Algorithms & Data Structures

# Complexity & Running Time

## Notations

|  | Asymptotically tight bound:  IN ORDER |
| --- | --- |
|  | Asymptotically upper bound:  NO MORE THAN |
|  | Asymptotically lower bound:  AT LEAST |

## Complexity



| P | Problem is solvable in polynomial time. |
| --- | --- |
| NP | Problem is solvable in polynomial time via “lucky” algorithm (Uses model of computation which is no-deterministic). |
| EXP | Problem is solvable in exponential time. |
| R | Problem is solvable in finite time (“recursion”). |

## Running Time Estimation

|  | |
| --- | --- |
|  |  |
|  |  |
|  |  |

# Data Structures

## Overview

| **Name** | **Description** |
| --- | --- |
| Array / Matrix | An array is a systematic arrangement of objects, usually in rows and columns |
| Stack | LIFO (array based) |
| Queue | FIFO (array/heap based) |
| Linked List   * Singly * Double * Circular |  |
| Heap   * Max * Min | A heap is a specialized tree-based data structure that satisfies the heap property. Used for creating priority queue. |
| Set | A set is an abstract data structure that can store certain values, without any particular order and no repeated values. Can be constructed using hash and tree. |
| Disjoint-Set | A disjoint-set data structure is a data structure that keeps track of a set of elements partitioned into a number of disjoint (non-overlapping) subsets. Used in solving spanning tree problems. |
| Tree |  |
| Hash Map | A hash map is a data structure used to implement an associative array, a structure that can map keys to values. |
| Graph |  |

## Disjoint-Set

, where is identified by representative

Operations

| **Name** | **Description** |
| --- | --- |
| Make() | Create a new set with representative |
| Union() | Unites and where and . A new representative is appointed for the created set. |
| Find() | Returns representative of the set where |

## Heap

| Generic Properties |
| --- |
|  |
| Max-Heap Properties |
|  |
| Min-Heap Properties |
|  |
|  |

## Tree

### Binary-tree

|  | Unbalanced | when balanced  when unbalanced |
| --- | --- | --- |

### AVL-tree

|  | Balanced  Uses node height;  Uses rotation for balance; |  |
| --- | --- | --- |

### Red and Black-tree

| h = number of black nodes  h – the same for all branches | Balanced  Uses red & black properties;  Uses rotation & recoloring for balance; |  |
| --- | --- | --- |

### Rotation (for AVL & RB trees)

| Rotation  Right | Before | After |
| --- | --- | --- |

| Rotation  Left | Before | After |
| --- | --- | --- |

### B-tree

Model



Tree



| Search |  |
| --- | --- |
| Sort |  |
| Permuting |  |
| Buffer Tree   * The dynamic version of the sort; * Insert & Delete via buffer; * Delay Batch Update | amortized |
| Find Min |  |
| Insert and Delete operations are performed similar to binary search trees with split and shrink nodes if they are full. The efficient split and shrink can be performed via control of “load factor” | |

### Cache Oblivious B-tree

| Running Time |  | Search, Insert & Delete operations; |
| --- | --- | --- |

## Hash Tables

| Definition |  |
| --- | --- |
| Load factor |  |

| Function types | |
| --- | --- |
| division |  |
| multiplication |  |
| universal |  |
| perfect | 2-levels|need to know all keys before hashing; |

| m | Hash table size | p | Large prime number |
| --- | --- | --- | --- |
| n | Number of keys | a | Constant |
| k | Key-value | b | Constant |
| w | Word |  |  |

| Running Time | amortized | Supports Insert & Delete operations; |
| --- | --- | --- |
| The “amortized” time is an average time overall operations | |

## Graph

|  | V - set of vertices |
| --- | --- |
| E - set of edges |
| W – set of weights |

|  | **Edge** | **Graph** |
| --- | --- | --- |
|  | unordered | Undirected |
|  | ordered | Directed |

Adjacency:

|  |  |
| --- | --- |

| Adjacency-List |  | Sparse |
| --- | --- | --- |
| Object-oriented |
| Adjacency-Matrix |  | Dense |

# Traversal

## Tree

### Depth-First Search

| * Pre-order : **4**-**2**-1-3-**6**-5-7 * In-order: 1-**2**-3-**4**-5-**6**-7 * Post-order: 1-3-**2**-5-7-**6**-**4** |  |
| --- | --- |

### Breadth-First Search

| * Level-Order: **4**-**2**-**6**-1-3-5-7 |  |
| --- | --- |

## Graph

### Depth-First Search

Gives the SP from “s” to “v”



| Running Time |  | Undirected Graph |
| --- | --- | --- |
|  | Directed Graph |

### Breadth-First Search

Explores the whole graph:

* Find cycles;
* Topology sort

Uses exploration method;



| Edges Classification | |
| --- | --- |
| TE – tree edge | BE – backward edge |
| FE – forward edge | CE – cross edge |

| Running Time |  | Undirected Graph |
| --- | --- | --- |
|  | Directed Graph |

| Find  Cycles | If (**BE** exists) then G has **cycles**; | |
| --- | --- | --- |
| Topology Sort | Input | Directed Acyclic Graph (DAG) is a directed graph with no directed cycles (can’t have negative cycles)  (e.g. Dressing problem) |
| Run | DFS |
| Output |  |
| Comment | Good for scheduling problems |

# Sort

| Comparison | | | | |
| --- | --- | --- | --- | --- |
| Insert/Bubble | |  | |  |
| Quick | |  | |
| Merge | |  |
| Heap | |
| B-tree | |
| Non-Comparison | | | | |
| Counting | *n* - number of elements;  *k*-number of keys | |  | |
| Radix | *d -* number of digits  *n* - number of elements;  k-possible values; | |  | |
| Bucket | *n* - number of elements;  *k*-number of buckets | |  | |

## Algorithms

### Shortest Path

| Dijkstra | Rate: | [+] edges only |
| --- | --- | --- |
| Bellmen-Ford | Rate: | [+/-] edges |

Operation Relax (u,v,w)

|  | The path from “s” to “v” |
| --- | --- |
|  | The length of the current SP from “s” to “v” |
|  | The length of a SP from “s” to “v” |
|  | The predecessor of “v” in the SP from “s” to “v” |

| **if** () {  ;  ;  } | Relax edge |
| --- | --- |

Note:

#### Dijkstra Algorithm (G,W,s)

| **while**  ;  ;  **foreach** (){  ;  }  } | Greedy!  Vertices in the priority “Q” need to be processed.  When vertices are processed they are moved in S. |
| --- | --- |

| Example of Dijkstra | | |
| --- | --- | --- |
| **{}** |  |  |
| {A} |  |
| {A,C} |  |
| {A,C,E} |  |
| {A,C,E,D} |  |

#### Bellmen-Ford Algorithm (G,W,s)

| **for**  **foreach** (){  ;  }  }  **foreach** (){  **if** () {  Report that negative cycles exists  }  } | If you ask for anything more algorithm moves to an exponential level of complexity (for example simple-shortest path) |
| --- | --- |

| Example of Bellmen-Ford | |
| --- | --- |
|  |  |
|  |
|  |
|  |
|  |
|  |

### (Min/Max) Spanning Tree

#### Kruskal

| **Complexity:**  **Note**: *uses disjoin set* |
| --- |
| **public** G execute(G in)  G out;  DS d <- **make**(V[in]);  **for**(e: **sort**(E[in])){  **if** (d.**find**(e.u) != d.**find**(e.v)) {  out <- (e.u, e.v, e.weight);  d <- **union**(e.u, e.v);  }  }  **return** out; |
| **Example:** |
|  |

#### Prim

| **Complexity:**  **Note**: *uses priority queue* |
| --- |
| **public** G execute(G in)  G out;  PQ q <- E[in(some vertex)]  **while**(!q.**empty**()){  e <- q.**poll**();  **if** (V[out(e.v)] == null) {  out <- (e.u, e.v, e.weight);  q <- E[in(e.v)]  }  }  **return** out; |
| **Example** |
|  |

## Dynamic Programming

### Solution

| **Generic Approach (5 “easy” Steps)** | **Shortest path** *for DAGs* | **Shortest path** **for all** |
| --- | --- | --- |
| 1. Define sub-problems (count # of sub=problems) 2. Guess part of solution ( count the # of choices for guess) 3. Relate sub-problem solutions (relate the time for sub-problem) 4. Recursion + Memorization || build DP table bottom-up (check that sub-problem recursion is acyclic) 5. Solve the original problem (make sure that problem you trying to solve is solved, sometime it requires extra time) | Note:   * Sub-problem dependencies should be acyclic; * Treating the recursion call as | Split cyclic graphs into layers:  - weight of the shortest path that uses edges  Note:   * Number of sub-problems |

### Knapsack example

| **Knapsack Problem** | | |
| --- | --- | --- |
| **Input:** | Item[] items = new Item[] { new Item("shaver", 2, 8),  new Item("gel", 4, 4), new Item("sleeping-bag", 7, 5),  new Item("bottle", 3, 6), new Item("knife", 1, 7),  new Item("light", 1, 5) }; | |
|  | **Recursive Calls** | **Recursive Calls + Memorization** |
| **Code:** | public List<Item> **REC**(List<Item> items, int SIZE) {  List<Item> taken = new ArrayList<Item>();  if (size(items) <= SIZE) {  taken = items;  } else {  for (Item item : items) {  if (item.size <= SIZE) {  List<Item> choice = new ArrayList<Item>();  choice.add(item);  choice.addAll(**REC**(exclude(item, items), SIZE - item.size));  taken = max(taken, choice);  }  }  }  return taken;  } | public List<Item> **DP**(Map<Set<Item>, List<Item>> mem, List<Item> items,  int SIZE) {  Set<Item> key = new HashSet<>(items);  List<Item> taken = new ArrayList<Item>();  if (mem.containsKey(key)) {  taken = mem.get(key);  } else {  if (size(items) <= SIZE) {  taken = items;  } else {  for (Item item : items) {  if (item.size <= SIZE) {  List<Item> choice = new ArrayList<Item>();  choice.add(item);  choice.addAll(**DP**(mem, exclude(item, items), SIZE-item.size));  taken = max(taken, choice);  }  }  }  mem.put(key, taken);  }  return taken;  } |
| **Example:** |  | |