Graph Algorithms

Depth First Search

DFS Algorithm Runtime $O(|V|+|E|)$

```
Algorithm DFS($G$)
  for $v \in V$ do
    explored($v$) $\leftarrow 0$
  for $v \in V$ do
    if explored($v$) $= 0$ then
      SEARCH($v$)
```

```
Algorithm SEARCH($v$)
  explored($v$) $\leftarrow 1$
  PREVISIT($v$)
  for $(v, w) \in E$ do
    if explored($w$) $= 0$ then
      SEARCH($w$)
  POSTVISIT($v$)
```

Applications

Detecting cycles	$\exists(u, v)$: postorder$(u) <$ postorder(v)
Topological sort on **DAGs**	Decreasing post-order $O(n+m)$
Strongly Conn. Components	Hint: DFS on reversed Graph, G^R

Single Source Shortest Paths

BFS Algorithm only for unweighted graphs

```
Algorithm BFS($G, s$)
  for $u \in V$ do
    explored($u$) $\leftarrow 0$
  inject($Q, s$); dist($s$) $= 0$; explored($u$) $\leftarrow 1$
  while $Q$ not empty do
    $u \leftarrow$ pop($Q$)
    for $(u, w) \in E$ s.t. explored($w$) $= 0$ do
      inject($Q, w$); $d(w) \leftarrow$ dist($u$) $+ 1$; explored($w$) $\leftarrow 1$
  return dist[]
```

Runtime $O(V+E)$

Dijkstra's Algorithm only for +ve weighted graphs

```
Algorithm DIJKSTRA'S($G, \omega, s$)
  for $v \in V$ do
    $d(v) \leftarrow \infty$
    $\pi(v) \leftarrow$NULL
  $d(s) = 0; H \leftarrow \{(s, 0)\}$          {Initialize $d(s)$ and min heap $H$}
  while $H \neq \emptyset$ do
    $v \leftarrow$ DELETEMIN($H$)          {Delete the min value from heap}
    for $(v, w) \in e$ do
      if $d(w) > d(v) + \omega(v, w)$ then
        $d(w) \leftarrow d(v) + \omega(v, w)$
        $\pi(w) \leftarrow v$
        INSERT($(w, d(w)), H$)
```

Runtime $O(E + V \log V)$ (using a binary min-heap)

Bellman-Ford Algorithm for +ve and −ve weighted graphs

```
Algorithm UPDATE($u, v$)
  if $d[v] > d[u] + length((u, v))$ then
    $d[v] = d[u] + length((u, v))$
    $\pi[v] = u$          {$\pi$ contains the previous vertex}
Algorithm BELLMAN-FORD($G, \omega, s$)
  for all $v \in V$ do
    $d[v] \leftarrow \infty$; $d[s] = 0$; $\pi[s] \leftarrow$ NULL          {Initialize}
  for $i = 1$ to $n - 1$ do
    for every $(u, v) \in E$ do
      UPDATE($u, v$)
  return $d[\cdots], \pi[\cdots]$
```

Runtime $O(E * V)$

Greedy Algorithms

Main Idea At each step, make a locally optimal choice in the hope of reaching the globally optimal solution.

Minimum Spanning Trees

Basic Properties Only weighted, connected, undirected graphs

1. A **tree** is connected, acyclic, and has $|V| - 1$ edges (any two implies the third).
2. **Cut Property**: For any cut of a connected, undirected graph, the minimum weight edge that crosses the cut belongs to MST.

Prim's Algorithm Runtime $O(|E| \log |V|)$ (Fibonacci-heaps)

1. Start with an arbitrary vertex and greedily add closest vertices
2. Similar to Dijkstra's algo, but $dist[v]$ is the weight of the edge connecting v to the MST instead of the distance from s.

Kruskal's Algorithm Runtime $O(|E| \log |V|)$ (Union-Find)

1. Sort edges in ascending order of weight.
2. Repeatedly add the lightest edge that does not create a cycle until we have $|V| - 1$ edges.

Set Cover

Given $X = \{x_1, \ldots, x_n\}$, and a collection of subsets $\mathcal{S}$ of X such that $\cup_{s \in \mathcal{S}} S = X$, find the subset $\mathcal{T} \subseteq \mathcal{S}$ s.t. sets of $\mathcal{T}$ cover X.

Algorithm Runtime $O(|U|)$

Greedily choose the set that covers the most number of the remaining uncovered elements at each iteration. **Claim**: Let k be the size of the smallest set cover for the instance $(X, \mathcal{S})$. Then the greedy heuristic finds a set cover of size at most $k \ln n$.

Note Not always optimal; achieves $O(\log n)$ approximation ratio.

Horn Formula

Algorithm Set all variables to false and greedily set ones to be true when forced to

Runtime Linear in the length of the formula (# of literals)

Huffman Coding

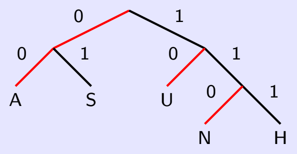
Algorithm Runtime $O(n \log n)$

Find the best encoding by greedily combining the two least frequently appearing items. Optimal in terms of encoding one character at a time.

1. Sort letters on frequency or probability
2. Construct tree by combining the two least frequent letters at a time

Example ANUSHA

Probabilities: A = 0.3, S = 0.3, U = 0.2, N = 0.1, H = 0.1



Divide and Combine

Main Idea Divide the problem into smaller pieces, recursively solve those, and then combine their results to get the final result.

Searching for max and min in a list

Naive approach takes $2n - 2$ comparisons. ($n - 1$ comparisons for finding the maximum and $n - 1$ comparisons for finding the minimum)

1. *Divide* the list into two sublists of equal size.
2. Recursively *solve* for the max and min for each sublist
3. *Combine* these two subproblem solutions by comparing the two minimums and comparing the two maximums.

Runtime Base case: $T(2) = 1$, and $T(n) = 2T(n/2) + 2$, By induction, $T(n) = 3n/2 - 2$. (25% speedup over the naive algorithm).

Closest points in a plane

Naive approach takes $O(n^2)$. Using divide and combine,

1. Sort points by x-coords and *divide* the list into 2 equal halves
2. Recursively *solve* for the closest pair of points in each half
3. Proceed by casework to *combine* these solutions.

Runtime The closest pair of points in the overall list is either one of the two pairs found recursively in the second step, or one point from each half. Let d denote the distance between the closer pair among the two found in the second step. If there is an even closer pair (p, q) where each half contains one of the points, then:

1. Each of p and q is within d of the median x-coordinate.
2. p and q are within d of each other vertically.

For any point p, there are at most 6 points q, located in a $d \times 2d$ rectangle, satisfying the two points above.

1. For each such point p, we compute its distance to q for at most 6 choices of q from the opposite half: $O(n)$ comparisons.
2. We return the closest pair among all of these comparisons and the subproblem solutions from step 2.
3. Step 3 is dominated by the cost of sorting by y-coordinate, which could be as large as $O(n \log n)$

So the recursion is $T(n) = 2T(n/2) + O(n \log n)$ and the runtime is $T(n) = O(n \log^2 n)$ (from Case 2 of Master Theorem)

Integer Multipliication

	Grade school multiplication	$\Theta(n^2)$
Karatsuba	3 products on $n/2$ digits	$\Theta(n^{\log_2 3}) = \Theta(n^{1.59})$
	5 products on $n/3$ digits	$\Theta(n^{\log_3 5}) = \Theta(n^{1.46})$

Matrix Multiplication

Strassen Divide into 4 submatrices of size $n/2$ by $n/2$. Results in 7 matrix multiplications, so $T(n) = 7T(n/2) + \Theta(n^2)$.

Runtime $O(n^{\log_2 7})$

Dynamic Programming

Main Idea Maintain a lookup table of correct solutions to sub-problems and build up this table towards the actual solution. **Steps**

1. Define sub-problems and recurrence to solve sub-problems
2. Combine with **reuse**
3. Runtime and space analysis

Fibonacci Sequence

Naive approach uses recursion and is exponential.

```
Algorithm NaiveFibonacci($n$)
  if $n = 0$ or $n = 1$ then
    return 0
  return NaiveFibonacci($i-1$) + NaiveFibonacci($i-2$)
```

The recursion equation is $T(n) = T(n-1) + T(n-2) + 1$. We can inductively show **Runtime** $O(2^{\Theta(n)})$

```
Algorithm DynamicFibonacci($n$)          {a.k.a Iterative Fibonacci}
  fib = [0, 1]                           {1st two fibonacci numbers}
  for $i = 2$ to $n$ do
    fib.append(fib[$i-1$] + fib[$i-2$])
  return fib[$n$]
```

Runtime $O(n)$

Traveling Salesman

The NP-hard Traveling Salesperson Problem (TSP) asks to find the shortest route that visits all vertices in a graph exactly once and returns to the start. The naive algorithm tries all $n!$ permutations of the n vertices, and to compute the cost of each, choosing the shortest.

Following is the Dynamic Programming approach:

1. Sub-problem: $C(S,t) =$The minimum-cost path starting at x and ending at t going through all vertices in S.
2. Recurrence for TSP:

$$C(S,t) = \begin{cases} w(x,t) & \text{if } S = \{x,t\}, \\ \min_{t' \in S, t' \notin \{x,t\}} C(S - \{t\}, t') + w(t',t) & \text{otherwise.} \end{cases}$$

Runtime $O(n^2 2^n)$

Longest Common Subsequence (LCS)

Find a maximum-length common subsequence of $X = \{x_1, \ldots, x_m\}$ and $Y = \{y_1, \ldots, y_n\}$. **Runtime and Space** $O(nm)$

1. Sub-problem: Let $L(i,j)$ be the length of an LCS of X_i and Y_j.
2. Recurrence equation from the optimal substructure:

$$L(i,j) = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0, \\ L(i-1,j-1)+1 & \text{if } i,j > 0, \text{ and } x_i = y_j, \\ \max(L(i,j-1),\ L(i-1,j)) & \text{if } i,j > 0, \text{ and } x_i \neq y_j \end{cases}$$

Longest Increasing Subsequence (LIS)

Eg. (3, 4, 5, 9) is the LIS of (3, 6, 4, 1, 5, 9).

Algorithm **Runtime** $O(n^2)$ **Space** $O(n)$

```
Algorithm LIS($A = (a_i)$) {$A$ is an integer array}
  for $k = 1$ to $n$ do
    $L[k] = 1$
    for $j = 1$ to $k-1$ do
      if $a_j < a_k$ then
        $L[k] = \max(L[k], L[j]+1)$
  return $\max\{L[0], \ldots, L[n-1]\}$
```

Rod Cutting

Find the maximum amount of money by cutting up a rod of length n and selling its pieces, given that a piece of length i is worth p_i dollars.

1. First, cut a piece off the left end of the rod, and sell it.
2. Then, find the optimal way to cut the remainder of the rod.

Runtime $O(n^2)$. The recurrence equation:

$$r(n) = \begin{cases} 0 & \text{for } n = 0 \\ \max_{1 \le i \le n} (p(i) + r(n-i)) & \text{for } n > 0 \end{cases}$$

Edit Distance

Find the minimum number of operations required to transform string $A[1 \ldots n]$ into string $B[1 \ldots m]$. The possible operations are:

1. Insert: Insert a character into A
2. Delete: Delete a character from A
3. Replace: Replace a character from A with another

If last characters of A and B are different, then that difference must have come about through either an insert, delete or replace.

Algorithm **Runtime and Space** $O(nm)$

1. Sub-problem: Let $D(i,j)$ represent the edit distance between $A[1 \ldots i]$ and $B[1 \ldots j]$
2. Recurrence equation: (Base cases: $D(i,0) = i, D(0,j) = j$)

$$D(i,j) = \min \begin{cases} D(i-1,j)+1, & \text{(delete)} \\ D(i,j-1)+1, & \text{(insert)} \\ D(i-1,j-1)+1 \text{ if } i=j, 0 \text{ otherwise} & \text{(replace)} \end{cases}$$

3. Return $D(n,m)$.

Union-Find

```
Algorithm Find($x$) {Using path compression}
  if $x != p(x)$ then
    $p(x) =$ find($p(x)$)
  return $p(x)$
```

Path compression allows for an amortized cost of $O(1)$ for the Union(x,y) and Find(x) operations.
Other important functions: MakeSet(x), which makes a disjoint dataset for element x, and Union(x,y), which makes the parent of one of the elements the parent of the other (based on rank or size).

All Pairs Shortest Paths

Floyd-Warshall Algorithm **for +ve and −ve weighted graphs**

Runtime $O(n^3)$ **Note** Does not work with negative cycles

The shortest path from u to v must have passed through one of the nodes w with the edge $w \to v$. Find the best previous node w in this shortest path by taking the minimum over all such possibilities of going from u to w in the shortest way possible and then finally $w \to v$.

1. Sub-problem: Let $D^{(k)}_{(i,j)}$ represent the shortest path between i and j using only nodes in $[1 \ldots k]$
2. Recurrence equation:

$$D^{(k)}_{(i,j)} = \begin{cases} w_{ij} & \text{if } k = 0 \\ \min(D^{(k-1)}_{(i,j)},\ D^{(k-1)}_{(i,k)} + D^{(k-1)}_{(k,j)}) & \text{if } k \ge 1 \end{cases}$$

3. Return $D^{(n)}_{(i,j)}$.

Fundamentals

Master Theorem

$$T(n) = a \cdot T\left(\frac{n}{b}\right) + dn^c$$

for $a \ge 1, b \ge 2, d, c \ge 0$.

Case 1	$c < \log_b a$	$T = \Theta(n^{\log_b a})$
Case 2	$c = \log_b a$	$T = \Theta(n^c \log n)$
Case 3	$c > \log_b a$	$T = \Theta(n^c)$

Asymptotics

Let f and g be non-negative functions. Then $f(n)$ is,

Relation	If and only if	$\lim_{n\to\infty} \frac{f(n)}{g(n)}$
$O(g(n))$	$\exists c, n_0 : 0 \le f(n) \le cg(n)$ for $n \ge n_0$	$\neq \infty$
$o(g(n))$	$\forall c, \exists n_0 : 0 \le f(n) < cg(n)$ for $n \ge n_0$	$= 0$
$\Omega(g(n))$	$\exists c, n_0 : 0 \le cg(n) \le f(n)$ for $n \ge n_0$	$\neq 0$
$\omega(g(n))$	$\forall c, \exists n_0 : 0 \le cg(n) < f(n)$ for $n \ge n_0$	$= \infty$
$\Theta(g(n))$	$f(n) = O(g(n))$ and $g(n) = O(f(n))$	$\neq 0, \neq \infty$

Implications

$f = o(g) \Rightarrow f = O(g)$
$f = \omega(g) \Rightarrow f = \Omega(g)$

Equivalences

$f = O(g) \iff g = \Omega(f)$
$f = o(g) \iff g = \omega(f)$
$f = O(g)$ and $f = \Omega(g) \iff f = \Theta(g)$

Sterling approximation $\binom{n}{k} \approx \frac{n^k}{k!}$

Min Heaps

Representation Visually a tree. Implemented as array, $A[0, \ldots, n-1]$ be an array where $A[0]$ is the root and for any i-th element, its parent, left and right children are $\lfloor i/2 \rfloor, 2i, 2i+1$ respectively.

Heap Property The parent element is smaller than its children.

Operation	Description
Insert(a)	Let $A[n] = a$ and MinHeapify(n)
DeleteMin	Let $A[0] = A[n-1]$ and MinHeapify(0)
MinHeapify	Repeatedly swap $A[i]$ with its parent until the heap property is restored

Runtime All operations take $O(\log n)$ with binary heaps.

Hashing

Balls into Bins and Hashing

Consider n balls that are randomly thrown into m bins. The probability that all of the n balls land in different bins, given that $n < m$ is, $p = \left(1-\frac{1}{m}\right)\left(1-\frac{2}{m}\right)\cdots\left(1-\frac{n-1}{m}\right) \leq \exp\left(-\frac{1}{m}\sum_{i=1}^{n-1} i\right)$ The above probability simplifies into $p \leq e^{-\Omega(n^2/m)}$. **Number of empty bins** For the first bin to be empty, it has to be missed by all n balls. Since each ball hits the first bin with probability $\frac{1}{m}$, the probability that the first bin remains empty is, $\left(1-\frac{1}{m}\right)^n \approx e^{-n/m}$. Since the same argument holds for all bins, on average an $e^{-n/m}$ fraction of the bins will remain empty. **False positive probability** The probability the hash function returns 1 when the element is not present in the array is $1 - e^{-n/m}$.

Bloom Filters

A Bloom filter is an array A of m bits representing a set $S = \{x_1, x_2, \ldots, x_n\}$ of n elements.
k independent hash functions, $h_1, h_2, \ldots, h_k$ with range $\{1, 2, \ldots, m\}$.
For eacch $x_i \in S$, the bit $h_j(x_i)$ in the array A is set to 1, for $1 \leq j \leq k$. **False positive probability**: The probability of false positive is the probability that all hash functions return 1. With m bins and kn balls analogy, the probability of false positive is, $P \approx (1 - e^{-kn/m})^k$ **Optimal k** $k \approx \frac{m}{n} \ln 2$.

Counting Bloom Filters

Counting Bloom filters are Bloom filters that support deletions. Each entry in a Counting Bloom filter is a small counter (instead of a single bit). When an item is inserted into the set, we increment the corresponding counters by 1. When an item is deleted from the set, we decrement the corresponding counters by 1 The counter size must be large enough to avoid overflow. Typically 4 bit counters are sufficient. **Counter Overflow Probabilitys** Consider a set of n elements, k hash functions and m counters. Let $c(i)$ be the count for the i-th counter. So, $P[c(i) = j] = \binom{nk}{j}\left(\frac{1}{m}\right)^j\left(1-\frac{1}{m}\right)^{nk-j}$, so $P[c(i) \geq j] \leq \binom{nk}{j}\frac{1}{m^j}$. A lose upper bound: $P[c(i) \geq j] \leq \left(\frac{enk}{jm}\right)^j$.

Fingerprinting by Modular Arithmetic

We want to find a pattern string P in a long document D. How can we do it quickly and efficiently? The idea of pattern matching using fingerprinting is to pick an appropriately large prime number q, so that $H(P) = P \pmod{q}$.
Naively computing $H(D_i)$ from scratch each time to check a document of length n takes $O(nk)$ time.
So, we use the simple identity: $D_i = 10D_{i+1} + d_i - 10^k \cdot d_{i+k}$.
Applying the hash function, we find,
$H(D_i) = D_i \pmod{q} = (10D_{i+1} + d_i - 10^k \cdot d_{i+k}) \pmod{q}$,
so $H(D_i)$ can be computed from $H(D_{i+1})$ in $O(1)$ time.

P, NP, NP-Hard and NP-Completeness

P versus NP

P: set of YES/NO decision problems that can be solved in polynomial time: runs in $O(n^k)$ steps for $k > 0$. (Ex: Shortest Paths, MSTs)
NP: set of YES/NO decision problems that can be verified n polynomial time by providing a polynomial length certificate (Ex: compositeness, 3-SAT, independent set, vertex cover)
NP-Hard: A problem A is NP-hard if a polytime algorithm for A would imply a polytime algorithm for every problem in NP. **To prove a problem A is NP-hard, reduce a known NP-hard problem to A**
NP-Complete: A problem is NP-complete if it is both NP-hard and is an element of NP. Informally, NP-complete problems are the hardest problems in NP. A polytime algorithm for even one NP-complete problem would imply a polytime algorithm for every NP-complete problem.

Proving a problem class

In P: Come up with a polynomial time algo & prove its correctness
In NP: State a polynomial-length witness & discuss how to verify that it's correct in polynomial time
In NP-Hard: Reduce a known NP-hard problem to this one

1. Show your reduction in polynomial time
2. Prove that your reduction is correct (one problem returns TRUE if and only if the other does)

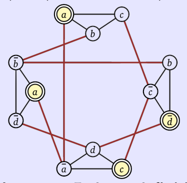
In NP-Complete: Prove that it is both in NP and is NP-Hard

Example: Prove Max Indep. Set is NP-Hard

We use a reduction from 3-SAT. Given an arbitrary 3CNF formula Φ, we construct a graph G as follows. Let k denote the number of clauses in Φ. The graph G contains exactly $3k$ vertices, one for each literal in Φ. Two vertices in G are connected by an edge if and only if either

1. they correspond to literals in the same clause, or
2. they correspond to a variable and its inverse.

For example, the formula $(a \vee b \vee c) \wedge (b \vee \neg c \vee \neg d) \wedge (\neg a \vee c \vee d) \wedge (a \vee \neg b \vee \neg d)$ is transformed into the following graph:



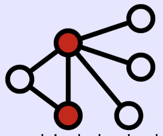
Each independent set in G contains at most one vertex from each clause triangle, because any two vertices in each triangle are connected. Thus, the largest independent set in G has size at most k.
We claim that G contains an independent set of size exactly k if and only if the original formula Φ is satisfiable.
Assume Φ is satisfiable and choose a subset S of k vertices in G that contains exactly one vertex per clause triangle, such that the corresponding k literals are all T. Then S is an independent set of size k. (because no 2 vertices in S are connected by a triangle edge, and no two edges in S are connected by a negation edge).
Assume G contains an independent set S of size k. So, each vertex in S must lie in a different clause triangle. Suppose we assign the value T to each literal in S; because contradictory literals are connected by edges, this assignment is consistent. Because S contains one vertex in each clause triangle, each clause in Φ contains (at least) one TRUE literal. We conclude that Φ is satisfiable.

Approximation Algorithms

Minimum Vertex Cover $\alpha = 2$

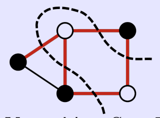
Let $G = (V, E)$ be a graph. The min vertex cover is the min. set of vertices that every edge has an endpoint in it (so the edge is *covered*).

```
Algorithm VC
  $C \leftarrow \emptyset$
  while $E$ is not $\emptyset$ do
      Arbitrarily pick $e = (u, v) \in E$
      $C \leftarrow C \cap \{u, v\}$
      Remove all edges incident to $u, v$ from $E$
```



The above algorithm repeatedly chooses an edge, and includes both of its endpoints into the cover. It throws the vertices and its adjacent edges out of the graph, and continues.
Algorithm VC has approximation ratio 2.
Proof. Clearly VC is a vertex cover. Next we prove the ratio. Suppose we pick k edges during the execution of algorithm VC. These k edges form a matching (they don't share vertices). So, the optimal vertex cover should contain at least one vertex from each of these k edges. So $OPT \geq k$. But, algorithm VC's output has size at most $2k$. □

Maximum Cut $\alpha = 1/2$



Given a graph $G = (V, E)$, find two disjoint subsets S and T such that $S \cap T = \emptyset, S \cup T = V$, and the number of $(u, v) \in E$ that cross the cut is maximized.
We discuss two **2-approximation algorithms**.
The **first algorithm** randomly assigns each $u \in V$ to either S or T, such that each edge has a 1/2 probability of crossing the cut, leading to an average outcome of 1/2 of the edges in the graph crossing the cut, which is 1/2 of the optimal solution (every edge crosses the cut).
The **second algorithm** begins with all vertices on one side of the cut, and deterministically switches a vertex to the other side if it increases the number of edges crossing the cut until the size of the cut can no longer be increased in this manner. This type of algorithm is a local search algorithm, and also results in a 1/2-approximation.

Euclidean Traveling Salesman Problem $\alpha = 2$

We are given n points (cities) in the x-y plane, and we seek the tour (cycle) of minimum length that travels through all the cities.
A greedy algorithm is to run DFS on the MST and skipping vertices that have already been visited (i.e. shortcutting). This will be within a factor of 2 of optimal, since running DFS on the MST visits each edge at most twice, and shortcutting a node can only reduce the travel time due to the triangle inequality, plus we know that the optimal tour is at least the size of the MST since any tour must be a spanning tree (with an additional edge for the final leg back to the origin). That is, if T is the length of the MST, then we have: $\text{tour}_{\text{greedy}} \leq 2T(\text{after shortcutting}) \leq 2\text{OPT}$,

Closest String $\alpha = 2$

Given n binary strings $s_1, \ldots, s_n$, each of length m, find a new string t, such that $d = \max_i d(t, s_i)$ is minimized. Note that d is the Hamming distance.
2-approximation solution: Return s_1. Because of triangle inequality, we find, $d(s_1, s_i) \leq d(t_{\text{OPT}}, s_1) + d(t_{\text{OPT}}, s_i) \leq 2d$.

Network Flows

The Ford-Fulkerson Method

Residual Capacity The capacity minus the flow.

```
Algorithm Ford-Fulkerson
  Input: Graph $G = (V, E)$ with capacities $c : E \to \mathbb{R}^{\geq 0}$,
  Initialize $f : E \to \mathbb{R}^{\geq 0}$ to 0
  Define Residual Graph $G_R := G$
  while $\exists$ Path $P$ from $s$ to $t$ in $G_R$ do
      Compute bottleneck capacity $\Delta$ of $P$
      Augment $f$ by $\Delta$ on edges of $P$
      Compute new residual graph $G_R$ from $G$, $c$ and update $f$
  Return: $f$
```

If f is a flow in a flow network $G = (V, E)$ with source S and sink T,then the following conditions are equivalent:

1. f is a maximum flow in G
2. The residual network G_f contains no augmenting paths.
3. $|f| = c(S, T)$ for some cut (S, T) of G.

Only the following edges can be used for augmentation:

1. Non-null forward edges
2. Non-empty backward edges

Runtime

1. Ford-Fulkerson has a run-time of $O(Ef)$ because each time we augment a path, we increase the total flow, so at worst the number of times we find an augmenting path is $O(f^*)$, where f^* is the value of the max flow. We can find an augmenting path in $O(E)$ with DFS.
2. **Edmonds-Karp algorithm**: If we use BFS instead of find our augmenting paths, then can be proved that the run-time is $(O(VE^2)$, which does not depend on f^* anymore.

Max Flow Min Cut Theorem For every graph G with capacities c the maximum flow on G with capacities c equals the minimum cut in G with capacities c.
Proof: A cut is a partitioning of the vertices V into S and $V - S$. The weight of the cut is the sum of all the edge weights crossing S and V-S. We know s and t must be separated in the cut.
Min Cut $\geq$ Max Flow Reason: If you have a particular cut, you know that all the units of flow must at one point pass between S and V-S. So, if the weight of the cut is c, then at most c units of flow could pass between s and t. The minimum s-t cut is a bottle neck for the flow.
Min Cut $\leq$ Max Flow Reason: At the end of Ford-Fulkerson, I can look at all the nodes that can be reached from s and all the nodes that can reach t in the residual graph. There must not be a path from s to t or else the algorithm has not terminated. Therefore, this partitions the vertices into S, those that can be reached from s in this final graph, and V-S, the remaining vertices. This is some cut of the graph and the weight of the cut is the max flow, so in particular the minimum cut can be no heavier than this particular cut.
Therefore, the min s-t cut and max-flow are equal.

Linear Programming of Network Flow Let f_{uv} be the amount of flow sent along the edge (u, v) and c_{uv} be the capacity of the edge (u, v). We want to maximize the flow into the sink subject to the following conditions:

1. **Capacity**: $f_{uv} \leq c_{uv}$.
2. **Conservation**: For every vertex w besides $w = s$ and $w = t$, we must have that $\sum_{u,w} f_{uw} - \sum_{w,v} f_{wv} = 0$.
3. **Nonnegativity**: $f_{uv} \geq 0$.

Randomized Algorithms

Primality Testing

Fermat's Little Theorem $a^{n-1} \equiv 1 \pmod n$ if n is prime and is relatively prime with a.

Rabin's Test If n is prime, then only solutions to $a^2 \equiv 1 \pmod n$ are $a = 1, -1$. Let $n - 1 = 2^t u$ for odd u.
Consider $a^u, a^{2u}, a^{4u}, \ldots, a^{n-1} \pmod n$, and verify whether they have the following property:

$$a^{2^{i-1}u} \not\equiv \pm 1 \pmod n \quad \text{and} \quad a^{2^i u} \equiv 1 \pmod n.$$

If we see the above property, we have encountered a non-trivial square root of $1 \pmod n$. Rabin primality test says that only composite numbers have non-trivial square roots of $1 \pmod n$. The value of a, for which we found a non-trivial square root of $1 \pmod n$ is the witness for the compositeness of n.

Example **True or False?** Let n be an odd integer and write $n - 1 = 2^t \cdot u$ with u odd. Then, if for some a, $a^{2^i u} \equiv 1 \pmod n$, but $a^{2^{i+1}u} \not\equiv \pm 1 \pmod n$, then n is composite. **False** - It is the wrong direction for Rabin-Miller Testing.

RSA Algorithm Alice wants to securely send message to Bob, but Eve is listening.
Bob's pubic key: (n, e), $n = pq$ and e is coprime to $(p - 1)(q - 1)$.
Bob's private key: $d = e^{-1} \pmod{(p - 1)(q - 1)}$
Encryption: $e(x) = x^e \pmod n$, **Decryption**: $d(y) = y^d \pmod n$
Claim: $d(e(x)) = x^{1+k(p-1)(q-1)} = x \pmod n$.

Example

1. Choose $p = 3$ and $q = 11$, so $n = pq = 33$
2. Compute $\phi(n) = (p - 1)(q - 1) = 20$
3. Choose the encryption key e such that $1 < e < \phi(n)$ and e and $\phi(n)$ are coprime. Let $e = 7$.
4. Compute the decryption key, d, which is the multiplicative inverse of $e \pmod{\phi(n)}$. We use Extended Euclidean algorithm to find the multiplicative inverse. Hence $d = 3$.
5. Public key is $(e, n) = (7, 33)$.
6. Private key is $(d, n) = (3, 33)$.

Let us say that Alice wants to send message $m = 2$ to Bob. She uses Bob's public key, $(e, n) = (7, 33)$ to encrypt the message as $m^e \pmod n = 2^7 \pmod{33} = 29$, and sends 29 to Bob.
Bob decrypts the encrypted message using his private key, $(d, n) = (3, 33)$ and obtains $m = 29^3 \pmod{33} = 2$.

Multiplicative Modulo Inverse

The decryption key in the RSA algorithm uses the multiplicative inverse of e with respect to $\phi(n) = (p - 1)(q - 1)$. This can be easily computed using the Extended Euclidean Algorithm.

Example Find the multiplicative inverse of 7 with respect to 31.

We are asked to find y where $31x + 7y = 1$. We fill-up the a, b, q columns of the table top-down, then find $d = 1$, and then fill-up the x and y columns bottom-up from the base case as shown.

a	b	q	x	y
31	7	4	-2	9
7	3	2	1	-2
3	1	3	0	1
1	0	d	1	0

Reading off the x and y values from the top row, we find $x = -2$ and $y = 9$. We confirm that $37(-2) + 7(9) = 1$, and therefore the multiplicative inverse of 7 with respect to 31 is 9.

Extended Euclid's Algorithm

```
Algorithm EXTENDED-EUCLID$(a, b)$)
  if $b = 0$ then
      return$(a, 1, 0)$
  Compute $q$ such that $a = bq + (a \pmod b)$)
  $(d, x, y)$ =Extended-Euclid$(b, a \pmod b)$)
  return (d, y, x - qy)
```

Random Walks for SAT

2-SAT **WalkSat Algorithm**

1. At every step pick a random unsatisfied clause and randomly flip one of the variables
2. For each step, in the worst-case, there's a 50-50 chance of moving 1 step closer to true satisfying assignment
3. Analysis of simple random walk shows that $O(n^2)$ steps suffice with high probability.

3-SAT

1. Using the above algorithm, each step gives 1/3 chance of moving closer to true satisfying assignment
2. Analysis shows that exponentially many steps are now needed
3. Can get better constant in exponent than brute force by restarting - achieves $\approx (4/3)^n$ time.

Linear Programming

Two Player Games

Consider a *mixed strategy* game between two players, X and Y, where each player can choose between 3 actions, namely A, B, and C. Following is the game matrix with X as the row player and Y as the column player.

$$\begin{array}{c|ccc} & A & B & C \\ \hline A & 3 & 1 & -1 \\ B & -2 & 3 & 2 \\ C & 1 & -2 & 4 \end{array}$$

Let X play actions A, B, and C with probabilities x_1, x_2 and x_3 respectively. Player X wants to choose these probabilities in such a way that no mater what her opponent (Y) chooses, X can be guaranteed to get a payoff of at least x_4. Hence, we have the following LP shown on the left below that **maximizes** the payoff for the row player X. We can also write the dual LP, which is the column player, B's **minimization** problem, which is shown on the right below.

$$\begin{aligned} &\max x_4 \\ x_1, x_2, x3 &\geq 0 \\ x_1 + x_2 + x_3 &= 1 \\ 3x_1 - 2x_2 + x_3 &\geq x_4 \\ x_1 + 3x_2 - 2x_3 &\geq x_4 \\ -x_1 + 2x_2 + 4x_3 &\geq x_4 \end{aligned} \qquad \begin{aligned} &\min y_4 \\ y_1, y_2, y_3 &\geq 0 \\ y_1 + y_2 + y_3 &= 1 \\ 3y_1 + y_2 - y_3 &\leq y_4 \\ -2y_1 + 3y_2 + 2y_3 &\leq y_4 \\ y_1 - 2y_2 + 4y_3 &\leq y_4 \end{aligned}$$

When $x_4 \leq y_4$, we have a **weak duality**, while when $x_4 = y_4$, we have a strong duality.

Canonical Form of a linear program is given by

maximize $c^T x$ **subject to** $Ax \leq b$ **and** $x \geq 0$

Simplex Algorithm Considers "polytope" of vertices. Jumps from vertex to neighboring vertex till local optimum is found. This is a local search algorithm that runs extremely fast.