|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Master Theorem in Data Structure - Dot Net Tutorials   **Prim’s** – Start **anywhere**, expand **min connected** edges. **Kruskal’s** – Start **minimum**, connect components greedily. **Dijkstra’s** – **Source**, expand **nearest** with priority queue. **Topological Sort** – **DAG**, process **dependencies** with DFS/Kahn’s. **Floyd-Warshall** – **All-pairs**, update **D0 → D1 → D2** iteratively. **Bellman-Ford** – **V-1 relaxations**, detect **negative cycles**.   |  |  | | --- | --- | | **Naive** | void naive(string t, string p) { int n = t.size(), m = p.size(); for (int i = 0; i <= n - m; i++) { int j = 0; while (j < m && t[i + j] == p[j]) j++; if (j == m) cout << i << " "; } } | | **RK** | void rk(string t, string p, int q = 101) { int d = 256, n = t.size(), m = p.size(), h = 1, tp = 0, tt = 0; for (int i = 0; i < m - 1; i++) h = (h \* d) % q; for (int i = 0; i < m; i++) { tp = (d \* tp + p[i]) % q; tt = (d \* tt + t[i]) % q; } for (int i = 0; i <= n - m; i++) { if (tp == tt) { int j = 0; while (j < m && t[i + j] == p[j]) j++; if (j == m) cout << i << " "; } if (i < n - m) { tt = (d \* (tt - t[i] \* h) + t[i + m]) % q; if (tt < 0) tt += q; } } } | | **KMP** | void kmp(string t, string p) { int n = t.size(), m = p.size(); vector<int> lps(m); for (int i = 1, len = 0; i < m;) { if (p[i] == p[len]) lps[i++] = ++len; else if (len) len = lps[len - 1]; else lps[i++] = 0; } for (int i = 0, j = 0; i < n;) { if (t[i] == p[j]) i++, j++; if (j == m) cout << i - j << " ", j = lps[j - 1]; else if (i < n && t[i] != p[j]) j ? j = lps[j - 1] : i++; } } |   **\*HASHING**\* hashing - maps keys to indices in hash table - used for fast insert/search/delete - avg. time O(1) - suit. for DBs, caches, symbol tables - depends on good hash func. - poor design → collisions & slow ops - appl. in sets, maps, indexing \*\*\*\***HASH FUNCTION**\* hash function - generates index from key - aims for uniform dist. - should be fast & deterministic - ex: modulo, multiplicative, folding - must reduce clustering & avoid collisions - ex: h(k) = k mod m \*\*\*\***COLLISION\*** collision - occurs when 2 keys hash to same index - unavoidable when key space > table size - handled via chaining or open addressing - too many collisions = degraded perf. **\*\*\*\*CHAINING\*** chaining - collision handling via linked list at each index - all colliding keys stored in same bucket - allows infinite entries per slot - simple to implement - needs extra space for links - ex: insert at head of list at index \*\*\*\***OPEN ADDRESSING**\* open addressing - resolves collision by finding alt. empty slot - all elements stored in table itself - no external structs - types: linear probing, quadratic probing, double hashing \*\*\*\***LINEAR PROBING**\* linear probing - checks next slot i+1, i+2... till empty - fast but causes clustering - insert/search with wrap around - ex: h(k) = (h(k) + i) mod m \*\*\*\***QUADRATIC PROBING**\* quadratic probing - checks i² steps from orig. pos - avoids primary clustering - still may suffer secondary clustering - ex: h(k) = (h(k) + c1·i + c2·i²) mod m **\*\*\*\*DOUBLE HASHING**\* double hashing - uses second hash func. for step size - best among probing for fewer collisions - reduces clustering - ex: h(k, i) = (h1(k) + i·h2(k)) mod m \*\*\*   |  |  | | --- | --- | | **Activity** | void select(vector<pair<int,int>>&a){sort(a.begin(),a.end(),[](auto &x,auto &y)  {return x.second<y.second;});int ct=1,end=a[0].second;for(int i=1;i<a.size();i++)  if(a[i].first>=end){ct++;end=a[i].second;}cout<<ct;} | | **Job** | bool cmp(pair<int,pair<int,int>>&a,pair<int,pair<int,int>>&b)  {return a.second.second>b.second.second;} void jobSeq(vector<pair<int,pair<int,int>>> &jobs,int n){sort(jobs.begin(),jobs.end(),cmp);vector<int>slot(n,-1);int p=0;  for(auto &j:jobs){for(int i=min(n,j.second.first)-1;i>=0;i--)if(slot[i]==-1)  {slot[i]=j.first;p+=j.second.second;break;}}cout<<p;} | | **HufF** | struct N{char d;int f;N\*l,\*r;};struct cmp{bool operator()(N\*a,N\*b){return a->f>b->f;}};void huffman(vector<pair<char,int>>&frq){priority\_queue<N\*,vector<N\*>,cmp>pq;  for(auto &[c,f]:frq)pq.push(new N{c,f,0,0});while(pq.size()>1){N\*a=pq.top();pq.pop();  N\*b=pq.top();pq.pop();pq.push(new N{'#',a->f+b->f,a,b});}/\*Tree ready at pq.top()\*/} | | **Fract** | bool cmp(pair<int,int>&a,pair<int,int>&b){return (double)a.first/a.second >  (double)b.first/b.second;}  double fracKnapsack(vector<pair<int,int>>&a,int W)  {sort(a.begin(),a.end(),cmp);double v=0;for(auto &[val,wt]:a)  {if(W>=wt)v+=val,W-=wt;else{v+=val\*((double)W/wt);break;}}return v;} |   \***ONLINE SORT**\* online sort - proc. input piece by piece - works w/o full input - ready to give partial results - suit. for streaming or real-time appl. - input handled as it comes - can't go back & modify - uses only past & curr. data - handles dynamic input - low mem. adaptive - insertion sort = proto. example - used in live stock price updates - log file proc. - streamed sensor data sort \*\*\*  \***STABLE SORT**\* stable sort - keeps orig. order of = elements - key in multi-level sorts (e.g. sort by age then name) - for = pri. key order is same - cntg, radix, merge, insertion, bubble are stable - crit. in DBs & record mgmt. - sorts nested data consistently - handles duplicates well - ensures same output in re-runs - ex: merge sort for DB rows - radix sort on id & score \*\*\*  \***IN PLACE SORT**\* in place sort - uses min. mem. (O(1) aux. space) - no extra buffers/arrays - swaps in input arr. - mem. efficient - good for embedded sys. - quick, selection, insertion, heap sort are in-place - may be unstable - faster due to cache locality - ex: quick sort for file indexing - insertion sort on MCUs - selection sort in low-RAM envs \*\*\*   |  |  | | --- | --- | | **BUBBLE** | void bubbleSort(int a[], int n){for(int i=0;i<n-1;i++)for(int j=0;j<n-i-1;j++)if(a[j]>a[j+1])swap(a[j],a[j+1]);} O(n) / O(n²) / O(n²) | | **INSERT** | void insertionSort(int a[], int n){for(int i=1;i<n;i++){int k=a[i],j=i-1;while(j>=0&&a[j]>k)a[j+1]=a[j--];a[j+1]=k;}} O(n) / O(n²) / O(n²) | | **SELECT** | void selectionSort(int a[], int n){for(int i=0;i<n-1;i++){int m=i;for(int j=i+1;j<n;j++)if(a[j]<a[m])m=j;swap(a[i],a[m]);}} O(n²) / O(n²) / O(n²) | | **QUICK** | int part(int a[],int l,int r){int p=a[r],i=l;for(int j=l;j<r;j++)if(a[j]<p)swap(a[i++],a[j]);swap(a[i],a[r]);return i;} void quickSort(int a[],int l,int r){if(l<r){int p=part(a,l,r);quickSort(a,l,p-1);quickSort(a,p+1,r);}} O(n log n) / O(n log n) / O(n²) | | **RAND-QS** | int part(int a[],int l,int r){int p=a[r],i=l;for(int j=l;j<r;j++)if(a[j]<p)swap(a[i++],a[j]);swap(a[i],a[r]);return i;} void randQuickSort(int a[],int l,int r){if(l<r){int i=l+rand()%(r-l+1);swap(a[i],a[r]);int p=part(a,l,r);randQuickSort(a,l,p-1);randQuickSort(a,p+1,r);}} O(n log n) / O(n log n) / O(n²) | | **MERGE** | void merge(int a[],int l,int m,int r){vector<int>v;int i=l,j=m+1;while(i<=m&&j<=r)v.push\_back(a[i]<a[j]?a[i++]:a[j++]);while(i<=m)v.push\_back(a[i++]);while(j<=r)v.push\_back(a[j++]);for(int k=0;k<v.size();k++)a[l+k]=v[k];} void mergeSort(int a[],int l,int r){if(l<r){int m=(l+r)/2;mergeSort(a,l,m);mergeSort(a,m+1,r);merge(a,l,m,r);}} O(n log n) / O(n log n) / O(n log n) | | **HEAP** | void heapify(int a[],int n,int i){int l=2\*i+1,r=2\*i+2,m=i;if(l<n&&a[l]>a[m])m=l;if(r<n&&a[r]>a[m])m=r;if(m!=i){swap(a[i],a[m]);heapify(a,n,m);}} void heapSort(int a[],int n){for(int i=n/2-1;i>=0;i--)heapify(a,n,i);for(int i=n-1;i>0;i--){swap(a[0],a[i]);heapify(a,i,0);}} O(n log n) / O(n log n) / O(n log n) | | **COUNT** | void countingSort(int a[],int n){int mx=\*max\_element(a,a+n);vector<int>c(mx+1);for(int i=0;i<n;i++)c[a[i]]++;int i=0;for(int j=0;j<=mx;j++)while(c[j]--)a[i++]=j;} O(n + k) / O(n + k) / O(n + k) | | **RADIX** | void countSort(int a[],int n,int e){int o[n],c[10]={};for(int i=0;i<n;i++)c[(a[i]/e)%10]++;for(int i=1;i<10;i++)c[i]+=c[i-1];for(int i=n-1;i>=0;i--)o[--c[(a[i]/e)%10]]=a[i];for(int i=0;i<n;i++)a[i]=o[i];} void radixSort(int a[],int n){int m=\*max\_element(a,a+n);for(int e=1;m/e>0;e\*=10)countSort(a,n,e);} O(d\*(n + k)) | | **BUCKET** | void bucketSort(float a[],int n){vector<vector<float>>b(n);for(int i=0;i<n;i++){int bi=n\*a[i];b[bi].push\_back(a[i]);}for(auto &v:b)sort(v.begin(),v.end());int i=0;for(auto &v:b)for(float x:v)a[i++]=x;} O(n + k) avg / O(n²) worst |   **int linear\_search**(vector<int>& a, int x) {for (int i = 0; i < a.size(); i++) if (a[i] == x) return i;return -1;} // O(n)  **int binary\_search\_iter**(vector<int>& a, int x) {int l = 0, r = a.size() - 1;while (l <= r) {int m = (l + r) / 2;if (a[m] == x) return m;if (a[m] < x) l = m + 1;else r = m - 1;}return -1;} // O(log n)int **binary\_search\_rec**(vector<int>& a, int l, int r, int x) {if (l > r) return -1;int m = (l + r) / 2;if (a[m] == x) return m;if (a[m] < x) return binary\_search\_rec(a, m + 1, r, x);return binary\_search\_rec(a, l, m - 1, x);} // O(log n)  **void tower\_of\_hanoi**(int n, char src, char aux, char dest) {if (n == 0) return;tower\_of\_hanoi(n - 1, src, dest, aux);cout << src << " -> " << dest << "\n";tower\_of\_hanoi(n - 1, aux, src, dest);} // O(2^n)  **int expo\_search**(vector<int>& a, int x) {if (a[0] == x) return 0;int i = 1;while (i < a.size() && a[i] <= x) i \*= 2;return binary\_search(a, i / 2, min(i, static\_cast<int>(a.size()) - 1), x);} // O(log i)  **int fib**(int n) {if (n <= 1) return n;return fib(n - 1) + fib(n - 2);} // O(2^n)  log\_b a = log a / log b  log\_b (x^y) = y log\_b x  log\_b (xy) = log\_b x + log\_b y  log\_b (x/y) = log\_b x - log\_b y  a (1 - r^n) / (1 - r)  S\_∞ = a / (1 - r), if |r| < 1  a\_n = a \* r^(n-1)  S\_n = (n(n + 1)) / 2  S\_n² = (n(n + 1)(2n + 1)) / 6  S\_n³ = [(n(n + 1)) / 2]²  2^n terms:\*\* S\_n = 2^(n+1) - 1 |  |
| **\*P CLASS\*** P class - probs solvable in poly time - deterministic algos run in O(n^k) for const. k - feasibly computable - real-time solutions possible - ex: sorting, matrix mult., Dijkstra (shortest path), Prim’s/Kruskal’s (MST), bipartite check \*\*\*\***NP CLASS\* NP** class - probs where solutions verifiable in poly time - may not be solvable quickly - non-deterministic guess + fast verification - includes all P probs - ex: 3-SAT, Hamiltonian path, subset sum, graph coloring \*\*\*\***POLYNOMIAL TIME**\* poly time - runtime grows at poly rate w.r.t. input size - O(n), O(n²), O(n³)... - tractable in practice - defines class P - opp. of exponential time O(2^n) \*\*\*\***POLYNOMIAL TIME VERIFICATION**\* poly time verification - checking a given solution takes poly time - core of NP defn - solver can be slow, verifier fast - verifier = deterministic algo taking input + certificate → true/false - ex: given a tour, verify TSP cost < K in O(n) \*\*\*\***NP COMPLETE\* NP complete** - class of hardest problems in NP - if any NP-C prob solved in P time → P = NP - every NP prob can be reduced to any NP-C in poly time - both in NP & NP-hard - ex: 3-SAT, clique, subset sum, vertex cover, TSP (decision), Hamiltonian cycle \*\*\*\***NP HARD**\* NP hard - at least as hard as NP-C probs - may not be in NP (verification may be non-poly) - includes optimization & undecidable probs - no known poly algo - ex: halting prob, optimization TSP, job scheduling, logic puzzles, AI strategy search \*\*\*  **\*IMPORTANCE OF NP COMPLETENESS\*** - boundary line between easy & hard probs - defines what may never be efficiently solved - if P ≠ NP, no fast algo for NP-C probs - pushes need for approx, heuristics, backtracking, brute-force - unifies wide range of problems under same complexity umbrella \*\*\*  **\*EXAMPLES\***  P → merge sort, matrix chain mult., BFS, DFS, MST  NP → 3-SAT, Hamiltonian path, coloring  NP complete → subset sum, TSP (decision), SAT, clique, vertex cover  NP hard → halting prob, TSP (min), knapsack (max), Sudoku solving \*\*\*   |  |  | | --- | --- | | SUBSET SUM | ```cpp bool subsetSum(int arr[], int n, int sum){vector<vector> dp(n+1,vector(sum+1,false));for(int i=0;i<=n;i++)dp[i][0]=true;for(int i=1;i<=n;i++)for(int j=1;j<=sum;j++)dp[i][j]=j<arr[i-1]?dp[i-1][j]:dp[j] | | TSP | ```cpp int tsp(vector<vector> &dist,int mask,int pos,vector<vector> &dp){int n=dist.size();if(mask==(1<<n)-1)return dist[pos][0];if(dp[mask][pos]!=-1)return dp[mask][pos];int ans=1e9;for(int city=0;city<n;city++)if((mask&(1<<city))==0)  ans=min(ans,dist[pos][city]+tsp(dist,mask |  |  |  | | --- | --- | | **FIB** | int fib(int n,vector<int>&dp){if(n<=1)return n;if(dp[n]!=-1)  return dp[n];return dp[n]=fib(n-1,dp)+fib(n-2,dp);} | | **FIG TABU** | int fib(int n){vector<int>dp(n+1);dp[0]=0;dp[1]=1;for(int i=2;i<=n;i++)dp[i]=dp[i-1]+dp[i-2];return dp[n];} | | **0/1** | int knap(int n,int W,vector<int>&wt,vector<int>&val){vector<vector<int>>dp(n+1,vector<int>(W+1));for(int i=1;i<=n;i++)for(int w=0;w<=W;w++)dp[i][w]=w<wt[i-1]?dp[i-1][w]:max(dp[i-1][w],val[i-1]+dp[i-1][w-wt[i-1]]);return dp[n][W];} | | **LCS** | int lcs(string a,string b){int n=a.size(),m=b.size();vector<vector<int>>dp(n+1,vector<int>(m+1));for(int i=1;i<=n;i++)for(int j=1;j<=m;j++)dp[i][j]=a[i-1]==b[j-1]?1+dp[i-1][j-1]:max(dp[i-1][j],dp[i][j-1]);return dp[n][m];} | | **Matrix.** | int mcm(vector<int>&a){int n=a.size();vector<vector<int>>dp(n,vector<int>(n));for(int l=2;l<n;l++)for(int i=1;i<n-l+1;i++){int j=i+l-1;dp[i][j]=1e9;for(int k=i;k<j;k++)dp[i][j]=min(dp[i][j],dp[i][k]+dp[k+1][j]+a[i-1]\*a[k]\*a[j]);}return dp[1][n-1];} | | **LIS** | int lis(vector<int>&a){int n=a.size();vector<int>dp(n,1);int mx=1;for(int i=1;i<n;i++)for(int j=0;j<i;j++)if(a[i]>a[j])dp[i]=max(dp[i],dp[j]+1);for(int x:dp)mx=max(mx,x);return mx;} | | |  |  | | --- | --- | | BFS | void bfs(int s, vector<int>adj[], int n){vector<bool>v(n);queue<int>q;q.push(s);v[s]=1;  while(!q.empty()){int u=q.front();q.pop();for(int x:adj[u])if(!v[x])q.push(x),v[x]=1;}} | | DFS | void dfs(int u, vector<int>adj[], vector<bool>&v){v[u]=1;for(int x:adj[u])if(!v[x])dfs(x,adj,v);} | | TOPO | void topo(int u, vector<int>adj[], vector<bool>&v, stack<int>&s){v[u]=1;for(int x:adj[u])if(!v[x])  topo(x,adj,v,s);s.push(u);} | | CYCLE | bool hasCycle(int u, vector<int>adj[], vector<int>&vis){vis[u]=1;for(int x:adj[u]){if(vis[x]==1)  return 1;if(!vis[x]&&hasCycle(x,adj,vis))return 1;}vis[u]=2;return 0;} | | KRUS | int find(int x,int p[]){return p[x]==x?x:p[x]=find(p[x],p);}  void kruskal(vector<tuple<int,int,int>>&e,int n){sort(e.begin(),e.end());int p[n];iota(p,p+n,0);for(auto &[w,u,v]:e)if(find(u,p)!=find(v,p)){p[find(u,p)]=find(v,p);cout<<u<<"-"<<v<<"\\n";}} | | PRIM | void prim(int n, vector<pair<int,int>>adj[]){vector<bool>v(n);priority\_queue<pair<int,int>,  vector<pair<int,int>>,greater<>>pq;pq.push({0,0});while(!pq.empty()){auto [w,u]=pq.top();pq.pop();  if(v[u])continue;v[u]=1;for(auto [x,c]:adj[u])if(!v[x])pq.push({c,x});}} | | DJIK | void dijkstra(int s, vector<pair<int,int>>adj[], int n){vector<int>d(n,1e9);d[s]=0;  priority\_queue<pair<int,int>,vector<pair<int,int>>,  greater<>>pq;pq.push({0,s});while(!pq.empty()){auto [w,u]=pq.top();pq.pop();if(w>d[u])  continue;for(auto [v,c]:adj[u])if(d[u]+c<d[v])d[v]=d[u]+c,pq.push({d[v],v});}for(int x:d)cout<<x<<" ";} | | BELL | void bellman(int s, vector<tuple<int,int,int>>&e, int n){vector<int>d(n,1e9);  d[s]=0;for(int i=0;i<n-1;i++)for(auto &[u,v,w]:e)if(d[u]!=1e9&&d[u]+w<d[v])  d[v]=d[u]+w;for(int x:d)cout<<x<<" ";} | | FLOY | void floyd(vector<vector<int>>&g, int n){for(int k=0;k<n;k++)for(int i=0;i<n;i++)for(int j=0;j<n;j++)if(g[i][k]<1e9&&g[k][j]<1e9)g[i][j]=min(g[i][j],g[i][k]+g[k][j]);for(int i=0;i<n;i++){for(int j=0;j<n;j++)cout<<(g[i][j]==1e9?"INF":to\_string(g[i][j]))<<" ";cout<<"\\n";}} | |
| #### \*\*Significance of Huffman Coding\*\*  Huffman Coding is a \*\*lossless data compression algorithm\*\* that assigns \*\*variable-length codes\*\* to characters based on their frequency. More frequent characters get \*\*shorter codes\*\*, while less frequent ones get \*\*longer codes\*\*, ensuring efficient compression. It is widely used in \*\*file compression formats\*\* like \*\*ZIP, GZIP, PNG, JPEG, and MP3\*\*, as well as in \*\*text transmission and data encoding\*\*.  struct Node {  char ch;  int freq;  Node \*left, \*right;  Node(char c, int f) : ch(c), freq(f), left(nullptr), right(nullptr) {}  };  struct Compare {  bool operator()(Node\* a, Node\* b) { return a->freq > b->freq; }  };  void printCodes(Node\* root, string code) {  if (!root) return;  if (root->ch != '$') cout << root->ch << ": " << code << endl;  printCodes(root->left, code + "0");  printCodes(root->right, code + "1");  }  void huffmanCoding(vector<char>& chars, vector<int>& freqs) {  priority\_queue<Node\*, vector<Node\*>, Compare> pq;  for (int i = 0; i < chars.size(); i++) pq.push(new Node(chars[i], freqs[i]));  while (pq.size() > 1) {  Node \*left = pq.top(); pq.pop();  Node \*right = pq.top(); pq.pop();  Node \*newNode = new Node('$', left->freq + right->freq);  newNode->left = left;  newNode->right = right;  pq.push(newNode);}  printCodes(pq.top(), "");}  #### \*\*How It Works\*\*  1. \*\*Build a Min-Heap\*\* of characters based on frequency.  2. \*\*Extract two lowest-frequency nodes\*\* and merge them into a new node.  3. \*\*Repeat until one node remains\*\*, forming the Huffman Tree.  void **quickSort**(vector<int>& arr, int low, int high) {  if (low >= high) return;  int pivot = arr[high], i = low - 1;  for (int j = low; j < high; j++)  if (arr[j] <= pivot) swap(arr[++i], arr[j]);  swap(arr[i + 1], arr[high]);  quickSort(arr, low, i);  quickSort(arr, i + 2, high);}  void **insertionSort**(vector<int>& arr) {  int n = arr.size();  for (int i = 1; i < n; i++) {  int key = arr[i], j = i - 1;  while (j >= 0 && arr[j] > key) arr[j + 1] = arr[j--];  arr[j + 1] = key;}}  void merge(vector<int>& arr, int l, int m, int r) {  vector<int> left(arr.begin() + l, arr.begin() + m + 1);  vector<int> right(arr.begin() + m + 1, arr.begin() + r + 1);  int i = 0, j = 0, k = l;  while (i < left.size() && j < right.size()) arr[k++] = (left[i] < right[j]) ? left[i++] : right[j++];  while (i < left.size()) arr[k++] = left[i++];  while (j < right.size()) arr[k++] = right[j++];}  voi**d mergeSort**(vector<int>& arr, int l, int r) {  if (l >= r) return;  int m = l + (r - l) / 2;  mergeSort(arr, l, m);mergeSort(arr, m + 1, r);merge(arr, l, m, r);}  void **heapify**(vector<int>& arr, int n, int i) {  int largest = i, left = 2 \* i + 1, right = 2 \* i + 2;  if (left < n && arr[left] > arr[largest]) largest = left;  if (right < n && arr[right] > arr[largest]) largest = right;  if (largest != i) { swap(arr[i], arr[largest]); heapify(arr, n, largest); }}  **void heapSort**(vector<int>& arr) {  int n = arr.size();  for (int i = n / 2 - 1; i >= 0; i--) heapify(arr, n, i);  for (int i = n - 1; i > 0; i--) { swap(arr[0], arr[i]); heapify(arr, i, 0); }  }  **MATRIX CHAIN MUL**  int matrixChainRec(vector<int>& p, int i, int j) {  if (i == j) return 0;  int minCost = INT\_MAX;  for (int k = i; k < j; k++) {  int cost = matrixChainRec(p, i, k) + matrixChainRec(p, k + 1, j) + p[i - 1] \* p[k] \* p[j];  minCost = min(minCost, cost); }return minCost; }  int matrixChainDP(vector<int>& p) {  int n = p.size();  vector<vector<int>> dp(n, vector<int>(n, 0));  for (int len = 2; len < n; len++) {  for (int i = 1; i < n - len + 1; i++) { int j = i + len - 1;  dp[i][j] = INT\_MAX;  for (int k = i; k < j; k++) {  int cost = dp[i][k] + dp[k + 1][j] + p[i - 1] \* p[k] \* p[j];  dp[i][j] = min(dp[i][j], cost); }}}  return dp[1][n - 1]; }  **0/1 KNAPSACK** int knapsackRec(int W, vector<int>& val, vector<int>& wt, int n) {  if (n == 0 || W == 0) return 0;  if (wt[n - 1] <= W)  return max(val[n - 1] + knapsackRec(W - wt[n - 1], val, wt, n - 1), knapsackRec(W, val, wt, n - 1));  return knapsackRec(W, val, wt, n - 1); }  int knapsackDP(int W, vector<int>& val, vector<int>& wt) {int n = val.size();  vector<vector<int>> dp(n + 1, vector<int>(W + 1, 0));  for (int i = 1; i <= n; i++) {  for (int j = 1; j <= W; j++) {  if (wt[i - 1] <= j) dp[i][j] = max(val[i - 1] + dp[i - 1][j - wt[i - 1]], dp[i - 1][j]);  else dp[i][j] = dp[i - 1][j]; }}return dp[n][W]; }  **TRAVELLING SALESMAN**  #define N 4#define INF INT\_MAX  int dist[N][N] = { {0, 10, 15, 20}, {10, 0, 35, 25}, {15, 35, 0, 30}, {20, 25, 30, 0} };  vector<vector<int>> dp(1 << N, vector<int>(N, -1));  int tsp(int mask, int pos) {  if (mask == (1 << N) - 1) return dist[pos][0];  if (dp[mask][pos] != -1) return dp[mask][pos];  int ans = INF; for (int city = 0; city < N; city++) {  if (!(mask & (1 << city))) {  int newAns = dist[pos][city] + tsp(mask | (1 << city), city);  ans = min(ans, newAns); }}  return dp[mask][pos] = ans; }  **LCS**  int lcsRec(string &s1, string &s2, int m, int n) {  if (m == 0 || n == 0) return 0;  if (s1[m - 1] == s2[n - 1]) return 1 + lcsRec(s1, s2, m - 1, n - 1);  return max(lcsRec(s1, s2, m, n - 1), lcsRec(s1, s2, m - 1, n));  }  int lcsDP(string &s1, string &s2) {  int m = s1.size(), n = s2.size();  vector<vector<int>> dp(m + 1, vector<int>(n + 1, 0));  for (int i = 1; i <= m; i++) {  for (int j = 1; j <= n; j++) {  if (s1[i - 1] == s2[j - 1]) dp[i][j] = dp[i - 1][j - 1] + 1;  else dp[i][j] = max(dp[i - 1][j], dp[i][j - 1]); }}  return dp[m][n]; }  **KMP**  void computeLPS(string pattern, vector<int> &lps) {  int len = 0, m = pattern.size();lps[0] = 0;  for (int i = 1; i < m; i++) {  while (len > 0 && pattern[i] != pattern[len]) len = lps[len - 1];  if (pattern[i] == pattern[len]) len++;  lps[i] = len;}}  void KMP(string text, string pattern) {  int n = text.size(), m = pattern.size();vector<int> lps(m);  computeLPS(pattern, lps);  int i = 0, j = 0;  while (i < n) {  if (text[i] == pattern[j]) { i++, j++; }  if (j == m) {  cout << "Pattern found at index " << i - j << endl; j = lps[j - 1];  } else if (i < n && text[i] != pattern[j]) {  j ? j = lps[j - 1] : i++;}}}  **ROBIN KARP**  #define d 256 void rabinKarp(string text, string pattern, int prime) {  int m = pattern.length(), n = text.length();  int pHash = 0, tHash = 0, h = 1;  for (int i = 0; i < m - 1; i++)h = (h \* d) % prime;  for (int i = 0; i < m; i++) {  pHash = (d \* pHash + pattern[i]) % prime;  tHash = (d \* tHash + text[i]) % prime;}  for (int i = 0; i <= n - m; i++) {  if (pHash == tHash && text.substr(i, m) == pattern)cout << "Pattern found at index " << i << endl;  if (i < n - m) {  tHash = (d \* (tHash - text[i] \* h) + text[i + m]) % prime;  if (tHash < 0) tHash += prime;}}} | **String Matching** is the process of finding occurrences of a pattern within a larger text. It is widely used in **search engines, DNA sequencing, text editing, plagiarism detection, and spam filtering**.**Types of String Matching Naïve Approach** – Check each substring one by one (**O(nm)** time complexity).**Rabin-Karp Algorithm** – Uses **rolling hash** for fast comparison (**O(n + m)** on average).**Knuth-Morris-Pratt (KMP) Algorithm** – Uses the **prefix function** to skip unnecessary comparisons (**O(n + m)** time complexity).**Boyer-Moore Algorithm** – Optimizes mismatches using **bad-character heuristic** (**O(n/m)** in best cases).  **Breadth-First Search (BFS)** and **Depth-First Search (DFS)** are two fundamental graph traversal algorithms with distinct approaches.  **Traversal Method**: BFS explores nodes **level by level**, while DFS explores **deep into one branch** before backtracking.  **Data Structure Used**: BFS uses a **queue (FIFO)**, whereas DFS uses a **stack (LIFO)** or recursion.  **Shortest Path**: BFS guarantees the **shortest path** in an unweighted graph, while DFS does not.  **Memory Usage**: BFS requires **more memory** as it stores all nodes at a level, whereas DFS is **memory-efficient** for deep graphs.  **Applications**: BFS is used in **shortest path finding, web crawling**, and **network broadcasting**, while DFS is useful for **cycle detection, topological sorting**, and **solving mazes**.  The **Rabin-Karp Algorithm** is a string matching technique that uses **hashing** to efficiently find a pattern within a larger text. Instead of checking each substring character by character, it compares **hash values**, making it faster in many cases.**How It Works**  Compute the **hash value** of the pattern. Compute the **hash value** of the first substring of the text (same length as the pattern). Compare the hash values: If they match, verify the actual characters to confirm a match. If they don’t match, slide the window one position and update the hash efficiently. Repeat until the entire text is scanned. **ExampleGiven:Text:** "geeksforgeeks"  Compute the hash of "geek".Compute the hash of the first four characters of "geeksforgeeks" ("geek").Since the hash values match, check the actual characters. Found a match at **index 0**. Slide the window and compute the hash for "eeks", "eksf", etc. Another match is found at **index 8**.  - \*\*Directed Graph\*\*: Edges have direction, allowing movement in one way only.  - \*\*DAG (Directed Acyclic Graph)\*\*: A directed graph with no cycles, used in scheduling and dependency resolution.  - \*\*MST (Minimum Spanning Tree)\*\*: Connects all nodes with the minimum edge cost in an undirected graph.  #### \*\*Algorithm Applications\*\*  - \*\*Dijkstra’s\*\* – Shortest path (GPS, network routing).  - \*\*Prim’s\*\* – Minimum spanning tree (network design).  - \*\*Kruskal’s\*\* – MST using edges (clustering, grid design).  - \*\*Bellman-Ford\*\* – Shortest paths with negative weights (currency exchange).  - \*\*Floyd-Warshall\*\* – All-pairs shortest paths (transportation planning).  - \*\*BFS\*\* – Level-wise traversal (web crawling, shortest path).  - \*\*DFS\*\* – Deep traversal (cycle detection, maze solving).  Let me know if you need further details! 🚀  You can also check related concepts [here](https://www.geeksforgeeks.org/graph-data-structure-and-algorithms/).  **P (Polynomial Time)** includes problems that can be solved efficiently using algorithms whose worst-case time complexity is polynomial in the input size. Sorting algorithms like Merge Sort and Quick Sort are examples of P problems because they run in **O(n log n)** or **O(n²)** time.**NP (Non-Deterministic Polynomial Time)** represents problems where verifying a given solution can be done in polynomial time, but finding the solution may require exponential time in the worst case. The **Subset Sum Problem**, where we check if a subset adds up to a given sum, is a classic NP problem because verifying an answer is straightforward, but finding the right subset among all possibilities can be hard.**NP-Complete problems** are those that belong to NP and are at least as difficult as any other NP problem. If a polynomial-time solution is found for any NP-Complete problem, then all NP problems would also be solvable in polynomial time. The **Travelling Salesman Problem (TSP)** is NP-Complete, as determining the shortest route visiting all cities exactly once is computationally complex, yet verifying a given solution is easy.**NP-Hard problems** are those that are at least as difficult as NP problems but may not belong to NP, meaning their solutions may not be verifiable in polynomial time. The Halting Problem, which asks whether a program will eventually stop or run indefinitely, is NP-Hard because it is undecidable. Unlike NP-Complete problems, solving an NP-Hard problem in polynomial time does not necessarily imply that all NP problems can be solved efficiently.  **Hashing** is a technique used to map data to a fixed-size value, typically for fastretrieval in data structures like hash tables. It converts input (keys) into an integer (hash value) using a hash function, which determines the index where the key-value pair is stored.  **Collision in Hashing** A collision occurs when two different keys produce the same hash value and get assigned to the same index in the hash table.  **Collision Handling Techniques**  Separate Chaining – Use a linked list at each index to store multiple elements that hash to the same index.  Open Addressing – Store all elements within the hash table itself and resolve collisions by finding alternative positions:  Linear Probing – Search sequentially for the next available slot.  Quadratic Probing – Use a quadratic function to find an open slot.  Double Hashing – Use a second hash function for collision resolution.  **1. Optimal Substructure** A problem exhibits optimal substructure if its optimal solution can be constructed from the optimal solutions of its subproblems. This allows us to break down a problem into smaller parts and solve them recursively or iteratively.Example (DP): The Shortest Path Problem follows optimal substructure because the shortest path from A → C can be built using the shortest path from A → B and B → C.  Example (Greedy): The Activity Selection Problem follows optimal substructure because selecting the earliest finishing activity ensures an optimal solution.  2. **Overlapping Subproblems** A problem has overlapping subproblems if the same subproblems are solved multiple times. DP optimizes such problems by storing results to avoid redundant computations.Example: Fibonacci Sequence exhibits overlapping subproblems because fib(n) = fib(n-1) + fib(n-2), meaning fib(n-1) and fib(n-2) are repeatedly computed.  3**. Greedy Choice PropertyA** problem has the greedy choice property if making a locally optimal choice at each step leads to a globally optimal solution.  Example: Huffman Coding follows the greedy choice property by always merging the two least frequent symbols first, leading to an optimal prefix code. |
| \*\*Heap Sort\*\* is a \*\*comparison-based sorting algorithm\*\* that uses the \*\*Binary Heap data structure\*\* to efficiently sort elements. It follows a two-step process: 1. \*\*Build a Max Heap\*\* – Convert the array into a \*\*Max Heap\*\*, where the largest element is always at the root. 2. \*\*Extract Elements\*\* – Swap the root (largest element) with the last element, reduce the heap size, and \*\*heapify\*\* to restore the heap property. Repeat until the array is sorted.\*\*How It Works (Step-byStep)\*\*1. \*\*Heap Construction\*\* – Convert the array into a \*\*Max Heap\*\* using the \*\*heapify\*\* function.2. \*\*Sorting  - Swap the root (largest element) with the last element. - Reduce heap size and \*\*heapify\*\* the new root.  - Repeat until all elements are sorted.  \*\* → `[90, 80, 70, 10, 40, 50, 30]`2. \*\*Extract Max (90)\*\* → Swap with last → `[30, 80, 70, 10, 40, 50, 90]`  3. \*\*Heapify\*\* → `[80, 40, 70, 10, 30, 50, 90]`  4. \*\*Repeat Until Sorted\*\* → `[10, 30, 40, 50, 70, 80, 90]`  ### \*\*Time Complexity\*\*  - \*\*Best/Average/Worst Case:\*\* \*\*O(n log n)\*\*  - \*\*Space Complexity:\*\* \*\*O(1)\*\* (In-place sorting) | **External sorting** is used when the dataset is too large to fit into memory, requiring chunks of data to be processed on disk before being merged into the final sorted sequence. For example, \*\*Merge Sort\*\* can be adapted for external sorting by dividing large files into smaller sorted parts and then merging them efficiently. This method is essential in \*\*database management and big data applications\*\* where handling massive datasets is necessary.  **In-place sorting** modifies the original array without using extra memory, making it efficient for memory-limited environments. **\*\*Quick Sort\*\*** is an example of in-place sorting, where elements are rearranged within the same array using a \*\*pivot-based partitioning technique\*\*. This reduces additional storage overhead while ensuring fast execution.  Online sorting processes data incrementally as elements arrive, making it suitable for real-time applications. \*\*Insertion Sort\*\* follows this approach by placing each new element into its correct position within the already sorted portion of the array. This allows continuous sorting without needing the entire dataset upfront, making it valuable in \*\*streaming data processing and dynamic systems\*\*.  Another important category is \*\*Stable Sorting\*\*, where equal elements retain their original order after sorting. \*\*Merge Sort\*\* and \*\*Bubble Sort\*\* are examples of stable sorting algorithms, ensuring consistency in sorting when dealing with \*\*identical values\*\*, which is useful in \*\*database operations and lexical sorting\*\*.  Insertion Sort – Best case **O(n)**, Average/Worst case **O(n²)**, Space **O(1)**. Selection Sort – Best/Average/Worst case **O(n²)**, Space **O(1)**. Bubble Sort – Best case **O(n)**, Average/Worst case **O(n²)**, Space **O(1)**. Merge Sort – Best/Average/Worst case **O(n log n)**, Space **O(n)**. Quick Sort – Best/Average case **O(n log n)**, Worst case **O(n²)**, Space **O(n)**. Randomized Quick Sort – Best/Average case **O(n log n)**, Worst case **O(n²)**, Space **O(n)**. Exponential Search – Best case **O(1)**, Worst case **O(log n)**, Space **O(1)**. Topological Sorting – Best/Average/Worst case **O(V + E)**, Space **O(V + E)**. Dijkstra’s Algorithm – Best case **O(E + V log V)**, Worst case **O(V²)**, Space **O(V)**. BFS – Best/Average/Worst case **O(V + E)**, Space **O(V + E)**. DFS – Best/Average/Worst case **O(V + E)**, Space **O(V + E)**. Prim’s Algorithm – Best case **O(E + V log V)**, Worst case **O(V²)**, Space **O(V)**. Kruskal’s Algorithm – Best/Average/Worst case **O(E log E)**, Space **O(V + E)**. 0/1 Knapsack – Best/Average/Worst case **O(N \* W)**, Space **O(N \* W)**. Fractional Knapsack – Best/Average/Worst case **O(N log N)**, Space **O(1)**. Travelling Salesman Problem (TSP) – Brute force **O(n!)**, Dynamic Programming **O(n² \* 2^n)**, Space **O(n \* 2^n)**. Huffman Coding – Best/Average/Worst case **O(n log n)**, Space **O(n)**. Bellman-Ford Algorithm – Best/Average/Worst case **O(VE)**, Space **O(V)**. Floyd-Warshall Algorithm – Best/Average/Worst case **O(V³)**, Space **O(V²)**.  **The \*\*0/1 Knapsack Problem\*\*** requires selecting items to maximize total value while ensuring the total weight does not exceed a given capacity. Unlike the \*\*Fractional Knapsack\*\*, items \*\*cannot be divided\*\*, making it an all-or-nothing choice.  \*\*Dynamic Programming Approach (Bottom-Up)\*\*  We use a \*\*DP table\*\* where:- `dp[i][w]` represents the \*\*maximum value possible\*\* using the first `i` items with weight limit `w`.  - The table is filled using the recurrence relation:  \[dp[i][w] = \max(dp[i-1][w], val[i] + dp[i-1][w - wt[i]])  This means:  - \*\*Exclude the item\*\* → Keep the previous best value. - \*\*Include the item (if weight allows)\*\* → Add item value to the best solution for the remaining capacity.  #### \*\*Steps\*\*1. \*\*Initialize the table\*\* with `0` when either no items are available or weight capacity is `0`. 2. \*\*Iterate through items\*\* and update values using the recurrence relation. 3. \*\*The final value in `dp[n][W]` gives the maximum profit achievable.\*\*  #### \*\*Example\*\*  Given:- Items `{A, B, C}`- Profits `{60, 100, 120}`- Weights `{10, 20, 30}`- Knapsack Capacity `W = 50`  The optimal selection is `{B, C}`, yielding a \*\*maximum profit of 220\*\*.This \*\*bottom-up DP approach\*\* ensures \*\*efficient computation in O(N \* W)\*\* time complexity.  The **0/1 Knapsack Problem** and **Fractional Knapsack Problem** are both optimization problems, but they differ in their approach and solution methods.  **0/1 Knapsack**: Items **cannot** be divided; you either take the whole item or leave it. It is solved using **dynamic programming** or **backtracking**. The time complexity is **O(N \* W)**, where **N** is the number of items and **W** is the knapsack capacity.  **Fractional Knapsack**: Items **can** be divided, meaning you can take a fraction of an item to maximize value. It is solved using a **greedy approach**, sorting items by **value/weight ratio** and picking the highest first. The time complexity is **O(N log N)** due to sorting.  **Greedy vs DP**: The **greedy approach** works optimally for **Fractional Knapsack**, but fails for **0/1 Knapsack**, where **dynamic programming** is required for an optimal solution. |