CS 6515 Introduction to Graduate Algorithm Lecture Notes

Jie Wu, Jack

Fall 2023

1 Overview

Course outline:

dynamic programming:

randomized algorithms: including cryptography and hashing

divide and conquer: including FFT

graph algorithms: using DFS for connectivity problems on directed graphs, including applications such as two-step problem, the page rank algorithm, algorithms for the maxflow problem

linear programming:

NP-completeness:

2 Dynamic Programming: FIB - LIS - LCS

2.1 Fibonacci Numbers

2.1.1 Recursive algorithm

Based on the recursive relation $F_n = F_{n-1} + F_{n-2}$ for $n > 1$. To analyze the running time define $T(n)$ = the number of steps to compute F_n. It satisfies the following recursive relation (where $O(1)$ comes from the base cases $F_0 = 0$ and $F_1 = 1$)

$$T(n) \leq O(1) + T(n-1) + T(n-2) \implies T(n) \geq F_n \approx \frac{\phi^n}{\sqrt{5}} \text{ where } \phi = \frac{1+\sqrt{5}}{2} \approx 1.618$$

which grows exponentially with n. The exponential order of growth comes from the fact that we compute F_{n-k} k times (draw a tree based on the recursive relation to see), i.e. we recompute the solution to small subproblems many times. To get around that we will flip the order and compute the small subproblems first.

2.1.2 DP algorithm

Create an array $F[n]$ to store n Fibonacci numbers. For $i : 2 \to n$ do $F[i] = F[i-1] + F[i-2]$. Each step of the $O(n)$ for-loop takes $O(1)$ time. Therefore the total time of the for-loop is $O(n)$. Note that

- there is no recursion in DP algorithm although we use the recursive relation to design it
- we can use memoization the save the result of the solved subproblems, although we are not going to use it at all in our course

2.2 Longest Increasing Subsequence

input: n numbers $a_1, a_2, \ldots, a_n$

goal: find the longest increasing subsequence in $a_1, a_2, \ldots, a_n$

Note that subsequence is different from substring. Substring requires consecutive elements (there are at most $O(n^2)$ substrings because we can specify a substring by a start index and an end index), whereas subsequence does not require consecutive thus can skip some element(s).
The recipe for designing a dynamic programming algorithm is as follows:

define the subproblem in words: the first attempt is always to use the identical problem on a prefix of the input—let $L[i]$ be the length of LIS on $a_1, a_2, \ldots, a_i$

state the recursive solution: we want the LIS with the minimum ending character, because the smallest ending character gives us the the most possibilities for appending a new character to the end ($\Rightarrow$ this actually is not reflected in the pseudocode, nor it is necessary because length comparison takes precedence over element comparison), for which we need to know the LIS ending at every element in the array

The revised recipe is

define the subproblem in words: let $L[i]$ be the length of LIS on $a_1, a_2, \ldots, a_i$ that ends at a_i

state the recursive solution: if $a_j \geq a_i$ then a_i cannot be added to any increasing sequence ending at a_j, otherwise it can be added thus we have

$$L[i] = 1 + \max_{1 \leq j < i} \{L[j] : a_j < a_i\}$$

Note the analogy between dynamic programming and induction proof. By adding the constraint we are essentially solving a stronger problem by strengthening the hypothesis and then using the solution to solve the weaker problem.
The pseudocode that implements the DP solution for LIS is as follows:

```
LIS(a_1, a_2, ..., a_n) :
    for i = 1 → n:
        L[i] = 1
        for j = 1 → i − 1:
            if a_j < a_i and L[i] < 1 + L[j]
                then L[i] = 1 + L[j]
    max = 1
    for i = 2 → n:
        if L[i] > L[max]
            then max = i
    return L[max]
```

Note that different from Fibonacci number LIS does not necessarily end at the last element. Hence we need another for-loop to look for the maximum length.
Since we have a nested for-loop and the if-else statement within the nested loop takes $O(1)$ time, the running time is $O(n^2)$.

2.3 Longest Common Subsequence

input: two strings $X = x_1x_2\cdots x_n$ and $Y = y_1y_2\cdots y_n$ (for simplicity we assume they have the same length) $\Rightarrow$ nowhere do we use this simplification thus it can be relaxed as in DPV 6.11

goal: find the length of the longest string which is a subsequence of both X and Y

Following the recipe for designing a dynamic programming algorithm:

define the subproblem in words: let $L[i]$ be the length of LCS in $x_1x_2\cdots x_i$, $y_1y_2\cdots y_i$

state the recursive solution: if $x_i = y_i$ then $L[i] = 1 + L[i-1]$; if $x_i \neq y_i$ then $L[i]$ does not include x_i or y_i or both
if excluding both then $L[i] = L[i-1]$; if excluding either then the resulted subproblem (of unequal lengths) does not fall into the category of initial subproblem definition, which calls for the modification of the subproblem definition to allow different lengths

The revised recipe is

define the subproblem in words: let $L[i,j]$ be the length of LCS in $x_1x_2\cdots x_i$, $y_1y_2\cdots y_j$

state the recursive solution: first the base cases: $L[i,0] = 0$ and $L[0,j] = 0$
if $x_i \neq y_j$ then $L[i,j]$ excludes x_i or y_i or both, we don't need to consider dropping both because we can drop one by one, hence we are left with only two cases: if dropping x_i then $L[i,j] = L[i-1,j]$; if dropping y_j then $L[i,j] = L[i,j-1]$, combining the two cases $L[i,j] = \max\{L[i-1,j], L[i,j-1]\}$
if $x_i = y_j$ then $L[i,j]$ includes x_i or y_i or both: if dropping x_i i.e. including only y_i then $L[i,j] = L[i,j-1]$, if dropping y_j i.e. including only x_i then $L[i,j] = L[i-1,j]$, if including both then $L[i,j] = 1 + L[i-1,j-1]$, combining the three cases $L[i,j] = \max\{L[i-1,j], L[i,j-1], 1 + L[i-1,j-1]\}$, actually it can be simplified to $L[i,j] = 1 + L[i-1,j-1]$, because any LCS we obtain by matching x_i/y_j to some earlier y/x element can be obtained by matching x_i/y_j to y_j/x_i thus $1 + L[i-1,j-1] \geq L[i-1,j]$ and $1 + L[i-1,j-1] \geq L[i,j-1]$
to summarize

$$L[i,j] = \begin{cases} \max\{L[i-1,j], L[i,j-1]\} & x_i \neq y_j \\ 1 + L[i-1,j-1] & x_i = y_j \end{cases}$$

The pseudocode that implements the DP solution for LCS is as follows:

```
LCS(X, Y) :
    for i = 0 → n:
        L[i, 0] = 0
    for j = 0 → n:
        L[0, j] = 0
    for i = 1 → n:
        for j = 1 → n:
            if x_i = y_j
                then L[i, j] = 1 + L[i − 1, j − 1]
            else
                then L[i, j] = max{L[i − 1, j], L[i, j − 1]}
    return L[n, n]
```

Since we have a nested for-loop and the if-else statement within the nested loop takes $O(1)$ time, the running time is $O(n^2)$.

Once we know the length of the LCS, to look up the corresponding common subsequence we can start by matching the last cell of the DP table and then trace back, first diagonally and then horizontally/vertically to look for the previous match.

3 Dynamic Programming: Knapsack - Chain Multiply

3.1 Knapsack Problem

input: n objects with integer weights w_1, w_2, ..., w_n & integer values v_1, v_2, ..., v_n, total capacity B

goal: find a subset S of the n object such that

- the total weight is within capacity (i.e. fit in the backpack) i.e. $\sum_{i \in S} w_i \leq B$
- the total value is maximized i.e. $\max \sum_{i \in S} v_i$

There are two natural variants of this problem and both with different DP solutions:

without repetition: there is only one copy of each object

with repetition: there is unlimited supply of each object

We will look at the without repetition version first and then the with repetition one.

3.1.1 Greedy algorithm

The greedy approach would take the most valuable object and try to fill up the backpack as much as possible with the (remaining) most valuable object. Note that the most valuable object is not the one with the maximum value but the one with the maximum value per unit of weight. In summary the greedy algorithm sorts the objects by $r_i = v_i/w_i$ and adds the (remaining) object with the highest r_i within the capacity.

3.1.2 Knapsack without repetition

Following the recipe for designing a dynamic programming algorithm:

define the subproblem in words: let $K[i]$ be the maximum value obtainable from a subset of the first i objects

state the recursive solution: for the given 4-object set we cannot express $K[3]$ in terms of $K[1]$ and $K[2]$ because both $K[1]$ and $K[2]$ contains the 1st object but $K[3]$ doesn't, the subset of $K[3]$ is built on the suboptimal solution that contains only the 2nd object which leaves enough capacity to add the 3rd object, i.e. to solve $K[i]$ it is not sufficient to know $K[i-1]$, we also need to know the remaining capacity before adding the ith object, this observation suggests taking not only a prefix of the objects but also a prefix of the capacity available

The revised recipe is

define the subproblem in words: let $K[i, b]$ be the maximum value obtainable from a subset of the first i objects with total weight $\leq b$

state the recursive solution: if $w_i \leq b$ then we can add the ith object and $K[i, b] = \max\{v_i + K[i-1, b-w_i], K[i-1, b]\}$; if $w_i > b$ then we cannot add the ith object and $K[i, b] = K[i-1, b]$

The pseudocode that implements the DP solution for Knapsack without repetition is as follows:

```
Knapsack No Repeat(w_1, ..., w_n, v_1, ..., v_n, B) :
    for b = 0 → B:
        K[0, b] = 0
    for i = 1 → n:
        K[i, 0] = 0
    for i = 1 → n:
        for b = 1 → B:
            if w_i ≤ b
                then K[i, b] = max{v_i + K[i − 1, b − w_i], K[i − 1, b]}
            else
                then K[i, b] = K[i − 1, b]
    return K[n, B]
```

As to the running time the two base cases take $O(B)$ and $O(n)$ time respectively. Since we have a nested for-loop and the if-else statement within the nested loop takes $O(1)$ time, the running time is $O(nB)$. This algorithm is however inefficient. Because by efficient algorithm we typically mean that the running time is polynomial in input size, but the running time of our algorithm has a factor B in $O(nB)$. To represent number B takes space $O(\log B)$ thus the input size is n and $\log B$. Therefore to be efficient the algorithm should have a running time of polynomial(n, $\log B$), where $O(nB)$ is exponential in input size $\log B$. This is not surprising because we will see later that knapsack problem is NP-complete, which means that if we are able to find a polynomial algorithm to solve knapsack problem, then every problem in NP will have a polynomial-time algorithm.

3.1.3 Knapsack with repetition

First try the recipe for the without repetition case:

define the subproblem in words: let $K[i, b]$ be the maximum value obtainable from a multiset of objects $\{1, 2, \ldots, i\}$ with total weight $\leq b$

state the recursive solution: if $w_i \leq b$ then we can add the ith object and $K[i, b] = \max\{v_i + K[i, b - w_i], K[i - 1, b]\}$; if $w_i > b$ then we cannot add the ith object and $K[i, b] = K[i - 1, b]$, notice that the only difference from the without repetition version is $v_i + K[i, b - w_i]$ instead of $v_i + K[i - 1, b - w_i]$ when $w_i \leq b$, because now it refers to the scenario of adding *another copy* of the ith object instead of adding *the only one* ith object

Often when we get a DP solution which uses a 2-dimensional or 3-dimensional table, it is useful to check whether we can simplify it to a smaller table. Then we might get a faster or less space or just a simpler solution. Previously the parameter i is used to keep track of which object we are considering adding. But here since we have unlimited supply of each object the parameter i loses its significance, and in fact we can get rid of it (⇒ does this hold whenever we have a multiset?). Hence we can try to formulate a DP solution with the single parameter b. The revised recipe is as follows:

define the subproblem in words: let $K[b]$ be the maximum value obtainable from all n objects with total weight $\leq b$

state the recursive solution: we will try all the possibilities for the last object to add, i.e. $K[b] = \max_i\{v_i + K[b - w_i] : w_i \leq b\}$

The pseudocode that implements this 1-dimensional DP solution for Knapsack with repetition is as follows (since it is a 1-dimensional table there is no base case to worry about):

```
Knapsack Repeat(w_1, ..., w_n, v_1, ..., v_n, B) :
    for b = 0 → B:
        K[b] = 0
        for i = 1 → n:
            if w_i ≤ b and v_i + K[b − w_i] > K[b]
                then K[b] = v_i + K[b − w_i]
    return K[B]
```

However because we still have a nested for-loop and the if-else statement within the nested loop takes $O(1)$ time, the running time is still $O(nB)$, the same as the 2-dimensional table solution. Nevertheless by using 1-dimensional table we use $O(B)$ instead of $O(nB)$ space.
To obtain the multiset that achieves the maximum value within capacity, we can define an array $S[b]$ to keep track of the last object we add in, initializing it to $S[b] = 0$ along with $K[b] = 0$ and setting $S[b] = i$ when updating $K[b]$ with $v_i + K[b - w_i]$. Then we can use $S[B]$ to backtrack the optimal multiset similar to what we did for LCS.

3.2 Chain Matrix Multiply

input: four matrices A, B, C, and D, each having integer value entries

goal: compute $A \times B \times C \times D$ most efficiently, since we have several options to carry out the multiplication (1) $((A \times B) \times C) \times D$ (2) $(A \times (B \times C)) \times D$ (3) $(A \times B) \times (C \times D)$ (4) $A \times (B \times (C \times D))$ for which we need to quantify the cost of parenthesization

To quantify the cost of matrix multiplication, take W of size $a \times b$ and Y of size $b \times c$. Their product $Z = W \times Y$ has a size of $a \times c$. To compute one entry $Z_{ij} = \sum_{k=1}^{b} W_{ik} Y_{kj}$ it takes b multiplications and $b - 1$ additions. Since Z has ac entries there will be a total of acb multiplications and $ac(b - 1)$ additions.
The general problem formulation is as follows:

input: n matrices A_1, A_2, ..., A_n where A_i is of size $m_{i-1} \times m_i$

goal: find the minimum cost of computing their product $A_1 \times A_2 \times \cdots \times A_n$

Instead of looking at it as a parenthesization we are going to represent it as a binary tree, where the root is the final product, the leaves are the matrices, and the internal nodes represent the intermediate multiplications. How the tree is structured tells us the parenthesization.
Following the recipe for designing a dynamic programming algorithm:

define the subproblem in words: let $C[i]$ be the minimum cost of computing $A_1 \times A_2 \times \cdots \times A_i$

state the recursive solution: $A_1 \times A_2 \times \cdots \times A_n = (A_1 \times A_2 \times \cdots \times A_i) \times (A_{i+1} \times A_2 \times \cdots \times A_n)$, we will try all possible i splits, however $A_{i+1} \times A_2 \times \cdots \times A_n$ is specified by a suffix and its further split j is neither a prefix nor a suffix, nonetheless notice that all the intermediate computations correspond to substrings specified by a start index and an end index, thus we will redefine our subproblems in terms of substring

The revised recipe is

define the subproblem in words: let $C[i,j]$ be the minimum cost of computing $A_i \times A_{i+1} \times \cdots \times A_j$ where $1 \le i \le j \le n$

state the recursive solution: the base case is $C_{ii} = 0$
we try all possible l splits of $A_i \times A_{i+1} \times \cdots \times A_j$ where $1 \le i \le j \le n$ and take the minimum, i.e. $C[i,j] = \min_l\{C[i,l] + C[l+1,j] + m_{i-1}m_l m_j : i \le l \le j-1\}$ where $m_{i-1}m_l m_j$ is the cost of multiplying $A_i \times A_{i+1} \times \cdots \times A_l$ and $A_{l+1} \times A_{l+2} \times \cdots \times A_j$

The recursive relation here is different from all previous DP problems in that we need to scan an internal variable thus not only adjacent cells are used. To fill up the 2-dimensional DP table we start from the base case $C_{ii} = 0$ which corresponds to zero diagonal in the DP table. Then we fill in the off-diagonal $C_{i,i+1}$ where the only choice of l is i thus only C_{ii} and $C_{i+1,i+1}$ are used. After that we fill in $C_{i,i+2}$ which uses only diagonal and off-diagonal entries and so on until we reach $C_{1,n}$ which is the final answer ($\Rightarrow$ to have an intuition of the table filling order it is instructive to have a look at where we start and where we want to end). To loop through these filling steps we define a width variable $w = j - i$ and vary it from 0 to $n-1$.
The pseudocode that implements this 2-dimensional DP solution for chain matrix multiplication is as follows:

```
Chain Multiply(m_0, m_1, ..., m_n) :
    for i = 0 → n:
        C[i,i] = 0
    for w = 1 → n − 1:
        for i = 1 → n − w:
            j = i + w
            C[i,j] = ∞
            for l = i → j − 1
                if C[i,j] > C[i,l] + C[l+1,j] + m_{i−1} m_l m_j
                    then C[i,j] = C[i,l] + C[l+1,j] + m_{i−1} m_l m_j
    return C[1,n] ⇒ how to backtrack to find the optimal parenthesization?
```

Since we have a 3-nested for-loop and the if-else statement within the nested loop takes $O(1)$ time, the running time is $O(n^3)$.
To summarize when you are defining your subproblem, try prefixes first. If that doesn't work then you might try substrings. However one important note to keep in mind is that if you do use substrings and you get a valid solution, then go back and look at whether substrings were actually necessary or could it be simplified using prefixes.

4 Dynamic Programming: Shortest Paths

4.1 Shortest Paths

4.1.1 Single-source shortest paths

The setting is a directed graph $\vec{G} = (V, E)$ with edge weights $w(e)$. We allow anti-parallel edges which are useful to encoding an undirected graph as a directed graph. In this way directed graph problems are more general than undirected graph problems, because we can encode any undirected graph as a directed graph by replacing each edge in the undirected graph by a pair of anti-parallel edges. Designate a vertex as the start vertex denoted by $s \in V$. We want to find the shortest path from s to every other vertex in this graph, i.e. for $z \in V$ define dist(z) = the length of the shortest path from s to z.

The classic algorithm for this problem is Dijkstra's algorithm, which explores the graph in the layered approach similar to BFS. Since Dijkstra's algorithm has weights thus we have to use a min-heap or priority queue data structure, each operation in these data structures takes order $O(\log |V|)$ time. Hence there is an additional overhead over BFS. Therefore Dijkstra's algorithm takes $O((|V| + |E|) \log |V|)$ time because BFS takes $O(|V|+|E|)$ time. One big limitation of Dijkstra's algorithm is that it requires that all the edge weights are positive, because once it has visited a vertex it would not recompute a shorter path to it. Thus Dijkstra's algorithm does not guarantee to produce the shortest paths when edge weights are allowed to be negative. We will look at this more general problem where edge weights are allowed to be negative and solve it using dynamic programming.
Note that when a graph has a negative weight cycle then the shortest path problem is not well-defined any longer. Because if it has such a negative weight cycle then it can loop through it infinitely many times resulting in $\text{dist}(z) = -\infty$ for all vertices reachable from the cycle. Hence we restrict our problem formulation as follows:

input: a directed graph $\vec{G}$ with edge weights $w(e)$, a start vertex $s \in V$, assuming there are no negative weight cycles

goal: find the shortest path $\mathcal{P}$ from s to z visiting every vertex at most once thus $|\mathcal{P}| \leq |V| - 1$

Following the recipe for designing a dynamic programming algorithm:

define the subproblem in words: let $D[i, z]$ be the length of the shortest path from s to z using $\leq i$ edges, the final solution is $D[|V| - 1, z]$

state the recursive solution: base case is $D[0, s] = 0$ while $D[0, z] = \infty \; \forall z \neq s$
use the shortest path from s to the penultimate vertex y with $i - 1$ edges, and consider $\widetilde{D}[i, z] = \min_{y:\vec{yz}\in E}\{\widetilde{D}[i-1, y]+w(y, z)\}$ which is the length of the shortest path from s to z using exactly i edges ($\Rightarrow$ idea: $\leq i-1 \vee = i$ is equivalent to $\leq i$), then we have $D[i, z] = \min\{D[i-1, z], \widetilde{D}[i, z]\} = \min\{D[i - 1, z] + \min_{y:\vec{yz}\in E}\{D[i - 1, y] + w(y, z)\}\}$ because the shortest path from s to z uses either at most $i - 1$ edges or exactly i edges

The pseudocode that implements this single-source shortest path solution is as follows:

```
Bellman-Ford(G⃗, s, w) :
    for z ∈ V:
        D[0, z] = ∞
    D[0, s] = 0
    for i = 1 → |V| − 1:
        for z ∈ V:
            D[i, z] = D[i − 1, z]
            for all yz⃗ ∈ E
                if D[i, z] > D[i − 1, y] + w(y, z)
                    then D[i, z] = D[i − 1, y] + w(y, z)
    return D[|V| − 1, ·]
```

Note that normally adjacency list stores the edges out from a vertex but here we need the edges into z. Thus we need to look at the adjacency lists of the reverse graph, and to construct the reverse graph takes $O(|E| + |V|)$ time. Since the nested for-loop for $z \in V$ and for all $\vec{yz} \in E$ combined go over all edges of the graph exactly once, the total running time of this Bellman-Ford algorithm is $O(|V||E|)$. Hence it is slower than Dijkstra's algorithm $O(|E| \log |V|)$ but it permits negative weight edges. In

addition it allows us to find negative weight cycles if it exists. If the row of $i = |V|$ in the DP table is different from that of $i = |V| - 1$ or equivalently $D[n, z] < D[n-1, z]$ for some $z \in V$, then there is a negative weight cycle. Hence we change for $i = 1 \to |V| - 1$ to for $i = 1 \to |V|$ in the above pseudocode.

4.1.2 All-pairs shortest path

input: a directed graph $\vec{G}$ with edge weights $w(e)$

goal: find the length of the shortest path $\text{dist}(y, z)$ from y to z for all $y, z \in V$

The most natural solution is to apply the Bellman-Ford algorithm to every $s \in V$, but its running time is $O(|V|^2|E|)$. We will instead use the Floyd-Warshall algorithm which has a running time of $O(|V|^3)$, which is typically more efficient because $|E|$ can be up to $|V|^2$. Different from the Bellman-Ford algorithm where we define subproblem by limiting the number of edges used, here we define subproblem by limiting the use of intermediate vertices. Indexing the vertices as $V = \{1, 2, \ldots, n\}$ and following the recipe for designing a dynamic programming algorithm:

define the subproblem in words: let $D[i, s, t]$ be the length of the shortest path from s to t using a subset of $\{1, 2, \ldots, i\}$ as intermediate vertices, the final solution is $D[n, s, t]$

state the recursive solution: base case is $D[0, s, t] = w(s, t)$ if $\vec{st} \in E$ and ∞ otherwise
if $i \notin$ the shortest path $\mathcal{P}$ from s to t using a subset of $\{1, 2, \ldots, i\}$ as intermediate vertices, then $D[i, s, t] = D[i-1, s, t]$
if $i \in$ the shortest path $\mathcal{P}$ from s to t using a subset of $\{1, 2, \ldots, i\}$ as intermediate vertices, then $D[i, s, t] = D[i-1, s, i] + D[i-1, i, t]$, because $\mathcal{P}$ must start from s, go through i, and end at t and all the intermediate vertices that can be used is from $\{1, 2, \ldots, i-1\}$

$$s \xrightarrow{\text{subset of } \{1,2,\ldots,i-1\}} i \xrightarrow{\text{subset of } \{1,2,\ldots,i-1\}} t$$

The pseudocode that implements this all-pairs shortest path solution is as follows:

```
Floyd-Warshall(G⃗, w) :
    for s = 1 → n:
        for t = 1 → n:
            if s⃗t ∈ E
                then D[0, s, t] = w(s, t)
            else
                then D[0, s, t] = ∞
    for i = 1 → n:
        for s = 1 → n:
            for t = 1 → n:
                D[i, s, t] = D[i − 1, s, t]
                if D[i, s, t] > D[i − 1, s, i] + D[i − 1, i, t]
                    then D[i, s, t] = D[i − 1, s, i] + D[i − 1, i, t]
    return D[n, ·, ·]
```

Since we have a 3-nested for-loop and the if-else statement within the nested loop takes $O(1)$ time, the running time is $O(n^3)$.

Note that the above algorithm assumes no negative weight cycle. It may not be correct if there exists one. We can detect the existence of negative weight cycle by definition—if there exists a vertex $y \in V$ such that $D[n, y, y] < 0$ then y must be on a negative weight cycle.
So we have two ways to detect negative weight cycle. However the Bellman-Ford algorithm can only find the negative weight cycle that is reachable from the start vertex chosen, whereas the Floyd-Marshall algorithm can detect negative weight cycle anywhere in the graph.

5 Divide and Conquer: Fast Integer Multiplication

5.1 Fast Integer Multiplication

input: given n-bit integers X and Y where n is a power of 2

goal: compute $Z = XY$ faster than $O(n^2)$ time

The standard divide-and-conquer approach is to divide both X and Y into halves—the first $n/2$ bits and the last $n/2$ bits: $X = 2^{n/2}X_L + X_R$ and $Y = 2^{n/2}Y_L + Y_R$. Therefore XY can be expressed as

$$XY = \left(2^{n/2}X_L + X_R\right)\left(2^{n/2}Y_L + Y_R\right) = 2^n X_L Y_L + 2^{n/2}\left(X_L Y_R + X_R Y_L\right) + X_R Y_R$$

which gives us a natural recursive algorithm for computing XY—computing the product of two n-bit integers by computing the product of four $n/2$-bit integers.
The pseudocode that implements this recursive algorithm is as follows:

EasyMultiply(X, Y) :
  X_L = first $n/2$ bits of X, X_R = last $n/2$ bits of X
  Y_L = first $n/2$ bits of Y, Y_R = last $n/2$ bits of Y
  A = EasyMultiply(X_L, Y_L), B = EasyMultiply(X_R, Y_R)
  C = EasyMultiply(X_L, Y_R), D = EasyMultiply(X_R, Y_L)
  $Z = 2^n A + 2^{n/2}(C + D) + B$
  return Z

As to the running time

- it takes $O(n)$ time to break X and Y into the first $n/2$ bits and the last $n/2$ bits
- let $T(n)$ be the running time of computing the product of two n-bit integers in the worst case, then recursively computing the four products of $n/2$-bit integers takes $4T(n/2)$ time
- it takes $O(n)$ time add to Z (multiplying by 2^n and $2^{n/2}$ simply shift the bits)

Combined we obtain a recurrence for $T(n)$ which can be solved by the master theorem

$$T(n) = 4T(n/2) + O(n) \implies T = O(n^2)$$

Thus the running time of this divide-and-conquer algorithm is the same as that of the straightforward multiplication.
We can improve it by following Gauss's idea—multiplication is expensive ($O(n^2)$ for n-bit integers) while adding or subtracting is cheap ($O(n)$ for n-bit integers), hence it is worth trading more additions for fewer multiplications. Gauss's problem is to multiply two complex numbers $(a + bi) \times (c + di) = (ac - bd) + (bc + ad)i$ which requires four real number multiplications. We can reduce it to three by

computing $bc+ad$ without computing the individual terms: $bc+ad = (a+b)(c+d)-ac-bd$. Applying to our case we only compute $A = X_L Y_L$, $B = X_R Y_R$, and $C = (X_L + X_R)(Y_L + Y_R)$ and compute Z as $Z = 2^n A + 2^{n/2}(C - A - B) + B$.
The pseudocode that implements this faster algorithm is as follows:

FastMultiply(X, Y) :
  X_L = first $n/2$ bits of X, X_R = last $n/2$ bits of X
  Y_L = first $n/2$ bits of Y, Y_R = last $n/2$ bits of Y
  $A = \text{FastMultiply}(X_L, Y_L), B = \text{FastMultiply}(X_R, Y_R)$
  $C = \text{FastMultiply}(X_L + X_R, Y_L + Y_R)$
  $Z = 2^n A + 2^{n/2}(C - A - B) + B$
  return Z

and the recurrence becomes $T(n) = 3T(n/2) + O(n)$. To solve it we can use the substitution method:

$$\begin{aligned}
T(n) &= 3T(n/2) + O(n) \\
&\leq cn + 3T(n/2) \\
&\leq cn + 3\left(c\frac{n}{2} + 3T\left(\frac{n}{2^2}\right)\right) \\
&\leq cn\left(1 + \frac{3}{2}\right) + 3^2\left(c\frac{n}{2^2} + T\left(\frac{n}{2^3}\right)\right) \\
&\leq cn\left(1 + \frac{3}{2} + \left(\frac{3}{2}\right)^2 + \left(\frac{3}{2}\right)^3 + \cdots + \left(\frac{3}{2}\right)^{\log_2 n}\right) \\
&= O\left(n\left(\frac{3}{2}\right)^{\log_2 n}\right) = O(3^{\log_2 n}) = O\left(\left(2^{\log_2 3}\right)^{\log_2 n}\right) = O\left(\left(2^{\log_2 n}\right)^{\log_2 3}\right) = O(n^{\log_2 3})
\end{aligned}$$

where $\log_2 3 \approx 1.59 < 2$. In fact we can get the exponent to be arbitrarily close to 1 by breaking the integers into more than two parts. But the price we pay is the growing constant inside the big-O.
There is similar idea for multiplying matrices and the algorithm that implements it is called the Strassen algorithm.

6 Divide and Conquer: Linear-Time Median

6.1 Linear-Time Median

input: an unsorted list $A = [a_1, \ldots, a_n]$ of n numbers

goal: find the median of A i.e. the $\lceil n/2 \rceil$th smallest element of A

More generally

input: an unsorted list $A = [a_1, \ldots, a_n]$ of n numbers

goal: find the kth smallest element of A where $1 \leq k \leq n$

An easy algorithm would be to sort A and then output the kth smallest element. But the fastest mergesort takes $O(n \log n)$ time. Our goal is to find a linear $O(n)$-time algorithm without sorting A.
The basic approach is quite a reminiscent of quicksort. But we don't have to recursively consider both $A_{<p}$ and $A_{>p}$ (where p is the pivot element). We only have to recursively search in one of the $A_{<p}$ or

$A_{=p}$ or $A_{>p}$ sublists. In fact if k falls into the $A_{=p}$ sublist we don't have to recursively search at all. Now we discuss this linear QuickSelect algorithm. The basic steps are (1) choose a pivot p (2) partition A into three sublists $A_{<p}$, $A_{=p}$, $A_{>p}$

- if $k \leq |A_{<p}|$ then return QuickSelect($A_{<p}$, k)
- if $|A_{<p}| < k \leq |A_{<p}| + |A_{=p}|$ then return p
- if $k > |A_{<p}| + |A_{=p}|$ then return QuickSelect($A_{>p}$, $k - |A_{<p}| - |A_{=p}|$)

As to the running time

- it takes $O(n)$ time to break A into $A_{<p}$, $A_{=p}$, and $A_{>p}$
- let $T(n)$ be the running time of searching for the kth smallest element in the worst case, then recursively search for the kth smallest element in one of the three sublists takes $T(n/2)$ time in the worst case

Our goal is an $O(n)$-time algorithm. Let $T(n)$ be the running time of searching for the kth smallest element in the worst case. An recurrence of $T(n)$ that yields $T(n) = O(n)$ is $T(n) = T(n/2) + O(n)$ by the master theorem. To achieve a subproblem of size at most $n/2$ we need the pivot p to be the median. Suppose p is only an approximate median that lies between $n/4$ and $3n/4$, then the subproblem is of size at most $3n/4$. Applying the master theorem to the recurrence $T(n) = T(3n/4) + O(n)$ still yields $T(n)$. In fact we can relax even more e.g. between $0.01n$ and $0.99n$. The key is we need a constant c which is strictly less than 1 in the recurrence $T(n) = T(cn) + O(n)$ which yields $T(n) = O(n)$ by the master theorem.
We say the pivot p is a good pivot if $n/4 \leq p \leq 3n/4$ such that $|A_{<p}| \leq 3n/4$ and $|A_{>p}| \leq 3n/4$. Our goal is to find a good pivot p in $O(n)$ time such that we will have a recurrence $T(n) = T(3n/4) + O(n)$. The simplest scheme is to choose p randomly from A, then the probability that p is good is $(3n/4 - n/4)/n = 50\%$. On the other hand to check whether a pivot is a good one we can partition A into $A_{<p}$, $A_{=p}$, and $A_{>p}$ and check the sizes of these sublists in $O(n)$ time. Hence we can keep on selecting until we get a good pivot, and the expected number of selections it takes to get a good one is $1/50\% = 2$. Therefore it takes $O(n)$ expected time to get a good pivot by random selection. However our goal is to find an algorithm whose guaranteed worst-case runtime is $O(n)$.
Recall that $T(n) = T(cn) + O(n)$ yields $T(n) = O(n)$ as long as $c < 1$. We are going to use this extra slack as extra time to help us find a good pivot and aim at a recurrence $T(n) = T(3n/4) + T(n/5) + O(n)$ where $T(n/5) + O(n)$ is the time to find a good pivot. To do so we choose a subset S of A where $|S| = n/5$ and set p to be the median of S. We want S to be a good representative sample of A in order for the median of S to approximate that of A. Since the median of A has half the elements that is smaller than it and half the elements that is larger than it, we want each element $x \in S$ to have 2 elements of A that are larger than x and 2 elements of A that are smaller than x thus 5 in total. We break A into $n/5$ (assume n is divisible by 5) groups of 5 elements each (the simplest way is to take the first 5 elements, the next 5 elements and so on). Then we choose one representative from each group ensuring that there are at least two elements from the group that are at least the chosen representative and at least two elements that are at most the chosen representative. This can be done by sorting the group and select the middle element. Note that since there are only 5 elements sorting it takes $O(1)$ time

The pseudocode for implementing this $O(1)$-time approximate median is:

```
FastSelect(A, k) :
    break A into n/5 groups G_1, G_2, ..., G_{n/5}
    for i = 1 → n/5
        sort G_i and select m_i = median(G_i)
    let S = {m_1, m_2, ..., m_{n/5}}
    p = FastSelect(S, n/10)
    partition A into A_{<p}, A_{=p}, A_{>p}
    if k ≤ |A_{<p}|
        return FastSelect(A_{<p}, k)
    else if k > |A_{<p}| + |A_{=p}|
        return FastSelect(A_{>p}, k − |A_{<p}| − |A_{=p}|)
    else
        return p
```

Let $T(n)$ be the time to implement the FastSelect algorithm above. Because

- breaking into $n/5$ groups takes $O(n)$ time
- sorting $n/5$ groups of 5 elements takes $O(1)$ time per group totaling $O(n)$
- FastSelect S takes $T(n/5)$ time
- partition A into $A_{<p}$, $A_{=p}$, $A_{>p}$ takes $O(n)$ time
- FastSelect one of $A_{<p}$, $A_{=p}$, $A_{>p}$ assuming p is a good pivot takes $T(3n/4)$ time

combined we have $T(n) = T(3n/4) + T(n/5) + O(n)$. Since $3/4 + 1/5 < 1$ it solves to $T(n) = O(n)$ by the master theorem, which achieves our goal.
To prove our claim that p is a good pivot, sort $G_1, G_2, \ldots, G_{n/5}$ by their medians so that $m_1 \leq m_2 \leq \cdots \leq m_{n/5}$. Hence $p = \text{median}(S) = m_{n/10}$ and $m_1, m_2, \ldots, m_{n/10-1}$ together with the two elements preceding them within their groups (recall that G_i's are sorted) are at most p. Therefore we have $3n/10$ elements that are at most p thus $|A_{>p}| \leq 0.7n$. Similarly $m_{n/10+1}, m_{n/10+2}, \ldots, m_{n/5}$ together with the two elements trailing them within their groups are at least p. Therefore we have $3n/10$ elements that are at least p thus $|A_{<p}| \leq 0.7n$. $\Rightarrow$ more general recurrence for n/s subgroups is

$$T(n) = T\left(\left(1 - \frac{\lfloor s/2 \rfloor + 1}{2s}\right) n\right) + T(n/s) + O(n) \implies c = 1 - \frac{\lfloor s/2 \rfloor + 1}{2s} + \frac{1}{s} = \frac{2s - \lfloor s/2 \rfloor + 1}{2s}$$

7 Divide and Conquer: Solving Recurrences

7.1 Solving Recurrences

Some examples of recurrence and their solutions are:

$$T(n) = 2T(n/2) + O(n) \implies T(n) = O(n \log n) \quad \text{mergesort}$$
$$T(n) = 4T(n/2) + O(n) \implies T(n) = O(n^2) \quad \text{naive integer multiplication}$$
$$T(n) = 3T(n/2) + O(n) \implies T(n) = O(n^{\log_2 3}) \quad \text{improved integer multiplication}$$
$$T(n) = T(3n/4) + O(n) \implies T(n) = O(n) \quad \text{linear-time median}$$

Consider $T(n) = 4T(n/2) + O(n)$. The big-O notation means that there is some constant $c > 0$ so that $T(n) \leq 4T(n/2) + cn$ with base case $T(1) \leq c$

$$\begin{aligned}
T(n) &\leq cn + 4T(n/2) \\
&\leq cn + 4\left[4T\left(\frac{n}{2^2}\right) + c\frac{n}{2}\right] \\
&= cn\left(1 + \frac{4}{2}\right) + 4^2 T\left(\frac{n}{2^2}\right) \\
&\leq cn\left(1 + \frac{4}{2}\right) + 4^2\left[4T\left(\frac{n}{2^3}\right) + c\left(\frac{n}{2^2}\right)\right] \\
&= cn\left(1 + \left(\frac{4}{2}\right) + \left(\frac{4}{2}\right)^2\right) + 4^3 T\left(\frac{n}{2^3}\right) \\
&\leq cn\left(1 + \left(\frac{4}{2}\right) + \left(\frac{4}{2}\right)^2 + \cdots + \left(\frac{4}{2}\right)^{i-1}\right) + 4^i T\left(\frac{n}{2^i}\right)
\end{aligned}$$

Let $i = \log_2 n$ then $n/2^i = 1$ and we can substitute in the base case $T(1) \leq c$ to get

$$\begin{aligned}
T(n) &\leq \underbrace{cn}_{O(n)} \underbrace{\left(1 + \left(\frac{4}{2}\right) + \left(\frac{4}{2}\right)^2 + \cdots + \left(\frac{4}{2}\right)^{\log_2 n - 1}\right)}_{O\left(\left(\frac{4}{2}\right)^{\log_2 n}\right) = O(n)} + \underbrace{4^{\log_2 n} T(1)}_{O(4^{\log_2 n}) = O(n^2)} \\
&= O(n) \times O(n) + O(n^2) = O(n^2)
\end{aligned}$$

In general for constant $\alpha > 0$ the geometric series

$$\sum_{j=0}^{k} \alpha^j = 1 + \alpha^1 + \alpha^2 + \cdots + \alpha^k = \begin{cases} O(\alpha^k) & \text{if } \alpha > 1 \text{ (e.g. integer multiplication)} \\ O(k) & \text{if } \alpha = 1 \text{ (e.g. mergesort)} \\ O(1) & \text{if } \alpha < 1 \text{ (e.g. find median)} \end{cases}$$

and the key to convert constant exponential to polynomial is changing log base so as to match the log base to the base of the exponent. For example $3^{\log_2 n} = \left(2^{\log_2 3}\right)^{\log_2 n} = \left(2^{\log_2 n}\right)^{\log_2 3} = n^{\log_2 3}$. With this we can work out another example $T(n) = 3T(n/2) + O(n)$. Following the expansion as above we have

$$\begin{aligned}
T(n) &\leq cn\left(1 + \left(\frac{3}{2}\right) + \left(\frac{3}{2}\right)^2 + \cdots + \left(\frac{3}{2}\right)^{i-1}\right) + 3^i T\left(\frac{n}{2^i}\right) \\
&\leq \underbrace{cn}_{O(n)} \underbrace{\left(1 + \left(\frac{3}{2}\right) + \left(\frac{3}{2}\right)^2 + \cdots + \left(\frac{3}{2}\right)^{\log_2 n - 1}\right)}_{O\left(\left(\frac{3}{2}\right)^{\log_2 n}\right) = O(3^{\log_2 n}/2^{\log_2 n})} + \underbrace{3^{\log_2 n} T(1)}_{O(3^{\log_2 n}) = O(n^{\log_2 3})} \\
&= O(n) \times O(n^{\log_2 3 - 1}) + O(n^{\log_2 3}) = O(n^{\log_2 3})
\end{aligned}$$

For the general form of recurrence: $T(n) = aT(n/b) + O(n)$ with constant $a > 0$ and $b > 1$. Following the same expansion as above we have

$$T(n) \leq cn\left(1 + \left(\frac{a}{b}\right) + \left(\frac{a}{b}\right)^2 + \cdots + \left(\frac{a}{b}\right)^{\log_b n - 1}\right) + a^{\log_b n} T(1)$$

where

$$cn\left(1 + \left(\frac{a}{b}\right) + \left(\frac{a}{b}\right)^2 + \cdots + \left(\frac{a}{b}\right)^{\log_b n - 1}\right) = \begin{cases} O\left(n^{\log_b a}\right) & \text{if } a > b \\ O(n \log n) & \text{if } a = b \\ O(n) & \text{if } a < b \end{cases}$$

Of course we can look at more general forms of recurrence with $O(n^d)$ for some constant d not only $d = 1$. The solution is given by the master theorem.

8 FFT: Part 1

8.1 Polynomial Multiplication

We will multiply polynomials using the well-known divide-and-conquer algorithm called fast Fourier transform or FFT. Consider two polynomials $A(x) = a_0 + a_1x + a_2x^2 + \cdots + a_{n-1}x^{n-1}$ and $B(x) = b_0 + b_1x + b_2x^2 + \cdots + b_{n-1}x^{n-1}$. Their product $C(x) = A(x)B(x) = c_0 + c_1x + c_2x^2 + \cdots + c_{2n-2}x^{2n-2}$, where $c_k = a_0b_k + a_1b_{k-1} + \cdots + a_kb_0$. We want to solve the following computational problem:

input: two coefficient vectors $a = (a_0, a_1, a_2, \ldots, a_{n-1})$ and $b = (b_0, b_1, b_2, \ldots, b_{n-1})$ defining two polynomials

goal: compute the coefficient vector $c = a * b = (c_0, c_1, c_2, \ldots, c_{2n-2})$ for their product polynomial in $O(n \log n)$ time

Note that naive algorithm takes $O(k)$ time to compute $c_k = a_0b_k + a_1b_{k-1} + \cdots + a_kb_0$, which leads to a $O(n^2)$ algorithm to compute c.
$c = a*b$ is known as the **convolution** of a and b. There are many convolution applications for example filtering, which replaces each data point in $y = (y_1, y_2, \ldots, y_n)$ by a function of their neighboring points for reducing noises and adding effects. All types of filtering can be represented as convoluting data with some vector:

mean filtering: replace y_j by $\widehat{y} = \frac{1}{2m+1}\sum_{i=m}^{-m} y_{j+i}$, the smoothed data $\widehat{y}$ can be viewed as the convolution of y with a vector f where $f = \left(\frac{1}{2m+1}, \ldots, \frac{1}{2m+1}\right)$

Gaussian filtering: $f = (1/Z)\left(e^{-m^2}, e^{-(m-1)^2}, \ldots, e^{-1}, 1, e^1, \ldots, e^{(m-1)^2}, e^{m^2}\right)$

Gaussian blur: 2-dimensional Gaussian filter

Note that there are two natural ways of representing a polynomial $A(x)$:

coefficients: $a = (a_0, a_1, \ldots, a_{n-1})$

values: $A(x_1)$, $A(x_2)$, ..., $A(x_n)$

The latter is based on the key fact that a polynomial of degree $n-1$ is uniquely determined by its values at any n distinct points. It is actually more convenient for the purpose of multiplying polynomials, while the coefficient representation is more convenient for representing polynomials. Suppose we know the polynomial $A(x)$ evaluated at $A(x_1)$, $A(x_2)$, ..., $A(x_{2n})$ and the polynomial $B(x)$ evaluated at $B(x_1)$, $B(x_2)$, ..., $B(x_{2n})$. Then we can compute $c(x_i) = A(x_i)B(x_i)$ for $i = 1$ to $2n$ which takes $O(1)$ time for each and $O(n)$ time in total (we need $2n$ points because $C(x)$ is a polynomial of degree at most $2n - 2$). What FFT does is to convert between the coefficient representation and the value representation, and we will use it to multiply polynomials as follows:

$$\text{coefficients} \xrightarrow{O(n \log n)\text{ FFT}} \text{values} \implies O(n) \text{ value multiplication} \implies \text{values} \xrightarrow{O(n \log n)\text{ FFT}} \text{coefficients}$$

One important point is that FFT converts from coefficients to values not for any choices of x_1 through x_n, but for a particularly well-chosen set of points x_1 through x_n. That is $x_1, x_2, \ldots, x_n$ are opposite of $x_{n+1}, x_{n+2}, \ldots, x_{2n}$ i.e. $x_{n+i} = -x_i$ for $i = 1$ to n. The advantage of such choice is that the even

terms $a_{2k}x^{2k}$ remains the same for x_{n+i} while the odd terms $a_{2k+1}x^{2k+1}$ are opposite for x_{n+i}. Therefore it makes sense to split $A(x)$ into even terms $a_{\text{even}} = (a_0, a_2, a_4, \ldots, a_{n-2})$ which defines a polynomial $A_{\text{even}}(y) = a_0 + a_2 y + a_4 y^2 + \cdots + a_{n-2}y^{(n-2)/2}$, and odd terms $a_{\text{odd}} = (a_1, a_3, a_5, \ldots, a_{n-1})$ which defines a polynomial $A_{\text{odd}}(y) = a_1 + a_3 y + a_5 y^2 + \cdots + a_{n-1}y^{(n-1)/2}$ so that $A(x) = A_{\text{even}}(x^2) + xA_{\text{odd}}(x^2)$. Apply the property that $x_{n+i} = -x_i$ we have that

$$A(x_i) = A_{\text{even}}(x_i^2) + x_i A_{\text{odd}}(x_i^2)$$
$$A(x_{n+i}) = A_{\text{even}}(x_i^2) - x_i A_{\text{odd}}(x_i^2)$$

Therefore given $A_{\text{even}}(y_1)$, $A_{\text{even}}(y_2)$, …, $A_{\text{even}}(y_n)$ and $A_{\text{odd}}(y_1)$, $A_{\text{odd}}(y_2)$, …, $A_{\text{odd}}(y_n)$ for $y_1 = x_1^2$, $y_2 = x_2^2$, …, $y_n = x_n^2$, in $O(n)$ time we get $A(x_1)$, $A(x_2)$, …, $A(x_{2n})$.
In summary to get $A(x)$ of degree $\leq n-1$ at $2n$ points x_1, x_2, …, x_{2n}

1. define $A_{\text{even}}(y)$ and $A_{\text{odd}}(y)$ of degree $\leq n/2 - 1$
2. recursively evaluate $A_{\text{even}}(y)$ and $A_{\text{odd}}(y)$ at n points, which are $y_1 = x_1^2 = x_{n+1}^2$, $y_2 = x_2^2 = x_{n+2}^2$, …, $y_n = x_n^2 = x_{2n}^2$
3. in $O(n)$ time get $A(x)$ at x_1, x_2, …, x_{2n}

Let $T(n)$ denote the running time of input size n. We have two subproblems of half the size, and it takes $O(n)$ time to partition into A_{even} and merge the solutions together. Hence $T(n)$ satisfies the recurrence $T(n) = 2T(n/2) + O(n)$, which is the same as mergesort and solves to $O(n \log n)$ by the master theorem. All we need is the $2n$ points satisfy the $x_{n+i} = -x_i$ property. We also need $y_1 = -y_{n/2+1}$, $y_2 = -y_{n/2+2}$, …, $y_{n/2} = -y_n$ at next level, which implies $x_1^2 = -x_{n/2+1}^2$. This can only be satisfied with complex number.

8.2 Complex Number

Recall that a complex number $z = a + bi$ can be represented either in Cartesian coordinate (a, b) or polar coordinate (r, θ) of the complex plane. The conversion between them is $(a, b) = (r\cos\theta, r\sin\theta)$. There is a third way of representing a complex number—Euler's formula which is based on the fact that $\cos\theta + i\sin\theta = e^{i\theta}$ thus $z = r\cos\theta + ir\sin\theta = re^{i\theta}$. Polar coordinate is convenient for complex number multiplication because $z_1 \times z_2 = (r_1, \theta_1) \times (r_2, \theta_2) = (r_1 r_2, \theta_1 + \theta_2)$. We will consider the nth complex root of unity z where $z^n = 1$. Using the polar coordinate product rule we have $(r^n, n\theta) = z^n = 1 = (1, 2\pi j)$ which implies $r = 1$ and $\theta = 2\pi j/n$ where $j = 0, 1, 2, \ldots, n-1$. Let $\omega_n = (1, 2\pi/n) = e^{i2\pi/n}$. Then the nth roots of unity are $\omega_n^0, \omega_n^1, \ldots, \omega_n^{n-1}$. They possess the following two properties:

± property: $\omega_n^j = -\omega_n^{j+n/2}$ for even n

smaller root: $(\omega_n^j)^2 = \left(\omega_n^{j+n/2}\right)^2 = \omega_{n/2}^j$ for $n = 2^k$ (for evaluating $A_{\text{even}}(y)$ and $A_{\text{odd}}(y)$)

9 FFT: Part 2

9.1 FFT

The goal is to evaluate a polynomial $A(x)$ of degree $\leq n-1$ at n points which are the nth roots of unity (assume $n = 2^k$). For this we

- define $A_{\text{even}}(y)$ and $A_{\text{odd}}(y)$ of degree $\leq n/2 - 1$ which takes $O(n)$ time, and recursively evaluate them at the square of the nth roots of unity which are the $n/2$th roots of unity

- take $O(n)$ time to get $A(x)$ at the nth roots of unity via $A(x) = A_{\text{even}}(x^2) + xA_{\text{odd}}(x^2)$

Hence the running time $T(n)$ satisfies the recurrence relation $T(n) = 2T(n/2) + O(n)$ which solves to $O(n \log n)$ by the master theorem.
The FFT pseudocode is as follows, where $a = (a_0, a_1, \ldots, a_{n-1})$ is the coefficient vector of polynomial $A(x)$ and we assume n is a power of 2, and $\omega = \omega_n = e^{2\pi i/n}$ is the nth root of unity:

```
FFT(a, ω) :
    if n = 1
        return A(1)
    let a_even = (a_0, a_2, ..., a_{n-2}) and a_odd(a_1, a_3, ..., a_{n-1})
    FFT(a_even, ω^2)
        get A_even(ω^0), A_even(ω^2), ..., A_even(ω^{n-2})
    FFT(a_odd, ω^2)
        get A_odd(ω^0), A_odd(ω^2), ..., A_odd(ω^{n-2})
    for j = 0 to n/2 − 1
        A(ω^j) = A_even(ω^{2j}) + ω^j A_odd(ω^{2j})
        A(ω^{n/2+j}) = A_even(ω^{2j}) − ω^j A_odd(ω^{2j})
    return A(ω^0), A(ω^1), ..., A(ω^{n−1})
```

Apply FFT to polynomial multiplication:

input: the coefficient vectors $a = (a_0, a_2, \ldots, a_{n-1})$ and $b = (b_0, b_1, \ldots, b_{n-1})$ for two polynomials $A(x)$ and $B(x)$

output: the coefficient vector $c = (c_0, c_1, \ldots, c_{2n-2})$ for $C(x) = A(x)B(x)$

We consider $A(x)$ and $B(x)$ as polynomials of degree $2n-1$ i.e. pad a and b with n trailing zeros. Then run FFT

```
(r_0, r_1, ..., r_{2n−1}) = FFT(a, ω_{2n})
(s_0, s_1, ..., s_{2n−1}) = FFT(b, ω_{2n})
for j = 0 to 2n − 1
    t_j = r_j × s_j
```

With $C(x)$ at the $2n$th roots of unity, we need inverse FFT to convert it back to the coefficient vector.

9.2 Inverse FFT

From the linear algebra point of view evaluation at point x_j can be represented as the inner product of two vectors $A(x_j) = (1, x_j, x_j^2, \ldots, x_j^{n-1}) \cdot (a_0, a_1, \ldots, a_{n-1})$. Hence evaluation at n points $x_0, x_1, \ldots, x_n$ can be represented as matrix-vector multiplication

$$\begin{pmatrix} A(x_0) \\ A(x_1) \\ \vdots \\ A(x_{n-1}) \end{pmatrix} = \begin{pmatrix} 1 & x_0 & x_0^2 & \cdots & x_0^{n-1} \\ 1 & x_1 & x_1^2 & \cdots & x_1^{n-1} \\ \vdots & \vdots & \vdots & \cdots & \vdots \\ 1 & x_{n-1} & x_{n-1}^2 & \cdots & x_{n-1}^{n-1} \end{pmatrix} \begin{pmatrix} a_0 \\ a_1 \\ \vdots \\ a_{n-1} \end{pmatrix}$$

Therefor $\text{FFT}(a, \omega_n)$ can be represented as $A = M_n(\omega_n)a$:

$$\begin{pmatrix} A(1) \\ A(\omega_n) \\ \vdots \\ A(\omega_n^{n-1}) \end{pmatrix} = \begin{pmatrix} 1 & 1 & 1 & \cdots & 1 \\ 1 & \omega_n & \omega_n^2 & \cdots & \omega_n^{n-1} \\ \vdots & \vdots & \vdots & \cdots & \vdots \\ 1 & \omega_n^{n-1} & \omega_n^{2(n-1)} & \cdots & \omega_n^{(n-1)(n-1)} \end{pmatrix} \begin{pmatrix} a_0 \\ a_1 \\ \vdots \\ a_{n-1} \end{pmatrix}$$

To perform inverse FFT converting evaluations to coefficient vector, we multiply A with the inverse of $M_n(\omega_n)$ i.e. $a = M_n(\omega_n)^{-1}A$, where the inverse matrix is given by

$$M_n(\omega_n)^{-1} = \frac{1}{n}M_n(\omega_n^{-1}) = \frac{1}{n}M_n(\omega_n^{n-1})$$

Intriguingly we can also represent inverse FFT as FFT: $na = M_n(\omega_n^{n-1})A = \text{FFT}(A, \omega_n^{n-1})$, where we treat the evaluation of A at the n points as the coefficient vector of a new polynomial. An additional interesting fact of this FFT is that, the powers of ω_n^{n-1} visit the same points of the powers of ω_n. The only difference is that the powers of ω_n^{n-1} go clockwise starting from ω_n^{n-1} whereas the powers of ω_n go counterclockwise starting from ω_n, i.e. for inverse FFT we go through the nth roots of unity in the opposite order.
To prove $M_n(\omega_n)^{-1} = \frac{1}{n}M_n(\omega_n^{-1})$ we use the fact that for any nth root of unity ω with $\omega \neq 1$ the sum $1 + \omega + \omega^2 + \cdots + \omega^{n-1} = (\omega^n - 1)/(\omega - 1) = 0$. Substitute it into the product $M_n(\omega_n)M_n(\omega_n^{-1})$:

diagonal: $\left(1, \omega_n^k, \omega_n^{2k}, \ldots, \omega_n^{(n-1)k}\right) \cdot \left(1, \omega_n^{-k}, \omega_n^{-2k}, \ldots, \omega_n^{-(n-1)k}\right) = 1 + 1 + 1 + \cdots + 1 = n$

off-diagonal: $\left(1, \omega_n^j, \omega_n^{2j}, \ldots, \omega_n^{(n-1)j}\right) \cdot \left(1, \omega_n^{-k}, \omega_n^{-2k}, \ldots, \omega_n^{-(n-1)k}\right) = 1 + \omega + \omega^2 + \cdots + \omega^{n-1} = 0$
where $\omega = \omega_n^{j-k} \neq 1$ since $k \neq j$

With inverse FFT we can spell out the $O(n \log n)$ polynomial multiplication algorithm outlined above

$$\left.\begin{array}{l}\text{FFT}(a, \omega_{2n}) = (r_0, r_1, \ldots, r_{2n-1}) \\ \text{FFT}(b, \omega_{2n}) = (s_0, s_1, \ldots, s_{2n-1})\end{array}\right\} \implies t_j = r_j \times s_j \implies \frac{1}{2n}\text{FFT}(t, \omega_{2n}^{2n-1}) = (c_0, c_1, \ldots, c_{2n-1})$$

10 Graph Algorithms: Strongly Connected Components

10.1 DFS on Undirected Graphs

To get connected components in a undirected graph G, we run DFS and keep track of the connected component number cc:

```
DFS(G)
    cc = 0
    for all v ∈ V do
        visited(v) = false
    for all v ∈ V do
        if not visited(v) then
            cc ++
            explore(v)
```

where the explore() subroutine is:

```
Explore(z)
    cc number(z) = cc
    visited(z) = true
    for all (z, w) ∈ E do
        if not visited(w) then
            Explore(w)
```

The running time is $O(n+m)$ where $n = |V|$ and $m = |E|$.
To find a path between connected vertices we use an prev(v) array as used in Dijkstra's algorithm to keep track of the predecessor vertex. The modified DFS algorithm is as follows (modification highlighted in bold):

```
DFS(G)
    cc = 0
    for all v ∈ V do
        visited(v) = false
        prev(v) = null
    for all v ∈ V do
        if not visited(v) then
            cc ++
            explore(v)
```

where the explore() subroutine is:

```
Explore(z)
    cc number(z) = cc
    visited(z) = true
    for all (z, w) ∈ E do
        if not visited(w) then
            Explore(w)
            prev(w) = z
```

We then can use this prev() array to backtrack to find a path between pairs of vertices in the same connected components.

10.2 DFS on Directed Graphs

To determine the connectivity properties of a directed graph we use DFS with preorder (not in this course) or postorder numbers (differences from DFS on undirected graphs are highlighted in bold):

```
DFS(G)
    clock = 1
    for all v ∈ V do
        visited(v) = false
        prev(v) = null
    for all v ∈ V do
        if not visited(v) then
            cc ++
            explore(v)
```

where the explore() subroutine is:

```
Explore(z)
    preorder(z) = clock
    clock++
    visited(z) = true
    for all (z, w) ∈ E do
        if not visited(w) then
            Explore(w)
            prev(w) = z
    postorder(z) = clock
    clock++
```

There are four types of edges:

tree edge: go down one depth of the DFS tree, postorder of the head > postorder of the tail

back edge: go from a descendant to an ancestor, postorder of the head < postorder of the tail

forward edge: go down multiple depths of the DFS tree, postorder of the head > postorder of the tail

cross edge: head and tail have no ancestor-descendant relation to each other, postorder of the head > postorder of the tail

Out of the four only back edge has increasing postorder number. A directed graph G has a cycle if and only if its DFS tree has a back edge:

cycle ⇒ back edge: consider a cycle $a \to b \to \cdots \to j \to a$, one of these vertices has to be explored first say i, by the edges that constitute the cycle all the other vertices in the cycle are reachable from i and contained in the subtree rooted at i, and the edge $i - 1 \to i$ is a back edge

back edge ⇒ cycle: for a back edge $a \to b$, a is a descendant of b, hence in the DFS tree there is a sequence of edges from the ancestor b to a, this sequence of edges together with $a \to b$ form a cycle

10.3 Directed Acyclic Graphs

A **directed acyclic graph** or DAG has no cycle thus no back edge. We will topologically sort a DAG i.e. order vertices so that all edges go from lower order number vertices to higher order number vertices. This echoes the fact that among the four types of edges only back edge has increasing postorder number from head to tail. To do so we run DFS on the DAG and sort the vertices by decreasing postorder number. Note that since all postorder numbers fall within the range from 1 to $2n$ where $n = |V|$, we can place the vertices in their appropriate positions in an array of size $2n$ according to their postorder numbers, which takes linear $O(n)$ time instead of $O(n \log n)$ of mergesort. Overall it takes $O(n)$ time to sort the vertices by decreasing postorder number and $O(n + m)$ time to run DFS, hence the total algorithm takes $O(n + m)$ time.
There are two types of vertices in a topological order:

source vertex: has no incoming edge, a DAG has at least one source—the first vertex of the topological order, the one with the highest postorder number

sink vertex: has no outgoing edge, a DAG has at least one sink—the last vertex of the topological order, the one with the lowest postorder number

Note that there might be multiple sources and/or sinks because there might be multiple topological orders. This motivates the following alternative topological sorting algorithm, which is not useful to DAG but useful to general directed graphs:

1. find a sink, output it, and delete it from the graph
2. repeat step 1 until the graph is empty

10.4 Strongly Connected Component

The directed graph analog of connected component in undirected graph is **strongly connected component** or SCC, which can be found using two vanilla versions of DFS. Vertices v and w are strongly connected if there is a path $v \rightsquigarrow w$ and a path $w \rightsquigarrow v$. Similar to undirected graph where a connected component is a maximal set of connected vertices, a SCC is a maximal set of strongly connected vertices. A source in a DAG is a SCC by itself because no other vertices can reach it, so is a sink in a DAG.

Take a representative vertex from each SCC we form a metagraph of SCCs, which may be a multigraph because there may be multiple edges that go from one SCC to another. Every metagraph on SCCs of a directed graph is a DAG. This can be proved by contradiction. Suppose there are two SCCs involved in a cycle. That means there is a path from some vertex A of SCC #1 to some vertex C of SCC #2 and a path from some vertex D from SCC #2 to some vertex B from SCC #1. But A and B are strongly connected in SCC #1 and C and D are strongly connected in SCC #2. Hence through A, B, C, D every vertex in SCC #1 can be reached from every vertex in SCC #2 and every vertex in SCC #2 can be reached from every vertex in SCC #1. Therefore neither SCC #1 or SCC #2 is a maximal set of strongly connected vertices, which contradicts with the definition of SCC.

Since every directed graph is a DAG of its SCCs, we can sort its SCCs into topological order. We can find the SCCs and their topological order with two runs of DFS. We will find a sink SCC, output it, delete it from the metagraph, and then repeat. We choose to work with sink SCC instead of source SCC (in principle you can also output it, delete it, and move forward instead of backward) because it is easier to work with than source SCC. To see why take any $v \in S$ where S is a sink SCC and run Explore(v) of DFS. We will visit all vertices of S and nothing else, because no edge goes from S to other SCC. This is the key property of a sink SCC. Hence we can rip out these explored vertices which deletes S from the metagraph and repeat the algorithm. In contrast if we explore a vertex in a source SCC we can reach the whole graph from it. Thus we have no way of marking off only the vertices in that SCC.

In a DAG the vertex with the lowest postorder number is a sink. However in general directed graphs the vertex with the lowest postorder number can lie in a source SCC. On the other hand in a DAG the vertex with the highest postorder number is a source. The same is true for general directed graphs—the vertex with the highest postorder number is guaranteed to lie in a source SCC. We can use this property to identify a sink SCC, because with all edges flipped a source SCC becomes a sink SCC and a sink SCC becomes a source SCC. More precisely for a directed graph $G = (V, E)$ we consider its reverse graph $G^R = (V, E^R)$, where $E^R = \{\overrightarrow{wv} : \overrightarrow{vw} \in E\}$. The set of SCCs are the same in G and G^R. Moreover a source SCC in G is a sink SCC in G^R, and a sink SCC in G is a source SCC in G^R. Therefore the algorithm of finding and topologically sorting SCCs in a directed graph G consists of the following steps:

1. construct the reverse graph G^R
2. run DFS on G^R to determine the postorder numbers of V

3. order V by decreasing postorder number

4. run explore() on G from the vertex with the highest postorder number to identify a SCC of G

5. mark off the vertices of the SCC and repeat step 4

With this algorithm the SCCs of G are output in reverse topological order, and the *cc* number produced by the undirected version of DFS, explore(), gives the topological order of the SCCs.
To prove the fact that the vertex with the highest postorder number must lie in a source SCC, we first prove the simpler claim that if there exists a edge that goes from $v \in S$ to $w \in S'$, then the maximum postorder number in SCC $S >$ the maximum postorder number in SCC S'. This claim provides a way to topologically sort SCCs—sort them by the maximum postorder number in them in decreasing order. In fact we can define the postorder number for a SCC as the maximum postorder number of the vertices in them, and the claim asserts that all edges will go from SCCs with larger postorder number to SCCs with smaller postorder number. Hence the SCC where the vertex with the highest postorder number lies in is at the beginning of the topological order thus is a source SCC.
To prove the simpler claim observe that while there is an edge that goes from S to S', there should not exist any path from S' to S. Because otherwise S and S' are strongly connected thus neither of them is a maximal set of strongly connected vertices, contradicting with the SCC definition. Run DFS on G. Some vertex z in $S \cup S'$ must be visited first:

- if $z \in S'$ then explore(z) will reach all vertices in S' but no vertex in S, hence z and all vertices of S' are assigned a postorder number before a postorder number is given to any vertex in S, therefore all postorder numbers in $S' <$ all postorder numbers in S, thus the maximum postorder number in $S >$ the maximum postorder number in S'

- if $z \in S$ then explore(z) will reach all vertices in S and all vertices in S' via the edge $v \to w$, hence all the rest of $S \cup S'$ lies in the subtree rooted at z, therefore z has the maximum postorder number in $S \cup S'$, thus the maximum postorder number in S which is that of $z >$ the maximum postorder number in S'

Since we can find SCCs and their topological order with two runs of DFS, the SCC topological sorting algorithm takes $O(n + m)$ time.

10.5 BFS and Dijstra's Algorithm

BFS explores graphs in layers. The input to BFS is similar to that for DFS except for an additional input parameter—start vertex s. BFS returns the distance from the start vertex s to every vertex. With unweighted G $\text{dist}_s(v)$ is the minimum number of edges from s to v for all $v \in V$, which is infinity if no path exists from s to v. BFS also returns the prev array which can be used to construct a path of minimum length from s to v. Like DFS BFS is linear time, so the running time is $O(n + m)$.
Dijkstra's algorithm is a more sophisticated version of BFS. It solves a weighted version of the graph and has an additional input—a length function $l : e \mapsto l(e)$ assigning a positive length to all edges. Note that one of the key requirements of Dijkstra's algorithm is that the lengths must be positive. It returns $\text{dist}_s(v)$ which is the length of the shortest path from s to v. Dijkstra's algorithm uses the BFS framework with the min-heap/priority queue data structure. Each operation in the min-heap data structure takes $O(\log n)$ time, hence the total running time of Dijkstra's algorithm is $O((n+m)\log n)$.

11 Graph Algorithms: Satisfiability

11.1 SAT Problem

One application of strongly connected component is the **satisfiability problem** or the SAT problem. Consider n Boolean variables $x_1, x_2, \ldots, x_n$. There are $2n$ literals: $x_1, \bar{x}_1, x_2, \bar{x}_2, \ldots, x_n, \bar{x}_n$. Use $\wedge$ to denote logical AND and $\vee$ for logical OR. We look at Boolean formulas in **conjunctive normal form** or CNF. A CNF is an AND of m clauses. Each clause is the OR of several literals, e.g. $x_3 \vee \bar{x}_5 \vee \bar{x}_1 \vee x_2$. An example CNF using this clause is $(x_2) \wedge (\bar{x}_3 \vee x_4) \wedge (x_3 \vee \bar{x}_5 \vee \bar{x}_1 \vee x_2) \wedge (\bar{x}_2 \vee \bar{x}_1)$, which evaluates to True if x_1 and x_3 are False and x_2 is True. Any Boolean formula can be converted into a CNF form, but the size of the formula may blow up.
The SAT problem is defined as follows:

input: a Boolean formula f in CNF with n Boolean variables $x_1, x_2, \ldots, x_n$ and m clauses

output: assignment (assigning True or False to each variable) that satisfies f (f = True) if one exists, and no if none exists

For example a satisfying assignment for $f = (\bar{x}_1 \vee \bar{x}_2 \vee x_3) \wedge (x_2 \vee x_3) \wedge (\bar{x}_3 \vee \bar{x}_1) \wedge (\bar{x}_3)$ is x_1 = False, x_2 = True, x_3 = False. We will look at a restrictive form of the SAT problem called k-**SAT**, where the sizes of the clauses—the number of literals in it—are at most k. The f above is an example of 3-SAT. We will see that the SAT problem is NP-complete. However while k-SAT is NP-complete for all $k \geq 3$, there exists a polynomial-time algorithm for 2-SAT which utilizes SCC.

11.2 2-SAT

An input f for 2-SAT can be simplified by removing unit clauses i.e. clauses with only one literal. For example to satisfy $f = (x_3 \vee \bar{x}_2) \wedge (\bar{x}_1) \wedge (x_1 \vee x_4) \wedge (\bar{x}_4 \vee x_2) \wedge (\bar{x}_3 \vee x_4)$, x_1 must be False and thus can be replaced with False throughout. The basic procedure for eliminating unit clauses is:

1. take a unit clause say literal a_i and satisfy it i.e. set a_i = True
2. remove clauses containing a_i and drop $\bar{a}_i$ from all other clauses
3. the resulted f' is satisfiable if and only if f is satisfiable
4. repeat the above steps until f' is empty which is satisfied or f' with all clauses of size exactly two

Applying this procedure, the above f simplifies to

$$\begin{aligned} f &= (x_3 \vee \bar{x}_2) \wedge (\bar{x}_1) \wedge (x_1 \vee x_4) \wedge (\bar{x}_4 \vee x_2) \wedge (\bar{x}_3 \vee x_4) && x_1 = \text{False} \\ &= (x_3 \vee \bar{x}_2) \wedge (x_4) \wedge (\bar{x}_4 \vee x_2) \wedge (\bar{x}_3 \vee x_4) && x_4 = \text{True} \\ &= (x_3 \vee \bar{x}_2) \wedge (x_2) && x_2 = \text{True} \\ &= x_3 && x_3 = \text{True} \\ &= \text{empty} \end{aligned}$$

Hence we can assume that the input to a 2-SAT problem is constituted by clauses of size exactly two. We can then encode any formula with a directed graph as follows:

- the $2n$ vertices correspond to the $2n$ literals $x_1, \bar{x}_1, x_2, \bar{x}_2, \ldots, x_n, \bar{x}_n$
- the $2m$ edges correspond to 2 implications per clause, e.g. to satisfy a clause $\bar{x}_1 \vee \bar{x}_2$ if x_1 = True then x_2 = False so we have an edge from x_1 to $\bar{x}_2$; if x_2 = True then x_1 = False so we have an edge from x_2 to $\bar{x}_1$, in general for a clause $\alpha \vee \beta$ we have an edge $\bar{\alpha} \to \beta$ and an edge $\bar{\beta} \to \alpha$

Each path in the directed graph is a chain of implications. If there is a path $x_i \rightsquigarrow \bar{x}_i$ and a path from $\bar{x}_i \rightsquigarrow x_i$, then the former requires x_i to be False whereas the latter requires x_i to be True at the same time. It implies that the formula is not satisfiable. Translated into the terminology of graph, if a literal and its negation are in the same SCC then the formula is not satisfiable. Conversely we will show that if all literals and their negations belong to different SCCs, then there exists a satisfying assignment. In summary

- if for some i, x_i and $\bar{x}_i$ are in the same SCC, then f is not satisfiable
- if for any i, x_i and $\bar{x}_i$ are in different SCCs, then f is satisfiable

We just showed the former and we will prove the latter by finding a satisfying assignment.
One approach to construct such a satisfying assignment is to modify the SCC topological order sorting algorithm:

1. take a sink SCC S, set S to be True by satisfying all of the literals in it so that we don't have to worry about the implications of all incoming edges
2. remove S from the graph and repeat

The reason why it works is that the complements of the literals in a sink SCC are in a source SCC, and because they are set to False we don't need to worry about their implications through outgoing edges. This suggests the second approach:

1. take a source SCC S', set S' to be False by not satisfying any of the literals in it so that we don't have to worry about the implications of all outgoing edges
2. this at the same time sets the sink SCC $\bar{S}'$ to be True so that we can remove both S' and $\bar{S}'$ from the graph and repeat

The above discussion is based on the key fact that if for any i, x_i and $\bar{x}_i$ are in different SCCs, then if S is a sink SCC then $\bar{S}$ is a source SCC. Using this we can develop the following 2-SAT algorithm:

2SAT(f)

1. construct directed graph G for f and run the SCC topological order algorithm on G
2. take a sink SCC S and set it to be True, which sets $\bar{S}$ to be False
3. remove S and $\bar{S}$ and repeat until we empty the graph

The running time is dominated by the SCC construction step, which takes $O(n+m)$ time.
To prove the key fact that S is a sink SCC if and only if $\bar{S}$ is a source SCC, we need a simpler claim—there is a path $\alpha \rightsquigarrow \beta$ if and only if there is a path $\bar{\beta} \rightsquigarrow \bar{\alpha}$. With this claim the key fact can be proved as follows:

- take a pair of vertices α and β in sink SCC S, since they belong to the same SCC there exist paths $\alpha \rightsquigarrow \beta$ and $\beta \rightsquigarrow \alpha$ which correspond to clauses $(\bar{\alpha} \vee \beta)$ and $\left(\alpha \vee \bar{\beta}\right)$
- this implies that there exist paths $\bar{\beta} \rightsquigarrow \bar{\alpha}$ and $\bar{\alpha} \rightsquigarrow \bar{\beta}$, which implies that $\bar{\alpha}$ and $\bar{\beta}$ are in the same SCC denoted by $\bar{S}$, hence S is a SCC if and only if $\bar{S}$ is a SCC
- since S is a sink SCC, for any $\alpha \in S$ there is no outgoing edge $\alpha \to \gamma$, by the claim above this implies that there is no incoming edge $\bar{\gamma} \to \bar{\alpha}$ for any $\bar{\alpha} \in \bar{S}$, hence $\bar{S}$ is a source SCC, reversing the argument we can prove the key fact

To prove the simpler claim let $\alpha \to \beta$ go along $\alpha = \gamma_0 \to \gamma_1 \to \cdots \to \gamma_{l-1} \to \gamma_l = \beta$, where the edge $\gamma_i \to \gamma_{i+1}$ comes from the clause $(\bar{\gamma}_i \vee \gamma_{i+1})$ which implies the other edge $\bar{\gamma}_{i+1} \to \bar{\gamma}_i$. This argument applies to $i = 0, 1, 2, \ldots, l-1$, which form a path $\bar{\beta} = \bar{\gamma}_l \to \bar{\gamma}_{l-1} \to \cdots \to \bar{\gamma}_1 \to \bar{\gamma}_0 = \bar{\alpha}$.

12 Graph Algorithms: Minimum Spanning Tree

12.1 The MST Problem

Greedy algorithm takes local optimal move. It does not always lead to global optimum as in knapsack problem, but it does for the MST problem:

input: undirected graph $G = (V, E)$ with weight $w(e)$ for $e \in E$

goal: find the minimum size of the connected subgraphs, since the connected subgraph of minimal size is a spanning tree, it is equivalent to finding the minimum-weight spanning tree

The basic properties of a **tree** (connected + acyclic) include

- an n-vertex tree has $n - 1$ edges
- there is exactly one (connected + acyclic) path between every pair of vertices
- any connected graph $G = (V, E)$ with $|E| = |V| - 1$ is a tree

The third property follows from combining the first property and second property.
Kruskal's algorithm is a greedy algorithm for MST, which consists of the following steps:

1. sort E by increasing weight
2. set X to be the edges that we have inserted into our MST and initialize it to $\emptyset$
3. for $e = (v, w) \in E$ in order, if $X \cup e$ doesn't have a cycle then add e to X

As to the runtime, the first step takes $O(m \log n)$ time where $m = |E|$ and $n = |V|$. To see whether the edge e would create a cycle when added in, we need to check whether there already exists a path between v and w thus they belong to the same connected component. If v and w belong to different connected components, then we can add e into X. Checking whether v and w are in the same connected component can be done using the union-find data structure, which takes $O(\log n)$ time because

find: takes $O(\log n)$ time to check the component containing v and that containing w

union: merging the two components after adding e takes $O(\log n)$ time

Hence the third step also takes $O(m \log n)$ time, and so is Kruskal's algorithm.

12.2 Cut Property

We prove Kruskal's algorithm via the following cut property: for an undirected graph $G = (V, E)$ take subgraph $X \subset T$ where T is a MST. Take any $S \subset V$ where no edge of X is in the cut $(S, \bar{S})$. Note that for an undirected graph $G = (V, E)$ a **cut** is a set of edges which partition the vertices into two sets $V = S \cup \bar{S}$, i.e. $\text{cut}(S, \bar{S}) = \{(v, w) \in E : v \in S, w \in \bar{S}\}$. We take e^* to be the edge of minimum weight among all edges in the $\text{cut}(S, \bar{S})$. Such an edge always exists because we assume the graph is connected. Then $X \cup e^* \subset T'$ where T' is a MST (T' need not be the same as T). Kruskal's algorithm uses a particular set S, but this cut property is true for any cut. The proof of the cut property is as follows:

1. fix G, X, T, S so that $X \subset T$ where T is a MST and there is no edge of X crossing S and $\bar{S}$
2. choose any edge of minimum weight across this cut $w(e^*) \leq w(e_1), \ldots, w(e_i)$

3. construct MST T' where $X \cup e^* \subset T'$, if $e^* \in T$ then let $T' = T$ and we are done, otherwise set $T' = T \cup e^* - e'$ where $e' \in T$ crosses S and it always exists because T is a MST thus by definition must connect S and $\bar{S}$

To prove we first need to show T' is a tree, for which we show that T'

has exactly $n-1$ edges: trivial because T has exactly $n-1$ edges

is connected: for any $y, z \in V$ let $\mathcal{P}$ be a path from y to z in T, let $e' = c \to d$, let C be the cycle in $T \cup e^*$, let $\mathcal{P}' = C - e'$ be a path from c to d in T', then to go from y to z in T' we modify $\mathcal{P}$ by replacing e' with $\mathcal{P}'$

Then we need to show T' is a MST. $T' = T \cup e^* - e'$ thus $w(T') = w(T) + w(e^*) - w(e')$. By construction $w(e^*) \leq w(e')$ thus $w(T') \leq w(T)$. Because $w(T)$ is minimum since it is a MST, $w(T')$ is minimum as well. In fact $w(T') = w(T)$ because otherwise we would find a spanning tree with smaller weight, which contradicts with the fact that T is a MST.
To apply the cut property to Kruskal's algorithm, define S to be the set of vertices connected through the edges in $X \cup v$ of the edge $e = (v, w)$. Then since adding e to X doesn't create a cycle, the other endpoint w of e must belong to $\bar{S}$, thus by sorting e is the minimum-weight edge across the cut$(S, \bar{S})$. It then follows from the cut property that $X \cup e$ belongs to a MST.
The two takeaway ideas about the cut property are:

- removing any edge of the cycle C in $T \cup e^*$ produces a new MST T'
- a minimum-weight edge across a cut lies in a MST

13 Max-Flow: Fork-Fulkerson Algorithm

13.1 Max-Flow

input: a directed graph $G = (V, E)$ and designated $s, t \in V$, a capacity $c_e > 0$ for each $e \in E$

goal: maximize the flow from s to t without exceeding capacities, specified by flow f_e for each $e \in E$

There are two constraints associated with f_e

capacity constraint: $0 \leq f_e \leq c_e$ for all $e \in E$

conservation of flow: inflow to v = outflow from v for all $v \in V - \{s, t\}$

and the flow f from s to t can be measured by either the outflow from s or the inflow to t. Note that cycles are OK because there is no reason to utilize a cycle in a flow network. The flow can bypass it and still achieve maximal size. The simplest cycle is formed by a pair of anti-parallel edges, and we can remove these anti-parallel edges to simplify the input to the flow network. The way we remove them is to introduce an auxiliary vertex between the pair of vertices. The resulted flow network is equivalent to the original one, because we can easily convert the max-flow solution for one network to the other.

13.2 Residual Network

A naive but straightforward idea is

1. start with $f_e = 0$ for all $e \in E$
2. run BFS/DFS to find st-path $\mathcal{P}$ with available capacity

3. let $c(\mathcal{P}) = \min_{e \in \mathcal{P}}(c_e - f_e)$ be the available capacity along $\mathcal{P}$

4. augment f by $c(\mathcal{P})$ along $\mathcal{P}$

5. repeat step 2 to 4 until there is no $\mathcal{P}$ with available capacity in the graph

However this graph of available capacities is not possible to represent equivalent backward flow that reduces an edge flow to below capacity and creates other $\mathcal{P}$ with available capacity. To accommodate this possibility we introduce an backward edge to every intermediate edge, and the resulted graph is called the residual flow network.
Formally since a **residual network** is a function of current flow, we should define it for a flow network $G = (V, E)$ with capacity c_e and flow f_e for $e \in E$. Denote it by $G^f = (V, E^f)$ where the subscript f emphasizes the dependence on current flow f, particularly an edge with nonzero flow size allows the addition of backward edge. In summary

forward edge addition: if $\overrightarrow{vw} \in E$ and $f_{vw} < c_{vw}$ then add $\overrightarrow{vw}$ to G^f with capacity $c_{vw} - f_{vw}$

backward edge addition: if $\overrightarrow{vw} \in E$ and $f_{vw} > 0$ then add $\overrightarrow{wv}$ to G^f with capacity f_{vw}

To avoid interaction between the forward edge and backward edge additions, we remove anti-parallel edges from the original flow network.

13.3 Ford-Fulkerson Algorithm

The Ford-Fulkerson algorithm works on residual network as follows:

1. set $f_e = 0$ for all $e \in E$

2. build the residual network G^f for current flow f

3. use BFS/DFS to look for a st-path $\mathcal{P}$ in G^f, if there is no such path then output f

4. given $\mathcal{P}$ let $c(\mathcal{P})$ be the minimum capacity along $\mathcal{P}$ in G^f

5. augment f by $c(\mathcal{P})$ along $\mathcal{P}$—increasing the flow by $c(\mathcal{P})$ along every forward edge while decreasing the flow by $c(\mathcal{P})$ along every backward edge

6. repeat step 2 to 5 until there is no such st-path, and current flow f is output in step 3

The proof of the correctness of the Ford-Fulkerson algorithm requires the application of max-flow = min-cut theorem below. As to the running time we need to assume that all capacities are integers. Because only under this assumption can we make a claim about the running time for the Ford-Fulkerson algorithm. The Edmonds-Karp algorithm that we will see later eliminates this assumption. The main point of this assumption is that when we augment the flow, we will augment it by an integer amount. Therefore the capacities in the residual network will maintain as integers. Moreover if all the capacities in the residual network are integers, then the flow will be increased by at least 1 per round of the algorithm for every forward edge. Now let C be the size of the max-flow. Then the algorithm will run for at most C rounds.
The time per round breakdown of the Ford-Fulkerson algorithm is as follows (repeated step 2 to 5):

2. the path $\mathcal{P}$ is of length at most $n - 1$ edges thus it takes $O(n)$ time to update residual network

3. the BFS/DFS takes $O(n + m)$ time, assume the number of edges is at least $n - 1$ then the time is bounded by $O(m)$

4. the path $\mathcal{P}$ is of length at most $n-1$ edges thus it takes $O(n)$ time to find the minimum capacity

5. the path $\mathcal{P}$ is of length at most $n-1$ edges thus it takes $O(n)$ time to augment current flow

Combined it takes $O(m)$ time per round to run the Ford-Fulkerson algorithm. Since the algorithm runs at most C rounds, it requires $O(Cm)$ time overall.
There are two unpleasant problems with this conclusion:

- the assumption of integral capacities limits the applicability of our argument
- the running time depends on the output which is a priori unknown and the size of the capacities in the input, since the running time depends on the input we say that it is pseudo-polynomial

The Edmonds-Karp algorithm overcomes both problems by using the shortest st-path via BFS instead of DFS, shortest in the sense of minimum number of edges without caring about the capacity and flow.

14 Max-Flow: Max-Flow = Min-Cut

14.1 Min-Cut Problem

Recall that a cut of a graph is a partition of the vertices into two sets $V = L \cup R$. Here we need **st-cut** which is a cut with $s \in L$ and $t \in R$. Note that L and R do not need to be connected sets. We define the **capacity** of a st-cut to be the total capacity of all edges from L to R:

$$\text{capacity}(L, R) = \sum_{\overrightarrow{vw} \in E, v \in L, w \in R} c_{vw}$$

The min-cut problem is to find the st-cut with minimum capacity.

14.2 Max-Flow = Min-Cut

The max-flow = min-cut theorem states that for any flow network the size of the max-flow equals the minimum capacity of an st-cut. We will prove the equivalence by showing that

max-flow $\leq$ min st-cut: We will show that for any flow f and any st-cut (L, R), size$(f) \leq$ capacity (L, R). Then we can take the maximum of the left-hand side and the minimum of the right-hand side to arrive at max-flow $= \max_f \text{size}(f) \leq \min_{(L,R)} \text{capacity}(L, R) =$ min st-cut.
To prove size$(f) \leq$ capacity (L, R) for any flow f and any st-cut (L, R), we will prove the following claim: size$(f) = f^{\text{out}}(L) - f^{\text{in}}(L)$, from which size$(f) \leq$ capacity (L, R) will follow immediately.

To show $\text{size}(f) = f^{\text{out}}(L) - f^{\text{in}}(L)$ we just spell out the definition on the right-hand side:

$$\begin{aligned}
f^{\text{out}}(L) - f^{\text{in}}(L) &= \sum_{\overrightarrow{vw} \in E, v \in L, w \in R} f_{vw} - \sum_{\overrightarrow{wv} \in E, w \in R, v \in L} f_{wv} \\
&= \sum_{\overrightarrow{vw} \in E, v \in L, w \in R} f_{vw} - \sum_{\overrightarrow{wv} \in E, w \in R, v \in L} f_{wv} + \sum_{\overrightarrow{vw} \in E, v \in L, w \in L} f_{vw} - \sum_{\overrightarrow{wv} \in E, w \in L, v \in L} f_{wv} \\
&= \left(\sum_{\overrightarrow{vw} \in E, v \in L, w \in R} f_{vw} + \sum_{\overrightarrow{vw} \in E, v \in L, w \in L} f_{vw} \right) - \left(\sum_{\overrightarrow{wv} \in E, w \in R, v \in L} f_{wv} + \sum_{\overrightarrow{wv} \in E, w \in L, v \in L} f_{wv} \right) \\
&= \sum_{v \in L} f^{\text{out}}(v) - \sum_{v \in L} f^{\text{in}}(v) \\
&= \underbrace{\sum_{v \in L - \{s\}} \left(f^{\text{out}}(v) - f^{\text{in}}(v) \right)}_{\text{conservation of flow}} + f^{\text{out}}(s) \\
&= f^{\text{out}}(s) = \text{size}(f)
\end{aligned}$$

What follows from this claim is that $\text{size}(f) = f^{\text{out}}(L) - f^{\text{in}}(L) \leq f^{\text{out}}(L) \leq \text{capacity}(L, R)$. Since this holds for any flow f and any st-cut (L, R), we can take the maximum of the left-hand side and the minimum of the right-hand side to arrive at the desired inequality max-flow $\leq$ min st-cut capacity.

max-flow $\geq$ min st-cut: Take the flow f^* output from the Ford-Fulkerson algorithm which has no st-path in its residual network G^{f^*}. We will construct an st-cut (L, R) such that $\text{size}(f^*) = \text{capacity}(L, R)$. It then follows that

$$\max_f \text{size}(f) \geq \text{size}(f^*) = \text{capacity}(L, R) \geq \min_{(L,R)} \text{capacity}(L, R)$$

To construct this st-cut, note that having no st-path in the residual network G^{f^*} means that t is not reachable from s in G^{f^*}. Hence we can define $L =$ vertices reachable from s in G^{f^*} and $R = V - L$. Then $s \in L$ while $t \in R$.
This st-cut has the following desired properties:

- for $\overrightarrow{vw} \in E$, $v \in L$, $w \in R$, $f^*_{vw} = c_{vw}$, because there is forward edge in the residual network for each not fully capacitated edge and that forward edge will connect s to w, contradicting with $w \in R$, it then follows that $\sum_{\overrightarrow{vw} \in E, v \in L, w \in R} f^*_{vw} = f^{*\text{out}}(L) = \text{capacity}(L, R)$
- for $\overrightarrow{wv} \in E$, $w \in R$, $v \in L$, $f^*_{wv} = 0$, because there is backward edge in the residual network for each edge with nonzero flow and that backward edge will connect s to w, contradicting with $w \in R$, it then follows that $\sum_{\overrightarrow{wv} \in E, w \in R, v \in L} f^*_{wv} = f^{*\text{in}}(L) = 0$

Combining them we have $\text{size}(f^*) = f^{*\text{out}}(L) - f^{*\text{in}}(L) = \text{capacity}(L, R)$.

14.3 Proof of Ford-Fulkerson Algorithm

Following the above claims, since max-flow $\leq$ min st-cut, the only way we can have equality $\text{size}(f^*) = \text{capacity}(L, R)$ is to have both sides optimal, i.e. f^* is a max-flow and (L, R) is a min st-cut. This proves the Ford-Fulkerson algorithm. In fact it proves any algorithm that stops when there is no augmenting path in the residual graph, e.g. the Edmond-Karp algorithm. The stopping condition takes $O(n + m)$ time to check whether a particular flow f is maximal or not, because it consists of building a residual network from current flow in $O(n + m)$ time and running DFS in $O(n + m)$ time to look for a st-path

in it.
The proof also provides a way of constructing a min st-cut—take a max-flow f^* and set $L =$ vertices reachable from s in its residual network. This will be utilized in the image segmentation application below.

15 Max-Flow: Image Segmentation

15.1 Image Segmentation

The goal of image segmentation is to separate an image into objects. The simpler task we will focus on is to separate an image into foreground and background. We will model an image as lying on a undirected graph with vertices being the pixels of the image and edges connecting neighboring pixels. The problem is formulated as follows:

input: undirected graph $G = (V, E)$, for each $i \in V$, let

- $f_i =$ likelihood/weight that i is in the foreground, $f_i \geq 0$
- $b_i =$ likelihood/weight that i is in the background, $b_i \geq 0$

for each $(i, j) \in E$, let $P_{ij} =$ separation penalty or the cost of separating i and j into different objects, $P_{ij} \geq 0$

goal: partition V into $V = F \cup B$ ($F =$ foreground, $B =$ background), for a partition (F, B) define its weight $w(F, B)$ as

$$w(F, B) = \sum_{i \in F} f_i + \sum_{j \in B} b_j - \sum_{\substack{(i,j) \in E \\ i \in F, j \in B}} P_{ij}$$

find a partition (F, B) with maximum weight

We will convert this maximization problem into a minimization problem in particular a min st-cut problem, for which we need to reformulate the weight definition as follows: let $L = \sum_{i \in V} (f_i + b_i)$ thus

$$\sum_{i \in F} f_i + \sum_{j \in B} b_j = L - \sum_{i \in B} f_i - \sum_{j \in F} b_j \implies$$

$$\begin{aligned} w(F, B) = \sum_{i \in F} f_i + \sum_{j \in B} b_j - \sum_{\substack{(i,j) \in E \\ i \in F, j \in B}} P_{ij} &= L - \sum_{i \in B} f_i - \sum_{j \in F} b_j - \sum_{\substack{(i,j) \in E \\ i \in F, j \in B}} P_{ij} \\ &= L - w'(F, B) \end{aligned}$$

where the new weight $w'(F, B)$ is defined as

$$w'(F, B) = \sum_{i \in B} f_i + \sum_{j \in F} b_j + \sum_{\substack{(i,j) \in E \\ i \in F, j \in B}} P_{ij}$$

$$\text{vs. } w(F, B) = \sum_{i \in F} f_i + \sum_{j \in B} b_j - \sum_{\substack{(i,j) \in E \\ i \in F, j \in B}} P_{ij}$$

Since $w(F, B) = L - w'(F, B)$, maximizing $w(F, B)$ is equivalent to minimizing $w'(F, B)$. In addition unlike $w(F, B)$ all terms in $w'(F, B)$ are positive.

15.2 Flow Network

We will reformulate this minimization problem into a min st-cut problem and solve it by finding a max-flow. Recall that the input to the image segmentation problem is a undirected graph $G = (V, E)$ while a flow network is a directed graph. Thus we need to define a directed graph $G' = (V', E')$ based on $G = (V, E)$ as follows:

- for $(i, j) \in E$, add $i \to j$ and $j \to i$ with capacity both equal to P_{ij}
- add s corresponding to the foreground pixels, for $i \in V$ add $s \to i$ of capacity f_i
- add t corresponding to the background pixels, for $i \in V$ add $i \to t$ of capacity b_i

For a cut (F, B) its capacity turns out to be exactly $w'(F, B)$ because

- for $i \in B$, we get $s \to i$ of capacity f_i
- for $j \in F$, we get $j \to t$ of capacity b_j
- for $(i, j) \in E$ with $i \in F$ and $j \in B$, we get $i \to j$ of capacity P_{ij}

capacity(F, B) sums them up and exactly equals to $w'(F, B) = \sum_{i \in B} f_i + \sum_{j \in F} b_j + \sum_{\substack{(i,j) \in E \\ i \in F, j \in B}} P_{ij}$.
In summary given input (G, f, b, P) for image segmentation, we can define a flow network (G', c) and get a max-flow f^* on this network, where size$(f^*) = \min_{(F,B)} w'(F, B)$. It follows that $\max_{(F,B)} w(F, B) = L - \min_{(F,B)} w'(F, B) = L - \text{size}(f^*)$. $\Rightarrow$ maximization $\Rightarrow$ minimization $\Rightarrow$ min-cut $\Rightarrow$ max-flow

16 Max-Flow: Edmonds-Karp Algorithm

16.1 Edmonds-Karp Algorithm

We first do a contrast of the Ford-Fulkerson and Edmonds-Karp algorithm:

Ford-Fulkerson: assume integral capacities, the main step is to find augmenting paths using DFS/BFS, the running time is $O(mC)$ where C = size of max-flow

Edmonds-Karp: no assumption on capacities, the main step is to find augmenting paths using BFS, the running time is $O(m^2 n)$

Hence the Edmonds-Karp algorithm is an example of the Ford-Fulkerson algorithm. Thus the running time analysis of the Ford-Fulkerson algorithm still applies to the Edmonds-Karp algorithm if we assume integral capacities, albeit we can make a stronger guarantee on its running time.
In fact the only distinction between the Edmonds-Karp algorithm and the Ford-Fulkerson algorithm lies in step 3, where

Ford-Fulkerson: use BFS/DFS to look for a st-path $\mathcal{P}$ in G^f, if there is no such path then output f

Edmonds-Karp: use BFS to look for a st-path $\mathcal{P}$ in G^f, if there is no such path then output f

One basic property of the Edmonds-Karp algorithm (and the Ford-Fulkerson algorithm as well) is that, the residual network has to change by at least one edge in every round. In particular at least one edge reaches full capacity, which corresponds to the one with minimum capacity $c(\mathcal{P})$ along the augmenting path. That fully capacitated edge will be removed from the residual network.
To show the running time is $O(m^2 n)$, we will show that the number of rounds run by the Edmonds-Karp algorithm is bounded by mn. Then since at each round we run BFS and BFS takes linear time,

which is $O(m)$ assuming the number of edges is at least the number of vertices, the total running time is $O(m) \times O(mn) = O(m^2 n)$.
To bound the number of rounds we will prove a lemma, which states that for every edge of the graph, the number of times it is deleted from the residual network and then re-inserted back later is at most $n/2$. It then follows that there are at most nm rounds because there are m edges and at least one edge is deleted in every round.
To prove the lemma recall that in BFS vertices are explored in layer or level. The level of a vertex is equal to the distance to it or the minimum number of edges from s to it. On the st-path $\mathcal{P}$ BFS finds from s to t the level goes up by 1 at every edge. We will show that for every vertex $z \in V$, level(z) does not decrease as G^f is updated.
To see why the level does not decrease, consider $\overrightarrow{vw} \in E$, possible residual network update for it include ($\Rightarrow$ reduced implies $\overrightarrow{wv} \in \mathcal{P}$ vs. augmented implies $\overrightarrow{vw} \in \mathcal{P}$)

1. add $\overrightarrow{vw}$ if the flow was full and then is reduced, implying $\overrightarrow{wv} \in \mathcal{P}$
2. remove $\overrightarrow{vw}$ if the flow is augmented to full, implying $\overrightarrow{vw} \in \mathcal{P}$
3. add $\overrightarrow{wv}$ if the flow was empty and then is augmented, implying $\overrightarrow{vw} \in \mathcal{P}$
4. remove $\overrightarrow{wv}$ if the flow was positive and then is reduced to empty, implying $\overrightarrow{wv} \in \mathcal{P}$

Hence if an edge is added to G^f then it corresponds to case 1 and 3 above. In both cases the opposite-direction edge is on the augmenting path. If an edge is removed from G^f then it corresponds to case 2 and 4 above. In both cases the edge itself is on the augmenting path.
With this observation we can then show that the level does not decrease. The level of a vertex z might decrease if we add an edge $\overrightarrow{yz}$ to give a shorter path from s to z. Suppose level$(z) = i$ and we add such a potential level-reducing edge $\overrightarrow{yz}$ to G^f. But we observed in the above that adding $\overrightarrow{yz}$ implies $\overrightarrow{zy} \in \mathcal{P}$, which in turn implies level(y) = level$(z) + 1 = i + 1$ by BFS. Hence the added edge goes from a higher level to a lower level thus will not decrease level(z).
But in order to bound the number of rounds the Edmonds-Karp algorithm runs we also need to show that occasionally the level of a vertex strictly increases. This happens when an edge is removed and re-inserted later. Say level$(v) = i$ initially. We then remove $\overrightarrow{vw}$ from G^f, which implies $\overrightarrow{vw} \in \mathcal{P}$ and hence level(w) = level$(v) + 1 \geq i + 1$ since we just showed level(v) never decreases. Suppose later we re-insert $\overrightarrow{vw}$ back to G^f, which implies $\overrightarrow{wv} \in \mathcal{P}$ and hence level(v) = level$(w) + 1 \geq i + 2$ since level(w) never decreases either. Therefore deletion + re-insertion increases the level of a vertex by at least 2. Because the minimum level is 0 (for source s) while the maximum level is n, we can delete and re-insert an edge for at most $n/2$ times. Since at least one edge is deleted in every round, it follows that there are at most mn rounds in the Edmonds-Karp algorithm where m is the number of edges.

17 Max-Flow: Generalization

17.1 Max-Flow with Demand

input: a directed graph $G = (V, E)$ and designated $s, t \in V$, a capacity $c(e) > 0$ for $e \in E$, a demand $d(e) > 0$ for $e \in E$

goal: whether a feasible flow from s to t exists, and if yes maximize it, a flow is **feasible** if $d(e) \leq f(e) \leq c(e)$ for $e \in E$

To reduce this feasible flow problem to a max-flow problem, we construct a max-flow input $(G', c'(e))$ from the feasible flow input $(G, c(e), d(e))$ as follows:

- for $e \in E$, add e to G' with $c'(e) = c(e) - d(e)$
- for $v \in V$, add $s' \to v$ with $c'(\overrightarrow{s'v}) = d^{\text{in}}(v)$, add $v \to t'$ with $c'(\overrightarrow{vt}) = d^{\text{out}}(v)$
- add $t \to s$ with $c'(\overrightarrow{ts}) = \infty$

The edge addition in the second step is to compensate for the capacity reduction by $d(e)$, while the edge addition in the third step is to relieve the conservation of flow constraint for s and t which become internal vertices in G'. Let $D = \sum_{e \in E} d(e)$. $D = \sum_{v \in V} d^{\text{in}}(v) = \sum_{v \in V} d^{\text{out}}(v)$ since every edge has exactly one tail and exactly one head. Notice that $c'^{\text{out}}(s') = \sum_{v \in V} d^{\text{in}}(v) = D$ and $c'^{\text{in}}(t') = \sum_{v \in V} d^{\text{out}}(v) = D$. Hence $\text{size}(f') \leq D$. We say f' is **saturating** if $\text{size}(f') = D$. Our goal is to show that G has a feasible flow if and only if G' has a saturating flow.

saturating ⇒ feasible: To construct a feasible flow f in G from a saturating flow f' in G', let $f(e) = f'(e) + d(e)$ for $e \in E$. f is

- feasible because $f'(e) \geq 0$ thus $f(e) \geq d(e)$
- valid because $f'(e) \leq c'(e)$ thus $f(e) \leq c'(e) + d(e) = c(e)$ and

$$f'^{\text{in}}(v) = f'(\overrightarrow{s'v}) + \sum_{w \in V} f'(\overrightarrow{wv}) = d^{\text{in}}(v) + \sum_{w \in V} \left(f(\overrightarrow{wv}) - d(\overrightarrow{wv})\right)$$
$$= d^{\text{in}}(v) + \sum_{w \in V} f(\overrightarrow{wv}) - d^{\text{in}}(v) = f^{\text{in}}(v)$$

$$f'^{\text{out}}(v) = f'(\overrightarrow{vt'}) + \sum_{w \in V} f'(\overrightarrow{vw}) = d^{\text{out}}(v) + \sum_{w \in V} \left(f\left(\overrightarrow{vw}\right) - d(\overrightarrow{vw})\right)$$
$$= d^{\text{out}}(v) + \sum_{w \in V} f(\overrightarrow{vw}) - d^{\text{out}}(v) = f^{\text{out}}(v)$$
$$\implies f^{\text{in}}(v) = f'^{\text{in}}(v) = f'^{\text{out}}(v) = f^{\text{out}}(v)$$

feasible ⇒ saturating: To construct a saturating flow f' in G' from a feasible flow f in G, let

- $f'(e) = f(e) - d(e)$ for $e \in E$
- $f'(\overrightarrow{s'v}) = d^{\text{in}}(v)$ and $f'(\overrightarrow{vt'}) = d^{\text{out}}(v)$ for $v \in V$ ⇒ saturating condition
- $f'(\overrightarrow{ts}) = \text{size}(f)$

f is valid because $d(e) \leq f(e) \leq c(e)$ implies $0 \leq f'(e) \leq c'(e) = c(e) - d(e)$ and by the algebra above $f'^{\text{in}}(v) = f^{\text{in}}(v) = f^{\text{out}}(v) = f'^{\text{out}}(v)$.

Hence to solve the feasible flow problem in G we construct G' as described above and run the max-flow algorithm. Then we check whether the size of the resulted max-flow equals D, i.e. whether it is saturating. If yes then we transform the max-flow in G' back to a feasible flow in G as described above. Once we obtain a feasible flow we can augment it to a maximum-sized feasible flow in the same manner as in the Ford-Fulkerson or Edmonds-Karp algorithm, but with the following modifications:

- start from the found feasible flow f instead of zero flow
- the capacity of the residual graph G^f is

$$c^f(\overrightarrow{vw}) = \begin{cases} c(\overrightarrow{vw}) - f(\overrightarrow{vw}) & \text{if } \overrightarrow{vw} \in E \\ f(\overrightarrow{wv}) - d(\overrightarrow{wv}) & \text{if } \overrightarrow{wv} \in E \\ 0 & \text{otherwise} \end{cases}$$

18 Randomized Algorithms: Modular Arithmetic

18.1 Modular Arithmetic

For an integer x, $x \bmod 2 =$ the least significant bit of x. For an integer N, $x \bmod N =$ the remainder when dividing x by N. $x \bmod N = r$ if $x = qN + r$ for some integer q. x and y are **congruent** mod N if $x \bmod N = y \bmod N$, denoted by $x \equiv y \bmod N$. One basic congruence property is that, if $x \equiv y \bmod N$ and $a \equiv b \bmod N$, then $x + a \equiv y + b \bmod N$ and $xa \equiv yb \bmod N$.
The basic modular arithmetic operation that we will do repeatedly is modular exponentiation: given n-bit integers x, y, and N, compute $x^y \bmod N$ in polynomial n time. The naive approach implements exponentiation as step-by-step multiplication, i.e. $x \bmod N = a_1$, $x^2 \bmod N = a_1 x \bmod N = a_2$, $x^3 \bmod N = a_2 x \bmod N = a_3$ and so on. Since it takes $O(n^2)$ time to multiply or divide two n-bit integers and there are $y \leq 2^n$ rounds of multiplication, the running time of this naive approach is $O(n^2 2^n)$.
We can speed up the exponentiation using repeated squaring instead of repeated multiplication, i.e. $x \bmod N = a_1$, $x^2 \bmod N = a_1^2 \bmod N = a_2$, $x^4 \bmod N = a_2^2 \bmod N = a_4$ and so on. Then for even y we compute $x^y = \left(x^{y/2}\right)^2$ and for odd y we compute $x^y = x\left(x^{\lfloor y/2 \rfloor}\right)^2$.

```
Mod-Exp(x, y, N)
    if y = 0 then
        return 1
    z = Mod-Exp(x, ⌊y/2⌋, N)
    if y is even then
        return z^2 mod N
    else
        return xz^2 mod N
```

Since it takes $O(n^2)$ time to multiply or divide two n-bit integers and there are $\log y \leq n$ rounds of multiplication, the running time is $O(n^3)$.

18.2 Multiplicative Inverse

x is the **multiplicative inverse** of $z \bmod N$ if $xz \equiv 1 \bmod N$, denoted by $x \equiv z^{-1} \bmod N$. Note that $x \equiv z^{-1} \bmod N$ and $z \equiv x^{-1} \bmod N$ are equivalent. For example $N = 14$, $3^{-1} \equiv 5 \bmod N$, $9^{-1} \equiv 11 \bmod N$, $13^{-1} \equiv 13 \bmod N$ while $2^{-1} \bmod N$, $4^{-1} \bmod N$, $6^{-1} \bmod N$, $7^{-1} \bmod N$, $8^{-1} \bmod N$ don't exist because they share a common divisor with N. We always report $x^{-1} \bmod N$ in $0, 1, \ldots, N-1$ if exists and does not exist otherwise. A general theorem states that $x^{-1} \bmod N$ exists if and only if $\gcd(x, N) = 1$ where gcd stands for greatest common divisor. When $\gcd(x, N) = 1$ we say x and N are **relatively prime**.
Suppose $x^{-1} \bmod N$ exists, then it must be unique. Suppose otherwise $y = x^{-1} \bmod N$ and $z = x^{-1} \bmod N$ and $y \not\equiv z \bmod N$, i.e. $0 \leq y \neq z \leq N - 1$. Apply the definition of multiplicative inverse $xy \equiv xz \equiv 1 \bmod N$. Now multiply both sides by x^{-1} we get $x^{-1}xy \equiv x^{-1}xz \bmod N$, which implies $y \equiv z \bmod N$ contradicting with our assumption.
We now prove the theorem that $x^{-1} \bmod N$ exists if and only if $\gcd(x, N) = 1$.

- if $\gcd(x, N) > 1$ then $x^{-1} \bmod N$ does not exist, suppose it exists and equals to z, let $\gcd(x, N)$ be r, then xz is certainly a multiple of r so is qN for any integer q, but then $xz \neq qN + 1$ since the left-hand side is divisible by r while the right-hand side is not, a contradiction
- if $\gcd(x, N) = 1$ then $x^{-1} \bmod N$ exists, we prove it by giving an algorithm to find it which is the extended Euclid algorithm

18.3 Extended Euclid's Algorithm

Euclid's rule states that for integers x, y where $x \geq y > 0$, $\gcd(x, y) = \gcd(x \bmod y, y)$. This follows from the fact that $\gcd(x, y) = \gcd(x - y, y)$, which is true because

- if d divides x and y, then d divides $x - y$
- if d divides $x - y$ and y, then d divides x

Utilizing Euclid's rule we devise **Euclid's GCD algorithm**

```
Euclid(x, y) where x ≥ y ≥ 0
    if y = 0 then
        return x
    else
        return Euclid(y, x mod y)
```

where the base case $\gcd(x, 0) = x$ can be shown by using Euclid's rule: $\gcd(x, 0) = \gcd(kx, x) = x$.
To compute the running time of Euclid's GCD algorithm, notice that the only nontrivial step in each round is computing $x \bmod y$ which takes $O(n^2)$ time for n-bit integers x and y, and the number of rounds the algorithm is recursively called is bounded by $2n$. This is because we have a lemma which states that if $x \geq y$ then $x \bmod y < x/2$. It implies that the input size is reduced by at least a factor of 2 after two rounds of recursive call

$$(x, y) \longrightarrow (y, x \bmod y) \longrightarrow (x \bmod y, y \bmod (x \bmod y)) \longrightarrow \cdots$$

Therefore the running time of Euclid's GCD algorithm is $O(n^3)$.
To prove the lemma we consider two cases:

$y \leq x/2$**:** $x \bmod y \leq y - 1 < y \leq x/2$

$y > x/2$**:** $x \bmod y = x - y < x - x/2 = x/2$

To compute multiplicative inverse we will use the following extended version of Euclid's GCD algorithm, which outputs $d = \gcd(x, y) = x\alpha + y\beta$ for some integer α, β

```
ExtendedEuclid(x, y) where x ≥ y ≥ 0
    if y = 0 then
        return (x, 1, 0)
    else
        (d, α', β') = ExtendedEuclid(y, x mod y)
        return (d, β', α' − ⌊x/y⌋β')
```

The running time is the same as Euclid's GCD algorithm, as each round still takes $O(n^2)$ time and the number of rounds is bounded by $n/2$.
With extended Euclid's algorithm we then can compute multiplicative inverse. If $d = \gcd(x, N) = 1$ then we have $1 = x\alpha + N\beta$ for some integer α and β. Taking mod N on both sides we get $1 \equiv x\alpha \bmod N$, which means $\alpha = x^{-1} \bmod N$. $\Rightarrow$ use Euclid to check whether inverse exists, use extended Euclid to find the inverse if exists

19 Randomized Algorithms: RSA

19.1 Fermat's Little Theorem

Fermat's little theorem states that, if p is prime then for every $1 \leq z \leq p-1$ we have $z^{p-1} \equiv 1 \mod p$. Note that if p is prime the condition $1 \leq z \leq p-1$ can be replaced with $\gcd(z,p) = 1$, which motivates its generalization to any integer N in Euler's theorem. This is the basis of both the RSA cryptosystem and the primality test.
To prove the theorem, let $S = \{1, 2, \ldots, p-1\}$ and construct $S' = zS \mod p = \{z \mod p, 2z \mod p, \ldots, (p-1)z \mod p\}$. S and S' are the same set with different orders because $|S'| = p-1$ and the elements of S' are

distinct: suppose $iz \equiv jz \mod p$ for some $i \neq j$, since p is prime and $\gcd(z,p) = 1$, $z^{-1} \mod p$ exists, thus we can multiply both sides of $iz \equiv jz \mod p$ with z^{-1} to get $izz^{-1} \equiv jzz^{-1} \mod p$, which implies $i \equiv j \mod p$ contradicting with $1 \leq i \neq j \leq p-1$

nonzero: suppose $iz \equiv 0 \mod p$, again multiply both sides by z^{-1} to get $i \equiv 0 \mod p$ which contradicts with $1 \leq i \leq p-1$

Now that we have shown $S = S'$, we multiply all elements of S together and do the same for S' which should give us an equality, in particular the two products read

$$\underbrace{1 \times 2 \times 3 \times \cdots \times (p-1)}_{\text{product of elements of } S} = \underbrace{1z \times 2z \times 3z \times \cdots \times (p-1)z \mod p}_{\text{product of elements of } S'}$$

$$\implies (p-1)! \equiv z^{p-1}(p-1)! \mod p$$

Since p is prime, 1^{-1}, 2^{-1}, 3^{-1}, $\ldots$, $(p-1)^{-1} \mod p$ exist. We can then multiply their inverses on both sides of the above equation to cancel $(p-1)!$ and arrive at $1 \equiv z^{p-1} \mod p$.

19.2 Euler's Theorem

We will actually use a generalization of Fermat's little theorem called **Euler's theorem**, which states that for any integer N and z where $\gcd(z,N) = 1$, $z^{\phi(N)} \equiv 1 \mod N$ where $\phi(N) =$ the number of integers between 1 and N which are relatively prime to N. This $\phi(N)$ is called *Euler's totient function.* For prime p, $\phi(p) = p-1$. Hence Euler's theorem reduces to Fermat's little theorem for primes. We will apply Euler's theorem for $N = pq$ where both p and q are primes. $\phi(N) = pq-p-q+1 = (p-1)(q-1)$ where we subtract q multiples of p and p multiples of p and $+1$ to correct the double subtraction of pq. Thus for any z where $\gcd(z,pq) = 1$ we have $z^{(p-1)(q-1)} \equiv 1 \mod pq$. This fact is the basis for the RSA algorithm.

19.3 RSA

The basic idea in terms of Fermat's little theorem is for prime p take two integers b, c where $bc \equiv 1 \mod p-1$ so that $z^{bc} \equiv z \times (z^{p-1})^k \mod p = z \mod p$, where $z \mapsto z^c$ is encryption and $z^c \mapsto (z^c)^b \equiv z$ is decryption. We want to refrain from disclosing the prime p thus we apply Euler's theorem instead: for primes p and q take two integers d, e where $de \equiv 1 \mod (p-1)(q-1)$ so that $z^{de} \equiv z \times \left(z^{(p-1)(q-1)}\right)^k \mod pq = z \mod pq$, where $z \mapsto z^e$ is encryption and $z^e \mapsto (z^e)^d \equiv z$ is decryption.
The general cryptography setting consisting of

Alice: encrypt a message $e : m \mapsto e(m)$ and send the encrypted message $e(m)$ to Bob over a communication line

Bob: receive the encrypted message and decrypt it to recover the original message $d : e(m) \mapsto m$

Eve: can eavesdrop and see what is transmitted over the communication line, but doesn't know how to decrypt the message

This is a **public-key** cryptosystem where no communication needs to happen between Alice and Bob in private. Bob will compute and publish a public key (N, e) where $N = pq$ and e is relatively prime to $(p-1)(q-1)$. He will also compute and keep to himself a private key $d = e^{-1} \bmod (p-1)(q-1)$. Only Bob can compute d because only he knows p and q.
In summary the RSA protocol receiver I consists of

1. Bob picks two n-bit random primes p and q (random n bits of 1's and 0's + primality test)
2. Bob chooses e relatively prime to $(p-1)(q-1)$ (try 3, 5, 7, etc. small e for easy encryption)
3. Bob publishes his public key (N, e) and computes his private key $d \equiv e^{-1} \bmod (p-1)(q-1)$

The RSA sender protocol sender consists of

1. Alice looks up Bob's public key (N, e)
2. Alice computes $y \equiv m^e \bmod N$ (use fast modular exponentiation)
3. Alice sends y

The RSA sender protocol receiver II consists of

1. Bob receives y
2. Bob decrypts y by computing $y^d \equiv (m^e)^d \equiv m^{de} \equiv m \bmod N$, this holds even when $\gcd(m, N) > 1$ (proved using Chinese remainder theorem)

There are in fact some pitfalls associated with the above RSA protocol.

gcd$(m, N) > 1$: Since $N = pq$, if $\gcd(m, N) > 1$ then $\gcd(m, N) = p$ or q. Without loss of generality assume it is p. Since $\gcd(m, N) > 1$ we cannot apply Euler's theorem to prove $(m^e)^d \equiv m \bmod N$ but have to use Chinese remainder theorem. More importantly since m is divisible by p so is $y \equiv m^e \bmod N$ thus $\gcd(y, N) = p$. This gives a way for the eavesdropper to learn p and factorize N into p and q to break the RSA protocol.

m **not too large:** need $m < N$, the binary version of a text is usually a huge number, we can break this huge number into n-bit segments each of length n thus $m < 2^n$, then to ensure $m < N$ we can choose p and q sufficiently large so that $N \geq 2^n$

m **not too small:** common practice is to use $e = 3$, but if $m^3 < N$ then $y = m^3 \bmod N = m^3$, thus taking cubic root would decrypt it $m = y^{1/3}$, to avoid it we can pad the small message with a random string

send the same message multiple times: if the eavesdropper learns multiple encryption of the same message $y_1 \equiv m^3 \bmod N_1$, $y_2 \equiv m^3 \bmod N_2$, $y_3 \equiv m^3 \bmod N_3$, then he can use Chinese remainder theorem to decrypt m

19.4 Primality Testing

To recap, RSA consists of the following steps

1. randomly choose primes p and q and let $N = pq$
2. find e where $\gcd(e, (p-1)(q-1)) = 1$
3. compute $d \equiv e^{-1} \bmod (p-1)(q-1)$ using extended Euclid's algorithm
4. publish public key (N, e)
5. encrypt m by $y \equiv m^e \bmod N$
6. decrypt y by $m \equiv y^d \bmod N$ using fast modular exponentiation

The RSA algorithm is as hard as factorizing N into p and q.
To randomly choose p and q, let r be a random n-bit number, check if r is prime. If yes then output r; if no then repeat the random n-bit number generation. Luckily primes are dense in the sense that the probability of r being prime is roughly $1/n$. To perform the primality test we use Fermat's little theorem: if r is prime then $z^{r-1} \equiv 1 \bmod r$ for all z in $\{1, 2, \ldots, r-1\}$; if there is z where $z^{r-1} \not\equiv 1 \bmod r$ then r is composite. We call this z a **Fermat witness** because it is a witness with respect to Fermat's little theorem.
Every composite r has at least two Fermat witnesses. Simply take z to be a divisor of r, then $\gcd(z, r) = z > 1$ which implies that z does not have multiplicative inverse by Euclid's GCD algorithm. $z^{r-1} \not\equiv 1 \bmod r$ because otherwise $z^{r-1} \equiv z^{r-2}z \equiv 1 \bmod r$ would imply z^{r-2} is the multiplicative inverse of z which is a contradiction. This z is called *trivial Fermat witness* because $\gcd(z, r) > 1$. There also exists *nontrivial Fermat witness* z with $\gcd(z, r) = 1$. Some composites have no nontrivial Fermat witness which are called *Carmichael numbers* or *pseudoprimes*. It is inefficient to use Fermat's primality test for them. Fortunately Carmichael numbers are rare, and if a composite has a nontrivial Fermat witness then it has many Fermat witnesses which are dense and efficient to test by Fermat's test.
If r has at least one nontrivial Fermat witness, then at least half of $z \in \{1, 2, \ldots, r-1\}$ are Fermat witnesses (ref. book for proof). Based on this fact and ignoring Carmichael numbers we can devise the following simple primality test: for an n-bit r

1. choose z randomly from $\{1, 2, \ldots, r-1\}$
2. compute $z^{r-1} \bmod r$
3. if $z^{r-1} \equiv 1 \bmod r$ then output r is prime; else output r is composite

For prime r the probability that this test outputs correctly r is prime is 1, whereas for composite and non-Carmichael r the probability that this test outputs incorrectly r is prime is bounded by $1/2$ because at least half of $z \in \{1, 2, \ldots, r-1\}$ are Fermat witnesses.
To reduce the false positive rate, we choose $z_1, z_2, \ldots, z_k$ randomly from $\{1, 2, \ldots, r-1\}$. If any z_i is a Fermat witness we then output r is composite. Hence the test is modified to:

1. choose $z_1, z_2, \ldots, z_k$ randomly from $\{1, 2, \ldots, r-1\}$
2. compute $z_i^{r-1} \bmod r$ for $i = 1$ to k
3. if $z_i^{r-1} \equiv 1 \bmod r$ for all i then output r is prime; else output r is composite

For composite and non-Carmichael r the probability this test outputs incorrectly that r is prime is now bounded by $(1/2)^k$.
To test Carmichael numbers efficiently we follow almost the same procedure, i.e. choose z randomly from $\{1, 2, \ldots, r-1\}$ and compute z^{r-1}. But during the fast modular exponentiation z^{r-1}, trace back to the first exponent of z that is not 1. This is a nontrivial root of unity mod r, e.g. $1065^2 \equiv 1 \bmod 1729$. Because it turns out that prime p only has the trivial square roots of unity 1 and -1 mod p, if there is nontrivial square root of unity mod r then r must be a composite. This works for at least 3/4 of the choices of z.

20 NP: Definitions

20.1 NP Problems

Define NP to be the class of all search problems. An alternative definition is the class of decision problems. But the appeal of search problems is that it gets rid of the need for a witness for particular instances. A rough definition of search problem is a problem where we can verify solutions in polynomial time. Note that in the definition of NP the time it takes to generate the solution doesn't matter. It matters in the definition of P, which is the class of search problems that are solvable in polynomial time. Of course if we can generate a solution in polynomial time, we can verify it in polynomial time. Thus P is a subset of NP, P $\subseteq$ NP, that is any problem that can be solved in P can also be solved in NP. Therefore P = NP means that it is as difficult to solve a problem or generate a proof as to verify its solution or check the proof. It appears the opposite to the majority of people, that is P $\neq$ NP.
Formally a search problem is a problem of the following form:

form: given instance I, find a solution S for I if one exists, or output no if I has no solution

requirement: given an instance I and a solution S, we can verify that S is a solution to I in polynomial time, that is the running time is a polynomial of $|I|$, to show that this requirement is fulfilled we need to show that there exists an algorithm that can verify the solution in polynomial time

Some examples of NP are

SAT: defined as

> **input:** Boolean formula f in CNF with n variables and m clauses
>
> **output:** satisfying assignment if one exists and no otherwise
>
> it belongs to NP because it is of the correct form, more importantly given f and an assignment to x_1, x_2, …, x_n it takes $O(n)$ time to check that one clause is satisfied and there are m clauses, hence the total time to verify is $O(nm)$

k-coloring: defined as

> **input:** undirected $G = (V, E)$ and integer $k > 0$ where k is the number of colors available
>
> **output:** assignment to each vertex a color in $\{1, 2, \ldots, k\}$ so that adjacent vertices get different colors if one exists and no otherwise
>
> it belongs to NP because it is of the correct form, more importantly given G and a coloring scheme it takes $O(m)$ time to check that for every edge $(v, w) \in E$ color of $v \neq$ color of w

MST: defined as

input: $G = (V, E)$ with positive edge lengths

output: tree T with minimum weight

it belongs to NP because it is of the correct form as the problem is formulated such that there always is a solution, more importantly given G and a tree T we can run

- $O(m + n)$ time BFS/DFS to check whether it is a tree (connected + acyclic)
- $O(m \log n)$ time Kruskal's/Prim's algorithm to check whether it has the minimal weight (not necessarily the same tree but the minimal weight is known)

in fact MST is also in P as it can be solved in polynomial time using Kruskal's/Prim's algorithm

knapsack: defined as

input: n objects with integer weights $\omega_1, \omega_2, \ldots, \omega_n$ and integer values $v_1, v_2, \ldots, v_n$, a capacity B

output: a subset S of objects with total weight $\sum_{i \in S} w_i \leq B$ and maximal total value $\max\{\sum_{i \in S} v_i\}$

it is not known to be in NP because we don't know a polynomial-time algorithm that can check whether the total value is maximal (recall that the dynamical programming solution is $O(nB)$ not $\text{poly}(n, \log B)$), it is not known to be in P either because we don't know a polynomial-time algorithm that can generate the maximal value subset, this argument applies to both the with repetition and without repetition version

knapsack-search: variant of knapsack that is a search problem, defined as

input: n objects with integer weights $\omega_1, \omega_2, \ldots, \omega_n$ and integer values $v_1, v_2, \ldots, v_n$, a capacity B, a goal g

output: a subset S of objects with total weight $\sum_{i \in S} w_i \leq B$ and total value $\sum_{i \in S} v_i \geq g$ and no otherwise

it belongs to NP because it is of the correct form, more importantly given subset S it takes $O(n)$ time to check whether $\sum_{i \in S} w_i \leq B$ and $\sum_{i \in S} v_i \geq g$ (more precisely $O(n \log W)$ and $O(n \log V)$ but the input size is $\log W$ and $\log V$ thus the running time is still polynomial in the input size) note that if we can solve this knapsack search version in polynomial time, then we can solve the original optimization version in polynomial time, because

- we can do binary search over g to find the maximal g that has a solution, which tells us the maximal total value we can achieve
- since total value is bounded by $V = \sum_{i \in S} v_i$, the total number of rounds of the binary search is $O(\log V)$
- because the size of the input to represent $v_1, v_2, \ldots, v_n$ is $\log V$, this binary search is polynomial in the input size

20.2 NP Completeness

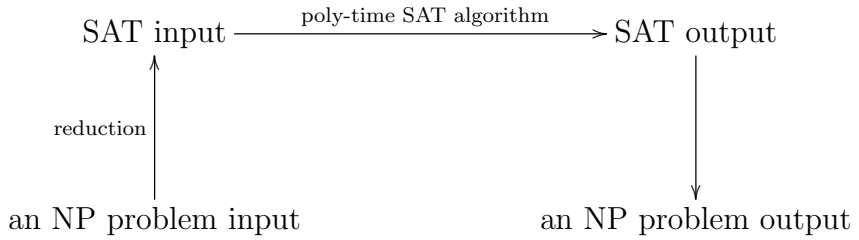
P stands for polynomial time while NP stands for nondeterministic polynomial time rather than not polynomial time. NP is a class of problems that can be solved in polynomial time on a nondeterministic machine. A **nondeterministic machine** is a machine that is allowed to guess at each step, and there is a series of choices of branching which lead to an accepting state.
Suppose P $\neq$ NP meaning there are some search problems which can't be solved in polynomial time.

They are called **NP-complete** problems and are guaranteed to lie in $NP - P$ if they exist. NP-complete problems are the hardest in the class NP, hardest in the sense that it is least likely to have an efficient solution, more precisely

- if P $\neq$ NP then all NP-complete problems are not in P
- if an NP-complete problem can be solved in polynomial time, then all NP problems can be solved in polynomial time i.e. P = NP

Hence to show a problem such as SAT is NP-complete, we need to show that if there is a polynomial-time algorithm for SAT then there is a polynomial-time algorithm for all problems in the class NP, in other words the polynomial-time algorithm that solves SAT can be used as a black box to solve all NP problems in polynomial time. To do that we need to show that every NP problem has a reduction to SAT.
SAT is NP-complete means that

- SAT $\in$ NP
- if we can solve SAT in polynomial time then we can solve every NP problem in polynomial time (hardest), this requires every NP problem to have a reduction to SAT



Thus if P $\neq$ NP then SAT $\notin$ P.

20.3 Reduction

A reduction from Problem A to Problem B is denoted by $A \to B$. An alternative notation is $A \leq B$, which means that if a reduction from A to B exists then it implies that B is computationally at least as hard as A, because if we can solve B then we can solve A as well. Formally a **reduction** from Problem A to Problem B means that if we can solve Problem B in polynomial time, then we can use that algorithm to solve Problem A in polynomial time:

$$\begin{array}{l}\text{poly-time}\\ \text{algorithm for A}\end{array} : I \xrightarrow{f} f(I) \longrightarrow \begin{array}{lllll}\text{poly-time} & \longrightarrow & S & \xrightarrow{h} & h(S)\\ \text{algorithm for B} & \longrightarrow & \text{no} & \longrightarrow & \text{no}\end{array}$$

To show such a reduction exists we need to define

f**:** input for Problem A $\mapsto$ input for Problem B

h**:** solution for Problem B $\mapsto$ solution for Problem A

and show that

- Problem A has a solution if and only if Problem B has a solution
- S is a solution for Problem B if and only if $h(S)$ is a solution for Problem A

With reduction we can reformulate our definition of NP completeness as follows: to show the independent set problem or IS for short is a NP-complete problem, we need to show that

- IS $\in$ NP
- $\forall$A $\in$ NP, A $\rightarrow$ IS

There is a simplified proof of the second requirement if we know another NP-complete problem. For example suppose we know SAT is NP-complete which means that $\forall$A $\in$ NP, A $\rightarrow$ SAT. Then to show $\forall$A $\in$ NP, A $\rightarrow$ IS it suffices to show that SAT $\rightarrow$ IS, because we can compose the reduction A $\rightarrow$ SAT $\rightarrow$ IS. Therefore to show IS is NP-complete we can show that

- IS $\in$ NP
- a known NP-complete problem such as SAT $\rightarrow$ IS $\Rightarrow$ note the direction of reduction

21 NP: 3-SAT

21.1 Reduction: Input

We will use the Cook-Levin theorem (1971) which proves that SAT is NP-complete to prove that 3-SAT is NP-complete. The importance of NP completeness was highlighted by Karps in 1972 who showed another 21 problems are NP-complete including 3-SAT.
The 3-SAT problem is defined as follows:

input: a Boolean formula f in CNF with n variables and m clauses where each clause has ≤ 3 literals

output: satisfying assignment if one exists and no otherwise

To show 3-SAT is NP-complete we need to show

- 3-SAT $\in$ NP
- SAT $\rightarrow$ 3-SAT

3-SAT belongs to NP because it is of the correct form, more importantly given Boolean formula f and an True/False assignment σ for $x_1, x_2, \ldots, x_n$, it takes $O(1)$ time to check that at least one literal is satisfied in each clause $C \in f$ because each of them has at most three literals. Hence overall it takes $O(m)$ time to verify the assignment satisfies f.
To show SAT $\rightarrow$ 3-SAT we need to

- create input f' for 3-SAT from input f for SAT
- transform satisfying assignment σ' for f' to satisfying assignment σ for f, and show σ' satisfies f' if and only if σ satisfies f so that no instance for 3-SAT corresponds to no instance for SAT

To demonstrate how to create input f' for 3-SAT from input f for SAT, consider $f = (x_3) \wedge (x_1 \vee x_2) \wedge (\bar{x}_2 \vee x_3 \vee \bar{x}_1 \vee \bar{x}_4)$, create a new variable y and rewrite the last clause as $(\bar{x}_2 \vee x_3 \vee y) \wedge (\bar{y} \vee \bar{x}_1 \vee \bar{x}_4)$. $C_3 = (\bar{x}_2 \vee x_3 \vee \bar{x}_1 \vee \bar{x}_4)$ is satisfiable if and only if $C_3' = (\bar{x}_2 \vee x_3 \vee y) \wedge (\bar{y} \vee \bar{x}_1 \vee \bar{x}_4)$ is satisfiable, because

$\Rightarrow$**:** consider two cases:

x_2 **= False or** x_3 **= True:** C_3' is satisfied by setting y = False
x_1 **= False or** x_4 **= False:** C_3' is satisfied by setting y = True

$\Rightarrow$: consider two cases:

y = **True:** x_1 = False or x_4 = False, each of which satisfies C_3

y = **False:** x_2 = False or x_3 = True, each of which satisfies C_3

Similarly to create input C' for 3-SAT from a 5-literal clause $C = (\bar{x}_2 \vee x_3 \vee \bar{x}_1 \vee \bar{x}_4 \vee x_5)$ we create 2 new variables y and z and define C' as $C' = (\bar{x}_2 \vee x_3 \vee y) \wedge (\bar{y} \vee \bar{x}_1 \vee z) \wedge (\bar{z} \vee \bar{x}_4 \vee x_5)$.

$C \Rightarrow C'$: if C is satisfied then at least one of the five literals is satisfied, then the remaining clauses of C' can be taken care by y and z

$C' \Rightarrow C$: if y = False then x_2 = False or x_3 = True each of which satisfies C
if z = True then x_4 = False or x_5 = True each of which satisfies C
if y = True and z = False then x_1 = False which satisfies C

In general for a size-k clause $C = (a_1 \vee a_2 \vee \cdots \vee a_k)$ where $a_1, a_2, \ldots, a_k$ are literals, create $k-3$ new variables $y_1, y_2, \ldots, y_{k-3}$ and replace C with the following $k-2$ clauses:

$$C' = (a_1 \vee a_2 \vee y) \wedge (\bar{y}_1 \vee a_3 \vee y_2) \wedge (\bar{y}_2 \vee a_4 \vee y_3) \wedge \cdots \wedge (\bar{y}_{k-4} \vee a_{k-2} \vee y_{k-3}) \wedge (\bar{y}_{k-3} \vee a_{k-1} \vee a_k)$$

We show that C is satisfiable if and only if C' is satisfiable:

$C \Rightarrow C'$: take assignment σ to $a_1, a_2, \ldots, a_k$ that satisfies C, let a_i be the minimal i where a_i is satisfied, since a_i = True the $(i-1)$th clause of C' is satisfied, then we can set $y_1 = y_2 = \cdots = y_{i-2}$ = True to satisfy the clauses before the $(i-1)$th clause and $y_{i-1} = y_i = \cdots = y_{k-2}$ = False to satisfy the clauses after the $(i-1)$th clause

$C' \Rightarrow C$: take assignment σ' to $a_1, a_2, \ldots, a_k, y_1, y_2, \ldots, y_{k-3}$ that satisfies C', it suffices to show at least one of $a_1, a_2, \ldots, a_k$ is True, suppose no then from the first clause we have y_1 = True, which in turn implies y_2 = True as a_3 = False, continue this inference until the penultimate clause we have $y_3 = y_4 = \cdots = y_{k-3}$ = True, but then we have $\bar{y}_{k-3} = a_{k-1} = a_k$ = False thus the last clause and hence C' is not satisfied, which is a contradiction, therefore at least one of $a_1, a_2, \ldots, a_k$ is True and thus C is satisfied

In summary we can create input f' for 3-SAT from input f for SAT by the following procedure

1. for each clause $C \in f$
   - if $|C| \leq 3$ then add C to f'
   - if $|C| > 3$ then create $k-3$ auxiliary variables to form C' as above and add C' to f'

f is satisfiable if and only if f' is satisfiable, because

$\Rightarrow$: given assignment σ to $x_1, x_2, \ldots, x_n$ satisfying f, for clause $C \in f$ if $|C| \leq 3$ then it is trivially satisfied; if $|C| = k > 3$ then we construct C' as above using $k-3$ auxiliary variables which only appears in C' thus can be set to satisfy C' independently

$\Leftarrow$: given assignment σ' satisfying f', we have proven by contradiction that for $C' \in f$ at least one literal in C is satisfied, hence f is satisfied

Hence if there is no satisfying assignment for f' then there is no satisfying assignment for f.

21.2 Reduction: Output

Suppose the polynomial-time 3-SAT algorithm gives us a satisfying assignment σ' for f', then the above proof by contradiction shows that we can simply ignore the auxiliary variables to get a satisfying assignment σ for f.
Note that f has n variables and m clauses. In the worst case

- we may create n auxiliary variables for each clause thus f' has $O(nm)$ variables
- we may replace every clause by size-$O(n)$ clause thus f' has $O(nm)$ clauses

which is OK because the size of f' is polynomial in the size of f, thus an algorithm that is polynomial in the size of f' is polynomial in the size of f as well.

22 NP: Graph Problems

22.1 Independent Set

For an undirected graph $G = (V, E)$ a subset $S \subset V$ is an **independent set** if no edges are contained in S. To show that S is an independent set we need to show that $\forall x, y \in S$, $(x, y) \notin E$. Any singleton vertex is an independent set. The maximal independent set problem, denoted as max-IS, is known to be not in NP, because there is no known way to verify a given independent set is maximal in polynomial time ($\Rightarrow$ cf. knapsack, is there a poly-time way to verify maximal for any problem?). The only way we can verify it is of maximal size is if we can solve the problem in polynomial time. But there is a simple fix so that the problem is in NP—the search version of the independent set problem (cf. knapsack). Similar to knapsack, the search version of the independent set problem is defined as follows:

input: an undirected graph $G = (V, E)$ and a goal g

output: independent set S with size $|S| \geq g$ if one exists and no otherwise

This version of the independent set problem is actually NP-complete. To prove it we will show

- IS $\in$ NP, which is true because given input (G, g) and a solution S
  1. we can check $\forall x, y \in S$, $(x, y) \notin E$ in $O(n^2)$ time
  2. we can check $|S| \geq g$ in $O(n)$ time
- 3-SAT $\rightarrow$ IS, consider a 3-SAT input f with variable $x_1, x_2, \ldots, x_n$ and clause $C_1, C_2, \ldots, C_m$, each clause has a size $|C_i| \leq 3$, we will define an undirected graph G and set a goal $g = m$, the idea is to create $|C_i|$ vertices for each clause C_i and add edges to encode this clause and its dependence on other clauses, since there are m clauses there will be at most $3m$ vertices

The reduction from 3-SAT to IS requires two types of edges

clause edge: for a clause $C = (x_1 \vee \bar{x}_3 \vee x_2)$ we have three vertices x_1, $\bar{x}_3$, x_2, note that if some of these literals say x_2 appear in other clause then we will have another vertex corresponding to x_2, thus there are multiple vertices corresponding to the same literal, to encode this clause we add edges between all pairs of these three vertices i.e. fully connected, thus

- an independent set S will have at most one vertex per clause in this graph
- since $g = m$ a solution S will have exactly one vertex per clause in this graph, which ensures that we have at least one satisfied literal in every clause

variable edge: $\forall x_i$ add an edge between x_i (from one clause) and $\bar{x}_i$ (from another) to ensure that any independent set contains x_1 or $\bar{x}_1$ or neither thus corresponds to a valid assignment

We need to prove that a 3-SAT input f has a satisfying assignment if and only if the corresponding undirected graph G has an independent set of size $\geq g$.

$\Rightarrow$**:** take a satisfying assignment σ for f, for each clause C take one of the satisfied literals and add the corresponding vertex to S, thus $|S| = m = g$ and we only need to show that S is an independent set, which is true because

- S contains exactly one vertex per clause thus it contains no clause edge
- S never contains both x_i and $\bar{x}_i$ thus it contains no variable edge

$\Leftarrow$**:** take an independent set S of size $\geq g$, S has exactly one vertex per clause because $|S| \geq g = m$, set the corresponding literal to be True so as to satisfy the clause that contains it, since every clause is satisfied the formula f is satisfied, it is a valid assignment because S contains no contradictory literals x_i and $\bar{x}_i$ constrained by the variable edges

The proof of f has a satisfying assignment $\Leftarrow$ the undirected graph G has an independent set of size $\geq g$ above also defines the transformation of the output $h : S \mapsto h(S)$. Therefore the search version of the independent set problem is NP-complete.
Note that it is straightforward to reduce the above search version of the independent set problem to the maximal independent set problem, because we only need to check whether the maximal size $\geq g$ or not. Since the search version of the independent set problem is NP-complete, this implies that every NP problem has a reduction to max-IS. Thus max-IS is at least as hard as every problem in the class NP. Hence if we know it is in NP then it must be NP-complete. We say such a problem is **NP-hard**. The difference between NP-hard and NP-complete is that NP-complete requires both NP-hard and the fact that the problem is in NP.

22.2 Clique

For an undirected graph $G = (V, E)$, a subset $S \subset V$ is a **clique** if $\forall x, y \in S$, $(x, y) \in E$ i.e. S forms a fully connected subgraph of G. Any singleton vertex by itself is a clique, and the two endpoints of an edge also form a clique. The (search version of) clique problem is defined as follows:

input: an undirected graph $G = (V, E)$ and a goal g

output: $S \subset V$ where S is a clique of size $|S| \geq g$ if one exists and no otherwise

We will show that this clique problem is NP-complete, for which we need to show

- clique $\in$ NP, which is true because given input (G, g) and solution S,
  1. it takes $O(n^3)$ time ($O(n)$ time for each pair and $O(n^2)$ pairs) to check $\forall x, y \in S$, $(x, y) \in E$ (in fact we can do it in $O(n^2)$ time but $O(n^3)$ is also polynomial time)
  2. it takes $O(n)$ time to check that $|S| \geq g$
- the NP-complete IS $\rightarrow$ clique, the key idea is that clique where all edges are within S is opposite to independent set where no edges are within S, thus for $G = (V, E)$ we define its opposite graph $\bar{G} = (V, \bar{E})$ where $\bar{E} = \{(x, y) : (x, y) \notin E\}$, such that S is a clique in $\bar{G}$ if and only if S is an independent set in G
  then given input $G = (V, E)$ and goal g for the IS problem, let $\bar{G}$ and g be an input to the clique problem, since S is a clique in $\bar{G}$ if and only if S is an independent set in G, if we get a solution S for the clique problem then we just return S as a solution for the IS problem, and we output no if no solution exists for the clique problem

22.3 Vertex Cover

For an undirected graph $G = (V, E)$, a subset $S \subset V$ is a **vertex cover** if it covers every edge, cover in the sense that at least one of the two endpoints of each edge is in S, i.e. $\forall (x, y) \in E$, either $x \in S$ and/or $y \in S$.
The vertex cover problem is defined as:

input: undirected graph $G = (V, E)$ and budget b

output: vertex cover S of size $|S| \leq b$ if one exists and no otherwise

We will show that the vertex cover problem or VC for short is NP-complete, for which we need to show

- VC $\in$ NP, which is true because given input (G, b) and solution S
  - it takes $O(n + m)$ time to check for every edge $(x, y) \in E$ at least one of x and y are in S
  - it takes $O(n)$ time to check whether $|S| \leq b$
- the NP-complete IS $\rightarrow$ vertex cover, the key idea is that the complement of a vertex cover form an independent set, thus S is a vertex cover if and only if $\bar{S}$ is an independent set

  vertex cover $\Rightarrow$ independent set: take a vertex cover S, $\forall (x, y) \in E$ at least one of x and y are in S, thus at most one of x and y are in $\bar{S}$ which means that no edge is contained in $\bar{S}$, hence $\bar{S}$ is an independent set

  vertex cover $\Leftarrow$ independent set: take an independent set $\bar{S}$, $\forall (x, y) \in E$ at most one of x and y are in $\bar{S}$, thus at least one of x and y are in S which means that S covers every edge

  then given input $G = (V, E)$ and goal g for the IS problem, let $b = n - g$ and run the VC problem on G and b, since G has a vertex cover of size $\leq n - g$ if and only if G has an independent set of size $\geq g$, if we get a solution S for the VC problem then we return $\bar{S}$ as a solution for the IS problem, and we output no if no solution exists for the VC problem

In summary there are roughly two flavors of NP-completeness reductions:

proof by generalization: show that the new problem is more general than a known problem

proof by gadget: modify input by introducing auxiliary variables to the formula or additional structure to the graph

23 Linear Programming: Introduction

23.1 Definition

Linear programming handles any problem that can be formulated as an optimization over a set of variables with a goal known as the **objective function** subject to **constraints** that can be expressed as linear functions of the variables. For example the max-flow problem defined as

input: a directed graph $G = (V, E)$ and designated $s, t \in V$, a capacity $c_e > 0$ for each $e \in E$

goal: maximize the flow from s to t without exceeding capacities, specified by flow f_e for each $e \in E$

has the following linear programming (LP) formulation: define m variables f_e for $e \in E$

objective function: $\max \sum_{\overrightarrow{sv} \in E} f_{sv}$

constraint: (1) $0 \leq f_e \leq c_e$ for $e \in E$ (2) $\sum_{\overrightarrow{wv} \in E} f_{wv} = \sum_{\overrightarrow{vz} \in E} f_{vz}$ for $v \in V - \{s, t\}$

It is a LP problem because

- the objective function is a max or a min of a linear function of the variables
- the constraints are linear functions of the variables

23.2 Geometric View

As a concrete example consider the following production problem: a company makes only two products A and B which satisfy

A: cost = 1 hour, profit = 1 dollar, demand $\leq$ 300 units

B: cost = 3 hours, profit = 6 dollars, demand $\leq$ 200 units

How many of each to produce in order to maximize the profit given the total supply of 700 hours? Let x_1 = the number of units of A to produce and x_2 = the number of units of B to produce. The corresponding LP problem can be formulated as follows:

objective function: $\max\{x_1 + 6x_2\}$

constraints: (1) $x_1 \geq 0$ (2) $x_2 \geq 0$ (3) $x_1 \leq 300$ (4) $x_2 \leq 200$ (5) $x_1 + 3x_2 \leq 700$

We are in 2D because we have two variables. Each of the five constraints is a half-plane. The intersection of these five half-planes gives us a set of feasible $x = (x_1, x_2)$ called **feasible region**. Then we will look into that feasible set to find the $x = (x_1, x_2)$ that maximizes the objective function. The objective function can be re-written as $\max_C\{x_1 + 6x_2 = C\}$. Hence we can move up the line $x_1 + 6x_2 = C$ until the last point where it intersects with the feasible region, which is at $(x_1, x_2) = (100, 200)$.
The key points are

ILP is NP-complete: Linear programming optimizes over a polygon intersection thus is polynomial-time solvable, hence LP $\in$ P. However the corresponding integer-valued problem, that is we only want integer points, also called **integer linear programming** or ILP is NP-complete. As such if the optimum is not an integer value, then we can try to round it to an integer point.

optimum at corner: The optimum always lies at a vertex of the polygon intersection. Suppose no i.e. the optimal line intersects at a point z which is not a corner. But then either one of the two endpoints of the edge z lies in must be better than z, or the optimal line intersects with the entire edge in which case all the points on the edge is optimal. In either case the optimal point lies at a vertex of the polygon intersection.

feasible region is convex: The line connecting any two points in the region is entirely contained in the region. Therefore if a point is better than its neighbors then it is optimal, because the feasible region is below the lines connecting it to its neighbors. Hence the optimum always lies at a vertex of the feasible region. This is the basis for the simplex algorithm.

Consider a slight twist to the above 2D example: suppose the company now produces three products A, B, and C which satisfy

A: cost = 1 hour, profit = 1 dollar, demand $\leq$ 300 units, no need package

B: cost = 3 hours, profit = 6 dollars, demand $\leq$ 200 units, need 1 package

C: cost = 2 hours, profit = 10 dollars, unlimited demand, need 3 packages

How many of each to produce in order to maximize the profit given the total supply of 1000 hours and 500 packages?
Let x_1 = the number of units of A to produce, x_2 = the number of units of B to produce, x_3 = the number of units of C to produce. The corresponding LP problem can be formulated as follows:

objective function: $\max\{x_1 + 6x_2 + 10x_3\}$

constraints: (1) $x_1 \geq 0$ (2) $x_2 \geq 0$ (3) $x_3 \geq 0$ (4) $x_1 \leq 300$ (5) $x_2 \leq 200$ (6) $x_1 + 3x_2 + 2x_3 \leq 1000$ (7) $x_2 + 3x_3 \leq 500$

We are in 3D because we have three variables. Each of the seven constraints is a half-space, and the intersection of these seven half-spaces gives us a set of feasible $x = (x_1, x_2, x_3)$.

23.3 Standard Form

The standard form for linear programs has n variables $x_1, x_2, \ldots, x_n$, and

objective function: $\max\{c_1x_1 + c_2x_2 + \cdots + c_nx_n\}$, or $\max\{c^Tx\}$ where $x = (x_1, x_2, \ldots, x_n)^T$ and $c = (c_1, c_2, \ldots, c_n)^T$

constraint: $x_1, x_2, \ldots, x_n \geq 0$ and

$$\begin{aligned} a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n &\leq b_1 \\ a_{21}x_1 + a_{22}x_2 + \cdots + a_{2n}x_n &\leq b_2 \\ \vdots \qquad\qquad & \quad \vdots \\ a_{n1}x_1 + a_{n2}x_2 + \cdots + a_{nn}x_n &\leq b_n \end{aligned}$$

or $x \geq 0$ and $Ax \leq b$ where

$$A = \begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ & & \vdots & \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{pmatrix} \qquad \text{and} \qquad b = \begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{pmatrix}$$

The non-negativity constraint is important, because if the feasible region is non-empty then we know that the zero vector is a feasible point. Hence we can trivially find a feasible point, or determine that the feasible region is empty and therefore the LP is infeasible.
To convert an arbitrary linear program into this standard form, we can use the following equivalences

- $\min\{c^Tx\} \Leftrightarrow \max\{-c^Tx\}$
- $a_1x_1 + a_2x_2 + \cdots + a_nx_n \geq b \Leftrightarrow -a_1x_1 - a_2x_2 - \cdots - a_nx_n \leq -b$
- $a_1x_1 + a_2x_2 + \cdots + a_nx_n = b \Leftrightarrow a_1x_1 + a_2x_2 + \cdots + a_nx_n \leq b$ and $a_1x_1 + a_2x_2 + \cdots + a_nx_n \geq b$
- unconstrained $x \Leftrightarrow$ replace x with $x = x^+ - x^-$ where $x^+, x^- \geq 0$

We are in an n-dimensional space because we have n variables. We have $m + n$ constraints including n non-negativity constraints, each of which corresponds to a half-space in the n-dimensional space. The feasible region is the intersection of these $n + m$ half-spaces, which corresponds to a convex polyhedron in the n-dimensional space. The vertices of this polyhedron are the points that satisfy n constraints

with equality as well as the remaining m constraints. The number of choices of these n constraints is $\binom{n+m}{n}$, therefore the upper bound of the number of vertices is exponential in n. In addition a neighboring vertex corresponds to swapping out one of the n constraints with equality and swapping in a different constraint. Since we have n choices for which constraint to swap out and m choices for which constraint to swap in, the number of neighbors for a given vertex is bounded by nm.
Strict inequalities are not allowed in linear programming. Because for non-strict inequalities the feasible region corresponds to a closed convex polyhedron, thus the optimal point lies on the boundary of this polyhedron. However for strict inequalities the points on the boundary of the polyhedron do not lie in the feasible region.

23.4 LP Algorithms

There are two types of algorithms that are guaranteed to solve linear programs in polynomial time in the worst case—ellipsoid algorithms and interior point methods. Ellipsoid algorithms are more of theoretical interest whereas interior point methods are used quite extensively. Another important algorithm is the simplex algorithm. Although it takes exponential time in the worst case, it is widely used because

- the output of the simplex algorithm is guaranteed to be optimal
- it works quite efficiently on humongous LPs

The basic idea of the simplex algorithm is to start at some feasible point e.g. the zero vector $x = 0$. This is a vertex of the feasible region because it satisfies the n non-negativity constraints with equality. Of course we still need to check whether $x = 0$ satisfies the other m constraints. If it does not satisfy any of the m constraints, then the feasible region is empty and thus the LP is infeasible. If it does then we do a local search around the feasible point to try to find a neighboring vertex with strictly higher objective value. If we managed to find one then we move there and repeat the local search. If there are multiple neighboring vertices with higher objective values, then we have several options—choose randomly or take the neighboring vertex with the highest objective value. This is one of the heuristics of the variants of the simplex algorithm. If all neighbors are smaller than the current feasible point, by convexity all the feasible region is smaller than it. Hence the current feasible point must be optimal.

24 Linear Programming: Geometry

24.1 Geometry

Consider a linear program in standard form $\max\{c^T x\}$ subject to $Ax \leq b$ and $x \geq 0$. Each constraint defines a half-space in n dimensions. Their intersection forms the feasible region which is a convex polyhedron in n dimensions. The simplex algorithm walks on the vertices/corners of this polyhedron. The optimum of LP is achieved at a vertex of the feasible region except if

the LP is infeasible: the feasible region is empty, resulted from the half-spaces defined by two constraints do not intersect

the LP is unbounded: the optimum is arbitrarily large, resulted from the optimum being a function of the objective

While whether an LP is feasible or infeasible depends only on the constraints, whether an LP is bounded or unbounded depends on the objective function as well.

24.2 Feasibility Check

To find out whether there is some x that can satisfy the constraints $Ax \leq b$ and $x \geq 0$, we introduce a new variable z and consider the following LP problem: $\max\{z\}$ subject to $Ax + z \leq b$ and $x \geq 0$ leaving z unconstrained. Since z is unconstrained there is always a solution to the new LP problem, as we can simply let z to be a vector of very small numbers to satisfy the constraints. The key is whether we can find a solution to this LP problem where $z \geq 0$. If yes then there is always a solution to the original constraints. Therefore to check whether the original LP is feasible, we can run the new LP to check whether the optimal value of z is non-negative. If yes then the point x satisfying the constraints involving z gives a feasible point to the original LP.

25 Linear Programming: Duality

25.1 Duality

To show a solution is optimal we can try to take a linear combination of the constraints to show that the solution is the upper bound of the objective function. For example consider the following LP problem and its solution:

objective function: $\max\{x_1 + 6x_2 + 10x_3\}$

constraints: (1) $x_1 \leq 300$ (2) $x_2 \leq 200$ (3) $x_1 + 3x_2 + 2x_3 \leq 100$ (4) $x_2 + 3x_3 \leq 500$

solution $x = (x_1, x_2, x_3) = (200, 200, 100)$ for which $x_1 + 6x_2 + 10x_3 = 2400$

Consider the following linear combination $y = (y_1, y_2, y_3, y_4) = (0, 1/3, 1, 8/3)$ of the four constraints:

$$\begin{aligned} & y_1 \times (1) + y_2 \times (2) + y_3 \times (3) + y_4 \times (4) \\ \iff \quad & x_1 (y_1 + y_3) + x_2 (y_2 + 3y_3 + y_4) + x_3 (2y_3 + 3y_4) \leq 300y_1 + 200y_2 + 1000y_3 + 500y_4 \\ \iff \quad & x_1 + 6x_2 + 10x_3 \leq 2400 \end{aligned}$$

This shows that the objective value corresponding to the optimum is the upper bound of the objective function, hence it must be optimal.
Actually we just need the linear combination to be at least the objective function, because the objective value of the optimum would still be able to serve as the upper bound of the objective function. This translates to the following LP problem:

objective function: $\min\{300y_1 + 200y_2 + 1000y_3 + 500y_4\}$

constraints: (1) $y_1 + y_3 \geq 1$ (2) $y_2 + 3y_3 + y_4 \geq 6$ (3) $2y_3 + 3y_4 \geq 10$ (4) $y_1, y_2, y_3, y_4 \geq 0$

The original LP is maximization while the new LP is minimization. Moreover the number of variables in the original LP defines the number of constraints in the new LP, and the number of constraints in the original LP without counting the non-negativity constraints defines the number of variables in the new LP. The new LP is called the **dual LP** while the original LP is called the **primal LP**.
In general to apply the duality as illustrated above we need the primal LP in the canonical form:

objective function: $\max\{c^T x\}$

constraints: $Ax \leq b$ and $x \geq 0$

which gives the following dual LP in canonical form:

objective function: $\min\{b^T y\}$

constraints: $A^T y \geq c$ and $y \geq 0$

While the primal LP has n variables and m constraints besides the non-negativity constraints, the dual LP has m variables and n constraints besides the non-negativity constraints.
As a quick sanity check, take a primal LP in the canonical form and take the dual twice. The resulted LP should be equivalent to the original LP:

$$\begin{array}{l} \max\{c^T x\} \\ Ax \leq b \\ x \geq 0 \end{array} \xrightarrow{\text{dual}} \begin{array}{l} \min\{b^T y\} \\ A^T y \geq c \\ y \geq 0 \end{array} \xrightarrow{\text{standard form}} \begin{array}{l} \max\{-b^T y\} \\ -A^T y \geq -c \\ y \geq 0 \end{array} \xrightarrow{\text{dual}} \begin{array}{l} \min\{-c^T z\} \\ -Az \geq -b \\ z \geq 0 \end{array}$$

25.2 Unbounded Check

The **weak duality theorem** states the following: take a feasible point x for the primal LP the objective value of which is $c^T x$; take a feasible point y for the dual LP the objective value of which is $b^T y$. Since any feasible y gives an upper bound on the objective function of the primal LP, we have $c^T x \leq b^T y$. There are two corollaries of this theorem.

corollary #1: if we find a feasible point x for the primal LP and a feasible point y for the dual LP such that $c^T x = b^T y$, then x and y are both optimal

corollary #2: if the primal LP is unbounded then the dual LP is infeasible as nothing can be larger than infinity, similarly if the dual LP is unbounded then the primal LP is infeasible as nothing can be smaller than negative infinity
note that this is not an equivalence, if the dual LP is infeasible then the primal LP can be either unbounded or infeasible

The second corollary above provides a method to check whether an LP is unbounded—if the dual LP is infeasible and the primal LP is feasible (as we know how to check whether an LP is feasible), then the primal LP must be unbounded. The existence of the optimal x and y in the first corollary is mandated by the **strong duality theorem**, which states that the primal LP is feasible and bounded if and only if the dual LP is feasible and bounded, or equivalently the primal LP has an optimal point x^* if and only if the dual LP has an optimal point y^*. By the first corollary of the weak duality theorem we have $c^T x^* = b^T y^*$. Hence for every LP given the optimal solution x^* there is always a certificate from its dual LP which certifies that this is an optimal solution.
One nice implication of this strong duality theorem is that if we write the LP for the max-flow problem, then the objective value for that LP is the size of the max-flow from s to t. Moreover its dual LP turns out to be the capacity of the min st-cut. Hence the max-flow min-cut theorem can also be proved by the strong duality theorem.

26 Linear Programming: Max-SAT Approximation

26.1 Max-SAT

Max-SAT is the optimization version of the search problem SAT. It is defined as

input: a Boolean formula f in CNF with n Boolean variables x_1, x_2, ..., x_n and m clauses

output: the assignment that maximizes the number of satisfied clauses

Max-SAT is NP-hard. It is not NP-complete because it is not a search problem thus not in NP. More precisely we have no way to verify that the number of satisfied clauses is maximal. Nonetheless max-SAT is at least as hard as SAT, because it is straightforward to reduce SAT to max-SAT and therefore max-SAT is NP-hard. Hence we can't hope to solve max-SAT in polynomial time. Instead we will aim at an approximate solution, and to do so we will use linear programming.

26.2 Approximate Max-SAT

For a Boolean formula f with m clauses, let $m^* = m^*(f)$ be the maximal number of satisfied clauses. Clearly $m^* \leq m$. We will construct an algorithm on input f which outputs l—the number of satisfied clauses (actually an assignment that satisfies l clauses of f), where $l \geq m^*/2$. If this holds for every Boolean formula f, then we call it a $\frac{1}{2}$**-approximation algorithm**.

26.3 Random Approximation Algorithm

Consider an input f in CNF with n Boolean variables $x_1, x_2, \ldots, x_n$ and m clauses $C_1, C_2, \ldots, C_m$. We implement the following random assignment: set $x_i = \text{True}$ with $1/2$ probability and False with $1/2$ probability for all $i = 1, 2, \ldots, n$. Let W be the number of clauses satisfied by this random assignment. Since the assignment is random, so is W. The expectation of W is given by

$$\mathbb{E}[W] = \sum_{l=0}^{m} l \times \Pr(W = l)$$

To avoid the complication caused by the interdependence among clauses, we define a random variable W_j for each clause C_i, which takes value 1 if C_j is satisfied and 0 otherwise. W_j is related to W by $W = \sum_{j=1}^{m} W_j$. Hence the expectation of W can be expressed as

$$\mathbb{E}[W] = \mathbb{E}\left[\sum_{j=1}^{m} W_j\right] = \sum_{j=1}^{m} \mathbb{E}[W_j]$$

where the last equality uses the linearity of expectation, and

$$\mathbb{E}[W_j] = 1 \times \Pr(W_j = 1) + 0 \times \Pr(W_j = 0) = \Pr(W_j = 1) = 1 - \frac{1}{2^k} \geq \frac{1}{2}$$

where k is the number of literals contained in C_j. Because there are 2^k possible assignments to these k literals, and only one of which makes C_j False (i.e. False $\vee$ False $\vee \cdots \vee$ False = False). Plug this into the expectation of W we have

$$\mathbb{E}[W] = \sum_{j=1}^{m} \mathbb{E}[W_j] \geq \frac{m}{2} \geq \frac{m^*}{2}$$

Hence the randomized algorithm achieves 1/2-approximation in expectation. It turns out that we can de-randomize it to arrive at a deterministic algorithm that guarantees 1/2-approximation. This can be achieved by using conditional expectation as follows:

```
for i = 1 → n do
    compute the expected performance W_iT conditioned on x_i = True
    compute the expected performance W_iF conditioned on x_i = False
    if W_iT >= W_iF then
        set x_i = True
    else
        set x_i = False
```

An important special case is so-called Ek-SAT, which stands for exactly-k-SAT, i.e. every clause has a size of exactly k. For max-Ek-SAT the randomized algorithm achieves a $(1 - 2^{-k})$-approximation. In particular it achieves a 7/8-approximation for max-E3-SAT. In fact Hovstad proved that this 7/8-approximation is the best possible for max-E3-SAT, because it is NP-hard to do any better than the 7/8-approximation for max-E3-SAT, meaning if we achieved an approximation that is better than 7/8 for max-E3-SAT then that would imply P = NP.

26.4 Integer Linear Programming

The canonical form of integer linear programming or ILP differs from that of LP by only one additional constraint—$x \in \mathbb{Z}^n$ or each x_i is an integer. LP has a nice property that there always is a vertex of the feasible region which is an optimal point. Due to the additional integral constraint ILP no longer has that property. Moreover while LP is in P, ILP is NP-hard. To show this we will show that max-SAT reduces to ILP:

1. take input f for max-SAT, for each variable x_i add integral variable y_i to ILP and for each clause C_j add integral variable z_j to ILP so that ILP has $n + m$ integral variables

2. add constraints $0 \leq y_i \leq 1$ and $0 \leq z_j \leq 1$, since $y_i, z_j \in \mathbb{Z}$ they take value 0 or 1, intuitively $y_i = 1$ corresponds to $x_i =$ True and $z_j = 1$ corresponds to C_j being satisfied

3. for each clause C_j let C_j^+ be the positive literals in C_j and C_j^- be the negative literals in C_j, for a Boolean formula f in CNF define an ILP $\max \sum_{j=1}^m z_j$ subject to $0 \leq y_i \leq 1 \ \forall i = 1, 2, \ldots, n$ and $0 \leq z_j \leq 1 \ \forall j = 1, 2, \ldots, m$ and

$$\sum_{i \in C_j^+} y_i + \sum_{i \in C_j^-} (1 - y_i) \geq z_j$$

   so that

   - if all literals in C_j are unsatisfied, then z_j is forced to take value 0 i.e. the clause is unsatisfied
   - if at least one literal is satisfied, then z_j will take value 1 because $0 \leq z_j \leq 1$, $z_j \in \mathbb{Z}$, and ILP maximizes z_j

26.5 LP Relaxation

Take the ILP max-SAT reduces to and let y^*, z^* be its optimal point. Then m^*, the maximum number of satisfied clauses in f, is given by $m^* = z_1^* + z_2^* + \cdots + z_m^*$. Dropping the integral constraints $y_i, z_j \in \mathbb{Z}$ we convert the ILP to an LP, which can be solved in polynomial time. Let $\widehat{y}^*, \widehat{z}^*$ be the optimal point of the resulted LP. Because the optimal point for the ILP is also a feasible point for the LP, we have $m^* = z_1^* + z_2^* + \cdots + z_m^* \leq \widehat{z}_1^* + \widehat{z}_2^* + \cdots + \widehat{z}_m^*$.
We will convert the LP optimal solution into a feasible point for the ILP, which is not necessarily the optimal solution to the ILP. The simplest way to do so is to convert the fractional points $\widehat{y}^*, \widehat{z}^*$ to an integral point by rounding them to the nearest integer point, and we will do this in a probabilistic way. Then we will prove that this rounded integer point is not far away from the optimal solution to the LP, thus is not far away from the optimal solution to the ILP because the former is the upper bound of the latter.
Given the optimal solution for the LP $\widehat{y}^*, \widehat{z}^*$ we use **randomized rounding**: since $0 \leq \widehat{y}^* \leq 1$ we can think of it as a probability and set $y_i = 1$ with probability $\widehat{y}^*$ and 0 with probability $1 - \widehat{y}^*$. Recall that the expectation of W, the number of satisfied clauses, is given by

$$\mathbb{E}[W] = \sum_{j=1}^m \mathbb{E}[W_j] = \sum_{j=1}^m \Pr(C_j \text{ is satisfied})$$

We will prove the lemma that

$$\Pr(C_j \text{ is satisfied}) \geq \left(1 - \frac{1}{e}\right) \widehat{z}_j^*$$

which can be interpreted as the probability of satisfying C_j after rounding is at least $(1-1/e)$ of the probability of satisfying C_j before rounding by the LP. Plugging it in the expectation of W becomes

$$\mathbb{E}[W] = \sum_{j=1}^{m} \mathbb{E}[W_j] = \sum_{j=1}^{m} \Pr(C_j \text{ is satisfied}) \geq \left(1 - \frac{1}{e}\right) \sum_{j=1}^{m} \widehat{z}_j^* \geq \left(1 - \frac{1}{e}\right) \sum_{j=1}^{m} z_j^* = \left(1 - \frac{1}{e}\right) m^*$$

To prove the lemma consider a clause $C_j = x_1 \vee x_2 \vee \cdots \vee x_k$. The LP constraint for this clause is that $\widehat{y}_1^* + \widehat{y}_2^* + \cdots + \widehat{y}_k^* \geq \widehat{z}_j^*$ which forces the clause to be False when all literals are False. It follows that

$$\begin{aligned} \Pr(C_j \text{ is satisfied}) = 1 - \Pr(C_j \text{ is unsatisfied}) &= 1 - \prod_{i=1}^{k} (1 - \widehat{y}_i^*) \\ &\geq 1 - \left[\frac{1}{k} \sum_{i=1}^{k} (1 - \widehat{y}_i^*)\right]^k = 1 - \left(1 - \frac{1}{k} \sum_{i=1}^{k} \widehat{y}_i^*\right)^k \end{aligned}$$

where we have used the arithmetic mean-geometric mean or AM-GM inequality (let $w_i = 1 - \widehat{y}_i^*$)

$$\frac{1}{k} \sum_{i=1}^{k} w_i \geq \left(\prod_{i=1}^{k} w_i\right)^{1/k} \qquad \text{for } w_1, w_2, \ldots, w_k \geq 0$$

Since $\widehat{y}_1^* + \widehat{y}_2^* + \cdots + \widehat{y}_k^* \geq \widehat{z}_j^*$, we have

$$\Pr(C_j \text{ is satisfied}) \geq 1 - \left(1 - \frac{1}{k} \sum_{i=1}^{k} \widehat{y}_i^*\right)^k \geq 1 - \left(1 - \frac{\widehat{z}_j^*}{k}\right)^k$$

To bring $\widehat{z}_j^*$ outside of the exponential, we apply calculus: define $f(\alpha) = 1 - (1 - \alpha/k)^k$, we claim that $f(\alpha) \geq \left[1 - (1-1/k)^k\right] \alpha$, because the inequality holds for $\alpha = 0$ and $\alpha = 1$ and $f''(\alpha) < 0$. Hence

$$\Pr(C_j \text{ is satisfied}) \geq 1 - \left(1 - \frac{\widehat{z}_j^*}{k}\right)^k \geq \left[1 - \left(1 - \frac{1}{k}\right)^k\right] \widehat{z}_j^* \geq \left(1 - \frac{1}{e}\right) \widehat{z}_j^*$$

where the last inequality follows from the Taylor series expansion of $e^{-\beta} \geq 1 - \beta$.
In summary we can approximately solve an NP-hard problem by

1. reduce it to ILP
2. relax the resulted ILP problem to an LP problem by dropping the integral constraint
3. apply randomized rounding to the optimal LP solution to obtain a feasible point of the ILP

26.6 Combined Algorithm

We compare the performance of the two max-SAT algorithms on Ek-SAT

k	randomized	LP-based
1	1/2	1
2	3/4	3/4
3	7/8	$(1-2/3)^3 \approx 0.704$
k	$1 - 2^{-k}$	$1 - (1-1/k)^k$

Note that the maximum of each row $\geq 3/4$. This motivates combining these two algorithms as follows:

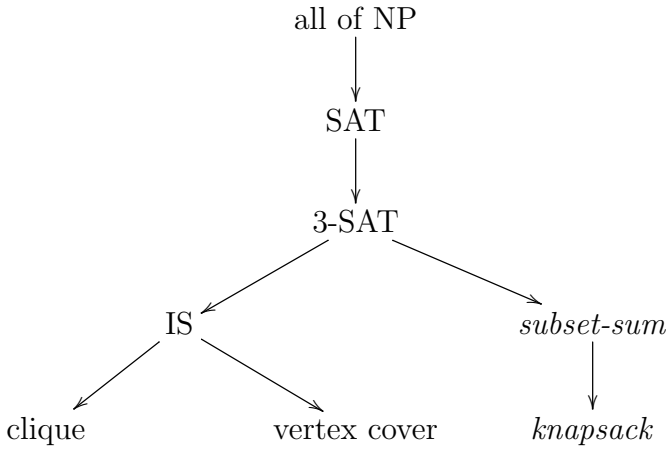
1. run the randomized algorithm to get an assignment that satisfies m_1 clauses
2. run the LP-based algorithm to get an assignment that satisfies m_2 clauses
3. take the better assignment of the two

The expected performance of this combined algorithm is $\mathbb{E}[\max\{m_1, m_2\}] \geq (3/4)m^*$, which guarantees a 3/4-approximation for max-SAT.

27 NP: Knapsack

27.1 Subset-Sum

To recap what we have proved to be NP-complete, we start from SAT which we take for granted, then



Those in italic are what we are going to prove.
The subset-sum problem is defined as

input: positive integers $a_1, a_2, \ldots, a_n$ and integer t

output: output subset S of $\{1, 2, \ldots, n\}$ where $\sum_{i \in S} a_i = t$ if such a subset exists and no otherwise

Similar to knapsack subset-sum has an $O(nt)$ DP solution, which is not polynomial in the input size. Hence subset-sum is not known to be in P. It is in NP instead because given input $a_1, a_2, \ldots, a_n$ and t and a solution S, it takes $O(n \log t)$ to check that $\sum_{i \in S} a_i = t$ since a_i has at most $\log t$ bits.
To show subset-sum is NP-complete we will show that 3-SAT reduces to it. The input to subset-sum consists of $2n + 2m + 1$ variables $v_1, v_1', v_2, v_2', \ldots, v_n, v_n', s_1, s_1', s_2, s_2', \ldots, s_m, s_m'$ and t. All are $\leq n + m$ digits long, and are base 10 so that there is no carry between digits thus all digits behave independently of each other. Moreover

- v_i corresponds to x_i, and $v_i \in S \iff x_i = \text{True}$
- v_i' corresponds to $\bar{x}_i$, and $v_i' \in S \iff x_i = \text{False}$

thus we need to ensure that exactly one of v_i or v_i' is in S. To do so we can put 1 in the ith digit of v_i, v_i' and t and 0 in the ith digit of all other numbers, thus the only way to achieve t which has 1 in its ith digit is to include either v_i or v_i' in S. Therefore this specification ensures that a solution to the subset-sum problem corresponds to an assignment.
For the remaining digits let the $(n + j)$th digit corresponds to clause C_j so that

- if $x_i \in C_j$ then put 1 in the $(n+j)$th digit for v_i
- if $\bar{x}_i \in C_j$ then put 1 in the $(n+j)$th digit for v'_i

To ensure that at least one of the literals in C_j is satisfied, we put 3 in the $(n+j)$th digit of t and use s_j, s'_j as buffers—put 1 in the $(n+j)$th digit of s_j, s'_j and 0 in the $(n+j)$th digit of all other numbers. Hence

- if all three literals are satisfied, then we get the desired 3 in the $(n+j)$th digit of t
- if only one or two of these literals are satisfied, then using the buffers we get the desired 3
- if none of these literals are satisfied, then there is no way to achieve 3 in the $(n+j)$th digit for t

We show that the subset-sum has a solution if and only if the input to 3-SAT is satisfiable

$\Rightarrow$: take a solution S to subset-sum, for the first n digits to get 1 in the ith digit of t we need to include v_i or v'_i but not both, and which one to include depends on the assignment to x_i, hence we get an assignment
for the remaining m digits to get 3 in the $(n+j)$th digit of t, we need to include at least one literal of C_j and use s_j, s'_j if necessary, because S is a solution there must be at least one satisfied literal in C_j, thus C_j is satisfied and hence all clauses are satisfied and we get a satisfying assignment

$\Leftarrow$: take a satisfying assignment for f, if $x_i =$ True then add v_i to S, otherwise add v'_i to S, hence the ith digit of t is correct
because it is a satisfying assignment each clause C_j has at least one satisfied literal, using this satisfied literal together with the buffers s_j, s'_j we get 3 in the $(n+j)$th digit of t, and therefore we have a solution to subset-sum

Using subset-sum being NP-complete, we can further prove that the knapsack problem is NP-complete by showing that subset-sum reduces to it.

28 NP: Halting Problem

28.1 Undecidable

NP-complete describes computationally difficult problems. If P $\neq$ NP then there is no algorithm which takes polynomial time on every input. There may be algorithm which takes polynomial time on some inputs or even almost all inputs, but it cannot guarantee polynomial time on every input. Undecidable describes computationally impossible problems. If a problem is undecidable, then there is no algorithm that can solve it on every input even given unlimited time and space. Turing proved in 1936 that the halting problem is undecidable. This same paper introduced the notation of Turing Machine, which captures the power of a conventional computer. What Turing showed is that the halting problem is undecidable on a Turing Machine.

28.2 Halting Problem

The halting problem is defined as

input: a program P in any language with an input I

output: True if P(I) ever terminates, False if P(I) never terminates i.e. it has an infinite loop

For example the following program P

$$\text{while } x\%2 == 1$$
$$x = x + 6$$

we have Halting(P, 5) = False.
We prove that the halting problem is undecidable by contradiction. Suppose we had an algorithm that solves the halting problem on every input. We call this algorithm Terminator(P, I). We will construct a new program Q and input J and show that Terminator(Q, J) is incorrect.
Consider the following "evil" program:

```
Harmful(J)
(1)   if Terminator(J, J)
          then go to (1)
      else
          return ()
```

- if J(J) terminates, then Harmful(J) never terminates
- if J(J) never terminates, then Harmful(J) terminates

But if J = Harmful, then we have

- if Harmful(Harmful) terminates, then Harmful(Harmful) never terminates
- if Harmful(Harmful) never terminates, then Harmful(Harmful) terminates

Since either case leads to contradiction, the assumption that Terminator solves the halting problem on every input is false.