# Graph

**Graph usecases:** road networks, social networks, Internet, transition graphs

## Graph Basics

- mimimum number of edges in an undirected graph = |V| - 1
- maximum number of edges in an undirected graph = (|V|(|V| - 1)) / 2
- Sparse graph is more efficiently represented using an adjacency list
- Dense graph is more efficiently reppresented using an adjacency matrix

## Tree

- exactly one path between any two vertices
- A tree with n vertices has n – 1 edges

## Connectivity and Graph Traveral Problem

**Input**: Graph G = (V, E) using linked lists and two vertices s, t $\in$ V

**Output**: Whether there is a path connecting s to t in G

## BFS

BFS($s$)

```
1: head ← 1, tail ← 1, queue[1] ← s
2: mark s as "visited" and all other vertices as "unvisited"
3: while head ≤ tail do
4:     v ← queue[head], head ← head + 1
5:     for all neighbors u of v do
6:         if u is "unvisited" then
7:             tail ← tail + 1, queue[tail] = u
8:             mark u as "visited"
```

## DFS using Recurrsion

DFS($s$)

```
1: mark all vertices as "unvisited"
2: recursive-DFS(s)
```

recursive-DFS($v$)

```
1: mark v as "visited"
2: for all neighbors u of v do
3:     if u is unvisited then recursive-DFS(u)
```

## DFS using Stack

**DFS($s$)**

```
1: S := EmptyStack
2: Push(S, s)
3: mark s as "visited" and all other vertices as "unvisited"
4: while not Empty(S) do
5:     v ← Pop(S)
6:     if there is at least one unvisited vertex in Adj[v] then
7:         Push(S, v)
8:         Pick u to be any unvisited vertex in Adj[v]
9:         Push(S, u), mark u as "visited"
```

## Bipartitie Graph

**Def.** A graph G = (v, E) is a bipartite graph if there is a partition of V into two sets L and R such that for every edge (u, v) ∈ E, either u ∈ L, v ∈ R or v ∈ L, u ∈ R

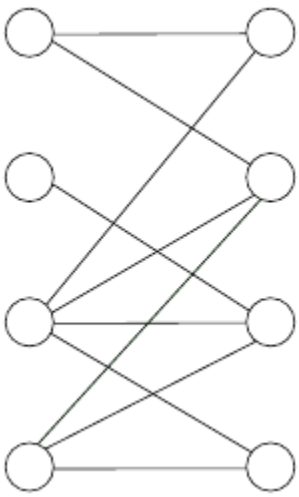
**Input**: Graph G = (V, E) using linked lists

**Output**: whether G is a bipartite graph or not

### Testing bipartiteness using BFS

**test-bipartiteness($s$)**

```
1: head ← 1, tail ← 1, queue[1] ← s
2: mark s as "visited" and all other vertices as "unvisited"
3: color[s] ← 0
4: while head ≤ tail do
5:     v ← queue[head], head ← head + 1
6:     for all neighbors u of v do
7:         if u is "unvisited" then
8:             tail ← tail + 1, queue[tail] = u
9:             mark u as "visited"
10:            color[u] ← 1 − color[v]
11:        else if color[u] = color[v] then
12:            print("G is not bipartite") and exit
```



### Testing bipartiteness using DFS

**test-bipartiteness-DFS($s$)**

```
1: mark all vertices as "unvisited"
2: color[s] ← 0
3: recursive-test-DFS(s)
```

**recursive-test-DFS($v$)**

```
1: mark v as "visited"
2: for all neighbors u of v do
3:     if u is unvisited then
4:         color[u] ← 1 − color[v], recursive-test-DFS(u)
5:     else if color[u] = color[v] then
6:         print("G is not bipartite") and exit
```

## Bipartite Observations:

- Bipartite graph may contain cycles
- A graph is bipartitie if and only if it has no odd cycle
- to determine whether a graph is bipartitable, verity non-existence of odd cycle
- the procedure of finding a bipartite graph is the same procesure of BFS, coloring L1 and L2 and so on.
- If a graph is a tree, then it is also a bipartite graph
- BFS and DFS naturally induce a tree

## Perfect Matching Problem

**Def.** A perfect matching in a graph is a matching where every vertex in the graph is included in exactly one edge of the matching.

## Perfect Matching background

- A perfect matching can only exist with an even number of vertices
- Hall's Marriage Theorem must be met for a perfect matching to exist.

## Hall's Marriage Theorem

A perfecf matching that covers all vertices of U exists if and only if for every subset S $\subseteq$ U, the number of vertices in S is less than or equamber of neighboring vertices in V. In other words: $|S| \leq |N(S)|$ for every S $\subseteq$ U, where N(S) is the set of vertices in V that are neighbors of at least one vertex in S.

Valid perfect metching Ex. U = {A, B, C}, V = {D, E, F}, edges = A-D, A-E, B-E, B-F, C-F

Invalid perfect matching Ex. U = {A, B, C}, V = {D, E, F}, edges = A-F, B-F, C-D, C-E, C-F

## Stable Matching Problem

## Stable Matching Background

- We can always find a stable matching regardless of the number of participants in each group
- A matching is stable if there exists no blocking pairs, meaning no participants should both prefer each other over their assigned match.
- Based on which side has the decision power and the order of preferences, the final result can differ.

## Gale-Shapley Algorithm

**Input**: A bipartite graph whose one set is girls and the other set is boys

**Output**: stable matching between the sets

*gale-shapley*$(GIRLS, BOYS)$

```
1:  mark all girls G ∈ GIRLS and boys B ∈ BOYS as free
2:  while there is a free girl G who has not proposed to every boy in her list do
3:      B ← highest-ranked boy in G's list to whom she has not yet proposed
4:      if B is free then
5:          pair (G, B) tentatively
6:      else let G' be the girl currently paired with B
7:          if B prefers G over G' then
8:              pair (G, B) tentatively and G' becomes free
9:          else
10:             G remains free
11: return the final matching set
```

Runtime: $O(n^2)$

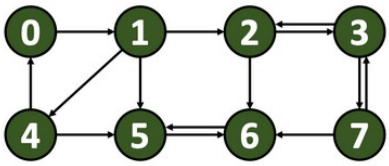
## Strongly Connected Components (SCCs) Problem

**Def.** Two notes, u and v, are strongly connected if there is a path from u to v and a path from v to u, that is, two notes u and v in a directed graph are mutually reachable.

## Kosaraju's Algorithm – detect all SSCs

node 5 was the node that finished DFS first time so that it is placed as the first item pushed into the stack.

1. Run **DFS** on the original graph and record finish times

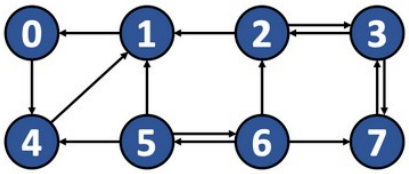



visited = {0, 1, 2, 3, 7, 6, 5, 4}

stack = [5, 6, 7, 3, 2, 4, 1, 0]

This time you run DFS on each set of soon-to-be SSC in the decreasing order of finish times.

2. Transpose the graph (reverse edges)




3. Run **DFS** again, in decreasing order of finish times

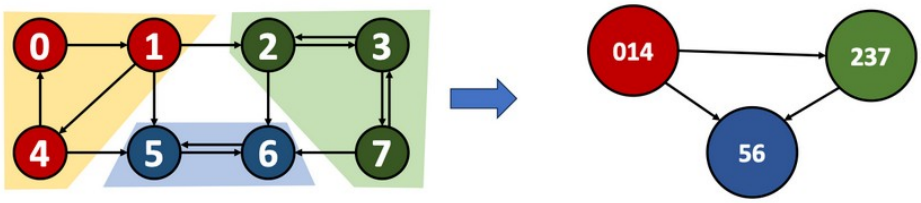
stack = [~~5~~, ~~6~~, ~~7~~, ~~3~~, ~~2~~, 4, ~~1~~, ~~0~~]

visited = {0, 4, 1, 2, 3, 7, 6, 5}

scc = ~~[0, 4, 1] [2, 3, 7]~~ [6, 5]

set_scc = [[0, 4, 1], [2, 3, 7], [6, 5]]

Strongly connected component Graph is acyclic and produces a DAG graph.




## Topological Ordering Problem (Khan's Algorithm)

**Input**: a directed acyclic graph (DAG) G = (V, E)

**Output**: 1-to-1 function $\pi : V \rightarrow \{1,2,3, \ldots, n\}$, so that if $(u, v) \in E$ then $\pi(v) < \pi(v)$

**$\pi(v)$** = the position number assigned to vertex v

**Algorithm**: each time take a vertex without incoming edges, then remove the vertex and all its outgoing edges

topological-sort($G$)

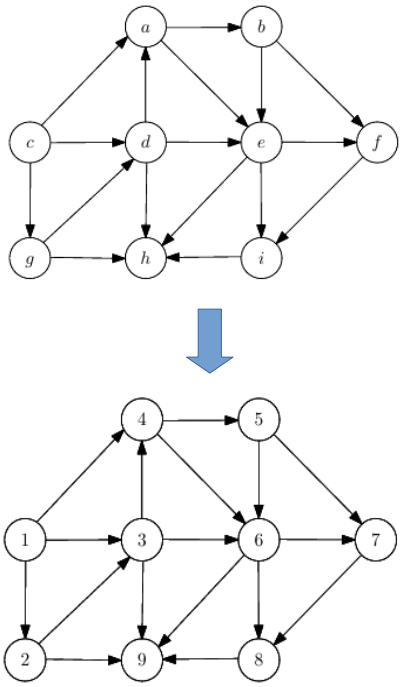
```
1: i ← 0
2: for V ≠ ∅ do
3:     v ← vertex in V without incoming edges
4:     i ← i + 1, π(v) ← i and V ← V \ {v}
5:     while V ≠ ∅ do
6:         for every u ∈ V such that (v, u) ∈ E do
7:             E ← E \ {(v, u)}
```

## How to make the algorithm more efficient?

- use linked lists of outgoing edges
- maintain the in-degree $d_v$ of vertices
- maintain a queue (or stack) of vertices v with $d_v = 0$

**topological-sort($G$)**

```
1: let d_v ← 0 for every v ∈ V
2: for every v ∈ V do
3:     for every u such that (v, u) ∈ E do
4:         d_u ← d_u + 1
5: S ← {v : d_v = 0}, i ← 0
6: while S ≠ ∅ do
7:     v ← arbitrary vertex in S, S ← S \ {v}
8:     i ← i + 1, π(v) ← i
9:     for every u such that (v, u) ∈ E do
10:        d_u ← d_u − 1
11:        if d_u = 0 then add u to S
```




DAGs are a very common structure in computer science and can be used to encode precedence relations or dependencies in a natural way.

# Greedy Algorithms

## Optimization problem <-> Decision problem

**Optimization problems**: you have many possible ways to do something (e.x, different routes to drive home), then you want to find the optimized solution based on some given constraints that apply an optimization aspect to the problem, such as which path is the fastest route to home, or the shortest path to home

**Decision problem:** yes/no version question of that optimization problem

## Greedy Algorithm Background

- Often used for those involving and solving optimization problems
- Often run in polynomial time due to simplicity, easy to come up with, and analyze running time
- Hard to see correctness.
- At each step, make an irrefutable decision using a "reasonable" strategy

## Analysis of the Greedy Algorithm

**Safety:** need to prove that the reasonable strategy is safe (key)

**Self-reduce:** shows that the strategy solves smaller instances of the problem (naturally easy)

## 1. The Box Packing Problem

**Input**: *n* boxes of capacities $c_1$, $c_2$, $c_n$ and *m* items of sizes $s_1$, $s_2$, $s_n$. Each box can hold only one item where $s_j <= c_i$.

**Output**: Fit as many items as possible into boxes.

### 1. Discuss various strategies

Strategy 1. Place the largest item into the largest box it can fit into

Strategy 2. Start with any item, and put it into the smallest box that it can fit ✅

Strategy 3. Place the smallest item in the smallest box that can hold it

**Lemma**: a preliminary or auxiliary proposition that suggests a strategy that needs to be proven for immediate use in solving a given problem.
In the Box Packing algorithm, our goal is to produce an optimal solution that fits as many items as possible into the boxes. In order to do this, we need to propose a lemma that supports the strategy and proves its safefy showing that applying it to justify each incremental decision along the way preserves optimality and ultimately yields an optimal solution

**Lemma**: When starting with any item j, it is safe to place it in the smallest box (b1) it can fit in. In other words, there is an optimum solution S where j is in b1.

### 2. Prove the accuracy (safety) of the lemma

**Proof**: We show that the proposed lemma is correct and can be safely applied while solving the problem. Establishing the lemma's correctness justifies our step-by-step choices and guarantees that the algorithm ultimately produces the desired optimal solution.

**proof by exchange argument**: We use an exchange argument to show that, at each step, the choice justified by the lemma can be incorporated into an optimal solution. In other words, if an optimal solution makes a different choice, we can "exchange" it for our choice without reducing optimality—so our step remains consistent with some optimal solution.

**proof by exchange argument:**

1. Let j = an item whose b1 is the smallest box that it can fit in.

2. Assume there exists an optimum solution S in which j is indeed included in b1, then the proposed lemma's choice matches the optimal solution. If so, the lemma is accurate (the proof is complete) because S is an optimal solution.

3. Otherwise (if above is not the case), suppose what happened in S was that the item j was placed in in another smallest box b2.

4. Since both b1 and b2 are the smasllest boxes that can fit j in, they can exchange their item with each other, which produces an optimal solution S'

5. Since S' contains j in b1, it is a desired concluded outcome symmetric to S. Therefore, the lemma is safe and accurate.

### 3. Prove the strategy is self-reducing

We need to show that the remaining task applying the strategy is to solve a (many) smaller instance(s) of the same problem.

This is trivial -> the remaining instance is obtained by removing Item j and b1 from the original instance.

### Box Packing Problem Algorithm

**This algorithm adapts a different strategy:** starts with any box bi, and place the largest item that it can fit in

Greedy Algorithm for Box Packing

```
1: T ← {1, 2, 3, ··· , m}
2: for i ← 1 to n do
3:     if some item in T can be put into box i then
4:         j ← the largest item in T that can be put into box i
5:         print("put item j in box i")
6:         T ← T \ {j}
```

**line 1**: sorted list of items

**line 2:** sorted box list loop

**line 3-4:** when some item can fit in, find the largest item that can fit in $b_i$

**line 6:** reduce T

**runtime:** with sorted items and boxes O(max{n, m})

## 2. Interval Scheduling Problem

**Input**: n jobs, job i with start time si and finish time fi. Job i and j are compatible if [si , fi) and [sj , fj ) are disjoint

**Output**: A maximum-size subset of mutually compatible jobs

### 1. Discuss strategies

Select jobs based on the earliest finishing time whose starting time doesn't conflict with the previous job's finish time if any. Since we are choosing the earliest finishing job each time, making a plenty of room for more jobs to be chosen in subsequent steps.

**Lemma:** When selecting jobs, it is safe to choose the job (j) with the earliest finish time to maximize the number of jobs at the end. In other words, there is an optimal solution S that includes j.

## 2. Prove the safety

**proof by exchange argument**:

1. Let j = the job of the earliest finish time

2. Assume there exists an optimal solution S in which j is included and produces a total of 4 chosen jobs as a result, following the lemma. Then, the proof is complete since applying the lemma would produce S.

3. Otherwise, there is another optimal solution which includes j' where j''s finishing time >= j's finishing time and produces a total of 4 chosen jobs as a result.

4. Since both j' and j have the earliest finish time, we can exchange the chosen job with each other, and still produce a total of 4 chosen jobs, which results in S' that contains j in its output.

5. Since S' produces an output that includes j, identical to S. The lemma is valid and accurate.

## 3. Prove self-reducing

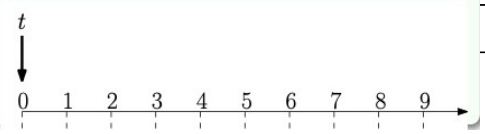
The remaining jobs yet to be considered is obtained by removing the job j from the original instance

## Interval Scheduling Problem Algorithm

**s:** list of job start times, **f:** list of job finish times, **n:** number of jobs

Schedule($s, f, n$)

1: sort jobs according to $f$ values
2: $t \leftarrow 0$, $S \leftarrow \emptyset$
3: **for** every $j \in [n]$ according to non-decreasing order of $f_j$ **do**
4: **if** $s_j \geq t$ **then**
5: $S \leftarrow S \cup \{j\}$
6: $t \leftarrow f_j$
7: **return** $S$




**line 2: t** contains the finish time of the last chosen job. **S** is the set of chosen jobs.

**line 3:** take each job j from the sorted jobs

**runtime:** O(nlogn) + O(n) = sorting + iteration

naive implementation without sorting would be O(n^2)

## 3. Interval Partitioning Problem

**Input**: n jobs, job i with start time si and finish time fi and j are compatible if [si , fi) and [sj , fj ) are disjoint

**Output**: A minimum number of machines to schedule **all jobs** so that all jobs on a single machine are compatible.

## 1. Discuss strategies

Sort the jobs in order to easily identify conflicting jobs, and choose jobs in that order while adding an additional machine every time confronting a conflict.

**Lemma:** It is safe to add an additional machine (m) each time encountering a conflict when selecting jobs based on a specific order. Thus, there is an optimal solution S that includes machine m.

## 2. Prove the safety

**Proof by exchange argument**

1. Let m = a machine added to resolve a conflict of two jobs j1 and j2.

2. Say there exists an optimal solution S in which m is included, following the proposed lemma. Then, the proof is complete since the optimal solution is derived from the lemma.

3. Otherwise, there is another optimal solution that adds machine x to resolve the conflict.

4. Since another output also includes one machine to resolve the conflict, replacing machine x with machine m produces S'.

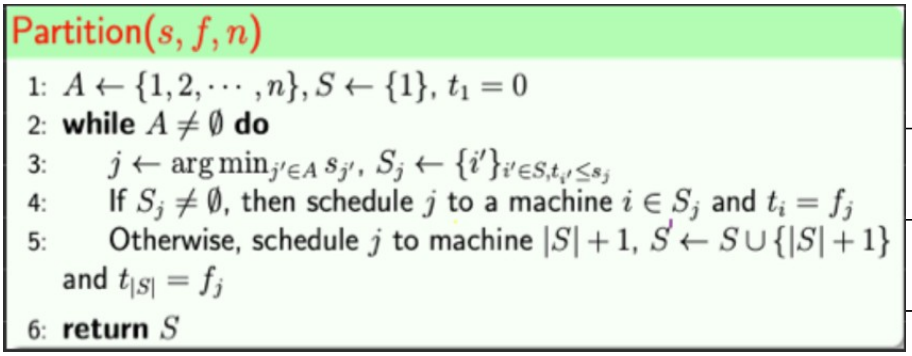
5. Since S' is an optimal solution that includes m, identical to S. The lemma is accurate and valid.

### 3. Prove self-reducing

The remaining jobs yet to be considered is obtained by removing the jobs that already have been considered from the original instance.

### Interval Partitioning Problem Algorithm

**s:** list of job start times, **f:** list of job finish times, **n:** number of jobs




**line 1:** A is sorted list of jobs to be scheduled. S is a set of machines. ti is the finish time of machine i.

**line 3 left:** choose the job with the earliest time sj

**line 3 right:** find machine that is free if ti <= sj

**line 4:** if free machine exists, assign the job to that machine

**line 5:** if no such machine, add one more machine

**runtime:** O(nlogn) with Priority Queue for fetching avaiable machines

naive implementation without priority queue would be O(n^2)

### 4. Offline Caching Problem

**Input**: k = size of cache, n = number of pages, $p_1, p_2, p_t \in [n]$ = sequence of requests, $p_1, p_2, p_k$ = initial set of pages in cache

**Output**: minimised number of misses

### 1. Discuss strategies

When evicting a page from the cache, evict the one that is furthest in the future

**Lemma:** When evicting a page from cache, it is safe to evict the one with the furthest future (p). In other words, there is an optimal solution S that evicts p in every event of eviction

### 2. Prove the safety

**Proof by exchange argument**

1. Let p = the furthest request from seq among the cached pages

2. Assume there exists an optimal solution S which would evict p in the event of eviction, and produces 4 misses, following the proposed lemma. If this is true, the proof is complete

3. Otherwise, there is another optimal solution which evicts p', and still produces 4 misses

4. since both p and p' are the furthest in the future, we can replace the evicted page to be p instead of p', which results in S'

5. Since S' evicts p and produces 4 misses, it is identical to S. Thus, the lemma is accurate

## 3. Prove self-reducing

Evicting p will make room for the to-be-cached page in the future, and we continue with the rest of the request sequence

## Offline Caching Problem Algorithm

```
1:  for every p ← 1 to n do
2:      times[p] ← array of times in which p is requested, in
    increasing order                          ▷ put ∞ at the end of array
3:      pointer[p] ← 1
4:  Q ← empty priority queue
5:  for every t ← 1 to T do
6:      pointer[ρ_t] ← pointer[ρ_t] + 1
7:      if ρ_t ∈ Q then
8:          Q.increase-key(ρ_t, times[ρ_t, pointer[ρ_t]]), print "hit",
    continue
9:      if Q.size() < k then
10:         print "load ρ_t to an empty page "
11:     else
12:         p ← Q.extract-max(), print "evict p and load ρ_t"
13:     Q.insert(ρ_t, times[ρ_t, pointer[ρ_t]])      ▷ add ρ_t to Q with key
    value times[ρ_t, pointer[ρ_t]]
```

**line1,2:** For each page p, build an array of times in which p is requested.

**Line3:** make pointers to point to the first time each p is requested.

**line 5:** sequence of requests loop

**line 6:** whichever the page of the request is, increment the pointer

**line 7:** if cache hit, update priority value of that page to the next time

**line 9-12:** if a slot in cache, no need to evict. Otherwise, evict the furthest in the future.

**line 13:** insert page into cache with the priority of that page for the next time

**runtime:** O(n + Tlogk)

T: length of request sequence

k: cache size (size to be considered within the queue)

n: number of distinct pages (for scanning)

## 5. Priority Heap

**insert(v, key_value)** -> place the node at the end and heapify up (logn)

**extract_min()** -> replace the head with the end node, and heapify down (logn)

**decrease(v, new_key_value)** -> update order of a value and re-heapify-up (logn)

| data structures | insert | extract_min | decrease_key |
|---|---|---|---|
| array | $O(1)$ | $O(n)$ | $O(1)$ |
| sorted array | $O(n)$ | $O(1)$ | $O(n)$ |
| heap | $O(\lg n)$ | $O(\lg n)$ | $O(\lg n)$ |

## 6. (Data decomposition) Huffman Encoding Problem

There are 8 letters a,b,c,d,e,f,g,h in a language, and **need to encode a message using bits**

**idea**: use 3 bits per letter

**q:** can we do better?

**a:** using variable-length encoding scheme might be more efficient

**idea:** using fewer bits for letters that are more frequently used, and more bites for letters that are less frequently used. In order to do this, we consider prefix codes that ensure each letter is assigned unique prefix

## Properties of encoding tree

- rooted binary tree
- left edges are labelled 0 and right edges labelled 1
- A leaf corresponds to a code for some letter

## Best Prefix (Huffman) Codes

**input**: frequencies of letters in a message

**Output**: prefix coding scheme with the shortest encoding for the message

**Lemma1**: it is safe to make the two leastt frequent letters brothers

**Lemma2**: There is an optimum encoding tree, where the two least frequent letters are brothers

## Discuss Strategies

You compute the frequency rate of each letter across a message.

Based on each frequency rate of each letter, you map two lowest frequent letters together until the rooted binary tree is merged to frequency rate 1

When mapping, it doesn't matter in which order you list each letter out in the bottom base

Based on the finalized mapping, assign prefix codes to each letter

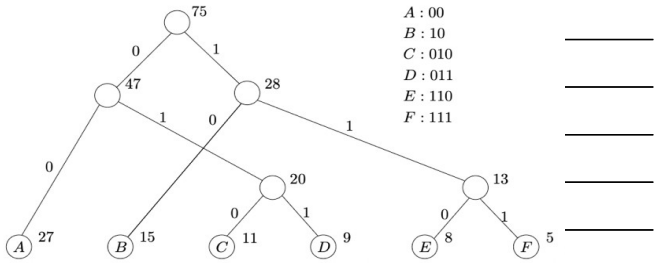
## Huffman Encoding Problem Algorithm

Huffman$(S, f)$

1: $Q \leftarrow$ build-priority-queue$(S)$
2: **while** $Q$.size $> 1$ **do**
3: $\quad x_1 \leftarrow Q$.extract-min$()$
4: $\quad x_2 \leftarrow Q$.extract-min$()$
5: $\quad$ introduce a new letter $x'$ and let $f_{x'} = f_{x_1} + f_{x_2}$
6: $\quad$ let $x_1$ and $x_2$ be the two children of $x'$
7: $\quad Q$.insert$(x', f_{x'})$
8: **return** the tree constructed

**line 1:** based on frequence of each letter, build priority queue

**line 3-7:** merge the lowest frequent letters and create an abstract parent that has combined frequencey



# Divide and Conquer Algorithms

## Divide and Conquer Algorithm Background

- Trivial algorithm already runs in polynomical time
- Divide-and-conquer gives a more efficient algorithm
- Main focus of analysis: runtime

**Divide**: Divide instance into many smaller instances

**Conquer**: Solve each of smaller instances recursively and separately

**Combine**: Combine solutions to small instances to obtain a solution for the original big instance

**Runtime analysis**: recursive programs

## Solving Recurrrences (The Master Theorem)

$$T(n) = aT(n/b) + O(n^c)$$

$$T(n) = \begin{cases} O(n^{\log_b a}) & \text{if } c < \log_b a \\ O(n^c \lg n) & \text{if } c = \log_b a \\ O(n^c) & \text{if } c > \log_b a \end{cases}$$

**a** = number of resursive sub-problems solved

**n/b** = size of each recursive subproblem

$\mathbf{n^c}$ = work done at the current level outside the recursive calls

This includes things like: scanning array, paritioning, combining answers, merging, comparisions.

**Ex:** constant work = $n^0 = 1$, linear work = $n^1 = n$, ...

## Counting Inversions

**Def.** Given an array A of n integers, an inversion in A is a pair (i,j) of indices such that i<j and A[i] >A[j].

**Input**: a sequence A of n numbers

**Output**: number of inversions in A

If we naively compare every element in A, it takes O(n^2).

However, if we can divide A into B and C, each sorted, finding inversions between B and C becomes O(n).

Based on this idea, we can divide the list like we did in Mergesort, and count inversions (conquer) as we combine.

sort-and-count$(A, n)$

```
1: if n = 1 then
2:     return (A, 0)
3: else
4:     (B, m1) ← sort-and-count(A[1..⌊n/2⌋], ⌊n/2⌋)
5:     (C, m2) ← sort-and-count(A[⌊n/2⌋ + 1..n], ⌈n/2⌉)
6:     (A, m3) ← merge-and-count(B, C, ⌊n/2⌋, ⌈n/2⌉)
7:     return (A, m1 + m2 + m3)
```

**line 4-5:** at first we divide A into B and C recursively. They will eventually return the sorted array along with the # of inversions from each function call.

**line 6:** this combines the total # of inversions while also attempting to count the # of inversions between B and C

**merge-and-count$(B, C, n_1, n_2)$**

```
1: count ← 0;
2: A ← array of size n1 + n2; i ← 1; j ← 1
3: while i ≤ n1 or j ≤ n2 do
4:     if j > n2 or (i ≤ n1 and B[i] ≤ C[j]) then
5:         A[i + j − 1] ← B[i]; i ← i + 1
6:         count ← count + (j − 1)
7:     else
8:         A[i + j − 1] ← C[j]; j ← j + 1
9: return (A, count)
```

To count inversions, we rely on j **in line 8** that increments whenever B is bigger than C.

Then, when B is smaller than C **in line 4**, we count inversions by adding j-1 **in line 6** (-1 is to prevent counting when B is initially smaller than C).

Also, when j > n2 **in line 4**, that means all the rest of elements in B is larger, so we count all the inversions in that respect.

**Runtime:**
There are **O(log n) levels**
Each level takes **O(n) to merge**
Therefore, O(logn) * O(n) = O(nlogn)
**Recurrence:** T(n) = 2T(n/2) + O(n)

## Mergesort

**merge-sort$(A, n)$**

```
1: if n = 1 then
2:     return A
3: B ← merge-sort(A[1..⌊n/2⌋], ⌊n/2⌋)
4: C ← merge-sort(A[⌊n/2⌋ + 1..n], n − ⌊n/2⌋)
5: return merge(B, C, ⌊n/2⌋, n − ⌊n/2⌋)
```

**line 3-4:** creates a recursion tree with logn levels, splitting itself takes O(1) per function call

**line 5:** split all elements in A, and merge them during combine steps

**merge$(B, C, n_1, n_2)$ \\ $B$ and ( length $n_1$ and $n_2$**

```
1: A ← []; i ← 1; j ← 1
2: while i ≤ n1 and j ≤ n2 do
3:     if B[i] ≤ C[j] then
4:         append B[i] to A; i ← i + 1
5:     else
6:         append C[j] to A; j ← j + 1
7: if i ≤ n1 then append B[i..n1] to A
8: if j ≤ n2 then append C[j..n2] to A
9: return A
```

**line 1:** A -> requires a new list allocation to store sorted elements

**line 2-6:** compare each element of B and C, and append to A in a sorted order.

**line 7-8:** if there is any remaining elements, since they are already sorted, and larger, simply append them all to A

**Runtime:**
**There are O(log n) levels**
**Each levle takes O(n) to merge**
**Therefore, O(logn) * O(n) = O(nlogn)**
**Recurrence: T(n) = 2T(n/2) + O(n)**

## Mergesort is a stable sorting algorithm

Property that equal items will appear in the same sorted order in the final list relative to how they appeared in the input list.

**Ex:** [A: 9, B: 8, C: 9, D: 7] – stable sorting → [D, B, A, C]

The initial order of A and C also appears in the sorted list.

## Quicksort

As opposed to Mergesort, Quicksort sorts elements during divide process by separating small numbers to the left and large numbers to the right based on a randomly chosen median.

## Quicksort Algorithm

**quicksort$(A, n)$**

1: **if** $n \leq 1$ **then return** $A$
2: $x \leftarrow$ a random element of $A$ ($x$ is called a pivot)
3: $A_L \leftarrow$ array of elements in $A$ that are less than $x$
4: $A_R \leftarrow$ array of elements in $A$ that are greater than $x$
5: $B_L \leftarrow$ quicksort$(A_L,$ length of $A_L)$
6: $B_R \leftarrow$ quicksort$(A_R,$ length of $A_R)$
7: $t \leftarrow$ number of times $x$ appear $A$
8: **return** concatenation of $B_L$, $t$ copies of $x$, and $B_R$

**line 2:** choose a random pivot (first element)
**line 3,4:** based on the pivot, create small and large sub-list
**line 5,6:** quicksort each left and right sub-list
**line 5,6:** they eventually reach elements, and return sorted sub-lists
**line 7:** merge sorted sub-lists (conquer)

## Quicksort is In-Place sorting algorithm

Above algorithm allocate new list spaces to return a sorted list. However, Quicksort can be in-place sorting algorithm, reducing memory space.

## In-place Quicksort Algorithm

**partition$(A, \ell, r)$**

1: $p \leftarrow$ random integer between $\ell$ and $r$, swap $A[p]$ and $A[\ell]$
2: $i \leftarrow \ell, j \leftarrow r$
3: **while true do**
4: while $i < j$ and $A[i] < A[j]$ do $j \leftarrow j - 1$
5: **if** $i = j$ **then** break
6: swap $A[i]$ and $A[j]$; $i \leftarrow i + 1$
7: **while** $i < j$ and $A[i] < A[j]$ **do** $i \leftarrow i + 1$
8: **if** $i = j$ **then** break
9: swap $A[i]$ and $A[j]$; $j \leftarrow j - 1$
10: **return** $i$

**line 1:** if p is the first element, we don't need to swap A[p] and A[l]
**line 4:** decrement j until A[i] becomes larger than A[j]
**line 6:** swap A[i] and A[j]
**line 7:** increment i until A[i] becomes larger than A[j]
**line 9:** swap A[i] and A[j]

**such that you can find the pattern that the algorithm is trying to find such A[i] to position it relatively more to the right than its original position.**

Meanwhile, when i = j in **line 5 and 8**, we break out of the loop, and returns the index I.

**quicksort$(A, \ell, r)$**

1: **if** $\ell \geq r$ **then return**
2: $m \leftarrow$ patition$(A, \ell, r)$
3: quicksort$(A, \ell, m - 1)$
4: quicksort$(A, m + 1, r)$

**line2-4:** based on the returned index i = m from parititon, we partition the left skewed and right skewed sublist again until they are sorted.

**Runtime:**
There are **O(log n) levels.**
Each level takes **O(n) to partition and divide.**
**O(1) to merge.**
Therefore, O(logn) * O(n) = O(nlogn)
**Recurrence:** T(n) = 2T(n/2) + O(n)

## Selection Problem - find i-th smallest number in an array in O(n) time

**Input**: a set A of n numbers, and $1 \leq i \leq n$

**Output**: the i-th smallest number in A

**Goal:** A sorting algorithm takes O(nlogn) to find such i-th smallest number in A. Our goal is to find it in O(n).

n = size of A, i = i-th smallest index

**selection**$(A, n, i)$

```
1: if n = 1 then return A
2: x ← lower median of A
3: A_L ← elements in A that are less than x          ▷ Divide
4: A_R ← elements in A that are greater than x       ▷ Divide
5: if i ≤ A_L.size then
6:     return selection(A_L, A_L.size, i)            ▷ Conquer
7: else if i > n − A_R.size then
8:     return selection(A_R, A_R.size, i − (n − A_R.size))   ▷ Conquer
9: else
10:    return x
```

**line 2-4:** based on a randomly chosen x, divide small and large sublist.

**line 5-8:** based on given index i, choose the left or right sub-list, and continue to find i-th index.

**line 9:** if neither small or large, we have found i-th element.

**Runtime:**

There are **O(log n) levels**

There are one sub-problem

Therefore, O(logn) = O(logn)

**Recurrence:** T(n) = T(n/2) + O(n)

## Closest Pair Problem

**Input**: A set of n points in the 2D plane

**Output**: find the pair of points with the smallest Euclidean distance

**Applications:**

Supposeyou're organizing a large outdoor event and want to ensure no two guests are too close

A logistics company trying to identify the two closest distribution centers to optimize routing

*closest-pair-recursive*$(P)$

```
1:  if |P| ≤ 3 then return (d, bestPair) using brute-force
2:  divide P into two subsets L and R
3:  (d_L, pair_L) ← closest-pair-recursive(L)
4:  (d_R, pair_R) ← closest-pair-recursive(R)
5:  d ← min(d_L, d_R)
6:  bestPair ← the pair corresponding to d
7:  find the vertical line that separates L and R
8:  S ← all points in P within distance d from that line
9:  sort the points in S by y-coordinate
10: for each point p_i in S do
11:     compare p_i with up to 7 next points in S
12:     if a closer pair is found then update d and bestPair
13: return (d, bestPair)
```

- sort points by x-coordinate
- recursively find closest pair in left and right halves
- check for closer pairs that cross the dividiing line ("strip")

Runtime: O(nlogn)

# Dynamic Programming Algorithms

## Dynamic Programming Algorithm Background

- Break up a problem into many overlapping sub-problems
- Build solutions for larger and larger sub-problems
- Use a table to store solutions for sub-problems for reuse

## Weighted Interval Scheduling

**Input**: Same as Interval Scheduling, and each job has a value > 0

**Output:** A maximum-value subset of mutually compatible jobs

Given jobs to be scheduled with values assigned to them, we want to find the most optimal set of jobs that returns the largest value.

**Define Subproblem:** opt[i] = the best value at instance I.

**Recurrence Relation:** $$opt[i] = \max\{opt[i-1], v_i + opt[p_i]\}$$

**opt[i - 1]** is the largest value observed from the previous job.

**vi** + **opt[pi]** is (the largest value observed from the job whose finish time <= start time of job i ) + current instance value.

### Interval Scheduling Algorithm

1: sort jobs by non-decreasing order of finishing times
2: compute $p_1, p_2, \cdots, p_n$
3: $opt[0] \leftarrow 0$
4: **for** $i \leftarrow 1$ to $n$ **do**
5: **if** $opt[i-1] \geq v_i + opt[p_i]$ **then**
6: $opt[i] \leftarrow opt[i-1]$
7: $b[i] \leftarrow$ N
8: **else**
9: $opt[i] \leftarrow v_i + opt[p_i]$
10: $b[i] \leftarrow$ Y

1: $i \leftarrow n, S \leftarrow \emptyset$
2: **while** $i \neq 0$ **do**
3: **if** $b[i] =$ N **then**
4: $i \leftarrow i - 1$
5: **else**
6: $S \leftarrow S \cup \{i\}$
7: $i \leftarrow p_i$
8: **return** $S$

## Subset Sum Problem

**Input**: an integer bound W > 0 and a set of n items, each with an integer value wi > 0

**Output:** a subset S items whose the sum of weights is nearest to W

## Define Subproblem

opt[i, W'] = the optimum weight of a set of instances at index i and capacity W'

This is a two dimensional DP (row is item index) (column is capacity parameter for W)

W' is used to judge whether the item at index i should be considered or not

if wi > W', we don't consider such instance (i) yet.

## Recurrence Relation

$$opt[i, W'] = \begin{cases} 0 & i = 0 \\ opt[i-1, W'] & i > 0, w_i > W' \\ \max \left\{ \begin{array}{c} opt[i-1, W'] \\ opt[i-1, W' - w_i] + w_i \end{array} \right\} & i > 0, w_i \leq W' \end{cases}$$

## Subset Sum Problem Algorithm (two dimensional DP)

```
for W' ← 0 to W do
    opt[0, W'] ← 0
for i ← 1 to n do
    for W' ← 0 to W do
        opt[i, W'] ← opt[i − 1, W']
        b[i, W'] ← N
        if w_i ≤ W' and opt[i − 1, W' − w_i] + w_i ≥ opt[i, W'] then
            opt[i, W'] ← opt[i − 1, W' − w_i] + w_i
            b[i, W'] ← Y
return opt[n, W]
```

### Recover the chosen items

```
1: i ← n, W' ← W, S ← ∅
2: while i > 0 do
3:     if b[i, W'] = Y then
4:         W' ← W' − w_i
5:         S ← S ∪ {i}
6:     i ← i − 1
7: return S
```

## Kanpsack Problem

**Input**: an integer bound W >0 a set of n items, each with an integer weight wi > 0 a value vi > 0 for each item I.

**Output**: finds the optimal set whose total weights are within W, and total values are maximized

### Define Subproblem

opt[i, W'] = the optimum value of a set of instances at index i and capacity W'

### Recurrence Relation

$$opt[i, W'] = \begin{cases} 0 & i = 0 \\ opt[i-1, W'] & i > 0, w_i > W' \\ \max \left\{ \begin{array}{c} opt[i-1, W'] \\ opt[i-1, W' - w_i] + v_i \end{array} \right\} & i > 0, w_i \leq W' \end{cases}$$

The only difference between Subset Sum Problem and Knapsack Problem is that instead of weight, we only consider the value of each item

## Longest Common Subsequence Problem

**Input**: A[1 .. n] and B[1 .. m]

**Output**: the longest common subsequence of A and B

## Common Subsequence

**Example:**

- $A =$ '$bacdca$'
- $B =$ '$adbcda$'
- $\text{LCS}(A, B) =$ '$adca$'

A common subsequence is basically a string sequence that is common in both of given inputs.

## Define Subproblem

opt[i, j] = the length of longest common sub-sequnce of A[i] and B[j]

## Recurrence Relation

$$opt[i,j] = \begin{cases} opt[i-1, j-1] + 1 & \text{if } A[i] = B[j] \\ \max \begin{cases} opt[i-1, j] \\ opt[i, j-1] \end{cases} & \text{if } A[i] \neq B[j] \end{cases}$$

When A[i] = B[j], we increment the length of common subsequence.

When A[i] != B[j], we take the previous instance that is larger.

## Longest Common Subsequence Algorithm

| | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|
| $A$ | b | a | c | d | c | a |
| $B$ | a | d | b | c | d | a |

```
1: for j ← 0 to m do
2:     opt[0, j] ← 0
3: for i ← 1 to n do
4:     opt[i, 0] ← 0
5:     for j ← 1 to m do
6:         if A[i] = B[j] then
7:             opt[i, j] ← opt[i − 1, j − 1] + 1, π[i, j] ← "↖"
8:         else if opt[i, j − 1] ≥ opt[i − 1, j] then
9:             opt[i, j] ← opt[i, j − 1], π[i, j] ←"←"
10:        else
11:            opt[i, j] ← opt[i − 1, j], π[i, j] ← "↑"
```

```
1: i ← n, j ← m, S ← ()
2: while i > 0 and j > 0 do
3:     if π[i, j] ="↖" then
4:         add A[i] to beginning of S, i ← i − 1, j ← j − 1
5:     else if π[i, j] ="↑" then
6:         i ← i − 1
7:     else
8:         j ← j − 1
9: return S
```

## Matrix Chain Multiplication

**Input**: n matrices A1,A2,⋯,An of sizes r1 ×c1,r2 ×c2,⋯,rn ×cn, such that ci = ri+1 for every i= 1,2,⋯,n−1.

**Output**: the order of computing A1A2⋯An with the minimum number of multiplications

**Fact:** Multiplying two matrices of size r×k and k×c takes r×k×c multiplications.

**Example:**

| **matrix** | $A_1$ | $A_2$ | $A_3$ | $A_4$ | $A_5$ |
|---|---|---|---|---|---|
| **size** | $3 \times 5$ | $5 \times 2$ | $2 \times 6$ | $6 \times 9$ | $9 \times 4$ |

## Define Subproblem

$opt[i, j]$ : the minimum cost of computing $A_i A_{i+1} \cdots A_j$

Assume the last step is $(A_1 A_2 \cdots A_i)(A_{i+1} A_{i+2} \cdots A_n)$

Cost of last step: $r_1 \times c_i \times c_n$

Optimality for sub-instances: we need to compute $A_1 A_2 \cdots A_i$ and $A_{i+1} A_{i+2} \cdots A_n$ optimally

## Recurrence Relation

$$opt[i, j] = \begin{cases} 0 & i = j \\ \min_{k:i \le k < j} \left( opt[i, k] + opt[k+1, j] + r_i c_k c_j \right) & i < j \end{cases}$$

## Matrix Chain Multiplication Algorithm

matrix-chain-multiplication$(n, r[1..n], c[1..n])$

```
1:  let opt[i,i] ← 0 for every i = 1,2,⋯,n
2:  for ℓ ← 2 to n do
3:      for i ← 1 to n − ℓ + 1 do
4:          j ← i + ℓ − 1
5:          opt[i,j] ← ∞
6:          for k ← i to j − 1 do
7:              if opt[i,k] + opt[k+1,j] + r_i c_k c_j < opt[i,j] then
8:                  opt[i,j] ← opt[i,k] + opt[k+1,j] + r_i c_k c_j
9:                  π[i,j] ← k
10: return opt[1,n]
```

The basic idea of this algorithm is that we intend to find the optimized cost matrix mulplication in between each set of mulplication order.

**line 9:** when we mark such k, that is the dividing point where the condition in line 7 ensures that the current order of mulplication was optimal, so that k provides that point.

## Optimum Binary Search Tree

**Def.** Binary search tree (BST), also called an ordered or sorted binary tree, is a rooted binary tree data structure with the key of each internal node being greater than all the keys in the respective node's left subtree and less than the ones in its right subtree.

**Input**: n elements e1 < e2 < e3 < · · · < en, n elements e1 < e2 < e3 < · · · < en

**Output**: build a binary search tree for {e1 , e2 , · · · , en } with the minimum accessing cost

- $\sum_{i=1}^{n} f_i \times (\text{depth of } e_i \text{ in the tree})$
- $C = C_L + C_R + \sum_{\ell=1}^{n} f_\ell$

In order to minimize C, need to minimize CL and CR respectively

### Define Subproblem

opt[i, j]: the optimum cost for the instance (fi , fi+1 , · · · , fj )

$$opt[1,n] = \min_{k:1\le k\le n} \left(opt[1,k-1] + opt[k+1,n]\right) + \sum_{\ell=1}^{n} f_\ell$$

### Recurrence Relation

In general, $opt[i,j] =$

$$\begin{cases} 0 & \text{if } i = j+1 \\ \min_{k:i\le k\le j} \left(opt[i,k-1] + opt[k+1,j]\right) + \sum_{\ell=i}^{j} f_\ell & \text{if } i \le j \end{cases}$$

### Optimum Binary Search Tree

```
1: fsum[0] ← 0
2: for i ← 1 to n do fsum[i] ← fsum[i − 1] + f_i
                                  ▷ fsum[i] = Σ_{j=1}^{i} f_j
3: for i ← 0 to n do opt[i + 1, i] ← 0
4: for ℓ ← 1 to n do
5:     for i ← 1 to n − ℓ + 1 do
6:         j ← i + ℓ − 1, opt[i, j] ← ∞
7:         for k ← i to j do
8:             if opt[i, k − 1] + opt[k + 1, j] < opt[i, j] then
9:                 opt[i, j] ← opt[i, k − 1] + opt[k + 1, j]
10:                π[i, j] ← k
11:        opt[i, j] ← opt[i, j] + fsum[j] − fsum[i − 1]
```

# Advanced Graph Algorithm - applications of algorithm techniques

## Minimum Spanning Tree Problems

**Def.** Given a connected graph G = (V, E), a spanning tree T = (V, F ) of G is a sub-graph of G that is a tree including all vertices V .

**Input**: Graph G= (V,E) and edge weights w: E → R

**Output**: the spanning tree T of G with the **minimum** total weight

## Spanning Tree T = (V, F)

- T is acylic and connected and has n − 1 edges.

- T is minimally connected: removal of any edge disconnects it.

- T has a unique simple path between every pair of nodes.

There are two classic greedy algorithms for MST: Kruskal's and Prim's Algorithm

## Kruskal Algorithm

### 1. Discuss Strategies

In order to find the lightest MST, it is safe to include the lightest edges in a specific order while avoiding any occurrence of a cycle. Preventing a cycle can be easily done through the Union-Find data structure. The Union-Find ADT allows us to check if several sets of nodes converge into the same root, meaning they are already in the same set, so choosing an edge between those two eventually causes a cycle to occur. We eventually aim to include all nodes into one same set to forge a MST

**Lemma**: when choosing an edge, it is safe to choose the lighest edge which doesn't converge into the same root from merging sets

### 2. Prove the safety

**Proof by exchange argument**

1. Let e = the lightest edge that connects two sets that don't have the same root in G

2. Say there exists an optimal solution S in which e is included, and produces an MST. If e is included in S produced by following the proposed lemma, the proof is complete

3. Otherwise, there is another output which chooses e' instead of e to connect the given sets

4. Since both e and e' are the lightest edge between the two sets, we can replace e' with e, connecting the two sets, which produces S'

5. Since S' includes e in its MST, symmetric to the MST of S. The lemma is accurate and valid

### 3. Prove the self-reducing

Selecting an edge leaves the remaining edges to be considered and the nodes that aren't yet connected via the Union-Find Structure.

## Union-Find Data Structure

Q: how can we check if u and v are in the same set?

A: Check if root(u) = root(v).

**root($v$)**

```
1: if par[v] = ⊥ then
2:     return v
3: else
4:     par[v] ← root(par[v])
5:     return par[v]
```

## Kruskal Algorithm

**MST-Kruskal($G$, $w$)**

```
1: F ← ∅
2: for every v ∈ V do: par[v] ← ⊥
3: sort the edges of E in non-decreasing order of weights w
4: for each edge (u, v) ∈ E in the order do
5:     u' ← root(u)
6:     v' ← root(v)
7:     if u' ≠ v' then
8:         F ← F ∪ {(u, v)}
9:         par[u'] ← v'
10: return (V, F)
```

**line 2**: Set each node as root of union-find

**line 3:** sort the edges to start from lightest edges

**line 5-7:** only choose an edge only if they don't have same root

**Runtime**: O(mlogn) for sorting

## Reverse Kruskal Algorithm

**1. Discuss Strategies**

The idea here is the same as Kruskal, but instead of choosing the lightest edges first, and we remove the heaviest edges from the entire graph. While we have previously used the Union-Find Structure to check whether two sets converge into the same root, this time we check whether the set doesn't diverge into different sets, meaning the graph will be disconnected if that happens.

**MST-Greedy($G, w$)**

```
1: F ← E
2: sort E in non-increasing order of weights
3: for every e in this order do
4:     if (V, F \ {e}) is connected then
5:         F ← F \ {e}
6: return (V, F)
```

## Prim's Algorithm (Greedy Algorithm)

### 1. Discuss Strategies

We will explore nodes one by one while re-adjusting the edge distances each time we choose a node. Upon choosing a node every time, we will calculate the distance of each edge from the chosen node to its neighbors. We will keep track of already-explored nodes in order to avoid re-considering them. Since each exploration chooses the lightest edge to a neighbor, we ensure we aim to forge an MST.

**Lemma:** when choosing an edge, it is safe to choose the lightest edge from u to v that doesn't involve the explored nodes.

### 2. Prove the safety

**Proof by exchange argument**

1. Let e = the lightest edge to a neighbor v that expands MST G

2. There exists an optimal solution S which includes e, and produces an MST. Then, if e is in the MST, since S is produced by following the proposed lemma, the proof is complete

3. Otherwise, there is another output that includes e' connecting u to v' instead of e connecting u to v.

4. Since both e and e' connect to a node that haven't been explored, we can replace e' with e, which produces S'

5. Since S' includes e in its MST, symmetric to S. The lemma is accurate and valid

### Prim's Algorithm

**MST-Prim($G, w$)**

```
1:  s ← arbitrary vertex in G
2:  S ← ∅, d(s) ← 0 and d[v] ← ∞ for every v ∈ V \ {s}
3:  Q ← empty queue, for each v ∈ V: Q.insert(v, d[v])
4:  while S ≠ V do
5:      u ← Q.extract_min()
6:      S ← S ∪ {u}
7:      for each v ∈ V \ S such that (u, v) ∈ E do
8:          if w(u, v) < d[v] then
9:              d[v] ← w(u, v), Q.decrease_key(v, d[v])
10:             π[v] ← u
11: return {(u, π[u]) | u ∈ V \ {s}}
```

- For every $v \in V \setminus S$ maintain
  - $d[v] = \min_{u \in S:(u,v) \in E} w(u, v)$: the weight of the lightest edge between $v$ and $S$
  - $\pi[v] = \arg\min_{u \in S:(u,v) \in E} w(u, v)$: $(\pi[v], v)$ is the lightest edge between $v$ and $S$

In every iteration

- Pick u ∈ V \ S with the smallest d[u] value. (extract_min)
- Add (π[u], u) to F.
- Add u to S, update d and π values. (decrease_key)

$O(n)\times$ (time for extract_min) + $O(m)\times$ (time for decrease_key)

| concrete DS | extract_min | decrease_key | overall time |
|---|---|---|---|
| heap | $O(\log n)$ | $O(\log n)$ | $O(m \log n)$ |
| Fibonacci heap | $O(\log n)$ | $O(1)$ | $O(n \log n + m)$ |

## Extension of MST algorithms: k-MST and k-MSF

### k-MST

**Input:** An undirected, connected graph G= (V,E) with edge weights w: E →R+.

An integer k such that 1 ≤k≤|V|

**Output**: A tree T ⊆E consisting of ≥k nodes.

The total cost e∈T w(e) is minimized.

**Goal**: Find a k-Minimum Spanning Tree (k-MST) of G.

Easy (i.e., poly-time solvable) when k or |V | − k is a constant.

NP-Hard when k is part of theinput.

If k = V, k-MST becomes a normal MST problem.

### Discussion

There is a particular drawback when using Kruskal to solve the k-MST problem: that is, Kruskal optimizes the cheapest tree that connects "all vertices". To do that, it may skip some edges that are "locally nice" (for k-MST) for some subset of vertices because there's a cheaper way to connect everyone globally. To solve k-MST, you need to optimize the "cheapest tree that connects some k vertices". The set of k vertices may want to use some edges that aren't in the global MST. As a result, k-MST is a NP problem when k is a part of input. To resolve this, we do

Ex: |V| = 6, k = 4

1. Find the number of combinations of a 4-vertex subset = C(6,4) = 15 sets
2. Run MST (prim's algorithm) on each set to find the cost of each set (the expensive part)
3. Compare the costs of each set to find the most optimal k-MST

Prim's algorithm can't solve the k-MST problem because it starts from an arbitrary vertex. The optimal solution of k-MST may not even consider that arbitrary vertex to be a part of its MST tree. Therefore, Prim is inappropriate even from the beginning.

### Runtime

If k is constant (fixed) → C(n, 4) → O(n^4 * mlogn) -> still poly.

If k is part of input → C(n, k) → not efficient.

### k-MSF

**Input**: An undirected, connected graph G= (V,E) with edge weights w: E →R+.

An integer k such that 1 ≤k≤|V|.

**Output**: A forest F ⊆ E consisting of ≤k connected components (trees).

The total cost e∈F w(e) is minimized.

**Goal**: Find a k-Minimum Spanning Forest (k-MSF) of G. **If k = 1, k-MSF becomes a normal MSF problem**

## Discussion

We can use the Kruskal algorithm to solve the k-MSF problem. We know that the Kruskal algorithm relies on the Union-Find Data Structure to check whether two sets converge to the same root. We can treat the number of sets in the structure as k sets in k-MSF. Since, in each set, we ensure only the lightest edges are chosen when formatting the sets, we can leverage that structure to find a k-MSF.

However, Prim's algorithm can't solve the k-MSF problem, since it constructs a single tree to find a globally optimal MST. Thus, it can't determine how many sets are currently forming, and it doesn't know which edge to remove to restore k sets from the complete MST.

## Runtime

Since the algorithm to solve k-MSF changes nothing from the formal Kruskal Algorithm, the runtime is identical to the Kruskal Algorithm.

## Shortest Path in DAGs

**Input**: directed acyclic graph G= (V,E) and w: E $\rightarrow$R. Assume V= {1,2,3$\cdots$,n} is topologically sorted: if (i,j) $\in$ E, then i<j.

**Output**: the shortest path from 1 to i, for every i $\in$ V.

## 1. Discuss strategies

Since we need to consider edges not only from u to v, but also from v to q (since the edge from v to q may be optimal), we need to leverage a dynamic programming scheme to cover all such cases (there may be more than one such occurrence). Therefore, we formulate a recurrence relation on account of that.

## 2. Recurrence Relation

f[i] = length of the shortest path from 1 to i

$$f[i] = \begin{cases} 0 & i = 1 \\ \min_{j:(j,i)\in E} \{f(j) + w(j,i)\} & i = 2, 3, \cdots, n \end{cases}$$

This is one dimentional DP

such that whichever vertex that i is, on expanding to each i where i $\in$ V, we will compute the minimum distance from 1 to i. Then, when the i finally reaches the end node, we can backtrace from i to 1, collecting the weight of each edge on the path.

## The Shortest Path in DAGs Algorithm

**Shortest Paths in DAG**

```
1: f[1] ← 0
2: for i ← 2 to n do
3:     f[i] ← ∞
4:     for each incoming edge (j, i) of i do
5:         if f[j] + w(j, i) < f[i] then
6:             f[i] ← f[j] + w(j, i)
7:             π(i) ← j
```

**line 2:** explore each node v

**line 3:** initially set inf for the length of shortest path

**line 5:** for each incoming edge j, compute opt[j] + w{j,i)

**line 5:** naturally set the lightest edge in the loop

**runtime**: O(n+m)
for each node, we check their edges

```
print-path(t)
1: if t = 1 then
2:     print(1)
3:     return
4: print-path(π(t))
5: print(",", t)
```

## Single-source shortest paths (Dijkstra Algorithm)

**Input**: (directed or undirected) graph G= (V,E), s,t∈V w: E $\rightarrow R_{\geq 0}$

**Output**: shortest paths from s to all other vertices v∈V

## Observations of a weighted edge shortest path problem

Reason for considering the single-source shortest paths problem:

We do not know how to solve s-t shortest path problem more efficiently than solving single source shortest path problem

Undirected Graph <-> Directed Graph

Shortest paths in directed graphs are more general than in undirected graphs, since if given an undirected graph, we can replace each edge with two antiparallel edges and run the SSSP algorithm to solve the problem.

Studies in Single-source shortest paths

Assuming the weight of all the edges in graph G is 1, we can easily compute the shortest path from s to each v via BFS. Based on this observation, we can consider each weighted edge as a unit-

weight edge, meaning that if w(u,v) = 4, then we can slice it into pieces to form a unit-weight edge 1 + 1 + 1 + 1 = 4. Thus, we can use BFS to solve the weighted shortest path problem, with a little tweak to the problem's output definition. Therefore, we formally redefine the input and output of the **Single-source shortest paths** based on the observations:

**Input**: directed graph G= (V,E), s∈V w: E $\to R_{\geq 0}$

**Output**:

π[v],v∈V \ S: the parent of v in shortest path tree

d[v],v∈V \ S: the length of shortest path from s to v

## Shortest Path Algorithm by Running BFS

**Shortest Path Algorithm by Running BFS**

1: replace $(u, v)$ of length $w(u, v)$ with a path of $w(u, v)$ unit-weight edges, for every $(u, v) \in E$
2: run BFS virtually
3: $\pi[v] \leftarrow$ vertex from which $v$ is visited
4: $d[v] \leftarrow$ index of the level containing $v$

Since w(u, v) may be too large, we now wish to consider taking each unit-weight edge as a single weighted edge. Furthermore, we need to factor in cases where there may be multiple paths to v, so we want to choose the optimal one among them. Therefore, we further develop the algorithm that turns into

**Shortest Path Algorithm by Running BFS Virtually**

1: $S \leftarrow \{s\}, d(s) \leftarrow 0$
2: **while** $|S| \leq n$ **do**
3: find a $v \notin S$ that minimizes $\min_{u \in S:(u,v) \in E} \{d[u] + w(u, v)\}$
4: $S \leftarrow S \cup \{v\}$
5: $d[v] \leftarrow \min_{u \in S:(u,v) \in E} \{d[u] + w(u, v)\}$

Finally, this formally defines the Dijkstra Algorithm

## Dijkstra Algorithm Analysis

**Dijkstra$(G, w, s)$**

1: $S \leftarrow \emptyset, d(s) \leftarrow 0$ and $d[v] \leftarrow \infty$ for every $v \in V \setminus \{s\}$
2: **while** $S \neq V$ **do**
3: $u \leftarrow$ vertex in $V \setminus S$ with the minimum $d[u]$
4: add $u$ to $S$
5: **for** each $v \in V \setminus S$ such that $(u, v) \in E$ **do**
6: **if** $d[u] + w(u, v) < d[v]$ **then**
7: $d[v] \leftarrow d[u] + w(u, v)$
8: $\pi[v] \leftarrow u$
9: **return** $(d, \pi)$

The current algorithm's runtime is O(n^2) because it must find the vertex with the minimum distance each loop in line 3. We can improve this by using the Priority Queue which can reduce the time to O(mlogn).

**O(mlogn) = O(n) x (time for extract-min) + O(m) x (time for decrease key)**

**Dijkstra$(G, w, s)$**

```
1:  s ← arbitrary vertex in G
2:  S ← ∅, d(s) ← 0 and d[v] ← ∞ for every v ∈ V \ {s}
3:  Q ← empty queue, for each v ∈ V: Q.insert(v, d[v])
4:  while S ≠ V do
5:      u ← Q.extract_min()
6:      S ← S ∪ {u}
7:      for each v ∈ V \ S such that (u, v) ∈ E do
8:          if d[u] + w(u, v) < d[v] then
9:              d[v] ← d[u] + w(u, v), Q.decrease_key(v, d[v])
10:             π[v] ← u
11: return (π, d)
```

## Shortest Paths in Graphs with Negative Edges (Bellmon Ford Algorithm)

**Input**: directed graph G= (V,E), s∈V assume all vertices are reachable from s w: E →R

**Output**: shortest paths from s to all other vertices v∈V

### Discussion

A problem with Dijkstra is that it can't solve the negative weighted shortest path problem since it greedily prioritizes locally lightest edges in search of shortest path (it is possible locally heavier edge leads to a negative edge that reduces total path weight). We need to inspect paths globally in order to consider an optimal solution that may contain negative edges. Therefore, we need a dynamic programming strategy.

Dealing with negative cycles

Assume the input graph doesn't contain negative cycels, or Allow the algorithm to report "negative cycle exists".

Unfortunately, computing the shortest simple path between two vertices in a graph that has negative cycles is an NP-hard problem

### Recurrence Relation

Why can't simple f[v] work? First of all, since Dijkstra can't work, we can't greedily pursue the lightest edges, as we have to consider negative edges, we have to use a dynamic approach. Then, we need some notion of the degree at which each edge reaching v is because, whereas in Dijkstra's we didn't have to worry about this since we are always going for the lightest edges, we don't know which edge the shortest path will come from. Therefore, based on the degrees of edges reaching v, we need to distinguish them. As a result, we construct the following recurrence relation:

$$f^{\ell}[v] = \begin{cases} 0 & \ell = 0, v = s \\ \infty & \ell = 0, v \neq s \\ \min \left\{ \begin{array}{c} f^{\ell-1}[v] \\ \min_{u:(u,v)\in E} \left(f^{\ell-1}[u] + w(u,v)\right) \end{array} \right. & \ell > 0 \end{cases}$$

$f^{\ell}[v]$ = length of the shortest path from s to v that uses at most $\ell$ edges

$f^{\ell-1}[u] + w(u,v)$ => adding weight here essentially means adding one more edge to the currently considered edges.

## Bellmon Ford Algorithm Analysis

**dynamic-programming$(G, w, s)$**

1: $f^0[s] \leftarrow 0$ and $f^0[v] \leftarrow \infty$ for any $v \in V \setminus \{s\}$
2: **for** $\ell \leftarrow 1$ to $n-1$ **do**
3: copy $f^{\ell-1} \rightarrow f^{\ell}$
4: **for** each $(u,v) \in E$ **do**
5: **if** $f^{\ell-1}[u] + w(u,v) < f^{\ell}[v]$ **then**
6: $f^{\ell}[v] \leftarrow f^{\ell-1}[u] + w(u,v)$
7: **return** $(f^{n-1}[v])_{v \in V}$

### Observations

- Assuming there are no negative cycles, then a shortest path contains at most n − 1 edges
- If there is a path containing at least n edges, then it contains a cycle.
- $f^{\ell}$ only depends on $f^{\ell-1}$ : only need 2 vectors

**dynamic-programming$(G, w, s)$**

1: $f^{old}[s] \leftarrow 0$ and $f^{old}[v] \leftarrow \infty$ for any $v \in V \setminus \{s\}$
2: **for** $\ell \leftarrow 1$ to $n-1$ **do**
3: copy $f^{old} \rightarrow f^{new}$
4: **for** each $(u,v) \in E$ **do**
5: **if** $f^{old}[u] + w(u,v) < f^{new}[v]$ **then**
6: $f^{new}[v] \leftarrow f^{old}[u] + w(u,v)$
7: copy $f^{new} \rightarrow f^{old}$
8: **return** $f^{old}$

**line 2:** for loop l <- 1 to n-1 ensures we reach the last node.

**line 3:** allocates space for $f^{new}$

**line 4:** check every (u,v) comparing $f^{old}$ with $f^{new}$

**line 7:** ensures $f^{old}$ is always updated by $f^{new}$

Instead of keeping all $f^{\ell->n-1}$, we have kept space only for $f^{old}$ and $f^{new}$ to minimize space. However, since we need both $f^{old}$ and $f^{new}$ only to compare them within the inner for loop, we can reduce the space even further by using only 1 vector, which gives an optimized version as below:

**dynamic-programming$(G, w, s)$**

1: $f[s] \leftarrow 0$ and $f[v] \leftarrow \infty$ for any $v \in V \setminus \{s\}$
2: **for** $\ell \leftarrow 1$ to $n-1$ **do**
3: copy $f \rightarrow f$
4: **for** each $(u,v) \in E$ **do**
5: **if** $f[u] + w(u,v) < f[v]$ **then**
6: $f[v] \leftarrow f[u] + w(u,v)$
7: copy $f \rightarrow f$
8: **return** $f$

**Bellman-Ford$(G, w, s)$**

1: $f[s] \leftarrow 0$ and $f[v] \leftarrow \infty$ for any $v \in V \setminus \{s\}$
2: **for** $\ell \leftarrow 1$ to $n-1$ **do**
3: **for** each $(u,v) \in E$ **do**
4: **if** $f[u] + w(u,v) < f[v]$ **then**
5: $f[v] \leftarrow f[u] + w(u,v)$
6: **return** $f$

In this way, f[v] in line 5 may change the value before the comparison, but this is okay since it only accelerates the algorithm (this would require careful ordering of edges to explore in line 4, since

later edges depend on closer edges to update their paths). Then, we formally define this dynamic strategy algorithm as the Bellman-Ford Algorithm.

**Bellman-Ford**$(G, w, s)$

1: $f[s] \leftarrow 0$ and $f[v] \leftarrow \infty$ for any $v \in V \setminus \{s\}$
2: **for** $\ell \leftarrow 1$ to $n$ **do**
3: $\quad updated \leftarrow$ false
4: $\quad$ **for** each $(u, v) \in E$ **do**
5: $\quad\quad$ **if** $f[u] + w(u, v) < f[v]$ **then**
6: $\quad\quad\quad f[v] \leftarrow f[u] + w(u, v)$, $\pi[v] \leftarrow u$
7: $\quad\quad\quad updated \leftarrow$ true
8: $\quad$ if not $updated$, then return $f$
9: output "negative cycle exists"

For improvement, we can halt the algorithm once we reach an "unrelaxable" point. This is the point where no further iteration won't find any better path than the current iteration. This is because further iterations will depend on the previous iteration in search of a better path. If the previous iteration has reached an "unrelaxable" point, meaning no more for better path discovery, it is determined that no further iteration will discover any better path. New node discovery is impossible since we iterate all pairs of edges in each iteration, and no further iteration is meaningful since each iteration depends on the previous iteration.

**Runtime**: O(nm) = line2 * line4

π[v]: the parent of v in the shortest path tree

## All-pairs Shortest paths and Floyd-Warshall

**Input**: directed graph G= (V,E), w: E →R (can be negative)

**Output**: shortest path from u to v for every u,v∈V -> meaning not only we know all shortest paths for s to v, we know all shortest paths from every u to v,

## Discussion

The simple idea is that we intend to find the shortest paths not only from s to all other v, but also from every u to all other v. Therefore, this is like we want to execute the Bellman-Ford algorithm on every v. We know that we get the shortest paths from s to v in the last row in the Bellman-Ford algorithm. Therefore, it's like we are constructing a matrix of vertices where we can find the shortest path from any u to any other v. This leads us to create a dynamic table.

## Recurrence Relation

$$w(i,j) = \begin{cases} 0 & i = j \\ \text{weight of edge } (i,j) & i \neq j, (i,j) \in E \\ \infty & i \neq j, (i,j) \notin E \end{cases} \qquad f^k[i,j] = \begin{cases} w(i,j) & k = 0 \\ \min \begin{cases} f^{k-1}[i,j] \\ f^{k-1}[i,k] + f^{k-1}[k,j] \end{cases} & k = 1, 2, \cdots, n \end{cases}$$

$f^k[i,j]$: length of the shortest path from i to j that only uses vertices {1,2,3,⋯,k} as intermediate vertices.

Since we now intend to run a dynamic programming algorithm, and we want to achieve data for all pair shortest paths, we now consider the number of intermediate vertices instead of the number of edges. In this way, we can create a matrix of vertices, filling it in with the shortest paths discovered. The w(i, j) may seem minor, but it plays a critical role. Since k=0 means there is no intermediate vertex, we can construct base matrix weights for all direct edges of $f^0[i,j]$.

## Floyd-Warshall algorithm

Floyd-Warshall$(G, w)$

```
1: f^old ← w
2: for k ← 1 to n do
3:     copy f^old → f^new
4:     for i ← 1 to n do
5:         for j ← 1 to n do
6:             if f^old[i,k] + f^old[k,j] < f^new[i,j] then
7:                 f^new[i,j] ← f^old[i,k] + f^old[k,j]
```

Floyd-Warshall$(G, w)$

```
1: f ← w
2: for k ← 1 to n do
3:     for i ← 1 to n do
4:         for j ← 1 to n do
5:             if f[i,k] + f[k,j] < f[i,j] then
6:                 f[i,j] ← f[i,k] + f[k,j]
```

Line 1 is basically constructing the base matrix weights for $f^0[i,j] = w(i,j)$.

We can reduce space from left to right since we actually don't need two vector matrices.

The core idea here is that we compare all possible paths from u to v incrementally from 1 to n. This is why we need an assumption that V = {1,2,3,...,n}.

**Lemma** Assume there are no negative cycles in G. After iteration k, for i, j ∈ V , f [i, j] is exactly the length of shortest path from i to j that only uses vertices in {1, 2, 3, · · · , k} as intermediate vertices.

**runtime**: $O(n^3)$

## Recovering Shortest Paths and Detecting Negative Cycles

**Floyd-Warshall$(G, w)$**

```
1:  f ← w, π[i, j] ← ⊥ for every i, j ∈ V
2:  for k ← 1 to n do
3:      for i ← 1 to n do
4:          for j ← 1 to n do
5:              if f[i, k] + f[k, j] < f[i, j] then
6:                  f[i, j] ← f[i, k] + f[k, j], π[i, j] ← k
7:  for k ← 1 to n do
8:      for i ← 1 to n do
9:          for j ← 1 to n do
10:             if f[i, k] + f[k, j] < f[i, j] then
11:                 report "negative cycle exists" and exit
```

**print-path$(i, j)$**

```
1:  if π[i, j] = ⊥ then
2:      print(i, j)
3:  else
4:      print-path(i, π[i, j]), print-path(π[i, j], j)
```

# NP-Completeness

We study hard (NP) problems here because we want to know which problems aren't solvable in polynomial time, so that we can avoid wasting time solving them more efficiently in advance.

## Reason for efficient algorithms = polynomial time

We call algorithms that run in polynomial time to be efficient. By nature, solving problems usually outputs either a polynomial runtime algorithm or an exponential runtime algorithm. If some algorithm gives an exponential form runtime like $O(2^n)$, n is usually very small or very large. Thus, we label the algorithms in the lower bound realm as efficient.

## Some Hard Problems

### Hamiltonian Cycle Problem (NP hard)

**Def**. Let G be an undirected graph. A Hamiltonian Cycle (HC) of G is a cycle C in G that passes each vertex of G exactly once.

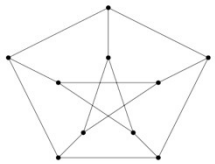
**Input**: graph G = (V, E)

**Output**: whether G contains a Hamiltonian cycle

#### Discussion

You need to explore the graph in a permutative way, inclusive of the starting node. You explore every possible path from v, v+1, and v+n to the rest of the vertices. This means if you have 4 nodes, then there are 4! paths to explore. This is because *some path is not a hamiltonian cycle != some path doesn't have a hamiltonian cycle in G.* Therefore, there are n! possible paths you may have to traverse. Then, each traverse may take O(n). Thus, the final runtime is $O(n!m) = 2^{O(n\log n)}$. Furthermore, a better runtime algorithm is known to be $2^{O(n)}$

If you are given a graph G and a Hamiltonian path t (certificate) in G, it merely takes O(n) for you to check its validity.



Some graph, like the **Petersen Graph** shown in the left picture, is known not to have any Hamiltonian path.

### Maximum Indenpendent Set Problem (NP Hard)

Def. An independent set of G = (V, E) is a subset $I \subseteq V$ such that no two vertices in I are adjacent in G.

**Input**: graph G = (V, E)

**Output**: the size of the maximum independent set of G

#### Discussion

An independent set of G is also a graph that contains vertices that construct the largest graph in which no two vertices are adjacent to each other.

For this problem, you also need to explore the graph in a permutative way, inclusive of the starting node. It is like you first try to compute the largest independent set from node v. Then, you don't know if starting from v would give you the largest independent set in the entire graph, so you need to explore all the paths from all other v as well. Traversing each path takes O(n), and there are total n! paths to explore. Therefore, the runtime is O(n!m) = $2^{O(nlogn)}$.

When you are given a graph G and the maximum independent set (certificate vertices with <= k), it takes you $O(n^2)$ to check the validity because you need to make sure that the independent set property is not violated from each vertex.

## Formula Satisfiability (NP Hard)

**Input**: Boolean formula with n variables, with ∨, ∧, ¬ operators.

**Output**: whether the Boolean formula is satisfiable

### Discussion

If a formula returns true for some assignment of variables, the formula is satisfiable. You have to check every assignment of variables because you can't be sure a formula wouldn't be satisfiable until you try all possible assignments of variables to the end. If there are 3 variables, there are $2^3$ possible assignments for true/false. Since the number of assignments increases exponentially, having to check every assignment will give an exponential runtime. When an answer is given, you can check it in polynomial time

### Optimization Problem to decision problem

For any given optimization problem, we can always transform it into a decision problem. When a problem is given, and asked to define a problem, we always mean to define a decision version of that problem. A decision problem always outputs yes or no, (1, 0). If that decision problem can be solved in polynomial time, that means the optimization version of that problem can also be solved in polynomial time.

**Shortest Path**
**Input:** graph $G = (V, E)$, weight $w, s, t$ and a bound $L$
**Output:** whether there is a path from $s$ to $t$ of length at most $L$

**Maximum Independent Set**
**Input:** a graph $G$ and a bound $k$
**Output:** whether there is an independent set of size at least $k$

These are the examples when optimization problems are given (input), we convert them into decision version of that problem

### In NP-completeness, we only focus on decision problems in defining P and NP

### The Complexity Class P

**Def.** The complexity class P is the set of decision problems X that can be solved in polynomial time

### Certificate and Certifier

Alice has a supercomputer, fast enough to run the $2^{O(n)}$ time algorithm for Ind-Set

Bob has a slow computer, which can only run an $O^{(n3)}$ time algorithm

Q: Given graph G = (V, E) and integer k, such that there is an independent set of size k in G, how can Alice

convince Bob that such a set exists?

A: Alice gives a set of size k to Bob, and Bob checks if it is really an independent set in G.

**Certificate**: a set of polynomial size k

**Certifier**: a polynomial algorithm that checks if the given set is really an independent set

## The Complexity Class NP

**Def**. The complexity class NP is the set of all problems for which there exists an **efficient certifier**.

**An efficient certifier:** B is an efficient certifier for a decision problem X if

- B is a polynomial-time algorithm that takes input strings s (input) and t (certificate), and outputs 0 or 1

- B has some polynomial function p that solves the problem X in at most p(|s|) steps for every string s, such that X(s) = B(s,t) = 1 iff there is a string polynomial certificate t such that |t| <= p(|s|) for some polynomial function p.

Therefore, to prove that some problem X is in NP, we prove that there exists an efficient certifier such that it can verify X(s) = B(s, t) for any given s and t.

## Encoding and defining a problem as a function

Encoding: the **input of every problem** is converted into a binary string, denoted as {0, 1}*, that has a length |s|. Whether it is an optimization problem or a decision problem, this input string, based on some encoding, will be passed to the solving algorithms.

## Defining a problem as a function and the reasons why we only consider decision problems

Suppose an arbitrary (whether NP or P) optimization problem P is given with an input string |s|. We first convert that into a decision problem X. Then, we design an algorithm A that takes the input s and solves the problem X (whether NP or P) that outputs {0, 1}. If the algorithm A runs in exponential time and gives 1, we want to convince others that the problem X indeed outputs 1. We share the input s, problem X, and a certificate t with others. They have a polynomial algorithm B that takes the input s and a polynomial certificate t, which outputs A(s) = B(s, t) for every string s. If such B exists that verifies problem X in polynomial time, problem P is proved to be in NP.

## The Complexity Class Co-NP

**Def**. For a problem X, the problem neg X is the problem such that neg X(s) = 1 if and only if X(s) = 0.

**Def**. Co-NP is the set of decision problems X such that neg X $\in$NP.

Such that NP certificer can't answer for no-instance

## Tautology Problem

**Def**. A tautology is a Boolean formula that always evaluates to 1

**Input**: a boolean formula

**Output**: whether the formula is a tautology

Tautology Problem is a Co-NP because it is impossible to share the problem X and a certificate t such that a polynomial time certifier B can be convinced that the problem X (input s always outputs 1) is indeed true by verifying only one instance certificate t indeed outputs 1 since B on its own can't run an exponential time algorithm A to check if other instances may or may not output 0. For such a problem, we call it Co-NP.

To prove that the Tautology problem is Co-NP, you can prove that neg X is NP with a certifier (verifying that no instance is true such that neg X(s) = 1), or prove X (s) = 0, which is equivalent to neg X(s) = 1.

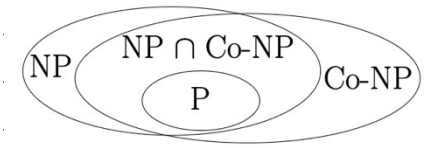
### P ⊆ NP ∩ Co-NP

Let problem X ∈ P and X(input s) = 1
Since X ∈ P, the certifier can check whether X(s) = 1 without the algorithm A's help
The certificate t is an empty string in this case
Thus, problem X ∈ NP, and P ⊆ NP

### A problem can be both in NP and Co-NP

If X is in P and NP, that is also Co-NP since for a yes-instance, the certifier doesn't need a certificate and just can run the poly-time algorithm to confirm X(s) = 1.

For example, for a shortest path problem P, we define X: "Is there a shortest path from s to t? Then, to prove X is in NP, the certifier B can run their algorithm to verify that there exists a path from s to t. Furthermore, if we also want to know if X is in Co-NP, we define neg X: "There exists no path from s to t", which the certifier can also verify on their own that there in fact exists a path from s to t.




NP ⊆ Co-NP is unknown

People commonly believe we are in the scenario

### Polynomial Time Reductions

We want polynomial-time reductions because we want to reduce the number of problems to a similar set of problems. Therefore, we can treat a set of problems as a bundle such that, for positive results, if you show Y ≤p X (Y is poly-time reduciable to X), and later discover X is in P, then Y is in P; for negative results, if you show a problem Y reduces in poly-time to a known hard problem X, then Y must be at least as hard as X.

If a black box algorithm A can solve both X and Y in a poly-time, that is, Y is polynomial-time reducible to X.

### HP ≤p HC

Assume that HP ≤p HC, then if we solve HC in an exponential time, that is, HP can also be solved with the same algorithm in an exponential time.

Step 1: valid instance polynomial construction Y -> X (so you need to make X from Y)

Step 2: we need to show two-directional proof that Y is yes if and only if X is yes

## NP Complete

**Def.** A problem X is called NP-complete if

1. X ∈NP, and

2. Y ≤p X for every Y ∈ NP.

## NP Hard

**Def.** A problem X is called NP-hard if

1. Y ≤p X for every Y ∈ NP, but X may or may not be in NP ~~(this means NP-hard can be "beyond NP-complete")~~

NP-hard problems are the hardest problems in NP

If NP hard problem is also in NP, that means that the problem is in NP-complete

NP-complete = NP-hard + in NP. If NP-hard problem can reduced to another NP-hard, that is NP-complete

## First NP-Complete Problem: Circuit-Sat

**Input**: a circuit

**Output**: whether the circuit is satisfiable

Fact: Any problem Y that takes n bits as input and outputs 0/1 with running time T(n) can be converted into a circuit problem X of size p(T(n)) for some poly-function p.

## HC (and all NP) ≤p Circuit-Sat (Proving that CS is in NP-complete by reducing check-Y to a Circuit-Sat instance)

Assume we have check-Y certifier function that takes a graph G, and a certificate S. G is a yes-instance iff check-HC(G, S) returns 1 for a valid S. We know every NP problem has a polynomial certifier Y

Then, we construct a circuit-sat problem C' by reducing check-Y. Then, we can hard-wire the inputs G and S into C' to obtain the circuit C.

Then, G is a yes-instance iff C is satisfiable

Since every NP has such a Y, every NP problem is reducible to the Circuit-Sat problem.

Therefore, Circuit-Sat is NP-complete

## 3-Sat Problem

**Input**: a 3-CNF formula

**Output**: whether the 3-CNF is satisfiable

To satisfy a 3-CNF, we need to satisfy all clauses

3-CNF (conjunctive normal form) is a special case of a formula

- boolean variables: x1, x2, ..., xn

- literals: Xi or neg Xi

- Clause: disjunction ("or") of at most 3 literals $x_3 \vee \neg x_4$, $x_1 \vee x_8 \vee \neg x_9$,

Finally, 3-CNF formula is conjuction of clauses

## Circuit-Sat ≤p 3-Sat (proving 3-Sat is in NP-complet

First, we know that Formula Satisfiability is in NP.

Then, we construct the 3-Sat problem by transforming Circuit-Sat such that

1. Associate every wire with a new variable, which gives the following formula, like the diagram on the right.

$$(x_4 = \neg x_3) \wedge (x_5 = x_1 \vee x_2) \wedge (x_6 = \neg x_4) \wedge (x_7 = x_1 \wedge x_2 \wedge x_4) \wedge (x_8 = x_5 \vee x_6) \wedge (x_9 = x_6 \vee x_7) \wedge (x_{10} = x_8 \wedge x_9 \wedge x_7) \wedge x_{10}$$

2. Then, in order to construct a correct 3-Sat version problem of the Circuit-Sat problem, we need to create a truth table for each new clause so that we can correctly create a new 3-CNF that respects that new clause, since now we need to care about the new variable values altogether to output the correct result

Convert each clause to a 3-CNF

$x_5 = x_1 \vee x_2 \quad \Leftrightarrow$

$(x_1 \vee x_2 \vee \neg x_5) \quad \wedge$

$(x_1 \vee \neg x_2 \vee x_5) \quad \wedge$

$(\neg x_1 \vee x_2 \vee x_5) \quad \wedge$

$(\neg x_1 \vee \neg x_2 \vee x_5)$

| $x_1$ | $x_2$ | $x_5$ | $x_5 \leftrightarrow x_1 \vee x_2$ |
|---|---|---|---|
| 0 | 0 | 0 | 1 |
| 0 | 0 | 1 | 0 |
| 0 | 1 | 0 | 0 |
| 0 | 1 | 1 | 1 |
| 1 | 0 | 0 | 0 |
| 1 | 0 | 1 | 1 |
| 1 | 1 | 0 | 0 |
| 1 | 1 | 1 | 1 |

To construct a new 3-CNF form, we find the "bad" assignments that output 0, which in this transformed problem means "unsatisfiable". Thus, to make sure that "unsatisfiable" assignment is taken into account, we create a clause that "rejects" such assignments, thus taking into account those "unsatisfiable" assignments will actually output 0 on the new 3-CNF form.

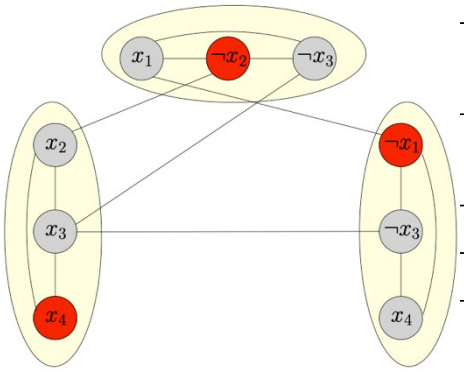
3. Now we have transformed a Circuit Problem into a 3-Sat problem, and at this point, we can prove the circuit is satisfiable if and only if the 3-CNF is satisfiable, and vice versa. We can do this by comparing the original 3-Sat with the new 3-Sat.

**Thus, Circuit-Sat $\leq$p 3-Sat is true, and 3-Sat is NP-complete**

**3-Sat $\leq$p Independent Set Problem** (proving Ind-Set is in NP-complete by reducing 3-Sat)

**Step1 (construction)**

1. We reduce 3-Sat problem into an independent set problem.
2. For each clause in 3-Sat, we create a group of vertices, each for a literal, and connect them all in each group.
3. Across groups, we connect contradicting literals.
4. for size k, k = # of clauses




Then, now we have a transformed instance of an Ind-set problem converted by a 3-Sat

**Step 2:**

Then, we show that a satisfying assignment leads to an independent set of size k.

We also show that an independent set of size k leads to a satisfying assignment.

**3-Sat $\leq$p Independent Set Problem is true, and Independent Set Problem is NP-complete**

**Clique Problem**

Clique is a complete subgraph in which every vertex is connected with each other

**Input**: G = (V, E) and integer k > 0,

**Output**: whether there exists a clique of size k in G

**Ind-Set $\leq$p Clique Problem**

**Relationship between Clique and Ind-Set**

**Def**. Given a graph G = (V, E), define neg G = (V, E) be the graph such that (u, v) ∈E if and only if (u, v) / ∈E.

such that neg G won't have any edges that in which G has

Obs. S is an independent set in G if and only if S is a clique in neg G.

### Step1 (construction)

Create the neg G by removing all edges and connectings once-unconnected edges from G

### Step2 (only if)

Show if S is an independent set in G if and only if S is a clique in neg G, and

vice versa

### Vertex Cover Problem

For every edge, there is some vertex that covers the entire edge in G.

If you choose all vertices, that is also a vertex cover.

**Input**: G = (V, E) and integer k

**Output**: whether there is a vertex cover of G of size at most k

### Ind- Set ≤p Vertex Cover Problem

### Relatioship between Vertex-Cover and Ind-Set

S is a vertex-cover of G = (V, E) if and only if V \S is an independent set of G.

### Step1 (construction)

With the same graph G', choose v - k vertices that originally formed an ind-set in G

### Step 2 (only if)

Show S in G is an indenpedent graph in G if and only if V \S is a vertex cover in G'

Show S is a vertex cover in G' if and only if V \S is an indenpendent set in G.

### k-Coloring Problem (won't be tested on exam)

**Def**. A k-coloring of G = (V, E) is a function f : V →{1, 2, 3,···, k} so that for every edge (u, v) ∈E, we have f (u)≠ f (v). G is k-colorable if there is a k-coloring of G.

if k = 1, when there is no edge, it is colorable -> p

if k = 2, when graph is bipartite, it is colorable -> p

if k = 3, it becomes a NP-hard

### A strategy of Polynomial Reduction

### Dealing with NP-Hard Problems

Faster exponential time algorithms
Solving the problem for special cases
Fixed parameter tractability
Approximation algorithms