CS4820: Introduction to Analysis of Algorithms

Instructor: Anke van Zuylen, Cornell

Notes by Joyce Shen, FA24

1 Stable Matching

The Stable Matching Problem

Input: integer n, two disjoint sets H, R (with $|H| = |R| = n$), and for each $h \in H$, a preference order (**permutation**) of elements in R, and for each $r \in R$, a permutation of H.

Output: a **stable matching** M on H, R

Definitions

M is a **matching** on H, R if $M \subseteq H \times R$ s.t. each $h \in H$ and $r \in R$ is in *at most* one pair. A matching is **perfect** if each $h \in H$ and $r \in R$ is in *exactly* one pair (i.e., $|M| = n$).

A pair (h, r) is an **instability** in perfect matching M if:

- h prefers r to its matching in M AND
- r prefers h to its matching in M

Basically, an instability exists if both parties of the pair prefer each other over their current partners (would have an affair). Note that instabilities cannot be in the match itself – $(h, r) \notin M$.

A matching is **stable** if it is **perfect** and has **no instabilities**.

The questions now are: does there always exist a stable matching? If so, can we find it efficiently? (Spoiler alert: yes)

**Note that an instance may have multiple stable matchings.

Gale-Shapley Algorithm '62

Algorithm 1 Gale-Shapley Algorithm - $O(n^2)$

```
M ← ∅
while there is an unmatched hospital that hasn't yet proposed to every resident do
    h ← unmatched hospital
    r ← first resident in h's preference order that h hasn't proposed to
    h proposes to r
    if r is unmached or prefers h to it's current match then
        add (h, r) to M
        if r was previously matched to h' then
            r rejects h' and remove (h', r) from M
        end if
    else r rejects h
    end if
end while
return M
```

Runtime Analysis

Note we know the Gale-Shapley Algorithm (GSA) always terminates because the `while` loop has at most n^2 iterations. Assuming all other operations are $O(1)$, GSA runs in $O(n^2)$ time.

Proof of Correctness

Theorem 1.1. *The Gale-Shapley Algorithm returns a* ***stable matching***

Proof: We observe:

- if a resident is matched, they stay matched throughout the execution of the algorithm + can only get matched to another hospital they *prefer over* their current matching
- $|H| = |R| = n$

These two facts imply that a hospital cannot be unmatched at the end of the algorithm because they *cannot be rejected more than* $n - 1$ *times*. Thus, the algorithm outputs a **perfect matching** M.

Now, we show that M has no instabilities (i.e., that M is stable): Let $(h, r) \notin M$. WTS: it is *not* true that h prefers r over their current match in M *and* r prefers h to their match in M. Assume h prefers r to their match: then h already proposed to r and was rejected during the execution of the algorithm. Therefore, r is matched to a hospital they prefer over h by the end of the algorithm and (h, r) is not an instability. Therefore, M is a **stable matching**. ∎

Additional Analysis

Let's define h and r to be **valid partners** if there exists a stable matching M^{stable} such that $(h, r) \in M^{\text{stable}}$.

Lemma 1.2. *No hostpital ever gets rejected by a valid partner.*

Proof: Suppose for the sake of a contradiction at some point during the execution of the algorithm (on some input) a hostpital gets rejected by a valid partner. Consider the *first time* a hospital h gets rejected by a valid partner r. At this moment, r must be matched to some h' that they prefer over h. Because h and r are valid partners, there exists an M^{stable} with $(h, r) \in M^{\text{stable}}$. Let r' by the match of h' in M^{stable}. Since r prefers h' over h and M^{stable} has no instabilities, we know h' must prefer r' over r – otherwise, (h', r) would be an instability in M^{stable}. However, if h' is matched to r at this point in the algorithm, then h' must have already proposed r' and *been rejected*. Then, h cannot be the *first* rejection of a valid partnership and we have a contradiction. ■

Corollary 1.2.1. *Note that the GSA outputs a matching that matches each hospital to their most-preferred valid partner: aka **hospital optimal**.*

Lemma 1.3. *If r is the most preferred valid partner for h, then h is the least preferred valid partner for r.*

Proof: Suppose for a contradiction that r is the most preferred valid partner for h and the least preferred valid partner for r is some $h' \neq h$. Let M^{stable} by the stable matching containing (h', r) and let r' be h's match in M^{stable}. Then h prefers r to r' because r is h's most preferred valid partner and r prefers h to h' because h' is r's least preferred valid partner. Thus, (h, r) is an instability in M^{stable} and we have a contradiction. ■

2 Greedy Algorithms

From ChatGPT: A greedy algorithm is a problem-solving approach that builds a solution step by step, **making the locally optimal choice at each step with the hope of finding a global optimum**. In each stage, it chooses the option that seems the best or most promising at that moment, **without revisiting previous decisions**.

Proof strategies for include **Greedy Stays Ahead** or an **Exchange Argument** .

- **Greedy Stays Ahead**: shows the greedy solution is at least as good as the optimal at every step (at every k-th iteration) using an inductive thought process
- **Exchange Argument**: transforms any optimal solution into the greedy solution through local swaps without losing optimality

2.1 Interval Scheduling

Input: n intervals (jobs), each with a start time s_j and finish time f_j

Output: a set $A \subseteq \{1, ..., n\}$ of *nonoverlapping* intervals that is as large (cardinality wise) as possible

Algorithm 2 Earliest Finish Time Algorithm - $O(n \log n)$

```
Sort jobs by finish time f_j                                   O(n log n)
A ← ∅
f ← −∞
for j = 1 to n do                                              O(n)
    if s_j > f then
        Add j to A
        f ← f_j
    end if
end for
return A
```

Where A is the set of selected jobs and f is the finish time of the last job added to A. The runtime of the Earliest Finish Time Algorithm (EFT Algorithm) is $O(n \log n)$.

Proof of Correctness

Exchange Argument: We will prove we can modify an optimal solution O^* into the set A returned by the EFT without losing optimality. Aka, prove $|O^*| = |A|$

Proof: Let O^* be an optimal (max cardinality) set of nonoverlapping jobs, ordered by finish time. Let A be the algorithm's solution. Consider the first job where O^* and A disagree. Let j be the prior job in O^* and A and let j^O and j^A be the next job in O^* and A respectively. Then we can exchange j^O for j^A in O^* while keeping it nonoverlapping because both j^O and j^A start after j and j^A ends *at least as early* as j^O by the algorithm. Thus, swapping them cannot create an overlap in O^*.

Greedy Stays Ahead: We will use the lemma as a 'loop invariant' that captures how after k steps, the greedy solution is ahead of any other solution −¿ prove by induction on k and show the correctness of the algorithm that follows.

Lemma 2.1. *Let A be the jobs given by EFT Algorithm and let O by any other set of nonoverlapping jobs, both ordered by nondecreasing finish time. Then the fnish time of the k-th job in A is at most the finish time of the k-th job of O.*

Proof: Let f_i^O, f_i^A be the finish time of the i-th job in O and A respectively. WTS: $f_k^A \leq f_k^O$ for all $k \geq 1$.

- *Base Case*: by the definition of the EFT, f_1^A is the smallest finish time amongst all jobs
- *Inductive Step*: Suppose the lemma holds for some arbitrary $k \geq 1$. Note that at the start of the $k+1$-th iteration of EFT, R still has all jobs that start after f_k^A. The $k+1$-st job in O, let's call $\hat{j}$, must start after f_k^O. By the IH, $f_k^A \leq f_k^O$ so $\hat{j}$ starts after f_k^A as well. Thus, $\hat{j}$ must be in R at the start of the $k+1$-th iteration so by definition of the EFT, $f_{k+1}^A \leq f_{k+1}^O$. ■

Now we prove the theorem that the EFT algorithm returns a maximum set of nonoverlapping jobs.

Proof: Let A be the set returned by EFT and O by any other set of nonoverlapping jobs. Suppose for a contradiction that $|O| > |A| = k$. Then the $k+1$-th job in O would still be in the set R at the end of the k-th iteration by the Lemma 2.1, so the algorithm would not terminate yet. ■

2.2 The Minimum Spanning Tree Problem (MST)

Input: an undirected graph $G = (V, E)$ with edge costs c_e for each $e \in E$

Output: $T \subseteq E$ such that (V, T) is a spanning tree that minimizes $\sum_{e \in T} c_e$

Note that a **spanning tree** is defined as connected and acyclic. Additionally, $|V| = n$ and $|E| = m$. That'll hold for the rest of this class.

We'll look at three algorithms that are all correct. Kruskal's and Prim's are most important, Reverse Delete is a bit weird.

Algorithm 3 Kruskal's Algorithm - $O(m \log n)$

Sort edges $e \in E$ from cheapest to most expensive
$T \leftarrow \emptyset$
for each edge $e \in E$ **do**
  if e does not create a cycle in T **then**
    Add e to T
  end if
end for
return T

Algorithm 4 Prim's Algorithm - $O(m \log n)$

$T \leftarrow \emptyset$
take an arbitrary $r \in V$ as the root
while (V, T) is not connected **do**
  add the cheapest edge out of r's components to T
end while
return T

Algorithm 5 Reverse Delete - $O(m \log m)$

Sort edges $e \in E$ from most expensive to cheapest
$T \leftarrow E$
for each edge $e \in E$ **do**
  if removing e does not disconnect T **then**
    remove e from T
  end if
end for
return T

Proof of Correctness

To prove correctness, we'll use one lemma:

Lemma 2.2 (The Cut Property)**.** *Let $S \subseteq V$ and let $\delta(S)$ be the edges with one endpoint in S and the other endpoint not in S (i.e., $\delta(S)$ contain all the edges leaving S, aka the* ***cut*** *). Then the cheapest edge in $\delta(S)$ must be in the minimum spanning tree.*

Proof: Let T be an arbitrary spanning tree and suppose that $e \notin T$ where e is the cheapest edge in $\delta(S)$ for some set S. We'll use an exchange argument to show that T cannot be the minimum spanning tree. Let $e = \{v, w\}$ with $v \in S, w \notin S$. T has a path P from v to w. Since P starts in $v \in S$ and ends in $w \notin S$, it must have an edge in $\delta(S)$, let's call f. We claim that $T' = T \cup \{e\} \setminus \{f\}$ (exchanging f by e) produces a cheaper spanning tree than T.

We must show that:

- T' **is acyclic**: The only cycle in $T \cup \{e\}$ is the cycle containing P and e. By removing f, we have an acyclic subgraph.
- T' **is connected**: Any path in T that used $f = \{a, b\}$ exists as a new path in T' by replacing the edge f by following P from a to v, adding e, and then following P from w to b.
- T' **is cheaper than** T: because e is the cheapest edge in $\delta(S)$ and f is also in $\delta(S)$, we know e is cheaper than f.

From these points, we know that T is not a minimum spanning tree. ∎

Now, we can prove the correctness of the three algorithms:

- **Kruskal's Algorithm**: The next edge $e = \{v, w\}$ added by Kruskal's algorithm is the cheapest edge in $\delta(S)$ where $S =$ set of vertices reachable by v in (V, T)
- **Prim's Algorithm**: The next edge e added by Prim is the cheapest edge in $\delta(S)$ where $S =$ the set of vertices reachable from r (the root) in (V, T) (i.e., the solution so far)
- **Reverse Delete**: If the next edge $e = \{v, w\}$ is *not* deleted, then e is the cheapest edge in $\delta(S)$ where $S =$ set of vertices reachable from v in $(V, T \setminus \{e\})$

Runtime Analysis

Kruskal and **Prim's Algorithms** have a naive implementation in $O(mn)$ time, but can be improved to $O(m \log n)$ time using **union-heap** and **heap** data structures respectively. **Reverse Delete** has a naive implementation of $O(m^2)$ and can be improved to $O(m \log m)$, which is worse than the other two.

Additional Analysis - Similar to the Cut Property, we have the **Cycle Property** :

Lemma 2.3 (The Cycle Property). *Let e be the most expensive edge in a cycle C in graph G. Then e is not in the minimum spanning tree.*

Proof: Suppose T is a spanning tree with $e = \{v, w\} \in T$. If we remove e from T, we get two components, one with v and one with w. Since C is a cycle containing $e = \{v, w\}$, C must have some other edge f that connects the v component to the w component. By replacing e with f in T, we get a new spanning tree T' that is cheaper than T, proving that T is not a minimum spanning tree. ∎

Takeaways:

- to show that an edge **must be in** the MST, find a $S \subseteq V$ s.t. e is the cheapest edge in S (cut property)
- to show that an edge **cannot be in** the MST, find a cycle C s.t. e is the most expensive edge in C (cycle property)

2.3 Scheduling - Minimizing Maximum Lateness

Input: n jobs, each with a processing time t_j and deadline d_j

Output: a schedule S that minimizies the maximum lateness for any individual job

Note that a schedule S is a permutation (ordering) in which to process the jobs. We call

- $C_j(S) = (\sum_{i<j} t_i) + t_j$ the completion time of job j in schedule S
- $L_j(S) - \max\{0, C_j(S) - d_j\}$ the lateness of job j in schedule S

Such that the maximum lateness of schedule S is $L_{\max}(S) = \max\{L_j(S)\}$

Algorithm 6 Earliest Deadline First - $O(n \log n)$

Sort jobs by deadline
return S where jobs are ordered by deadline

Proof of Correctness

Proof (by exchange argument): Let A be ordering by deadline. Let O^* be an optimal schedule. Suppose that $A \neq O^*$. We'll use an exchange argument to transform O^* into A without making O^* worse, i.e., ensuring $L_{\max}(O^*)$ stays the same.

Let i, j be consecutive jobs in O^* such that j is scheduled before i in A (called an **inversion**). Exchange i and j in O^* to get O^*_{new}. O^*_{new} is closer to A than O^* was because it has fewer inversions. To analyze $L_{\max}(O{*}_{\max})$, observe that for any job $k \neq i, j$, $C_k(O^*_{\text{new}}) = C_k(O^*)$, meaning their latness doesn't change. For j, $C_j(O{*}_{\text{new}}) \leq C_j(O{*}^{)}$ so its lateness decreases. For i, note that $C_i(O^*_{\text{new}}) = C_j(O^*)$; its lateness increases, but because j was scheduled *before* i in A, $d_j \leq d_i$. Therefore, $L_i(O^*_{\text{new}})$ cannot be more than $L_j(O^*)$. Thus, $L_{\max}(O^*_{\text{new}}) \leq L_{\max}(O^*)$. ■

2.4 Huffman Codes

Given a text over an alphabet Σ, we want to encode the text using 0s and 1s by mapping each character to a bit string.

- **Fixed length encoding**: e.g., $|\Sigma| = 32 \rightarrow \log_2(32) = 5$ bits per character
- **Variable length encoding**: because we know the text (or frequencies of the characters) we can minimize the length of the encoding by assigning shorter strings to frequently used chars and longer strings to infrequently used chars

  Note – with variable length encoding we require the use of a **prefix code to ensure our strings have no ambiguity (no character's string is a prefix of another character's string)

With that in mind...

Input: an alphabet Σ and the frequencies f of the characters in Σ

Output: a prefix code minimizing the length of the encoded text

Observe that each prefix code can be represented by a leaf-labeled binary tree. Additionally, note that

I) the binary tree is **full**; each node has either zero children or two children (never only one)

II) the deepest leaves should correspond to the least frequent characters

III) without loss of generality, two least frequent characters are siblings and we can swap one for the other without changing the length of the encoded text (i.e., the code is the same until the last character)

Algorithm 7 Huffman Codes Algorithm - $O(n \log n)$

---

if $|\Sigma| \leq 2$ **then**
  map the characters to 0 and 1
else
  Let x, y be the least frequent characters.
  Contract x and y to a new character, α and set frequency of α to $f_x + f_y$
  Recurse to find the optimal prefix code for now smaller alphabet instance
  Uncontract α to x and y, with code for x = string$_\alpha$ + '0' and y = string$_\alpha$ + '1'
  Return resulting prefix code
end if

Runtime Analysis:

We have $|n-1|$ recursive calls when $|\Sigma| = n$. Can definitely implement each step in the loop with $O(n)$ time – using a heap, can improve to $O(\log(n))$ per iteration for a total runtime of $O(n \log(n))$.

Proof of Correctness:

Proof: *by induction on* $|\Sigma|$:

- *Base case*: if $|\Sigma| \leq 2$, the algorithm is clearly correct
- *Inductive Step*: By observation III, there exists an optimal prefix code in which x and y only differ in the last bit. By the IH, the recursive step returns an optimal prefix code for the instance with x and y concatenated to α. Setting f_α to $f_x + f_y$ guarantees the uncontraction yields an optimal prefix code for the original instance. ■

  ** Note that this is a very shoddy and condensed proof, textbook goes into much more detail

3 Dynamic Programming

The approach for DPs is to solve a larger problem by using solutions to smaller problems. This informs our definition of the **subproblem** and **recurrence relation** (and **base case(s)**). Once we've established the subproblem, we compute its values from small to large, ensuring the recurrence uses only already-computed values. Once we know $OPT(n)$ – the *value* of the obtimal choices – we need to **backtrace** to find the corresponding solution (what the question is actually asking for).

3.1 Weighted Interval Scheduling

Input: n jobs (intervals), each job with a starting and ending interval $[s_j, f_j]$ and weight or value v_j **

**Assume jobs are sorted by finish time

Output: maximum weight set of non-overlapping jobs **

** No correct greedy algorithm known

The idea is to consider the last job. There are two possibilities:

1. take it $\rightarrow$ the rest of the optimal solution is simply the max weight set of nonoverlapping jobs that finish $< s_n$
2. don't take it $\rightarrow$ the optimal solution is the max weight set of nonoverlapping jobs taken from $1, ..., n-1$.

The optimal solution is the best out of these two possibilities.

Notation:

- $p(j)$ = index of last job that finishes before s_j = $\max\{i : f_i < s_j\}$

Subproblem definition: $OPT(j)$ = weight of max weight set of nonoverlapping jobs taken from jobs $1, ..., j$

- **Recurrence relation**: $OPT(j) = \max\{v_j + OPT(p(j)), OPT(j-1)\}$
- **Base case**: $OPT(0) = 0$

We can find $OPT(n)$ be a recursive algorithm or an iterative one. The recursive approach is shitty exponential runtime. We can solve this by either using **memoization** – storing already computed $OPT(j)$ values and only recursing if we cannot retrieve $OPT(j)$ from memory – or by iterating. Either way, once we find $OPT(n)$, we need to actually get the set of jobs. We can do so by backtracking from $OPT(n)$.

Algorithm 8 Weighted Interval Scheduling - $O(n \log(n))$

$OPT(0) = 0$
for $j = 1$ to n **do**
    $OPT(j) = \max\{v_j + OPT(p(j)), OPT(j-1)\}$
end for
Let $j = n$
Let $A \leftarrow \emptyset$
while $j > 0$ **do**
    if $OPT(j) = v_j + OPT(p(j))$ **then**
        add j to A
        decrease j to $p(j)$
    else
        decrease j to $j - 1$
    end if
end while
return A

3.2 Segmented Least Squares

Input: n points $p_1, p_2, ..., p_n \in \mathbb{R}^2$ ordered by x-coordinate

Output: partition of the points into segments minimizing:
$C\times$number of segments$+$total SSE (sum of squared error) for all line segments (called the **objective value**)
** where a segment is a sequence of consecutive points $p_i, p_{i+1}, ..., p_j$, C is a tunable penalty parameter

Recall the **Ordinary Least Squares (OLS)** problem: find the regression line $f(x) = ax + b$ that minimzes the sum of squared error (SSE): $\sum_{i=1}^{n} (y_i - f(x_i))^2$. Moving towards the segemented least squares problem, there is a tradeoff between the model error (SSE) and the model complexity (number of line segments).

With the follwing DP:

Subproblem definition: $OPT(j)$ = objective value of optimal SLS solution on points $p_1, ..., p_j$

- **Recurrence relation**: $OPT(j) = \min_{i=1,...,j}\{C + e_{ij} + OPT(i-1)\}$
- **Base case**: $OPT(0) = 0$ or $OPT(1) = C$

Algorithm 9 Segmented Least Squares - $O(n^2)$

Precompute all possible line segments we could use (for each $p_i, ...p_j$)
Let e_{ij} = SSE of the OLS (Ordinary Least Squares) through $p_i, p_{i+1}, ..., p_j$

$OPT(0) = O$
for $j = 1$ to n **do**
    $OPT(j) = \min_{i=1,...,j}\{C + e_{ij} + OPT(i-1)\}$
    store an extra parameter $pred(j)$
end for

SLS-partition(j):
if $j = 0$ **then**
    return ()
else
    return (SLS-partition($pred(j) - 1$**),** $[pred(i), j]$**)**
end if

Runtime Analysis:

I) **Preprocessing**: $O(n)$ per segment and we have $\binom{n}{k}$ segments – total $O(n^3)$ runtime (though it is possible with $O(n^2)$ with some thought)

II) **DP**: n iterations, $O(n)$ per iteration for a total of $O(n^2)$

III) **Backtracking**: in $O(n)$ time with the addition of the $pred(j)$ pointer

3.3 The Knapsack Problem

Input: n items, labelled $1, ..., n$ where item j has weight w_j and value v_j and a weight limit W

** w_j, v_j, W all positive integers

Output: a maximum value subset of the items whose combined weight $\leq W$

Step I:

- **Subproblem**: $OPT(j, r)$ value of optimal subset of items $1, ..., j$ if the weight limit is r
- **Recurrence Relation**:

$$OPT(j,r) = \begin{cases} OPT(j-1, r) & \text{if } w_j < r \\ \max\{v_j + OPT(j-1, r-w_j), OPT(j-1, r)\} & \text{if } w_j \leq r \end{cases}$$

- **Base Cases**:
  - $OPT(0, r) = 0$ for $r = 0, ..., W$
  - $OPT(j, 0) = 0$ for $j = 1, ..., n$

We can either compute OPT values recursively with memoization or iteratively. With iteration, our **order of computation** to ensure we only use values we've already computed is $r = 0, ..., W$ for each $j = 0, ..., n$.

Algorithm 10 The Knapsack Problem - $O(nW)$

```
for j = 0 to n do
    for r = 0 to W do
        if j = 0 then
            OPT(j, r) = 0                                        base case
        else
            if w_j > r then
                OPT(j, r) = OPT(j − 1, r)
            else
                OPT(j, r) = max v_j + OPT(j − 1, r − w_j), OPT(j − 1, r)
            end if
        end if
    end for
end for
return OPT(n, W)
```

Runtime Analysis:
To compute $OPT(n, W)$, we have nW iterations, each of which takes $O(1)$ time. $O(nW)$ is not polynomial, but **pseudopolynomial** (polynomial with respect to the number of bits reqiured to represent an input). When an integer as an input can be arbitrarily large, we define *input size = number of bits it takes to represent*:

- W can be represented with $O(\log W)$ bits, meaning $O(W)$ is not polynomial

Step II: now we compute the actual optimal subset ($O(n)$)

```
A ← ∅
j ← n, r ← W
while j > 0 do
    if OPT(j, r) = OPT(j − 1, r) then
        Reduce j by 1
    else
        Add j to A
        Reduce r to r − w_j
        Reduce j by 1
    end if
end while
return A
```

3.4 Shortest Path Problem (Bellman Ford)

Input: directed graph $G = (V, E)$ with edge costs c_e for each $e \in E$, special start and end vertices $s, t \in V$

Output: the shortest (cheapest) path from s to t

If all our edge costs were the same, we could use Breadth First Search (BFS). If all our edge costs were non-negative, we could use Dijkstra's Algorithm. However, if we have negative costs, we need to use **Bellman Ford**. (This is the same Bellman mf who coined *dynamic programming* as a term)

We are going to assume (for now) that G is acyclic. Observe that if G is acyclic, then the cheapest path from $s \to t$ uses at most $n-1$ edges. (If the first edge is (s, v), then the remainder of the path is the cheapest $v \to t$ path and has most $n-2$ edges... etc) We can use the following DP:

Subproblem Definition: $OPT(v, k) =$ cost of cheapest path from $v \to t$ that uses at most k edges
Recurrence Relation: consider the cheapest $v \to t$ path. It is the minimum of:

- **Using** $\leq k-1$ **edges**: $OPT(v, k-1)$
- **Consisting of an edge + path up to** $k-1$ **edges** *using up to* k *edges*: $\min_{w \in V, (v,w) \in E} C_{(v,w)} + OPT(w, k-1)$

$$OPT(v, k) = \min\{OPT(v, k-1), \min_{w \in V, (v,w) \in E} C_{(v,w)} + OPT(w, k-1)\}$$

- **Base Case**:

$$OPT(v, 0) = \begin{cases} 0 & \text{if } v = t \\ +\infty & \text{if } v \neq t \end{cases}$$

**I think this is stupid and I really dislike the recurrence relation – it feels redundant and the two cases are not mutually exclusive/intuitively divisible – and I feel like modifying the base case makes a lot more sense. However, I digress and acknowledge this may be useful for later when we revisit the assumption that G is acyclic. I could also be spitting complete bs.

With this, we can compute all $OPT(v, k)$ values. Order of computation will be $v = s, ..., t$ for all $k = 0, ..., n-1$. We have n^2 iterations, each computed from recurrence relations: $O(m)$, giving us $O(n^2 m)$ which can be reduced to $O(mn)$. Once we have $OPT(s, n-1)$, we can backtrace to find the original path in $O(n)$.

Now we consider if G is *not acyclic* (i.e., contains a cycle). Note that only **negative cycles** (the sum of the edge costs in the cycle is negative) are relevant to this problem**. If there exists a negative cycle in G, the entire question doesn't make sense as we can just loop around the cycle indefinitely to decrease our cost. In this case, want to return that the input is ass.

** if there are no negative cycles, then the cheapest $s \to t$ path will still have $\leq n-1$ edges

Question now becomes: **how do we detect negative cycles?**

Observe that if there exists a negative cycle (that can reach t and s), then there exists some vertex v for which there is a cheaper $v \rightarrow t$ path using n edges rather than $n-1$. Thus, we can run our algorithm up to $OPT(\_, n)$ and compare that value to $OPT(\_, n-1)$ to see if G contains a negative cycle.

Algorithm 11 Bellman Ford - $O(mn)$

Compute $OPT(v, k)$ for all $v \in V, k = 1, ..., n$ **note we use n instead of $n-1$ for cycle detection
if $OPT(v, n) < OPT(v, n-1)$ for some v **then**
    return 'negative cycle detected'
else
    backtrace from $OPT(s, n-1)$ to find and return cheapest $s \rightarrow t$ path
end if

---

4 Divide & Conquer + Randomized Algorithms

Like Dynamic Programming, Divide and Conquer is another recursive approach to solving problems. Unlike DP, D+C's recusion itself is what is needed to compute solutions faster. In my opinion, this is all made up. Anyways.

The general divide and conquer paradigm:

1. solve problem of size n by:
   - $\rightarrow$ splitting the problem into q problems of size $\frac{n}{p}$ that are solved recursively
   - $\rightarrow$ do $f(n)$ work to create recursive calls and combine results we get from them
2. analyze runtime $T(n) \leq qT(\frac{n}{p}) + f(n)$

Algorithms like **binary search** and **merge sort** are all divide and conquer;

Binary Search

Input: sorted array of n integers**, integer x **assume n is a power of 2 for ease of analysis

Output: `true` if x is in the array, `false` otherwise

Algorithm 12 Binary Search - $O(\log n)$

```
if x = a[n/2] then
    return true
else
    if x > a[n/2] then
        recurse on right half of array
    else
        recurse on left half of array
    end if
end if
```

Merge Sort

Input: unsorted array of n integers** **assume n is a power of 2 for ease of analysis

Output: sorted array

Algorithm 13 Binary Search - $O(n \log n)$

```
recurse to sort left
recurse to sort right
merge the sorted left and right into one array
```

To find a closed form runtime, we can either unroll the recursive call tree (ass approach + tedious) or use a **Master Theorem** (summary of what you will get if you did roll out that recursive tree)

Theorem 4.1 (Master Theorem)**.** *Let $T(n)$ = runtime of algorithm on input of size n. Suppose $T(n) \leq qT(\frac{n}{p}) + cn$ for some constant c and $T(n) \leq c$ for $n \leq p$. Then:*

- *if $p < q$, $T(n) = O(n^{\log_p q})$*
- *if $p = q$, $T(n) = O(n \log(n))$*
- *if $p > q$, $T(n) = O(n)$*

4.1 Quick Select (D&C + Randomization)

Input: set S of n integers, $k \in \{1, ..., n\}$

Output: the k-th smallest integer in S

Algorithm 14 Quick Select I - $O(n^2)$

```
Take an arbitrary x in S
Set S⁻ = {y ∈ S | y < x}
Set S⁺ = {y ∈ S | y > x}
Let i = |S⁻| + 1
if i == k then
    return x
else
    if i > k then
        return QSI(S⁻, k)
    else
        return QSI(S⁺, k − i)
    end if
end if
```

Runtime Analysis: Let $T(n)$ be the runtime of input size n. Then

$$T(n) \leq T(n-1) + cn$$

- $T(n-1)$ is the worst case – recurse on input size $n-1$
- cn comes from the time to compute S^- and S^+

By the master theorem, this eventually evaluates to an $O(n^2)$ runtime. Let's do better: *new idea* – look for a 'good' x using **randomization**

- We define x to ba a 'good splitter' if $\{y \in S \mid y < x\}$ and $\{y \in S \mid y > x\}$ both have $< \frac{2n}{3}$. elements (i.e., if x is in the middle third.)

Algorithm 15 Quick Select II(s, k) - $O(n)$

Let n = $|S|$
while we haven't found a good splitter **do**
  choose x from S *uniformly at random*
  Set $S^- = \{y \in S \mid y < x\}$
  Set $S^+ = \{y \in S \mid y > x\}$
  Let $i = |S^-| + 1$
  if $\frac{n}{3} < i < \frac{2n}{3}$ **then**
    then x is a good splitter
  end if
end while
if $i == k$ **then**
  return x
end if
if $i > k$ **then**
  return $QSII(S^-, k)$
else
  return $QSII(S^+, k - i)$
end if

Runtime Analysis: Let $T(n)$ be the *expected run time* on a set S of size n. Then

$$T(n) \leq T(\frac{2n}{3}) + \mathbb{E}[Z]$$

where Z = time it takes to find a good splitter (Z is a random variable!)

If we choose x at random, it is a good splitter with probability $\frac{1}{3}$. (Observe that if x is the i-th smallest value in S for $i \in \{\frac{n}{3} + 1, \frac{n}{3} + 2, ..., \frac{2n}{3}\}$, then x is a good splitter) It takes $O(n)$ time to check if a splitter is good. Therefore,

$$\begin{aligned}\mathbb{E}[Z] &= cn \times \mathbb{E}[\text{num of random } x \text{ we try until } x \text{ is a good splitter}] \\ &= 3cn \\ &= c'n \text{ for some constant } c'\end{aligned}$$

Note that the 'number of x we try until it is a good splitter' follows a **geometric distribution** of probability $p = \frac{1}{3}$ with expected value 3. Therefore,

$$T(n) \leq T(\frac{2n}{3}) + c'n$$

for some constant c' *in expectation.*

$$p = \frac{3}{2}, q = 1 \rightarrow T(n) = O(n) \qquad \textit{by the Master Theorem}$$

4.2 Multiplying Two 'Yuge' Integers

Input: let W, Z be two very large (yuge) integers that require n bits to represent in binary (so $W, Z \approx 2^n$)

Output: algorithm to calculate WZ efficientiy

5 Network Flow

We'll start be looking at the Maximum Flow – Minimum Cut Problem before moving into reductions to Network Flow questions.

5.1 The Maximum Flow + Minimum Cut Problem

Input: directed graph $G = (V, E)$, source and sink vertices $s, t \in V$, capacities $c_e > 0$ for each edge $e \in E$

Output: find a **maximum flow** from s to t that maximizes $v(f) = \sum_{e \text{ into } t} f(e)$
Recall that a valid flow is subject to constraints:

- **capacity**: $f(i, j) \leq c(i, j)$ on each arc $(i, j) \in A$
- **conservation**: flow coming into a vertex = flow going out of a vertex

Ford-Fulkerson Algorithm

At every iteration, this algorithm constructs a new **feasible flow** with larger flow value from a given feasible flow. As we can always start with a feasible flow of 0, this ends up giving us an optimal solution.

Algorithm 16 Ford-Fulkerson – $O(mC)$

```
initialize f(e) = 0 for each e ∈ E
construct the residual graph G_f = (V, E_f) for the current flow
while there is a path from s to t in the residual graph (augmenting path ) do
    let P be the augmenting path
    let δ be the minimum residual capacity along P
    for each (i, j) ∈ P do
        if (i, j) is a foward arc then
            increase the flow of f(i, j) by δ
        else
            decrease the flow of f(i, j) be δ
        end if
    end for
    update residual graph G_f
end while
let A=set of vertices reachable from s in G_f
return f, A
```

Recall that the residual graph of each iteration can be contructed as $G_f = (V, A_f)$ such that V = the same node set V as the input and A_f consists of

1. **forward arcs** for arcs in the input graph where we can *incease flow*
2. **backwards arcs** for arcs in the input graph where we can *decrease flow*

Runtime Analysis

Let $|E| = m, |V| = n$. Suppose $m \geq n$ and all capacities are integers. Let $C = \sum_{e \text{ into } t} c_e$. Then the maximum number of iterations we do in Ford Fulkerson is C because the algorithm must increase the flow value by at least 1 each iteration and the max flow has an upper bound of C.

Each iteration of Ford Fulkerson can be done in $O(m)$ – BFS or DFS on G_f is $O(m)$ and finding δ + updating the flow along P is $O(|P|) \leq O(n) \leq O(m)$. Thus, the total runtime of Ford-Fulkerson is $O(mC)$ – notpolynomial.

If we want a polynomial runtime, we can use **Edmunds-Karp** algorithm, a variation of Ford Fulkerson with runtime $O(m^2 n)$ by always choosing the shortest augmenting path P in each iteration.

Proof of Correctness

To show correctness (for FFA, not EKA), there are two parts:

1. FF algorithm maintains an $s - t$ flow f (that is, f is a valid flow satisfying capacity and conservation constraints) – *prove by induction on the number of iterations*
2. FF algorithms returns a **maximum** flow f, with the **minimum $s-t$ cut** $(A, V \setminus A)$ certifying f's optimality

To prove the second point, we will first define an **$s - t$ cut** : given a flow network, let $A \subseteq V$ and let $B = V \setminus A$: we call (A, B) an $s - t$ cut if $s \in A, t \in B$.

- For an $s - t$ cut (A, B) and $s - t$ flow f, the **net flow across** (A, B) is defined as

$$\sum_{e \text{ out of } A} f(e) - \sum_{e \text{ into } A} f(e)$$

Lemma 5.1. *If f is an $s - t$ flow and (A, B) is an $s - t$ cut, then the net flow across (A, B) is equal to $v(f)$ (the value of the flow)*

We can prove this lemma using the fact that flow conservation holds at all vertices in b except for t – the base case is that the flow is at capacity for edges leaving A and 0 for edges entering A.

Now, let's define the **capacity** of an $s - t$ cut (A, B) to be

$$c(A, B) = \sum_{e \text{ out of } A} c(e)$$

Theorem 5.2 (Max-Flow, Min-Cut Theorem)**.** *Comprised of two parts:*

1. *If f is an $s - t$ flow and (A, B) is an $s - t$ cut, then $v(f) \leq c(A, B)$*
2. *The FF algorithm returns an $s - t$ flow f^* and a set A^* such that setting $B^* = V \setminus A^*$ gives us $v(f^*) = c(A^*, B^*)$*

Proof of I:

$$\begin{aligned}
v(f) &= \text{net flow across } (A, B) \\
&= \sum_{e \text{ out of } A} f(e) - \sum_{e \text{ into } A} f(e) && \text{by Lemma} \\
&\leq \sum_{e \text{ out of } A} c(e) - \sum_{e \text{ into } A} f(e) && \text{by capacity constraints} \\
&= c(A, B) - \sum_{e \text{ into } A} f(e) && \text{by definition of } c(A, B) \\
&\leq c(A, B) && \text{because } f(e) \geq 0 \text{ for all } e \in E
\end{aligned}$$

Proof of II: Let * be the flow returned by FF Algorithm and A^* tbe set of vertices reachable from s in G_f and $B^* = V \setminus A^*$. Then $s \in A, t \in B$.

- Then all edges in G leaving A^* must be at capacity – otherwise, G_{f^*} would have a forward edge leaving A^*
- All edges in G entering A^* must have 0 flow – otherwise, G_{f^*} would have a backwards edge leaving A^*

Thus, the two inequalities hold *at equality*: $v(f^*) = c(A^*, B^*)$

Integrality Property

Note that if all capacities are integers, then the FF algorithm returns a maximum flow that has an integer flow across every edges. (This does not mean *only* integer max-flows exist, just that FF returns an integer flow.)

5.2 Bipartite Matching

Input: undirected bipartite graph $G = (V, E)$ with $V = L \cup R$ with L, R being disjoint, wich each edge $e \in E$ having one endpoint in L and one endpoint in R. Assume $|L| = |R| = n, |E| \geq n$

Output: decide whether G has a perfect matching – if yes, output it

Algorithm 17 Bipartite Matching (Reduction to Max-Flow) – $O(mn)$

```
create a flow network G' from G:
add a source s and sink t, edges (s, u) for each u ∈ L and (v, t) for each v ∈ R
direct each edge {u, v} ∈ E with u ∈ L, v ∈ R as (u, v)
set all edge capacities to 1
find a max s − t flow f* in G' using FF or EK
if v(f*) < n then
    return no
else
    return yes with matching M = {{u, v} | u ∈ L, v ∈ R, f*(u, v) = 1}
end if
```

Runtime Analysis

Proof of Correctness

5.2.1 Hall's Theorem for Bipartite Matching

Given that $A \subseteq L$, let $N(A)$ be the neighbors of A. It is impossible to have a perfect matching if A doesn't have enough neighbors. More formally:

Theorem 5.3 (Hall's Theorem)**.** *G has a perfect matching $\iff$ for all $A \subseteq L$, $|A| \leq |N(A)|$*

Proof of Correctness

5.3 Disjoint Paths Problem

Input: graph $G = (V, E)$ with sources $s_1, s_2 \in V$ and sink $t \in V$

Output: decide whether G has an $s_1 - t$ path P_1 and $s_2 - t$ path P_2 which are **disjoint** – if yes, output such P_1, P_2

Note there are many versions of this question: the graph G can be either directed or undirected, and 'disjoint' can refer to either *edge-disjoint* or *vertex disjoint* – in this case, we will examine a **directed** G' + **edge-disjoint** P_1, P_2

Algorithm 18 Disjoint Paths (Reduction to Max-Flow) – runtime

```
create a flow network G' from G:
add a 'super source' s and edges (s, s1), (s, s2)
set all edge capacities to 1
find a max s − t flow in G' using FF or EK
if v(f*) < 2 then
    return no
else
    return yes, 'trace' through (s, s1), (s, s2) to find P1, P2
end if
```

Runtime Analysis

Proof of Correctness

The one-to-one correspondence boils down to:

$$\boxed{\begin{array}{c} G \text{ has } s_1 - t \text{ path } P_1 \text{ and } s_2 - t \text{ path } P_2 \text{ that are edge disjoint} \\ \iff G' \text{ has an } s - t \text{ flow of value } 2 \end{array}}$$

To prove the correctness of a one-to-one correspondence (usually an $\iff$), prove both directions. Use the **contrapositive** when appropriate: recall that

$$p \implies q \equiv \bar{q} \implies \bar{p}$$

5.4 Project Matching

Input: set V of n possible projects, a profit p_i for every $i \in V$ where p_i can be $>, <, = 0$, and a set of co-requisites $R(i) \subseteq V$ for each $i \in V$

Output: find a valid** project selection of maximum total profit

**a selection is valid if for the set $A \subseteq V$, $R(i) \subseteq A$ for each $i \in A$

Note that *maximizing the profit of the projects we do choose* is equivalent to *minimizing the profit of the projects we forgo*

Given $A \subseteq V$, let

- $p(A) = \sum_{i \in A} p_i$ – total profit
- $p^+(A) = \sum_{i \in A, p_i \geq 0} p_i$ – total *positive* profit
- $p^-(A) = \sum_{i \in A, -p_i < 0} p_i$ – total *negative* profit (negated, so a positive value)

Now, we can rewrite the profit $p(A)$ as

$$\begin{aligned} p(A) &= p^+(A) - p^-(A) \\ &= p^+(V) - p^+(V \setminus A) - p^-(A) \end{aligned}$$

and perform a reduction to the minimum cut.

Algorithm 19 Project Selection (Reduction to Min-Cut) – runtime

create a flow network $G' = (V', E')$
set $V' = V \cup \{s, t\}$
for $i \in V$ and $j \in R(i)$ **do**
  create an edge (i, j) with $c_{ij} = \infty$
end for
for $i \in V$ with $p_i \geq 0$ **do**
  create an edge (s, i) with $c_{si} = p_i$
end for
for $i \in V$ with $p_i < 0$ **do**
  create an edge (i, t) with $c_{it} = -p_i$
end for
find a minimum $s - t$ cut $(A^*, B*^)$
return $A^* \setminus \{s\}$ **as the project selection**

Proof of Correctness

6 NP Completeness

Let X, Y be problems: we will write $Y \leq_p X$ as a shorthand for: *if there exists a polynomial time algorithm for X, then there is a polynomial time algorithm for Y as well*, i.e., Y is not harder to solve in polynomial time than X. Note that $\leq_p$ is transitive.

A **decision problem** is a problem to which the output is either YES or NO.

Some classifications of problems:

- P : the set of decision problems for which we have a **polynomial time algorithm** to solve
- NP : the set of decision problems for which every YES-input has a polynomial sized **certificate** that takes **polynomial time to verify** – the decision problem itself is *non-deterministic polynomial time*
- Co-NP : decision problems for which there is a polynomial certificate and certifier for every NO-input
- NP-Hard : set of problems (not necessarily constrained to *decision* problems) that any NP problem can be reduced to – i.e., problems that are harder to solve than any NP problem
- NP-Complete : set of decision problems that are the 'hardest' problems in NP, i.e., NP-Complete = NP ∩ NP-Hard

We know that

$$\text{P} \subseteq \text{NP}$$

every problem in P is also in NP. To prove this, we simply take the certifier algorithm to be the algorithm itself. Now, the question becomes:

Is P = NP? Is there a problem in NP that is not in P?

(We don't know)

We'll focus a lot on showing problems are NP-Complete. To show that a problem Y is $\in$ NP-Complete:

1. Show that $Y \in$ NP by showing there exists a polynomial-sized **certificate** corresponding to a yes-input AND a polynomial-time **certifier** algorithm that takes the input plus certificate and verifes correctness
2. Show that $Y \in$ NP-Hard by showing that for some NP-Complete problem X, $X \leq_p Y$ – i.e., Y is at least as hard as an NP-Complete problem. To do so, we usually need to show:
   - we can transform the input I to X into an input I' for Y
   - prove the reduction is correct:
     $$I \text{ to } X \text{ outputs YES} \iff I' \text{ to } Y \text{ outputs YES}$$
   - show the reduction is polynomial time (+ sized)

6.1 Independent Set

Input: undirected graph $G = (V, E)$, integer $k > 0$

Output: Y/N to 'is there an $S \subseteq V$ with $|S| = k$ such that each edge $e \in E$ has at most one endpoint in S?' – i.e., *is there a subset S w/ size k with no edges in between?*

6.2 Vertex Cover

Input: undirected graph $G = (V, E)$, integer $k > 0$

Output: Y/N to 'is there a $T \subseteq V$ with $|T| = k$ such that each edge $e \in E$ has at least one endpoint in T?'

6.3 Set Cover

Input: set U of n elements, m sets $S_1, ..., S_m$ that are subsets of U, integer $k > 0$

Output: Y/N to 'is it possible to choose k elements from $S_1, ..., S_m$ such that their union is U?'

6.4 Satisfiability (SAT)

Input: a formula Φ over n boolean (T/F) variables in **conjective normal form (CNF)** – Φ is a conjunction (AND) of m clases where each clause is a disjunction (OR) or literals (variables + negated variables)

Output: Y/N to 'does Φ have a satisfying assignment?'

6.5 Hamiltonian Cycle (+ TSP)

First, we will look at the Traveling Salesman problem. For this problem, we take in

- **Input**: complete directed graph w/ edge costs
- **Output**: simple cycle (no repeated vertices) through all vertices with a minimum total cost

Note that the Traveling Salesman problem is *not* a decision problem, so it cannot be in NP or NP-Complete. However, it is NP-Hard.

Now, we will look at the Hamiltonian Cycle.

Input: directed graph $G = (V, E)$

Output: Y/N to 'does G have a simple cycle that goes through all vertices?'

We can see that Hamiltonian Cycle $\leq_p$ Traveling Salesman:
Given an input $G = (V, E)$ to Hamiltonian Cycle, we can create a Traveling Salesman input G', c' where G' has vertices V and all possible edges. Set c'_e to be 0 if $e \in E$ and 1 otherwise. G has a Hamiltonian Cycle $\iff$ G' has a Traveling Salesman tour of cost 0.

6.6 Subset Sum (+ Knapsack)

Input: n integers $w_1, ..., w_n > 0$, target integer W **assume $w_i \leq W$ for $i = 1, ..., n$

Output: Y/N to 'does there exist a subset I of $\{1, ..., n\}$ such that $\sum_{i \in I} w_i = W$?'

7 Approximation Algorithms

Instead of always searching for the **optimal solution** in polynomial time, we now pivot to searching for a solution that is **'good enough'** in polynomial time. We call this an **α-approximation algorithm** , where we are guaranteed to be α times within the optimal solution.

We'll be using some concepts from probability:

1. **Linearity of Expectation**: $\mathbb{E}[aX + bY] = a\mathbb{E}[X] + b\mathbb{E}[y]$

   Recall that if we express a random variable Y in terms of a sum of indicator variables Z_i such that $Y = \sum_{i=1}^{m} Z_i$, then we know

   $$\mathbb{E}[Y] = \sum_{i=1}^{m} \mathbb{E}[Z_i]$$

2. **Markov's Inequality**: $\Pr(Y \geq k) \leq \frac{\mathbb{E}[Y]}{k}$

   Take note that this is a $Y \geq k$ – for strictly greater thans, modify accordingly.

3. **(Expectation of) Geometric Distribution**: if X is the number of trials until the first success (given that trials are mutually independent and have probability of (at least) p)

   $$\mathbb{E}[X] \leq \frac{1}{p}$$

7.1 Load Balancing Problem

Input: n jobs with processing times $t_1, ..., t_n \geq 0$, m machines

Output: assign the jobs to machines to minimize the **makespan** (time for all jobs to be processed) *i.e., minimize the maximum overall machine time/load*

Algorithm 20 Greedy Balance Algorithm - $O(RUNTIME)$

Consider the jobs in arbitrary order
Assign the job to the machine with the smallest current load

elaborate bruh

7.2 MAX-3 SAT

Input: CNF formula Φ with m disjuntive clauses, each containing *exactly* 3 literals over n boolean variables

Output: find a truth assignment that satisfies the maximum number of clauses

**This is not a decision problem, so this is not in NP– however, it is NP-Hard

Algorithm 21 Random-Assignment Algorithm - $O(RUNTIME)$

for $j = 1$ to n **do**
  set x_j to TRUE with probability $\frac{1}{2}$ and FALSE otherwise
end for

Let Y = number of clauses satisfied by the algorithm's solution. Then

$$Y = \sum_{i=1}^{m} Z_i$$

where $Z_i = \begin{cases} 1 \text{ if the } i\text{th clause is satisfied} \\ 0 \text{ otherwise} \end{cases}$

Then

$$\begin{aligned} \mathbb{E}[Z_i] &= 0 * \Pr(Z_i = 0) + 1 * \Pr(Z_i = 1) \\ &= 0 * (\frac{1}{2})^3 + 1 * (1 - (\frac{1}{2})^3) \\ &= \frac{7}{8} \end{aligned}$$

By **linearity of expectation**,

$$\mathbb{E}[Y] = \sum_{i=1}^{m} \mathbb{E}[Z_i] = \frac{7m}{8} = 87.5\%$$

If the *expected* number of clauses satisfied by a random assignment is $\frac{7m}{8}$, then there *must exist* an assignment that satisfies at least $\frac{7m}{8}$ clauses.

We call this the **probabilistic method** – prove existence by arguing about properties of a random solution.

Algorithm 22 Randomized MAX-3 SAT **Approximation** Algorithm

run the Random Assignment Algorithm until we find a truth assignment that satisfies at least $\frac{7}{8}$ of the clauses

Let N be the number of times we run the Random Assignment Algorithm – $\mathbb{E}[N]$?

- By the expectation of the geometric distribution, we know that $\mathbb{E}[N] = \frac{1}{p}$ where $p = \Pr(Y \geq \frac{7}{8})$
- To find an *upper bound* on $\mathbb{E}[N]$ we need a *lower bound* on p – we can use Markov's Inequality to find an *upper bound* on $1 - p$

$$1 - p = \Pr(Y < \frac{7m}{8}) = Pr(\text{\# of unsatisfied clauses} > \frac{m}{8})$$

**for simplicity sake, we are going to assume m is a multiple of 8

$$\begin{aligned}
1 - p &= \Pr(\text{\# of unsatisfied clauses} > \frac{m}{8}) \\
&= \Pr(\text{\# of unsatisifed clauses} \geq \frac{m}{8} + 1) \\
&\leq \frac{\mathbb{E}[\text{\# of unsatisfied clauses}]}{\frac{m}{8} + 1} \\
&= \frac{\frac{m}{8}}{\frac{m}{8} + 1} = 1 - \frac{1}{\frac{m}{8} + 1}
\end{aligned}$$

Finally, we can conclude that

$$p \geq \frac{1}{\frac{m}{8} + 1} \text{ and } \mathbb{E}[N] \leq \frac{m}{8} + 1 \in O(m)$$

7.3 Knapsack Problem

7.4 (Online) Bipartite Matching

8 Computability

9 Cryptography

10 Tldr

10.1 Stable Matching

Gale-Shapley Algorithm: $O(n^2)$

- **Input**: integer n, two disjoint sets H, R, with $|H| = |R| = n$, and a permutation of set R for each $h \in H$ and a permutation of set H for each $r \in R$
- **Output**: a stable matching M on H and R
- GSA returns a *hospital optimal*, *resident pessimal* matching

10.2 Greedy Algorithms

Interval Scheduling: $O(n \log(n))$ (Earliest Finish Time Algorithm)

- **Input**: n intervals (jobs), each with a start time s_j and finish time f_j
- **Output**: a set $A \subseteq \{1, ..., n\}$ of *nonoverlapping* intervals that is as large (cardinality wise) as possible

Minimum Spanning Trees (MST): $O(m \log(n))$ (Kruskal's, Prim's)

- **Input**: an undirected graph $G = (V, E)$ with edge costs c_e for each $e \in E$
- **Output**: $T \subseteq E$ such that (V, T) is a spanning tree that minimizes $\sum_{e \in T} c_e$
- the **Cut Property**: the cheapest edge in cut $\delta(S)$ must be in the MST
- the **Cycle Property**: the most expensive edge in cycle C cannot be in the MST

Scheduling: Minimizing Maximum Lateness: $O(n \log(n))$

- **Input**: n jobs, each with a processing time t_j and deadline d_j
- **Output**: a schedule S that minimizies the maximum lateness for any individual job

Huffman Codes: $O(n \log(n))$

- **Input**: an alphabet Σ and the frequencies f of the characters in Σ
- **Output**: a prefix code minimizing the length of the encoded text

10.3 Dynamic Programming

Weighted Interval Scheduling: $O(n \log(n))$

- **Input**: n jobs (intervals), each job with a starting and ending interval $[s_j, f_j]$ and weight or value v_j **

**Assume jobs are sorted by finish time

- **Output**: maximum weight set of non-overlapping jobs **

  ** No correct greedy algorithm known

Segmented Least Squares: $O(n^2)$

- **Input**: n points $p_1, p_2, ..., p_n \in \mathbb{R}^2$ ordered by x-coordinate
- **Output**: partition of the points into segments minimizing:
  $C \times$number of segments$+$total SSE (sum of squared error) for all line segments (called the **objective value**)
  ** where a segment is a sequence of consecutive points $p_i, p_{i+1}, ..., p_j$, C is a tunable penalty parameter

Knapsack Problem: $O(nW)$ (pseudopolynomial)

- **Input**: n items, labelled $1, ..., n$ where item j has weight w_j and value v_j and a weight limit W

  ** w_j, v_j, W all positive integers

- **Output**: a maximum value subset of the items whose combined weight $\leq W$

Shortest Path: Bellman Ford: $O(mn)$

- **Input**: directed graph $G = (V, E)$ with edge costs c_e for each $e \in E$, special start and end vertices $s, t \in V$
- **Output**: the shortest (cheapest) path from s to t (or telling us the graph contains a negative cycle)

10.4 Divide & Conquer

Binary Search: $O(\log(n))$

- **Input**: sorted array of n integers**, integer x

  **assume n is a power of 2 for ease of analysis

- **Output**: `true` if x is in the array, `false` otherwise

Merge Sort: $O(n \log(n))$

- **Input**: unsorted array of n integers**

  **assume n is a power of 2 for ease of analysis

- **Output**: sorted array

Quick Select: $O(n)$

- **Input**: set S of n integers, $k \in \{1, ..., n\}$
- **Output**: the k-th smallest integer in S