f(n)∈o(g(n))	F dominated by g	Strictly below	<
f(n)∈O(g(n))	B. from above	Upper bound	<=
f(n)∈Θ(g(n))	B. both sides	Equal to	=
f(n)∈Ω(g(n))	B. from below	Lower bound	>=
f(n)∈ω(g(n))	F dominates g	Strictly bound	>

Def: exists c and n_0 so that for every c>0, n_0>0, 0 ●g(n)●f(n) for all n_0>n.

Merge-sort: def merge(A,p,r): if p < r: q=(p+r)/2; merge(A,p,q); merge(A,q+1,r); combine(A,p,q,r)
def combine(A,B): //compare each element at position i, then append the smallest one to the result. O(nlogn)
Quicksort: def quick(A,p,r): if p < r: q = partition(A,p,r); quick(A,p,q-1); quick(A,q+1,r);
def partition(A,p,q): x = A[p]; i = p; for j in range(p+1,q): if A[j] <= x: i+=1; A[i],A[j] = A[j],A[i]; endfor; A[p],A[i] = A[i],A[p]; return i;
wc: O(n^2) bc: O(nlnn)
Lists Array: i,s,d = O(n); a[i],append,pop last=O(1)
LLIST:each el. has a pointer to the next. Dlink has a pointer to the next and prev. elements. I,s,d: a[1]=O(1),s:a[n]=O(1),i,d:a[n]=O(n), i,s,d:a[k]=O(n) (unless we have a pointer). DLINK: i,s,d: a[n]=O(1), except search (O(n)).
XOR llist: dlinked list, where pointers to next and prev are XOR-ed together.
Binary search: compare middle of list to value. Search only appropriate sublist (left or right), until found: O(logn)
Sorting lower bound O(nlogn) using comparisons.
Count sort: O(n)

```
for each input item x:
    increment Count[key(x)]
total = 0
for i = 0, 1, ... k:
    oldCount = Count[i]
    Count[i] = total
    total = total + oldCount
# allocate an output array Output[0..n-1] ;
for each input item x:
    Output[Count[key(x)]] = x
    increment Count[key(x)]
return Output
```

Skip list: structure of elements, where some element can be randomly elevated to the next „level", Upper levels cut down on search time: O(logn) for all cases.
Trees and heaps Binary tree: 0-2 children. Perfect: all subtrees filled at equal height. Complete: completely filled at every level (e. last) from left to right. Orders: pre – walk first (root first, then children), post – walk after (children first), in (walk to bottom, then move up).
BST: keys ordered: left < root <= right. Operations depend on the height of the tree. O(h)
AVL: heights of subtrees differ by at most 1.
RB trees: each node is R or B. Root is B. Every R node has B children. All leaves are B (NIL). Every path from a node to a descendant leaf has same number of B nodes. O(lgn). Rebalancing O(logn).
B-tree: every node has at most m children (m order) and at least m/2 children. Root has at least 2, unless its a leaf. All leaves at the same level and carry info. Every none-leaf with k children has k-1 keys.
k-dimensional trees: trees that section the available data into nodes by dividing the data by an attribute by level. Ie. 2d kd tree of a number of 2d points would have the root node as the median of all the nodes by x. Its children would be the 2 median nodes by y on both sides of the root.
Binary heap: insert/delete: O(logn). Sort: O(nlogn). Binary tree that has higher valued nodes at the top. Complete.
Interval trees/search:
Union find: data structure for disjoint sets. operations: make (make set), union(x,y) unite sets containing x and y, find(x) find a representative set that has x.

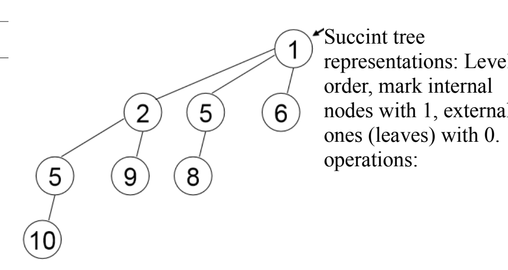
Function make-set(x)
$p[x] \leftarrow x$
$rank[x] \leftarrow 0$

Function find(x)
if $p[x] \neq x$ **then**
  $p[x] \leftarrow$ find($p[x]$)
return $p[x]$

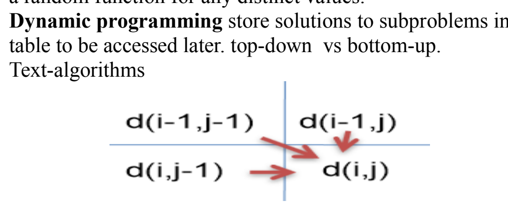
Function union(x, y)
link(find(x), find(y))

Function link(x, y)
if $rank[x] > rank[y]$ **then**
  $p[y] \leftarrow x$
else
  $p[x] \leftarrow y$
  if $rank[x] = rank[y]$ **then**
    $rank[y] \leftarrow rank[y] + 1$

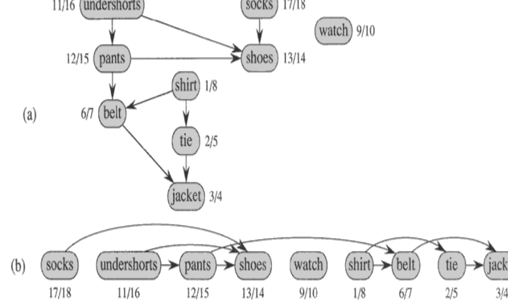
Binominal tree: recursively defined tree that consists of 2 binomial trees that are linked together: the root of one is the leftmost child of the other.
Binominal heap: A collection of binomial trees with at most one of every rank.



Succint tree representations: Level order, mark internal nodes with 1, external ones (leaves) with 0. operations:
leftchild=2x,rc=2x+1,parent=x/2,rank=count 1s, select=pos of 1. 2n+1 bits to store.
Parentheses repr: pre-order traversal.
Hashing: Collisions (only m to m). Resolve by chaining (each element has pointer to next), probing (use some incremental variable in the hash function). Probing strategies: linear (h(k)+i), quadratic (h(k)+i^2), double hashing (h1(k)+ih2(k)).
Another problem: an adversary can increase the overall speed of a hash table exponentially by picking, fe, all keys that map to slot X.
2 views on hashing: h is fixed, ks are random or ks are fixed and hs are chosen randomly. Both assume that when k1!=k2, then the probability that they map to the same bin is at most 1/m. A universal hashing function is a function that acts like a random function for any distinct values.
Dynamic programming store solutions to subproblems in a table to be accessed later. top-down vs bottom-up.
Text-algorithms



Graphs directed vs undirected, cyclical, clique, simple, sparse vs. dense. BFS (view all connected by "level", queue), DFS, randomized search.
A topological sort of a directed acyclic graph (DAG) is a linear order of all its vertices such that if G contains an edge (u, v), then u appears before v in the ordering.

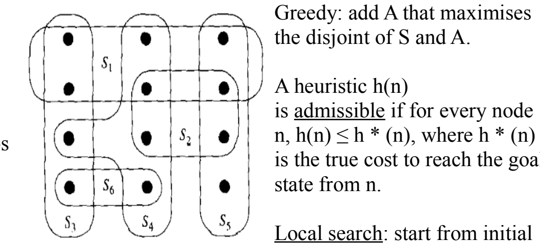


Gis strongly connected if every pair (u, v) of vertices in G is reachable from one another.
A minimum spanning tree of a weighted graph G is a tree that has the same set of vertices as G whose edges sum to a minimum weight.
Prim's algorithm: greedy algorithm that starts from a node and grows the mst one node at a time by selecting the node that is connected to the tree by an edge with minimal weight.
Kruskal's algorithm: Adds a the smallest edge to the tree that does not form a cycle.
Relaxing an edge (u, v) consists of testing whether we can improve the shortest path found so far by going through u and, if so, update d[v] and π[v] .
Bellman-Ford: Just repeatedly relax all edges.

DIJKSTRA(G, w, s)
1 INITIALIZE-SINGLE-SOURCE(G, s)
2 $S \leftarrow \emptyset$
3 $Q \leftarrow V[G]$
4 **while** $Q \neq \emptyset$
5   **do** $u \leftarrow$ EXTRACT-MIN(Q)
6     $S \leftarrow S \cup \{u\}$
7   **for each** vertex $v \in Adj[u]$
8   **do** RELAX(u, v, w)

Note: relax requires updating of min values in Q

1. Assign dist. values. 0 for start, inf for others.
2. Set all n unvisited and inital current. Create set of all n w/o init.
3. For cur, calc distances of neighbors (if A has dist 6 and A-B is l2, then dist to B is 8).
4. After all n, remove current as visited.
5. Stop condition.
6. select smallest dist n and set it as cur. goto3
Matrix repr. 1 if edge between nodes, 0 if not (ltrnttb).
Transitive closure of a digraph G is a graph G' with same verMces, and edge between any u and v from G if there is a path from u to v in G (G*G*G).
Set cover: Find the minimal number of sets whose union comprises the Universe.



Greedy: add A that maximises the disjoint of S and A.

A heuristic h(n) is admissible if for every node n, h(n) ≤ h * (n), where h * (n) is the true cost to reach the goal state from n.

Local search: start from initial position, iteratively move from current position to neighboring position, use evaluation function for guidance.
Hill-climb: select best state by changing a single element in the search space. Gets stuck on local optimums.
Random Search (Blind Guessing): In each step, randomly select one element of the search space.
(Uninformed) RandomWalk: In each step, randomly select one of the neighbouring positions of the search space and move there.
RR-HC: randomly step into a different direction.
Sannealing: Select a neighbor at random.
If better than current state go there.
Otherwise, go there with some probability.
Probability goes down with time (similar to temperature cooling).
Genetic algorithms:
initialize population;
evaluate population;
while TerminationCriteriaNotSatisfied
{select parents for reproduction;
perform recombination and mutation;
evaluate population;}
Uses ideas like crossover (take 2 or more solutions and splice parts of them together using some criteria) and mutation (reorder, flip bit, etc).
ACO uses the idea of pheromones. Starts randomly, frequent paths get covered with pheromones while less walked paths have theirs evaporate.
Clustering group similarity: min,max,average,centroid distances
k-nn: 1. Partition objects into k nonempty subsets
2. Compute seed points as the centroids of the clusters of the current partition. The centroid is the center (mean point) of the cluster.
3. Assign each object to the cluster with the nearest seed point.
4. Go back to Step 2, stop when no more new assignments.
Amdahl's law:

$$S(N) = \frac{1}{(1-P) + \frac{P}{N}}.$$

N – processors
P – % of program that can be made parallel.