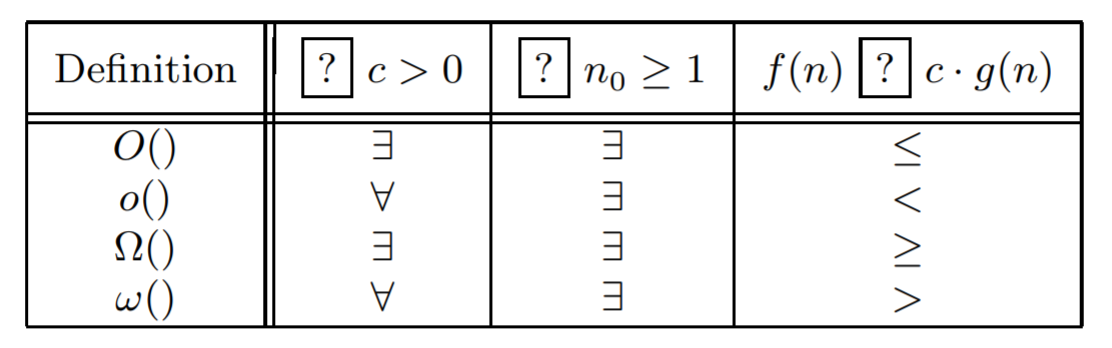
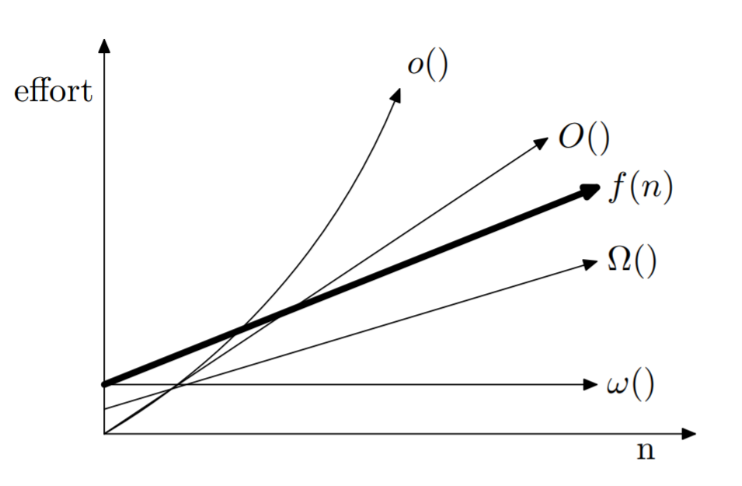
## **Asymptotic analysis and notation**



|  |  |
| --- | --- |
| Transitivity: | O, o, Θ, ω, Ω |
| Reflexivity: | O, Θ, Ω |
| Symmetry: | Θ |



|  |  |  |
| --- | --- | --- |
| BIG O | tight upper-bound | |
| Little o | upper-bound that cannot be tight | |
| BIG Omega | tight lower bound | |
| Little-omega | the loose lower bound | |
| BIG THETA | If and only if f(n)=O(g(n)) and g(n) = O(f(n)) | |
| log (n!) = Θ(n log (n)) | True | |
| // Small Oh | | T |
|  |  | |

## **DATA STRUCTURES**

|  |  |
| --- | --- |
| STACK (LIFO) | Last In First Out, 3 operations: Push(), Pop(), Top() |
| QUEUE (FIFO)  IDEA:  Multiple Linked QUEUE, 1, 3,4,2,3,1() | First In First Out, dequeue(), enqueue() |
| Linked LISTS | a sequence of records where every record has a field that points to the next record |
| SET |  |
| MAP  (Associative Array) | Put(k,v) {associates key: k with value v}, get(k), contain(k) |
|  |  |
| TREE  (n vertices and n-1 edges) | A connected acyclic graph. |
| Binary Tree | A rooted tree where every node has at most two children.  a[1] is the root,  for node a[j], the child nodes are a[2j] and a[2j+1]  for node a[j], the parent node is a[j/2] (integer division) |
| Binary Search Trees  Operations: | Left subtree less than x, right subtree greater than x.  Insert(x), delete(x), search(x) |
| B-tree | Self-balancing search trees, can have more than two children and different nodes can have different number of children. |
| HEAPS | http://mathworld.wolfram.com/Heap.html |
|  |  |

## **Divide and Conquer**

# Insertion sort: pick up one element and compare with the previous one, insert it into right position.

|  |  |
| --- | --- |
| ***T(n) = aT(n/b) + f(n)***  Divide single task into sub-problems and each has a size of n/b. f(n) is the time for dividing and merging. | * **Case 1.** If *f(n) = O(nc)* where *c < logb a*, then *T(n) = θ(n^logb a)* |
| * **Case 2.** If it is true, for some constant *k* ≥ 0, that *f(n) = θ(nc logk n)* where *c = logb a*, then *T(n) = θ(nc logk+1 n)* |
| * **Case 3.** If it is true that *f(n) = Ω(nc) w*here *c > logb a*, then *T(n) = θ(f(n))* |
| BINARY SEARCH  Θ(log n) | # Given a sorted arrary A and key,  BinarySearch (A, low, high, key)  If(high – low < 5) {  return}  Mid = (low+high)/2  If(key < A[mid]) {  Return BinarySearch (A, low, mid, key)  }else {  Return BinarySearch (A, mid+1, high, key) } |
| Merge Sort  T(n) = 2T(n/2) + Θ (n),  T(n) = Θ(n log n) | Merging worst case, x + y -1 comparisons. |
| Recursively sorting the first and second halves of the given array, and then merging the sorted sections. |
| Quick Sort  Selection a ‘partition’ element, and then left or right. | \*without need extra space, no merge part.  T(n) = T(n1)+T(n2) + cn, where n1+n2=n-1  Worst Case: O(n^2)  Best case: O(n log n)  Average Case: O(n log n) |
| MEDIAN FINDING  QuickSelect  1.Probabilistic Version  T(n)=2cn+T(3n/4)  PS: (2 reflects the expected # of times of partitioning.) | 1. Randomly lands at k’, value x. 2. All elements greater than x go right, rest go left. 3. If k == k’, got it; If k<k’, go A’; if k>k’, go A’’.   T(n) = O(n) |
| 2. Median of Medians  Assuming 5n elements > n medians > median of medians (x).  Then: n + n/2 (30%) < x and n+n/2 > x | T(n) = cn + T(n/5)+T(7n/10) <= 10cn  ( Provement: T(n/5)<2cn, T(7n/10)<7cn)  T(n) = O(n) |
| CLOSEST PAIR OF POINTS | 1.Split points > two equal-sized subsets by x median. O(n)  2.Solve problem recursively to left and right. Define /delta to be minimum of ? |
| POLYNOMIAL MULTIPLICATION AND FAST FOURIER TRANSFORM |  |

## **Greedy Method**

|  |  |
| --- | --- |
| Merging Sorted Lists  Construct Data Structure:   1. Remove the two smallest elements 2. Add an element 3. Repeat these steps until we have only one element   Merge the two shortest remaining arrays.  For i = 1 to n-1  {Remove two smallest elements from the heap}//2log(n)  {add a new element to the merged list} // log(n)  In total: T(n) = O(n log n) | https://upload.wikimedia.org/wikipedia/commons/thumb/e/e6/Merge_sort_algorithm_diagram.svg/330px-Merge_sort_algorithm_diagram.svg.png |
| Knapsack Problem:  Determine which items to take and how much of each item so that the total weight is ≤ C, and the total value (profit) is maximized |  |
| Minimum Spanning Tree:  (A spanning tree of a graph is a tree that has all nodes in the graph, and all edges come from the graph)  Problem: Minimum weighted edge that does NOT create a cycle.  {## Kruskal’s Greedy Algorithm  Sort edges: e[1], e[2], .. e[m].  //O(m log m) < O(m log n^2), bc m<n^2,  Initialize counter j = 1  Initialize tree T to empty  While (number of edges in Tree < n-1) {  Does adding an edge e[j] create a cycle?  //O(n log n)  If No, add edge e[j] to tree T  } // In total O(m log n) time.  Find(x) = O(log n)  Union(x,y) = O(log n)  Prim’s algorithm  Boruvka’s algorithm |  |
| Proof of Correctness of Krusal’s Algoritm. P71  This returns the ‘set’ that contains the elements x.  Merges the two sets containing elements x and y. |

## **Dynamic Programming**

|  |  |
| --- | --- |
| 1.Bottom up traversal  2.Stores results of sub-solutions to  avoid repeated calculation.   * Optimal Substructure * Overlapping sub-problems | Example:  Fibonacci Numbers: f(1)=f(2)=1, f(n)=f(n-1)+f(n-2)  If using DP, O(n)  If using D&C, much slower. |
| DP Template Steps: | 1.Notation 2. Optimality 3. Recurrences 4. Algorithm (NORA) |
| Matrix Chain Multiplication  (Determine the order in which gives the minimum calculation.) | A\_1=a x b, A\_2=b x c. Total cost/time is abc. |
| All Pairs Shortest Path(APSP)  Find the shortest distance between every pair of nodes.  Algorithm:  For i=1 to n  For j=1 to n  D[0][i,j] := W[i,j]  For k=1 to n  For i=1 to n  For j=1 to n  D[k][i, j]=min{D[k-1][i, j], D[k-1][i, k] + D[k-1][k, j] }  O(n^3) time complexity  O(n^2) in space | Notation:  D^k (i,j), is the length of the shortest path from node i to node j and allowed using nodes{i…k} in between.  Optimality:  Using contradiction to prove. If we can find another way which is shorter in this sub-portion, then in total, the shortest distance would be the rest + shorter sub-portion, which is against the optimality of whole problem.  Recurrence Relation:  D^(k)[i,j] = min{ D^(k-1)[i,j], D^(k-1)[i, k] + D^(k)[k, j] }  From a graph,  1.all the distance information between two nodes were given  2. Using these distance info, reconstruct the virtual 3D structure of graph data.!!!  3. Lets see, what we can do with that 3D structure… |
| Maximum value contiguous subsequence (MVCS)  Given an Array A(1…n), and we need to find a subarray A(i…j), such that the sum of the elements in the subarray is maximized.  Algorithm: Ref: P87 O(n) | Algorithm: MVCS2, similar to Brutal force, O(n^2)  Algorithm: MVCS3,  Notation: MVCS(i) represents the max value contiguous subarray that ends at position i.  Optimality: if MVCS(i) includes any elements before i, then MVCS(i-1) must the max value contiguous subarray ending at position i-1.  Recurrence Relation:  MVCS(i)= max{MVCS(i-1)+A[i], A[i]} |
| Longest Increasing Subsequence(LIS)  X(1)=1  For i = 2 : n  For j = i : n  If j < i, and A[j]<A[i]:  X(i)=max{X(j)}+1,  O(n^2) | Notation:  Suppose X(i) represents the size of longest strictly increasing  subsequence that ends at position i.  Optimality:  Yes, the longest increasing sequence requires that local optimality transfer to whole optimality by X(i) = max{X(j)} + 1 |
|  |  |
|  |  |

## **Graph Traversal Techniques**

|  |  |
| --- | --- |
| **GRAPH**  G=(V,E) vertices and edges  Representations: Adjacency Matrix A  Articulation points (cut Vertices.)  ArticulationPoints2 | Directed: edge (v1,v2) != edge(v2,v1)  Undirected: edge (v1,v2) = edge(v2,v1)  This requires more memory.  For any graph, the sum of the degrees of vertices equals twice the number of edges.  **Hamiltonian path**(or traceable **path**) is a **path** in an undirected or directed graph that visits each vertex exactly once  Vertex coloring: no two adjacent vertices share the same color |
| We can construct an undirected graph on 6 vertices where the degrees of 6 vertices are: 1, 1, 2, 2, 2, 3 | F  For any graph, the sum of the degrees of vertices equals twice the number of edges. |
| Tree edges (u,v) | If node v is first discovered by exploring edge(u,v) |
| Back Edges (u,v) (Self loops are also BEs) | Forward edge and rest all are Cross edges. |
| http://www.algolist.net/Algorithms/Graph/Undirected/Depth-first\_search | |
| Depth First Search (DFS)  Preferred Data Structure: Stack  1. Select an unvisited node s, visit it, treat as current node.  2. Find an unvisited neighbor, visit it, mark as current node.  3. If no neighbors available, backtrack to parent node.  4. Repeat 2~3 until no more nodes can be visited.  Time Complexity:  O( n+ m ) … n nodes, m edges  If graph is connected, O(n+m) = O(m) | Algorithm DFS(input: graph G)  Stack T, int s, int x  While (G has an unvisited node) do {  s= an unvisited node  visit(s)  T.push(s)  While(T is not empty){  x = T.top()  if (x has an unvisited neighbor y){  visit(y)  T.push(y)  } else {  T.pop()  } } }  Because graph has at least n-1 edges >>>> m >= (n-1), and therefore 2m+1>= (n+m), O(2m+1) = O(m) |
| APPLICATIONS: |  |
| 1. Connectivity 2. Minimum Spanning Trees 3. Biconnectivity | 1. Articulation points (cut Vertices.) 2. If a connected graph has no articulation points, it is called biconnected. |
|  |  |
| Breadth First Search (BFS)  Preferred Data Structure: Queue.  1.Select an unvisited node s, visit it, have it be the root in a BFS tree, called current level.  2.visit all the unvisited neighbors of x, newly visited nodes called the next current level.  3.Repeat until no more nodes can be visited.  4.if there are still nodes unvisited, repeat from 1. | Algorithm BFS (input: graph G)  Queue Q; Integer s, x  While (G has an unvisited node) do  S := an unvisited node  Visit(s)  Enqueue(s, Q)  While (Q is not empty) do  X := Dequeue(Q)  For (unvisited neighbor y of x) do  Visit(y)  Enqueue(y, Q) |
|  |  |
|  |  |

## **Branch and Bound**

|  |  |
| --- | --- |
| B&B TEMPLATE | 1.Model the solution in graph, where  a) Node represents a partial or complete solution.  b) Each edge represents a step/decision/constraint in the solution building process.  2. Develop a strategy to find upper bound and lower bound.  3. Conduct a BFS, i)Bound each node ii) Discard a node due to its bound. Iii) Branch the rest nodes.  4.Terminate if solution meets requirement |
|  |  |
| KNAPSACK  Problem  The knapsack can hold 10 kgs | i5 |
|  |  |
|  |  |
| Job Assignment Problem  Given jobs, resources, and cost matrix (cost if resource I does job j), how to assign jobs to resources, such that each job is done by a different resource, and the overall cost is minimu.  Calculation of Cost Function:  For each worker, we choose job with minimum cost from list of unassigned jobs (take minimum entry from each row).  Or take minimum entry from each column. | jobassignment2 |
| jobassignment6 | |

## **NP Completeness**

|  |  |
| --- | --- |
| Define a Turing Machine as a 7-tuple M =<Q,T,B,S,q\_0, F, /delta>,  In short, if the TM1 is ends in one of the final states after reading the entire word w, then we say w is accepted by TM1, otherwise, the TM1 is set to reject the word w.  The string of the valid input is then said to belong to the language L corresponding to the sorting problem, the string of the invalid input is then said to not belong in the language L.  A problem p, if we can design a Turing machine M that accept encoding p as language L, then the machine M is said to solve the problem p. | |
| P={ L | L is accepted by a deterministic Turing Machine in polynomial time}  The TM takes most O(n^c) steps to accept a string of length n. | NP={L | L is accepted by a non-deterministic Turing Machine in polynomial time}  Solution verified in polynomial time.  The TM takes most O(n^c) steps to accept a string of length n. |
| P-Complete: Problem p if problem p belongs C, and can be used to solve all of the problem p belongs in class C. | NP-Complete: A problem X if X is in NP and every problem in class NP is reducible to X in polynomial time.  NP-HARD: TSP(Traveling Salesperson Problem) |
| https://cdncontribute.geeksforgeeks.org/wp-content/uploads/NP-Completeness-1.png | |
| Boolean Satisfiability (SAT) Problem  (x1 or x2 or x3) & (x1 or nx2 or nx3)  & (nx1 or x2 or nx3) & (nx1 or nx2 or nx3)  &(nx1 or x2 or x3) & (x1 or nx2 or x3)  &(x1 or nx2 or nx3) & (nx1 or x2 or nx3)  (nx1 represents the negation of x1.) | If we assign x1=x2=x3=true, overall clause gives:  T and T and T and F and  T and T and T and T, which is false. |
| The Template to prove that problem X is NP-Complete | |
| 1. Show that X is in NP, a polynomial time verifier exists for X. 2. Select CSAT or another known NP-complete problem S. 3. Show a polynomial algorithm to reduce S to X.   https://image.slidesharecdn.com/np-completenessp2-160417125728/95/npcompleteness-ii-3-638.jpg?cb=1491947711 | |
| Example: NP-COMPLETE PROBLEMS:  Clique.  A clique is a set of vertices such that there is an edge between each pair of vertices in that set.  (A group of people who all know each other.) | K-Clique problem:  Given a graph G, identify whether or not G has a clique of size K.  (This is a NP, A non-deterministic TM can randomly select k vertices and then check whether or not the k vertices are connected. Polynomial Time.)  Prove that a polynomial time reduction from every problem in NP to clique problem, then clique problem is NP-hard. |
| Transfer an instance of CSAT into clique problem. Each vertex is connected to all vertices in other clauses except the vertices that correspond to their negations. | Proof that Clique Problem is NP-hard  Given an instance of CSAT with k clauses, we make a vertex for each literal, such as x. if we had a solve of k-clique problem, we can use it to solve CSAT, in other words, k-clique is an NP-hard problem. |

## **Example**

|  |  |
| --- | --- |
| Independent Set={G,k} where G has an independent set of size k}  An independent set is a set S of vertices such that for every two vertices in S, there is no edge connecting the two.  This is an NP-hard problem since it is the dual of the clique problem  Given a graph G, we can construct the complement graph G’ that has the same set of vertices, and an edge exists in G’ if and only if the corresponding edge does not exist in graph G. | https://upload.wikimedia.org/wikipedia/commons/thumb/3/34/Independent_set_graph.svg/220px-Independent_set_graph.svg.png |
| https://image.slidesharecdn.com/np-completeness-141204140156-conversion-gate02/95/np-completeness-30-638.jpg?cb=1493167461 | A clique of size k exists in G if and only if an independent set of size k exists in G’? |
|  |  |
| Vertex Cover:  Given a graph G, we would like to find a smallest set of vertices, such that every edge in G is incident upon at least one of the vertices in the selected set. | Given a vertex cover, the remaining set of vertices is an independent set. |
| https://image.slidesharecdn.com/np-completeness-141204140156-conversion-gate02/95/np-completeness-33-638.jpg?cb=1493167461 | |
|  |  |
|  |  |
|  |  |

|  |  |
| --- | --- |
| Hamiltonian Path and Hamiltonian Cycle  If there is a path that includes every vertex exactly one. | Reducing Hamiltonian Path problem to Hamiltonian Cycle problem:  Given a graph: G=(V,E), construct a graph G’=(V’,E’),  V’ = V union{z}, where z is a new vertex  E’ = E union{(z,v) | all v in V}  Claim:  G has a Hamiltonian Path if and only if G’ has Hamiltonian Cycle. |
|  |  |
| Graph k-coloring: Given a graph G and an integer k, does there exist a valid vertex coloring that uses only k colors? (Color the vertices of the graph with no two adjacent vertices has the same color.) | |
| Graph 3-coloring is NP-complete |  |
|  |  |
|  |  |
|  |  |
|  |  |

**Traveling Salesman Problem**

|  |  |
| --- | --- |
| Given a set of cities and distance between every pair of cities, the problem is to find the shortest possible tour that visits every city exactly once and returns to the starting point. | Consider two cases:  1.The original problem is based on a graph in Euclidian space and therefore satisfies the triangle inequality  2.The original problem does not satisfy the triangle inequality. |
| http://www.clipular.com/c/6584134164807680.png?k=ZS9LM2MlH5TyywxznfWDzS1CO2A | The lower bound:  1/2 Sum{(u1,v), (u2,v)}, where v in V, and (u1,v), (u2, v) are the two least cost edges adjacent to v.  **Time Complexity:**The worst case complexity of Branch and Bound remains same as that of the Brute. |
|  |  |
|  |  |

## **Sample**

|  |  |
| --- | --- |
|  |  |
| Final Exam | gcd(n,m) |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |

gcd(n,m) {

r = n%m

if r == 0 return m;

// else

return gcd(m, r)

}