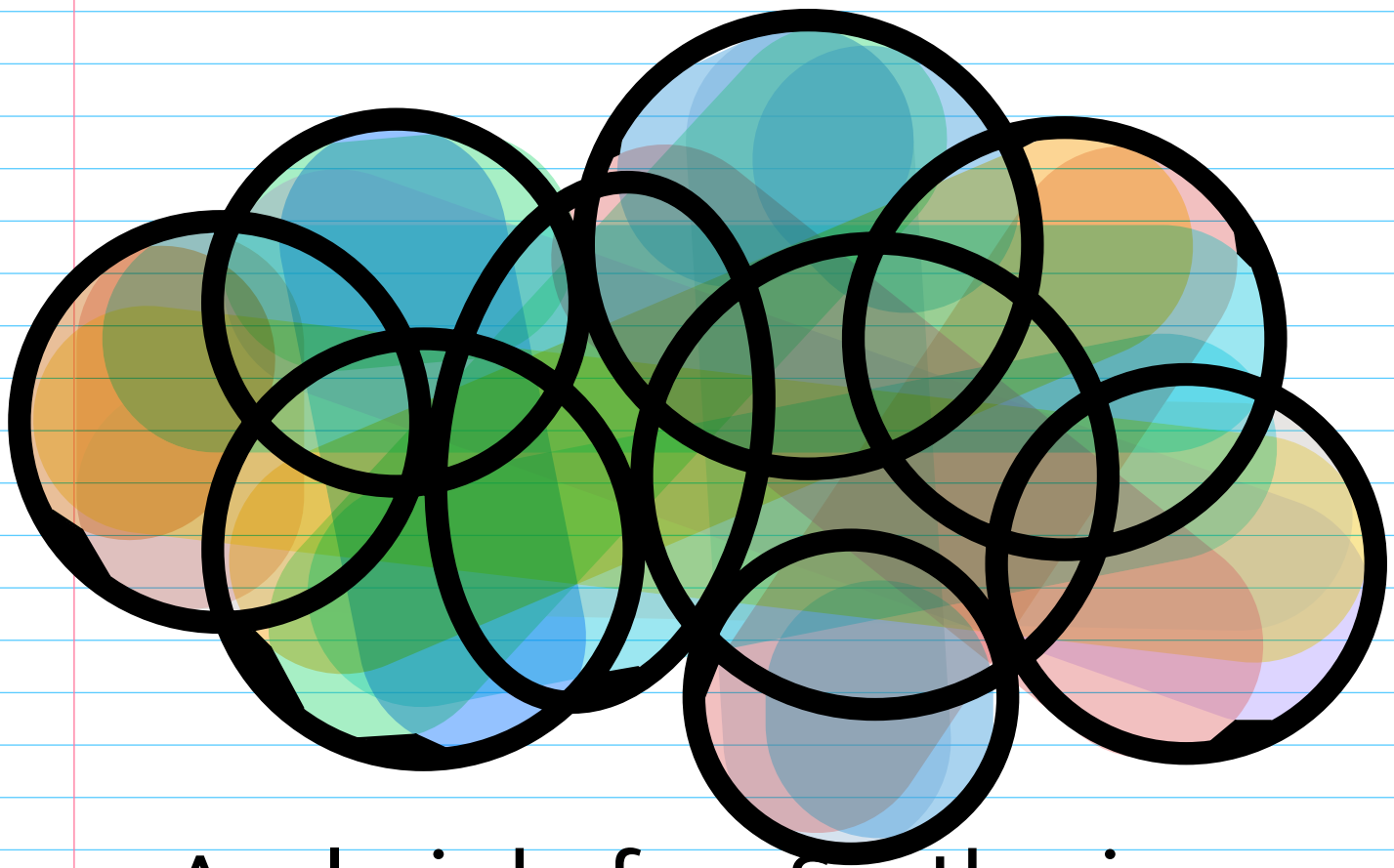
CS 180



Analysis before Synthesis

CS-180 Notes:

Notes for Big O, Ω, Θ are on the online google docs.

Converting from code to Big O.

```
for (i=1 ; i<N; i++)
  for(j=1;j<N;j++)
    work
```

$\Rightarrow \sum_{i=1}^{N} \sum_{j=1}^{N} 1 \Rightarrow N^2$

```
for (i=1 ; i<N; i++)
  for(j=1;j<I;j++)
    work
```

$\Rightarrow \sum_{i=1}^{N} \sum_{j=1}^{i} 1 \Rightarrow \sum_{i=1}^{N} i \Rightarrow \frac{n(n+1)}{2}$ (1+2+3+...+N)

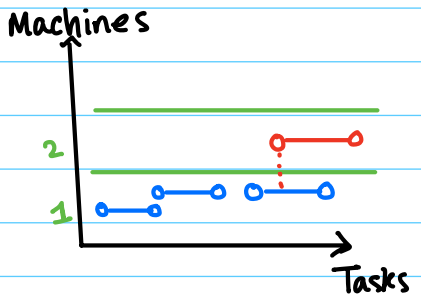
Greedy Method: Style: iteration ; Beginning → End

- We process input in some order, and make progress towards the final result using only the information we have at some point. Typically there is no backtracking or undoing of progress.

- An algorithm is greedy if it builds up a solution in small steps choosing a decision at each step myopically to optimize some underlying criterion.

(myopically: Near sighted)

Interval Task Partitioning Problem:



We have N tasks each has start and finish two tasks I, J conflict if Start[i] ≤ start[J] and start[j] < Finish[i]

Output: Minimum number of machines needed to reschedule all N tasks, and an assignement of each task to exactly one machine.

In any instance of interval partioning, the number of resources needed is at least the depth of the set of intervals. Where the depth is where a set of intervals pass over a common point on the timeline.

Algorithm:

```
Sort the intervals by their start times, breaking ties arbitrarily
Let I_1, I_2, ..., I_n denote the intervals in this order
For j = 1, 2, 3, ..., n
  For each interval I_i that precedes I_j in sorted order and overlaps it
     Exclude the label of I_i from consideration for I_j
  Endfor
  If there is any label from {1, 2, ..., d} that has not been excluded then
    Assign a nonexcluded label to I_j
  Else
    Leave I_j unlabeled
  Endif
Endfor
```

Proof by Induction ⇒ helps prove correctness of iterations and sequences

① Base Case: holds for "first" case
② Inductive Step: if it holds for n it is true for n+1.

Proof by Contradiction: theorem has form "p ⇒ q". Logically that is equivalent to the contrapositive "¬q ⇒ ¬p" ¬ ⇒ not symbol

---

Ex#1 Classroom Scheduling:

We have n classes, 1 classroom.
goal: Choose classes to maximize # of classes in classroom.

Pseudocode:

- Input : s, f, T (start, finish, time limit)

```
s,f = sortByFinishTimes(s,f)   # nlog(n)
```

⇒ The best Greedy Algorithm is to sort based on finish times

```
chosen_classes = []   # → linked list
last_f = 0;

for i in range(len(f)):   # n
    ★ if( s[i] ≥ last_f ):
        chosen_classes.append(names[i])
        last_f = f[i]

return chosen_classes;
```

Net complexity : nlog(n)

Proof of Correctness:

- Lemma : Let $\{i_1, i_2, i_3, \dots, i_k\}$ be the set of chosen classes

  Let $\{j_1, j_2, \dots j_k\}$ be another set of compatible classes

Then, $\forall r = 1, -, k \quad f[i_r] \le f[j_r]$

Proof by Contradiction: Assume $\exists \{j_1, j_2, \dots, j_k, j_{k+1}\}$ (there is one extra element than what our algorithm outputed)

From the lemma $f[j_k] \ge f[i_k]$, from compatability $s[j_{k+1}] \ge f[j_k] \ge f[i_k]$

$s[j_{k+1}] \ge f[i_k]$ ✗ our algorithm would have found the class j_{k+1} this violates ★ in psvedocode

- Now Prove the lemma: Proof by induction:

1. Base Case: $r=1$ $(f[i_1] \leq f[j_1])$ since we sorted classes based on finish time.

2. Inductive Step: assume that $f[i_r] \leq f[j_r]$

Know that $s[j_{r+1}] \geq f[j_r] \geq f[i_r]$

$\Rightarrow j_{r+1}$ is compatible with i_r.

$\Rightarrow f[j_{r+1}] \geq f[i_{r+1}]$; Since we sorted based on finish time we would encounter class j_{r+1} before i_{r+1} if it had an earlier finish time.

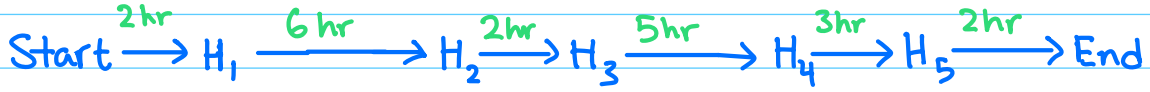
Ex # 2: Interval Partition Scheduling Based on classrooms.

We do proof by contradiction: assume that we are adding a room $d+1$ (d is depth)

implies $\Rightarrow$ we have a class that conflicts with d other classes

implies $\Rightarrow$ max_depth = d+1 which is a contradiction.

Ex #3: Road Trip:



Get from start to end in as few days. Can atmost drive 8 hrs in one day and must sleep in hotel.

Greedy Strategy: drive to farthest hotel each day.

Lemma: let $\{i_1, i_2, \ldots, i_k\}$ be where you sleep at day $1, 2, \ldots k$
let $\{j_1, j_2, \ldots j_k\}$ be another compatible set.
$\forall r \in 1, \ldots, k$

Induction:

1. Base case: $r = 1$ of course $i_1 \geq j_1$ because we maximize distance travelled.
2. Induction Step: Assume $i_r \geq j_r$

Because we can only travel 8 hours/day we know that there is no way to get farther to a point from j_r than i_r since $i_r \geq j_r$.

Proofs for Greedy:

- Stays ahead (and stays true)
- Contradiction
- Induction
- Achieves optinum value *

All proofs often require an invarient

Selection Sort:

Algorithm: ⇒ A greedy algorithm

```
for i←1 to n-1 do:
   //find smallest element in A[i..n]
    s←i
    for j←i+1 to n do
       if A[j] < A[s] then
          s←j

    if i≠s then
      t←A[s]; A[s]←A[i] ; A[i]←T
```

The invarient for this algorithm:

due to the first inner loop, we are guarenteed that elements up to the index will have the smallest values.

Insertion Sort:

```
for i←2 to n do:
    x ← A[i]
    j ← i
    while j>1 and x < A[j-1] do:
        A[j] ← A[j-1]
        j ← j-1

    A[j] ← x
return A
```

Bubble Sort:

```
for i=0 to length; i++ {
   for j to 0 to length-1-i; j++
   {
     if (A[j] > A[j+1])

        //swap values
   }
}
```

Stable Sorting: if two elements are the same in terms of sorting. then the hold the same order as the output.

In-place Sorting: We say that the amount of storage required is it uses is independent of the number of elements being sorted.

Binary Search ∋ only works when structure is sorted:

At halfway see if value is greater or less or equal. If greater explore halfway of upper. Otherwise explore the lower half.

Time Complexity: $\log_2(n)$.

```
bisearch(arr, start, end, x)
{
  if start ≥ end
      mid = (end + start)/2

      if arr[mid] == x
          return mid;
      elif arr[mid] > x
          return bisearch(arr, start, mid-1, x)
      elif arr[mid] < x
          return bisearch(arr, mid+1, end, x)
   else:
     return -1
}
```

Divide and Conquer:

Breaking the input into several parts, solves the problem in each part recursively, and then combines the solution to these sub-problems into an overall solution

More often than not we will see that the brute force algorithm when divide and conquer is used is is already polynomial time and divide and conquer serves to reduce it to an even lower polynomial time

The MergeSort Algorithm:

Divide the list in two equal halves, sorting each half seperately and then combining these two lists in linear time.

Analayzing Time Complexity:

let $T(n)$ denote the worst case time complexity of the Algorithm:

- The algorithm spends $O(n)$ to divide the input, $T(n/2)$ to sort the elements and $O(n)$ to merge them together.

$$T(n) \leq 2T(n/2) + cn \quad \text{and for } n > 2 \text{ we get } T(2) \leq c$$

$$T(n) \leq 2T(n/2) + O(n)$$

$\Rightarrow$ we can use two differrent methods to then solve this reccurrance relation the master method or the substitution tree method.

* Merge sort is $O(n\log(n))$

Recurrence forms for divide conquer:

$$T(n) = \begin{cases} c & \text{if } n < d \\ aT(n/b) + f(n) & \text{if } n \geq d \end{cases}$$

Master Theorem:

#1: if $f(n)$ is $O(n^{\log_b a - \varepsilon})$ then $T(n)$ is $\Theta(n^{\log_b a})$

#2: if $f(n)$ is $\Theta(n^{\log_b a}(\log n)^k)$ then $T(n)$ is $\Theta(n^{\log_b a}(\log n)^{k+1})$

#3: if $f(n)$ is $\Omega(n^{\log_b a + \varepsilon})$ then $T(n)$ is $\Theta(f(n))$

*The master theorem cannot be applied if $T(n)$ is mono-tone, $f(n)$ is not a polynomial or if a is not a constant.

Examples of the Master Theorm:

Ex#1: (Merge Sort)

$T(n) = 2T(n/2) + cn$; $a=2$; $b=2$ and $\log_b a = 1$

Since $f(n) = \Theta(N^{\log_b a}(\log N)^k)$ for $k=0$ we get that $f(n) = \Theta(N)$. Hence by the Master theorm we get that $T(N) = n\log(n)$

Ex#2:

$T(n) = 4T(n/2) + n$; $a=4$; $b=2$

hence we get that $\log_b a = \log_2 4 = 2 \Rightarrow n^2$ since $f(n)$ is $O(n^2)$

By the master theorem case #1 we say that $T(n) = \Theta(n^2)$

Ex#3: (Binary Search)

$T(n) = T(n/2) + 1$; $a=1$; $b=2 \Rightarrow \log_b a = 0$

is $f(1) \cong \Theta(n^0(\log n)^k)$ for some k; $\Rightarrow$ take $k=0 \Rightarrow$ it is true. Hence $T(n)$ is $O(\log n)$

Ex#4:

$T(n) = 9T(n/3) + n^3$; $a=9, b=3 \Rightarrow \log_b a = 2$

Our special function will be n^2. Since $f(n^3)$ is $\Omega(n^2)$ we can use case 3 of the master theorm we get $T(n)$ is (n^3).

Other method to solve a reccurrance relations is:

Substitution method: guess an answer and solve

Expansion: expand the relationship out and look for a pattern.

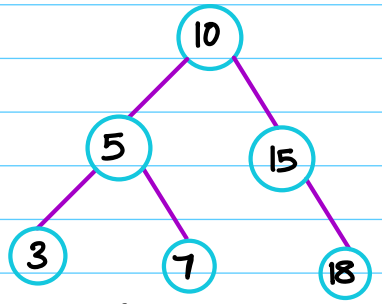
Breadth for Search on a Graph:

```
BFS(G)
Input: Rooted, Directed Graph G = (V, E, root)
Assume we are only interested in the vertices reachable from the root

ADD root of G to Queue
WHILE ( Queue is not empty )
    REMOVE next vertex in Queue and assign it to u
    VISIT u
    FOR each edge (u, v) in E
      IF ( v has not been visited ) THEN
            ADD v to end of Queue
```

Depth for Search for Graph:

```
DFS(G)
Input: Rooted, Directed Graph G = (V, E, root)
Assume we are only interested in the vertices reachable from the root

PUSH root of G onto top of Stack
WHILE ( Stack is not empty )
    POP top of stack and assign it to u
    VISIT u
    FOR each edge (u, v) in E
      IF ( v has not been visited ) THEN
            PUSH v onto top of Stack
```

Binary-Search Trees:



Post Order Traversal: Left, Right, Root : 3, 7, 5, 18, 15, 10
In Order Traversal: Left, Root, Right: 3, 5, 7, 10, 15, 18
Pre Order Traversal: Root, Left, Right: 10, 5, 3, 7, 15, 18

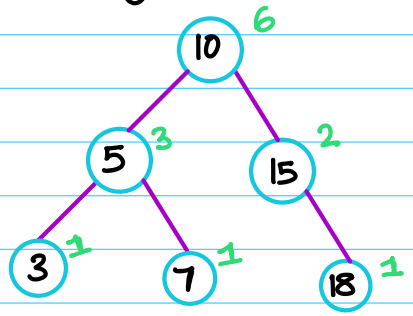
To delete a node: find the node that immediately occurrs after the in-order traversel. and replace those nodes.

ie. finding maximum value in the left subtree. Or finding the minimum value in the right subtree.

Augmented Binary Search Tree:

Each node not only stores it's corresponding value but it will also store the number of nodes below it including itself.

Having such tree makes it easier to find either the K^{th} largest or smallest value.



Dynamic Programming:

memoization, "Bottom → up".

Solve the problem by dividing into sub-problems which entails a recurrence. However, unlike divide and conquer which uses recursion we will instead solve each sub-problem in an iterative fashion. We will start with the smaller problems and working up to the final solution.

Three Principles of Dynamic Programming:

(i) Principle of Optimality:
- Solution is independent of how subproblems are solved.
- We can compute the subproblems in any order
- Cost of computation is manageable: # subproblems, # comp to combine

(ii) The number of potential calls for sub-problems is exponential; but the number of potential subproblems is polynomial.

(iii) We can arrange the solution as a sequence of computations of sub-problems, such that we compute all the sub-problems of the current problem before we try and compute the solution of the current problem.

Dynamic Programming Maximal Sub-array Sum:

Given an A[1:N] for $1 \le i \le j \le N$ find the maximum sub-array.
return the maximal sum and the indices i & j where it occurs.

A = 

Create a cumulative sum array where cs[1] = A[1] and cs[i] = cs[i-1] + A[i]

CS = 

⇒ this would take O(n) time to create cs.

Let this be denoted by a helper function be called createCS()
By using this technique we can calculate the sum between any two intervals in constant time: for i→j we get cs[J] - cs[I-1].

This would simplify our algorithm from $O(n^3)$ to $O(n^2)$.

We could further improve this by finding the largest sum and then the smallest sum. And this would then be our interval. Where the smaller element must come before the larger element.

Dynamic Programming Weighted Interval Task Scheduling:

The input is a list L, with requests. Each request, i, will have the following elements:

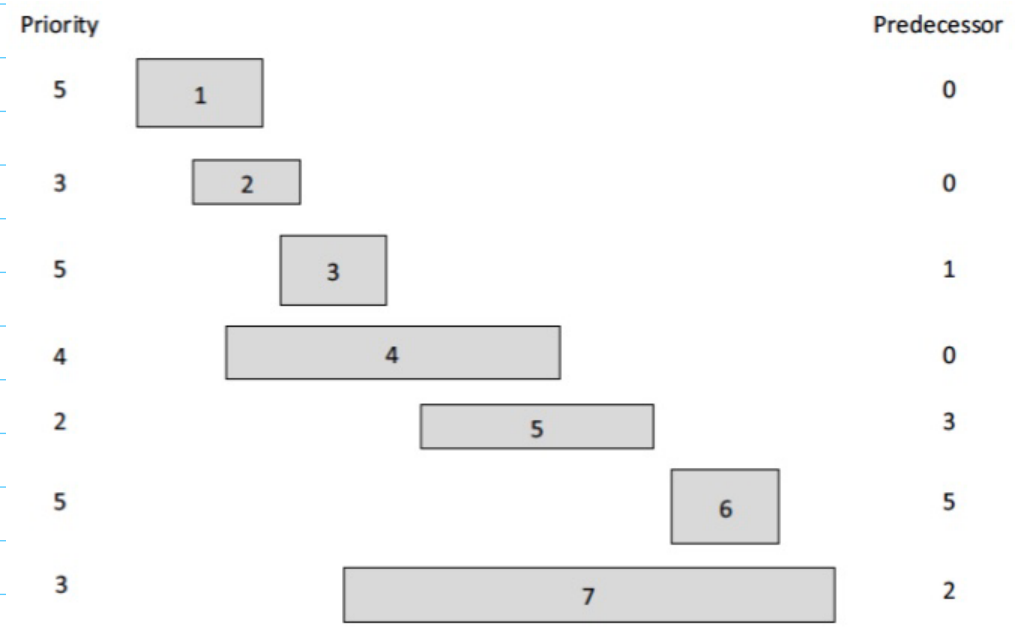
- a start time: s_i
- a finish time: f_i
- a positive numerical benefit, b_i,

Two requests i and j conflict if the time interval $[s_i, f_i]$ and intersects the time interval $[s_j, f_j]$

We look to optimize so that our requests do not conflict and we maximize the sum of the benefits that are included in the schedule.

For each request we will count the number of requests before the current task that do not conflict with it. If there are no predecessors we say that $P(i) = 0$.

Ex:



Assume two lists one sorted based on finish time. and a second list lf which is sorted based on start time.

Merge these two lists which means each item appears twice.
We take two indices t and p: t stops at each start time to compute P[i]. When t passes a finish time p is updated to that finish time.

To then incorporate the benefit:

$B[0] = 0$

for (i = 1 to n do)

$B[i] \leftarrow \max \{ B[i-1], B[P[i]] + b_i \}$

The recurrence for this is as follows:

$$B_i = \max(B_{i-1}, B_{pred(i)} + b_i)$$

A schedule that achievies the optimal B_i, either it includes or does not include the request.

M[1] = 1

M[12] = 3

Dynamic Programming: Coins

We have coins of denomination: 1, 10, 25, 100.
Get to target t with minimum number of coins

Problem substructure: OPT[T] = minimum # required to reach T

$$OPT[T] = \min \begin{cases} OPT[T-1]+1 \\ OPT[T-10]+1 \\ OPT[T-25]+1 \\ OPT[T-100]+1 \end{cases}$$

The naive approach to this would recursively call the min function. However there is a lot of overlap if you draw the tree. Instead:

```
def MinCoins(T):
    let M= length of T
    set M[1]=1, M[10]=1, M[25]=1, M[100]=1
    for i=1:T
        if i not in [1,10,25,100]
            M[i] = min(M[i-1]+1, M[i-10]+1, M[i-25]+1, M[i-100]+1)
    return M[T]
```

Complexity of this is O(T). ⇒ Suppose we want to print the list of coins

```
def MinCoins(T):                                  → last coin chosen
    let M= length of T, Let C= length T array
    set M[1]=1, M[10]=1, M[25]=1, M[100]=1
    for i=1:T
        if i not in [1,10,25,100]
            M[i] = min(M[i-1]+1, M[i-10]+1, M[i-25]+1, M[i-100]+1)
                set  C[i]=1   C[i]=10   C[i]=25   C[i]=100
    let t=T
    while t>0
        print C[t]
        t = t - C[t]
    return M[T]
```

Dynamic Programming Knapsack Problem:

A robber is in a treasurer, he can carry up to w_{max} weight, he has n items to choose from. the items each have a value of v_i which can be the same or different. What should he steal to maximize the total value stolen.

We will keep track of $\sum v, \sum w$ for items that are chosen

Let $OPT[n, w_{max}]$ = max value achievable when choosing from items 1 to n, with a limit of w_{max}.

$$OPT[i, w_{max}] = \max(\underbrace{OPT[i-1, w_{max} - w_i] + v_i}_{\text{Case to take item i}}, \underbrace{OPT[i-1, w_{max}]}_{\text{Don't take item i}})$$

```
robbery() {
  let M = n × w_max  ⇒ 2D array ; C = n × w_max 2D array ⇒ contains the items we take
  for (i = 1 to n)
    for(w_tot = 1 to w_max)
      if w_tot ≥ w_i
        M[i] = max( M[i-1, w_tot - w_i] + v_i , M[i-1, w_tot] )
                    set C[i, w_tot] = 1          set C[i, w_tot] = 0
      else:
        M[i] = M[i-1, w_tot] ⇒ set C[i, w_tot] = 0
  return M[n, w_max]
}
```

```
Print the choices:
nt = n ; wt = w_max
while nt > 0 and wt > 0
    if C(nt, wt] = 1
        print(nt)
        wt = wt - w_nt
        nt = nt - 1
    else:
        nt = nt - 1
        wt = wt
```

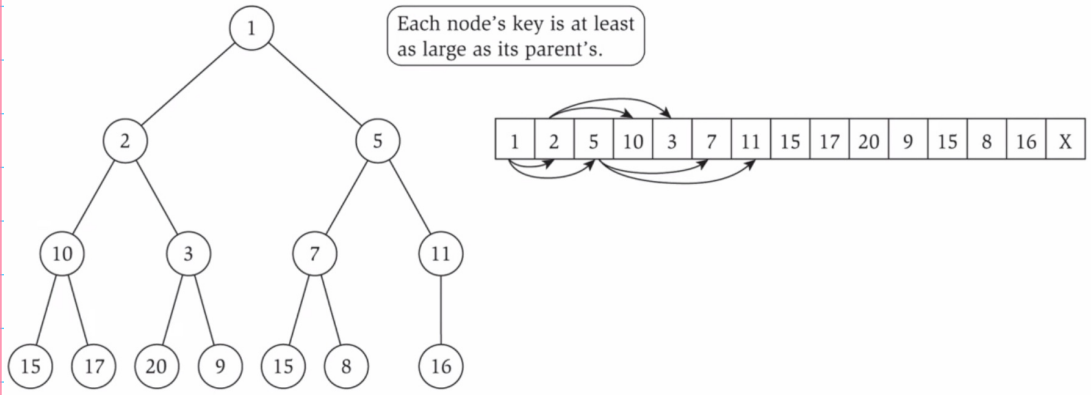
What would happen if w_{max} and weights were real numbers. Would this still be dynamic Programming?

No, because then there would be an infinite number of subproblems that we can then not calculate using dynamic programming.

Heaps Data Structure:

min-heap: every node is less than or equal to all of it's children.
max-heap: every node is greater than or equal to all of it's children.

We often implement a heap data structure using an array.
We assume we start at an index of 1 and an index of i has a left child at 2i, and the right child is at 2i+1



Heapify for Insertions:

Modify the tree to become a heap after insertion:

```
Heapify-up (H,i):
  if(i > 1)
    if(H[i] > H[i/2]) ⇒ this equality would change based on a min or max heap
        //swap the values
        heapify-up(H, i/2)
```

We will keep moving our inserted value up until the structure satsifies that of a heap.

Finding an element in a heap is O(n) which means we will need to search the entire array to determine if there is an element.

Finding the minimum value or max value (depending on whether it is a min or max heap) is easy since it would be the root value.

Deletion from a heap:

We will look to move the node we want to delete all the way to one of the leaf nodes:

```
Heapify-down(H,i):
  Let n = length(H)
  If 2i > n then
    Terminate with H unchanged
  Else if 2i < n then
    Let left = 2i, and right = 2i + 1
    Let j be the index that minimizes key[H[left]] and key[H[right]]
  Else if 2i = n then                    ↑ Or maximize if it's a max-heap
    Let j = 2i
  Endif
  If key[H[j]] < key[H[i]] then
     swap the array entries H[i] and H[j]
     Heapify-down(H, j)
  Endif
```

Priority Queue (PQ):

Similar to a regular queue, but instead each element has a priority, and values with a higher priority are dequed first.

The best way to implement a priority queue using a heap;
This is because it takes log(n) for insertion into an already created heap log(n) for deletion, and constant time for access for min, max value and length to be constant time.

Ex: We could use a min-heap to make the interval task partioning problem run in nlog(n) instead of n^2.

Heapsort :

- this method does not do in-place sorting:
- Build a max heap from input data
- Replace the root (which is the max value) with the last element in the heap.
- Repeat while the size of the heap is greater than 1.

```
Heapsort(A) :
    BuildHeap(A)
    for( i = A.length downto 2)
        exchange A[1] with A[i]
        A. heapsize = A.heapsize - 1
        Heapify(A, 1)
```

Bucketsort :

Suppose we have a sequence of n elements with a defined range. The bucket sort will use each of the values as an index. And then reads from all the buckets back to the array

Phase 1: Empty sequence S by moving each entry (k, o) into its bucket $B[k]$

Phase 2: For $i = 0, \ldots, N - 1$, move the entries of bucket $B[i]$ to the end of sequence S

Graph Definitions:

○ = vertices , —— = edges , ⟶ = directed edges

|V| = # vertices , |E| = # edges , degree(v) = total number of edges associated with vertex v

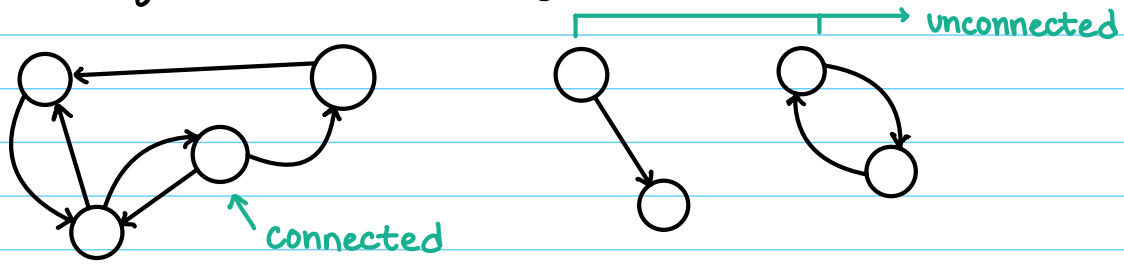
Representation:

- array or linked list of vertices
- each vertex carries a linked list of edges

Path: a series of nodes that connect a start and end node.

Cycle: path that starts and ends at same node.

Acyclic: a graph that has no cycles



Graph Algorithms:

n = (|V| + |E|)

Shortest Path Algorithm:

- No weights, single source: BFS O(n)
- Weights >0, single source: Dijkstra O(n log(V))
- Weights positive, negative, no negative cycles: Floyd-Warshall $O(V^3)$

Connected Components (undirected, no weights):

- Off line case: DFS or BFS O(n)
- On line case: Union find for insertion.

Minimum Weight Spanning Tree Problems:

- No weights: DFS or BFS traversal O(n)
- Positive, Negative Weights: Prim's and Dijkastra O(E log(V))
- Positive, Negative Weights: Kruskal's Algorithm O(E log(V))

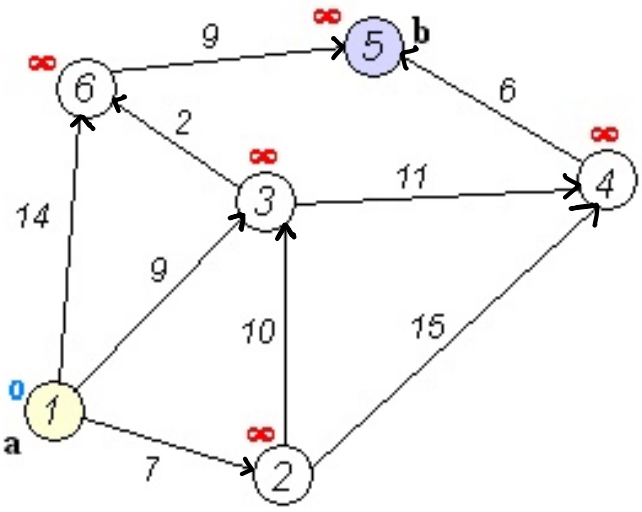
Dijkstra's Algorithm:

Maintain a set of explored nodes S for which we have determined the shortest path $d(u)$ from s to u.
Initialize $S = \{s\}$, $d(s) = 0$.

Choose unexplored node v which minimizes: $\pi(v) = \min_{e(u,v): u \in S} d(u) + \ell_e$
add v to S and set $d(v) = \pi(v)$.

Dijkstra's will always look at all of the nodes; when going in a BFS traversal.

The invarient will be that the n-nodes traversed will contain the shortest path traversed so far.




```
def Dijkstra(G, s):
    s.d = 0
    for v in {G.vertices \ s}:
        v.d = +∞
    Let X = list of expanded nodes
    Let Q = priority_queue of nodes to expand
    Q.insert(s)
    while ( !Q.isEmpty()): ⇒ up to |V| times
        v = Q.pop ⇒ O(log N)
        for (v1, w) in v0 edges: ⇒ All edges encountered once |E|
            if (v1.d > v.d + w):
                v1.d = v.d + w
            if v1 in Q:
                Q.update(v1) ⇒ O(log N)
            else:
                Q.insert(v1) ⇒ O(log N)
```

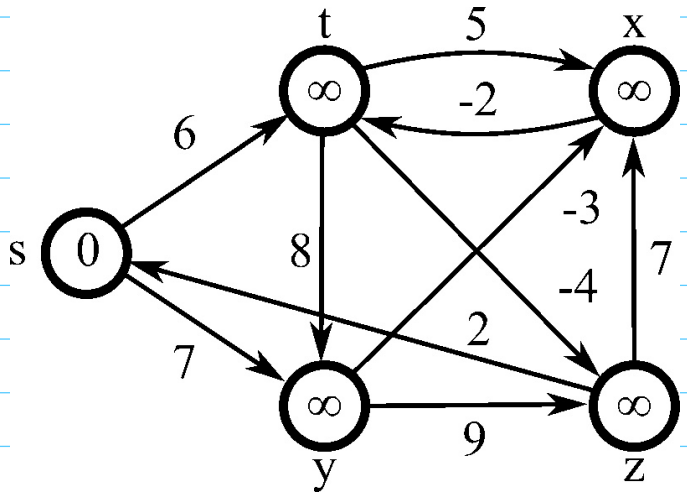
Total Complexity:

$O(|V| O(\text{pop}) + |E| \max(\text{insert}, \text{update}))$

Bellman-Ford:

Finds single source shortest path → this can have negative weights but cannot have negative cycles.

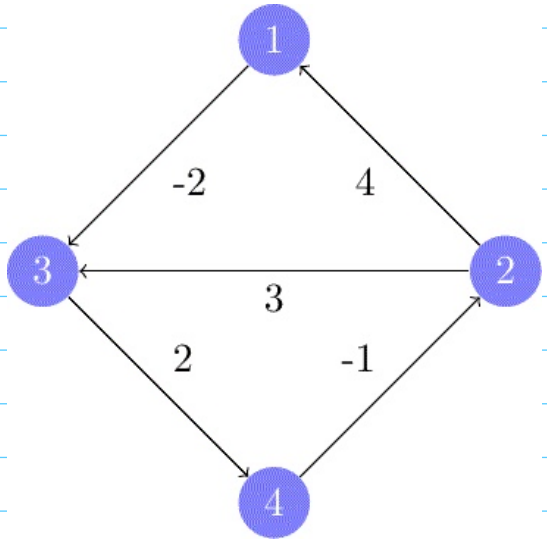
```
def BellmanFord[G, s]:
    s.d = 0
    for (i=1, ..., |V|):                      # Main Work
        for (v0, v1, w) in G.edges:           # O(|V||E|)
            if (v1.d > v0.d + w):
                v1.d = v0.d + w

    for (v0, v1, w) in G.edges:               # negative weight cycle detection
        if v1.d > v0.d + w
            return False
    return True
```



Floyd-Marshall Algorithm:

This algorithm finds the shortest path between all vertices in a weighted directed graph. Can have positive and negative weights but it cannot have a negative cycle.

```
def AllPair(G)
  for all vertexPairs(i,j)
    if i==j
      D0[i,j] = 0
    else if (i,j) is an edge
      D0[i,j] = w.edge
    else:
      D0[i,j] = +∞

  for k←1 to v:
    for i←1 to v:
      for j←1 to v:
        Dk[i,j] = min{Dk-1[i,j], Dk-1[i,k] + Dk-1[k,j]}
```



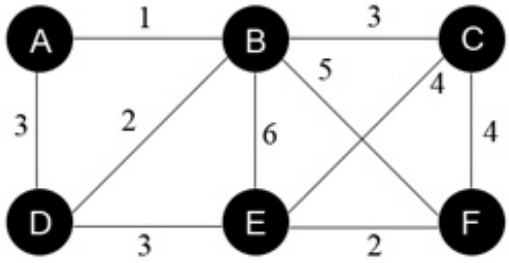
Minimum Spanning Tree: the spanning tree of a graph that connects all of the vertices together with no cycles and minimum weight.

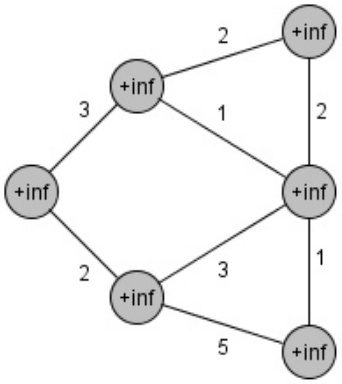
Prims Algorithm:

Finds the MST and is generally best for dense graphs.

```
def primMST(G, s) {
    parent[G.vertices]
    key[G.vertices]
    mstSet[G.vertices]
    for(int i=0; i<V; i++)
      key[i] = ∞
      mstSet[i] = false
    key[0] = 0
    parent[0] = -1

    for(int count = 0; count < V-1; count++)
      int u = minKey(key, mstSet)  ⇒ find the min key of the vertex that isn't in the MST array

      mstSet[u] = true
      for vertex in (u, vertex) in G.edges:
        if (mstSet == false) && (u, vertex).weight < key[v]
            parent[v] = u
            key[v] = graph[u][v]
}
```

SET: { }





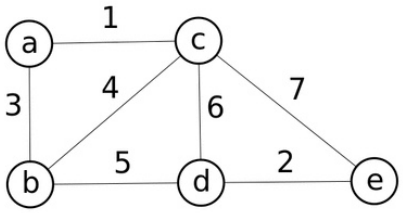
Prims algorithm is a greedy algorithm, the time complexity of the above algorithm is $O(V^2)$ but with the help of a heap we can simplify the time complexity of this algorithm to be $O(E \log V)$.

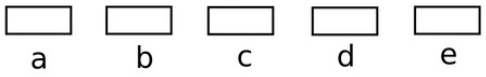
Kruskal's Algorithm:

Unlike prims algorithm, kruskals algorithm first looks at all of the edges first. Kruskals algorithm is a greedy algorithm.

1. Sort All edges in non-decreasing order
2. Pick the smallest edge
3. If it forms a cycle with the MST discard it
4. Repeat Steps 2 and 3 until there are V-1 edges in the MST.

```
Kruskals (G, c) {
   minEdges = sort  G.edges
   MST = []

   for each  v ∈ G.vertex :
      make a single set  with v.

   for i to |E| :
      (u, v) = minEdges[i-1]
      if (u and v are in different sets) {
         MST.add[minEdges[i-1]]
         merge sets containing  u, v
      }
  return MST
}
```





$O(E \log(V))$ + $O(E\,\alpha(E,V))$

$O(E \log(V))$: for sorting edges

$O(E\,\alpha(E,V))$: for union find

It is important to note that $\alpha(E,V)$ is inverse Ackermans function which almost acts like a constant.

Union-Find Partition:

- makeSet(x): Create a singleton set containing the element x and return the position storing x.
- union(A, B): return A∪B, destroying the old A and B.
- find(p): return the set containing the element p.

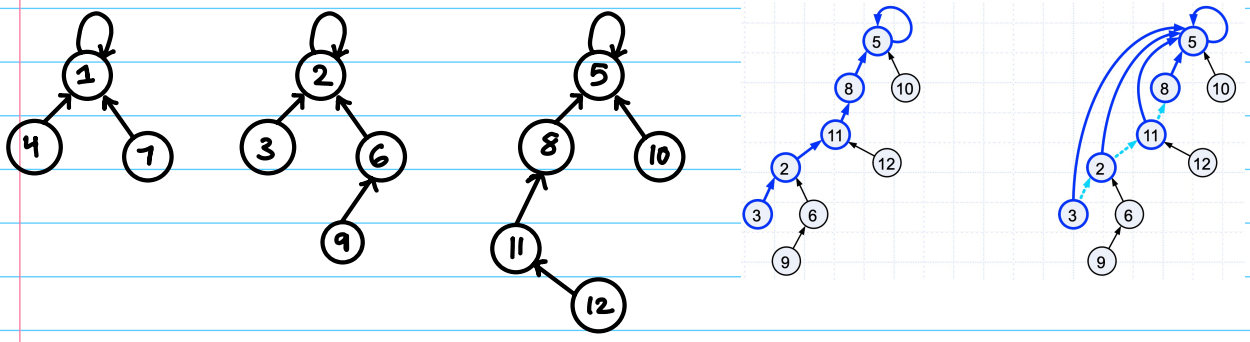
Implementation Option #1: Linked List

- Each set is stored in a sequence represented with a linked list. Each node should store an object containing the element and a reference to a set name.

The total time that is needed to do n unions and m finds would be $O(n\log(n) + m)$

Implementation Option #2: Tree

- Each element is a node that contains a pointer to a set name
- A node v that points back to itself is a set name.



When doing a union we make the root of one tree point to the root of another.

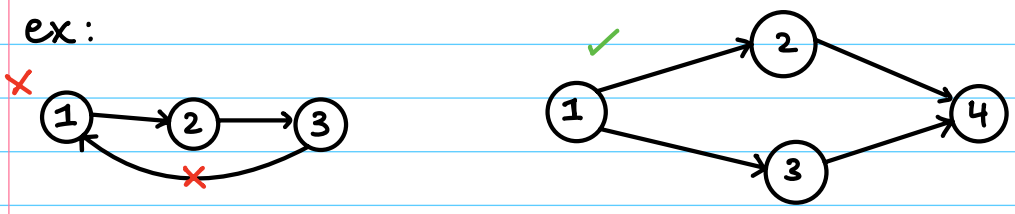
To do a find we traverse until we reach an element that points to itself.

Path compression: when doing a find we will update all pointers on a path to point to the root

Topological Sort:

A topoligal ordering is an ordering of nodes v_1 to v_n such that all edges point forward in the ordering.

ex:

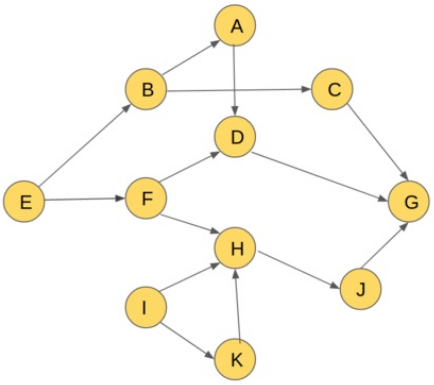


If g is a directed acyclic graph (DAG) ⇒ G has a topological ordering.
If g is a DAG there exists a node with no incoming edge.

Topological Algorithm:

Find Node with 0^{th} degree and remove from graph
Repeat process

```
def TopologicalSort(G):
  let L[] = linked list of n-degree nodes
  let O[] = output
  let numDegree[|V|]

  for v0 in G.vertices:
    for v1 in v0.edges:
      numDegree[v0] += 1
  for v in G.vertices:
    if numDegree[v] == 0
      L.add(v)
  while (!L.empty())
    O.add(L.pop)
    for v1 in v0.edges : numDegree[v1] --
    if(numDegree[v1] == 0) : L.add(v1)
```



RESULT :

Graph Representations:

Adjacency Matrix:

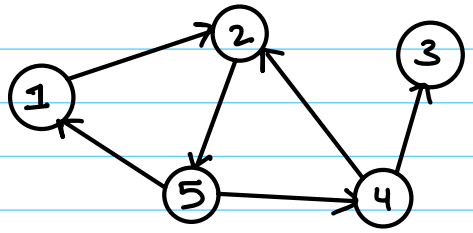
A $|V| \times |V|$ matrix with $A_{uv} = 1$ if (u, v) is an edge. If it is an undirected graph then there will be 1 at both (u, v) and (v, u) ⇒ 2 representations.

Determining if (u, v) is and edge is $O(1)$

Finding All edges, takes $|V|^2$.

Adjacency List:

Each node in a graph will refer to an index of array lists. which will be all of the edges. Identifying an edge takes $O(\deg(u))$. Identifying all edges takes $O(|E| + |V|)$.



1 : 2
2 : 5
3 :
4 : 2 → 3
5 : 1 → 4

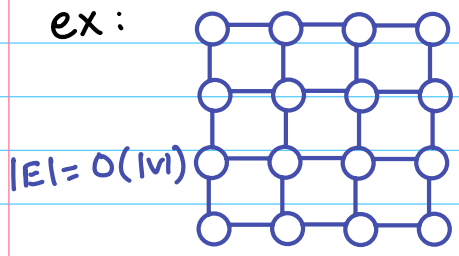
The adjacency list will often will end up being more space efficient but slightly slower than the adjacency matrix

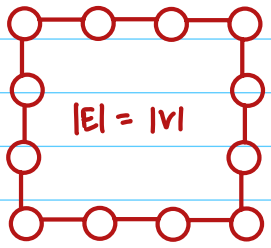
The other factor that will often affect performance of the algorithm is the shape of the graph.

Sparse Graphs:

There is no universal definition for what constitutes a sparse graph. A definition that will help or most commonly used is that: where $|E|$ is $O(|V|)$

ex:





All trees and acyclic graphs satisfy this since $|E| = |V| - 1$

Low Degree Graphs:

- A sparse graph may not nessicarily be a low degree graph, but a low degree graph is a sparse graph.

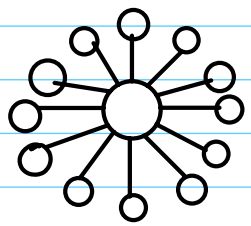
- constant degree graph $\Rightarrow$ $|E|$ is $O(|V|)$

- "limited degree" (no formal definition) $\Rightarrow$ $d = \log|V| \Rightarrow |E| = O(V \log V)$

High-Degree Graphs:

a sparse graph with a high degree.

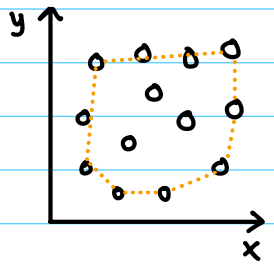
Ex: Star Graph is a graph that satisfies the definition of a sparse graph, but has a high degree.



Sparse matrices are often a result of sparse graphs and will mostly contain zeros in the matrix. There are two main ways to represent these matrices which is either banded or unbanded.

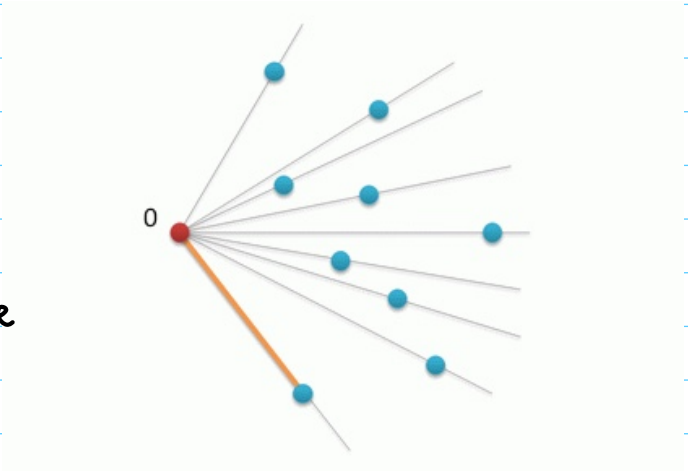
Introduction to Computational Geometry:

- Ex #1: Convex Hull (Jarvis Algorithm)



Find the smallest convex hull from a set of points such that the hull contains all of the points. Despite having 4 dots consecutively we only show it as a single line.

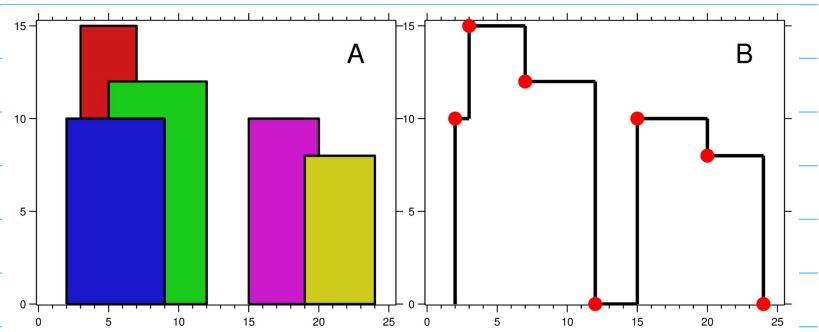
1. Find the left-most point
2. While we are not at the point
3. Find the next point q.
such that (p,q,r) for any point r. q is the moust counter clockwise point. ⇒ O(m×n) time




```
def convexHull(points[])
{
  hull[]
  left = null;

  for point in points[]:
    if left.x > point.x
      left = point

  p = left ; q = null
  do {
    hull.add(p)
    for i in points:
      if (orient(p, i, (p+1)%.n  == 2)
        q = i

    p=q
  } while(p != left)
}
```

```
def orient(p, q, r)
{
  val = (q.y - p.y)*(r.x - q.x)

  val = val - (q.x - p.x)*(r.y - q.y)
  if (val == 0 || val > 0)
    return 0
  return 2
}
```

- Ex #2 Skyline Problem:



- Ex #3 Maxima Set Finding Problem :

Complexity Theory:

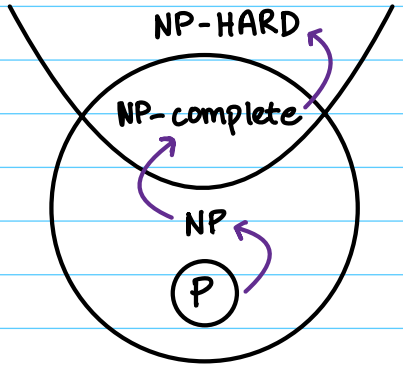
Complexity Classes:

P ⇒ tractable problem that is solved in polynomial time

NP ⇒ intractable problem that is verified in polynomial time

By definition P is a subset of NP. So P = NP is whether P is a proper subset of NP.

Polynomial Time Reductions:



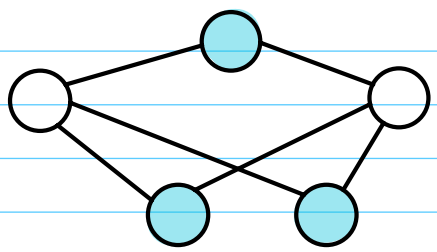
Suppose that we have a problem A and problem B. If we can transform *all* instances of A into instances of B, where B is a harder problem. Then if the transformation is done in polynomial time. Then A is a polynomial time reduction to B: $A \leq_p B$.

NP-Complete Problems:

3SAT: (Boolean Satisfiability problem)
We have a boolean expression of variables $x_1, x_2, x_3, \ldots x_n$ and an expression Φ_1 are there values of T/F from $x_1 \ldots x_n$ that make the expression Φ_1 true. The 3sat is a variation of this problem where each clause has only 3 variables.

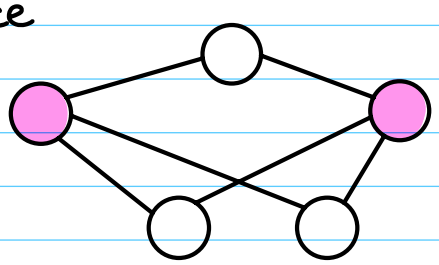
Equal Partition: given a set of n elements, divide it to have two disjoint sets whose sum is equal if possible.

Independent Set: given graph $G = (V, E)$, find the maximum $S \subseteq V$ such that no two nodes are adjacent.



Independent Set

Vertex Cover: Given a graph $G = (V, E)$, find the minimum number set of nodes such that all edges touch at least once



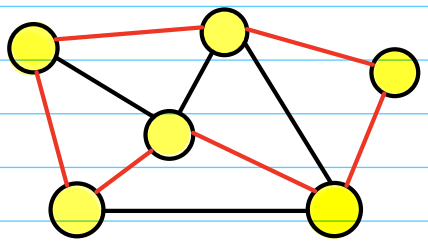
Vertex Cover

Set Cover: Given a set U and m subsets $S_1, S_2, S_3 \cdots S_m$ of U find the minimum number of subsets whose union is U

$U = \{1,2,3,4,5,6\}$; $S_1 = \{1,3,4,5\}$; $S_2 = \{2,4,6\}$; $S_3 = \{4,6\}$; $S_4 = \{5\}$

$\Rightarrow S_1, S_2$

Hamiltonian Circuit: given a graph G (E, V) does it contain a cyclic path that passes through every vertex once, without repeating a vertex.



Red path indicates a hamiltonian cycle

3D Matching: given disjoint sets W, X, Y with the same number of elements n, given a set M of triplets of the form (W, X, Y) is there a subset of M called M' of size q where no two elements of M' agree in any coordinate.

Ex: $W: \{1,2,3,4\}$, $X = \{10,20,30,40\}$, $Y = \{100,200,300,400\}$

A valid M' would be $\{(1,20,300), (2,30,400), (3,40,100), (4,10,200)\}$ assuming all 4 of those happened to be in M.

Problem Reductions:

Given two problems A and B:

1. Rewrite A as an instance of B
2. Use an existing algorithm to solve B
3. Rewrite the solution of B as a solution of A.

If we can do both rewriting steps in polynomial time: $A \leq_p B$

If $B \leq_p A$, then we show that: $A \equiv_p B$

Ex #1: Vertex Cover $\leq_p$ Independent Set

Let S be a vertex cover of $G = (E, V)$.

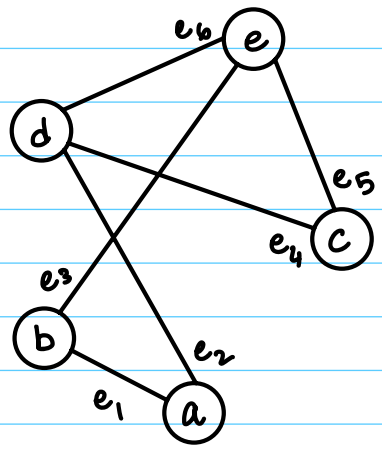
Since S is a vertex cover, nodes not in S have edges with S.

↳ Hence nodes in V-S are not adjacent

↳ Hence they are an independent set.

A vertex cover of size $|k|$ will be an independent set of size $|v|-k$

Ex #2: Vertex Cover $\leq_p$ Set Cover



$U = \{e_1, e_2, e_3, e_4, e_5, e_6\}$

$S_a = \{e_1, e_2\}$; $S_b = \{e_1, e_3\}$; $S_c = \{e_4, e_5\}$

$S_d = \{e_3, e_5, e_6\}$; $S_e = \{e_2, e_3, e_6\}$

⇒ Suppose we have a vertex cover. Then the union of the nodes in the vertex cover would cover the whole set.

⇒ Let's say we have a set cover. Then those sets will correspond to the vertex cover.

Ex #3: Equal Partition proofs

Two parts:

1. Prove that verifying is done in polynomial time ⇒ it is NP
2. Prove that it is NP hard

If we can reduce a problem A to a problem B, then we say that problem A is no more difficult than the second problem. This also means that the second problem is as at least as difficult as the first problem.

Part #1: We can verify in linear time that the solution is correct since we just have have to add up each element to see if they are equal.

Part #2: show 3-D Matching $\leq_p$ Equal Partition

Instance of 3D

$|G| = |B| = |Q| = k$

$M = \{m_1, \ldots, m_n\}$

where M' exists:

$M' \subseteq M$; $|M'| = k$

and every element of G, B, & Q appears only once.