CSC373 Algorithms

Jonah Chen

Contents

1 Divide and Conquer

- Divide and Conquer algorithm:
  1. Divide problem of size n into a smaller subproblems of size n/b each
  2. Recursively solve each subproblem
  3. Combine the subproblem solutions into the solution of the original problem
- Runtime: $T(1) = c$ and $T(n) = aT(n/b) + cn^d$ for $n > 1$
- Master Theorem: $T(n)$ depends on relation between a and b^d.

$$\begin{cases} a < b^d : T(n) = \Theta(n^d) \\ a = b^d : T(n) = \Theta(n^d \log n) \\ a > b^d : T(n) = \Theta(n^{\log_b a}) \end{cases} \tag{1}$$

  - Note that the running time does not depend on the constant c
  - In many algorithms $d = 1$ (combining take linear time)
- Examples:
  - Merge sort — sorting array of size n ($a = 2, b = 2, d = 1 \to a = b^d$) so $T(n) = \Theta(n \log n)$
  - Binary search — searching sorted array of size n ($a = 1, b = 2, d = 0 \to a = b^d$) so $T(n) = \Theta(\log n)$

1.1 Karatsuba Multiplication

- **Add** two binary n-bit numbers naively takes $\Theta(n)$ time
- **Multiply** two binary n-bit numbers naively takes $\Theta(n^2)$ time
- Divide and Conquer approaches
  1. Multiply x and y. We can divide them into two parts

$$x = x_1 \cdot 2^{n/2} + x_0 \tag{2}$$
$$y = y_1 \cdot 2^{n/2} + y_0 \tag{3}$$
$$x \cdot y = x_1 \cdot y_1 \cdot 2^n + (x_1 \cdot y_0 + x_0 \cdot y_1) \cdot 2^{n/2} + x_0 \cdot y_0 \tag{4}$$

     - $T(n) = 4T(n/2) + cn; T(1) = c$
     - $a = 4, b = 2, d = 1$ Master Theorem case 3, $T(n) = \Theta(n^{\log_2 4}) = \Theta(n^2)$.
     - This is the same complexity of the naive approach, making this approach useless.
  2. Reconsider (4), we may rewrite $(x_1 \cdot y_0 + x_0 \cdot y_1)$ as $(x_1 + x_0) \cdot (y_1 + y_0) - x_1 \cdot y_1 - x_0 \cdot y_0$

$$x \cdot y = x_1 \cdot y_1 \cdot 2^n + ((x_1 + x_0) \cdot (y_1 + y_0) - x_1 \cdot y_1 - x_0 \cdot y_0) \cdot 2^{n/2} + x_0 \cdot y_0 \tag{5}$$

     - $T(n) = 3T(n/2) + cn; T(1) = c$
     - $a = 3, b = 2, d = 1$, Master Theorem case 3, $T(n) = \Theta(n^{\log_2 3}) = \Theta(n^{\log_2 3}) \approx \Theta(n^{1.585})$
     - Minor issue: a carry may increase $x_1 + x_0$ and $y_1 + y_0$ to $\frac{n}{2} + 1$. We can easily prove this by isolating the carry bit and reevaluating the complexity.
- To deal with integers which doesn't have a power of 2 number of bits, we can pad the numbers with 0s to make them have a power of 2 number of bits. This may at most increase the complexity by 3x.
- 1971: $\Theta(n \cdot \log n \cdot \log \log n)$
- 2019: Harvey and van der Hoeven $\Theta(n \log n)$. We do not know if this is optimal.

1.2 Strassen's MatMul Algorithm

- Let A and B be two $n \times n$ matrices (for simplicity n is a power of 2), we want to compute $C = AB$.
- The naive approach takes $\Theta(n^3)$ time.
  1. Divide A and B into 4 submatrices of size $\frac{n}{2} \times \frac{n}{2}$ each

$$A = \begin{bmatrix} A_1 & A_2 \\ A_3 & A_4 \end{bmatrix}. \tag{6}$$

Then, C can be calculated with

$$C_1 = A_1B_1 + A_2B_3 \tag{7}$$
$$C_2 = A_1B_2 + A_2B_4 \tag{8}$$
$$C_3 = A_3B_1 + A_4B_3 \tag{9}$$
$$C_4 = A_3B_2 + A_4B_4 \tag{10}$$

- $T(n) = 8T(n/2) + cn^2; T(1) = c$
- $a = 8, b = 2, d = 2$, case 3 $T(n) = \Theta(n^{\log_2 8}) = \Theta(n^3)$

2. **Idea:** Compute C_1, C_2, C_3, C_4 with only 7 multiplications, not 8.

$$M_1 = (A_2 - A_4)(B_3 + B_4) \tag{11}$$
$$M_2 = (A_1 + A_4)(B_1 + B_4) \tag{12}$$
$$M_3 = (A_1 - A_3)(B_1 + B_2) \tag{13}$$
$$M_4 = (A_1 + A_2)B_4 \tag{14}$$
$$M_5 = A_1(B_2 - B_4) \tag{15}$$
$$M_6 = A_4(B_3 - B_1) \tag{16}$$
$$M_7 = (A_3 + A_4)B_1 \tag{17}$$

With these we can compute C_1, C_2, C_3, C_4 with only additions of the M matrices.

$$C_1 = M_1 + M_2 - M_4 + M_6 \tag{18}$$
$$C_2 = M_4 + M_5 \tag{19}$$
$$C_3 = M_6 + M_7 \tag{20}$$
$$C_4 = M_2 - M_3 + M_5 + M_7 \tag{21}$$

- $T(n) = 7T(n/2) + cn^2; T(1) = c$
- $a = 7, b = 2, d = 2$, case 3 $T(n) = \Theta(n^{\log_2 7}) = \Theta(n^{\log_2 7}) \approx \Theta(n^{2.807})$

- If n is not a power of 2, we zero-pad the matrices to have n as a power of two. This may increase the complexity by at most a factor of 7.

1.3 Median of Unsorted Arrays

- For an unsorted array A, we can find the average, max, min, sum, etc. in linear time.
- The trivial algorithm is to sort A then get the median. That takes $O(n \log n)$ time.
- We will solve a more general problem: Find the k^{th} smallest element in A. (e.g. $A = 5, 2, 6, 7, 4$, $\text{Select}(A, 1) = 2, \text{Select}(A, 4) = 6$)
- if $|A| = 1$, then return $A[1]$. Otherwise find a splitter s in arbitrary element of A. Partition A into A^+ and A^-, then divide
- $T(n) = T(\max(|A^-|, |A^+|)) + cn = T(\max(i - 1, n - i)) + cn.$
- Worst case: $T(n) = T(n - 1) + cn = \Theta(n^2)$

- Best case: $T(n) = T(n/2) + cn = \Theta(n)$. Suppose $b > 1$, by the Master Theorem $T(n) = T(n/b) + cn = \Theta(n)$.

We define s is a good splitter if s is greater than $1/4$ of the elements of A and less than $1/4$ of the elements of A. We can make the following observation:

1. With this splitter, we will reduce the size to at most $3n/4$.
2. Half the elements are good splitters.

We should select splitter s uniformly at random.

- $P(\text{splitter is good}) = \frac{1}{2}$
- $P(\text{splitter is bad}) = \frac{1}{2}$
- We can show that the expected number of trials (splitter selections) until obtaining a good splitter is 2.

1.3.1 Expected Runtime

$$\underbrace{n_0 \to n_1 \to n_2}_{\text{Phase 0, size}\le n} \to \underbrace{n_3 \to n_4}_{\text{Phase 1, size}\le \frac{3}{4}n} \to \underbrace{n_5 \to n_6}_{\text{Phase 2, size}\le \frac{3}{4}^2 n} \to \dots \tag{22}$$

- Phase j: input size $\le (\frac{3}{4})^j n$
- Random variable y_j =# of recursive calls in phase j. Note that $E[y_j] = 2$.
- Random variable x_j =# of steps to all the recursive calls in phase j.
- Total number of steps is $x = x_0 + x_1 + x_2 + \dots$.
- We can compute $E[x] = E[x_0] + E[x_1] + E[x_2] + \dots$.

$$x_j \le c y_j \frac{3}{4}^j n \tag{23}$$

$$E[x_j] \le c E[y_j] \frac{3}{4}^j n \le 2c \frac{3}{4}^j n \tag{24}$$

$$E[x] = \sum_j E[x_j] \le \sum_{j=1}^{\infty} 2c \frac{3}{4}^j n = \frac{2c}{1-\frac{3}{4}} n = 8cn = \Theta(n) \tag{25}$$

1.3.2 Deterministic Algorithm

- If $|A| \leq 5$ then we sort A and return the k^{th} smallest.
- Otherwise, partition A into $n/5$ groups of size 5 each, then find the median of each group (constant time) and store in list M. This takes linear time.
- Select the median of M with the Select algorithm, this is a good splitter.
- the worst case running time is $T(n) = T(\lceil \frac{n}{5} \rceil) + T(\lfloor \frac{3n}{4} \rfloor) + cn$.
- This recursive relation cannot be solved by the Master Theorem. We can prove using induction that $T(n) < 20cn$.

Question: Why groups of 5?

- With groups of 5, the total size of subproblems: $\frac{n}{5} + \frac{3n}{4} = \frac{19n}{20} < n$
- With groups of 3, the total size of subproblems: $\frac{n}{3} + \frac{3n}{4} = \frac{13n}{12} > n$, not sufficient.
- So group size of $5, 7, 9, 11, \ldots$ would also work.

2 Closest Pair of Points

- Problem: Given a set of n points, find the pair of points that are the closest in $O(n \log n)$.

2.1 Closest Pair in 2D

- Divide: points roughly in half by drawing vertical line on midpoint
- Conquer: Find closest pair on each half, recursively.
- Combine: Find the closest pair (p, q), $p \in L$, $q \in R$. However, there may be $\Theta(n^2)$ pairs.

- Claim: Let $p = (x_p, y_p) \in B_L, q = (x_q, y_q) \in B_R$ with $y_p \leq y_q$. If $d(p, q) < \delta$ then there are at most **six** other points (x,y) in B such that $y_p \leq y \leq y_q$.
- Proof:
- $S_L = \{p' = (x, y) : p' \neq p \in B_L \wedge y_p \leq y \leq y_q\}$ (other points on the left of the middle)
- $S_R = \{p' = (x, y) : p' \neq q \in B_R \wedge y_p \leq y \leq y_q\}$ (other points on the right of the middle)

- Assume by contradiction that $|S_L \cup S_R| \geq 7$. WLOG assume $|S_L| \geq 4$.
- In a $\delta \times \delta$ square there are at least $4 + 1 = 5$ points. Divide the square into 4 smaller squares, by Pigeonhole Principle, there is a square with at least 2 points, whose distance is at most $\delta/\sqrt{2}$. This contradicts the assumption that the closest pair on the left is at most δ.

- Then, we can sort everything in the y axis, and check the next seven points by the y coordinate for the minimum distance. This takes linear time.
- We only need to modify the combine step in the algorithm so it's $\Theta(n)$ runtime.

Algorithm 1 Closest Pair in 2D

```
1: procedure ClosestPair(P)
2:     P_x := the list of points in P sorted by x-coordinate
3:     P_y := the list of points in P sorted by y-coordinate
4: procedure RCP(P_x, P_y)
5:     if |P_x| ≤ 3 then return brute force(P_x)
6:     L_x := the first half of P_x; R_x := the second half of P_x
7:     m := (max x-coordinate of L_x + min x-coordinate of R_x)/2
8:     L_y := sublist of P_y with points in L_x
9:     R_y := sublist of P_y with points in R_x
10:    (p_L, q_L) := RCP(L_x, L_y); (p_R, q_R) := RCP(R_x, R_y)
11:    δ := min{d(p_L, q_L), d(p_R, q_R)}
12:    if δ = d(p_L, q_L) then
13:        p := p_L; q := q_L
14:    else
15:        p := p_R; q := q_R
16:    B := sublist of P_y with points in [m − δ, m + δ]
17:    for each p in B do
18:        for each next seven q after p in B do
19:            if d(p, q) < d(p*, q*) then (p*, q*) := (p, q)
```

- So $T(n) = 2T(\frac{n}{2}) + cn$, which is $O(n \log n)$.

3 Greedy Algorithms

- There is an optimization problem: given an input, find a solution that minimize/maximize an objective function f under some constraint.
- Build the solution incrementally in stages
- At each stage, extend the solution greedily and irrevocably.

- For some problems this gives optimal solutions (i.e. MST), but for other problems it does not.
- The order of the stages is very important.

3.1 Interval Scheduling

- Input: n intervals, with interval j starts at time s_j and finishes at time f_j.
- Output: maximum-size set of intervals that do not overlap
- Naive algorithm is to try each subset of n intervals by brute force $O(2^n)$, way too slow.
- Greedy algorithm sorts interval in some order, then if it doesn't overlap then add it to the solution.
- **What order gives the biggest feasible set?**
  1. Increasing start time: s_j
  2. Increasing finish time: e_j
  3. Shortest interval: $e_j - s_j$
  4. Fewest conflicts
- The question is which one is optimal? Option 1 has a clear counterexample with one very long interval that overlaps each other interval. Option 3 is also not optimal using a short interval between two long interval. Option 4 is not optimal (not very clear counterexample).
- Option 2 is optimal. The intuition is that chosing these intervals first will leave the most time for the rest of the intervals.
- To find if the interval is compatible, we just need to check if the start time of the new interval is greater than the finish time of the latest scheduled interval.

3.1.1 Proof of optimality

- Suppose for contradiction that this greedy algorithm is not optimal
- Say greedy selects interval $i_1, \ldots, i_k$ sorted by increasing finish time.
- Suppose the optimal schedule $j_1, \ldots, j_m$ has $m > k$ intervals, and sort by increasing finish time. Consider an optimal schedule that can match the greedy $j_1 = i_1, \ldots, j_r = i_r$ for the greatest possible r.
- By the nature of the greedy algorithm, then i_{r+1} finishes the earliest amongst the compatible intervals remaining. So consider the schedule $S : i_1, \ldots, i_r, \mathbf{i_{r+1}}, j_{r+2}, \ldots, j_m$.
- This is also optimal (contains m intervals) but matches the greedy algorithm by one more position. This is a contradiction.

3.2 Interval Partitioning

- Motivation: given a set of lecture time intervals, schedule them into as few classrooms as possible.
- Input: n intervals, interval j starts at s_j and finishes at f_j.
- Output: group interval into fewest partitions such that intervals in each partition are compatible.
- An idea is to find the maximum set of compatible intervals using the previous algorithm. This doesn't work.
- We can try the same orders as the previous problem, but only earliest start time is optimal in this case.
- To implement it efficiently, we will use a heap with increase-key operation.

3.2.1 Proof of optimality

- We define the **depth** at time t as the number of intervals that contain time t. The **maximum depth** d_{max} is the maximum depth over all times.
- Clearly, the number of partitions needed is at least d_{max}. We will show that this greedy algorithm create only d_{max} partitions.
- Let d be the number of partition the greedy algorithm opened.
- Partition d was created because there was an interval j that overlaps with some previously scheduled interval in each of the $d-1$ other partitions.
- This means that for $d-1$ intervals, their start times are all before s_j and their finish times are all after f_j (otherwise j must be compatible). Hence, the depth at s_j is exactly d.
- Thus, $d_{max} \geq d$ so the greedy algorithm is optimal.
- Warning: this proof rely on the fact that the start time

3.3 Minimum Lateness Scheduling

- n intervals: $1, 2, \ldots, n$, with each interval j requires t_j units of time and has deadline d_j.
- The lateness of an interval is $l_j = \max\{0, f_j - d_j\}$
- Output the schedule that minimizes the maximum lateness $L = \max_j l_j$.
- Fact 1: there is an optimal schedule with no gaps.

- The naive algorithm is to try all possible schedules, which is $O(n!)$.
- For greedy algorithm, we will sort intervals in this order
  1. Shortest processing time, t_j
  2. Smallest slack first $d_j - t_j$
  3. Earliest deadline first d_j
- 1 and 2 are not optimal, and they have simple counterexamples with $n = 2$.
- an **inversion** is two intervals i, j such that $d_i > d_j$ but i is scheduled before j.
- **Note that in general:** Define inversion as a violation of what your order is, then prove that inversion is slightly worse or bad.

3.3.1 Proof of optimality

- Observation 1: The greedy algorithm has no gaps.
- Observation 2: The greedy algorithm has no inversions.
- We will prove
- **Claim 1**: If a schedule S with no gaps has an inversion, then S has a pair of inverted intervals that are adjacent.
- Suppose for contradiction that there is S with an inversion but does not have adjacent intervals that are inverted. Then, $d_j < d_i \leq d_{i+1} \leq d_{i+2} \leq \cdots \leq d_{j-1} \leq d_j$, so $d_j < d_j$ which is a contradiction.
- **Claim 2**: All schedules with no gaps and no inversions have the same lateness
- Let S and S' be two distinct schedules with no gaps and no inversions
- Note S and S' differ only by the schedule of intervals with the same deadline
- Consider the intervals with the same deadline, they must be adjacent in both S and S'. As a group, the maximum lateness of these intervals is the same because the group will finish at the same time.
- **Claim 3**: Swapping adjacent inverted interval does not increase lateness and reduces the total number of inversions by one.
- Let i and j denote two adjacent inverted intervals in the schedule S. By swapping i and j, we get a schedule S' with one fewer inversion.
- Let l and l' be the lateness before/after swap. $\forall k \neq i, j, l_k = l'_k$.
- We know $d_i > d_j$ because it was an inversion. Thus, $l_j \geq l'_j$ and $l_j \geq l'_i$
- $l_j = f_j - d_j \geq f'_i - d_i = l'_i$

- Suppose by contradiction that the greedy schedule S is not optimal.
- Let S^* is the schedule with no gaps and fewest inversion.
- **Case 1:** S^* has no inversion. By claim 2, this is contradiction.
- **Case 2:** S^* has at least one inversion. By claim 1, it must have two inverted intervals that are adjacent. By claim 3, we can swap them to get a schedule with no greater lateness with one fewer inversion so it must be optimal. Then S^* does not have the fewest inversions among all optimal schedules. This is also a contradiction.

3.4 Huffman Code

- Given an alphabet Γ: a set of n symbols
- You need to encode symbols using binary code.
- Fixed length code requires $\lceil \log n \rceil$ bits per symbol.
- This is easy to decode, but this is not optimal
  - "e": 12.7% of letters in english
  - "z": 0.07% of letters in english
  - We should give a shorter code to "e" than "z".
- The goal is to find a code that minimize the length of text coding
- Variable length code can save space, but is harder to decode. Suppose $\Gamma = \{a, b, c\}$ and $a = 1, b = 01, c = 010$. Then 0101 is ambiguous as bb or ca.
- A **prefix code** is a code where no codeword is a prefix of any other. Scan from left to right until you see a codeword.
- Prefix code can be represented as a binary tree (edges are leaf), with the leaves as the symbols.
- The efficiently of a code is the weighted height of this tree, where the weight of a leaf is the probability of that symbol.

3.5 Prefix Code Problem

- Input: A set of Γ with their frequencies $f : \Gamma \to \mathbb{R}$ where $\sum_{x \in \Gamma} f(x) = 1$
- Output: Binary tree T representing the optimal prefix code for Γ, f
- Optimal Solution is a tree with the weighted average height $AD(T)$ of the tree is minimum.
- **Fact 1:** An optimal tree is a full binary tree, where each internal has two children.

- **Fact 2:** In an optimal tree T, $\forall x, y; f(x) < f(y) \implies \text{depth}_T(x) \geq \text{depth}_T(y)$.
- **Fact 3:** If x, y have minimum frequency, then there is an optimal tree T such that x, y are siblings and are at max depth.
- **Huffman's Algorithm**: Combine the two smallest frequencies into a new symbol z with frequency $f(z) = f(x) + f(y)$, and repeat recursively until there are only two symbol left, in which case we assign them 0 and 1.

3.5.1 Proof of Optimality

- We will prove by induction. The base case for $n = 2$ is trivial.
- Induction hypothesis: for all Γ, f for $n - 1$ symbols, the algorithm produces an optimal tree.
- Let Γ, f be an alphabet with frequencies with n symbols. Let H be the tree produced by the algorithm.
- The algorithm constructed by H by: (1) replacing two symbols $x, y \in \Gamma$ of minimum frequency with a new symbol z, and

$$\Gamma' = (\Gamma \setminus \{x, y\}) \cup \{z\} \tag{26}$$

$$f'(\alpha) = \begin{cases} f(x) + f(y), & \alpha = z \\ f(\alpha), & \text{otherwise} \end{cases} \tag{27}$$

- We know that x, y are siblings (with parent z) in H
- Since H' has $n - 1$ symbols, H' is optimal for Γ', f'
- We know $AD(H) = AD(H') + f(x) + f(y)$
- Now take an optimal tree T for (Γ, f) where x, y are siblings. We know T exists by fact 3.
- Let T' be the tree constructed from T by removing x, y and replacing their parent with symbol z (and $f(z) = f(x) + f(y)$)
- Clearly, T' is a prefix code for (Γ', f') with $AD(T) = AD(T') + f(x) + f(y)$
- We know $AD(T') \geq AD(H')$ as H' is optimal for (Γ', f') Thus, $AD(T) - f(x) - f(y) = AD(T') \geq AD(H') = AD(H) - f(x) - f(y) \implies AD(T) \geq AD(H)$. but T is optimal so H is optimal.

3.6 Dijkstra's Single-Source Shortest Path Algorithm

- Input: A weighted directed graph $G = (V, E)$ with weights $w : E \to \mathbb{R}_+$ and a source vertex $s \in V$
- Output: Length of the shortest path from s to every other node $t \in V$
- Suppose R is a subset of nodes of G that includes s. A path $s \to v$ is a **R-path** from $s \to v$ is restricted to contain only in R before reaching v.
- At a high level, the algorithm maintains a set $R \subseteq V$, (at the beginning s, and at the end V)
- The algorithm also contains $d(v)$ where
  1. if $v \in R$, then $d(v)$ is the length of the shortest path from $s \to v$. This is exactly what we want.
  2. if $v \in V \setminus R$, then $d(v)$ is the length of the shortest $s \to v$ R-path.
- The goal of the algorithm is to have R contain all the nodes. In each iteration, the algorithm adds one more node from R and maintains its properties.
- The algorithm will greedily choose the node $u \in V \setminus R$ with the minimum $d(u)$.
- We know that $d(u)$ is the length of the shortest $s \to u$ R-path. We will show that this is also the length of the shortest $s \to u$ path. If the claim is true, we can
  a) move u to R and still maintain property 1.
  b) update $d(v)$ for nodes in $V \setminus R$ to satisfy property 2.
- The proof of the claim is simple, suppose by contradiction that there is a shorter path P with $len(P) < d(u)$. P must cross from R to $V \setminus R$ at least once. Let v be the first node this P reaches in $V \setminus R$, so first section of P, we denote P_1 from $s \to v$ is an R-path. As all the weights are non-negative, $len(P) \geq len(P_1) \geq d(v) \geq d(u)$, so $len(P)$ cannot be less than $d(u)$, which is a contradiction.
- Upon adding u to R, we gain more freedom for R-paths. For points in $V \setminus R$, we can choose to retain the previous R-path or to use u for the new R-path. Thus, for some $v \in V \setminus R$,
  - if the shortest R-path does not contain u, do nothing.
  - if the shortest R-path contains u, then the new $d(v) = d(u) + w(u, v)$.
    - Note that u must be the last node in the shortest R-path. Suppose there is another node $r \in R$ where $s \to u \to r \to v$ is shorter than $s \to u \to v$.
    - Since $r \in R$ when u was still in $V \setminus R$, $d(r)$ must be the length of the shortest path of $R \setminus \{u\}$. So in this case, the shortest path to $v : s \to r \to v$ doesn't require u.
- So for every $v \in V \setminus R$, we update $d(v) \leftarrow \min\{d(v), d(u) + w(u, v)\}$.
- 1: **procedure** Dijkstra

2: $R \leftarrow \{s\}$
3: $d(s) \leftarrow 0$ for each nodes $v \neq s$ do if (s, v) is an edge then $d(v) \leftarrow w(s, v), p(v) \leftarrow s$ else $d(v) \leftarrow \infty, p(v) \leftarrow Nil$
4: **while** $R \neq V$ **do**
5: $u \leftarrow$ node not in R with the minimum $d(u)$
6: $R \leftarrow R \cup \{u\}$
7: **for each** v s.t. (u, v) is an edge **do**
8: **if** $d(v) > d(u) + w(u, v)$ **then**
9: $d(v) \leftarrow d(u) + w(u, v)$
10: $p(v) \leftarrow u$

- **Note** that the algorithm does not work with negative weights.

4 Dynamic Programming

- DP works on problems that have the **optimal substructure property,** i.e. optimal solution can be computed efficiently from the optimal solution of subproblems.
- The method is to break the problem down into subproblems, solve each subproblem only once and **store** the solutions.
- When solving a subproblem, we look up previously computed solutions instead of recomputing it.
- Storing the solution to subproblems is called **memoization.**
- DP is **different** from divide and conquer because:
  - Divide and conquer is a special case of dynamic programming where the subproblems that are solve **never overlap**, so there's no need for memoization to avoid recomputing some previously solved solution.

4.1 DAG Shortest Path (Toy Problem)

- Given some directed acyclic graph, find the shortest path from the root to any leaf.
  1. The greedy algorithm is not optimal (very simple counterexample).
  2. Naive recursion works, but it is very slow.
     - If each node has k children, then $T(0) = c, T(n) = kT(n-1) + c$ then the runtime is $\Theta(k^n)$.
     - Note that in naive recursion, we are solving the same subproblems over and over again. Dynamic programming will aim to solve this problem.

3. Dynamic programming. We can solve the problem by caching the solutions of the subproblems and in a bottom-up order rather than top-down. With

$$\text{COST}(i) \leftarrow \min\{\text{COST}(\text{nextup}(i)) + \text{costup(i)}, \tag{28}$$
$$\text{COST}(\text{nextdown}(i)) + \text{costdown(i)}\}. \tag{29}$$

The cost of this approach is equal to the number of nodes, which is $\Theta(n^k)$.

4.2 Weighted Interval Scheduling

- In normal interval scheduling, every interval is of equal importance. So we want to schedule the maximum number of intervals.
- Now, each interval have a weight (as some jobs are more important than others), so we want to schedule a set of compatible intervals with the maximum total weight.
- Clearly, the greedy algorithm by earliest finish time is not optimal. Sorting by weight is not optimal either. There are obvious counterexamples. Actually, no greedy approaches will work.

4.2.1 Dynamic Programming Approach

- We will sort the intervals by increasing finish time $f_1 \leq f_2 \leq \cdots \leq f_n$.
- The **predecessor** of interval $p[j]$ is the largest index i such that $f_i \leq s_j$. ($p[j] = 0$ if interval j has no predecessors) We can find $p[j]$ in $O(\log j)$ time by doing binary search.
- Let S be the optimal subset of intervals (for intervals $\{1, \ldots, n\}$)
- Consider the last interval n, either
  1. $n \notin S$, then S is optimal for intervals $\{1, \ldots, n-1\}$.
  2. $n \in S$, then $S = \{n\} \cup$ optimal subsets of intervals $\{1, \ldots, p[n]\}$.
- S is the best of case 1 and case 2, (that with the highest total weight).
- For dynamic programming in general, we want to define our notation and subproblems (30) to reach the **Bellman equation** (31).

$$OPT(j) = \text{ max weight for intervals in } \{1, \ldots, j\} \tag{30}$$

$$OPT(j) = \begin{cases} 0 & \text{if } j = 0 \\ \max\{\text{OPT}(j-1), \text{OPT}(p[j]) + w[j]\} & \text{if } j > 0 \end{cases} \tag{31}$$

- We can use a top-down approach with recursion

```
1: procedure TopDown(w, s, f)
2:     Sort the intervals (w, s, f) by increasing finish time f_1 ≤ f_2 ≤ ··· ≤ f_n.
3:     Compute p[j] for j = 1, ..., n. using binary search for each p[j].
```

```
    Let OPT be a global array of size n + 1 where OPT[0] = 0
    return TD-OPT(n)
procedure TD-OPT(j)
    if OPT[j] is not initialized then
        TD-OPT[j] ← max{TD-OPT(j − 1), TD-OPT(p[j]) + w[j]}
    return OPT[j]
```

- However, a bottom up approach is preferred as it is more simpler and shows that you understand dynamic programming better.

```
procedure BottomUp(w, s, f)
    Sort the intervals (w, s, f) by increasing finish time f_1 ≤ f_2 ≤ ··· ≤ f_n.
    Compute p[j] for j = 1, ..., n. using binary search for each p[j].
    Let OPT be an array of size n + 1 where OPT[0] = 0
    for j = 1 to n do
        OPT[j] ← max{OPT[j − 1], OPT[p[j]] + w[j]}
    return OPT[n]
```

- The runtime of both approaches is $\Theta(n \log n)$ due to the sorting and binary search. The space complexity is $\Theta(n)$.

- **Warning**: We need to be careful with our implementation. If we do not store enough information, we will still be solve the same problem multiple times. For example, the following approach does not work.

```
procedure NaiveRec(j)
    if j = 0 then
        return 0
    else
        return max{NaiveRec(j − 1), NaiveRec(p[j])}
```

- Notes on bottom up versus top down:
  - Top down may be preferred when not all sub-solutions need to be computed on some inputs, and one does not need to think about the right order as much.
  - Bottom up may be preferred when all sub-solutions will anyway need to be computed, and it is sometimes faster because it prevents the overhead of recursive calls.

- One way of figuring out what subproblems to define is to think "we already know what the optimal solution is, then what is needed to reach that."

- We can modify the algorithm to return the actual set of intervals that are chosen. There are two ways of doing this.

1. Computing the optimal set **simultaneously** with the optimal weight.

```
procedure BottomUp(w, s, f)
    Sort the intervals (w, s, f) by increasing finish time f_1 ≤ f_2 ≤ ··· ≤ f_n.
    Compute p[j] for j = 1, ..., n. using binary search for each p[j].
    Let OPT be an array of size n + 1 where OPT[0] = 0
    Let S be an array of size n where S[0] ← ∅.
    for j = 1 to n do
```

```
7:        OPT[j] ← max{OPT[j − 1], OPT[p[j]] + w[j]}
8:        if OPT[j] > OPT[j − 1] then
9:            S[j] ← S[p[j]] ∪ {j}
10:       else
11:           S[j] ← S[j − 1]
12:   return S[n]
```

2. Computing the optimal set **after** the optimal weight is computed.

```
1: procedure GetSet(OPT, p)
2:     Let S be an empty set.
3:     i ← n
4:     while i > 0 do
5:         if OPT[i] = OPT[i − 1] then
6:             i ← i − 1                                  ▷ S does not contain i
7:         else
8:             S ← S ∪ {i}                                ▷ S does contain i
9:             i ← p[i]                  ▷ S does not contain p[i] + 1, . . . , i − 1
10:    return S
```

4.3 Edit Distance

- Input of two strings of $X = x_1, \ldots, x_m$ and $Y = y_1, \ldots, y_n$. An "operation" is defined as a deletion of a character, or replacement of a character with another character.
- Some applications are spelling correction, DNA sequencing, plagiarism detection, etc.
- We will solve a generalization of the edit distance problem. We will allow the cost of each operation to be different. So, we will also have as input
  - Cost $d(a)$ of deleting a character a.
  - Cost $r(a, b)$ of replacing a character a with a character b with $r(a, b) = r(b, a)$ and the triangle inequality $r(a, c) \leq r(a, b) + r(b, c)$ holds for any b.

  We would try to find the minimum total cost for matching X and Y.
- Sometimes it is helpful to think of if you have the optimal solution, what is the last operation that was performed? Here, consider the last symbols x_m and y_n. There are three cases:

(Case A) Deleting x_m and optimally match $x_1, \ldots, x_{m-1}$ with $y_1, \ldots, y_n$.

(Case B) Deleting y_n and optimally match $x_1, \ldots, x_m$ with $y_1, \ldots, y_{n-1}$.

(Case C) Match x_m with y_n and optimally match $x_1, \ldots, x_{m-1}$ with $y_1, \ldots, y_{n-1}$.

- Let $E(i, j)$ be the edit distance between $x_1, \ldots, x_i$ and $y_1, \ldots, y_j$. Then, the Bellman

equation is

$$A = E(i-1, j) + d(x_i) \tag{32}$$
$$B = E(i, j-1) + d(y_j) \tag{33}$$
$$C = E(i-1, j-1) + r(x_i, y_j) \tag{34}$$
$$E(i,j) = \begin{cases} \min\{A, B, C\} & \text{if } i, j > 0 \\ \sum_{k=1}^{i} d(x_k) & \text{if } i > 0 \text{ and } j = 0 \\ \sum_{k=1}^{j} d(y_k) & \text{if } j > 0 \text{ and } i = 0 \\ 0 & \text{if } i = j = 0 \end{cases} \tag{35}$$

- **Make sure to remember the base cases!**

```
1: procedure EditDistance(X, Y, d, r)
2:     Let E be an array of size (m + 1) × (n + 1) indexed from 0.
3:     E(0, 0) ← 0
4:     for i = 1 to m do
5:         E(i, 0) ← E(i − 1, 0) + d(x_i)
6:     for j = 1 to n do
7:         E(0, j) ← E(0, j − 1) + d(y_j)
8:     for i = 1 to m do
9:         for j = 1 to n do
10:            A ← E(i − 1, j) + d(x_i)
11:            B ← E(i, j − 1) + d(y_j)
12:            C ← E(i − 1, j − 1) + r(x_i, y_j)
13:            E(i, j) ← min{A, B, C}
14:    return E(m, n)
```

- This algorithm is $O(mn)$. **When you need to compute an element (i.e. $E(m, n)$), make sure its subproblems (i.e. $E(m-1, n-1), E(m, n-1), E(m-1, n)$) are computed!**

- To reconstruct the edits, we can obtain

```
1: procedure RecoverPath(E, X, Y, d, r)
2:     Ops ← ∅
3:     i ← m; j ← n
4:     while i > 0 ∧ j > 0 do
5:         if E(i, j) = d(x_i) + E(i − 1, j) then
6:             Ops ← Ops ∪ {Delete x_i}
7:             i ← i − 1
8:         if E(i, j) = d(y_j) + E(i, j − 1) then
9:             Ops ← Ops ∪ {Delete y_j}
10:            j ← j − 1
11:        if E(i, j) = r(x_i, y_j) + E(i − 1, j − 1) then
12:            Ops ← Ops ∪ {Replace x_i with y_j}
13:            i ← i − 1
14:            j ← j − 1
```

- The recover path procedure is $O(m + n)$.

4.4 0-1 Knapsack

- The knapsack problem is: when given n items, each with a value $v_i > 0$ and weight $w_i > 0$ and the knapsack can hold a weight C. The goal is to find a subset S of items with maximum total value such that the total weight of S is at most C.
- 0-1 knapsack refers to you can either take an item or not take an item, nothing in between.
- Let S be an optimal knapsack content for items $1, \ldots, n$ and capacity C.
- There are two possible cases for the last item n.
  1. If $n \notin S$, then S is optimal for items $1, \ldots, n-1$ and capacity C.
  2. If $n \in S$, then S is optimal for items $1, \ldots, n-1$ and capacity $C - w_n$.
- The subproblems we want to solve now have two parameters n and C, with $K(i, c)$ being the value of the optimal knapsack for $1, \ldots, i$ and capacity c.
- The bellman equation is

$$K(i,c) = \begin{cases} 0 & \text{if } i = 0 \text{ or } c = 0 \\ K(i-1,c) & \text{if } w_i > c \\ \max\{K(i-1,c), K(i-1,c-w_i) + v_i\} & \text{if } w_i \leq c \end{cases} \tag{36}$$

- We need to compute this by 0 to C first, then 0 to n.

```
1:  procedure KNAPSACK(n, C, w, v)
2:      For c = 0 to C do K(0, c) ← 0                          ▷ no items
3:      For i = 0 to n do K(i, 0) ← 0                          ▷ no capacity
4:      for i = 1 to n do
5:          for c = 1 to C do
6:              if c < w_i then                                ▷ item i is too heavy
7:                  K(i, c) ← K(i − 1, c)
8:              else                                           ▷ item i can fit, take it or not
9:                  K(i, c) ← max{K(i − 1, c), K(i − 1, c − w_i) + v_i}
10:     return K(n, C)
```

- To find S, we can backtrack from $K(n, C)$ to $K(0, 0)$.

```
1:  procedure RECOVERKNAPSACK(K, n, C, w, v)
2:      S ← ∅; c ← C; i ← n
3:      while i > 0 and c > 0 do
4:          if K(i, c) = K(i − 1, c) then                      ▷ item i is not in S
5:              i ← i − 1
6:          else
7:              S ← S ∪ {i}
8:              c ← c − w_i
9:              i ← i − 1
10:     return S
```

- The time complexity is $O(nC)$. The space complexity is also $O(nC)$.
- Question: is the running time polynomial in the input size? **No!** because the number C is **exponential in the number of bits** used to represent the input.
- This is **pseudo-polynomial** because it is polynomial in the input values, not input size.
- The problem is NP-hard.

4.4.1 Large Knapsack, Small Items

- Consider that instead of $C, w_1, \ldots, w_n$ being small, we were told $v_1, \ldots, v_n$ are small. Can we solve this problem in $O(nV)$ where $V = \sum v_i$. Yes, with a different dynamic programming algorithm.
- Define our subproblems $K(i, v)$ to be the minimum capacity of a knapsack that can hold a total value of at least v using items $1, \ldots, i$.
- For item i, either we should take i to obtain value v or we should not take i.
  - If we do not choose i, we need $K(i-1, v)$ capacity.
  - If we do choose i, we need $K(i-1, v-v_i)$ capacity.
- The Bellman equation is

$$K(i,v) = \begin{cases} 0 & \text{if } v \leq 0 \\ \infty & \text{if } v > 0, i = 0 \\ \min\{K(i-1,v), K(i-1, v-v_i) + w_i\} & \text{if } v > 0, i > 0 \end{cases} \tag{37}$$

- We should compute this array from 0 to V first, then 0 to n.

4.4.2 Approximation Algorithms

- Unless $P = NP$, we cannot hope to solve knapsack in $O(poly(n, \log C, \log V))$.
- We can find a good approximate solution in polynomial time. $\forall \varepsilon > 0$ we can get a solution that is at least $(1-\varepsilon)$-optimal, in time $O(poly(n, \log C, \log V, 1/\varepsilon))$.

4.5 Chain Matrix Multiplication

- Multiplying three matrices $D = ABC$, then $D = (AB)C = A(BC)$. Although the answers are the same, the cost depends on the multiplication order.
- The problem is to find the optimal order of multiplication of a chain of matrices $A_1, \ldots, A_n$ and each matrix has dimension $d_{i-1} \times d_i$.

- To multiply a $p \times q$ matrix by a $q \times r$ matrix, we obtain a $p \times r$ matrix with pqr multiplications
- We define $m(i, j)$ as the minimum number of multiplication to compute $A_i \cdot A_{i+1} \cdot \cdots \cdot A_j$.
- Try all the ways to put the outer-level parentheses, then compute

$$m(i,j) = \begin{cases} 0 & \text{if } i = j \\ \min_{i \le k < j}\{m(i,k) + m(k+1,j) + d_{i-1}d_k d_j\} & \text{if } i < j \end{cases} \tag{38}$$

- The naive recursive procedure takes $T(n) = \sum_{k=1}^{n-1} T(k) + T(n-k) + c$. We can notice that both terms of the sum occur twice so simplify to

$$T(n) = 2\sum_{k=1}^{n-1} T(k) + (n-1)c \tag{39}$$

$$T(n+1) = 2\sum_{k=1}^{n-1} T(k) + n \tag{40}$$

$$T(n+1) - T(n) = 2T(n) + c \tag{41}$$

$$T(n+1) = 3T(n) + c \tag{42}$$

$$T(n) = \Theta(3^n) \tag{43}$$

- This does not work. We can use dynamic programming to solve this problem in $O(n^3)$. We will compute from the bottom up, starting from chains of length 1 to chains of length n.

```
1: procedure MatrixChainOrder(d_1, ..., d_n)
2:     for i ← 1 to n do
3:         m(i, i) ← 0
4:     for l ← 1 to n − 1 do
5:         for i ← 1 to n − l do
6:             j ← i + l
7:             m(i, j) ← min_{i≤k<j}{m(i, k) + m(k + 1, j) + d_{i−1} d_k d_j}
8:     return m(1, n)
```

4.6 Bellman-Ford Single Source Shortest Paths

- We want to revisit the shortest paths problem. When the edge weights may be negative, Dijkstra's algorithm does not work. We want an algorithm that works with some negative edge weights.
- **Note that if the graph has a negative cycle, the problem makes no sense.**
- **Claim 1:** With no negative cycles, for every node $t \in V$ there is a shortest path that is **simple** (i.e. contains no repeated nodes). Any cycle will have weight ≥ 0, so just remote it and the path will not be any longer.

- Hence, for any node $t \in V$, there is a shortest path with at most $n-1$ edges.
- For $t \neq s$, consider an oracle that tells us the shortest path from s to t that contains i edges $s \to \cdots \to u \to t$. Then the shortest path from $s \to u$ has at most $i-1$ edges.
- We define our subproblems as $OPT(i,t)$ as the length of a shortest $s \to t$ path using **at most** i edges. So, $OPT(i,t) = OPT(i-1,u) + w(u,t)$ for some $u \in V$.
- So,

$$OPT(i,t) = \begin{cases} 0 & \text{if } i = 0 \text{ and } t = s \\ \infty & \text{if } i = 0 \text{ and } t \neq s \\ \min_{u \in V}\{OPT(i-1,u) + w(u,t)\} & \text{if } i > 0 \end{cases} \tag{44}$$

  Note that part of the $\min$ is the case where $u = t$. In that case $w(t,t) = 0$ so this case would be $OPT(i-1,t)$.
- Then, $OPT(n-1,t)$ is the length of the shortest path from s to t. The naive implementation is $O(n^3)$.

```
1: OPT(0, s) ← 0
2: for all t ≠ s, OPT(0, t) ← ∞
3: for i = 1 to n − 1 do
4:     for all t ∈ V do
5:         OPT(i, t) ← min_{u∈V}{OPT(i − 1, u) + w(u, t)}
6: return OPT(n − 1, t)
```

- We can improve this by not checking all the nodes, but only nodes that have an edge going **into** t as well as t to itself. This is like reversing an adjacency list. Hence, instead of minimizing over all $u \in V$, we only minimize over $u \in \{v \in V | w(u,t) < \infty\}$. This way the runtime is $O(nm)$.
- We do not have to store the matrix, which is $O(n^2)$ space. As each row only depends on the previous row, we can just store the two rows reducing the space complexity to $O(n)$.
- To recover the path, we keep track of the current predecessor. Each time our path decreases, we update the predecessor.

4.6.1 Detecting Negative Cycles

- **Claim 2:**

$$\Big(\forall t \in V,\, OPT(k,t) = OPT(k-1,t)\Big) \implies \Big(\forall t \in V,\, OPT(k,t) = OPT(k+1,t)\Big). \tag{45}$$

- Using the bellman equation,

$$\begin{aligned} OPT(k+1,t) &= \min_{u \in V}\{OPT(k,u) + w(u,t)\} && (46)\\ &= \min_{u \in V}\{OPT(k-1,u) + w(u,t)\} && (47)\\ &= OPT(k,t). && (48) \end{aligned}$$

- Even if we don't care about negative cycles, this may help us stop early when we go through one iteration with no improvement.
- If G has no negative cycle reachable from s, **iff** $\forall t \in V$, $OPT(n,t) = OPT(n-1,t)$.
- Proof [$\Longrightarrow$] : If there are no negative cycles reachable from s, there exists a shortest path from s to t with at most $n-1$ edges. Hence, $OPT(n,t) = OPT(n-1,t)$.
- Proof [$\Longleftarrow$] : Suppose $\forall t \in V$, $OPT(n,t) = OPT(n-1,t)$. Then for any $n' \geq n, OPT(n',t) = OPT(n-1,t)$. Suppose for contradiction that there is a negative cycle reachable from s. Then, we can find a path from $s \to t$ for some t in the negative cycle. We will repeat the negative cycle enough time so that $OPT(n',t) < OPT(n-1,t)$. This is a contradiction.
- Hence, to detect negative cycles we run Bellman Ford one extra time and check if $OPT(n,t) = OPT(n-1,t)$. If there is some $t \in V$ such that $OPT(n,t) < OPT(n-1,t)$, then there is a negative cycle reachable from s.
- **To detect all negative cycles**, we add a new node s to the graph and connect it to all other nodes with weight 0. Then, we run Bellman Ford on this new graph. If there is a negative cycle it is reachable from s.
- **Claim 4:** Suppose that $OPT(n,t) \neq OPT(n-1,t)$ for some node t. Then let P be an $s \to t$ path of length $OPT(n,t)$ and with at most n edges then
  1. P contains a cycle.
  2. Every cycle of P has a negative length.
- 1 is trivial, as the path has n edges and $n+1$ nodes. By pigeonhole principle, there at least one node is included twice because G only has n nodes.
- Suppose $P = s \to \cdots \to v \to \cdots \to v \to \cdots \to t$. Suppose by contradiction the path $v \to \cdots \to v$ has non-negative weight. Then, let $P' = s \to \cdots \to v \to \cdots \to t$. The length of P' is at most the length of P but has at most $n-1$ edges. Hence, $OPT(n,t) \geq len(P') \geq OPT(n-1,t)$ which is a contradiction.

4.7 Floyd-Warshall All-Pairs Shortest Paths

- Suppose we are given a graph $G = (V,E)$ with no negative length cycles. We want the shortest path from each node s to each node t.
- If we use Bellman Ford, it will run in $O(n^2 m)$ and if the graph is dense $m = O(n^2)$ then the runtime is $O(n^4)$, which is bad.
- $V = \{1, 2, \cdots, n\}$. We define P as a $i \to^k j$ path if every intermediate node in P is $\leq k$. (i.e. $P = 1 \to 5 \to 2 \to 7$ is a $1 \to^5 7$ path but not a $1 \to^6 7$ path)
- Hence, we will define $OPT(i,j,k)$ as the length of the shortest (simple) $i \to^k j$ path. Consider the following two cases

1. k is not an intermediate node of P. Then, $OPT(i,j,k) = OPT(i,j,k-1)$.
2. k is an intermediate node then $OPT(i,j,k) = OPT(i,k,k-1) + OPT(k,j,k-1)$.

- The bellman equation is

$$OPT(i,j,0) = \begin{cases} 0 & \text{if } i = j \\ w(i,j) & \text{if } i \to j \text{ is an edge} \end{cases} \tag{49}$$

$$OPT(i,j,k) = \min\{OPT(i,j,k-1), OPT(i,k,k-1) + OPT(k,j,k-1)\} \tag{50}$$

```
1: for i = 1 to n do
2:     for j = 1 to n do
3:         if i = j then
4:             OPT(i,j,0) = 0
5:         else
6:             OPT(i,j,0) = w(i,j)
7: for k = 1 to n do
8:     for i = 1 to n do
9:         for j = 1 to n do
10:            if OPT(i,j,k-1) > OPT(i,k,k-1) + OPT(k,j,k-1) then
11:                OPT(i,j,k) = OPT(i,k,k-1) + OPT(k,j,k-1)
12:            else
13:                OPT(i,j,k) = OPT(i,j,k-1)
```

- The runtime complexity is $O(n^3)$. Naively, the space complexity is also $O(n^3)$, but note that row k only depends on row $k-1$, so we can reduce the space complexity to $O(n^2)$.
- To recover the path, each node need to store for every source, what is the previous node in the shortest path.

4.7.1 Transitive Closure Graph

- The **transitive closure** of a graph $G^* = (V, E^*)$, where

$$E^* = \{(u,v) | G \text{ has an edge from } u \to v\}. \tag{51}$$

- To compute the transitive closure of $G(V,E)$, we can use Floyd Warshall after changing all edge weights to i. If $OPT(u,v,n) \neq \infty \iff G$ has a $u \to v$ path $\iff G^*$ has an edge from $u \to v$.
- A better way is to modify the Floyd Warshall (replace $+$ with $\wedge$ and $\min$ with $\vee$)

4.7.2 Detecting Negative Cycles

- **Claim 4:** If G has a negative cycle, then $\exists u \in V, OPT(u,u,n) < 0$

5 Network Flow

- A **flow network** $\mathcal{F} = (G, s, t, c)$ where $G = (V, E)$ is a directed graph and $s \in V$ is the start node and have no incoming edge, $t \in V$ is the terminal node and have no outgoing edge, and $c : E \to \mathbb{R}_+$ is the capacity function.
- A **flow** $f : E \to \mathbb{R}$ that respects the following constraints:
  1. $\forall\, e \in E, 0 \leq f(e) \leq c(e)$
  2. $\forall\, v \in V \setminus \{s, t\}, \sum_{e \in E^+(v)} f(e) - \sum_{e \in E^-(v)} f(e) = 0$
- The **value** of a flow f is $v(f) = f^{out}(s) = f^{in}(t)$.
- The **max-flow problem** is: given a flow network $\mathcal{F}$ to find the flow f with maximum $v(f)$.
- Given a flow f, the **residual graph** is the graph $G_f = (V, E_f)$ where for each edge $(u, v) \in E$, E_f has at most two edges
  - Forward edge: $e = (u, v)$ with residual capacity $c(e) - f(e) > 0$
  - Reverse edge: $e' = (v, u)$ with residual capacity $f(e') > 0$)

  Note that if an edge is saturated, the edge is not included in E_f.
- An **augmenting path** is a path P in G_f from s to t. The **bottleneck** of P is the smallest residual capacity of any edge in P.
- To augment f, we can add the bottleneck x of an augmenting path to f to obtain f' by
  - For each forward edge of P, increase the flow by x.
  - For each reverse edge of P, decrease the flow by x.
- We will first argue that $v(f') > v(f)$. As there is no edge into s, we only have forward edges from s, then $v(f') = f'^{out}(s) = f^{out}(s) + x > v(f)$
- We also need to argue we do not violate the capacity constraints. This is trivial.
- Finally, we must argue that flow conservation is preserved. At each node e, we must consider 4 cases.
  1. If v is not on P, then nothing changes.
  2. The edge entering v and the edge leaving v in the augmenting path P are both forward edges. Then both $f^{out}(v)$ and $f^{in}(v)$ increase by x.
  3. If one is forward and one is reverse, then $f^{in}(v)$ and $f^{out}(v)$ are unchanged.
  4. If both are reverse, then both $f^{in}(v)$ and $f^{out}(v)$ decrease by x.

5.1 Ford-Fulkerson Algorithm

```
1: procedure FordFulkerson(F = (G, s, t, c))
2:     for each edge (u, v) of G do
3:         f(u, v) ← 0
4:     construct G_f
5:     while G_f has an s → t path do
6:         P ← any simple s → t path in G_f
7:         augment(f, P)
8:         update G_f
9:     return f
```

- First, assume edge capacities are integers. Note at every step, everything remains as integers. In every step, the flow increases by at least 1. An upper bound on the max flow is $C = \sum_{e \text{ leaving } s} c(e)$. Assume every node is reachable from s, then an upper bound on the runtime $O((m+n) \cdot C)$. This is pseudo-polynomial time because C can be large.
- To make the algorithm run in polynomial time, we can pick the augmenting path with the maximum bottleneck capacity.
  - Using a modified Dijkstra's algorithm with a Fibonacci heap, we each iteration takes $O(m + n \log n)$ time.
  - We can prove that this takes at most $O(m \log C)$ iterations. Hence, the total running time is $O(m^2 \log C + mn \log n \log C)$.
- We can also pick the augmenting path with the minimum number of edges.
  - Using BFS takes $O(m)$ time per iterations.
  - We can prove at most $O(mn)$ iterations. Hence, the total running time is $O(m^2 n)$.

5.2 Correctness of Ford-Fulkerson

- Given a network flow $\mathcal{F} = (G = (V, E), s, t, c)$, then (A, B) is an *s-t* **cut** of $\mathcal{F}$ if (A, B) is a partition of V with $s \in A$ and $t \in B$.
- The **cut capacity** of (A, B) is $cap(A, B) = \sum_{e:A \to B} c(e)$.
- **Claim:** For any flow f and any s-t cut (A, B), then $v(f) = f^{out}(A) - f^{in}(A)$ where $f^{out}(A) = \sum_{e:A \to B} f(e)$ and $f^{in}(A) = \sum_{e:B \to A} f(e)$. The proof is trivial but tedious.
- **Claim:** For any flow f and any s-t cut (A, B), then $v(f) \leq cap(A, B)$. The proof is $v(f) = f^{out}(A) - f^{in}(A) \leq f^{out}(A) = \sum_{e:A \to B} f(e) \leq \sum_{e:A \to B} c(e) = cap(A, B)$.
- **Claim:** For any flow f and any s-t cut (A, B), if $v(f) = cap(A, B)$ then f is a max flow and (A, B) is a **min cut** (i.e. a cut with minimum cut capacity).

- Proof: Let f^* be a max flow and (A^*, B^*) be a min cut. Let f be any flow and (A, B) be any s-t cut. Then, $v(f) \leq v(f^*)$ and $v(f) \leq v(f^*) \leq cap(A^*, B^*) \leq cap(A, B)$. If $v(f) = cap(A, B)$, then $v(f) = v(f^*) = cap(A^*, B^*) = cap(A, B)$. Thus, f is a max flow and (A, B) is a min cut. This is known as the **max-flow min-cut theorem.**
- Now we are ready to prove that Ford-Fulkerson algorithm is correct. Let f be the flow returned by the algorithm, and G_f be the corresponding residual graph. Let A be the set of nodes that are reachable from s in G_f and let $B = V \setminus A$. Clearly, (A, B) is an s-t cut because $s \in A$ and $t \in B$. Note that
  - Let $u \in A$ and $v \in B$. Suppose for contradiction that $f(u, v) < c(u, v)$ Then, G_f will have an edge from (u, v) so will an path from $s \to v$. However, by definition v is not reachable from s. This is a contradiction.
  - We also know $f(v, u) = 0$. Suppose for contradiction $f(v, u) \neq 0$. Then G_f will have an reverse edge (u, v) so v is reachable from s. However, by definition v is not reachable from s. This is a contradiction.

  By claim 2,

$$\begin{aligned} v(f) &= f^{out}(A) - f^{in}(A) && (52)\\ &= \sum_{u \in A, v \in B} f(u, v) - \sum_{u \in B, v \in A} f(v, u) && (53)\\ &= \sum_{u \in A, v \in B} c(u, v) - 0 && (54)\\ &= cap(A, B) && (55) \end{aligned}$$

  By max-flow min-cut, we know f is a max flow. As a bonus, we also know (A, B) is a min cut.
- **Integrality Theorem:** If all the edge capacities are integers, $\exists$ an integral max flow f.
- Ford-Fulkerson algorithm can also be used to find the min cut

```
1: procedure MinCut(F)
2:     f ← Ford-Fulkerson(F)
3:     compute residual graph G_f
4:     A ← {all nodes that are reachable from s in G_f}
5:     B ← V \ A
6:     return (A, B)
```

5.3 Bipartite Matching

- $G = (V, E)$ is an undirected graph. A **matching** M in G is a subset of edges of G such that no two edges share the same node.
- A **maximal matching** is not a subset of another matching.
- A **maximum matching** is a maximal matching that contains the greatest number of nodes.

A **bipartite graph** is an undirected graph such that $G = (V, E)$ such that V is partitioned into X and Y, and every $e \in E$ connects a node in X to a node in Y. It is also sometimes written as $G = [(X, Y), E]$. Reminder: an undirected graph G is bipartite $\iff$ G has no odd cycle. Can determine if G is bipartite in $O(m + n)$ using BFS or DFS.

- Our problem is given a bipartite graph $G = (V, E)$ output the maximum matching of G.
- We can reduce this to the max-flow problem by turning $G = [(X, Y), E]$ into $\mathcal{F} = (G', s, t, c)$ by
  1. Directing all edges of G from X to Y.
  2. Connect s to each $x \in X$ and connect each $y \in Y$ to t.
  3. Assign a capacity of 1 to each edge.

```
1: procedure BipartiteMatching(G)
2:     construct flow network F from G
3:     f ← MaxFlow(F)
4:     M_f ← {(x, y)|x ∈ X, y ∈ Y, f(x, y) = 1}
5:     return M
```

- By the integrality theorem, f is 0 or 1 on each edge.
- The running time of the algorithm is $O(mC)$. Here, $C = |X| = O(n)$, so it's $O(mn)$.
- Now, we will show that M is a matching and it is optimal.
- **Claim:** $\exists$ matching of size k in G $\iff$ $\exists$ integral flow of value k in $\mathcal{F}$.
- $[\implies]$ Given a matching M of size k, construct $f(s, x) = f(x, y) = f(y, t) = 1$ if $x \in X$ is matched with $y \in Y$. As each x is only matched with at most one y and vise versa, $f(s, x) \leq 1$ and $f(y, t) \leq 1$. Hence, this flow satisfies a capacity constraint.
- To show the flow conservation rules are satisfied, we can enumerate all six cases where it could fail and show they are impossible.
- $[\impliedby]$ Given an integral flow f in $\mathcal{F}$ of value k, we can construct a matching M_f in G of size k by
  - For each edge (x, y) s.t. $f(x, y) = 1$, match x with y.
  - If this is a matching, there is a flow of k from s, and by flow conservation k edges from X to Y must have flow 1, so the size of the matching is k.
  - By contradiction suppose this is not a matching. Then either $x_1, x_2 \in X$ connects to $y \in Y$ then by flow conservation $f(y, t) \geq 2$ which violates capacity constraint. Or, $x \in X$ connects to $y_1, y_2 \in Y$. Then $f(s, x) \geq 2$ which also violate the capacity constraint.
- This shows that M is an optimal matching because we found the max flow of $\mathcal{F}$.

5.4 Min Vertex Cover

- Given an undirected graph $G = (V, E)$. A **vertex cover** $V' \subseteq V$ is a subset s.t. every edge has at least one endpoint in V'. A minimum vertex cover is a vertex cover of minimum size.
- In a general graph, finding a min vertex cover is NP-Complete, but we can have a polynomial time algorithm for bipartite graphs.
- **Claim:** for any matching M in G and any vertex cover C of G, then $|M| \leq |C|$. The proofs follow from every edge in M is covered by at least one node in C and every node in C is covered by at least one edge in M.
- **Claim:** For any matching M in G and any vertex cover C of G, if M is a maximum matching, then C is a minimum vertex cover. This follows straight from the inequality of the previous claim.
- **Claim:** If $G = [(X, Y), E]$ is a bipartite graph, then the size of the maximum matching is equal to the size of the minimum vertex cover. **This is not true for general graphs.** Consider a complete graph K_3 with three nodes A, B, C. The maximum matching is size 1 and the minimum vertex cover is size 2.
  - Given a bipartite graph G construct a flow network $\mathcal{F}$ as before. Let f be the integral max flow in $\mathcal{F}$ from the Ford-Fulkerson algorithm, and let G_f be the residual graph.
  - Let S be the nodes reachable from s in G_f, and $T = V \setminus S$. (S, T) is a min-cut with capacity of $cap(S, T) = v(f)$.
  - Consider the type of edges in F from S to T :
    1. $s \to X \cap T$, there are $|X \cap T|$ edges.
    2. $Y \cap S \to t$, there are $|Y \cap S|$ edges.
    3. ~~$X \cap S \to Y \cap T$~~, there are 0 edges (we will show this later)
  - $C = (X \cap T) \cup (Y \cap S)$ is a vertex cover of G. We know G has 4 types of edges
    1. $X \cap S \to Y \cap S$ is covered by nodes in $Y \cap S$.
    2. $X \cap T \to Y \cap T$ is covered by nodes in $X \cap T$.
    3. $X \cap T \to Y \cap S$ is covered by nodes in $X \cap T$ (and $Y \cap S$).
    4. $X \cap S \to Y \cap T$ contains no edges.
  - Let M be the matching corresponding to the flow f

    $$|C| = |X \cap T| + |Y \cap S| = |X \cap T| + |Y \cap S| = cap(S, T) = v(f) = |M| \quad (56)$$

    So $|C| = |M|$ for some matching M of G so C is a min vertex cover.
  - Now we will show there are no edges from $X \cap S$ to $Y \cap T$. By contradiction let $x \in X \cap S$ and $y \in Y \cap T$ where G has edge from $x \to y$ In the residual graph, there is no edge from $x \to y$. So, $f(x, y) = 1$.
  - As $x \in S$, there is a path from $s \to x$ in G_f. If the path is simple, then $f(s, x) = 0$ so flow conservation is violated.

– With reverse edges in G_f, a path $P : s \to x_1 \to y_1 \to \cdots \to x_k \to y_k \to x$ is possible where $y_1, \ldots, y_k \in Y \cap S$. Because G_f has a reverse edge $y_k \to x$, we know $f(x, y_k) = 1$. Also, $f(x, y) = 1$ so the flow leaving x is at least 2, which violates flow conservation.

```
1: procedure BipartiteVC(G = [(X, Y), E])
2:     construct flow network F from G
3:     f ← MaxFlow(F)
4:     S ← nodes reachable from s in G_f
5:     T ← V \ S
6:     VC ← (X ∩ T) ∪ (Y ∩ S)
7:     return VC
```

Running time is the same as max matching, $O(mn)$.

5.5 Hall's Theorem

- A **perfect matching** in a bipartite graph $G = ([X, Y], E)$ with $|X| = |Y|$ is a matching M s.t. M covers all nodes in G.
- Hall's theorem states that a bipartite graph $G = ([X, Y], E)$ has a perfect matching iff $\exists X' \subseteq X$ s.t. $|X'| > |N(X')|$.
- [$\Longleftarrow$] is straightforward.
- [$\Longrightarrow$] Assume no perfect matching for G exists. So, any matching M will have $|M| < n/2$. Let R be a min vertex cover of G so $|R| < n/2$. Let $X' = X \setminus R$ then $|X'| > n/2$. Then, $|X'| > n/2 > |R| > |Y \cap R|$. There is no neighbor of X' in $Y \setminus R$ otherwise we can match them. Thus, $|N(X')| \leq |Y \cap R| < |X'|$.
- The contrapositive is also useful.

5.6 Edge Disjoint Sets

6 Linear Programming

- Linear programming aims to minimize/maximize an objective function. The objective function is a linear combination of variables, subject to equality or inequality constraints (also linear combinations of variables).
- Many optimization problems can be formulated as linear programs. The main idea for us is how to recognize and formulate problems as linear programming problems.

6.1 Examples

Diet Problem:

- We have n types of food: $F_1, \ldots, F_n$, m types of nutrients: $N_1, \ldots, N_m$, and each unit of food F_j for $j \in \{1, \ldots, n\}$ costs c_j and provides a_{ij} units of nutrient N_i.
- We must obtain at least b_i units of nutrient N_i for $i \in \{1, \ldots, m\}$.
- Our goal is to minimize the cost of the food we buy.
- To formulate this problem as a linear programming problem, we select our
  - **variables** as $x_1, \ldots, x_n$ where x_j is the amount of food F_j in the diet.
  - **objective function** as $c_1x_1 + \cdots + c_nx_n$ to be minimized.
  - **constraints** as $a_{11}x_1 + \cdots + a_{1n}x_n \geq b_1, \cdots, a_{m1}x_1 + \cdots + a_{mn}x_n \geq b_m$ **and** $x_1, \ldots, x_n \geq 0$.

Profit Maximization Problem:

- We can produce n types of products $P_1, \ldots, P_n$ and m types of resources $R_1, \ldots, R_m$ where each unit of product P_j requires a_{ij} units of resource R_i and produces c_j profit.
- You only have b_i units of resource R_i for $i \in \{1, \ldots, m\}$. We want to maximize the profit.
- To formulate this problem as a linear programming problem, we select our
  - **variables** as $x_1, \ldots, x_n$ where x_j is the amount of product P_j produced.
  - **objective function** as $c_1x_1 + \cdots + c_nx_n$ to be maximized.
  - **constraints** as $a_{11}x_1 + \cdots + a_{1n}x_n \leq b_1, \cdots, a_{m1}x_1 + \cdots + a_{mn}x_n \leq b_m$ and $x_1, \ldots, x_n \geq 0$.

Transportation Problem:

- We have factories $F_1, \ldots, F_k$ that manufacture the same product, and outlets $O_1, \ldots, O_l$ that sell the product. Each factory F_i has a supply s_i and each outlet O_j has a demand at least d_j. The cost of shipping one unit of product from factory F_i to outlet O_j is c_{ij}.
- We want to minimize the total cost of shipping.
- To formulate this problem as a linear programming problem, we select our
  - **variables** as x_{ij} where x_{ij} is the amount of product shipped from factory F_i to outlet O_j.
  - **objective function** as $\sum_i \sum_j c_{ij}x_{ij}$ to be minimized.
  - **constraints** are $\sum_j x_{ij} = s_i$ for $i \in \{1, \ldots, k\}$, $\sum_i x_{ij} = d_j$ for $j \in \{1, \ldots, l\}$, and $x_{ij} \geq 0$ for all i, j.

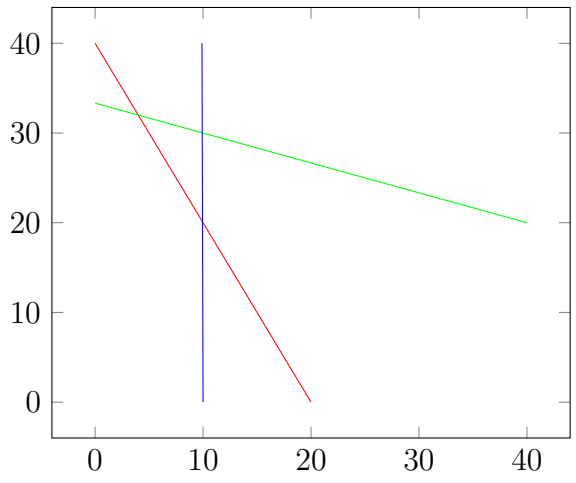
6.2 General Form

- **Variables:** $x_1, \dots, x_n \in \mathbb{R}$ or $\mathbf{x} = \begin{pmatrix} x_1 \\ \vdots \\ x_n \end{pmatrix} \in \mathbb{R}^n$
- **Objective Function:** $c_1 x_1 + \cdots + c_n x_n$ or $\mathbf{c}^T \mathbf{x}$ where $\mathbf{c} = \begin{pmatrix} c_1 & \cdots & c_n \end{pmatrix} \in \mathbb{R}^n$.
- **Constraints:** $\sum_{j=1}^{n} a_{ij} x_j \geq b_i$ or $\mathbf{Ax} \leq \mathbf{b}$ where $\mathbf{A} = \begin{pmatrix} a_{11} & \cdots & a_{1n} \\ \vdots & \ddots & \vdots \\ a_{m1} & \cdots & a_{mn} \end{pmatrix} \in \mathbb{R}^{m \times n}$ and $\mathbf{b} = \begin{pmatrix} b_1 \\ \vdots \\ b_m \end{pmatrix} \in \mathbb{R}^m$. where is elementwise. And, also, $\mathbf{x} \geq 0$.

To visualize this problem, we consider the following example. We have a very big field and we can plant two kinds of crops C_1 for human consumption and C_2 for animal feed.

- C_1 requires 2 hours of labor, 1 kg of seed and 1 bag of pesticide per hectare, and produces 3 dollar per hectare.
- C_2 requires 1 hour of labor, 3 kg of seed, and 0 bags of pesticide per hectare, and produces 2 dollar per hectare.
- We have at most 40 hours of labor, 100 kg of seed, and 10 bags of pesticide.

To formulate this as a linear programming problem, we have two variables x_1, x_2 and we our objective function is $\max 3x_1 + 2x_2$ and our constraints $\begin{cases} 2x_1 + x_2 \leq 40 \\ x_1 + 3x_2 \leq 100 \\ x_1 \leq 10 \\ x_1, x_2 \geq 0 \end{cases}$. We can visualize this in an x_1-x_2 plane as shown below.



We will be raising b for the line $x_2 = -\frac{3}{2}x_1 + b$, until we reach the optimal profit. Note that it is a fact that the **feasible region** of solution in a convex polygon.

It is also a fact that **if** an optimal solution to a linear programming problem exists, then at least one optimal solution is a **vertex** of the feasible region.

Now consider if the farmer can plant three crops. Then, each constraint is a plane, and the feasible region will be a convex 3D polygon. For n crops, there will be an exponential number of vertices.

The simplex method looks at a vertex and all its neighbors. Once we find a vertex that is a local optimum, it will be a global optimum (because of convexity of the feasible region). In the worst case, the simplex method is exponential in the number of variables, but in practice it is very fast. There are now algorithms that can solve linear programming problems in polynomial time (not easy).

Optimal Solutions may not exist when:

1. The problem is overconstrained (feasible region is empty i.e. $x < 0 \wedge x > 0$).
2. The problem is underconstrained (unbounded i.e. $x_1 + x_2$ for $x_1, x_2 \geq 0$).

6.3 Integer Linear Programming

- In **integer linear programming**, the variables $x_1, \ldots, x_n \in \mathbb{Z}$.
- In 0-1 **LP**, the variables $x_1, \ldots, x_n \in \{0, 1\}$. Note that two additional constraints on top of integer linear programming ($x_u \leq 1 \wedge x_u \geq 0$) will gives us 0-1 LP.
- The decision variants of ILP and 0-1 LP are NP-hard. They are much harder than ordinary LP.
- Consider the Min Vertex Cover problem for $G = (V, E)$. It can be formulated as a 0-1 LP problem with
  - Variables: $x_u \in \{0, 1\}, \forall u \in V$
  - Objective function: $\sum_{u \in V} x_u$.
  - Constraints: $x_u + x_v \geq 1$ for all $(u, v) \in E$.

7 Complexity and NP-Completeness

- We consider an algorithm to be efficient if it runs in polynomial time in the size of the input.

- What if you must solve a problems that there are no algorithms that can solve it in polynomial time?
- There is a large set of well-known problems such that
  - Many people have tried to solve efficiently but failed.
  - If you can solve any one of these problems in polynomial time, you can solve all of them in polynomial time.
- This set of problems is called **NP Complete**. Our goal is to determine if a problem is NP complete, and to prove problems are NP complete.
- The key tool for proving NP completeness is **reduction**. A reduction is a way to transform one problem into another. If we can show that one problem is NP complete by reducing it to another problem, then we have shown that the other problem is NP complete.
- This is useful because if you are able to prove that a problem is NP complete, you may stop trying to solve it efficiently and look for alternatives (like approximation algorithms).
- To illustrate the key concepts, definition, and techniques we will use two **decision problems**.

- Consider the **clique** problem. Given a graph $G = (V, E)$, and an integer k, is there a subset $S \subseteq V$ such that S is a clique of size k.
- The Brute force algorithm is to generate every subset $S \in \mathcal{P}(V)$ and check if it is a clique. This is exponential in the size of the input as there are $2^{|V|}$ subsets.

- The **Conjunctive Normal Form** (CNF) is a boolean formula consisting of
  - Boolean variables $x_1, \ldots, x_n$.
  - Their negations $\bar{x}_1, \ldots, \bar{x}_n$.
  - A literal is a variable of its negation.
  - A clause $C = l_1 \vee l_2 \vee \cdots \vee l_r$ is a disjunction of literals.
  - A CNF formula is a conjunction of clauses $\varphi = C_1 \wedge C_2 \wedge \cdots \wedge C_m$.
- A k-CNF formula is a CNF formula with exactly k literals in each clause.
- A CNF formula is satisfiable if there is a truth assignment (T/F) to each variable x_i under which φ evaluates to true. izz.e. under which each clause of φ has at least one literal that evaluates to true.
- The 3SAT problem asks if an input 3CNF formula φ is satisfiable.

- So SAT is polynomially reducible to clique if we can transform any given CNF formula φ in polynomial time into a graph and an integer (G, k) such that φ is satisfiable **if and only if** G has a clique of size k.

- If we can show this, then that means if we can solve clique in polynomial time, then we can solve SAT in polynomial time.
- The contrapositive is if we cannot solve SAT in polynomial time, then we cannot solve clique in polynomial time.
- For the reduction, we can choose k to be the number of clauses. Then, we can construct a graph with one node for every literal of every clause of φ. With no edge between nodes in the same clause. An edge between two every pair of nodes representing literals that are compatible. We claim that φ is satisfiable iff G has a clique of size k.
- Suppose φ is satisfiable. Then, each clause has a literal that is true. Every pair must be compatible, meaning that they are not both true or both false. So, every pair of literal represents an edge in the graph, so that is a clique of size k in G.
- Suppose G has a clique of size k. Then, each of the k nodes belong to different clauses. Set these k nodes to be true, then at least one literal in each clause is true. So, φ is satisfiable.
- The hamiltonian cycle problem, partition problem are all decision problems.
- An **optimization problem** is stronger than a decision problem. However, it is usually not too much more difficult.
  - For clique, an optimization problem is to find a clique of maximum size.
  - A harder optimization problem is to find the clique of maximum size that is a subgraph of a given graph.
  - We claim that if we can solve the decision problem of clique in polynomial time, then we can solve the optimization problems of clique in polynomial time.

7.1 Polynomial Time Reductions

Definition–(Karp): A is **polynomially reducible** to B denoted $A \leq_p B$ if we can transform any instance of x of A into an instance of y of B in polynomial time such that x is a yes instance of A iff y is a yes instance of B.

Definition–(Cook): $A \leq_p B$ if, given an oracle for solving B we can solve A in polynomial time by using the oracle polynomially many times.

Theorem–: If $A \leq_p B$ and B can be solved in polynomial time, then A can be solved in polynomial time.

Proof. To solve any instance x of A, we can transform x to y with the same answer than solve instance y of B and return the yes or no answer. The transformation takes polynomial time, and the solution of B takes polynomial time, so the solution of A takes polynomial time. □

- Intuitively: A decision problem has an efficient verifier if every yes instance x of P has a short "justification" that can be used to efficiently verify that x is indeed a yes instance of P.
- Consider the SAT problem. If someone claims that a given CNF formula φ is satisfiable, then they provide a truth assignment to each variable. This can be verified in polynomial time.
- Consider the clique problem with and someone claims that a given graph has a clique of size k. Then, if they provide a set of k nodes, then we can verify that this is a clique in polynomial time.

7.2 Definitions

Definition–Efficient Verifier: A decision problem P has an **efficient verifier** if there is a polynomial-time verifier algorithm $V(*,*)$ such that

1. $\forall$ yes instances x of P, $\exists$ a justification y of polynomial size in the size of x, such that $V(x,y)$ returns yes.
2. $\forall$ no instances x of P, running the verifier $V(x,y)$ returns no $\forall y$.

Note that justification is also called "witness", "certificate", or "advice".

- Note that any decision problem that can be solved in polynomial time has an efficient verifier (the trivial verifier, that requires no justification).
- For example s-t connectivity of an undirected graph $G = (V, E)$. A polynomial time verifier discards the justification and runs BFS starting from s. If the search reaching t then we return yes, otherwise return no.
- Note that some decision problems **may not** have an efficient verifier.
  - Consider the complement of Clique (we call "Clique-Freedom") problem. Given a graph $G = (V, E)$ and an integer k, does G not have a clique of size k? It is non-trivial to design an efficient verifier. We actually don't know if one exists.
  - Consider the complement of SAT (UNSAT) i.e. given a CNF formula φ, does φ not have a truth assignment under which it evaluates to true? Or equivalently tautology (does all truth assignments satisfy φ). We don't know if an efficient verifier exists.

Definition–P,NP,NP-Complete: We will use "efficient/efficiently" to describe polynomial in the input size.

- P is the set of decision problems that can be solved efficiently.

- NP is the set of decision problems that have an efficient verifier. We know $\mathrm{P} \subseteq \mathrm{NP}$.
- $\mathrm{NP-hard}$ is set of problems such that if a problem $P \in \mathrm{NP-hard}$ then $\forall P' \in \mathrm{NP},\ P' \leq_p P$. Intuitively, P is at least as hard to solve as any problem in NP because if you can solve P in polynomial time then you can solve **every** problem in NP in polynomial time.
- $\mathrm{NP-Complete}$ is the intersection of NP and $\mathrm{NP-hard}$. It is not obvious that $\mathrm{NP-Complete}$ is a non-empty.

- The Cook-Levin Theorem states that SAT is $\mathrm{NP-Complete}$. As a corollary, we know that if SAT$\notin \mathrm{P}$ then $\mathrm{P} \neq \mathrm{NP}$. Otherwise, $\mathrm{P} = \mathrm{NP}$.
- Since there are thousands of well-known NP-complete problems that no one has been able to solve in polynomial time, most people believe that SAT$\notin \mathrm{P}$.
- As we show that SAT is reducible to clique, by Cook-Levin we know that $\forall P \in \mathrm{NP},\ P \leq_pSAT\leq_p$Clique. We also show there is an efficient verifier for clique, so clique is NP-complete.
- Note that we proved that clique is NP-hard by reducing a known NP-hard problem to it. This is much easier than proving it from scratch.
- If you suspect a problem P cannot be solved efficiently, then
  1. See if P is one of the known NP-complete problems.
  2. Try to prove P is NP-hard by finding one of the known NP-complete problem P' and show that $P' \leq_p P$.
- We will do many reductions to both expand our list of known NP-complete problems and help us gain experience with reductions.

7.3 Independent Set

- Given an undirected graph $G = (V, E)$, a set of nodes is independent if there are no edges between them in E. The independent set problem is to find if an independent set of size k.
- We can easily reduce clique to independent set. Given a graph $G = (V, E)$, define its complement $\bar{G} = (V, \bar{E})$ where $\bar{E} = \{(u, v) | (u, v) \notin E\}$. Then, given G and k, then G has a clique of size k iff $\bar{G}$ has an independent set of size k.
- Clearly it is in NP, because we can verify in polynomial time that a given set of nodes that is independent as justification.

7.4 Vertex Cover

- Given an undirected graph $G = (V, E)$, a vertex cover is a set of nodes such that every edge in E is incident to at least one node in the set. The vertex cover problem is to find if a vertex cover of size k.
- We claim that if S is a vertex cover of G then $V \setminus S$ iff independent set of G. Then, we can reduce independent set to vertex cover, since G has an independent set S of size k iff G has a vertex cover $V \setminus S$ of size $n - k$.

7.5 Set Cover

- Given a universe of elements U and a family of subsets $S \subseteq \mathcal{P}(U)$ and an integer k, is there a subset $S' \subseteq S$ such that $|S'| = k$ and $\bigcup_{T \in S'} T = U$.
- We will try to reduce vertex cover to set cover. Given $G = (V, E)$ and k, let $U = E$ and $S = \{S_v | v \in V\}$ where $S_v = \{\text{edges incident on } v\}$. Clearly, G has a vertex cover of size k iff there are k sets in S such that there union is E.

7.6 Integer Linear Programming Feasibility

- Given $b \in \mathbb{Q}^m$ and $A \in \mathbb{Q}^{m \times n}$, does there exists $x \in \{0, 1\}^n$ such that $Ax \leq b$.
- We can reduce 3SAT to ILP feasibility. Given a 3SAT formula φ, with literals $x_1, \ldots, x_n$ and clauses $C_1, \ldots, C_m$. For every boolean variable x_i create an integer variable $0 \leq y_i \leq 1$. Represent boolean literals x_i and $\bar{x}_i$ as y_i and $1 - y_i$ respectively.
- For each clause, we want at least one of the three literals to be true, so we require their sum to be at least 1. So this ILP system has a feasible solution iff φ is satisfiable.

7.7 Graph Coloring

- Given an undirected graph $G = (V, E)$ and an integer k can we color each node of G using k colors such that no two adjacent nodes have the same color.
- We will reduce 3SAT to graph coloring.
- Given a 3CNF formula φ with k variables, we can construct a graph G such that G is $(k + 1)$-colorable iff φ is satisfiable.
- Let V be the set of variables and an additional node F (we will represent false). Connect every pair of nodes. Clearly, this graph is $k + 1$-colorable and not k-colorable.
- We will color the node F with a "dull" color, and the other nodes with "bright" colors.

- We will add additional nodes for $x_1, \bar{x}_1, \ldots, x_n, \bar{x}_n$ and connect $x_i, \bar{x}_i$ because we want to make sure that x_i and $\bar{x}_i$ are not both true. We also connect v_i to v_j and $\bar{v}_j$ for $j \neq i$. This way, either $x_i, \bar{x}_i$ must have one colored dull, and the other colored as the same color as v_i.
- Any $(n+1)$ coloring of G can be converted to a truth assignment of the n literals. To represent these, create a new node c_i for each clause C_i.
- To be satisfiable, we need each clause to evaluate to true. So we connect each c_i to F. Then, we connect c_i to the negation of each literal in C_i to prevent c_i from being true from the false (in or).
- By construction, G is $n+1$-colorable iff φ is satisfiable.
- Note that 3-coloring (when $k = 3$) is also NP-complete, but 2-coloring is not NP-complete.

7.8 Exact Set Cover

- Given a universe of elements U and a family of subsets $S \subseteq \mathcal{P}(U)$, are there disjoint sets in S whose union is U.
- Clearly, this problem is NP as if an alleged exact set cover is given we can verify in polynomial time that it is disjoint and their union is U.
- To prove that it is NP-hard, we will reduce graph coloring to exact set cover.
- Given an instance $G = (V, E)$ and k of the coloring problem, we need to construct an instance U, S of the exact set cover problem such that G can be colored with k colors iff $\exists$ disjoint sets in S whose union is U.
- The basic idea is to transform a color conflict over an edge into a set intersection.
- Let $C = \{n \in \mathbb{Z} \mid 1 \leq n \leq k\}$ be the set of colors, and $U = V \cup (E \times C)$.
- Let S_{vi} denote the set corresponding to coloring vertex v with color i. Then, $S_{vi} =: \{v\} \cup \{(e, i) \in E \times C \mid e \text{ is an edge incident on } v\}$.
- Because each edge is colored with at most 2 colors, but we need to cover every color so we can introduce singletons $T_{ei} = \{(e, i)\}$ for us to cover every color.
- It suffices to show that V is covered by non-intersecting sets, as then we can choose singletons T_{ei} to cover U.
- Suppose G is k-colorable. Then, there is a k-coloring $c : V \to C$. Then, we can select sets $S_{vc(v)}$ for each vertex, so $S' = \bigcup_{v \in V} S_{vc(v)} \supset V$. These sets do not intersect, since each vertex only has one color and adjacent vertices have different colors. We can then choose all singletons T_{ei} where $T_{ei} \cap S' = \emptyset$, and let $T' = \bigcup_{e \in E, i \in C, T_{ei} \cap S' = \emptyset} T_{ei}$. Then, $S' \cup T'$ is a set cover of U, and by construction they don't intersect. Hence, U has an exact cover.

- Suppose U, S has an exact cover $S' \cup T'$ that covers U, where S' are composed of sets S_{vi} and T' are composed of singletons T_{ei}. Since $S' \cup T'$ is an exact cover of U, then for each $v \in V, \exists! S_{vi} \in S'$. Then for that vertex let $c(v) = i$.
- Clearly, there are at most k colors. Suppose for contradiction that there are two vertices u, v where $e = (u, v) \in E$ and $c(u) = c(v) := i$. By construction $S_{ui}, S_{vi} \in S'$. However, $(e, i) \in S_{ui} \cap S_{vi}$. This is a contradiction, so c is a valid k-coloring.