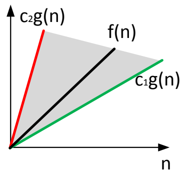
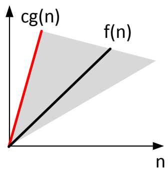
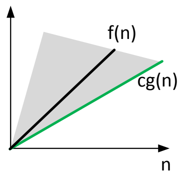
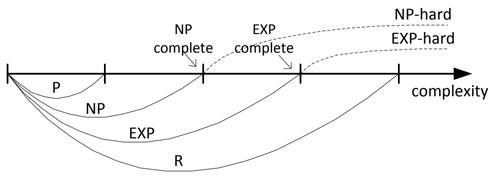
Algorithms & Data Structures

Complexity & Running Time

Notations

	Asymptotically tight bound: $f(n) = \mathbf{\Theta}(g(n))$ IN ORDER $c_1 g(n) \leq f(n) \leq c_2 g(n)$
	Asymptotically upper bound: $f(n) = \mathbf{O}(g(n))$ NO MORE THAN $f(n) \leq cg(n)$
	Asymptotically lower bound: $f(n) = \mathbf{\Omega}(g(n))$ AT LEAST $f(n) \geq cg(n)$

Complexity



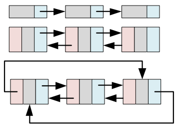
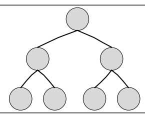
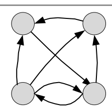
P	Problems solvable in polynomial time.
NP	Problems solvable in polynomial time via "lucky" algorithm (Uses model of computation which is no-deterministic).
EXP	Problem solvable in exponential time.
R	Problem solvable in finite time ("recursion").

Running Time Estimation

$T(n) = aT\left(\frac{n}{b}\right) + f(n)$	
$f(n) = O(n^{\log_b a - \varepsilon})$	$T(n) = O(n^{\log_b a})$
$f(n) = \Theta(n^{\log_b a})$	$T(n) = \Theta(n^{\log_b a} \log n)$
$f(n) = \Omega(n^{\log_b a + \varepsilon})$	$T(n) = \Theta(f(n))$

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 problem.
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

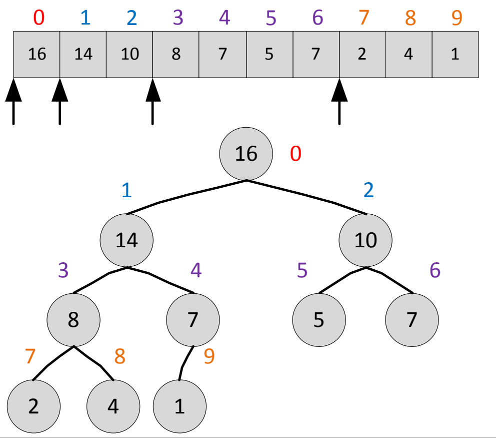
$S = \{S_1, S_2, \ldots, S_K\}$, where S_i is identified by representative x_i

Operations

Name	Description
Make(x)	Create a new set S_x with representative x
Union(x, y)	Unites S_x and S_x where $x \in S_x$ and $y \in S_y$ .A new representative is appointed for the created set.
Find(x)	Returns representative of the set S_i where $x \in S_i$

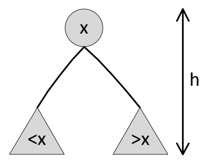
Heap

Generic Properties
$Root() \rightarrow 1$ $Parent(i) \rightarrow i/2$ $Left(i) \rightarrow 2i$ $Right(i) \rightarrow 2i+1$
Max-Heap Properties
$key[i] \geq \begin{cases} key[Left(i)] \\ key[Right(i)] \end{cases}$

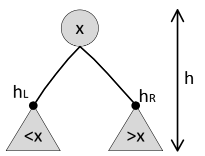
Min-Heap Properties
$key[i] \le \begin{cases} key[Left(i)] \\ key[Right(i)] \end{cases}$


Tree

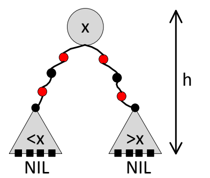
Binary-tree

	Unbalanced	when balanced $\boldsymbol{O}(\boldsymbol{log}(n))$ when unbalanced $\boldsymbol{O}(n^2)$

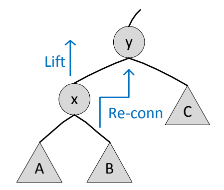
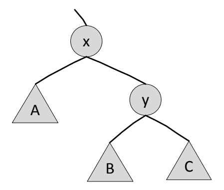
AVL-tree

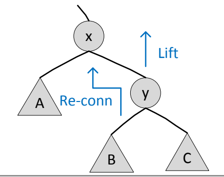
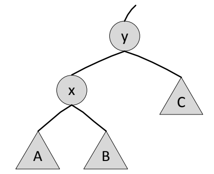
 $\lvert h_l - h_r \rvert \le 1$	Balanced Uses node height; Uses rotation for balance;	$\boldsymbol{O}(\boldsymbol{log}(n))$

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;	$\boldsymbol{O}(\boldsymbol{log}(n))$

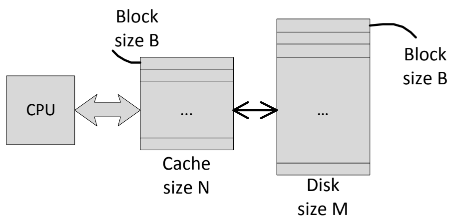
Rotation (for AVL & RB trees)

	Before	After
Rotation Right		

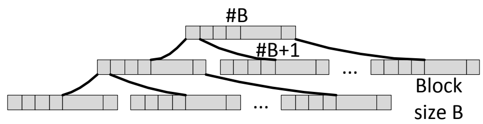
	Before	After
Rotation Left		

B-tree

Model



Tree



Search	$\boldsymbol{O}(\boldsymbol{log}_{B+1}(N))$
Sort	$\boldsymbol{O}\left(\frac{N}{B}\boldsymbol{log}_{M/B}\left(\frac{N}{B}\right)\right)$
Permuting	$\boldsymbol{O}\left(\boldsymbol{min}\left\{N, \frac{N}{B}\boldsymbol{log}_{M/B}\left(\frac{N}{B}\right)\right\}\right)$
Buffer Tree • Dynamic version of sort; • Insert & Delete via buffer; • Delay Batch Update	$\boldsymbol{O}\left(\frac{1}{B}\boldsymbol{log}_{M/B}\left(\frac{N}{B}\right)\right)$ amortized
Find Min	$\boldsymbol{O}(0)$
Insert and Delete operations are performed similar to binary search tree 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	$\boldsymbol{O}(\boldsymbol{log}_{\boldsymbol{B}}(N))$	Search, Insert & Delete operations;

Hash Tables

Definition	$h: U \to \{0,1,\dots,m-1\}$
Load factor	$\alpha = n/m$

Function types	
division	$h(k) = n \ \boldsymbol{mod} \ m$
multiplication	$h(k) = [(ak) \ \boldsymbol{mod} \ 2^w] \gg (w-r)$
universal	$h(k) = [(ak+b) \ \boldsymbol{mod} \ p] \ \boldsymbol{mod} \ m$
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		

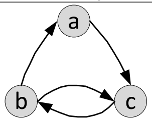
Running Time	$O(1)$ amortized	Supports Insert & Delete operations;
	The "amortized" time is an average time over all operations	

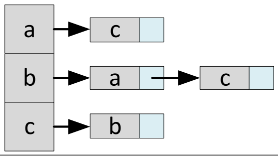
Graph

$G = (V, E, W)$	V - set of vertices
	E - set of edges
	W – set of weights

	Edge	Graph
$e \in E: e = \{u, v\}$	unordered	Undirected
$e \in E: e = (u, v)$	ordered	Directed

Adjacency: $Adj[u] = \{v \in V \mid (u, v) \in E\}$

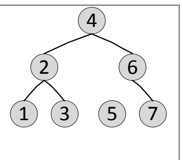
	$Adj[a] = \{c\}$ $Adj[b] = \{a, c\}$ $Adj[c] = \{b\}$

Adjacency-List 	$\lvert E\rvert \ll \lvert V^2\rvert$	Sparse
Object Oriented		
Adjacency-Matrix	$\lvert E\rvert \approx \lvert V^2\rvert$	Dense

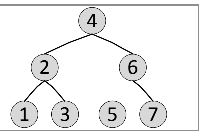
Traversal

Tree

Depth-First Search

• Pre-order ($R \longleftrightarrow$): **4**-**2**-1-3-**6**-5-7 • In-order($\leftarrow R \rightarrow$): 1-**2**-3-**4**-5-**6**-7 • Post-order($\longleftrightarrow R$): 1-3-**2**-5-7-**6**-**4**	

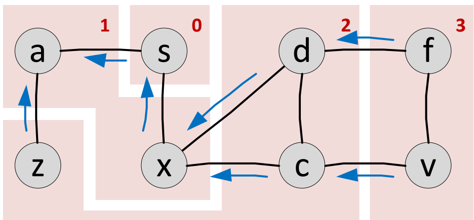
Breadth-First Search

• Level-Order(↓→): **4**-**2**-**6**-1-3-5-7	

Graph

Depth-First Search

Gives the SP from "s" to "v"



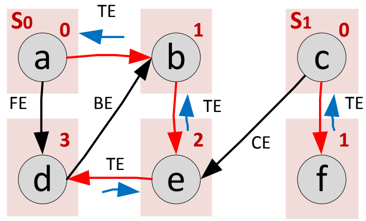
Running Time	$O(2\lvert E\rvert)$	Undirected Graph
	$O(\lvert E\rvert)$	Directed Graph

Breadth-First Search

Explores the whole graph:

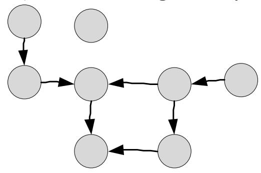
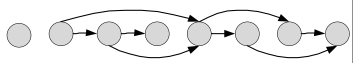
- Find cycles;
- Topology sort

Uses exploration method;



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

Running Time	$O(2\lvert E\rvert)$	Undirected Graph
	$O(\lvert E\rvert)$	Directed Graph

Find Cycle	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	$O(n^2)$	$O(n^2)$
Quick	$O(nlog(n))$	
Merge		$O(nlog(n))$
Heap		
B-tree		
Non-Comparison		
Counting	$O(n+k)$ *n* - number of elements; *k*-number of keys	$O(n+k)$
Radix	$O(d(n+k))$ *d* - number of digits; *n* - number of elements; k-possible values;	$O(d(n+k))$
Bucket	$O(n+k)$ *n* - number of elements; *k*-number of buckets	$O(n^2)$

Algorithms

Shortest Path

Dijkstra	$O(Vlog(V)+E)$ Rate: $O(V^2)$	[+] edges only
Bellmen-Ford	$O(VE)$ Rate: $O(V^3)$	[+/-] edges

Operation Relax (u,v,w)

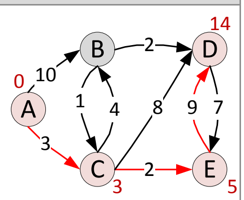
$s \to v$	The path from “s” to “v”
$d[v]$	The length of the current SP from “s” to “v”
$\delta(s,v)$	The length of a SP from “s” to “v”
$\pi[v]$	The predecessor of “v” in the SP from “s” to “v”

if $(d[v] > d[u] + w(u,v))$ {
$d[v] = d[u] + w(u,v);$
$\pi[v] \leftarrow u;$
}
	Relax edge

Note: $E = O(V^2)$

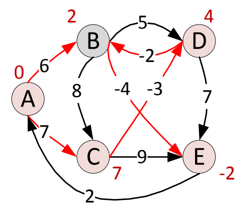
Dijkstra Algorithm (G,W,s)

$d[s] = 0; S \leftarrow \emptyset; Q \leftarrow V[G];$
while $(Q \neq \emptyset)${
$u \leftarrow$ **extract_min**$(Q);$
$S \leftarrow S \cup \{u\};$
foreach $(v \in Adj[u])${
relax$(u,v,w);$
}
}
	Greedy!
	Vertices in the priority “Q” need to be processed.
	
	When vertices processed they are moved in S.

Example of Dijkstra		
{}	$\{A,B,C,D,E\}$	
{A}	$\{0,\infty,\infty,\infty,\infty\}$	
{A,C}	$\{-,\infty,3,\infty,\infty\}$	
{A,C,E}	$\{-,7,-,11,5\}$	
{A,C,E,D}	$\{-,\infty,-,-,14\}$	

Bellmen-Ford Algorithm (G,W,s)

$d[s] \leftarrow 0; d[v] \leftarrow \infty; \pi[v] = \emptyset;$
for $(i = 1..\|V\|-1)${
foreach $(edge(u,v) \in E)${
relax$(u,v,w);$
}
}
foreach $(edge(u,v) \in E)${
if $(d[v] > d[u] + w(u,v))$ {
Report that negative cycles exists
}
}
	If you ask for anything more algorithm moves to exponential level of complexity (for example: simple-shortest path)

Example of Bellmen-Ford	
$\{A,B,C,D,E\}$	
$\{0,\infty,\infty,\infty,\infty\}$	
$\{0,6,\infty,7,\infty\}$	
$\{0,6,4,7,2\}$	
$\{0,2,4,7,2\}$	
$\{0,2,4,7,-2\}$	

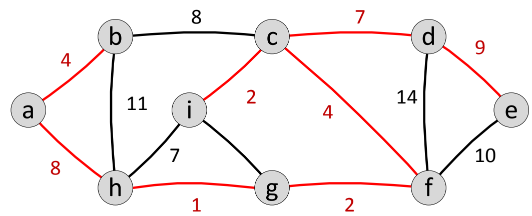
(Min/Max) Spanning Tree

Kruskal

Complexity: $O(Elog(V))$
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:

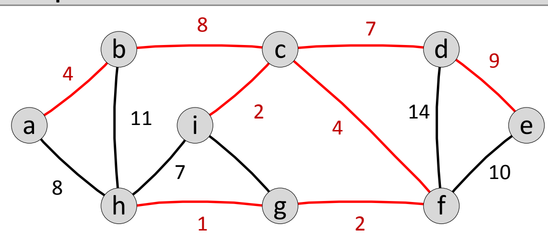


Prim

Complexity: $O(Elog(V))$
Note: *uses priority queue*

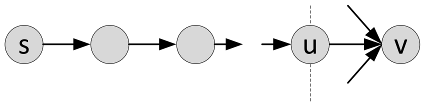
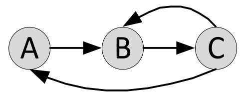
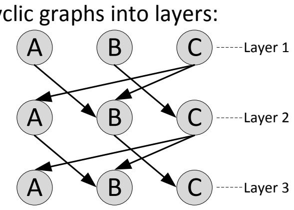
```
public G execute(G in)
    G out;

    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 $\delta(s,v) \forall v$ *for DAGs* $\boldsymbol{O}(V+E)$	Shortest path $\delta_k(s,v) \forall v$ for all $\boldsymbol{O}(VE)$
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)	 $\delta(s,v) = min_{(u,v)\in E}\{\delta(s,u) + w(u,v)\}$ Note: • Sub-problem dependencies should be acyclic; • $time_{DP} = \#_{sub-prob.} * time_{sub-prob.}$ • Treating the recursion call as $\boldsymbol{O}(1)$	 Split cyclic graphs into layers:  $\delta_k(s,v) = min_{(u,v)\in E}\{\delta_{k-1}(s,u) + w(u,v)\}$ $\delta_k(s,v)$- weight of the shortest path $s \to v$ that uses $\leq k$ edges Note: • Number of sub-problems V^2

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) };
```

Code:

Recursive Calls

```
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;
}
```

Recursive Calls + Memorization

```
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: