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

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

- Phase j: input size $\leq (\frac{3}{4})^j\, n$
- 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 $|A| \le 5$ 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 $5, 7, 9, 11, \dots$ 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 $|P_x| \leq 3$ 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:    $\delta := \min\{d(p_L, q_L), d(p_R, q_R)\}$
12:    if $\delta = d(p_L, q_L)$ 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 $[m - \delta, m + \delta]$
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: $1, 2, \dots, n$, 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, $\forall x, y; f(x) < f(y) \implies \mathrm{depth}_T(x) \geq \mathrm{depth}_T(y)$.
- **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

$$\begin{aligned} \mathrm{COST}(i) \leftarrow \min\{&\mathrm{COST}(\mathrm{nextup}(i)) + \mathrm{costup(i)}, && (28)\\ &\mathrm{COST}(\mathrm{nextdown}(i)) + \mathrm{costdown(i)}\}. && (29)\end{aligned}$$

  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).

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

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

- 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 $f_1 \leq f_2 \leq \dots \leq f_n$.
3:     Compute $p[j]$ for $j = 1, \dots, n$. using binary search for each $p[j]$.
4:     Let $OPT$ be a global array of size $n+1$ where $OPT[0] = 0$
5:     return TD-OPT($n$)
6: procedure TD-OPT($j$)
7:     if $OPT[j]$ is not initialized then
8:         TD-OPT$[j] \leftarrow \max\{$TD-OPT$(j-1)$, TD-OPT$(p[j]) + w[j]\}$
9:     return $OPT[j]$
```

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

```
1: procedure BottomUp($w, s, f$)
2:     Sort the intervals $(w, s, f)$ by increasing finish time $f_1 \leq f_2 \leq \dots \leq f_n$.
3:     Compute $p[j]$ for $j = 1, \dots, n$. using binary search for each $p[j]$.
4:     Let OPT be an array of size $n+1$ where $\mathrm{OPT}[0] = 0$
5:     for $j = 1$ to $n$ do
6:         $\mathrm{OPT}[j] \leftarrow \max\{\mathrm{OPT}[j-1], \mathrm{OPT}[p[j]] + w[j]\}$
7:     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.

```
1: procedure NaiveRec($j$)
2:     if $j = 0$ then
3:         return 0
4:     else
5:         return $\max\{\mathrm{NaiveRec}(j-1), \mathrm{NaiveRec}(p[j])\}$
```

- 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.

```
1: procedure BottomUp($w, s, f$)
2:     Sort the intervals $(w, s, f)$ by increasing finish time $f_1 \le f_2 \le \cdots \le f_n$.
3:     Compute $p[j]$ for $j = 1, \dots, n$. using binary search for each $p[j]$.
4:     Let $\mathrm{OPT}$ be an array of size $n+1$ where $\mathrm{OPT}[0] = 0$
5:     Let $S$ be an array of size $n$ where $S[0] \leftarrow \emptyset$.
6:     for $j = 1$ to $n$ do
7:         $\mathrm{OPT}[j] \leftarrow \max\{\mathrm{OPT}[j-1], \mathrm{OPT}[p[j]] + w[j]\}$
8:         if $\mathrm{OPT}[j] > OPT[j-1]$ then
9:             $S[j] \leftarrow S[p[j]] \cup \{j\}$
10:        else
11:            $S[j] \leftarrow S[j-1]$
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:     $i \leftarrow n$
4:     while $i > 0$ do
5:         if $OPT[i] = OPT[i-1]$ then
6:             $i \leftarrow i - 1$                          ▷ $S$ does not contain $i$
7:         else
8:             $S \leftarrow S \cup \{i\}$                       ▷ $S$ does contain $i$
9:             $i \leftarrow p[i]$              ▷ $S$ does not contain $p[i]+1, \dots, i-1$
10:    return $S$
```

4.3 Edit Distance

- Input of two strings of $X = x_1, \dots, x_m$ and $Y = y_1, \dots, y_n$. 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 $r(a, c) \le r(a, b) + r(b, c)$ 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 $x_1, \dots, x_{m-1}$ with $y_1, \dots, y_n$.

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

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

- Let $E(i,j)$ be the edit distance between $x_1, \dots, x_i$ and $y_1, \dots, y_j$. 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 $(m+1) \times (n+1)$ indexed from 0.
3:     $E(0,0) \leftarrow 0$
4:     for $i = 1$ to $m$ do
5:         $E(i,0) \leftarrow E(i-1,0) + d(x_i)$
6:     for $j = 1$ to $n$ do
7:         $E(0,j) \leftarrow E(0,j-1) + d(y_j)$
8:     for $i = 1$ to $m$ do
9:         for $j = 1$ to $n$ do
10:            $A \leftarrow E(i-1,j) + d(x_i)$
11:            $B \leftarrow E(i,j-1) + d(y_j)$
12:            $C \leftarrow E(i-1,j-1) + r(x_i, y_j)$
13:            $E(i,j) \leftarrow \min\{A,B,C\}$
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:     $Ops \leftarrow \emptyset$
3:     $i \leftarrow m; j \leftarrow n$
4:     while $i > 0 \wedge j > 0$ do
5:         if $E(i,j) = d(x_i) + E(i-1,j)$ then
6:             $Ops \leftarrow Ops \cup \{$Delete $x_i\}$
```

```
7:            $i \leftarrow i - 1$
8:        if $E(i, j) = d(y_j) + E(i, j - 1)$ then
9:            $Ops \leftarrow Ops \cup \{\text{Delete } y_j\}$
10:           $j \leftarrow j - 1$
11:       if $E(i, j) = r(x_i, y_j) + E(i - 1, j - 1)$ then
12:           $Ops \leftarrow Ops \cup \{\text{Replace } x_i \text{ with } y_j\}$
13:           $i \leftarrow i - 1$
14:           $j \leftarrow j - 1$
```

- 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 $1, \dots, n$ and capacity C.
- There are two possible cases for the last item n.
  1. If $n \notin S$, then S is optimal for items $1, \dots, n - 1$ and capacity C.
  2. If $n \in S$, then S is optimal for items $1, \dots, n - 1$ 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 $1, \dots, i$ 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 $K(0, c) \leftarrow 0$                  ▷ no items
3:     For $i = 0$ to $n$ do $K(i, 0) \leftarrow 0$                  ▷ 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:                 $K(i, c) \leftarrow K(i - 1, c)$
8:             else                              ▷ item $i$ can fit, take it or not
9:                 $K(i, c) \leftarrow \max\{K(i - 1, c), K(i - 1, c - w_i) + v_i\}$
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:     $S \leftarrow \emptyset; c \leftarrow C; i \leftarrow n$
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:             $i \leftarrow i - 1$
6:         else
7:             $S \leftarrow S \cup \{i\}$
8:             $c \leftarrow c - w_i$
9:             $i \leftarrow i - 1$
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 $A_1, \dots, A_n$ 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 $A_i \cdot A_{i+1} \cdot \dots \cdot A_j$.
- 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($d_1, \dots, d_n$)
2:     for $i \leftarrow 1$ to $n$ do
3:         $m(i,i) \leftarrow 0$
4:     for $l \leftarrow 1$ to $n-1$ do
5:         for $i \leftarrow 1$ to $n-l$ do
6:             $j \leftarrow i + l$
7:             $m(i,j) \leftarrow \min_{i \le k < j}\{m(i,k) + m(k+1,j) + d_{i-1}d_k d_j\}$
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: $OPT(0,s) \leftarrow 0$
2: for all $t \neq s$, $OPT(0,t) \leftarrow \infty$
3: for $i = 1$ to $n-1$ do
4:     for all $t \in V$ do
5:         $OPT(i,t) \leftarrow \min_{u \in V}\{OPT(i-1,u) + w(u,t)\}$
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 $[\implies]$: 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 $[\impliedby]$: 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 $i \rightarrow^k j$ path if every intermediate node in P is $\leq k$. (i.e. $P = 1 \rightarrow 5 \rightarrow 2 \rightarrow 7$ is a $1 \rightarrow^5 7$ path but not a $1 \rightarrow^6 7$ path)
- Hence, we will define $OPT(i, j, k)$ as the length of the shortest (simple) $i \rightarrow^k j$ 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

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

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

```
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

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

- 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 $u \rightarrow v$ path $\iff G^*$ has an edge from $u \rightarrow v$.
- 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. $\forall\, e \in E, 0 \le f(e) \le c(e)$
  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($\mathcal{F} = (G, s, t, c)$)
2:     for each edge $(u, v)$ of $G$ do
3:         $f(u, v) \leftarrow 0$
4:     construct $G_f$
5:     while $G_f$ has an $s \to t$ path do
6:         $P \leftarrow$ any simple $s \to t$ path in $G_f$
7:         augment$(f, P)$
8:         update $G_f$
       return $f$
```