CPSC 041. Algorithms

Fall 2024

Accumulated Notes

Roger Wang

Lecture 1

2024-09-04
Week 1, Wed

What is An Algorithm?

- CLRS Algorithm Definition: An algorithm is any well-defined computational procedure that takes some value(s) as input and produces some value(s) as output.

- 3 aspects relevant for CS41:
  1. The algorithm must always halt eventually.
  2. An algorithm for solving problem X must always return what problem X asks for.
  3. An algorithm does not use randomness ("random" vs. "arbitrary").

The Hiking Problem:

Your friend is on the Appalachian Trail (AT). You want to meet your friend. You don't know the trail at all but find yourself on the trail.

Q: How do you find your friend?

Algorithm #1:

Hike North until we reach our friend or if the trail ends. Hike South if we reached the Northern terminus.

Q: Is this a good algorithm?

A: No. For example, if you friend is one mile south of you at the Southern terminus, you end up walking almost twice the trail.

Algorithm #2:

$k = 1$. While we don't friend: hike k miles N, hike $2k$ miles S, hike k miles N (returns to the origin), $k = k + 1$.

Analysis:

Case 1: friend is m miles North.

Last iteration: m miles.

For non-final iterations:

For $k = 1, \dots, m - 1$:

We travel $4k$ miles.

Total distance: $m + \sum_{k=1}^{m-1} 4k = m + 4\sum_{k=1}^{m-1} k$

This algorithm travels $O(m^2)$ miles if friend is m miles away.

Lecture 2

2024-09-06
Week 1, Fri

The Hiking Problem (Continued):

Algorithm #3 (Binary Exponential Backoff):

$k = 1$. Repeat until you find friend: hike k miles N, hike $2k$ miles S, hike k miles N (returns to the origin), $k = 2k$.

Analysis:

Case 1

Friend is m miles North.

- Final Iteration: m miles.
- Other Iterations: $4k$ miles for $k = 1, 2, 4, 8, 16, \dots = 2^0, 2^1, 2^2, 2^3, 2^4, \dots$.

Q: What is the final iteration?

A: At least t such that $2^t \geq m$.

Total Distance:

$$m + \sum_{i=0}^{t-1} 4 \cdot 2^i = m + 4 \cdot \sum_{i=0}^{t-1} 2^i = m + 4\left(2^t - 1\right) < m + 4 \cdot 2m = 9m = \mathrm{O}(m)$$

Case 2

Friend is m miles South.
- Final Iteration: $2 \cdot 2^t + m$ miles.
- Other Iterations: $4 \cdot \sum_{i=0}^{t-1} 2^i$ miles.

Q: What is the final iteration?
A: At least t such that $2^t \geq m$.

Total Distance:

$$2 \cdot 2^t + m + 4 \sum_{i=0}^{t-1} 2^i$$

4 Tasks of Algorithm Design:

- Formulate problem.
- Understand underlying structure.
- Design algorithm.
- Analyze algorithm.
- Separate design & analysis, as they are separate skills.

National Resident Matching Program (NRMP):

Input: List of n hospitals, list of n doctors. Each hospital has a preference list of doctors. Each doctor has a preference list of hospitals.
Output: List of n (hospital-doctor) matchings.

For example:

Hospitals: Abington, Brandywine, CHOP, DCMH
Doctors: Alice, Bob, Carol, Dave
Output: Abington-Carol, Brandywine-Bob, DCMH-Alice, CHOP-Dave.

- Matching: A set of (hospital-doctor) pairs where everyone is matched at most once. A matching S is perfect is everyone is matched.

For example:

$d_2 > d_1 > d_3$	h_1		d_1	$h_1 > h_2 > h_3$
$d_1 > d_2 > d_3$	h_2		d_2	$h_3 > h_1 > h_2$
$d_2 > d_3 > d_1$	h_3		d_3	$h_2 > h_3 > h_1$

Lecture 3

2024-09-11
Week 2, Wed

The lecture on 2024-09-09 was cancelled due to instructor sickness.

The NRMP Problem (Continued):

Definition: (h, d) is an instability in a matching S if:
1. (h, d') and (h', d) are in S.
2. h prefers d to d'.
3. d prefers h to h'.

Definition: A stable matching is a perfect matching without instabilities.
Q: Do stable matchings always exist? If so, can we produce stable matchings efficiently?
A: Yes. And yes.

Algorithm #1 (Gale-Shapley Intuition):

Hospitals "propose" matchings while doctors can accept/reject proposals based on their preferences.

Initialize all hospitals, doctors to be free.
While there is a hospital and a doctor that the hospital hasn't proposed to:
    Choose this hospital h.
    d = the highest-ranked doctor h hasn't yet proposed to.

If d is free:
    h, d become engaged.
Else:
    If d prefers h to the current assignment h':
        h' becomes free.
        h, d become engaged.
Return set S of engaged pairs.

Q: Is this algorithm correct? (i.e., does it return a perfect matching? If so, is it stable?)
A: Yes.

Q: Is it efficient?
A: G-S algorithm terminates after at most n^2 iterations.
Proof:
Idea: Define measure of progress, show that progress is always made, and there's only so much progress to make.
Examples:
1. $M(k)$ is the number of engaged pairs after the kth iteration of loop.
   $M(k) \leq n$.
   However, sometimes $M(k) = M(k-1)$ (we sometimes break an engagement while making one).
   Therefore, this is not a good measure of progress.
2. $P(k)$ is the number of $h - d$ proposals made after the kth iteration of the loop.
   $P(k) \leq n^2$ as there are n hospitals and n doctors.
   $P(k) = P(k-1) + 1$.

Therefore, G-S runs in $O(n^4)$ time. This is the case since each loop iteration takes $O(n^2)$ time, as choosing a hospital h which can make a proposal might take $O(n^2)$ time.

Lecture 4

2024-09-13
Week 2, Fri

The Gale-Shapley Algorithm (Continued):

Facts:
1. Hospitals always propose to doctors in decreasing preference order.
2. Doctors become engaged the first chance they get, and they stay engaged unless when they immediately get a new engagement.
3. Doctors always accept proposes in increasing preference order.

Claim 1: The S returned by the G-S algorithm is a matching.
Claim 2: The matching is perfect.
Claim 3: S is stable.

Proof of Claim 3 (by Contradiction):

Assume that S has an instability: (h, d'), (h', d) are matchings in S where h prefers d to d' and d prefers h to h'.
Note that at the end of the algorithm, h matched with d'. Also, h prefers d to d'. By Fact 1, h at some point proposes to d.
- Either d was free but eventually broke engagement in favor of some hospital $h'' > h$.
- Or, d engaged but preferred h, so become engaged to h… but then eventually breaks engagement for some $h'' > h$.
- Or, d engaged to some $h'' > h$ and decides to reject h.

In any case, at some point d makes the decision $h'' > h$. Eventually, d ends up with hospital h'.
By Fact 3, $h' \geq h'' > h$, so d prefers h' to h, contradicting instability.

Efficient Algorithms:

Q: What is an efficient algorithm?
A: (Textbook Definition) An algorithm is efficient if it uses a polynomial amount of resources (e.g., n, $n\log(n)$, n^4, …). However, this definition is not perfect as bubble sort, which uses a polynomial runtime, is generally considered not efficient.
Definition: A problem is tractable if there is an efficient algorithm for it.

Why are inefficient algorithms bad?

n	n^2	$n\log(n)$	2^n

1	1	0	2
2	4	2	4
4	16	8	16
10	100	~332	1024
100	10,000	~604	Lots (126765060022822940149670320)

Lecture 5

2024-09-16
Week 3, Mon

Lecture by Lila Fontes

Efficient Algorithms:

- Recall that an algorithm is efficient if it uses a polynomial amount of resources.

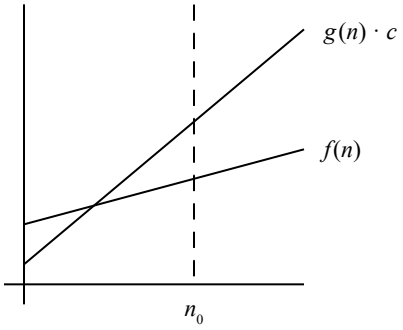
Q: What are the resources?
A: Time, storage space, threads, hardware, electricity.

Search Space & Brute Force:

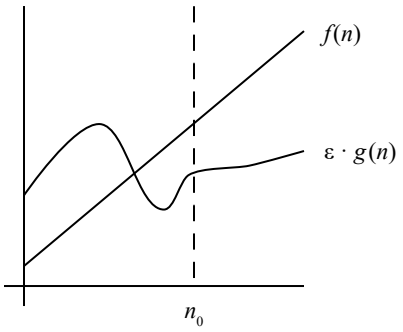
- Search Space: The set of all possible solutions to a problem is the search space.
- Brute Force Search Algorithm: An algorithm that checks every possible solution in the search space.

Upper Bound, Lower Bound, and Tight Bound:

Def: Let f and g be functions $\mathbb{N} \to \mathbb{N}$. $f(n)$ is $O\big(g(n)\big)$ if there exist constants $c > 0$ and $n_0 > 0$ such that, for all $n \geq n_0$, $f(n) \leq c \cdot g(n)$. In this case, we say "f is big O of g", or "f is asymptotically upper-bounded by g".



Def: $f(n)$ is $\Omega\big(g(n)\big)$ if there exist constants $\varepsilon > 0$ and $n_0 \geq 0$ such that, for all $n \geq n_0$, $f(n) \geq \varepsilon \cdot g(n)$. In this case, we say "$f$ is big Omega of g", or "f is asymptotically lower bounded by g".



Def: If f is $O(g)$ and f is $\Omega(g)$, then f is $\Theta(g)$.

Example

$f(n) = 3n^2 + 7n + 12$.

Claim: $f(n)$ is $O\big(n^2\big)$.
Proof: We need to find c and n_0 such that $f(n) \leq c \cdot g(n)$ for all $n \geq n_0$.
Pick $c = 22$ and $n_0 = 1$, we have:

$3n^2 + 7n + 12 \leq 3n^2 + 7n^2 + 12$
$3n^2 + 7n + 12 \leq 10n^2 + 12$
$3n^2 + 7n + 12 \leq 10n^2 + 12n^2$ (Note that in this step we assumed $n \geq 1$.)
$3n^2 + 7n + 12 \leq 22n^2$
Thus, we have $3n^2 + 7n + 12 \leq cn^2$, $f(n) \leq c \cdot g(n)$. ∎

Claim: $f(n)$ is $\Omega\left(n^2\right)$.
Proof: Let $\varepsilon = 3$ and $n_0 = 0$.
$3n^2 + 7n + 12 \geq 3n^2 + 0n^2 + 12$ (Note that in this step we assumed $7n \geq 0n^2$, as $n \geq n_0 = 0$.)
$3n^2 + 7n + 12 \geq 3n^2 + 0 + 0n^2$
$3n^2 + 7n + 12 \geq 3n^2$
$3n^2 + 7n + 12 \geq \varepsilon \cdot n^2$
Thus, we have $f(n) \geq \varepsilon \cdot g(n)$ for all $n \geq n_0$. ∎

Claim: $f(n)$ is $\Theta\left(n^2\right)$.
Proof: Since $f(n)$ is both $O\left(n^2\right)$ and $\Omega\left(n^2\right)$, the definition of Θ is satisfied. ∎

Lecture 6

2024-09-18
Week 3, Wed

Lecture by Lila Fontes

Theorem: If $f(n)$ is $O\big(h(n)\big)$ and $g(n)$ is $O\big(h(n)\big)$, then $f(n) + g(n)$ is $O\big(h(n)\big)$.

Q: If $f(n)$ is $O\big(h(n)\big)$ and $g(n)$ is $O\big(h(n)\big)$, then is $\big(f(n) \cdot g(n)\big)\ O\big(h(n)\big)$?
A: No.
Counter Example: $f(n) = n^2$, $g(n) = n^2$, $h(n) = n^2$. We can easily check that $f(n)$ is $O\big(h(n)\big)$ and $g(n)$ is $O\big(h(n)\big)$. We now need to check that $f(n)$ is $f(n) \cdot g(n) = n^4$ is not $O\big(h(n)\big)$.
Proof: In order to show that n^4 is not $O\left(n^2\right)$, we need to show that there are no possible constants $c > 0$ and $n_0 \geq 0$ which satisfy the definition of big O. We will prove this by contradiction.
Assume that there are constants $c > 0$ and $n_0 \geq 0$ such that $n^4 \leq c \cdot n^2$ for all $n \geq n_0$. Note that $n^2 \cdot n^2 \leq c \cdot n^2$ and $n^2 \leq c$.
However, if $n = \max\left(\sqrt[2]{c+1}, n_0\right)$, then $n^2 = \max\left(c + 1, n_0^2\right) > c$.
This is a contradiction. Therefore, the assumption must not be true. ∎

Claim: If $f(n)$ is $O\big(h(n)\big)$ and $g(n)$ is $O\big(h(n)\big)$, then is $\big(f(n) \cdot g(n)\big)\ O\big(h(n) \cdot h(n)\big)$?
Proof: We know $f(n) \leq c_1 \cdot h(n)$ for $n \geq n_1$ and $g(n) \leq c_2 \cdot h(n)$ for $n \geq n_2$. Multiplying gives us $f(n) \cdot g(n) \leq \big(c_1 \cdot h(n)\big)\big(c_2 \cdot h(n)\big) = c_1 c_2 h(n) h(n)$ for $n \geq \max(n_1, n_2)$. ∎

Theorem: If $f(n)$ is $O\big(g(n)\big)$, then $\log f(n)$ is not necessarily $O(\log g(n))$.
Counter Example: Constant functions.

Theorem: If $f(n)$ is $O\big(g(n)\big)$, then $2^{f(n)}$ is not necessarily $O\big(2^{g(n)}\big)$.
Example: $f(n) = 3n$ and $g(n) = n$, $2^{f(n)} = 2^{3n} = \left(2^3\right)^n = 8^n$. $2^{g(n)} = 2^n$.
Claim: 8^n is not $O(2^n)$.
Proof: In order to show that 8^n is not $O(2^n)$, we need to show that there are no possible constants $c > 0$ and $n_0 \geq 0$ which satisfy the definition of big O. We will prove this by contradiction.
Assume that there are constants $c > 0$ and $n_0 \geq 0$ such that $8^n \leq c \cdot 2^n$ for all $n \geq n_0$. Note that $8^n \leq c \cdot 2^n$ and $4^n \leq c$.
However, if $n = \max(n_0, \log_4(c + 1))$, then $4^n > c$.
This is a contradiction. Therefore, the assumption must not be true. ∎

Lecture 7

2024-09-20
Week 3, Fri

Lecture by Lila Fontes

Common Runtimes:

- Polynomial.
- Logarithmic.
- Exponential.

Polynomial Runtime:

Def: A polynomial is a function like $f(n) = n^k$ where k is a constant.

Logarithmic:

Def: $x = \log_b(n)$ means that x is the number such that $b^x = n$.

Fact: For any constants $b > a > 1$, $\log_a(n)$ is $O(\log_b n)$.

Proof: $n = b^x$ so $x = \log_b(n)$. $\log_a(n) = \log_a(b^x)$. Taking $\log_a$ of both sides, $\log_a(n) = x \cdot \log_a(b)$. Thus, $\log_a(n)$ is $O(\log_b(n)) = O(x)$. By the same reasoning (swapping a and b), $\log_b(n)$ is $O(\log_a(n))$.

Fact: Let f and g be such that $\lim_{n\to\infty} \frac{f(n)}{g(n)} = c$ constant. if $c > 0$, then $f(n)$ is $O\big(g(n)\big)$. If $c = 0$, then $f(n)$ is $O\big(g(n)\big)$ but $g(n)$ is not $O\big(f(n)\big)$.

Fact: For any $\varepsilon > 0$, $\log(n)$ is $O(n^\varepsilon)$.

Proof: $h(n) = \frac{\log(n)}{n}$, $h'(n) = \frac{-\ln(n)-1}{n^2}$. We want to show that $h'(n) < 0$ for sufficiently large n. $h'(n) < 0$ if and only if $1 < \ln(n)$.

Therefore, $h(n)$ is decreasing for $n \geq 3$. $h(3) = \frac{\ln(3)}{3}$.

For all $n \geq n_0 = 3$, $\frac{\ln(n)}{n} = h(n) < h(n_0) = \frac{\ln(3)}{3}$. ∎

Exponential:

Def: $f(n) = r^n$ for constant $r > 1$.

Fact: For all $r > 1$ and all $d > 0$, n^d is $O(r^n)$.

Lemma: If $r > s > 1$, s^n is $O(r^n)$, but r^n is not $O(s^n)$.

Proof: If $r > s$, then $\frac{r}{s} > 1$, $\frac{s}{r} < 1$. $\left(\frac{r}{s}\right)^n$ is increasing; $\left(\frac{s}{r}\right)^n$ is decreasing. Therefore, $\lim_{n\to\infty} \left(\frac{s}{r}\right)^n = 0$. ∎

Common Runtime Comparison:

"Efficient"	$O(1)$	Constant	
	$O(\log n)$	Logarithmic	
	$O(n)$	Linear	
	$O(n \log n)$		(e.g., merge sort)
	$O\big(n^2\big)$	Quadratic	(e.g., bubble sort, quick sort)
	$O\big(n^3\big)$	Cubic	(e.g., triply nested loops)
	$O\big(n^k\big)$	Polynomial	
	$O(2^n)$	Exponential	(e.g., brute force search)
	$O(n!)$	Factorial	

Claim: For $f(n) = 2(\log(n))^3 + 6$, $g(n) = 5n^{\frac{1}{4}} + 10$, $f(n)$ is $O\big(g(n)\big)$.

Proof: Use the log & poly fact with $\varepsilon > \frac{1}{12}$: $\log(n)$ is $O\left(-n^{\frac{1}{12}}\right)$.

From Wednesday, we know that if $h(n)$ is $O\big(m(n)\big)$, then $h(n) \cdot h(n)$ is $O\big(m(n) \cdot m(n)\big)$. Therefore, $h(n) = \log(n)$, $m(n) = n^{\frac{1}{2}}$; $(\log n)^2$ is $O\left(n^{\frac{2}{12}}\right)$.

Using the same fact again, $\log(n)$ is $O\left(n^{\frac{1}{12}}\right)$, $(\log(n))^2$ is $O\left(n^{\frac{2}{12}}\right)$. So $(\log n)(\log n)^2 = (\log(n))^3$ is $O\left(n^{\frac{3}{12}}\right) = O\left(n^{\frac{1}{4}}\right)$.

2 is $O(5)$, so $(\log n)^3$ is $O\left(n^{\frac{1}{4}}\right)$ and $2 \cdot (\log n)^3$ is $O\left(5n^{\frac{1}{4}}\right)$.

Since 6 is $O(10)$, we can use the fact that if f is $O(h)$ and g is $O(d)$, then $f + g$ is $O(h + d)$ to get $2(\log(n))^3 + 6$ is $O\left(5n^{\frac{1}{4}} + 10\right)$. ∎

Lecture 8

2024-09-23
Week 4, Mon

- High-Level Pseudocode: Abstract pseudocode that can be used to argue about the correctness of the algorithm.
- Low-Level Pseudocode: Pseudocode containing abstract data types which can be used to argue about both the correctness and runtime of the algorithm.

Recap (ADT List Runtime):

	Sample Operations	Array/ArrayList	LinkedList
	query(*i*): return *i*th element	$O(1)$	$O(n)$
	search(*e*): is *e* in the list?	$O(n)$ $O(\log n)$ if sorted	$O(n)$
	first(): return the first item in the list	$O(1)$	$O(1)$
Dynamic Operations	**add(*e*):** add *e* to the end of the list *// What about add to the beginning?*	$O(1)$ if capacity $O(n)$ otherwise	$O(1)$
	delete(*e*): delete *e* from the list	$O(n)$	$O(1)$ plus **search()**

Better Runtime for Gale-Shapely:

Algorithm:

Initialize all hospitals, doctors as free.
While there is a free hospital and a doctor that the hospital hasn't proposed to:
  Choose this hospital h.
  d = highest ranked doctor in h's preference list that they haven't proposed to.
  If d is free:
    (h, d) become engaged.
  Else: // (h', d) is engaged.
    If d prefers h to h':
      (h, d) become engaged.
      h' becomes free.
Return the set S of engaged pairs.

Inputs:

- List of hospitals: **Hospital[1, …, *n*]**.
- List of doctors: **Doctor[1, …, *n*]**.
- For each hospital, the list of preferences: **HPref[*h*, *i*]** // *i*th most preferred doctor in *h*'s preference list.
- For each doctor, the list of preferences: **DPref[*d*, *i*]** // *i*th most preferred hospital in *d*'s preference list.

Tasks (We Need to Do in Each Iteration):

1. Identify a free hospital that hasn't proposed to all the doctors.
2. Identify the highest ranked doctor *d* that *h* hasn't proposed to.
3. Check to see if *d* is free or identify who *d* has engaged to.
4. Compare preferences of *h* vs. h' in *d*'s preference list.

* *Assume all input lists are arrays.*

Task 1 (List of Free Hospitals):

- Start out with all the hospitals free.
- Delete a hospital when it becomes engaged.
- Add a hospital to the free list when the engagement is broken.
- Choose a hospital *h* (that is free and hasn't proposed to all the doctors).
- Thus, for preprocessing, we add each hospital to the list and return the first hospital.
- Solution: Use a **LinkedList** so that we have total work inside the loop be $O(1)$.

Task 4:

- Compare *h* to h' in *d*'s preferences.
- Solution: Maintain an array of rankings. E.g., **Rank[*d*, *h*]** // Rank of *h* in *d*'s preferences. // **Rank[*d*, *h*] = *i*** if **DPref[*d*, *i*] = *h***.

Lecture 9 2024-09-25
Week 4, Wed

Graph Algorithms:

- A graph is an ordered pair $G = (V, E)$.
- V is the set of vertices.
- E is the set of edges.
- Edges can be directed or undirected.
- Conventions: n is the number of vertices while m is the number edges.

Graph Data Structures:

- Adjacency Matrix: $O(n^2)$ space; an $n \times n$ matrix M where $M[i, j] = \{1 \text{ if } (i, j) \in E, ; 0 \text{ otherwise}\}$.
- Adjacency List: $O(n + m)$ space; list A of n lists where $A[i]$ is the list of neighbors of i.

Graph Operations	Adjacency Matrix	Adjacency List
Edge Query ***Is*** $(i, j) \in E$;	$O(1)$	$O(\text{degree}) = O(n)$
search(*e*): is e in the list?	$O(1)$	$O(\text{degree})$

$\text{degree}(v) = \#$ of neighbors of v.

Breadth-First Search (BFS):

Algorithm:

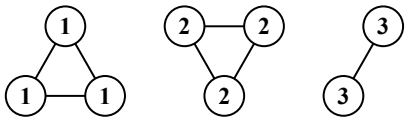
```
BFS(Vertex s)
    Queue q                                          O(1)
    Visit(s)                                         O(1)
    q.enqueue(s)                                     O(1)
    while(q not empty):
        v ← q.dequeue()                              O(1), only happens ≤ n times.
        for each neighbor u of v:                    O(degree(v))
            if u not visited:                        O(1) per iteration
                Visit(u)                             O(1) per iteration
                q.enqueue(u)                         O(1) per iteration
Visit(s)
    s.visited ← true
    // Do stuff.
```

Total Work:

$$O(1) + \sum_v 1 + O\big(\text{degree}(v)\big) = O(1) + \sum_v 1 + \sum_v \text{degree}(v) = O(1) + O(n) + O(m) = O(n + m)$$

Applications:

1. S-T connectivity.
   Input: $G(V, E)$, $s, t \in V$.
   Output: Yes if there is a path from s to t. No otherwise.
   Idea: Maintain previous vertex, then when you find the target vertex, follow nodes back to the source vertex.
2. Connectivity.
   Input: $G(V, E)$.
   Output: Are all vertices connected?
3. Find all components of a graph.



   Idea: Instead of keeping track if nodes are visited, maintain $k = 1$ and Component$[i]$.
   Component$[i] = 0$ if i is not visited yet and Component$[i] = k$ if i is in component k.
   For each unvisited node v: BFS(v, k), $k = k + 1$.

Bipartite Graphs:

- **Def:** $G(V, E)$ is bipartite if you can partition V into two sets A, B such that all edges $e \in E$ have one vertex in A and one vertex in B.
- **Alternate Definition:** G is bipartite if we can color the vertices using two colors such that each edge is biochromatic (endpoints have different colors).
- V is the set of vertices.
- E is the set of edges.
- Edges can be directed or undirected.
- Conventions: n is the number of vertices while m is the number edges.

Testing Bipartiteness:

- Input: $G = V, E$.
- Output: YES, if G is bipartite. NO, if G is not bipartite.
- Conventions: An algorithm accepts G if it outputs YES. An algorithm rejects G if it outputs NO.

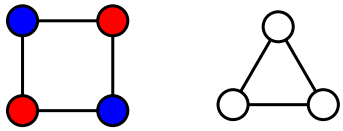
Algorithm:

```
TestBipartiteness(G)                                   // This is not the correct algorithm!
    Queue q
    Initialize Color[v] = NULL for all vertices v
    Pick arbitrary s ∈ V:

    // Visit s.
    Color[s] = blue
    q.enqueue(s)
    while(q not empty):
        u ← q.dequeue()
        for each neighbor v of u:
            c ← Color[v]
            if(c == NULL):
                Color[v] == opposite of Color[u]
                q.enqueue(v)
            Else if(Color[u] == Color[v]):
                return NO
    return YES
```

Potential Issue:

- We might not visit all nodes!



Solution:

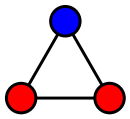
- After initializing, add the following code:

```
for all verticies v ∈ V:
    if Color[v] == NULL:
        // BFS from v.
```

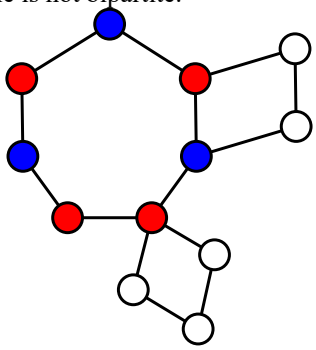
return YES if we haven't returned NO yet.

Not Bipartite Graphs:

- Example: triangles.



- **Fact:** Any odd-length cycle is not bipartite.
- **Fact:** Any graph containing an odd-length cycle is not bipartite.

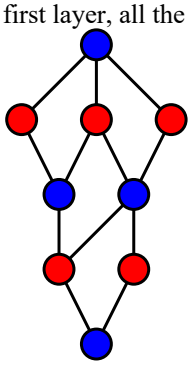


Lecture 11

2024-09-30
Week 5, Mon

Bipartite Graphs (Continued):

- Imagine we have a graph. We can pick up the graph by holding onto one node, and we would get a graph that's separated into layers, with all immediate neighbors of our node in the first layer, all the second immediate neighbors in the second layer, etc.



- **Theorem:** Let G be a connected graph and $L_0, L_1, L_2, \ldots$ be the layers of the BFS search tree:
  1. If there are no edges (u, v) with u, v on the same layer, then G is bipartite.
  2. If there is an edge (u, v) with u, v on the same layer, then G is not partite.

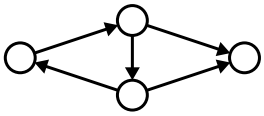
- **Def:** A node w is the Least Common Ancestor (LCA) for nodes u, v if:
  1. u, w and v, w are connected.
  2. w is visited before u, v.
  3. Among all such nodes, w is closest to u, v.

Proof of Claim 2:

Fix edges (u, v) with u, v on the same layer. Let w be the Least Common Ancestor (LCA) of u, v.
Note that $u \to\to w \to\to v \to\to u$ is an odd length cycle. Therefore, G is not bipartite.

Directed Graphs:

- All edges are directed.



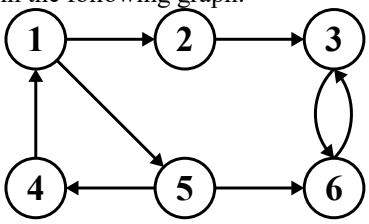
Graph Representation:

- Adjacency Matrix: $M[i, j] = 1$ if the edge $i \to j$ exists, 0 otherwise.
- Adjacency List: Two list with one for incoming edges and one for outgoing edges.

Search Algorithms (BFS/DFS):
Q: Can we search?
A: Yes.
- **Fact:** BFS(s) computes the set of vertices reachable from s. I.e., $\{t$: there is a path $s \to t\}$.
- **Def:** u, v are weakly connected if there is either a $u \to v$ path or a $v \to u$ path in G.
- **Def:** u, v are strongly connected if there is a $u \to v$ path and a $v \to u$ path in G.
- **Note:** The definition of weakly connected nodes uses a logical OR. All strongly connected nodes are also weakly connected.
- **Def:** C is a strongly connected component if u, v are strongly connected for all $u, v \in C$ and no other vertices are strongly connected to some $v \in C$.

Exercises:

1. Identify all SCC's (strongly connected cycles) in the following graph:



   $(1, 4, 5)$ and $(3, 6)$.

2. Given a directed graph G and a start vector s, design an $O(n + m)$ algorithm that identifies all SCC's containing s.
   Algorithm: Define G^{rev}: G but with all the edges reversed:
   1. Run BFS(s) to get all t such that there is a $s \to t$ path in G.
   2. Run BFS(s) on G^{rev} to get all t such that there is a $t \to s$ path in G.
   3. Return the intersection of 1 and 2.

Lecture 12

2024-10-02
Week 5, Wed

Cycles:

- The smallest undirected graph cycle has 3 vertices, while the smallest directed graph cycle has 2 vertices.
- **Def:** A connected graph without any cycle is called a tree.
- **Def:** A graph without any cycle but can be unconnected is a forest.
- **Def:** A directed graph without cycles is called a directed, acyclic graph (or a DAG). DAGs don't need to be connected.
- DAGs are useful as they can be used to analyze and avoid deadlocks in operating systems.

Topological Ordering:

- **Def:** A topological ordering in a DAG is an ordering of vertices $v_1, v_2, \ldots, v_n$ such that $i < j$ for all edges $(v_i, v_j) \in E$.
- **Fact:** If G has a topological ordering, then G is a DAG.
- **Theorem:** If G is a DAG, then G has a topological ordering.

The Topological Sorting (Topsort) Problem:

- Input: Directed acyclic graph $G = (V, E)$.
- Output: Topological ordering of vertices.

- Claim: If G is a DAG then there is $v \in V$ with no incoming edges.
- Proof (Sketch): (By Contradiction) Assume all vertices have ≥ 1 incomding edges. Pick any vertex. Walk back along incoming edges.

Algorithm:

```
List TopSort(DAG G) {
    List InCount[1, ..., n];        // Current in degree of vertex i.
    Queue InZero;                   // Queue of vertices safe to add to TopSort.
    List tSort;
```

```
    // Preprocess to build Incount[...], InZero[...].
    while(InZero not empty) {
        v = InZero.dequeue();
        tSort.insertAtTail(v);
        for(each neighbor u of v) {
            InCount(u) -= 1;
            if(InCount(u) == 0) {
                InZero.enqueue();
            }
        }
    }

    return tSort;
}
```

Lecture 13

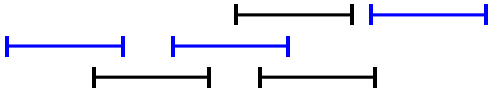
2024-10-04
Week 5, Fri

Greedy Algorithms:

- **Def:** A decision problem is a problem where output is binary (e.g., YES/NO).
- **Def:** An optimization problem is a problem where we want to maximize or minimize time, profit, # jobs, etc.

- Greedy Algorithms: An algorithm is greedy if it builds up a solution one step at a time, without backtracking.

The Interval Scheduling Problem:

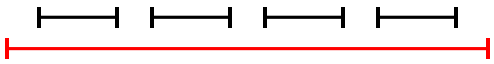


- Input: A list of n requests. Each request i has a start time $s(i)$ and a finish time $f(i)$.
- **Def:** Two requests are compatible if they don't overlap in time.
- Output: Largest possible set of compatible requests.

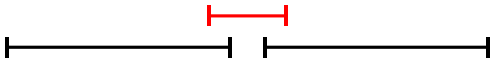
Greedy Strategy for Interval Scheduling:

- Build up requests one at a time, maintaining compatibility at all times.

1. Earliest Available Start: Pick interval i to minimize $S(i)$. — Does Not Work

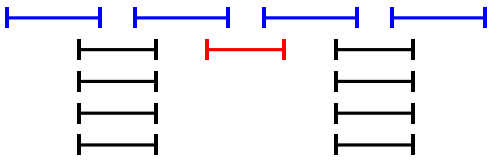


2. Shortest Interval: Pick i to minimize $f(i) - s(i)$. — Does Not Work



3. Earliest Available Finish: Pick i to minimize $f(i)$. — Works

4. Minimize # Conflicts: Pick i to minimize the number of j incompatible with i. — Does Not Work



Algorithm:

```
List EarliestAvailableFinish() {
    List R = set of requests;
    List A = [];
    while(R is not empty) {
        pick i in R that minimizes f(i);
        add i to A;
        remove i from R;
```

```
        remove all j from R that are incompatible with i;
    }
    return A;
}
```

Proving EAF is an Optimal Algorithm:
- Stay Ahead Method: Argue that at each step, our algorithm is making more progress than any other algorithm.
- Claim: Let $A = \{i_1, \dots, i_k\}$ be requests returned by EAF, sorted by finish time. Let $B = \{j_1, \dots, j_m\}$ be any other set of compatible requests, sorted by finish time. For all $r \leq k$, $f(i_r) \leq f(j_r)$.

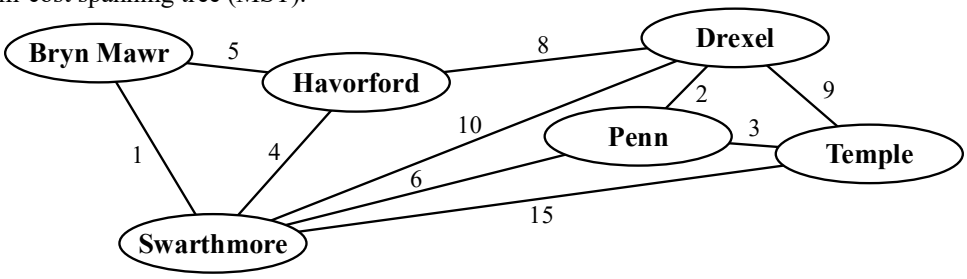
Lecture 14

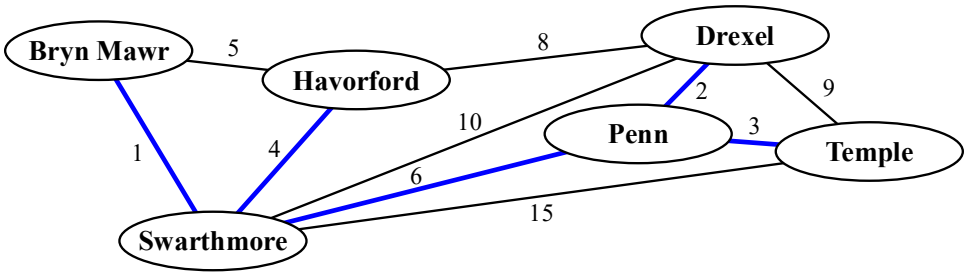
2024-10-07
Week 6, Mon

EAF Analysis (Stays Ahead Method):
- Let $A = \{i_1, \dots, i_k\}$ be the set of requests returned by EAF.
- Let $B = \{j_1, \dots, j_m\}$ be any other compatible set.
- Claim: For all $r \leq k$, $f(i_r) \leq f(j_r)$.
- Proof: We will prove this by induction on r.
  - *Base Case*: $r = 1$. Then $f(i_1) \leq f(j_1)$. By construction, we construct the earliest finish time.
  - *Induction Hypothesis*: Assume for all $L < r$ that $f(i_L) \leq f(j_L)$.
  - *Induction Step*: We want to show that $f(i_r) \leq f(j_r)$ as well.
  - We know that $s(j_r) > f(j_{r-1})$ since B is a compatible set. Then, $f(j_{r-1}) \geq f(i_{r-1})$ by the induction hypothesis.
  - $s(j_n) \geq f(i_{r-1})$ so j_r is compatible with $\{i_1, \dots, i_{r-1}\}$, but j_r is a compatible request with the EAF, so $f(i_r) \leq f(j_r)$.

- Theorem: EAF returns an optimal set.
- Proof: We will prove this by contradiction.
  - Suppose that $m > k$, i.e., $|B| > |A|$.
  - Look at j_{k+1}. We know that $s(j_{k+1}) > f(j_k) \geq f(i_k)$.
  - j_{k+1} is compatible with A, but then EAF wouldn't have terminated.

The Communication Network Problem:
- Build a communication network at the minimum cost.
- **Def:** A spanning tree of a graph $G = (V, E)$ is a subset of edges $T \subseteq E$ such that $G^1 = (V, E)$ is connected and has no cycles.
- Cost of spanning time: $\text{Cost}(T) = \sum_{e \in T} \text{Cost}(e)$.
- Problem: MinimumSpanningTree (MST).
- Input: Graph $G = (V, E)$ with edge costs $\{c_e : e \in E\}$. (Assume edge costs are distinct.)
- Output: Minimum-cost spanning tree (MST).





Kruskal's Algorithm (Kruskal '56):

```
List KruskalsAlgorithm() {
```

```
    T = [];
    repeat until |T| = n - 1 {
        add cheapest edge to T that does not create a cycle;
    }
    return T;
}
```

Prim's Algorithm (Prim '57, Jarník '30, Dijkkstra '56):

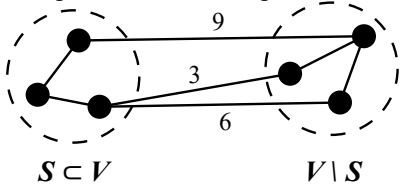
```
List PrimsAlgorithm() {
    start at s;
    S = [s];
    T = [];
    repeat until |T| = n - 1 {
        pick cheapest edge (u, v) with u in S, v not in S;
        add (u, v) to T;
        add v to S;
    }
    return T;
}
```

Lecture 15

2024-10-09
Week 6, Wed

Cut Property:

- Let $S \subset V$ be any nonempty proper subset of the vertices.
- Let $e = (u, v)$ be the cheapest edge with one endpoint in S, one endpoint not in S. Then e must be in the MST.



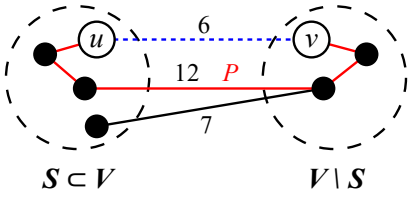
Proving Prim's Algorithm:

- Claim: Prim's algorithm returns an MST.
- Proof:
  - At each iteration, we have $S = \{\text{vertices we're connected so far}\}$.
  - Cut property says that the cheapest edge out of S is in MST. Note that this is the exactly the edge we add.
  - This is true for every edge we add in Prim's algorithm.
  - Therefore, Prim's algorithm returns an MST.

Proving Kruskal's Algorithm:

- Claim: Kruskal's algorithm returns an MST.
- Proof:
  - At each iteration, look at edge (u, v) that we add to T.
  - $S = \{\text{vertices connected to } u \text{ given edges currently in } T\}$.
  - By the cut property, the cheapest edge between $S, V \backslash S$ must be in MST.
  - We can say this about any edge we add.
  - Therefore, T is an MST.

Proving the Cut Property:



- Proof by Contradiction:
  - Fix $S \subset V$. Let (u, v) be the cheapest edge while $u \in S$, $v \notin S$.
  - Let T' be a spanning tree that doesn't contain the edge (u, v).

- Since T' is a spanning tree, so all vertices are connected. Let P be a $u \rightsquigarrow v$ path in T'.
- Notice that $P \cup \{(u, v)\}$ is a cycle. There must be some edge $e' \in P$ that crosses cut (crosses S and $V \backslash S$).
- Swap e' for (u, v), and we get a new, cheaper spanning tree.

Lecture 16

2024-10-11
Week 6, Fri

Q: Among all closed shapes with perimeter P, what is the maximum area A?

A: $A \leq \frac{P^2}{4\pi}$. The largest shape possible is a circle.

- We can prove this answer using exchange arguments.

Exchange Argument:

- Motivation: Solutions can be hard to compare, but comparing closer solutions might be manageable.

1. Say that our solution is A.
2. Start with any solution B.
3. Make small changes to get B' without making things worse.
4. Repeat this process.
5. Arrive at our solution A.
6. Conclude that this solution is optimal.

The Interval Scheduling with Deadlines Problem:

- Input: List of n jobs where each job i has a time required to complete t_i and a deadline d_i.
- Output: A schedule of when to complete all jobs that *minimizes maximum lateness*.
- Note: All jobs must be completed.
- Notation: For any schedule A, let $s(i)$, $f(i)$ be the start and finish times of job i in schedule A.
- Def: The lateness of job i, l_i, is defined as $f(i) - d_i$ if $f(i) > d_i$. l_i is 0 otherwise.
- Goal: Come up with schedule A that minimizes $L(A) = \max_i l_i$.

Possible Greedy Algorithms:

1. Earliest Deadline: $\min_i d_i$. Works
2. Earliest Completion Time: $\min_i t_i$. Does Not Work

   Counterexample: $t_1 = 1, d_1 = 100, t_2 = 20, d_2 = 20$. We will pick the first job first which actually results in a larger maximum lateness.

3. Minimize Slack Time: $\min_i (d_i - t_i)$. Does Not Work

   Counterexample: $t_1 = 1, d_1 = 4, t_2 = 5, d_2 = 6$. We will pick the second job first which actually results in a larger maximum lateness.

- Theorem: Earliest Deadline returns the optimal schedule.
  - Def: Idle time exists when there are jobs left to complete but nothing is being done at the moment.
  - Fact: There is an optimal schedule with no idle time.
  - Def: An inversion in a schedule is a pair of jobs i, j such that $d_i < d_j$ but i is scheduled after j.
  - Fact: Earliest Deadline returns schedule A with no inversions.
  - Def: i, j is an adjacent inversion if $d_i < d_j$ but i is scheduled immediately after j.
  - Claim: If schedule B has an inversion, then it has an adjacent inversion.
- Proof (Via Exchange Argument):
  - Let B be any schedule that has an inversion. Then B has an adjacent inversion (i, j).
  - Let B' be B, but with jobs i, j swapped.
  - Note: B' now has one less inversion compared to B.
  - Claim: $L(B') \leq L(B)$.

Lecture 17

2024-10-21
Week 7, Mon

Q: What is a greedy sorting algorithm?

A: Selection sort. In each step, we take the next smallest element, and we put it into place.

Divide and Conquer:

Merge Sort:

```
Array MergeSort(Array A) {
    Divide A into 2 halves L, R;
    MergeSort(L);
    MergeSort(R);
    Merge(L, R);
}
```

- Divide and Conquer Algorithms:
  1. Divide input into two (or more) pieces of equal size.
  2. Solve each subproblem.
  3. Combine results into overall solution.

Runtime of Merge Sort:

- Let $T(n)$ denote the time needed to merge sort a list of n elements. Then,
- $T(n) = 2T(n/2) + cn$ for $n > 2$ and some constant c. $T(2) = c$.
- Techniques for solving recurrence relations:
  - Substitution method.
  - Recursion trees.

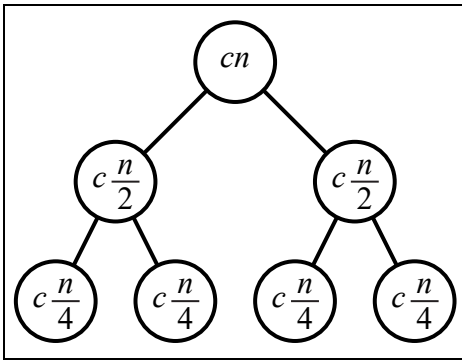
Substitution Method:

1. Guess solution: $cn \log n$.
2. Substitute into equation: $T(n) \leq cn \log n$.
3. Try to prove by induction.
4. Repeat steps 1-3 until we get the correct solution.

- Claim: $T(n) \leq cn \log n$.
- Proof: We will attempt to prove this by induction on n.
  - *Base Case*: $n = 2$. $cn \log n = c \cdot 2 \cdot \log 2 = 2c$. Notice $T(2) = c \leq 2c$. The base case is correct.
  - *Induction Hypothesis*: Assume $T(m) \leq cm \log m$ for all $m < n$.
  - *Induction Step*: We want to show $T(m) \leq cm \log m$ for $m = n$ as well.
  - From the recurrence relation, we know that $T(n) = 2T(n/2) + cn$. We can substitute in $T(n/2)$ using our induction hypothesis where $m = n/2$. We get $T(n) \leq 2\left[c\frac{n}{2}\log\frac{n}{2}\right] + cn$. We now have $T(n) \leq cn \log\frac{n}{2} + cn = cn(\log n - \log 2) + cn = cn \log n - cn + cn = cn \log n$. Note that $\log 2 = 1$. ∎

Recursion Trees (Unrolling the Recursion):

1. Analyze the first few levels.
2. Try to identify the general pattern.
3. Look at the amount of work at the last level.
4. Sum up all the work over all levels.



Level	Problem Size	# of Nodes	Work Per Node	Total Work
0	n	1	cn	cn
1	$n/2$	2	$c\frac{n}{2}$	cn
2	$n/4$	4	$c\frac{n}{4}$	cn
⋮				
j	$\frac{n}{2^j}$	2^j	$c\frac{n}{2^j}$	cn
k (Leaf)	$\frac{n}{2^k} = 2$	2^k	c	$2^k \cdot c = \frac{cn}{2}$

- Notice that from the final problem size, we have $\frac{n}{2^k} = 2$ and $k = \log\frac{n}{2} = \log(n) - 1$.
- The total amount of work is $T(n) = c\frac{n}{2} + \sum_{i=0}^{k-1} cn = c\frac{n}{2} + cnk = c\frac{n}{2} + cn[\log(n) - 1]$.
- Therefore, $T(n) = O(n \log n)$.

Recurrence Relations:

Intuition:

- The runtime for any recurrence relation depends on 3 factors:

$$M(n) = 3M(n/2) + 2n$$
$$M(1) = 1$$

1. The branching factor (# of recursive calls).
2. The change in problem size.
3. The work per node.

Lecture 18

2024-10-23
Week 7, Wed

Similarity Ranking:

- Ranking: A sequence of n distinct numbers $a = (a_1, \dots, a_n)$. For example, movie 1 is my $a_1{}^{\text{th}}$ favorite, movie 2 is my $a_2{}^{\text{th}}$ favorite.
- Inversion: Given two rankings $a = (a_1, \dots, a_n)$ and $b = (b_1, \dots, b_n)$. An inversion is a pair i, j such that $a_i < a_j$ but $b_i > b_j$.

Inversion Counting Problem:

- Counting the number inversions.
- Simple version: Assume b sorted. For example, $b = (1,2,3, \dots, n)$.
- Input: Ranking $a = (a_1, \dots, a_n)$.
- Output: The number of inversions that belong to a and $(1, \dots, n)$. I.e., the number of (i, j) pairs such that $i < j$ and $a_i > a_j$.

Exercise:

- Design $O(n^2)$ algorithm for ranking similarity. Produce pseudocode.

Brute-Force Algorithm:

```
int BruteForceCount(Array a) {
    int count = 0;
    for(i in range(1, n)) {
        for(j in range(i + 1, n + 1)) {
            if(a_i > a_j) {
                count += 1;
            }
        }
    }
    return count;
}
```

Divide and Conquer Algorithm:

- Divide a into 2 halves.
- Count the number of inversions in the left half.
- Count the number of inversions in the right half.
- Count the number of inversions between halves.

```
Tuple MergeCount(Array L, Array R) {
    count = 0;
    i, l, r = 0;
    s = L.size();
    while(L, R not exhausted) {
        if(L[l] < R[r]) {
            A[i] = L[l];
            l += 1;
        }
        else {
            A[i] = R[r];
            r += 1;
            count += (S - l);
        }
        i += 1;
    }
    // Add in remaining items from subarray.
```

```
    return (A, count);
}

Tuple SortCount(Array A) {
    // Return sorted list.
    // Also return the number of inversions in A.
    if(A.size() <= 1) {
        return (A, 0);
    }
    else {
        Divide A into 2 halves L, R;
        (L, C_L) = SortCount(L);
        (R, C_R) = SortCount(R);
        (A, C) = MergeCount(L, R);
        return (A, C_L + C_R + C);
    }
}
```

Q: What is the runtime?
A: $T(n) = 2T(n/2) + cn, T(1) = c$. $T(n) = O(n \log n)$.

Example:
- $a = (4, 7, 8, 3, 5, 1, 6, 2)$.
- Number of inversions involving the right-hand side elements:
- 5:2, 6:2, 1:4, 2:4.

Lecture 19

2024-10-25
Week 7, Fri

The Integer Multiplication Problem:

- Inputs: 2 n-digit, base-N numbers $a = a_{n-1}a_{n-2} \dots a_1a_0$ and $b = b_{n-1}b_{n-2} \dots b_1b_0$.
- Output: $a \times b$.

Elementary School Multiplication:

```
int ESMultiplication(int a, int b) {
    // Precompute N * N timestable.
    int sum = 0;
    for(i in range(0, n)) {
        for(j in range(0, n)) {
            sum += b_i * a_j * N^(i + j);
        }
    }
    return sum;
}
```

- Operations involved:
  - n-digit addition: $O(n)$.
  - Single-digit multiplication (e.g., $b_i \times a_j$): $O(1)$.
  - N-shifting: $O(1)$.
- Traditionally, multiplication is $O(n^2)$ since we need to multiply every single digit of a with every single digit of b (and moving and adding zeros appropriately).

Divide and Conquer for Multiplication:

- $a = a_L \cdot N^{\frac{n}{2}} + a_R$.
- $b = b_L \cdot N^{\frac{n}{2}} + b_R$.
- $a \cdot b = \left(a_L \cdot N^{\frac{n}{2}} + a_R\right)\left(b_L \cdot N^{\frac{n}{2}} + b_R\right) = a_Lb_LN^n + (a_Lb_R + a_Rb_L)N^{\frac{n}{2}} + a_Rb_R$.
- For this algorithm, we split into 4 different multiplications each step. Therefore, $T(n) = 4T\left(\frac{n}{2}\right) + cn, T(1) = c$.
- Solving this recurrence relation, we get $T(n) = \Theta(n^2)$.
- Why didn't we get an improvement? We have too many (4) recursive calls in each step.

- Observe $(a_L + a_R)(b_L + b_R) = a_L b_L + a_L b_R + a_R b_L + a_R b_R$. This contains the exact 4 terms we need in one multiplication, so we can subtract $a_L b_L$ and $a_R b_R$ from $(a_L + a_R)(b_L + b_R)$. This results in getting $a_L b_R + a_R b_L$ and all terms needed in 3 instead of 4 multiplications.

Karatsuba Multiplication:
- $A = (a_L + a_R)(b_L + b_R)$.
- $B = a_L b_L$.
- $C = a_R b_R$.
- $D = A - B - C$ (Note that this is $a_L b_R + a_R b_L$).

```
int Karatsuba(int a, int b) {
    int A = Karatsuba(a_L + a_R, b_L + b_R);
    int B = Karatsuba(a_L, b_L);
    int C = Karatsuba(a_R, b_R);
    return (B * N^n + (A - B * C) * N^(n / 2) + C);
}
```

- Overall runtime; $K(n) = 3K\left(\frac{n}{2}\right) + cn$, $K(1) = c$.

Partial Substitution for Karatsuba Multiplication:
- Guess: $K(n) \leq a \cdot n^k + bn$ for constants a, b, and k to be determined.
- Proof: We will attempt to prove this by induction on n.
  - *Base Case*: For $n = 1$, $K(1) = c$ and $a \cdot n^k + bn = a + b$. The base case holds as long as $c \leq a + b$.
  - *Induction Hypothesis*: Assume for all $m < n$ that $K(m) \leq am^k + bm$.
  - *Induction Step*: $K(n) = 3K\left(\frac{n}{2}\right) + cn \leq 3\left[a\left(\frac{n}{2}\right)^k + b\frac{n}{2}\right] + cn = \frac{3an^k}{2^k} + \left(\frac{3b}{2} + c\right)n \leq an^k + bn$.
  - Notice that this last inequality holds if $\frac{3an^k}{2^k} \leq an^k$ and $\left(\frac{3b}{2} + c\right)n \leq bn$.
  - For $\frac{3an^k}{2^k} \leq an^k$, we have $k \geq \log_2 3$. So we set $k = \log_2 3 \approx 1.585$.
  - For $\left(\frac{3b}{2} + c\right)n = bn$, we have $b = -2c$.

Lecture 20

2024-10-28
Week 8, Mon

The Steel Rod Problem:
- Inputs: n representing the length of rod and a list P of prices where $P[i]$ represents the revenue from selling an i feet rod.
- Output: Maximum possible revenue from chopping n feet rod into pieces, selling the pieces.

Greedy Algorithms for Steel Rod:
1. Cut rod to get price that maximize revenue, then repeat with the remaining rod. Does Not Work
   Counterexample: $n = 10$, $P[1] = 1$, $P[2] = 4$, $P[5] = 5$, $P[k] = 0$ for all other k. The greedy algorithm outputs 2 of $P[5]$, equaling 10, while the optimal is 5 of $P[2]$, equaling 20.

2. Cut rod to get price with maximum revenue per foot, then repeat with the remaining rod. Does Not Work
   Counterexample: $n = 10$, $P[1] = 1$, $P[5] = 10$, $P[6] = 13$, $P[k] = 0$ for all other k. The greedy algorithm outputs 1 of $P[6]$ and 4 of $P[1]$ equaling 16, while the optimal is 2 of $P[5]$ equaling 20.

Divide and Conquer Algorithm for Steel Rod:
1. Cut the rod in half, recurse so that $R[n] = ? \cdot 2R\left[\frac{n}{2}\right]$. Does Not Work
   Counterexample: $n = 10$, $P[7] = 9$, $P[10] = 10$, $P[k] = k$ for all other k. The greedy algorithm outputs $P[1] \cdot n = 10$, while the optimal is $P[7] + P[3] = 12$. This algorithm just doesn't make sense.

Brute Force Algorithm for Steel Rod:
1. Try all the possible ways to cut a rod. Then calculate the revenue for each method. Works But Inefficient
   Problem: There are an exponential number of possible solutions. For each rod (assuming integer cuts), we have $n - 1$ places to cut, which leads to 2^{n-1} possible solutions to go over.

Dynamic Programming:
- Solve problem by identifying a collection of subproblems, solving subproblems one-by-one, using solutions to smaller subproblems to understand the larger ones.

Dynamic Programming Design Process:
1. Characterize the structure of the optimal solution.
2. Recursively define the value of the optimal solution.
3. Compute the value of the optimal solution.
4. (If needed,) construct the optimal solution from the computed information.

Dynamic Programming for the Steel Rod Problem:
1. Let r_n be the maximum revenue we can get from an n-feet rod. If it is optimal to cut the rod at k feet, then $r_n = r_k + r_{n-k}$. (*Alternative Characterization*) If the leftmost cut in the optimal solution is at k feet, then $r_n = P[k] + r_{n-k}$.
2. $r_n = \max_{1 \le i \le n} P[i] + r_{n-i}$. Define $r_0 = 0$.
3. See the pseudocode below.

```
int SteelRod(int n, list P) {
    if(n == 0) {
        return 0;
    }
    int q = -1;
    for(int i = 1; i <= n; i++) {
        q = max(q, P[i] + SteelRod(n - i, P));
    }
    return q;
}
```

Q: What is the runtime?
A: $T(0) = 1$, $T(n) = \sum_{i=1}^{n} T(n-1) + cn = cn + \sum_{k=0}^{n+1} T(k)$. In the end, $T(n) = O(2^n)$.

Lecture 21

2024-10-30
Week 8, Wed

The RNA Substructure Problem:

RNA Molecule:
- A string of bases $B = \{b_1, \dots, b_n\}$, where each $b_i \in \{\text{A}, \text{C}, \text{G}, \text{U}\}$.

Secondary Structure on B:
- A set of pairs $S = \{(i, j)\}$ such that:
  1. (No sharp turns.) Each pair $(i, j) \in S$ has $|i - j| > 4$.
  2. Elements of any pair must be A–U, C–G.
  3. S must be a matching.
  4. (No crossing.) If (i, j) and $(k, l) \in S$, we can't have $i < k < j < l$.

- Big Question: What secondary structure is most likely to form.
- Big Answer: The one that minimizes total free energy. (CS Translation: Maximize the size of matching $|S|$.)

The Problem:
- Inputs: The RNA molecule $B = (b_1, \dots, b_n)$.
- Output: ① The size of the largest possible secondary structure S and ② S itself.

Dynamic Programming Step 1:
- Characterize the structure of the optimal solution.
- OPT(j) = the maximum number of pairs in string $b_1, \dots, b_j$.
- Choice: Does b_j get paired? If so, with what?
  - b_j not paired: the score is OPT($j - 1$).
  - b_j paired: say b_j is paired with b_i, the score is 1 for (i, j) + OPT($i - 1$) for $b_1, \dots, b_{i-1}$ + ??? for $b_{i+1}, \dots, b_{j-1}$.
  - Notice that we don't have a good way to characterize the score for $b_{i+1}, \dots, b_{j-1}$. Therefore, this is not the right structure.

Lecture 22

2024-11-01
Week 8, Fri

The RNA Substructure Problem (Continued):

Dynamic Programming Step 1, Attempt 2:

- Let OPT(i, j) be the size of the best matching for substring $b_i, \ldots, b_j$.
- In the optimal matching, b_j is either going to get matched or not.
- If b_j is not matched, then OPT$(i, j) =$ OPT$(i, j-1)$.
- If b_j is matched to b_t, then OPT$(i, j) = 1 +$ OPT$(i, t-1) +$ OPT$(t+1, j-1)$.

Dynamic Programming Step 2:

- Define a 2D table $T[0 \cdots n, 0 \cdots n]$ where $T[i, j]$ stores OPT(i, j).
- Note: We want to output $T[1, n]$.
- General Case: $T[i, j] = \max\left(T[i, j-1], \max_t(1 + T[i, t-1] + T[t+1, j-1])\right)$.
- The $\max_t \ldots$ term means that to take over all the t's that are valid matches for j (valid base-pairing and $|t - j| > 4$).

Memorization:

- Store results of previous function calls.
- Return cached results if some function call is called again.

Dynamic Programming Step 3:

```
List RNA(List B) {
    Initialize T[i, j] = -1 for all i, j;
    return RNAH(B, T, 1, n);
}

List RNAH(B, T, i, j) {
    if(|i - j| <= 4) {
        return 0;
    }
    if(T[i, j] >= 0) {
        return T[i, j];
    }

    // Computing T[i, j].
    q = RNAH(B, T, i, j - 1);
    for(t = i, ..., j - 5) {
        if(B[t], B[j] are valid matches) {
            temp = 1 + RNAH(B, T, i, t - 1) + RNAH(B, T, t + 1, j - 1);
            if(temp > q) {
                q = temp;
            }
        }
    }
    T[i, j] = q;
    return T[i, j];
}
```

Example:

- In the case of RNAH$(B, T, 1, 100)$:
- $T[1, 100] = \max(T[1, 99], 1 + T[1, 0] + T[2, 99], 1 + T[1, i] + T[3, 99], 1 + T[1, 2] + T[4, 99], \ldots, 1 + T[1, t-1] + T[t+1, 99])$.
- Notice that in the equation above, we are assuming that b_j can be matched with $b_0, b_1, b_2, \ldots, b_t$.

Lecture 23

2024-11-04
Week 9, Mon

Memorization:

- "Top-down" dynamic programming implementation.
- Recursively compute problems when needed.
- Cache stored solutions to subproblems so they don't need to be recomputed.

Tabulation:

- "Bottom-up" dynamic programming implementation.
- Compute the smallest subproblems first, then using them to compute larger subproblems.

The RNA Substructure Problem (Continued):

Tabulation for the RNA Substructure Problem:

- Let $T[0 \cdots n][0 \cdots n]$ be a 2D table with all entries initialized to 0.

```
List RNA(B) {
    Initialize T[i, j] = 0 for all i, j;
    for(k = 5, ..., n) {
        for(i = 1, ..., n - k) {
            j = i + k;

            // Computing T[i, j].
            q = T[i, j - 1]
            for(t = 1, ..., j - 4) {
                if MATCH(B[t], B[j]) {
                    score = 1 + T[i, t - 1] + T[t + 1, j - 1];
                    if(score > q) {
                        q = score;
                    }
                }
            }
            T[i, j] = q;
        }
    }
    return T[1, n];
}
```

- Notice that we should check for whether or not the indices are valid when we index previous table entries.

Memorization vs. Tabulation:

- Memorization: For problems where the numbers needed in the table entries is less than the total number. If so, memorization is much faster as we don't need to compute every possible entry through recursion.
- Tabulation: When most or all of the subproblems are needed, tabulation is usually constant-factor faster. Why? Recursion calls need to set up and manage a new stack frame which is not needed for iteration.

Generalization Dynamic Programming Implementation Details:

- We can use an array or dictionary for tables.
- We have a surprisingly large $O(1)$-factor overhead for dictionary.
- Watch out for edge cases (e.g., negative table indices).
- Design: Often the problem constraints can be handled at the expense of having extra dimensions in the table. The table can store int/float, but also bool. We can also perform postprocessing when needed.

Dynamic Programming Step 4:

- For the RNA substructure problem, our pseudocode returns the size of the optimal matching, not the matching itself. We have two ways of constructing the solution.
- Create a second table S. We now have two options:

1. Each entry in S stores a partial solution to that subproblem (e.g., $S[i, j]$ stores the optimal matching for $b_i, \ldots, b_j$ for the RNA substructure problem).
2. Store in $S[i, j]$ the choice which gives us the optimal solution. Perform postprocessing to recover the optimal solution.

Example for Option 2:

```
List RNA(B) {
    Initialize T[i, j] = 0 for all i, j;
    Let S[0 ... n][0 ... n] be a 2D table storing what j should be matched with in the optimal matching.
    for(k = 5, ..., n) {
        for(i = 1, ..., n - k) {
            j = i + k;

            // Computing T[i, j].
            q = T[i, j - 1]
            for(t = 1, ..., j - 4) {
```

```
                if MATCH(B[t], B[j]) {
                    score = 1 + T[i, t - 1] + T[t + 1, j - 1];
                    if(score > q) {
                        q = score;
                        S[i, j] = t; // The best matching for B[i] ... B[j] that matches (t, j).
                    }
                }
            }
            T[i, j] = q;
        }
    }
    return T[1, n];
}
```

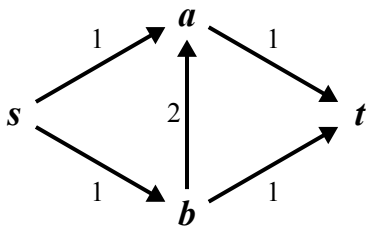
- Then perform postprocessing where we make recursive calls to rebuild the matching.

Lecture 24

2024-11-06
Week 9, Wed

Network Flow:

- Say $G = (V, E)$ is a directed graph where each edge $e \in E$ has an integer capacity c_e.
- There are two vertices $s, t \in V$.
- Think of the integer capacity as the size of the water pipes (edges).
- A flow is a function $f: E \to \mathbb{R}^+$, describing how much flow is on each edge such that:
  1. Flow is conserved. For all $v \in V$ (v not equal to s or t), $f^{\text{out}}(v) = \sum_{e \text{ out of } v} f(e)$ must equal $f^{\text{in}}(v) = \sum_{e \text{ into } v} f(e)$.
  2. Capacity is respected. For all $e \in E$, must have $0 \le f(e) \le c_e$.
- Def: The value of a flow f is $V(f) := f^{\text{out}}(s)$.



The Network Flow Problem:

- Inputs:
  - A directed graph $G = (V, E)$.
  - Edge capacities $\{c_e\}$.
  - $s, t \in V$.
- Outputs: The maximum flow f (or the value of $V(f)$).
- **Q:** Does $f^{\text{out}}(s) = f^{\text{in}}(t)$?
- **A:** Yes, because flow is conserved.

Residual Graphs:

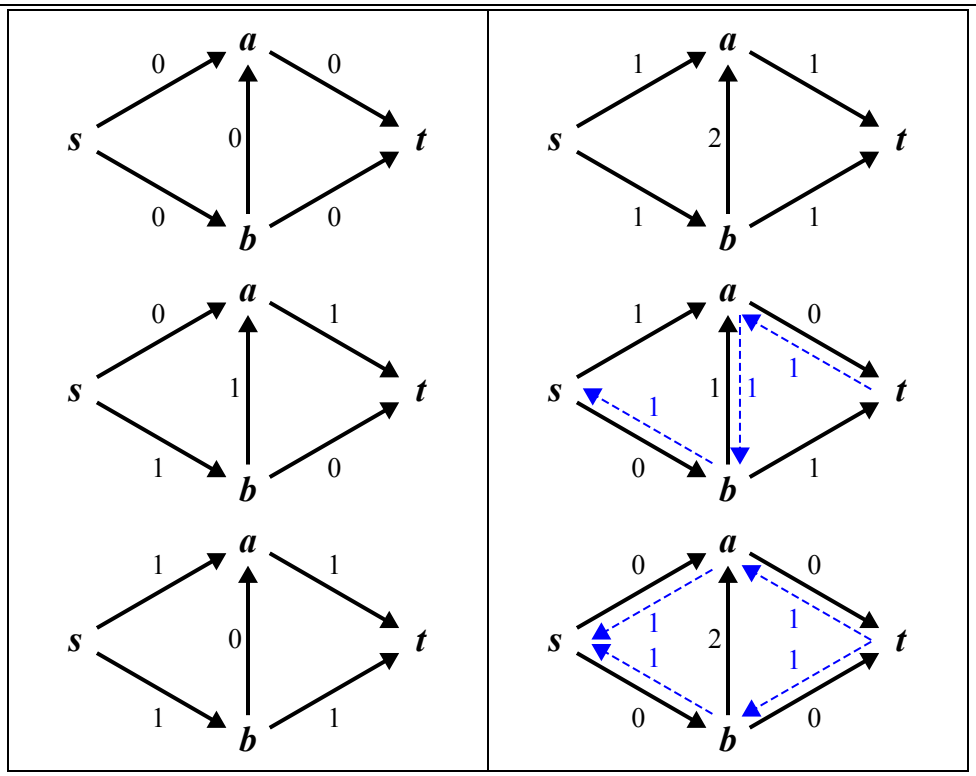
- $V' = V$.
- For each edge $e \in E$, add an edge e to E' with capacity $\boldsymbol{c'_e = c_e - f(e)}$ ("forward edge").
- For each edge e with $f_e > 0$, add a reverse edge e^{rev} to E' with the capacity $\boldsymbol{c_{e^{\text{rev}}} = f(e)}$ ("backwards edge").
- Def: Given $G = (V, E)$ and a flow f, the residual graph is agraph $G_f = (V_f, E_f)$ where $V_f = V$ and with 2 kinds of edges:
  - Forward edges: For any $e \in E$, add $e \in E_f$ with $c_e^f := c_e - f(e)$.
  - Backward edges: For any $e \in E$ with $f(e) > 0$, add $e^{\text{rev}} \in E_f$ with $c_{e^{\text{rev}}} = f(e)$.

Lecture 25

2024-11-08
Week 9, Fri

Residual Graph Examples:

Flow Graph	Residual Graph



The Ford-Fulkerson Algorithm:

```
function FordFulkerson(Graph G) {
    Initialize f(e) = 0 for all edges;
    Initialize residual graph G_f;
    while(there is augmenting s -> t path P in the residual graph) {
        f' = augment(f, P);
        f = f';
        Recompute the residual graph G_f with new flow;
    }
    return f;
}

int bottleNeck(function f, Path P) {
    return minimum edge capacity of edge in P in G_f;
}

function augment(function f, Path P) {
    b = bottleNeck(f, P);
    for(edge e = (u, v) in P) {
        if((u, v) is a forward edge):
            f((u, v)) += b;
        if((u, v) is a backward edge):
            f((v, u)) -= b;
    }
    return f;
}
```

Questions:

- Does FF halt?
- Does FF run in polynomial time?
- Does FF return a valid flow?
- Is the returned flow maximal?

Halting Proof:

- Claim: FF algorithm halts.
- Initial flow f has value $v(f) = 0$.

- Fact: For any flow f, $v(f) \leq C == \sum_{e \text{ out of } s} c_e$.
- Every iteration of FF, we find augmenting path, i.e., $s \rightsquigarrow t$ in residual graph where every edge has nonzero capacity.
- Value of flow increases in bottleneck > 0.

Runtime Analysis:
- For any flow f, creating residual graph can be done in $O(n+m)$ time.
- Use BFS to find augmenting paths can be done in $O(n+m)$ time.
- Other work inside the while loop is $O(n+m)$ for $\leq C$ iterations.
- Therefore, the overall runtime is $O\big(C(n+m)\big)$.

Lecture 26

2024-11-11
Week 10, Mon

The Ford-Fulkerson Algorithm (Continued):
- Claim: The Ford-Fulkerson algorithm returns a valid flow.
- Proof Sketch:
  - Claim: For all $i \geq 0$, after i iterations of while loop, we have a valid flow.
  - Proof: By induction on i.
    - *Base Case*: Our flow is zero everywhere, so this is definitely a valid flow.
    - *Induction Hypothesis*: Assume claim holds for all $j < i$.
    - *Induction Step*: We want to show that this holds for $j = i$ as well. By the induction hypothesis, we start with a valid flow f. Say $P =$ an augmenting path. Look at any vertex v on the augmenting path:
      - *Case 1*: We have 2 forward edges. In this case, we can increase flow into v and out of v.
      - *Case 2*: We have a forward edge and a backward edge. In this case, increasing flow on the forward edge causes the exact decrease on the backward edge.
      - *Case 3 and Case 4* are similar.

$S-T$ Cut:
- Def: An $s-t$ cut is a partition of V into two sets A, B such that $s \in A, t \in B$.
- The capacity of cut(A, B) is $c(A, B) = \sum_{\text{edges } (u,v) \text{ with } u \in A, v \in B} c_{(u,v)}$.
- Lemma 1: For any $s-t$ cut (A, B) and any flow f, $v(f) = f^{\text{out}}(A) - f^{\text{in}}(A)$.
- Lemma 2: For any flow f and any $s-t$ cut (A, B), $v(f) \leq c(A, B)$.
- Theorem: The Ford-Fulkerson algorithm returns a maximum flow.
- Proof Sketch: Cleverly choose $s-t$ cut (A, B) so that for flow f returned by the Ford-Fulkerson algorithm, $v(f) = c(A, B)$.

Proof of Lemma 1:
- Suppose $A = \{s, v_1, \dots, v_k\}$.
- $v(f) = \sum_{e \text{ out of } s} f(e) = f^{\text{out}}(s) = f^{\text{out}}(s) - f^{\text{in}}(s) = f^{\text{out}}(s) - f^{\text{in}}(s) + f^{\text{out}}(v_1) - f^{\text{in}}(v_1) + f^{\text{out}}(v_2) - f^{\text{in}}(v_2) + \cdots$.
- Notice that the last equality is true since $f^{\text{out}}(v_1) - f^{\text{in}}(v_1) = f^{\text{out}}(v_2) - f^{\text{in}}(v_2) = \cdots = 0$ since flow is conserved.
- $v(f) = \sum_{v \in A} f^{\text{out}}(v) - \sum_{v \in A} f^{\text{in}}(v) = \sum_{e \text{ out of } A} f(e) - \sum_{e \text{ into } A} f(e) = f^{\text{out}}(A) - f^{\text{in}}(A)$. ∎

Proof of Lemma 2:
- By Lemma 1, we have $v(f) = f^{\text{out}}(A) - f^{\text{in}}(A) \leq f^{\text{out}}(A) = \sum_{e \text{ out of } A} f(e) \leq \sum_{e \text{ out of } A} c(e) = c(A, B)$ ∎

Proof of $S-T$ Cut:
- Let f be the flow returned by the Ford-Fulkerson algorithm.
- Let $A = \{$vertices v such that there is an augmenting $s \rightsquigarrow v$ path in $G_f\}$.
- Let $B = \{$vertices v such that there is no augmenting $s \rightsquigarrow v$ paths$\}$.
- Claim: $v(f) = c(A, B)$.
- Notice that any forward edge from A to B in G_f has capacity 0 in G_f. Therefore, in the flow graph, $f(e) = c_e$.
- In other words, any backwards edge (u, v) in G_f with $u \in A$, $v \in B$ has capacity 0 in G_f. Thus, there is no $v \to u$ flow in the flow graph and $f\big((v, u)\big) = 0$.
- $v(f) = f^{\text{out}}(A) - f^{\text{in}}(A) = \sum_{e \text{ out of } A} f(e) - \sum_{e \text{ into } A} f(e)$.
- Notice that $\sum_{e \text{ into } A} f(e) = 0$.
- $v(f) = \sum_{e \text{ out of } A} c_e = c(A, B)$. ∎

Lecture 27

2024-11-13
Week 10, Wed

Computational Complexity Theory:

- The classification of problems occurring to their inherent difficulty.
- P: The set of all decision problems solvable in polynomial time.
- Def: A is polynomial-time reducible to B (write $A \leq_p B$) if arbitrary instances of problem A can be solved using a polynomial amount of time, plus a polynomial number of solutions to B.
- Note: "$A \leq_p B$" means that "A is polynomial-time reducible to B" or "reduce from problem A to problem B".
- Consequences of $A \leq_p B$:
  1. If B can be solved in polynomial time, then so can A.
  2. If A cannot be solved in polynomial time, then neither can B.

Some "Hard" Problems:

- Hard in quotes as we don't have rigorous proofs for an inherent lack of polynomial-time solutions for these problems.

1. Satisfiability (SAT):
   - Input: n boolean variables $x_1, \ldots, x_n$; m clauses $c_1, \ldots, c_n$ where each clause c_i is OR of some variables or their negation.
   - Output: YES if there is a satisfying assignment to $x_1, \ldots, x_n$. (I.e., if we can assign values so that every clause evaluates to TRUE.)
2. 3-Color (Tripartiteness):
   - Input: Graph $G = (V, E)$.
   - Output: YES if we can color vertices using one of 3 colors so that for each edge, endpoints are different colors.

- Def: An independent set is a set of vertices $S \subseteq V$ such that for all $i, j \in S$, $(i, j) \notin E$.
- Def: A vertex cover of a graph $G = (V, E)$ is a subset of vertices $T \subseteq V$ such that for all $(i, j) \in E$, either $i \in T$ or $j \in T$ or both (every single edge has an endpoint in T).

3. Independent Set (IS):
   - Input: Graph $G = (V, E)$, integer k.
   - Output: YES if and only if G has an independent set of at least k vertices.
4. Vertex Cover (VC):
   - Input: Graph $G = (V, E)$, integer k.
   - Output: YES if and only if G has an vertex cover of size $\leq k$.

Theorem:

1. IS $\leq_p$ VC.
2. VC $\leq_p$ IS.

Lecture 28

2024-11-15
Week 10, Fri

- Lemma: Let $G = (V, E)$ be a graph. Then $S \subseteq V$ is an independent set if and only if $V \backslash S$ is a vertex cover.
- Proof: Let $T = V \backslash S$. If T is a vertex cover, then every edge $e = (u, v)$ has some endpoint in T.
  - Notice that it can't be the case where there is an edge (u, v) with both endpoints in S.
  - Therefore, S is an independent set.
  - If S is an independent set, then any edge $e = (u, v)$ has at least one endpoint not in S. I.e., $v \in T$ or $u \in T$.
  - Therefore, T is a vertex cover.
- Corollary: S is an independent set of size $\geq k$. T is a vertex cover of size $\leq n - k$.

Polynomial Reduction Examples:

- Theorem 1: IS $\leq_p$ VC.

```
set IS(Graph G, int k) {
    return VC(G, n - k);
}
```

- Theorem 2: VC $\leq_p$ IS.

```
set VC(Graph G, int k) {
    return IS(G, n - k);
}
```

The Set Cover Problem:

- Input: A set U of n elements. U is "the universe"; m subsets of U: $S_1, \dots, S_m \subseteq U$; an integer k.
- Output: YES if and only if there is a set $T \subseteq \{1, \dots, m\}$ such that $|T| \leq k$ and $\bigcup_{i \in T} S_i = U$.
- Theorem 3: VC $\leq_p$ SC.
- Proof: Let $G = (V, E)$ and k be the inputs for the VC problem. For each vertex i, define $S_i =$ {edges e such that i is an endpoint of e}. $U =$ {edges} $= E$.

```
set VC(Graph G, int k) {
    for(int i = 1; i <= n; i++) {
        S_i = {edges e with i as endpoint};
    }
    return SC(E, S_1 ... S_n, k);
}
```

Lecture 29

2024-11-18
Week 11, Mon

- Claim 1: If U has a set cover T with $|T| \leq k$, then G has a vertex cover of size $\leq k$.
- Claim 2: If there is a vertex cover of size $\leq k$, then there is a set cover of size $\leq k$.
- Proof of Claim 1: (Claim 2 is similar by construction.) Let T be a set cover with size $\leq k$. Then for all $v \in U$, there is an $i \in T$ such that $v \in S_i$. For each edge $e \in E$, there is $i \in T$ such that $e \in S_i$. I.e., e has v_i as its endpoint. Let $C = \{v_i : i \in T\}$.
  - Then for all edges e, one of its endpoints is in C. Thus, C is a vertex cover.
- Nuance: Pay attention to the problem size.

The Satisfiability Problem (SAT):

- Inputs: n Boolean variables $x_1, \dots, x_n$; m clauses $c_1, \dots, c_m$.
- Literal: A variable x_i or its negation $\overline{x_i}$.
- Each clause c_j is OR of literals. For example. $c_j = x_1 \vee x_2 \vee x_{17}$.
- Outputs: YES if and only if we can assign truth values to variables so that each clause is satisfied, i.e., evaluates to TRUE.

The 3-SAT Problem:

- Like SAT, but each clause is OR of three literals.
- More generally: k-SAT. Each clause is OR of k literals.

- Fact: For any k, k-SAT $\leq_p$ SAT.
- Fact: For any $k \geq 3$, SAT $\leq_p$ k-SAT.
- Fact: There is a polynomial-time algorithm for 2-SAT.

Polynomial Time Verifiers:

- V is a polynomial-time verifier for a decision problem L if:
  1. V takes two inputs $X \in \{0,1\}^*$, $w \in \{0,1\}^*$.
  2. There is a polynomial function P such that for all strings x, $x \in L$ if and only if there is a string w with $|w| \leq P(|x|)$ and $V(x, u) =$ YES.

Important High-Level Considerations:

- The length of w is polynomial in the length of x.
- If x is YES input for L, there is some w which causes V to output YES.
- If x is NO input, V should output NO no matter what w is.

Verifier for the 3-Color Problem:

1. Convert x into graph $G(V, E)$. Let n be the number of vertices in the graph.
2. If w is not a $2n$ bit string, return NO.
3. Treat w as an array of n colors.
   - We define the following encoding for w with a length $2n$ bits: 00 for red; 10 for blue; 01 for green; 11 is not used.
4. If color$[v] = 11$ for any $1 \leq v \leq n$, return NO.
5. For each edge $(u, v) \in E$, return NO if color$[v]$ == color$[v]$.
6. Return YES.

Verifier for the Not-Factoring Problem (Lab Problem 6):

- $X = \langle n, k \rangle$.
- w represents prime factorization of n. $w = \langle P_1, P_2, \ldots, P_m \rangle$.

1. Validate format of w.
2. Check that $P_i \geq k$ for all $1 \leq i \leq m$.
3. $n = \prod_{i=1}^{m} P_i$.
4. Verify that each P_i is prime.

Lecture 30

2024-11-20
Week 11, Wed

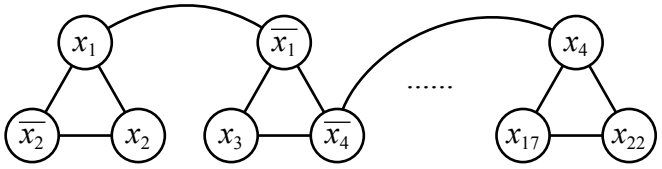
The 3-SAT Problem:

- Inputs: n Boolean variables $x_1, \ldots, x_n$; m clauses $c_1, \ldots, c_n$, each clause is an OR of 3 literals (x_i or $\overline{x_i}$), e.g., $c_j = x_3 \vee x_4 \vee \overline{x_{17}}$).
- Output: YES if there is a satisfying assignment.

Reductions with Gadgets:

- When reducing A $\leq_P$ B, convert each piece of input in problem A into a piece of input in problem B.

- Theorem: 3-SAT $\leq_P$ IS.
- Proof: Let $x_1, \ldots, x_n, c_1, \ldots, c_n$ be a 3-SAT input.
  1. For each clause $c_j = l_{j_1} \vee l_{j_2} \vee l_{j_3}$, create a triangle.
  2. Add an edge between any pair of conflicting literals (e.g., $x_1 - \overline{x_1}$).



  3. Run IS algorithm on constructed graph and m.
  4. Return whatever IS algorithm returns.

- Claim: A 3-SAT input is only satisfiable if and only if graph has IS of $\geq m$ vertices.
- Proof (Forward Direction): Suppose a 3-SAT input is satisfiable, then fix a satisfying truth assignment for clauses. We can pick one literal in each clause that evaluates to TRUE. The vertices corresponding to these literals form an independent set in the graph.
- Proof (Reverse Direction): If a constructed graph has an IS of size m, this IS has no conflicting literals. We can find a satisfying truth assignment for the 3-SAT input by setting variables so that all literals in the IS evaluate to TRUE and setting arbitrary for the rest of our assignment.

- Theorem: Prove if A $\leq_P$ B and B $\leq_P$ C, then A $\leq_P$ C.
- Proof Sketch: Take an algorithm for A that uses B as a subroutine. Instead of calling the subroutine for B, emulate the algorithm for B that uses C as a subroutine.
- To-do: Argue that we do polynomial amount of work plus a polynomial number of calls to C.

SAT Reducibility Chain:

The Cook-Levin Theorem:

- Every decision problem in the complexity class NP can be reduced to the SAT problem.
- If we can solve the SAT problem in polynomial time, then P = NP.

Lecture 31

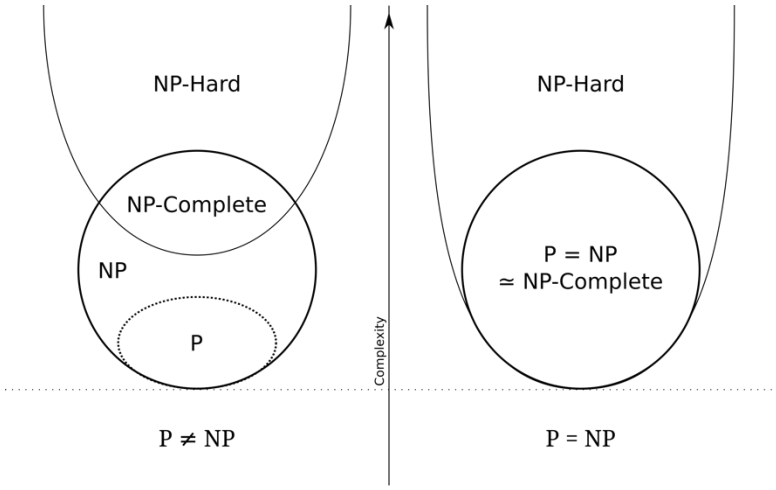
2024-11-22
Week 11, Fri

Complexity Classes:

- P: The set of all decision problems solvable in polynomial time.

- NP: The set of all decision problems verifiable in polynomial time.
- Fact: P ∈ NP.
- NP-Hard: The set of all decision problems B such that for any $A \in$ NP, $A \leq_p B$.
- For example, the Cook-Levin theorem tells us that SAT is in NP-Hard.
- NP-Hard does not intersect P. Also contrary to naming, NP-Hard $\not\subseteq$ NP.

- NP-Complete: NP ∩ NP-Hard.
- There are polynomial-time verifiable problems to which any problem in NP can be reduced. These seem to be the key to finding out whether P = NP, or P $\subsetneq$ NP.



Showing a Problem X is NP-Complete:

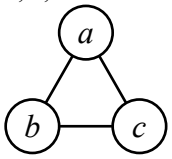
1. Show $X \in$ NP.
2. Pick problem Y known to be NP-Complete.
3. Reduce $Y \leq_p X$. Given an instance S_Y for problem Y, construct (in polynomial time) instance S_X for X such that:
   a. If S_Y is YES input, then so is S_x.
   b. If S_Y is NO input, then so is S_X.

- Theorem: 3-Color is NP-Complete.
- Proof: ① 3-Color ∈ NP. ② Pick 3-SAT. ③ Reduce 3-SAT $\leq_p$ 3-Color using *gadgets*.

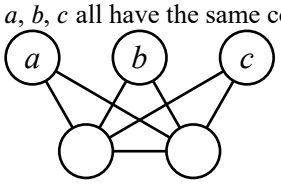
3-Color Graph Coloring:

- For each problem, create a 3-colorable graph with exactly that property and no other constraints. You can add vertices you want, and can hardwire vertices to be red, blue, or green.

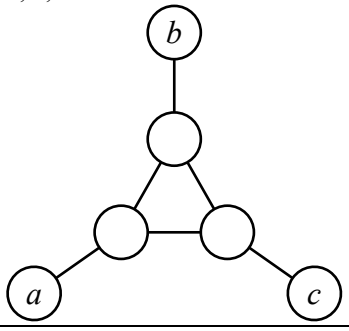
1. *a*, *b*, *c* all have different colors.



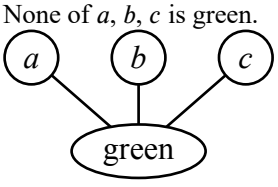
2. *a*, *b*, *c* all have the same color.



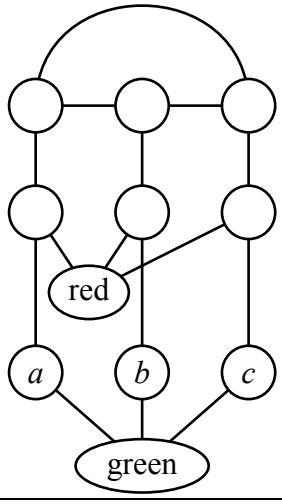
3. *a*, *b*, *c* do not all have the same color.



4. None of *a*, *b*, *c* is green.

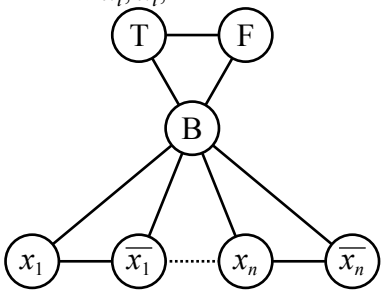


5. None of *a*, *b*, *c* is green and they cannot all be blue.

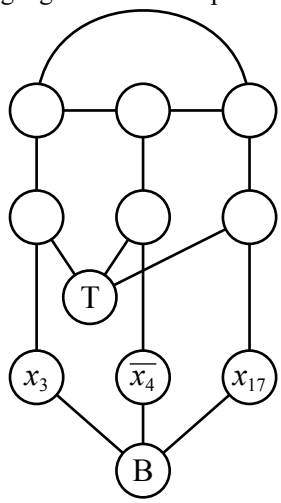


3-SAT → 3-Color Reduction:

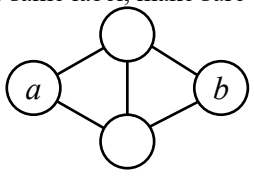
1. Create a triangle with vertices T, F, and B.
2. For each Boolean variable x_i, create a triangle between x_i, $\overline{x_i}$, and B.



3. For each clause (e.g., $c_j = x_3 \vee \overline{x_4} \vee x_{17}$), create a gadget as shown in problem 5 in the previous section.



4. Connect everything together. For any nodes with the same label, make sure that they have the same color.



Note that vertices a and b in the graph above will have the same color.

Lecture 32

2024-11-25
Week 12, Mon

Approximation Algorithm:

- Def: An approximation algorithm is an algorithm that runs in polynomial time and produces solutions that are guaranteed to be to be close to optimal.
- Note that approximation algorithms still must produce the correct answer.
- Approximation Ratio: An algorithm has an approximation ratio of P(n) ("Rho" of n) if for any input of size n, the "cost" c produced by the algorithm is within a factor of P(n) of the "optimal cost" OPT. I.e., $\max\left\{\frac{c}{\text{OPT}}, \frac{\text{OPT}}{c}\right\} \leq \text{P}(n)$.
- Note: If a problem is a minimization problem, then $c \geq \text{OPT}$, so $\text{P}(n) = \frac{c}{\text{OPT}}$.
- Note: If a problem is a maximization problem, then $c \leq \text{OPT}$, so $\text{P}(n) = \frac{\text{OPT}}{c}$.

Approximation Ratios:

- OK: $\text{P}(n) = \sqrt{n}, \log(n)$.
- Better: $\text{P}(n) = O(1)$. (Often constant-factor approximations are not hard to get.)
- Best: $1 + \varepsilon$ for any $\varepsilon > 0$.

The VC-OPT Problem:

- Input: Graph $G = (V, E)$.

- Output: Smallest possible vertex cover.

Algorithm VC-Approx:

- Provided a graph $G = (V, E)$.

```
set VCApprox(Graph G) {
    set C <- {};
    E' <- E;
    while(E' != {}) {
        pick edge (u, v) in E';
        add u, v to C;
        remove from E' all edges incident to u or v;
    }
    return C;
}
```

- Theorem: VC-Approx is a 2-approximation algorithm.
- Proof: Let C^* be the optimal (i.e., smallest) vertex cover for G.
  - Let A. Be the set of all edges picked by our algorithm in the first line in the while loop. Notice that the edges in A share no endpoints (they are endpoint disjoint).
  - Therefore, $|C^*| \geq |A|$ since each edge in A must be covered by different vertices. Also notice that the size of our covering is $|C| = 2|A|$. Therefore, $|C| = 2|A| \leq 2|C^*|$, so $|C| \leq 2|C^*|$. ∎
- For the vertex cover problem, the approximation resistance applies for any approximation smaller than 2. Finding such an approximation that is also efficient requires P = NP.

The Weighted Vertex Cover (WVC) Problem:

- Input: Graph $G = (V, E)$ with vertex weights $\{w_i : i \in V\}$.
- Output: Minimum weight vertex cover. For any set $w(s) = \sum_{i \notin S} w_i$.
- Theorem: WVC is NP-Hard.
- Proof: VC-OPT $\leq_P$ WVC. Given $G = (V, E)$, assign weight = 1 for all vertices.

MTSP Approximation Algorithm (Lab Problem 1):

1. Compute the MST (minimum spanning tree).
2. Pick an arbitrary start vertex.
3. Perform DFS on the MST.
4. Return the vertices in order of when we visit in our DFS, and add the start vertex at the end.

Lecture 33

2024-11-27
Week 12, Wed

The Weighted Vertex Cover (WVC) Problem (Continued):

The Pricing Method:

- The pricing method, aka, the "primal-dual" approach.
- Edge e will pay P_e to be covered. Vertex i wants to be paid w_i to provide coverage.
- Fair Pricing: A price is unfair if for any vertex i, $\sum_{e=(i,j)} P_e \leq w_i$.
- We call a vertex i tight if $\sum_{e=(i,j)} P_e = w_i$.
- Note: Let S^* be the optimal weighted vertex cover, and suppose prices are fair, then $\sum_e P_e \leq \sum_{i \in S^*} \sum_{e=(i,j)} P_e \leq \sum_{i \in S^*} w_i = w(S^*)$.

Algorithm:

```
set WVCApprox(Graph G, set {w_i}) {
    Initialize P_e = 0 fo all edges;
    while(there is edge e = (i, j) such that neither i nor j are tight) {
        Fix such an edge e;
        Increase P_e until either i or j is tight;
    }
    Return set S of tight vertices;
}
```

- Claim 1: S is a vertex cover.
- Proof: If S is not a vertex cover, then there is some edge $e = (i, j)$ such that neither i nor j is covered. S is a set of tight nodes, so this means neither i nor j are tight. But then we wouldn't have stopped the while loop.

- Claim 2: $w(s) \leq 2 \sum_e P_e$.
- Proof: $w(s) = \sum_{i \in S} w_i = \sum_{i \in S} \sum_{e=(i,j)} P_e \leq 2 \sum_e P_e$. The second equality is true because all $i \in S$ are tight. The third equality is true because each edge is incident to only 2 vertices, so each P_e can appear at most 2 times.

Lecture 34

2024-12-02
Week 13, Mon

Linear Programming (LP):

Example:

- Goal: Minimize $2x_1 + x_2$ (the objective function).
- Subject To: $x_1 + 2x_2 \geq 2$; $3x_1 + 2x_2 \geq 4$; $x_1, x_2 \geq 0$ (constraints).
- The space that satisfies all the constraints is called the feasible solution space.

General LP Form:

- Goal: Minimize the objective function $f(x_1, \ldots, x_n) = \sum_{i=1}^{n} a_i x_i$.
- Subject To:
  1. *Constraints*: $c_{11}x_1 + c_{12}x_2 + \cdots + c_{1n}x_n \geq b_1$; $c_{21}x_1 + c_{22}x_2 + \cdots + c_{2n}x_n \geq b_2$; …; $c_{n1}x_1 + c_{n2}x_2 + \cdots + c_{nn}x_n \geq b_n$.
  2. *Nonnegative Variables*: $x_1, \ldots, x_n \geq 0$.

Matrix Form:

- $a, x \in \mathbb{R}^n$.
- $b \in \mathbb{R}^m$.
- $c \in \mathbb{R}^{m \times n}$.
- We want to minimize $a^{\mathrm{T}} \cdot x$ subject to $c_x \geq b$ and $x \geq 0$.

History of LP:

- Motivation: Optimization and planning.
- [Kantorovich '39, Koopmans '42]: WWII planning and logistics; 1975 Nobel Economics Prize.
- [Dantzig '47]: A simple method that is efficient in practice, but exponential time in the worst case.
- [Khachiyan '81]: LP ∈ P. The ellipsoid method.

Weighted Vertex Cover as a ~~Linear~~ Integer Program:

- For each vertex, create a variable x_i, where $x_i = 1$ if $i \in$ VC and $x_i = 0$ if $i \notin$ VC.
- Goal: Minimize $\sum_{i=1}^{n} w_i \cdot x_i$.
- Subject To:
  1. *Edge Constraints*: For each edge (i, j), add constraint $x_i + x_j \geq 1$.
  2. *Boolean Variables*: Each $x_i \in \{0, 1\}$.
- **Note that this is actually an integer program (IP).**
- Integer Program: Like LP, but the variables are restricted to be integers.
- Problem: IP is known to be NP-Hard. The fact that integers are discrete make integer programming hard, while linear programming is generally solvable in polynomial time.

LP Relaxations:

- Express an optimization problem as an integer program (IP).
- Unfortunately, IP is hard unless P = NP.
- Good news: we can solve a related LP is polynomial time, and then use the solution to get a reasonable approximation.

LP Relaxation of WVC:

- Run the following LP: Minimize $\sum_{i=1}^{n} w_i x_i$ subject to:
  1. $x_i + x_j \geq 1$ for all $(i, j) \in E$.
  2. $x_i \geq 0$ for all $1 \leq i \leq n$.
- Output $S := \{i : x_i \geq ½\}$.
- Next time, we will show:
  - Claim 1: S is a vertex cover.
  - Claim 2: $w(S) \leq 2 \cdot w(S^*)$, where S^* is the optimal weighted vertex cover.