ALGORITHMS

JACOB REINHOLD

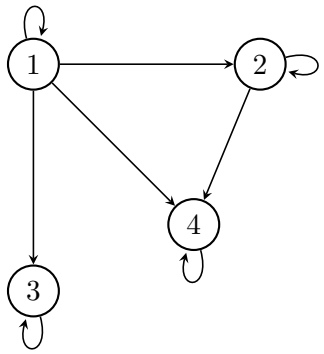
Contents

1. Discrete Math Review

Definition 1.1. A binary relation R between sets X and Y is specified by its graph G, which is a subset of the Cartesian product $X \times Y$.

We can visualize a binary relation R over a set A as a graph, where the nodes are the elements of A, and for $x, y \in A$, there is an edge from x to y if and only if $x \, R \, y$.

Example 1.2. The relation $a \mid b$ over the set $\{1, 2, 3, 4\}$ looks like this:



Definition 1.3. An equivalence relation is a relation that is reflexive, symmetric, and transitive.

Definition 1.4. Let R be an equivalence relation on A. The equivalence class of an element $a \in A$ is defined as the set $[a] = \{x \in A \mid a \sim x\}$.

Definition 1.5. Partial order is a relation that is reflexive, antisymmetric, and transitive, i.e. $\leq$.

Definition 1.6. Total order is a partial order with the additional property of comparability, i.e. for any a, b on a totally ordered set S, either $a \leq b$ or $b \leq a$.

1.1. **Functions.** Here are some images, courtesy of Wikipedia, that show what surjective, injective, and bijective functions are:

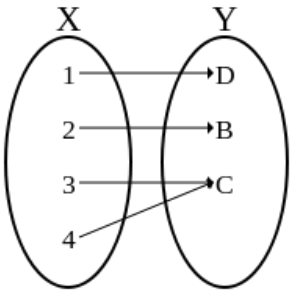


Figure 1. Sujective or onto function

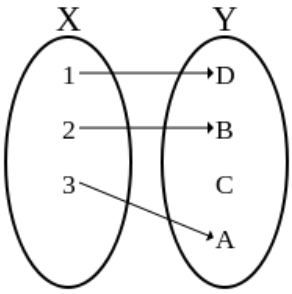


FIGURE 2. Injective or one-to-one function

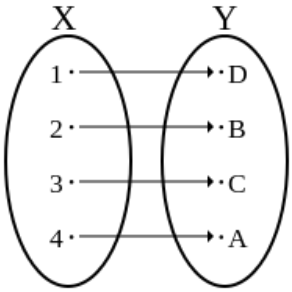


FIGURE 3. Bijective function or a one-to-one correspondence

A set is countable if there is an injection from the set to the positive integers, otherwise the set is called uncountable.

Definition 1.7. A *sequence* is an ordered collection of objects, i.e. a function $f : \mathbb{Z}^{\geq 0} \to X$ where X is an arbitrary set. If $f(n) = x_n$, for $n \in \mathbb{Z}^{\geq 0}$, we denote the sequence f with $\{x_n\}$.

1.2. **Basic Graphs and Definitions.**

Definition 1.8. Connectivity in graphs

(1) Connected undirected graph — each pair of vertices connected by path
(2) Connected components — equivalence class of vertices under "is reachable from" relation
(3) Strongly connected components of directed graph — equivalence class of vertices under "are mutually reachable" relation
(4) Strongly connected directed graph — every two vertices reachable from one another, exactly one strongly connected component

Definition 1.9. A graph isomorphism between graphs G and H is a bijection between the vertex sets of G and H,

$$f : V(G) \to V(H)$$

such that any two vertices $u, v \in G$ are adjacent in G if and only if $f(u)$ and $f(v)$ are adjacent in H.

Definition 1.10. Special graphs

(1) Complete graph — an undirected graph with every pair of vertices adjacent
(2) Bipartite graph — undirected graph in which the vertex set is partitioned into two sets V_1 and V_2 such that every edge are of the form (x, y) where $x \in V_1$ and $y \in V_2$
(3) Tree — connected, acyclic undirected graph
(4) Forest — acyclic undirected graph (i.e. made up of many trees)
(5) DAG — directed acyclic graph
(6) Multigraph — like undirected graph, but can have multiple edges between vertices as well as self-loops
(7) Hypergraph — like undirected graph, but each hyperedge can connect arbitrary number of vertices

Example 1.11. If G is an undirected graph on n nodes, where n is an even number, prove that if every node of G has degree of at least $n/2$, then G is connected. Prove this by contradiction.

Proof. Suppose every node of G has degree of at least $n/2$ and G is not connected. Let u and v be from separate components of G. Let U and V be the set of neighbors of u and v respectively. Since the nodes in U and V are separate, $|U \cap V| = 0$. Recall

$$|U \cap V| = |U| + |V| - |U \cup V|$$

by the inclusion-exclusion principle. Notice that $|U| = |V| = n/2$ by our supposition, which implies

$$|U \cap V| = n - |U \cup V| .$$

Since u and v are not included in $U \cup V$, then $|U \cup V| \leq n - 2$. It follows that 2 is the lower bound for $|U \cap V|$, i.e. $|U \cap V| \geq 2$. A contradiction. □

Proposition 1.12. *Every connected graph $G = (V, E)$ with $|V| \geq 2$ has two vertices x_1 and x_2 so that $G \setminus \{x_1\}$ is connected and $G \setminus \{x_2\}$ is connected.*

Proof. **Base case:** Let G be a connected graph with $V = \{x_1, x_2\}$. Then $G \setminus \{x_1\}$ and $G \setminus \{x_2\}$ are both trivially connected.

Inductive step: Suppose every connected graph G with $|V| = n$ has two vertices x_1 and x_2 so that $G \setminus \{x_1\}$ and $G \setminus \{x_2\}$ are connected. Let G' be a graph as defined in the previous statement but with one additional connected node v'.

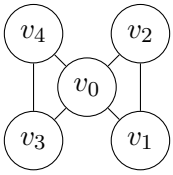
v' has either an edge connected to $G \setminus \{x_1, x_2\}$ or is connected to either (or both) x_1 or x_2. If v' is connected to $G \setminus \{x_1, x_2\}$, then v' is still connected when x_1 or x_2 is removed. Without loss of generality, consider the case when v' has an edge with x_1. Then we can change the vertex that we are removing to be v' instead of x_1, i.e. $G \setminus \{v'\}$ will still be connected. Thus we have shown that if G is connected with $|V| \geq 2$ that there are two vertices so that their indiviual removal results in a connected graph. □

Example 1.13. A connected, undirected graph is vertex biconnected if there is no vertex whose removal disconnects the graph. A connected, undirected graph is edge biconnected if there is no edge whose removal disconnects the graph.

(1) Prove a vertex biconnected graph is edge biconnected for graphs with more than one edge.
(2) Give a counterexample that an edge biconnected graph is vertex biconnected.

Proof. (1) Suppose a vertex biconnected graph is not edge biconnected for graphs with more than one edge. Let G be such a graph and V its set of vertices and E is its set of edges. Then there is an edge $e = (a, b)$, where $a, b \in V$ such that the graph G with edges $E \setminus \{e\}$ is not connected. Notice that G is now made of two connected components with one edge e connecting them. Note that if we remove either a or b that the edge e is implicitly removed, and the graph would no longer be connected. This contradicts our supposition that G was vertex biconnected. Thus a vertex biconnected graph is edge biconnected for graphs with more than one edge. □

(2)



Remove node v_0 in the above edge biconnected graph and we see it is not vertex biconnected.

Example 1.14. We have a connected graph $G = (V, E)$ and a specific vertex $v \in V$. Suppose we compute a depth-first search tree rooted at u and obtain a tree T that includes all nodes of G. Suppose we then compute a breadth-first search tree rooted at u and obtain the same tree T. Prove that $G = T$. (In other words, if T is both a depth-first search tree and a breadth-first search tree rooted at u, then G cannot contain any edges that do not belong to T.)

Proof. Suppose G is a connected graph and its depth-first search (DFS) tree T rooted at vertex u is the same as the breadth-first search (BFS) tree rooted at u, yet $G \neq T$. Then G has at least one edge not included in T. Let E be the set of edges for G, E' be the set of edges for T, and $X = E \setminus E'$ be this set of missing edges. Note that $X \neq \varnothing$. Then the graph with $E' \cup X$ will contain at least one cycle. Let C be one such cycle. In DFS, the vertices of C will be in the same path in T. However, this will not be the case in BFS, as the node that is started on will have a branch that contains one of the vertices of C. A contradiction. Thus $G = T$. □

2. Stable Marriage

Theorem 2.1. *The set of pairs returned by the Gale-Shapley algorithm is a stable matching.*

Proof. Suppose not, that is suppose all matchings have unstable pairs. Let M be the set of men and W be the set of women, both with equal size. Since the matching contains unstable pairs, there exist two pairs of men and women $(m, w) \in M \times W$ and $(m', w') \in M \times W$ such that m prefers w' and m' prefers w, by the definition of unstable. Note, by our algorithm, that m must have proposed to w last. Since m prefers w', m must have proposed to w' earlier; however, since m is no longer proposed to w', this implies that w' prefers another man more than m. Choose this preferred man $m'' \in M$. Since w' is engaged with m', either $m'' = m'$ or w' prefers m' to m''. Notice that this contradicts our supposition that w' prefers m. Thus the set of pairs returned by the Gale-Shapley algorithm is a stable matching. □

Example 2.2. Disprove the following: In every instance of the "Stable Marriage" problem, there is a stable matching containing a pair (m, w) such that m is ranked first on the preference list of w and w is ranked first on the preference list of m.

Algorithm 1 Stable Marriage

```
1: procedure GALE-SHAPLEY(M, W)
2:     All m ∈ M and w ∈ W are free
3:     while ∃m who is free and has not proposed to every w ∈ W do
4:         Choose such a man m
5:         Let w be the highest ranked in m's preference list to whom m has not yet proposed
6:         if w is free then
7:             (m, w) become engaged
8:         else w is engaged to m′
9:             if w prefers m′ to m then
10:                m remains free
11:            else w prefers m to m′
12:                (m, w) become engaged
13:                m′ becomes free
```

Proof. Consider the following "Stable Marriage" problem. Let $A, B \in M$, the set of men, and $C, D \in W$, the set of women.

TABLE 1. The set of men and women and their preference lists

M	1st Pick	2nd Pick
A	C	D
B	D	C

W	1st Pick	2nd Pick
C	B	A
D	A	B

Then A proposes to C, who accepts the marriage since she is available, and B proposes to D, who accepts the marriage since she is available. The result is that there does not exist a pair such that both m and w are each others first pick. This contradicts the proposition in the problem statement. □

Example 2.3. In the Stable Matching problem, using the Gale-Shapley algorithm, can a woman end up better off by lying about her preferences?

Proof. Consider the following set of men and women and their real preference lists:

M	1st Pick	2nd Pick	3rd Pick
X	A	B	C
Y	B	A	C
Z	A	B	C

W	1st Pick	2nd Pick	3rd Pick
A	Y	X	Z
B	X	Y	Z
C	X	Y	Z

In this scenario, the Gale-Shapely algorithm results in the pairing: A-X, B-Y, C-Z.

If A lies and says she prefers Z to X, the Gale-Shapely algorithm results in the pairing: A-Y, B-X, C-Z, which is a better result for A. Thus there is a switch that would improve the partner of a woman who switched preferences. □

3. Asymptotic Growth of Functions

- $f(n) = o(g(n))$ if and only if $\limsup_{n\to\infty} \frac{f(n)}{g(n)} = 0$.
- $f(n) = O(g(n))$ if and only if $\limsup_{n\to\infty} \frac{f(n)}{g(n)} < \infty$.
- $f(n) = \Theta(g(n))$ if and only if $0 < \limsup_{n\to\infty} \frac{f(n)}{g(n)} < \infty$.
- $f(n) = \Omega(g(n))$ if and only if $\limsup_{n\to\infty} \frac{f(n)}{g(n)} > 0$.
- $f(n) = \omega(g(n))$ if and only if $\limsup_{n\to\infty} \frac{f(n)}{g(n)} = \infty$.

Here are some properties of the above:

- Transpose symmetry — $f(n) = O(g(n)) \iff g(n) = \Omega(f(n))$
- If $f = O(h)$ and $g = O(h)$, then $f + g = O(h)$
- If $g = O(f)$, then $f + g = \Theta(f)$

Recall that you change bases of logarithms with: $\log_b a = \frac{\log_c a}{\log_c b}$.

Remark 3.1. Note that an algorithm that requires an operation such as $\binom{n}{k}$, then that algorithm is polynomial time as $O(n^k)$.

Theorem 3.2. *Any comparison sort algorithm is* $\Omega(n \log n)$.

Proof. Assume the input to the algorithms are distinct numbers $1, \ldots, n$. Then there are $n!$ permutations of these numbers. Let h be the height of the decision tree that corresponds to the algorithm. Notice that

$$\begin{aligned} n! \leq \text{the total number of leaves on the decision tree} \leq 2^h \implies \\ h \geq \log n! &= \log n + \log n - 1 + \cdots + \log 2 \\ &= \sum_{i=2}^{n} \log i \\ &= \sum_{i=2}^{n/2-1} \log i + \sum_{n/2}^{n} \log i \\ &\geq \sum_{n/2}^{n} \log \frac{n}{2} \\ &= \frac{n}{2} \log \frac{n}{2} \\ &= \Omega(n \log n). \end{aligned}$$

□

Example 3.3. You are given 9 identical looking balls and told that one of them is slightly heavier than the others. Your task is to identify the defective ball. All you have is a balanced scale that can tell you which of the two sides of the scale is heavier (any number of balls can be placed on each side of the scale).

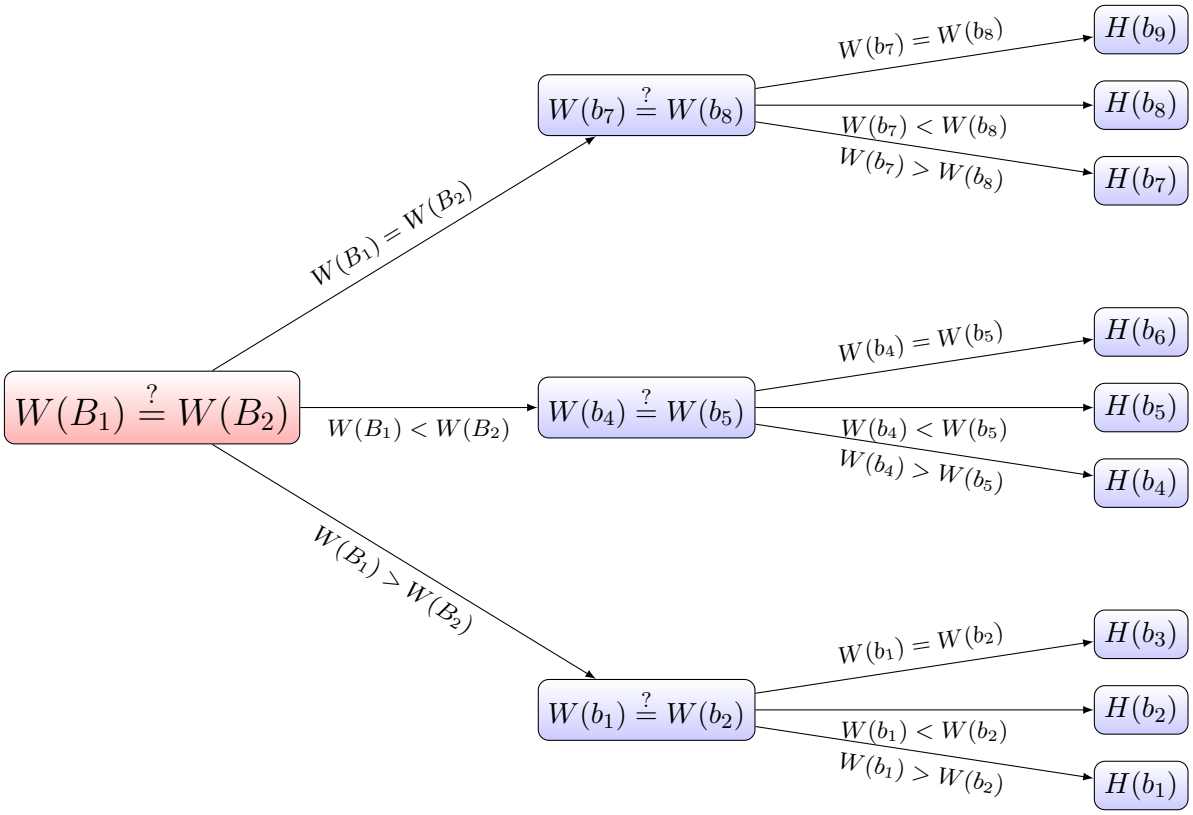
(1) Show how to identify the heavier ball in just 2 weighings.
(2) Give a decision tree lower bound showing that it is not possible to determine the defective ball in fewer than 2 weighings.

(1) Let $B = \{b_1, \ldots, b_9\}$ be the set of balls. First we weigh $\frac{1}{3}$ of the balls. Without loss of generality, we can say we first weigh balls $B_1 = \{b_1, b_2, b_3\}$ and $B_2 = \{b_4, b_5, b_6\}$. Let B_3 be the remaining set of balls, i.e. $B_3 = \{b_6, b_7, b_8\}$. Define $W(\cdot)$ to be the mapping that weighs a ball or group of balls.

If $W(B_1) = W(B_2)$, then we split B_3 into thirds and weigh two balls, WLOG say b_7 and b_8. If $b_7 = b_8$, then b_9 is the heaviest ball.

If $W(B_1) \neq W(B_2)$, then choose the heavier group and do the algorithm done to set B_3.

(2) Let B_1, B_2, B_3, and $W(\cdot)$ be defined as above and let $H(\cdot)$ represent the heaviest ball.



It is clear from the decision tree that there are no ways to reduce the amount of weighings from 2.

4. Data Structures

Definition 4.1. A *priority queue* is like a queue or stack, except each element also has a priority associated with it; an element with higher priority is served before an element with low priority.

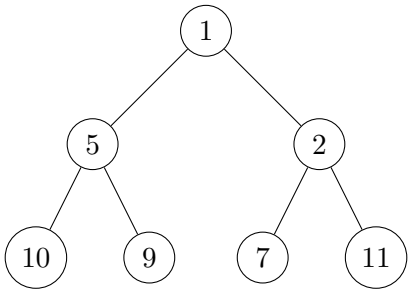
Definition 4.2. A *heap* is a specialized tree-based data structure that satisfies the heap property: If A is a parent node of B then the key (the value) of node A is ordered with respect to the key of node B with the same ordering applied across the heap.

A heap is a convenient data structure on which to implement a priority queue.

Example 4.3. Here is an example of a heap, all parents are smaller than their children:

1	5	2	10	9	7	11

This is implemented such that if the parent is stored at index i, then the two children at most are stored at $2i$ and $2i + 1$. Here is the graph associated with the heap above.



To insert a new element on a heap, we implement an algorithm called *heapify-up*. The element is added to the bottom level of the heap (at the left most freely available spot). The element is compared to its parent; if they are in the correct order, then we are done. However, if not, then we swap the element with its parent and start again at the last step until we are done.

To remove an element on the heap, we implement an algorithm called *heapify-down*. First we replace the root of the heap with the last element on the last level. Then we compare the new root with its children; if they are in the correct order, then we are done. However, if not, then we swap the element with one of its children and return to the previous step. (The swap depends on the ordering of the heap)

Example 4.4. Give an efficient algorithm to find all keys in a min-heap that are smaller than a provided value. The provided value does not have to be a key in the min-heap. Evaluate the time complexity of your algorithm.

Recall a min-heap is a complete binary tree in which the data contained in each node is less than (or equal to) the data in that node's children.

Algorithm 2 Find all smaller values in a min-heap

```
1: procedure FIND SMALLER(root, value)
2:     if root > value then
3:         return
4:     Find smaller(root's left node, value)
5:     Find smaller(root's right node, value)
6:     return root
```

Let n be the number of nodes in the min-heap. This algorithm's time complexity is $O(n)$, since at worst case the value will be greater than every node in the min-heap.

5. Graph Theory and Algorithms

An adjacency list contains one linked list per node, such that each linked list contains the edges to all adjacent nodes. It requires $\Theta(|V| + |E|)$ total memory. Good for sparse graphs.

An adjacency matrix has rows i and columns j of the nodes of the graph, and the non-zero entries represent if there exist an edge between the nodes i, j. Requires $\Theta(|V|^2$ total memory. Better for dense graphs, and good when we need to check if an edge exists $O(1)$ vs $O(|V|)$ for the adjacency list.

Definition 5.1. *Breadth-first search* (BFS) is an algorithm for traversing or searching tree or graph data structures. It starts at the tree root (or some arbitrary node of a graph, sometimes referred to as a 'search key') and explores the neighbor nodes first, before moving to the next level neighbors.

BFS is useful for finding the shortest path from one node to another (the shortest path from the root of the BFS tree is the height of the tree to that node's level). BFS is also useful in finding the connected components of a graph.

Definition 5.2. *Depth-first search* (DFS) is an algorithm for traversing or searching tree or graph data structures. One starts at the root (selecting some arbitrary node as the root in the case of a graph) and explores as far as possible along each branch before backtracking.

A *cross edge* is an edge, not in the BFS or DFS tree, that exists in the original graph and connects two edges on the same level. A *back edge* is an edge, also not in the BFS or DFS tree, that exists in the original graph and connects two edges on different levels.

Algorithm 3 Depth-first search

```
1: procedure DFS(G, v)
2:     label v as discovered
3:     for each edge from v to w ∈ G where w is adjacent to v do
4:         if w is not labeled discovered then
5:             call DFS(G, w)
```

Definition 5.3. An *independent set* is a set of vertices in a graph, no two of which are adjacent

Definition 5.4. A *bipartite graph* is a graph whose vertices can be divided into two disjoint sets U and V (U and V are each independent sets) such that every edge connects a vertex in U to one in V.

We can test for bipartiteness with BFS and check to see there are no cross edges.

Definition 5.5. *Strong connectivity* is a property of a graph that means all nodes are mutually reachable.

Theorem 5.6. *We can determine if a graph G is strongly connected in $O(m + n)$.*

Algorithm 4 Test for Strongly Connected Graph

```
1: procedure STRONGLY CONNECTED TEST(G, v)
2:     Run BFS from v
3:     Run BFS from v but reverse the directions of the edges of G
4:     return true if and only if all nodes reached in both BFS trees
```

Definition 5.7. A *topological ordering* of a directed graph is a linear ordering of its vertices such that for every directed edge (u, v) from vertex u to vertex v, u comes before v in the ordering.

Remark 5.8. Any DAG has at least one topological ordering, and there exist algorithms for constructing a topological ordering of any DAG in $O(n)$.

6. Greedy algorithms

Definition 6.1. A problem is said to have *optimal substructure* if an optimal solution can be constructed efficiently from optimal solutions of its subproblems.

Definition 6.2. A problem has the *greedy-choice property* if a global optimum can be arrived at by selecting a local optimum.

If a problem has both an optimal substructure and the greedy-choice property, then the problem can be solved with a greedy algorithm.

Example 6.3. To find a solution to the problem of interval scheduling, a greedy algorithm will pick the job with the earliest finish time.

Algorithm 5 Interval scheduling algorithm

```
1: procedure INTERVAL SCHEDULER(Jobs)
2:     Sort jobs by finish time
3:     Selected Jobs ← ∅
4:     for each job j do
5:         if j is compatible with Selected Jobs then
6:             Selected Jobs ∪ {j}
```

To show that a greedy algorithm is optimal, we must show that the greedy algorithm stays ahead or is equal to any solution that can be constructed.

Example 6.4. Consider the problem of making change for n cents using the fewest number of coins. Assume that each coin's value is an integer.

(1) Describe a greedy algorithm to make change consisting of quarters, dimes, nickels, and pennies. Prove that your algorithm yields an optimal solution.
(2) Give a set of coin denominations for which your greedy algorithm does not yield an optimal solution. Your set should include a penny to ensure that you can always successfully make change.

(1)

Algorithm 6 Make change with American coins

```
1:  procedure MAKE CHANGE(n)
2:      Initialize C as an empty array to hold the change
3:      while n > 0 do
4:          if n ≥ 25 then
5:              n ← n − 25
6:              Add a quarter to C
7:          else if n ≥ 10 then
8:              n ← n − 10
9:              Add a dime to C
10:         else if n ≥ 5 then
11:             n ← n − 5
12:             Add a nickel to C
13:         else
14:             n ← n − 1
15:             Add a penny to C
```

Proposition 6.5. *The* MAKE CHANGE *algorithm yields an optimal solution.*

Proof. To show that MAKE CHANGE yields an optimal solution, we need to show that the problem has an optimal substructure and that the greedy choice property holds.

Let $C = \{c_1, c_2, \ldots, c_k\}$ be the optimal solution for the amount n. The optimal substructure of the problem can be seen by noticing that we can remove a coin from C, without loss of generality say $C' = C \setminus \{c_k\}$, and noticing that C' is the optimal solution for the amount $n' = n -$ value(c_k).

If C' were not the optimal solution for n', then there would exist an optimal solution, say C'' for n'. However, if C'' is more optimal than C', we can add c_k back to C'' for a more optimal solution to the amount n. Which contradicts our supposition that C is the optimal solution to the amount n.

Now we will show that the greedy choice property applies to the problem. Suppose the greedy choice is not in the optimal solution S. Since the greedy choice has not been followed, this means that lower value coins $G \subset S$ were used to make change for some higher value coin c. However, we can replace G with our high value coin c to yield a more optimal solution. Thus the greedy choice property holds. Since we have shown the problem has both an optimal substructure and the greedy choice property, it follows that the greedy algorithm MAKE CHANGE produces an optimal solution. □

(2) Let the set of coin denominations be $\{1, 4, 5, 7\}$ and get change for 9 cents. The greedy algorithm will pick 7 and then 1 twice for a total of 3 coins. However, the optimal solution is picking the 4 and 5 cent coins for a total of 2 coins.

Example 6.6. Let X be a set of n intervals on the real line. A subset of intervals $Y \subseteq X$ is called a tiling path if the intervals in Y cover the intervals in X, that is, any real value that is contained

in some interval in X is also contained in some interval in Y. The size of a tiling cover is just the number of intervals.

Describe an algorithm to compute the smallest tiling path of X. Argue that your algorithm is correct

Algorithm 7 Find the smallest path that tiles an interval

```
1: procedure FIND SMALLEST TILING PATH(X)
2:     T ← end of interval X
3:     t ← start of interval X
4:     Initialize Y = ∅, the set that holds the tiling intervals
5:     while t < T do
6:         y ← farthest right stretching interval starting from or before t in X
7:         Y ∪ {y}
8:         t ← end of y
```

Proposition 6.7. *The* FIND SMALLEST TILING PATH *algorithm solves the problem correctly.*

Proof. We must first show that the problem that is to be solved has an optimal substructure, then we will show that the greedy choice property holds.

Notice that FIND SMALLEST TILING PATH breaks up the tiling problem into smaller problems by changing the interval of search for smaller regions when we update t.

Suppose the greedy choice is not optimal. Let O be the optimal tiling. Then O contains an interval chosen such that it does not have the furthest right end-point. Let i be that interval. Note we can be replace i with the interval with the furthest right end-point, and the result will be as or more efficient in solving the smallest tiling path problem. □

Example 6.8. Consider a long, quiet country road with houses scattered sparsely along it. (We can picture the road as a long line segment, with an eastern endpoint and a western endpoint.) Further, let's suppose that despite the bucolic setting, the residents of all these houses are avid cell phone users. You want to place cell phone base stations at certain points along the road, so that every house is within 4 miles of one of the base stations.

(1) Give an efficient algorithm that achieves this goal, using as few base stations as possible.
(2) Prove that the greedy choice that your algorithm makes is the optimal choice.

(1) I will assume that the ball surrounding the base station is closed, i.e. the region of coverage around a base station contains its endpoints. I will also assume that the road can be represented by a compact set, i.e. if the road is a subset of the real line, it is a closed and bounded subset. Finally, I will assume that the road is at least 4 miles long. Note that this algorithm will work if you switch all instances of east with west and vice versa. View algorithm 8 for the algorithm.

(2)

Proposition 6.9. *The greedy choice that* BASE STATION PLACER *makes is the optimal choice.*

Proof. Let $B = \{b_1, b_2, \ldots, b_n\}$, for $n \in \mathbb{Z}^{\geq 0}$, be the set of base station locations placed by the BASE STATION PLACER algorithm and $B' = \{b'_1, b'_2, \ldots, b'_k\}$, for $k \in \mathbb{Z}^{\geq 0}$, be the set of base station

Algorithm 8 Place base stations to create an optimal solution

```
1: procedure Base station placer(Road)
2:     Start at the western most point of the road
3:     while not at the eastern end of the road do
4:         Travel east until we are exactly 4 miles east of an uncovered house
5:         Place a base station at our current location
6:         Ignoring all houses covered by this base station, repeat this process
```

location for the optimal solution. Note that both B and B''s elements are indexed from west to east. Without loss of generality, assume that the western-most point is 0 and the eastern-most point is some positive real number N. It follows that the locations of the base stations b_i are real numbers between 0 and N. Now we will show that the solution provided by B is the same or better to the solution provided by B'.

Suppose that Base station placer did not result in the optimal solution. Then there is a $b'_i \in B'$, such that b'_i is not 4 miles east of a house. However, we can swap this element with a base station that is exactly 4 miles east of that house and have an equivalent or better solution. In fact, we can do this to every element where this is the case change the optimal solution into B. Thus the greedy algorithm Base station placer results in an optimal solution. □

6.1. **Shortest Path.** To find the shortest path in a weighted graph with non-negative edges, we use Djikstra's algorithm.

Algorithm 9 Dijkstra's algorithm to find a shortest path in a graph

```
1: procedure Dijkstra(G : graph, s : vertex)
2:     for each vertex v ∈ V_G do
3:         dist[v] ← ∞
4:         parent[v] ← NIL
5:     dist[s] ← 0
6:     Q ← V_G
7:     while Q ≠ ∅ do
8:         u ← Extract-min(Q)
9:         for each edge e = (u, v) do
10:            if dist[v] > dist[u] + weight[e] then
11:                dist[v] ← dist[u] + weight[e]
12:                parent[v] ← u
13:    H ← (V_G, ∅)
14:    for each vertex v ∈ V_G where v ≠ s do
15:        E_H ← E_H ∪ {(parent[v], v)}
16:    return H, dist
```

Example 6.10. A looped tree is a weighted directed graph built from a binary tree by adding an edge from every leaf back to the root. Every edge has a non-negative weight.

(1) How much time would Dijkstra's algorithm require to compute the shortest path from u to v in a looped tree with n nodes? (Do NOT assume that either u or v is the root of the tree though one could be.)
(2) Describe and analyze a faster algorithm to find the shortest path from u to v in a looped tree.

(1) Dijkstra's algorithm's complexity is $O(|E| \log n)$. In this case $|E| = O(n)$, so it follows that the overall complexity for this will be $O(n \log n)$.
(2)

Algorithm 10 Find the shortest path in a looped tree

```
1: procedure LOOPED TREE SHORTEST PATH(T, u, v)
2:     D ← DFS(T) starting at root
3:     if u is an ancestor of v then
4:         Choose path directly down D from u to v
5:     else
6:         S ← subtree of u with the root as the children of the leaves (not connected to rest of T)
7:         Path ← TOPOLOGICAL SORT SHORTEST PATH(S)
8:         Take Path to get from u to root, then follow the path in D from root to v.
```

DFS is known to be $O(n)$ and TOPOLOGICAL SORT SHORTEST PATH is $O(n + |E|)$. Since $|E| = O(n)$, LOOPED TREE SHORTEST PATH is $O(n)$.

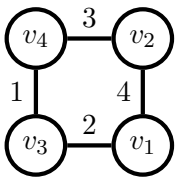
6.2. **Minimum Spanning Tree.** There are two main algorithms for finding the minimum spanning tree of a graph: (1) Kruskal's algorithm and (2) Prim's algorithm. Here are (oversimplified) overviews of the two algorithms.

Kruskal's algorithm. For a graph G, start with $|V|$ trees (one for each vertex). Consider edges E in increasing order of weights, and add an edge if it connects two trees.

Prim's algorithm. Start with spanning tree containing arbitrary vertex v and no edges. Grow spanning tree by repeatedly adding minimal weight edge connecting vertex v in current spanning tree with a vertex not in the tree.

Example 6.11. Is the path between a pair of vertices in a minimum spanning tree necessarily a shortest path between the two vertices in the full graph? Give a proof or a counter example.

Proof. Consider the following graph:



Note that the minimum spanning tree contains the edges weighted by 1, 2, 3. However, the edge between v_1 and v_2, weighted with 4, is the shortest path between the two nodes. Thus the path between a pair of vertices in a minimum spanning tree is not necessarily a shortest path between the two vertices in the full graph. □

Example 6.12. Let us say that a graph $G = (V, E)$ is a near-tree if it is connected and has at most $n + 8$ edges, where $n = |V|$. Give an algorithm with running time $O(n)$ that takes a near-tree G with costs on its edges and returns a minimum spanning tree of G. You may assume that all of the edge costs are distinct.

Algorithm 11 Algorithm to find a minimum spanning tree in a near-tree

```
1: procedure NEAR-TREE MINIMUM SPANNING TREE(G)
2:     B ←BFS(G)
3:     for each edge e of G not found in B do
4:         Insert e into B
5:         Find the greatest weight edge in the cycle created by the insertion and remove it
```

Since there are at most 9 edges to check, the above algorithm runs in $O(n)$, due to running BFS.

6.3. **Fractional Knapsack.** Suppose you are a thief, and you have a knapsack with a weight capacity. You want to maximize the value you can steal. If you can fill a knapsack with fractions of items instead of whole items, we can solve the problem with a greedy solution. Just fill your knapsack with the highest value per weight first, then fill with the next highest value per weight, and so on, until the knapsack is full.

6.4. **Huffman Codes.** Huffman codes are a variable length data compression scheme. It relies on finding the probability of each character appearing in the file to be compressed, and using less bits for characters that appear with higher probability, and more for characters that appear with less probability.

Algorithm 12 Huffman encoding algorithm

```
1: procedure HUFFMAN(C : characters to encode)
2:     n ← |C|
3:     Q ← C                          ▷ initialize heap with elements of C, probability as key
4:     for i ← 1 to n − 1 do
5:         allocate new node z
6:         left[z] ← x ← EXTRACT-MIN(Q)
7:         right[z] ← y ← EXTRACT-MIN(Q)
8:         f[z] ← f[x] + f[y]
9:         INSERT(Q, z)
10:    return EXTRACT-MIN(Q)
```

Note HUFFMAN is $O(|V| \log |V|)$.

Example 6.13. Consider the binary tree that is constructed in the Huffman coding procedure. Prove that a binary tree that is not full cannot correspond to an optimal prefix code.

Proof. Suppose that there is a binary tree that is not full that does correspond to an optimal prefix code. Let that tree be designated by T, with the node v which only has one child c. However, we can replace v with c and have a more optimal code, since less bits will be needed to represent the value of c. A contradiction. □

7. Divide and Conquer

The divide-and-conquer strategy solves a problem by:

(1) Breaking it into subproblems that are themselves smaller instances of the same type of problem
(2) Recursively solving these subproblems
(3) Appropriately combining their answers

Divide-and-conquer, generally, can only reduce time complexity from a higher polynomial to a lower polynomial.

7.1. Exponentiation.

We can speed up the polynomial time (in this case $O(n)$) operation of normal exponentiation with a divide and conquer algorithm.

Algorithm 13 Implementation of an exponentiation function with divide and conquer

```
1: function FAST POWER(a, n)
2:     if n = 1 then
3:         return a
4:     x ← FASTPOWER(a, ⌊n/2⌋)
5:     if n is even then
6:         return x · x
7:     else
8:         return x · x · a
```

7.2. Merge-sort.

Divide the unsorted list into n sublists, each containing 1 element (a list of 1 element is considered sorted). Repeatedly merge sublists to produce new sorted sublists until there is only 1 sublist remaining. This will be the sorted list.

Example 7.1. Give an efficient algorithm to find the k^{th} largest element in the merge of two sorted sequences S_1 and S_2. The best algorithm runs in $O(\log(\max(m, n)))$, where $|S_1| = n$ and $|S_2| = m$.

We will assume that $k \leq |S_1| + |S_2|$, either S_1 or S_2 is non-empty, and both S_1 and S_2 are sorted from largest to smallest. When each sequence is sorted in this way, the k^{th} largest element is in the k^{th} index (or potentially in the $k - 1^{th}$ spot, if indexed as most programming languages).

Algorithm 14 Find the k^{th} largest element in two sorted arrays

```
1: function FIND k^th LARGEST(S_1, S_2, k)
2:     if S_1 = ∅ and S_2 ≠ ∅ then
3:         return S_2[k]
4:     else if S_2 = ∅ and S_1 ≠ ∅ then
5:         return S_1[k]
6:     m_1 ← middle index of S_1
7:     m_2 ← middle index of S_2
8:     if m_1 + m_2 < k then
9:         if S_1[m_1] ≤ S_2[m_2] then
10:            return FIND k^th LARGEST(S_1[> m_1], S_2, k − m_1 − 1)
11:        else
12:            return FIND k^th LARGEST(S_1, S_2[> m_2], k − m_2 − 1)
13:    else
14:        if S_1[m_1] ≤ S_2[m_2] then
15:            return FIND k^th LARGEST(S_1, S_2[< m_2], k)
16:        else
17:            return FIND k^th LARGEST(S_1[< m_1], S_2, k)
```

7.3. **Algorithm analysis.**

Master method overview. The master theorem concerns recurrence relations of the form:

$$T(n) = a\,T\left(\frac{n}{b}\right) + f(n) \text{ where } a \geq 1,\, b > 1$$

- n is the size of the problem.
- a is the number of subproblems in the recursion.
- n/b is the size of each subproblem. (Here it is assumed that all subproblems are essentially the same size.)
- $f(n)$ is the cost of the work done outside the recursive calls, which includes the cost of dividing the problem and the cost of merging the solutions to the subproblems.

There are three cases:

(1) If $f(n) \in O\left(n^c\right)$ where $c < \log_b a$, then $T(n) \in \Theta\left(n^{\log_b a}\right)$.

(2) If for some constant $k \geq 0$, that $f(n) \in \Theta\left(n^c \log^k n\right)$ where $c = \log_b a$, then $T(n) \in \Theta\left(n^c \log^{k+1} n\right)$.

(3) If $f(n) \in \Omega\left(n^c\right)$ where $c > \log_b a$ and if $af\left(\frac{n}{b}\right) \leq kf(n)$ for some constant $k < 1$, then $T\left(n\right) \in \Theta\left(f(n)\right)$.

Example 7.2. Use the master method to give asymptotic bounds the following recurrences:

(1) $T(n) = 5T(n/2) + n$

(2) $T(n) = 5T(n/2) + n^2$
(3) $T(n) = 5T(n/2) + n^3$

(1) Here $f(n) = n$, $a = 5$, and $b = 2$. Since $1 < \log_2 5$, we are in case 1. Thus $T(n) \in \Theta\left(n^{\log_2 5}\right)$.

(2) Here $f(n) = n^2$, $a = 5$, and $b = 2$. Since $2 < \log_2 5$, we are in case 1. Thus $T(n) \in \Theta\left(n^{\log_2 5}\right)$.

(3) Here $f(n) = n^3$, $a = 5$, and $b = 2$. Since $3 > \log_2 5$ and $\frac{5}{8}n^3 \leq \frac{5}{8}n^3$, we are in case 3. Thus $T(n) \in \Theta\left(n^3\right)$.

7.4. **Closest Pairs.** Given n points in the Euclidean plane, find pair with smallest distance between them. Note that the brute force solution would require us to check all points with $\Theta(n^2)$ comparisons. See algorithm 15.

Algorithm 15 Find closest pairs using divide and conquer

```
1:  procedure CLOSEST PAIR(P : set of all points)
2:      Compute a vertical line in the plane L such that half points on each side of L
3:      δ1 ← CLOSEST PAIR(P1...L)
4:      δ2 ← CLOSEST PAIR(PL+1...n)
5:      δ ← min(δ1, δ2)
6:      Delete all points further than δ from L
7:      sort remaining points by y-coordinate
8:      Scan points in y order and compare distance between each point and next 11 neighbors
9:      If any of these distances are less than δ, update δ
10:     return δ
```

7.5. **More Examples.**

Example 7.3. Among the collection of n cards, is there a set of more than $n/2$ of them that are all equivalent to one another? Assume that the only feasible operations you can do with the cards are to pick two of them and plug them in to the equivalence tester. Show how to decide the answer to their question with only $O(n \log n)$ invocations of the equivalence tester.

Let C be the set of cards.

Algorithm 16 Find if half or more of the cards in a set are equivalent

```
1: function CARD EQUIVALENCE DRIVER(C)
2:     if |C| = 1 then
3:         return c_1 ∈ C
4:     if |C| = 2 and EQUIVALENCE TESTER(c_1, c_2) for c_1, c_2 ∈ C is true then
5:         return c_1 ∈ C
6:     C_l ← half of C
7:     C_r ← other half of C
8:     c_l ← CARD EQUIVALENCE DRIVER(C_l)
9:     c_r ← CARD EQUIVALENCE DRIVER(C_r)
10:    if c_l ≠ ∅ then
11:        for c ∈ C do
12:            EQUIVALENCE TESTER(c_l, c)
13:        if c_l is equivalent to |C|/2 cards then
14:            return c_l
15:        else if c_r ≠ ∅ then
16:            for c ∈ C do
17:                EQUIVALENCE TESTER(c_r, c)
18:            if c_l is equivalent to |C|/2 cards then
19:                return c_r
20:            else
21:                return ∅
```

Example 7.4. In this problem, you are to analyze the asymptotic time complexity $T(n)$ for the following divide and conquer algorithm. Assume that BAR(k, n) performs a constant time operation on whatever input it receives (that is, don't worry about what exactly it does). Show your work. Write the complete recurrence, and solve it using either the master method or a recursion tree.

LISTING 1. Problem code

```
int[] foo(int[] A) {
   n = A.size();
   int B[] = new int[n];
   if (n <= 1) return;
   int[] even = new int[n/2];
   int[] odd = new int[n/2];
   int j = 0;
   for(int i = 0; i<n; i=i+2) {
      even[j] = A[i];
      j++;
   }
   j = 0;
   for(int i = 1; i<n; i=i+2) {
      odd[j] = A[i];
```

```
        j++;
    }
    foo(even);
    foo(odd);
    for(int k = 0; k<n/2; k++) {
        int t = bar(k, n) * odd[k];
        B[k] = even[k] + t;
        B[k+n/2] = even[k] - t;
    }
    return B;
}
```

After analysis, We see the recurrence relation for `foo()` is

$$T(n) = 2T(n/2) + 3n/2.$$

To find the asymptotic bound we will use the master method. Note $f(n) = 3n/2$, $a = 2$, and $b = 2$. Since $1 = \log_2 2$, we are in case 2. It follows that $T(n) = \Theta(n \log n)$.

8. Dynamic Programming

Dynamic programming breaks problems up into a series of overlapping subproblems, then builds the solutions to larger and larger subproblems. It differs from divide and conquer since the subproblems are no independent, and solves subproblems just once to save time.

Dynamic programming typically involves the following steps:

(1) Characterize the structure of an optimal solution
(2) Recursively define the value of optimal solution
(3) Compute the value of an optimal solution from the bottom-up
(4) Construct optimal solution from computed information

Dynamic programming is a technique for solving problems recursively and is applicable when the computations of the subproblems overlap.

Memoization is a term describing an optimization technique where you cache previously computed results, and return the cached result when the same computation is needed again.

When you solve a dynamic programming problem using tabulation you solve the problem "bottom up", i.e., by solving all related sub-problems first, typically by filling up an n-dimensional table. Based on the results in the table, the solution to the "top" / original problem is then computed.

If you use memoization to solve the problem you do it by maintaining a map of already solved sub problems. You do it "top down" in the sense that you solve the "top" problem first (which typically recurses down to solve the sub-problems).

Example 8.1. The Fibonacci function is defined as follows:

1: **function** F(n)
2: F(0) = 1
3: F(1) = 1
4: **if** $n > 2$ **then**
5: F(n) = F(n-1) + F(n-2)

(1) Assume you implement a recursive procedure for computing the Fibonacci sequence based directly on the function defined above. Then the running time of this algorithm can be expressed as:
$$T(n) = T(n-1) + T(n-2) + 1$$
Determine the asyptotic bound that best satisfies the above recurrance relation.
(2) What specifically is bad about your algorithm? (i.e., what observation can you make to radically improve its running time?)
(3) Give a memoized recursive algorithm for computing $\mathrm{F}(n)$ efficiently. Write the recurrence for your algorithm and give its asymptotic upper bound.
(4) Give a traditional bottom-up dynamic programming algorithm for computing $\mathrm{F}(n)$ efficiently. Write the recurrence for your algorithm and give its asymptotic upper bound.

(1) The asymptotic bound that best satisfies the recurrance relation is $\Omega(c^n)$ (where $c = 2$).
(2) We unnecessarily recompute previous values in every iteration.
(3)

Algorithm 17 Memoized Fibonacci function

```
1: Initialize global vector M = ∅
2: function F(n)
3:     F(0) = 1
4:     F(1) = 1
5:     if M[n] = ∅ then
6:         M[n] = F(n-1) + F(n-2)
7:     return M[n]
```

The asymptotic upper bound of this function is $O(n)$.
(4)

Algorithm 18 Dynamic programming Fibonacci function

```
1: Initialize global vector M = ∅
2: function F(n)
3:     M[0] = M[1] = 1
4:     for i ∈ {2, ..., n} do
5:         M[i] = M[i − 1] + M[i − 2]
6:     return M[n]
```

The asymptotic upper bound of this function is $O(n)$.

8.1. **Weighted Interval Scheduling.** The goal of the weighted interval scheduling problem is to find the maximum weight subset of mutually compatible jobs. It differs from the interval scheduling problem, that can be solved by greedy, since the intervals have weights.

First we sort jobs by finishing time.

Define $p(j)$ to be the largest index $i < j$ such that job i is compatible with job j. Compatible meaning the jobs do not overlap.

Define $\textsc{Opt}(j)$ as the value of an optimal solution to a problem consisting of the job requests $1, 2, \ldots, j$.

We will first show the solution to this problem using memoization.

Algorithm 19 Memoized weighted interval scheduling

```
1: procedure MEMOIZED(n : number of jobs, s : start times, f : finish times, w : weights)
2:     Sort jobs by f
3:     Compute p(1), p(2), ..., p(n)
4:     for j ∈ {1, ..., n} do
5:         M[j] ← ∅
6:     M[0] ← 0
7:     M-COMPUTE-OPT(j)
1: function M-COMPUTE-OPT(j : job number)
2:     if M[j] = ∅ then
3:         M[j] ← max(w_j + M-COMPUTE-OPT(p(j)), M-COMPUTE-OPT(j − 1))
4:     return M[j]
```

Now we will "unwind" the recursion and solve it using dynamic programming.

Algorithm 20 Dynamic programming weighted interval scheduling

```
1: procedure DP(n : number of jobs, s : start times, f : finish times, w : weights)
2:     Sort jobs by f
3:     Compute p(1), p(2), ..., p(n)
4:     M[0] ← 0
5:     for j ∈ {1, ..., n} do
6:         M[j] ← max(w_j + M[p(j)], M[j − 1]))
```

8.2. **Segmented Least Squares.** In the segmented least squares problem, we have points that lie roughly on a sequence of several line segments. Given the points in the plane, find the sequence of lines that minimize the error of $f(x)$.

Algorithm 21 Segmented Least Squares

```
1: procedure SLS(n : number of points, p : points, c : cost per line)
2:     M[0] ← 0
3:     for j ∈ {1, ..., n} do
4:         for i ∈ {1, ..., j} do
5:             compute least square error e_ij for segment p_i, ..., p_j
6:     for j ∈ {1, ..., n} do
7:         M[j] ← min_{1≤i≤j}(e_ij + c + M[i − 1])
8:     return M[n]
```

This runs in $O(n^3)$ time.

8.3. **0-1 Knapsack.** This problem is as defined in the fractional knapsack, except we cannot take parts of items, we must either take the item or not.

Define $\textsc{Opt}(i, w)$ to be the max profit of items $1, \ldots, i$ with weight limit w.

Algorithm 22 0-1 Knapsack

```
1: procedure 0-1 KS(n : number of items, W : capacity of knapsack, 𝒲 : weight of items, v :
   value of items)
2:     M[0, w] ← 0
3:     for i ∈ {1, ..., n} do
4:         for w ∈ {1, ..., W} do
5:             if 𝒲_i > w then
6:                 M[i, w] = M[i − 1, w]
7:             else
8:                 M[i, w] = max(M[i − 1, w], v_i + M[i − 1, w − 𝒲_i])
9:     return M[n, W]
```

8.4. **Coin Changing.** We can use dynamic programming to optimally (least amount of coins) make change for *any* denomination of currency. See algorithm 23.

8.5. **Shortest Path Revisited.** We now consider finding the shortest path when the graph contains negative weights, a problem which Djikstra's algorithm cannot solve.

To do this we can use the Bellman-Ford algorithm. Note that the graph cannot contain net negative cycles.

First we define $\textsc{Opt}(i, v)$ as the length of the shortest path P from node v to node t using at most i edges.

- Case 1: P uses at most $i - 1$ edges, i.e. $\textsc{Opt}(i, v) = \textsc{Opt}(i - 1, v)$.
- Case 2: P uses exactly i edges, i.e. if (v, w) is the first edge, the $\textsc{Opt}$ uses (v, w), and then selects best $w - t$ path using at most $i - 1$ edges.

Algorithm 23 Dynamic Programming Coin Changing algorithm

```
1: procedure CC(d : denomination of currency, k : number of denominations, n : amount of
   change to be made)
2:     C[0] ← 0
3:     for p ∈ {1, ..., n} do
4:         min ← ∞
5:         for i ∈ {1, ..., k} do
6:             if d[i] ≤ p then
7:                 if 1 + C[p − d[i]] < min then
8:                     min ← 1 + C[p − d[i]]
9:                     coin ← i
10:        C[p] ← min
11:        S[p] ← coin
12:    return C, S
```

For the Bellman-Ford algorithm, we can formally define $\textsc{Opt}(i, v)$ to be:

$$\textsc{Opt}(i,v) = \begin{cases} 0, \text{ if } i = 0 \\ \min(\textsc{Opt}(i-1,v), \min_{(v,w)\in E}(\textsc{Opt}(i-1,w) + c_{vw}), \text{ otherwise.} \end{cases}$$

See algorithm 24 for the Bellman-Ford algorithm.

Algorithm 24 Bellman-Ford algorithm

```
1: procedure SHORTEST PATH(G, s, t)
2:     for i ∈ {1, ..., |V| − 1} do
3:         for v ∈ V do
4:             M[i, v] ← min(M[i − 1, v], min_{w∈V}(M[i − 1, w] + c_vw)
5:     return M[|V| − 1, s]
```

8.6. **Box Stacking.** Given n boxes each with height h_i, width w_i, and depth d_i, construct a stack with maximal height under the constraint that we can only stack box i on box j if $w_i < w_j$ and $d_i < d_j$. Let $H(j)$ be the max height using box j on top. See algorithm 25.

Algorithm 25 Maximal height box stacking algorithm

```
1: procedure BOX STACK(n : number of boxes, h : heights, w : widths, d : depths)
2:     Sort boxes from largest base area to smallest
3:     H(j) ← { h_j, if j = 1
               { max_{i<j, w_i>w_j, d_i>d_j}{H(i)} + h_j, if j > 1
4:     return max_j{H(j)}
```

To compute the maximum as descibed in the algorithm requires n^2 operations, which makes the algorithm run in $O(n^2)$.

9. Network Flow

Given a directed graph $G = (V, E)$, where each edge e is associated with its capacity $c(e) > 0$. Two special nodes source s and sink t are given where $s \neq t$. We want to maximize the total amount of flow from s to t subject to two constraints:

(1) Flow on edge e doesn't exceed $c(e)$;
(2) For every node $v \neq s$ and $v \neq t$, incoming flow is equal to outgoing flow.

9.1. Minimum Cut.

Definition 9.1. An $s - t$ cut $C = (S, T)$ is a partition of G_V such that $s \in S$ and $t \in T$. The cut-set of C is the set

$$(u, v) \in E \ : \ u \in S, v \in T.$$

Note that if the edges in the cut-set of C are removed, $|f| = 0$.

Definition 9.2. The capacity of an $s - t$ cut is defined by

$$c(S, T) = \sum\nolimits_{(u,v)\in(S\times T)\cap E} c_{uv} = \sum\nolimits_{(i,j)\in E} c_{ij} d_{ij},$$

where $d_{ij} = 1$ if $i \in S$ and $j \in T$, 0 otherwise.

The *minimum $s - t$ cut problem* is to minimize $c(S, T)$, that is, to determine S and T such that the capacity of the $S - T$ cut is minimal.

Theorem 9.3. *In a flow network, the maximum amount of flow passing from the source to the sink is equal to the minimum cut, i.e. the smallest total weight of the edges which if removed would disconnect the source from the sink.*

Definition 9.4. A *residual graph* provides a systematic way to search for forward-backward operations in order to find the maximum flow.

Given a flow network G, and a flow f on G, we define the residual graph G_f of G with respect to f as follows:

- The node set of G_f is the same as that of G.
- Each edge $e = (u, v)$ of G_f is with a capacity of $c_e - f(e)$.
- Each edge $e' = (v, u)$ of G_f is with a capacity of $f(e)$

Example 9.5. Given a flow network with unit capacity edges consisting of a directed graph $G = (V, E)$, a source $s \in V$, and a sink $t \in V$, and $c_e = 1$ for all $e \in E$, and also a parameter k. The goal is to delete k edges so as to reduce the maximum $s - t$ flow in G by as much as possible. In other words, find a set of edges $F \subseteq E$ so that $|F| = k$ and the maximum $s - t$ flow in $G' = (V, E - F)$ is as small as possible subject to this. Give a polynomial time algorithm to solve this problem. Argue (prove) that your algorithm does in fact find the graph with the smallest maximum flow.

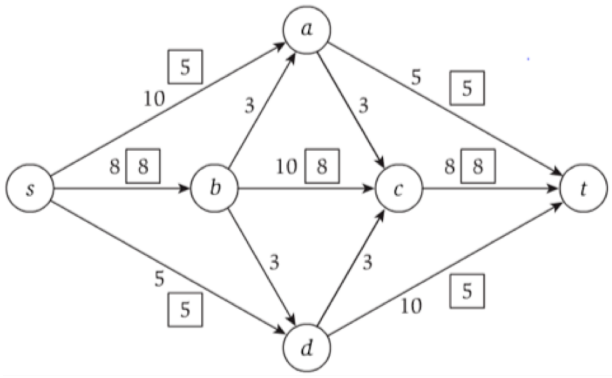
Algorithm 26 is bottlenecked by finding the minimum cut. We can use Nagamochi and Ibaraki's algorithm which finds a minimum cut in $O(|V||E| + |V|^2 \log |V|)$ [3].

Algorithm 26 Delete k edges to minimize flow

```
1: procedure REDUCE FLOW(G, k)
2:     Find the minimum cut (A, B) where s ∈ A and t ∈ B.
3:     E' ← edges that go from A to B
4:     if |E'| ≥ k then
5:         F ← any k edges in E'
6:     else
7:         F ← all edges in E' ∪ any k − |E'| other edges in G
```

Example 9.6. Consider the flow network below. The capacity of each edge appears as a label next to the edge, and the numbers in the boxes give the amount of flow on each edge. (Edges without boxed numbers have no flow being sent on them.)

(1) What is the value of the currently assigned (s, t) flow?
(2) What is the maximum (s, t) flow for this flow network?
(3) Find a minimum $s - t$ cut in the flow network and give its capacity?



(1) The value of the flow passing from s to t is 18.
(2) By applying the Ford-Fulkerson algorithm by hand, we can see the maximum flow for this network is 21.
(3) The minimum cut are the two sets: $\{s, a, b, c\}$ and $\{d, t\}$, with capacity of 21. This jives with the max-flow min-cut theorem.

Example 9.7. Consider a set of mobile computing clients in a certain town who each need to be connected to one of several possible base stations. We'll suppose there are n clients, with the position of each client specified by its (x, y) coordinates in the plane. There are also k base stations; the position of each of these is specified by (x, y) coordinates as well. For each client, we wish to connect it to exactly one of the base stations. Our choice of connections is constrained in the following ways. There is a range parameter r. A client can only be connected to a base station that is within distance r. There is also a load parameter L. No more than L clients can be connected to any single base station.

Your goal is to design a polynomial-time algorithm for the following problem. Given the positions of a set of clients and a set of base stations, as well as the range and load parameters, decide whether every client can be connected simultaneously to a base station, subject to the range and load conditions in the previous paragraph.

Algorithm 27 Check if all clients can connect to a base station

```
1: procedure CHECK ALL CONNECT(C : clients, B : base stations, r, L)
2:     V ← source s ∪ C ∪ B ∪ sink t
3:     E ← edges with capacity 1 connecting s to all C, edges with capacity 1 connecting all C to
       B if c_i are close enough, and edges with capacity L from all B to t.
4:     G ← (V, E)
5:     f ← max-flow of G
6:     if f ≥ |C| then
7:         return true
8:     else
9:         return false
```

In algorithm 27 the maximum number of clients that can connect is equal to the maximum flow value of the created graph. This algorithm is bottlenecked by the algorithm to find maximum flow, which if we use Ford-Fulkerson, is $O(|E|f)$ where f is the maximum flow value.

10. Complexity Theory

Definition 10.1. P is the set of problems that can be solved in polynomial time.

Definition 10.2. NP is the set of problems that can be verified in polynomial time.

Definition 10.3. We can *reduce* a problem A to a problem B, by showing that solving B is the same as solving A. This shows that problem B is at least as hard as problem A.

If we can reduce a problem to a known problem in P, then we know that that problem is also in P.

The shorthand for this idea of reduction is encapsulated in the symbol: $\leq_P$. Suppose we have two problems A and B, and $B \leq_P A$. Then we say that B is polynomial-time reducible to A. It also implies that A is at least as hard as B since a solution to A is a solution to B.

Definition 10.4. A problem is in NP-complete if every problem in NP can be reduced to that problem. So an NP-complete problem is as hard as the hardest problem in NP.

Theorem 10.5. *If problem A is in NP-complete and*

(1) problem B is in NP
(2) $A \leq_P B$

then B is in NP-complete

In other words, we can prove a new problem is NP-complete by reducing some other NP-complete problem to it.

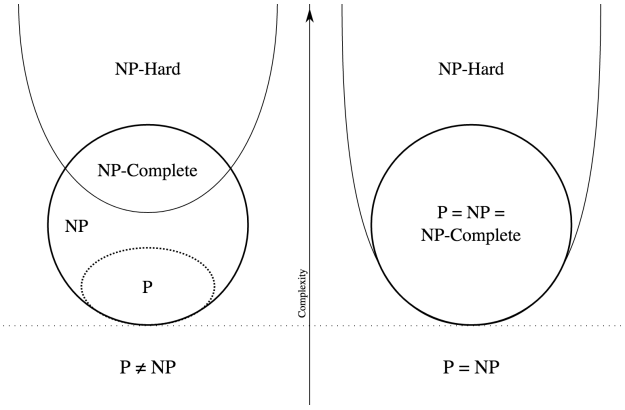


FIGURE 4. Diagram showing how P and NP are related in terms of complexity

Example 10.6. Prove that any language L such that $L \in$ NP can be decided by an algorithm that runs in time $2^{O(n^k)}$, for some constant k.

Proof. Recall that for each language L in NP, there is an algorithm A that can verify each item $x \in L$ in polynomial time, given a certificate y of length $O(|x|^c)$.

To establish if a given $x \in \{0,1\}^*$ is in a given language $L \in$ NP, we determine if there is a certificate y of length $O(|x|^k)$, for some constant k, such that an accepting algorithm A can check if $A(x,y) = 1$ or 0.

First we must determine if such a certificate y exists. To do this we need to test every $y \in \{0,1\}^*$ where $|y| = O(|x|^k)$ and evaluate $A(x,y)$. Note that there are $2^{|y|}$ possible certificates. If we evaluate $A(x,y)$ with every possible y and the given x, then we will show if there exists a certificate y for which $A(x,y) = 1$.

Thus, we have shown that there exists an algorithm to decide if a language is in NP in $2^{O(n^k)}$. □

Example 10.7. You are give a directed graph $G = (V,E)$ with weights w_e on its edges $e \in E$. The weights can be negative or positive. The Zero-Weight Cycle Problem is to decide if there is a simple cycle in G so that that sum of the edge weights on this cycle is exactly 0. Prove that Zero-Weight Cycle is NP-Complete by reducing from the subset sum problem.

The Subset Sum Problem: Given natural numbers $w_1, w_2, \ldots, w_n$ and a target number W, is there a subset of $\{w_1, w_2, \ldots, w_n\}$ that adds up to precisely W?

Proof. Suppose we are given a set of natural numbers $\{w_1, w_2, \ldots, w_n\}$. Let G be a directed graph with n vertices, labeled $0, 1, \ldots, n$, and edges (i,j) for all pairs $i < j$. Let the weights of each edge (i,j) be w_j. Also let there be edges $(j,0)$ of weight $-W$.

If there exists a subset $H \subseteq G$ such that the elements of H sum to W, then we define a cycle starting at vertex 0, going through the vertices of H, and returning to 0 on an edge $(j,0)$. The edges $(j,0)$ have weight $-W$ which cancels the sum of all other edge weights in the cycle.

If there exists a zero-weight cycle in G, then it must use the edge from $(j, 0)$ and that weight $-W$ must cancel all elements in the cycle. So excluding the edge $(j, 0)$, all other edges in the cycle must add up to W.

Thus, there is a subset of natural numbers that adds up to W if and only if G has a zero-weight cycle. Therefore, the subset sum problem $\leq_p$ the zero-weight cycle problem. Since we know the subset sum problem is in NP, it follows that the zero-weight cycle problem is in NP as well. □

Example 10.8. Suppose you're helping to organize a summer sports camp, and the following problem comes up. The camp is supposed to have at least one counselor who is skilled at each of the n sports covered by the camp (baseball, volleyball, etc.). They have received job applications from m potential counselors. For each of the n sports, there is some subset of the m applicants qualified in that sport. The question is: For a given number $k < m$, is it possible to hire at most k of the counselors and have at least one counselor qualified in each of the n sports? We'll call this the Efficient Recruiting Problem. Show that Efficient Recruiting is NP-Complete.

The Vertex Cover Problem: Given a graph G and a number k, does G contain a vertex cover of size at most k? (Recall that a vertex cover $V' \subseteq V$ is a set of vertices such that every edge $e \in E$ has at least one of its endpoints in V'.)

Proof. Suppose we are given a graph G and $k \in \mathbb{Z}^{\geq 0}$. Map each sport to an individual edge, and map the counselors to an individual vertex. A counselor is qualified in sport if and only if the corresponding edge goes to that counselor.

If there are k counselors who all together are qualified in all sports, then the vertices corresponding to those counselors have at least one edge that goes to them. Thus the counselor's vertices create a vertex cover of size k.

If there is a vertex cover of size k, then each counselor is qualified in at least one sport.

Thus, G has a vertex cover of size at most k if and only if the we can be solve the efficient recruiting problem with at most k counselors. Therefore, the vertex cover problem $\leq_p$ the efficient recruiting problem. Since we know the vertex cover problem to be in NP, it follows that the efficient recruiting problem must be in NP as well. □

11. Approximation Algorithms

Approximation algorithms are an approach to determining solutions to computationally intractable problems. They run in polynomial time and find solutions that are guaranteed to be close to optimal, yet are not guaranteed to be optimal.

References

[1] N. Touba, 'Algorithms', The University of Texas at Austin, 2016.
[2] J. Kleinberg and E. Tardos, Algorithm design. Boston: Pearson/Addison-Wesley, 2006.
[3] Computing Edge-Connectivity in Multigraphs and Capacitated Graphs, Hiroshi Nagamochi and Toshihide Ibaraki, SIAM Journal on Discrete Mathematics 1992 5:1, 54-66