }    /\* Utility function to get max of 2 integers \*/  int max(int a, int b)  {      return (a > b)? a : b;  }    /\* Driver program to test above function \*/  int main()  {    char X[] = "AGGTAB";    char Y[] = "GXTXAYB";      int m = strlen(X);    int n = strlen(Y);      printf("Length of LCS is %d\n", lcs( X, Y, m, n ) );      return 0;  } |

Run on IDE

Time Complexity of the above implementation is O(mn) which is much better than the worst case time complexity of Naive Recursive implementation.

The above algorithm/code returns only length of LCS. Please see the following post for printing the LCS.  
[Printing Longest Common Subsequence](http://www.geeksforgeeks.org/printing-longest-common-subsequence/)

You can also check the space optimized version of LCS at  
[Space Optimized Solution of LCS](http://www.geeksforgeeks.org/space-optimized-solution-lcs/)

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

**References:**  
<http://www.youtube.com/watch?v=V5hZoJ6uK-s>  
<http://www.algorithmist.com/index.php/Longest_Common_Subsequence>  
<http://www.ics.uci.edu/~eppstein/161/960229.html>  
<http://en.wikipedia.org/wiki/Longest_common_subsequence_problem>

# Dynamic Programming | Set 5 (Edit Distance)

Given two strings str1 and str2 and below operations that can performed on str1. Find minimum number of edits (operations) required to convert ‘str1’ into ‘str2’.

1. Insert
2. Remove
3. Replace

All of the above operations are of equal cost.  
 **Examples:**

Input: str1 = "geek", str2 = "gesek"

Output: 1

We can convert str1 into str2 by inserting a 's'.

Input: str1 = "cat", str2 = "cut"

Output: 1

We can convert str1 into str2 by replacing 'a' with 'u'.

Input: str1 = "sunday", str2 = "saturday"

Output: 3

Last three and first characters are same. We basically

need to convert "un" to "atur". This can be done using

below three operations.

Replace 'n' with 'r', insert t, insert a

## [We strongly recommend that you click here and practice it, before moving on to the solution.](http://www.practice.geeksforgeeks.org/problem-page.php?pid=164)

**What are the subproblems in this case?**  
The idea is process all characters one by one staring from either from left or right sides of both strings.  
Let we traverse from right corner, there are two possibilities for every pair of character being traversed.

**m:** Length of str1 (first string)

**n:** Length of str2 (second string)

1. If last characters of two strings are same, nothing much to do. Ignore last characters and get count for remaining strings. So we recur for lengths m-1 and n-1.
2. Else (If last characters are not same), we consider all operations on ‘str1’, consider all three operations on last character of first string, recursively compute minimum cost for all three operations and take minimum of three values.
   1. Insert: Recur for m and n-1
   2. Remove: Recur for m-1 and n
   3. Replace: Recur for m-1 and n-1

Below is C++ implementation of above Naive recursive solution.

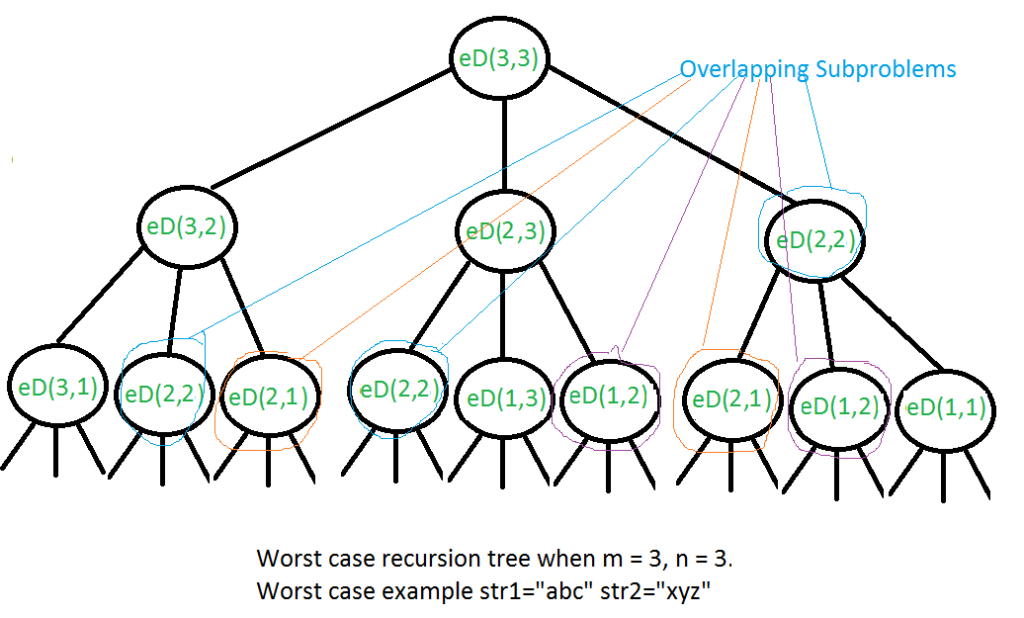
* C++
* Java
* Python

|  |
| --- |
| // A Naive recursive C++ program to find minimum number  // operations to convert str1 to str2  #include<bits/stdc++.h>  using namespace std;    // Utility function to find minimum of three numbers  int min(int x, int y, int z)  {     return min(min(x, y), z);  }    int editDist(string str1 , string str2 , int m ,int n)  {      // If first string is empty, the only option is to      // insert all characters of second string into first      if (m == 0) return n;        // If second string is empty, the only option is to      // remove all characters of first string      if (n == 0) return m;        // If last characters of two strings are same, nothing      // much to do. Ignore last characters and get count for      // remaining strings.      if (str1[m-1] == str2[n-1])          return editDist(str1, str2, m-1, n-1);        // If last characters are not same, consider all three      // operations on last character of first string, recursively      // compute minimum cost for all three operations and take      // minimum of three values.      return 1 + min ( editDist(str1,  str2, m, n-1),    // Insert                       editDist(str1,  str2, m-1, n),   // Remove                       editDist(str1,  str2, m-1, n-1) // Replace                     );  }    // Driver program  int main()  {      // your code goes here      string str1 = "sunday";      string str2 = "saturday";        cout << editDist( str1 , str2 , str1.length(), str2.length());        return 0;  } |

Run on IDE

Output:

3

The time complexity of above solution is exponential. In worst case, we may end up doing O(3m) operations. The worst case happens when none of characters of two strings match. Below is a recursive call diagram for worst case.  
[](http://d1hyf4ir1gqw6c.cloudfront.net/wp-content/uploads/EditDistance.png)

We can see that many subproblems are solved again and again, for example eD(2,2) is called three times. Since same suproblems are called again, this problem has Overlapping Subprolems property. So Edit Distance problem has both properties (see [this](http://www.geeksforgeeks.org/archives/12635)and [this](http://www.geeksforgeeks.org/archives/12819)) of a dynamic programming problem. Like other typical Dynamic Programming(DP) problems, recomputations of same subproblems can be avoided by constructing a temporary array that stores results of subpriblems.

* C++
* Java
* Python

|  |
| --- |
| // A Dynamic Programming based C++ program to find minimum  // number operations to convert str1 to str2  #include<bits/stdc++.h>  using namespace std;    // Utility function to find minimum of three numbers  int min(int x, int y, int z)  {      return min(min(x, y), z);  }    int editDistDP(string str1, string str2, int m, int n)  {      // Create a table to store results of subproblems      int dp[m+1][n+1];        // Fill d[][] in bottom up manner      for (int i=0; i<=m; i++)      {          for (int j=0; j<=n; j++)          {              // If first string is empty, only option is to              // isnert all characters of second string              if (i==0)                  dp[i][j] = j;  // Min. operations = j                // If second string is empty, only option is to              // remove all characters of second string              else if (j==0)                  dp[i][j] = i; // Min. operations = i                // If last characters are same, ignore last char              // and recur for remaining string              else if (str1[i-1] == str2[j-1])                  dp[i][j] = dp[i-1][j-1];                // If last character are different, consider all              // possibilities and find minimum              else                  dp[i][j] = 1 + min(dp[i][j-1],  // Insert                                     dp[i-1][j],  // Remove                                     dp[i-1][j-1]); // Replace          }      }        return dp[m][n];  }    // Driver program  int main()  {      // your code goes here      string str1 = "sunday";      string str2 = "saturday";        cout << editDistDP(str1, str2, str1.length(), str2.length());        return 0;  } |

Run on IDE

Output:

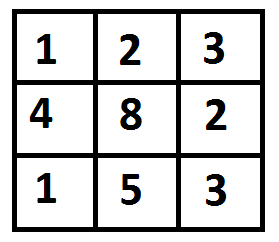
3

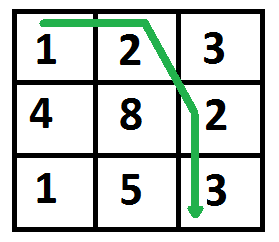
Time Complexity: O(m x n)  
Auxiliary Space: O(m x n)

**Applications**: There are many practical applications of edit distance algorithm, refer [Lucene](http://en.wikipedia.org/wiki/Lucene) API for sample. Another example, display all the words in a dictionary that are near proximity to a given word\incorrectly spelled word.

# Dynamic Programming | Set 6 (Min Cost Path)

Given a cost matrix cost[][] and a position (m, n) in cost[][], write a function that returns cost of minimum cost path to reach (m, n) from (0, 0). Each cell of the matrix represents a cost to traverse through that cell. Total cost of a path to reach (m, n) is sum of all the costs on that path (including both source and destination). You can only traverse down, right and diagonally lower cells from a given cell, i.e., from a given cell (i, j), cells (i+1, j), (i, j+1) and (i+1, j+1) can be traversed. You may assume that all costs are positive integers.

For example, in the following figure, what is the minimum cost path to (2, 2)?  
[](http://d1hyf4ir1gqw6c.cloudfront.net/wp-content/uploads/dp.png)

The path with minimum cost is highlighted in the following figure. The path is (0, 0) –> (0, 1) –> (1, 2) –> (2, 2). The cost of the path is 8 (1 + 2 + 2 + 3).  
[](http://d1hyf4ir1gqw6c.cloudfront.net/wp-content/uploads/dp2.png)

**1) Optimal Substructure**  
The path to reach (m, n) must be through one of the 3 cells: (m-1, n-1) or (m-1, n) or (m, n-1). So minimum cost to reach (m, n) can be written as “minimum of the 3 cells plus cost[m][n]”.

minCost(m, n) = min (minCost(m-1, n-1), minCost(m-1, n), minCost(m, n-1)) + cost[m][n]

**2) Overlapping Subproblems**  
Following is simple recursive implementation of the MCP (Minimum Cost Path) problem. The implementation simply follows the recursive structure mentioned above.

|  |
| --- |
| /\* A Naive recursive implementation of MCP(Minimum Cost Path) problem \*/  #include<stdio.h>  #include<limits.h>  #define R 3  #define C 3    int min(int x, int y, int z);    /\* Returns cost of minimum cost path from (0,0) to (m, n) in mat[R][C]\*/  int minCost(int cost[R][C], int m, int n)  {     if (n < 0 || m < 0)        return INT\_MAX;     else if (m == 0 && n == 0)        return cost[m][n];     else        return cost[m][n] + min( minCost(cost, m-1, n-1),                                 minCost(cost, m-1, n),                                 minCost(cost, m, n-1) );  }    /\* A utility function that returns minimum of 3 integers \*/  int min(int x, int y, int z)  {     if (x < y)        return (x < z)? x : z;     else        return (y < z)? y : z;  }    /\* Driver program to test above functions \*/  int main()  {     int cost[R][C] = { {1, 2, 3},                        {4, 8, 2},                        {1, 5, 3} };     printf(" %d ", minCost(cost, 2, 2));     return 0;  } |

Run on IDE

It should be noted that the above function computes the same subproblems again and again. See the following recursion tree, there are many nodes which apear more than once. Time complexity of this naive recursive solution is exponential and it is terribly slow.

mC refers to minCost()

mC(2, 2)

/ | \

/ | \

mC(1, 1) mC(1, 2) mC(2, 1)

/ | \ / | \ / | \

/ | \ / | \ / | \

mC(0,0) mC(0,1) mC(1,0) mC(0,1) mC(0,2) mC(1,1) mC(1,0) mC(1,1) mC(2,0)

So the MCP problem has both properties (see [this](http://www.geeksforgeeks.org/archives/12635)and [this](http://www.geeksforgeeks.org/archives/12819)) of a dynamic programming problem. Like other typical [Dynamic Programming(DP) problems](http://www.geeksforgeeks.org/archives/tag/dynamic-programming), recomputations of same subproblems can be avoided by constructing a temporary array tc[][] in bottom up manner.

* C++
* Java
* Python

|  |
| --- |
| /\* Dynamic Programming implementation of MCP problem \*/  #include<stdio.h>  #include<limits.h>  #define R 3  #define C 3    int min(int x, int y, int z);    int minCost(int cost[R][C], int m, int n)  {       int i, j;         // Instead of following line, we can use int tc[m+1][n+1] or       // dynamically allocate memoery to save space. The following line is       // used to keep te program simple and make it working on all compilers.       int tc[R][C];         tc[0][0] = cost[0][0];         /\* Initialize first column of total cost(tc) array \*/       for (i = 1; i <= m; i++)          tc[i][0] = tc[i-1][0] + cost[i][0];         /\* Initialize first row of tc array \*/       for (j = 1; j <= n; j++)          tc[0][j] = tc[0][j-1] + cost[0][j];         /\* Construct rest of the tc array \*/       for (i = 1; i <= m; i++)          for (j = 1; j <= n; j++)              tc[i][j] = min(tc[i-1][j-1],                             tc[i-1][j],                             tc[i][j-1]) + cost[i][j];         return tc[m][n];  }    /\* A utility function that returns minimum of 3 integers \*/  int min(int x, int y, int z)  {     if (x < y)        return (x < z)? x : z;     else        return (y < z)? y : z;  }    /\* Driver program to test above functions \*/  int main()  {     int cost[R][C] = { {1, 2, 3},                        {4, 8, 2},                        {1, 5, 3} };     printf(" %d ", minCost(cost, 2, 2));     return 0;  } |

Run on IDE

Output:

8

Time Complexity of the DP implementation is O(mn) which is much better than Naive Recursive implementation.

# Dynamic Programming | Set 7 (Coin Change)

Given a value N, if we want to make change for N cents, and we have infinite supply of each of S = { S1, S2, .. , Sm} valued coins, how many ways can we make the change? The order of coins doesn’t matter.

For example, for N = 4 and S = {1,2,3}, there are four solutions: {1,1,1,1},{1,1,2},{2,2},{1,3}. So output should be 4. For N = 10 and S = {2, 5, 3, 6}, there are five solutions: {2,2,2,2,2}, {2,2,3,3}, {2,2,6}, {2,3,5} and {5,5}. So the output should be 5.

## [We strongly recommend that you click here and practice it, before moving on to the solution.](http://www.practice.geeksforgeeks.org/problem-page.php?pid=225)

**1) Optimal Substructure**  
To count total number solutions, we can divide all set solutions in two sets.  
1) Solutions that do not contain mth coin (or Sm).  
2) Solutions that contain at least one Sm.  
Let count(S[], m, n) be the function to count the number of solutions, then it can be written as sum of count(S[], m-1, n) and count(S[], m, n-Sm).

Therefore, the problem has optimal substructure property as the problem can be solved using solutions to subproblems.

**2) Overlapping Subproblems**  
Following is a simple recursive implementation of the Coin Change problem. The implementation simply follows the recursive structure mentioned above.

|  |
| --- |
| #include<stdio.h>    // Returns the count of ways we can sum  S[0...m-1] coins to get sum n  int count( int S[], int m, int n )  {      // If n is 0 then there is 1 solution (do not include any coin)      if (n == 0)          return 1;        // If n is less than 0 then no solution exists      if (n < 0)          return 0;        // If there are no coins and n is greater than 0, then no solution exist      if (m <=0 && n >= 1)          return 0;        // count is sum of solutions (i) including S[m-1] (ii) excluding S[m-1]      return count( S, m - 1, n ) + count( S, m, n-S[m-1] );  }    // Driver program to test above function  int main()  {      int i, j;      int arr[] = {1, 2, 3};      int m = sizeof(arr)/sizeof(arr[0]);      printf("%d ", count(arr, m, 4));      getchar();      return 0;  } |

Run on IDE

It should be noted that the above function computes the same subproblems again and again. See the following recursion tree for S = {1, 2, 3} and n = 5.  
The function C({1}, 3) is called two times. If we draw the complete tree, then we can see that there are many subproblems being called more than once.

C() --> count()

C({1,2,3}, 5)

/ \

/ \

C({1,2,3}, 2) C({1,2}, 5)

/ \ / \

/ \ / \

C({1,2,3}, -1) C({1,2}, 2) C({1,2}, 3) C({1}, 5)

/ \ / \ / \

/ \ / \ / \

C({1,2},0) C({1},2) C({1,2},1) C({1},3) C({1}, 4) C({}, 5)

/ \ / \ / \ / \

/ \ / \ / \ / \

. . . . . . C({1}, 3) C({}, 4)

/ \

/ \

. .

Since same suproblems are called again, this problem has Overlapping Subprolems property. So the Coin Change problem has both properties (see [this](http://www.geeksforgeeks.org/archives/12635)and [this](http://www.geeksforgeeks.org/archives/12819)) of a dynamic programming problem. Like other typical [Dynamic Programming(DP) problems](http://www.geeksforgeeks.org/archives/tag/dynamic-programming), recomputations of same subproblems can be avoided by constructing a temporary array table[][] in bottom up manner.

**Dynamic Programming Solution**

* C
* Java
* Python

|  |
| --- |
| #include<stdio.h>    int count( int S[], int m, int n )  {      int i, j, x, y;        // We need n+1 rows as the table is consturcted in bottom up manner using      // the base case 0 value case (n = 0)      int table[n+1][m];        // Fill the enteries for 0 value case (n = 0)      for (i=0; i<m; i++)          table[0][i] = 1;        // Fill rest of the table enteries in bottom up manner      for (i = 1; i < n+1; i++)      {          for (j = 0; j < m; j++)          {              // Count of solutions including S[j]              x = (i-S[j] >= 0)? table[i - S[j]][j]: 0;                // Count of solutions excluding S[j]              y = (j >= 1)? table[i][j-1]: 0;                // total count              table[i][j] = x + y;          }      }      return table[n][m-1];  }    // Driver program to test above function  int main()  {      int arr[] = {1, 2, 3};      int m = sizeof(arr)/sizeof(arr[0]);      int n = 4;      printf(" %d ", count(arr, m, n));      return 0;  } |

Run on IDE

Output:

4

Time Complexity: O(mn)

Following is a simplified version of method 2. The auxiliary space required here is O(n) only.

* C
* Python

|  |
| --- |
| int count( int S[], int m, int n )  {      // table[i] will be storing the number of solutions for      // value i. We need n+1 rows as the table is consturcted      // in bottom up manner using the base case (n = 0)      int table[n+1];        // Initialize all table values as 0      memset(table, 0, sizeof(table));        // Base case (If given value is 0)      table[0] = 1;        // Pick all coins one by one and update the table[] values      // after the index greater than or equal to the value of the      // picked coin      for(int i=0; i<m; i++)          for(int j=S[i]; j<=n; j++)              table[j] += table[j-S[i]];        return table[n];  } |

Run on IDE

Thanks to Rohan Laishram for suggesting this space optimized version.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

References:  
<http://www.algorithmist.com/index.php/Coin_Change>

# Dynamic Programming | Set 8 (Matrix Chain Multiplication)

Given a sequence of matrices, find the most efficient way to multiply these matrices together. The problem is not actually to perform the multiplications, but merely to decide in which order to perform the multiplications.

We have many options to multiply a chain of matrices because matrix multiplication is associative. In other words, no matter how we parenthesize the product, the result will be the same. For example, if we had four matrices A, B, C, and D, we would have:

(ABC)D = (AB)(CD) = A(BCD) = ....

However, the order in which we parenthesize the product affects the number of simple arithmetic operations needed to compute the product, or the efficiency. For example, suppose A is a 10 × 30 matrix, B is a 30 × 5 matrix, and C is a 5 × 60 matrix. Then,

(AB)C = (10×30×5) + (10×5×60) = 1500 + 3000 = 4500 operations

A(BC) = (30×5×60) + (10×30×60) = 9000 + 18000 = 27000 operations.

Clearly the first parenthesization requires less number of operations.

Given an array p[] which represents the chain of matrices such that the ith matrix Ai is of dimension p[i-1] x p[i]. We need to write a function MatrixChainOrder() that should return the minimum number of multiplications needed to multiply the chain.

**Input: p[] = {40, 20, 30, 10, 30}**

**Output: 26000**

There are 4 matrices of dimensions 40x20, 20x30, 30x10 and 10x30.

Let the input 4 matrices be A, B, C and D. The minimum number of

multiplications are obtained by putting parenthesis in following way

(A(BC))D --> 20\*30\*10 + 40\*20\*10 + 40\*10\*30

**Input: p[] = {10, 20, 30, 40, 30}**

**Output: 30000**

There are 4 matrices of dimensions 10x20, 20x30, 30x40 and 40x30.

Let the input 4 matrices be A, B, C and D. The minimum number of

multiplications are obtained by putting parenthesis in following way

((AB)C)D --> 10\*20\*30 + 10\*30\*40 + 10\*40\*30

**Input: p[] = {10, 20, 30}**

**Output: 6000**

There are only two matrices of dimensions 10x20 and 20x30. So there

is only one way to multiply the matrices, cost of which is 10\*20\*30

**1) Optimal Substructure:**  
A simple solution is to place parenthesis at all possible places, calculate the cost for each placement and return the minimum value. In a chain of matrices of size n, we can place the first set of parenthesis in n-1 ways. For example, if the given chain is of 4 matrices. let the chain be ABCD, then there are 3 ways to place first set of parenthesis outer side: (A)(BCD), (AB)(CD) and (ABC)(D). So when we place a set of parenthesis, we divide the problem into subproblems of smaller size. Therefore, the problem has optimal substructure property and can be easily solved using recursion.

Minimum number of multiplication needed to multiply a chain of size n = Minimum of all n-1 placements (these placements create subproblems of smaller size)

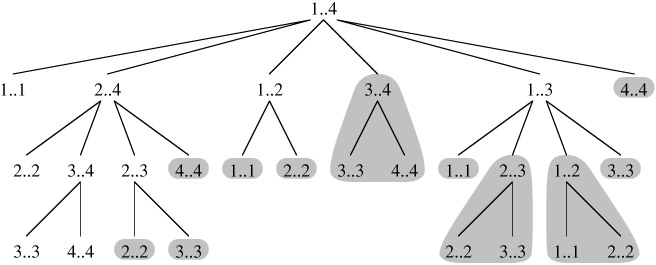
**2) Overlapping Subproblems**  
Following is a recursive implementation that simply follows the above optimal substructure property.

* C
* Java

|  |
| --- |
| /\* A naive recursive implementation that simply    follows the above optimal substructure property \*/  #include<stdio.h>  #include<limits.h>    // Matrix Ai has dimension p[i-1] x p[i] for i = 1..n  int MatrixChainOrder(int p[], int i, int j)  {      if(i == j)          return 0;      int k;      int min = INT\_MAX;      int count;        // place parenthesis at different places between first      // and last matrix, recursively calculate count of      // multiplications for each parenthesis placement and      // return the minimum count      for (k = i; k <j; k++)      {          count = MatrixChainOrder(p, i, k) +                  MatrixChainOrder(p, k+1, j) +                  p[i-1]\*p[k]\*p[j];            if (count < min)              min = count;      }        // Return minimum count      return min;  }    // Driver program to test above function  int main()  {      int arr[] = {1, 2, 3, 4, 3};      int n = sizeof(arr)/sizeof(arr[0]);        printf("Minimum number of multiplications is %d ",                            MatrixChainOrder(arr, 1, n-1));        getchar();      return 0;  } |

Run on IDE

Time complexity of the above naive recursive approach is exponential. It should be noted that the above function computes the same subproblems again and again. See the following recursion tree for a matrix chain of size 4. The function MatrixChainOrder(p, 3, 4) is called two times. We can see that there are many subproblems being called more than once.

[](http://d1hyf4ir1gqw6c.cloudfront.net/wp-content/uploads/MatrixChain1.jpg)

Since same suproblems are called again, this problem has Overlapping Subprolems property. So Matrix Chain Multiplication problem has both properties (see [this](http://www.geeksforgeeks.org/archives/12635)and [this](http://www.geeksforgeeks.org/archives/12819)) of a dynamic programming problem. Like other typical [Dynamic Programming(DP) problems](http://www.geeksforgeeks.org/archives/tag/dynamic-programming), recomputations of same subproblems can be avoided by constructing a temporary array m[][] in bottom up manner.

**Dynamic Programming Solution**  
Following is C/C++ implementation for Matrix Chain Multiplication problem using Dynamic Programming.

* C
* Java
* Python

|  |
| --- |
| // See the Cormen book for details of the following algorithm  #include<stdio.h>  #include<limits.h>    // Matrix Ai has dimension p[i-1] x p[i] for i = 1..n  int MatrixChainOrder(int p[], int n)  {        /\* For simplicity of the program, one extra row and one         extra column are allocated in m[][].  0th row and 0th         column of m[][] are not used \*/      int m[n][n];        int i, j, k, L, q;        /\* m[i,j] = Minimum number of scalar multiplications needed         to compute the matrix A[i]A[i+1]...A[j] = A[i..j] where         dimension of A[i] is p[i-1] x p[i] \*/        // cost is zero when multiplying one matrix.      for (i=1; i<n; i++)          m[i][i] = 0;        // L is chain length.      for (L=2; L<n; L++)      {          for (i=1; i<n-L+1; i++)          {              j = i+L-1;              m[i][j] = INT\_MAX;              for (k=i; k<=j-1; k++)              {                  // q = cost/scalar multiplications                  q = m[i][k] + m[k+1][j] + p[i-1]\*p[k]\*p[j];                  if (q < m[i][j])                      m[i][j] = q;              }          }      }        return m[1][n-1];  }    int main()  {      int arr[] = {1, 2, 3, 4};      int size = sizeof(arr)/sizeof(arr[0]);        printf("Minimum number of multiplications is %d ",                         MatrixChainOrder(arr, size));        getchar();      return 0;  } |

Run on IDE

Output:

Minimum number of multiplications is 18

Time Complexity: O(n^3)  
Auxiliary Space: O(n^2)

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

**Applications:**  
[Minimum and Maximum values of an expression with \* and +](http://www.geeksforgeeks.org/minimum-maximum-values-expression/)

**References:**  
<http://en.wikipedia.org/wiki/Matrix_chain_multiplication>  
<http://www.personal.kent.edu/~rmuhamma/Algorithms/MyAlgorithms/Dynamic/chainMatrixMult.htm>

# Dynamic Programming | Set 9 (Binomial Coefficient)

Following are common definition of [Binomial Coefficients](http://en.wikipedia.org/wiki/Binomial_coefficient).  
1) A [binomial coefficient](http://en.wikipedia.org/wiki/Binomial_coefficient) C(n, k) can be defined as the coefficient of X^k in the expansion of (1 + X)^n.

2) A binomial coefficient C(n, k) also gives the number of ways, disregarding order, that k objects can be chosen from among n objects; more formally, the number of k-element subsets (or k-combinations) of an n-element set.

**The Problem**  
Write a function that takes two parameters n and k and returns the value of Binomial Coefficient C(n, k). For example, your function should return 6 for n = 4 and k = 2, and it should return 10 for n = 5 and k = 2.

**1) Optimal Substructure**  
The value of C(n, k) can be recursively calculated using following standard formula for Binomial Coefficients.

C(n, k) = C(n-1, k-1) + C(n-1, k)

C(n, 0) = C(n, n) = 1

Following is a simple recursive implementation that simply follows the recursive structure mentioned above.

* C/C++
* Python

|  |
| --- |
| // A Naive Recursive Implementation  #include<stdio.h>    // Returns value of Binomial Coefficient C(n, k)  int binomialCoeff(int n, int k)  {    // Base Cases    if (k==0 || k==n)      return 1;      // Recur    return  binomialCoeff(n-1, k-1) + binomialCoeff(n-1, k);  }    /\* Driver program to test above function\*/  int main()  {      int n = 5, k = 2;      printf("Value of C(%d, %d) is %d ", n, k, binomialCoeff(n, k));      return 0;  } |

Run on IDE

**2) Overlapping Subproblems**  
It should be noted that the above function computes the same subproblems again and again. See the following recursion tree for n = 5 an k = 2. The function C(3, 1) is called two times. For large values of n, there will be many common subproblems.

C(5, 2)

/ \

C(4, 1) C(4, 2)

/ \ / \

C(3, 0) C(3, 1) C(3, 1) C(3, 2)

/ \ / \ / \

C(2, 0) C(2, 1) C(2, 0) C(2, 1) C(2, 1) C(2, 2)

/ \ / \ / \

C(1, 0) C(1, 1) C(1, 0) C(1, 1) C(1, 0) C(1, 1)

Since same suproblems are called again, this problem has Overlapping Subproblems property. So the Binomial Coefficient problem has both properties (see [this](http://www.geeksforgeeks.org/archives/12635)and [this](http://www.geeksforgeeks.org/archives/12819)) of a dynamic programming problem. Like other typical [Dynamic Programming(DP) problems](http://www.geeksforgeeks.org/archives/tag/dynamic-programming), re-computations of same subproblems can be avoided by constructing a temporary array C[][] in bottom up manner. Following is Dynamic Programming based implementation.

* C
* Java
* Python

|  |
| --- |
| // A Dynamic Programming based solution that uses table C[][] to  // calculate the Binomial Coefficient  #include<stdio.h>    // Prototype of a utility function that returns minimum of two integers  int min(int a, int b);    // Returns value of Binomial Coefficient C(n, k)  int binomialCoeff(int n, int k)  {      int C[n+1][k+1];      int i, j;        // Caculate value of Binomial Coefficient in bottom up manner      for (i = 0; i <= n; i++)      {          for (j = 0; j <= min(i, k); j++)          {              // Base Cases              if (j == 0 || j == i)                  C[i][j] = 1;                // Calculate value using previosly stored values              else                  C[i][j] = C[i-1][j-1] + C[i-1][j];          }      }        return C[n][k];  }    // A utility function to return minimum of two integers  int min(int a, int b)  {      return (a<b)? a: b;  }    /\* Drier program to test above function\*/  int main()  {      int n = 5, k = 2;      printf ("Value of C(%d, %d) is %d ", n, k, binomialCoeff(n, k) );      return 0;  } |

Run on IDE

Output:

Value of C[5][2] is 10

Time Complexity: O(n\*k)  
Auxiliary Space: O(n\*k)

Following is a space optimized version of the above code. The following code only uses O(k). Thanks to [AK](http://www.geeksforgeeks.org/archives/17806/comment-page-1#comment-7460)for suggesting this method.

* C/C++
* Python

|  |
| --- |
| // C++ program for space optimized Dynamic Programming  // Solution of Binomial Coefficient  #include<bits/stdc++.h>  using namespace std;    int binomialCoeff(int n, int k)  {      int C[k+1];      memset(C, 0, sizeof(C));        C[0] = 1;  // nC0 is 1        for (int i = 1; i <= n; i++)      {          // Compute next row of pascal triangle using          // the previous row          for (int j = min(i, k); j > 0; j--)              C[j] = C[j] + C[j-1];      }      return C[k];  }    /\* Drier program to test above function\*/  int main()  {      int n = 5, k = 2;      printf ("Value of C(%d, %d) is %d ",              n, k, binomialCoeff(n, k) );      return 0;  } |

Run on IDE

Output:

Value of C[5][2] is 10

Time Complexity: O(n\*k)  
Auxiliary Space: O(k)

Explanation:  
1==========>> n = 0, C(0,0) = 1  
1–1========>> n = 1, C(1,0) = 1, C(1,1) = 1  
1–2–1======>> n = 2, C(2,0) = 1, C(2,1) = 2, C(2,2) = 1  
1–3–3–1====>> n = 3, C(3,0) = 1, C(3,1) = 3, C(3,2) = 3, C(3,3)=1  
1–4–6–4–1==>> n = 4, C(4,0) = 1, C(4,1) = 4, C(4,2) = 6, C(4,3)=4, C(4,4)=1  
So here every loop on i, builds i’th row of pascal triangle, using (i-1)th row

At any time, every element of array C will have some value (ZERO or more) and in next iteration, value for those elements comes from previous iteration.  
In statement,  
C[j] = C[j] + C[j-1]  
Right hand side represents the value coming from previous iteration (A row of Pascal’s triangle depends on previous row). Left Hand side represents the value of current iteration which will be obtained by this statement.

Let's say we want to calculate C(4, 3),

i.e. n=4, k=3:

All elements of array C of size 4 (k+1) are

initialized to ZERO.

i.e. C[0] = C[1] = C[2] = C[3] = C[4] = 0;

Then C[0] is set to 1

For i = 1:

C[1] = C[1] + C[0] = 0 + 1 = 1 ==>> C(1,1) = 1

For i = 2:

C[2] = C[2] + C[1] = 0 + 1 = 1 ==>> C(2,2) = 1

C[1] = C[1] + C[0] = 1 + 1 = 2 ==>> C(2,2) = 2

For i=3:

C[3] = C[3] + C[2] = 0 + 1 = 1 ==>> C(3,3) = 1

C[2] = C[2] + C[1] = 1 + 2 = 3 ==>> C(3,2) = 3

C[1] = C[1] + C[0] = 2 + 1 = 3 ==>> C(3,1) = 3

For i=4:

C[4] = C[4] + C[3] = 0 + 1 = 1 ==>> C(4,4) = 1

C[3] = C[3] + C[2] = 1 + 3 = 4 ==>> C(4,3) = 4

C[2] = C[2] + C[1] = 3 + 3 = 6 ==>> C(4,2) = 6

C[1] = C[1] + C[0] = 3 + 1 = 4 ==>> C(4,1) = 4

C(4,3) = 4 is would be the answer in our example.

See this for [Space and time efficient Binomial Coefficient](http://www.geeksforgeeks.org/space-and-time-efficient-binomial-coefficient/)

# Dynamic Programming | Set 10 ( 0-1 Knapsack Problem)

Given weights and values of n items, put these items in a knapsack of capacity W to get the maximum total value in the knapsack. In other words, given two integer arrays val[0..n-1] and wt[0..n-1] which represent values and weights associated with n items respectively. Also given an integer W which represents knapsack capacity, find out the maximum value subset of val[] such that sum of the weights of this subset is smaller than or equal to W. You cannot break an item, either pick the complete item, or don’t pick it (0-1 property).

## [We strongly recommend that you click here and practice it, before moving on to the solution.](http://www.practice.geeksforgeeks.org/problem-page.php?pid=909)

A simple solution is to consider all subsets of items and calculate the total weight and value of all subsets. Consider the only subsets whose total weight is smaller than W. From all such subsets, pick the maximum value subset.

**1) Optimal Substructure:**  
To consider all subsets of items, there can be two cases for every item: (1) the item is included in the optimal subset, (2) not included in the optimal set.  
Therefore, the maximum value that can be obtained from n items is max of following two values.  
1) Maximum value obtained by n-1 items and W weight (excluding nth item).  
2) Value of nth item plus maximum value obtained by n-1 items and W minus weight of the nth item (including nth item).

If weight of nth item is greater than W, then the nth item cannot be included and case 1 is the only possibility.

**2) Overlapping Subproblems**  
Following is recursive implementation that simply follows the recursive structure mentioned above.

* C/C++
* Java
* Python

|  |
| --- |
| /\* A Naive recursive implementation of 0-1 Knapsack problem \*/  #include<stdio.h>    // A utility function that returns maximum of two integers  int max(int a, int b) { return (a > b)? a : b; }    // Returns the maximum value that can be put in a knapsack of capacity W  int knapSack(int W, int wt[], int val[], int n)  {     // Base Case     if (n == 0 || W == 0)         return 0;       // If weight of the nth item is more than Knapsack capacity W, then     // this item cannot be included in the optimal solution     if (wt[n-1] > W)         return knapSack(W, wt, val, n-1);       // Return the maximum of two cases:     // (1) nth item included     // (2) not included     else return max( val[n-1] + knapSack(W-wt[n-1], wt, val, n-1),                      knapSack(W, wt, val, n-1)                    );  }    // Driver program to test above function  int main()  {      int val[] = {60, 100, 120};      int wt[] = {10, 20, 30};      int  W = 50;      int n = sizeof(val)/sizeof(val[0]);      printf("%d", knapSack(W, wt, val, n));      return 0;  } |

Run on IDE

Output:

220

It should be noted that the above function computes the same subproblems again and again. See the following recursion tree, K(1, 1) is being evaluated twice. Time complexity of this naive recursive solution is exponential (2^n).

In the following recursion tree, K() refers to knapSack(). The two

parameters indicated in the following recursion tree are n and W.

The recursion tree is for following sample inputs.

wt[] = {1, 1, 1}, W = 2, val[] = {10, 20, 30}

K(3, 2) ---------> K(n, W)

/ \

/ \

K(2,2) K(2,1)

/ \ / \

/ \ / \

K(1,2) K(1,1) K(1,1) K(1,0)

/ \ / \ / \

/ \ / \ / \

K(0,2) K(0,1) K(0,1) K(0,0) K(0,1) K(0,0)

Recursion tree for Knapsack capacity 2 units and 3 items of 1 unit weight.

Since suproblems are evaluated again, this problem has Overlapping Subprolems property. So the 0-1 Knapsack problem has both properties (see [this](http://www.geeksforgeeks.org/archives/12635)and [this](http://www.geeksforgeeks.org/archives/12819)) of a dynamic programming problem. Like other typical [Dynamic Programming(DP) problems](http://www.geeksforgeeks.org/archives/tag/dynamic-programming), recomputations of same subproblems can be avoided by constructing a temporary array K[][] in bottom up manner. Following is Dynamic Programming based implementation.

* C++
* Java
* Python

|  |
| --- |
| // A Dynamic Programming based solution for 0-1 Knapsack problem  #include<stdio.h>    // A utility function that returns maximum of two integers  int max(int a, int b) { return (a > b)? a : b; }    // Returns the maximum value that can be put in a knapsack of capacity W  int knapSack(int W, int wt[], int val[], int n)  {     int i, w;     int K[n+1][W+1];       // Build table K[][] in bottom up manner     for (i = 0; i <= n; i++)     {         for (w = 0; w <= W; w++)         {             if (i==0 || w==0)                 K[i][w] = 0;             else if (wt[i-1] <= w)                   K[i][w] = max(val[i-1] + K[i-1][w-wt[i-1]],  K[i-1][w]);             else                   K[i][w] = K[i-1][w];         }     }       return K[n][W];  }    int main()  {      int val[] = {60, 100, 120};      int wt[] = {10, 20, 30};      int  W = 50;      int n = sizeof(val)/sizeof(val[0]);      printf("%d", knapSack(W, wt, val, n));      return 0;  } |

Run on IDE

Output:

220

Time Complexity: O(nW) where n is the number of items and W is the capacity of knapsack.

### Asked in: [Microsoft](http://www.practice.geeksforgeeks.org/tag-page.php?isCmp=1&tag=%20Microsoft), [Amazon](http://www.practice.geeksforgeeks.org/tag-page.php?isCmp=1&tag=Amazon), [Flipkart](http://www.practice.geeksforgeeks.org/tag-page.php?isCmp=1&tag=Flipkart), [GreyOrange](http://www.practice.geeksforgeeks.org/tag-page.php?isCmp=1&tag=GreyOrange), [Mobicip](http://www.practice.geeksforgeeks.org/tag-page.php?isCmp=1&tag=Mobicip), [Morgan Stanely](http://www.practice.geeksforgeeks.org/tag-page.php?isCmp=1&tag=Morgan%20Stanely), [Oracle](http://www.practice.geeksforgeeks.org/tag-page.php?isCmp=1&tag=Oracle), [Payu](http://www.practice.geeksforgeeks.org/tag-page.php?isCmp=1&tag=Payu), [Snapdeal](http://www.practice.geeksforgeeks.org/tag-page.php?isCmp=1&tag=Snapdeal), [Visa](http://www.practice.geeksforgeeks.org/tag-page.php?isCmp=1&tag=Visa)

References:  
<http://www.es.ele.tue.nl/education/5MC10/Solutions/knapsack.pdf>  
<http://www.cse.unl.edu/~goddard/Courses/CSCE310J/Lectures/Lecture8-DynamicProgramming.pdf>

# Dynamic Programming | Set 11 (Egg Dropping Puzzle)

The following is a description of the instance of this famous puzzle involving n=2 eggs and a building with k=36 floors.

Suppose that we wish to know which stories in a 36-story building are safe to drop eggs from, and which will cause the eggs to break on landing. We make a few assumptions:

…..An egg that survives a fall can be used again.  
…..A broken egg must be discarded.  
…..The effect of a fall is the same for all eggs.  
…..If an egg breaks when dropped, then it would break if dropped from a higher floor.  
…..If an egg survives a fall then it would survive a shorter fall.  
…..It is not ruled out that the first-floor windows break eggs, nor is it ruled out that the 36th-floor do not cause an egg to break.

If only one egg is available and we wish to be sure of obtaining the right result, the experiment can be carried out in only one way. Drop the egg from the first-floor window; if it survives, drop it from the second floor window. Continue upward until it breaks. In the worst case, this method may require 36 droppings. Suppose 2 eggs are available. What is the least number of egg-droppings that is guaranteed to work in all cases?  
The problem is not actually to find the critical floor, but merely to decide floors from which eggs should be dropped so that total number of trials are minimized.

Source: [Wiki for Dynamic Programming](http://en.wikipedia.org/wiki/Dynamic_programming#Egg_dropping_puzzle)

## [We strongly recommend that you click here and practice it, before moving on to the solution.](http://www.practice.geeksforgeeks.org/problem-page.php?pid=162)

In this post, we will discuss solution to a general problem with n eggs and k floors. The solution is to try dropping an egg from every floor (from 1 to k) and recursively calculate the minimum number of droppings needed in worst case. The floor which gives the minimum value in worst case is going to be part of the solution.  
In the following solutions, we return the minimum number of trials in worst case; these solutions can be easily modified to print floor numbers of every trials also.

**1) Optimal Substructure:**  
When we drop an egg from a floor x, there can be two cases (1) The egg breaks (2) The egg doesn’t break.

1) If the egg breaks after dropping from xth floor, then we only need to check for floors lower than x with remaining eggs; so the problem reduces to x-1 floors and n-1 eggs  
2) If the egg doesn’t break after dropping from the xth floor, then we only need to check for floors higher than x; so the problem reduces to k-x floors and n eggs.

Since we need to minimize the number of trials in worstcase, we take the maximum of two cases. We consider the max of above two cases for every floor and choose the floor which yields minimum number of trials.

k ==> Number of floors

n ==> Number of Eggs

eggDrop(n, k) ==> Minimum number of trials needed to find the critical

floor in worst case.

eggDrop(n, k) = 1 + min{max(eggDrop(n - 1, x - 1), eggDrop(n, k - x)):

x in {1, 2, ..., k}}

**2) Overlapping Subproblems**  
Following is recursive implementation that simply follows the recursive structure mentioned above.

|  |
| --- |
| # include <stdio.h>  # include <limits.h>    // A utility function to get maximum of two integers  int max(int a, int b) { return (a > b)? a: b; }    /\* Function to get minimum number of trials needed in worst    case with n eggs and k floors \*/  int eggDrop(int n, int k)  {      // If there are no floors, then no trials needed. OR if there is      // one floor, one trial needed.      if (k == 1 || k == 0)          return k;        // We need k trials for one egg and k floors      if (n == 1)          return k;        int min = INT\_MAX, x, res;        // Consider all droppings from 1st floor to kth floor and      // return the minimum of these values plus 1.      for (x = 1; x <= k; x++)      {          res = max(eggDrop(n-1, x-1), eggDrop(n, k-x));          if (res < min)              min = res;      }        return min + 1;  }    /\* Driver program to test to pront printDups\*/  int main()  {      int n = 2, k = 10;      printf ("\nMinimum number of trials in worst case with %d eggs and "               "%d floors is %d \n", n, k, eggDrop(n, k));      return 0;  } |

Run on IDE

Output:

Minimum number of trials in worst case with 2 eggs and 10 floors is 4

It should be noted that the above function computes the same subproblems again and again. See the following partial recursion tree, E(2, 2) is being evaluated twice. There will many repeated subproblems when you draw the complete recursion tree even for small values of n and k.

E(2,4)

|

-------------------------------------

| | | |

| | | |

x=1/\ x=2/\ x=3/ \ x=4/ \

/ \ / \ .... ....

/ \ / \

E(1,0) E(2,3) E(1,1) E(2,2)

/\ /\... / \

x=1/ \ .....

/ \

E(1,0) E(2,2)

/ \

......

Partial recursion tree for 2 eggs and 4 floors.

Since same suproblems are called again, this problem has Overlapping Subprolems property. So Egg Dropping Puzzle has both properties (see [this](http://www.geeksforgeeks.org/archives/12635)and [this](http://www.geeksforgeeks.org/archives/12819)) of a dynamic programming problem. Like other typical [Dynamic Programming(DP) problems](http://www.geeksforgeeks.org/archives/tag/dynamic-programming), recomputations of same subproblems can be avoided by constructing a temporary array eggFloor[][] in bottom up manner.

**Dynamic Programming Solution**  
Following are C++ and Python implementations for Egg Dropping problem using Dynamic Programming.

* C++
* Java
* Python

|  |
| --- |
| # A Dynamic Programming based C++ Program for the Egg Dropping Puzzle  # include <stdio.h>  # include <limits.h>    // A utility function to get maximum of two integers  int max(int a, int b) { return (a > b)? a: b; }    /\* Function to get minimum number of trials needed in worst    case with n eggs and k floors \*/  int eggDrop(int n, int k)  {      /\* A 2D table where entery eggFloor[i][j] will represent minimum         number of trials needed for i eggs and j floors. \*/      int eggFloor[n+1][k+1];      int res;      int i, j, x;        // We need one trial for one floor and0 trials for 0 floors      for (i = 1; i <= n; i++)      {          eggFloor[i][1] = 1;          eggFloor[i][0] = 0;      }        // We always need j trials for one egg and j floors.      for (j = 1; j <= k; j++)          eggFloor[1][j] = j;        // Fill rest of the entries in table using optimal substructure      // property      for (i = 2; i <= n; i++)      {          for (j = 2; j <= k; j++)          {              eggFloor[i][j] = INT\_MAX;              for (x = 1; x <= j; x++)              {                  res = 1 + max(eggFloor[i-1][x-1], eggFloor[i][j-x]);                  if (res < eggFloor[i][j])                      eggFloor[i][j] = res;              }          }      }        // eggFloor[n][k] holds the result      return eggFloor[n][k];  }    /\* Driver program to test to pront printDups\*/  int main()  {      int n = 2, k = 36;      printf ("\nMinimum number of trials in worst case with %d eggs and "               "%d floors is %d \n", n, k, eggDrop(n, k));      return 0;  } |

Run on IDE

Output:

Minimum number of trials in worst case with 2 eggs and 36 floors is 8

Time Complexity: O(nk^2)  
Auxiliary Space: O(nk)

# Dynamic Programming | Set 12 (Longest Palindromic Subsequence)

Given a sequence, find the length of the longest palindromic subsequence in it. For example, if the given sequence is “BBABCBCAB”, then the output should be 7 as “BABCBAB” is the longest palindromic subseuqnce in it. “BBBBB” and “BBCBB” are also palindromic subsequences of the given sequence, but not the longest ones.

The naive solution for this problem is to generate all subsequences of the given sequence and find the longest palindromic subsequence. This solution is exponential in term of time complexity. Let us see how this problem possesses both important properties of a Dynamic Programming (DP) Problem and can efficiently solved using Dynamic Programming.

**1) Optimal Substructure:**  
Let X[0..n-1] be the input sequence of length n and L(0, n-1) be the length of the longest palindromic subsequence of X[0..n-1].

If last and first characters of X are same, then L(0, n-1) = L(1, n-2) + 2.  
Else L(0, n-1) = MAX (L(1, n-1), L(0, n-2)).

Following is a general recursive solution with all cases handled.

// Everay single character is a palindrom of length 1

L(i, i) = 1 for all indexes i in given sequence

// IF first and last characters are not same

If (X[i] != X[j]) L(i, j) = max{L(i + 1, j),L(i, j - 1)}

// If there are only 2 characters and both are same

Else if (j == i + 1) L(i, j) = 2

// If there are more than two characters, and first and last

// characters are same

Else L(i, j) = L(i + 1, j - 1) + 2

**2) Overlapping Subproblems**  
Following is simple recursive implementation of the LPS problem. The implementation simply follows the recursive structure mentioned above.

|  |
| --- |
| #include<stdio.h>  #include<string.h>    // A utility function to get max of two integers  int max (int x, int y) { return (x > y)? x : y; }    // Returns the length of the longest palindromic subsequence in seq  int lps(char \*seq, int i, int j)  {     // Base Case 1: If there is only 1 character     if (i == j)       return 1;       // Base Case 2: If there are only 2 characters and both are same     if (seq[i] == seq[j] && i + 1 == j)       return 2;       // If the first and last characters match     if (seq[i] == seq[j])        return lps (seq, i+1, j-1) + 2;       // If the first and last characters do not match     return max( lps(seq, i, j-1), lps(seq, i+1, j) );  }    /\* Driver program to test above functions \*/  int main()  {      char seq[] = "GEEKSFORGEEKS";      int n = strlen(seq);      printf ("The length of the LPS is %d", lps(seq, 0, n-1));      getchar();      return 0;  } |

Run on IDE

Output:

The length of the LPS is 5

Considering the above implementation, following is a partial recursion tree for a sequence of length 6 with all different characters.

L(0, 5)

/ \

/ \

L(1,5) L(0,4)

/ \ / \

/ \ / \

L(2,5) L(1,4) L(1,4) L(0,3)

In the above partial recursion tree, L(1, 4) is being solved twice. If we draw the complete recursion tree, then we can see that there are many subproblems which are solved again and again. Since same suproblems are called again, this problem has Overlapping Subprolems property. So LPS problem has both properties (see [this](http://www.geeksforgeeks.org/archives/12635)and [this](http://www.geeksforgeeks.org/archives/12819)) of a dynamic programming problem. Like other typical [Dynamic Programming(DP) problems](http://www.geeksforgeeks.org/archives/tag/dynamic-programming), recomputations of same subproblems can be avoided by constructing a temporary array L[][] in bottom up manner.

**Dynamic Programming Solution**

* C++
* Java
* Python

|  |
| --- |
| # A Dynamic Programming based Python program for LPS problem  # Returns the length of the longest palindromic subsequence in seq  #include<stdio.h>  #include<string.h>    // A utility function to get max of two integers  int max (int x, int y) { return (x > y)? x : y; }    // Returns the length of the longest palindromic subsequence in seq  int lps(char \*str)  {     int n = strlen(str);     int i, j, cl;     int L[n][n];  // Create a table to store results of subproblems         // Strings of length 1 are palindrome of lentgh 1     for (i = 0; i < n; i++)        L[i][i] = 1;        // Build the table. Note that the lower diagonal values of table are      // useless and not filled in the process. The values are filled in a      // manner similar to Matrix Chain Multiplication DP solution (See      // <http://www.geeksforgeeks.org/archives/15553>). cl is length of      // substring      for (cl=2; cl<=n; cl++)      {          for (i=0; i<n-cl+1; i++)          {              j = i+cl-1;              if (str[i] == str[j] && cl == 2)                 L[i][j] = 2;              else if (str[i] == str[j])                 L[i][j] = L[i+1][j-1] + 2;              else                 L[i][j] = max(L[i][j-1], L[i+1][j]);          }      }        return L[0][n-1];  }    /\* Driver program to test above functions \*/  int main()  {      char seq[] = "GEEKS FOR GEEKS";      int n = strlen(seq);      printf ("The lnegth of the LPS is %d", lps(seq));      getchar();      return 0;  } |

Run on IDE

Output:

The lnegth of the LPS is 7

Time Complexity of the above implementation is O(n^2) which is much better than the worst case time complexity of Naive Recursive implementation.

This problem is close to the [Longest Common Subsequence (LCS) problem](http://www.geeksforgeeks.org/archives/12998). In fact, we can use LCS as a subroutine to solve this problem. Following is the two step solution that uses LCS.  
1) Reverse the given sequence and store the reverse in another array say rev[0..n-1]  
2) LCS of the given sequence and rev[] will be the longest palindromic sequence.  
This solution is also a O(n^2) solution.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

**References:**  
<http://users.eecs.northwestern.edu/~dda902/336/hw6-sol.pdf>