YUKE's TRICKS

1. Greedy

HOW TO PROVE: FOLLOW THESE 2 TEMPLATES:

Exchange argument: http://www.cs.cornell.edu/courses/cs482/2007su/exchange.pdf

Stay ahead: https://www.cs.cornell.edu/courses/cs482/2003su/handouts/greedy_ahead.pdf

- Minimize/maximize the objective, assuming the request is the last request
- Always stay ahead (with every step we take, we're always better than or equal to an optimal solution) OR exchange argument
- Finish First (Always stay ahead - at every time step i, |Si| ≥ |S∗i|.):

Heuristic 4: Finish First

```
Algorithm: FinishFirst
Let $S$ be an initially empty set.
while $\sigma$ is not empty do
    Choose $r_i \in \sigma$ with the smallest finish time (break ties arbitrarily).
    Add $r_i$ to $S$.
    Remove all incompatible request in $\sigma$.
end
return $S$
```

Sort requests O(nlogn), removing incompatible requests until a request has a compatible start time O(n). In total O(nlogn) + O(n) = O(nlogn).

- Minimize Lateness (exchange argument)

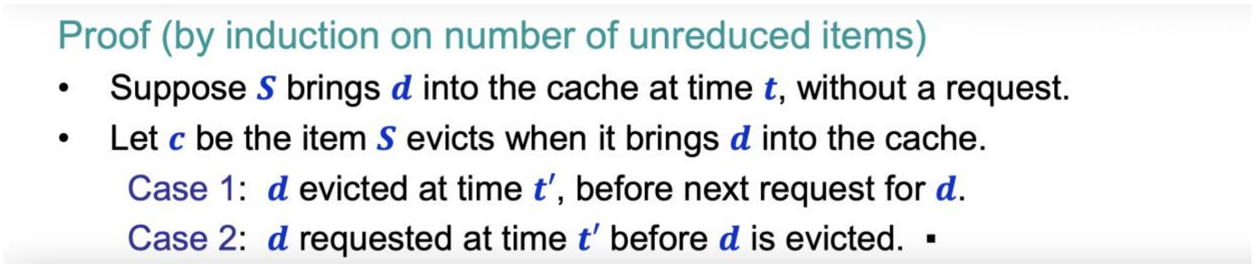
Heuristic 3: Earliest deadline first.

```
Algorithm: EDF
Let $J$ be the set of jobs.
Let $S$ be an initially empty list.
while $J$ is not empty do
    Choose $j \in J$ with the smallest $d_i$ (break ties arbitrarily).
    Append $j$ to $S$.
end
return $S$
```

- Shortest Path:

```
Algorithm: Dijkstra's
Let $S$ be the set of explored nodes.
For each $u \in S$, we store a distance value $d(u)$.
Initialize $S = \{s\}$ and $d(s) = 0$
while $S \neq V$ do
    Choose $v \notin S$ with at least one incoming edge originating from a node in $S$ with the smallest
        $d'(v) = \min_{e=(u,v):u \in S} \{d(u) + \ell_e\}$
    Append $v$ to $S$ and define $d(v) = d'(v)$.
end
```

# of iterations of loop: n-1
Find the minimum: O(m)
Overall: O(mn)
-Paging - Evict the page whose next request is the furthest into the future:
Use exchange argument: swap choices to convert other schedules to FF w/o losing quality
A reduced schedule is a schedule only inserts an item into the cache in a step in which that item is requested.
Unreduced schedule S → reduced schedule S. with no more cache misses.



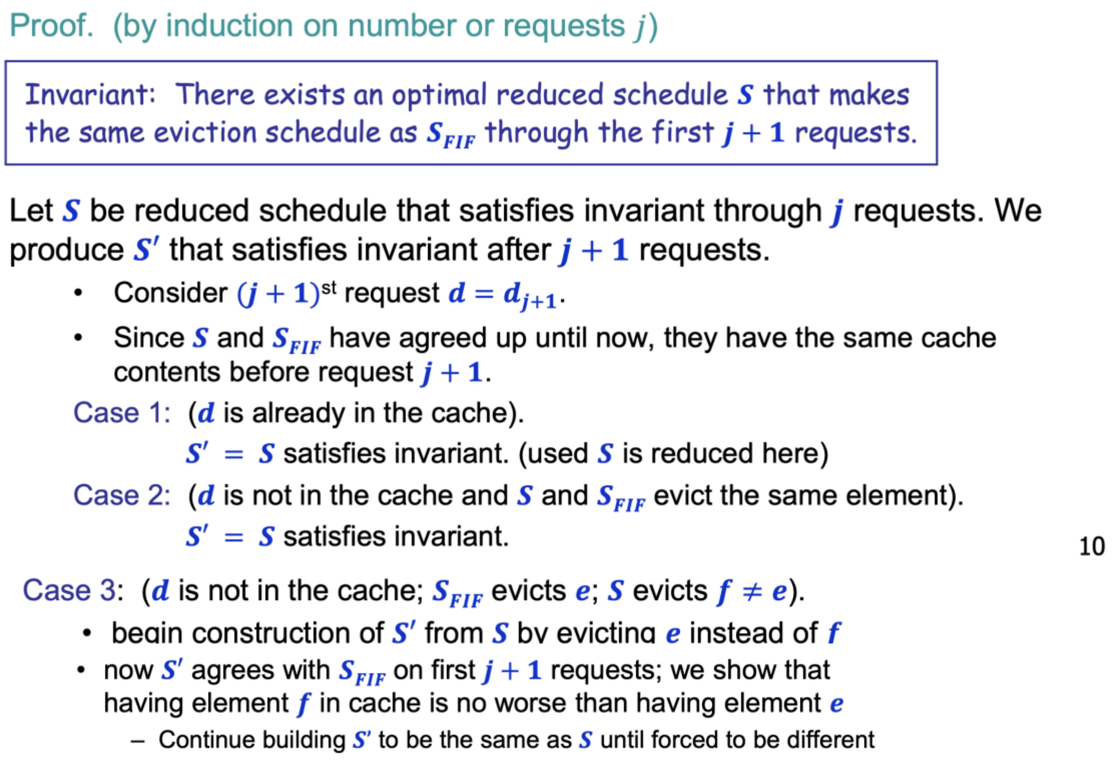
Proof (by induction on number of unreduced items)

- Suppose S brings d into the cache at time t, without a request.
- Let c be the item S evicts when it brings d into the cache.

  Case 1: d evicted at time t', before next request for d.

  Case 2: d requested at time t' before d is evicted. ▪

FF is optimal:



Proof. (by induction on number or requests j)

Invariant: There exists an optimal reduced schedule S that makes the same eviction schedule as S_{FIF} through the first $j+1$ requests.

Let S be reduced schedule that satisfies invariant through j requests. We produce S' that satisfies invariant after $j+1$ requests.

- Consider $(j+1)$st request $d = d_{j+1}$.
- Since S and S_{FIF} have agreed up until now, they have the same cache contents before request $j+1$.

Case 1: (d is already in the cache).
$S' = S$ satisfies invariant. (used S is reduced here)

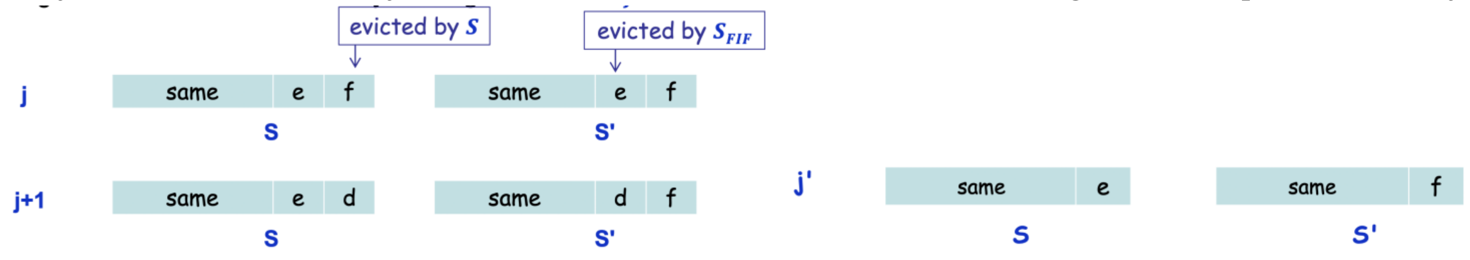
Case 2: (d is not in the cache and S and S_{FIF} evict the same element).
$S' = S$ satisfies invariant.

10

Case 3: (d is not in the cache; S_{FIF} evicts e; S evicts $f \neq e$).

- begin construction of S' from S by evicting e instead of f
- now S' agrees with S_{FIF} on first $j+1$ requests; we show that having element f in cache is no worse than having element e
  - Continue building S' to be the same as S until forced to be different

Let j' be the first time after j+1 that S and S' must take a dif./ action, and let g be item requested at time j':



Case 3a: g=e: can’t happen because e was evicted by FF so there must be a request for f before e.
Case 3b: g=f: Element f not in cache of S, so let e’ be the element S evicts:
    -if e’=e: S’ accesses f from cache, now S and S’ have same cache

-else: S’ evicts e’ and brings e into cache, now S and S’ have the same cache (S’ no longer reduced, but can be transformed into a reduced schedule that agrees with FF thru step j+1)
Case 3c: g≠e and g≠f: S must evict e, make S’ evict f; now S and S’ have the same cache.
In each case can now extend S’ using rest of S at no extra cost. S’ is optimal, reduced, and agrees with FF for j+1 steps. Optimality of FF follows by induction.

-MST:

Kruskal: Sort edges by cost from lowest to highest. Insert edges unless insertion would create a cycle.
Prim: Initialize a node set S with an arbitrary node s. Keep the least expensive edge as long as it does not create a cycle.
Reverse-Delete: Sort edges by cost highest to lowest. Remove edges unless graph become disconnected.

2. Divide Conquer

- When to use DC:
  Split problem into smaller sub-problems.
  Solve (usually recurse on) the smaller sub-problems.
  Use the output from the smaller sub-problems to build the solution. Recursive solutions.
- Review:

Binary Search

Given a sorted array A and a target value t, locate the middle element, m. If m equals t, return the index of m. If m is greater than t, recurse on A’s right half subarray. Otherwise, recurse on A’s left half subarray. The recurrence relation is expressed as T(n) = T(n/2) + O(1), allowing for a time complexity of O(log(n)).

Merge Sort

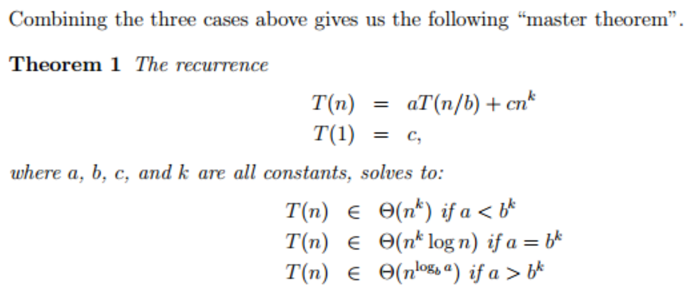
Given an array **A**, split **A** in half into two subarrays, **ALeft** and **ARight**. Then recurse over these subarrays. Now, since we can assume they are sorted, iteratively merge **ALeft** and **ARight** into a single array, **AFinal**, by determining which subarray contains the next smallest element. In the end, return **AFinal**. The recurrence relation is expressed as **T(n) = 2T(n/2) + O(n)**, allowing for a time complexity of **O(nlog(n))**.

- Goal: Reduce a term n in the brute force time into log(n)

- If brute force is O(n), go sth like binary search, if bruce force O(n^2), go sth like merge sort. → When you’re doing a problem, try to solve it w/ greedy first. Another approach is to think about which way you can solve your problem. 1 question ask yourself: Do you need MERGE?

- Generalized recurrence: T(n)≤q*T(n/2)+cn, T(1)≤c
  If q>2: O(n^log(q)); If q=2: O(n*logn); if q=1: O(n)

-Master theorem:



Combining the three cases above gives us the following “master theorem”.

Theorem 1 *The recurrence*

$$T(n) = aT(n/b) + cn^k$$
$$T(1) = c,$$

where a, b, c, and k are all constants, solves to:

$$T(n) \in \Theta(n^k) \text{ if } a < b^k$$
$$T(n) \in \Theta(n^k \log n) \text{ if } a = b^k$$
$$T(n) \in \Theta(n^{\log_b a}) \text{ if } a > b^k$$

A≥1, b≥2, c>0, f(n) ∈ Θ(n^d), d>0

- Count the # of inversions ((ai,aj), i<j, ai>aj)

O(n^2) that checks all pairs → want O(nlogn)

```
Algorithm: CountSort
Input  : A list $A$ of $n$ comparable items.
Output: A sorted array and the number of inversions.
if $|A| = 1$ then return $(A, 0)$
$(A_1, c_1)$ := CountSort(Front-half of $A$)
$(A_2, c_2)$ := CountSort(Back-half of $A$)
$(A, c)$ := MergeCount($A_1$,$A_2$)
return $(A, c + c_1 + c_2)$
```

```
Algorithm: MergeCount
Input  : Two lists of comparable items: $A$ and $B$.
Output: A merged list and the count of inversions.
Initialize $S$ to an empty list and $c := 0$.
while either $A$ or $B$ is not empty do
    Pop and append min{front of $A$, front of $B$} to $S$.
    if Appended item is from $B$ then
        $c := c + |A|$.
    end
end
return $(S, c)$
```

Approach:

Suppose the number of inversions in the left half and right half of the array (let be inv1 and inv2); what kinds of inversions are not accounted for in Inv1 + Inv2? The answer is – the inversions that need to be counted during the merge step. Therefore, to get the total number of inversions that needs to be added are the number of inversions in the left subarray, right subarray, and merge().

How to get the number of inversions in merge()?

In merge process, let i is used for indexing left sub-array and j for right sub-array. At any step in merge(), if a[i] is greater than a[j], then there are (mid – i) inversions. because left and right subarrays are sorted, so all the remaining elements in left subarray (a[i+1], a[i+2] … a[mid]) will be greater than a[j].

- Closest Pair of Points:

Divide: Split point set in half

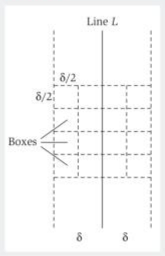
Conquer: Find closest pair in each partition

Combine: merge the solutions (L is the line in the middle, Q is left region, R is right region)

Lemma 3

Let S be the set of points within δ of L. If there exists a $s, s' \in S$ and $d(s, s') < \delta$, then s and s' are within 15 positions of each other in S_y.

Proof.



- Partition δ-space around L into $\delta/2$ squares.
- At most 1 point per square else contradicts definition of δ.
- By way of contradiction, say $d(s, s') < \delta$ and s and s' separated by 16 positions.
- By counting argument, s and s' are separated by 3 rows which is at least $3\delta/2$. □

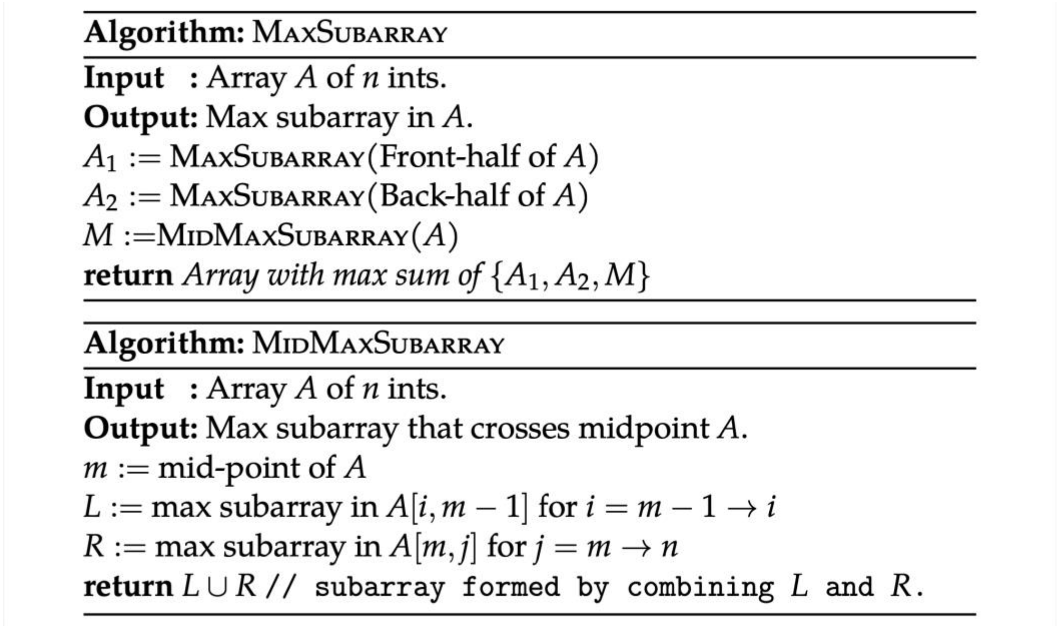
Lemma 3

Let S be the set of points within δ of L. If there exists a $s, s' \in S$ and $d(s, s') < \delta$, then s and s' are within 15 positions of each other in S_y.

Completing the Algorithm

- Find the min pair (s, s') in S.
    - For each $p \in S$, check the distance to each of next 15 points in S_y.
- If $d(s, s') < \delta$, return (s, s')
- else return min of (q_0^*, q_1^*) and (r_0^*, r_1^*).

Correctness:
By induction on the number of points.
Use the definition of the algorithm and the claims establish in Step 3 (combine the solution).
Sorting by x and by y (O(n log n)).
How many recursive calls? 2.
What is the size of the recursive calls? n/2.
Work per call: check points in S = 15 · |S| = O(n).
What is the recurrence? T(n) ≤ 2T(n/2) + cn = O(n log n).
- Max subarray: Given an array A of integers, find the contiguous subarray of A of maximum sum.
O(n^2): check all possible contiguous subarrays



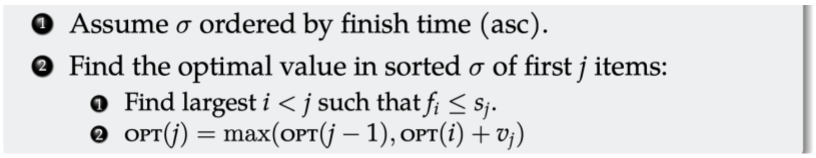
```
Algorithm: MaxSubarray
Input  : Array $A$ of $n$ ints.
Output: Max subarray in $A$.
$A_1$ := MaxSubarray(Front-half of $A$)
$A_2$ := MaxSubarray(Back-half of $A$)
$M$ :=MidMaxSubarray($A$)
return Array with max sum of $\{A_1, A_2, M\}$
```

```
Algorithm: MidMaxSubarray
Input  : Array $A$ of $n$ ints.
Output: Max subarray that crosses midpoint $A$.
$m$ := mid-point of $A$
$L$ := max subarray in $A[i, m-1]$ for $i = m-1 \to i$
$R$ := max subarray in $A[m, j]$ for $j = m \to n$
return $L \cup R$ // subarray formed by combining $L$ and $R$.
```

Correctness: By induction, $A1$ and $A2$ are max for subarray and M is max mid-crossing array.
Complexity: Same recurrence as MergeSort.

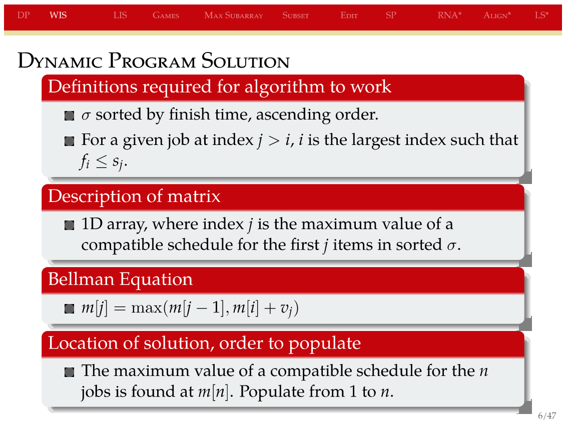
We can easily find the crossing sum in linear time. The idea is simple, find the maximum sum starting from mid-point and ending at some point on left of mid, then find the maximum sum starting from mid + 1 and ending with some point on right of mid + 1. Finally, combine the two and return the maximum among left, right and combination of both.
3. Dynamic Programming:
What you want to ask yourself is whether your problem solution can be expressed as a function of solutions to similar smaller problems.
Current step/decision depends on our next step/decision, aka recursive structure between our subproblems?
Typically, all the problems that require maximizing or minimize certain quantities or counting problems that say to count the arrangements under certain conditions or certain probability problems can be solved by using Dynamic Programming.
1 Classical example: https://cornelltech.github.io/CS5112-F18/Lectures/Lecture%2015%20-%20Dynamic%20Programming.pdf
All dynamic programming problems satisfy the overlapping subproblems property and most of the classic dynamic problems also satisfy the optimal substructure property. Once, we observe these properties in each problem, be sure that it can be solved using DP.

- Step2: Identify problem variables: express the problem in terms of the function parameters and see which of those parameters are changing (changing for every subproblem). They can uniquely identify a certain position or standing in the given problem.
- Step3: Express the recurrence relation – How do problems relate to each other? (Remember LCS(…) = ? given each observed condition)
- Step4: Recognize the base case: a subproblem that doesn't depend on any other subproblem.
"In order to find such subproblems, you typically want to try a few examples, see how your problem simplifies into smaller subproblems, and identify at what point it cannot be simplified further. The reason a problem cannot be simplified further is that one of the parameters would become a value that is not possible given the constraints of the problem."
- Step5: Iteratively or Recursively?
- Step6: Memoization - storing the results of expensive function calls and returning the cached result when the same inputs occur again.
This means that you should: Store your function result into your memory before every return statement. Then, look up the memory for the function result before you start doing any other computation
- Weighted Interval Scheduling:



1. Assume σ ordered by finish time (asc).
2. Find the optimal value in sorted σ of first j items:
   1. Find largest $i < j$ such that $f_i \leq s_j$.
   2. $\text{OPT}(j) = \max(\text{OPT}(j-1), \text{OPT}(i) + v_j)$

opt(j) = max(opt(j − 1), opt(i) + vj) - ff assures that the last interval is either in the solution or not.



DP WIS LIS GAMES MAX SUBARRAY SUBSET EDIT SP RNA* ALIGN* LS*

DYNAMIC PROGRAM SOLUTION

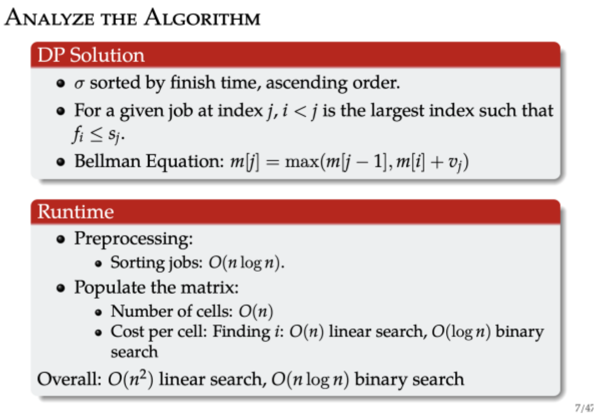
Definitions required for algorithm to work
- σ sorted by finish time, ascending order.
- For a given job at index $j > i$, i is the largest index such that $f_i \leq s_j$.

Description of matrix
- 1D array, where index j is the maximum value of a compatible schedule for the first j items in sorted σ.

Bellman Equation
- $m[j] = \max(m[j-1], m[i] + v_j)$

Location of solution, order to populate
- The maximum value of a compatible schedule for the n jobs is found at $m[n]$. Populate from 1 to n.

6/47



ANALYZE THE ALGORITHM

DP Solution
- σ sorted by finish time, ascending order.
- For a given job at index j, $i < j$ is the largest index such that $f_i \leq s_j$.
- Bellman Equation: $m[j] = \max(m[j-1], m[i] + v_j)$

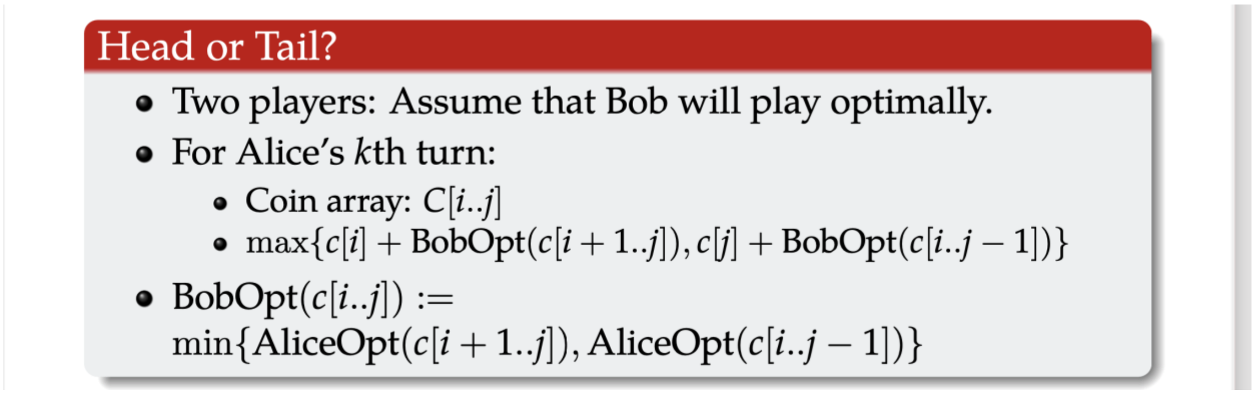
Runtime
- Preprocessing:
  - Sorting jobs: $O(n \log n)$.
- Populate the matrix:
  - Number of cells: $O(n)$
  - Cost per cell: Finding i: $O(n)$ linear search, $O(\log n)$ binary search

Overall: $O(n^2)$ linear search, $O(n \log n)$ binary search

7/47

- Longest Increasing Subsequence:
First, we need some function or array that represents the answer to the problem from a given state. For this problem, let's say that we have an array dp. As just stated, this array needs to represent the answer to the problem for a given state, so let's say that dp[i] represents the length of the longest increasing subsequence that ends with the ith element. The "state" is one-dimensional since it can be represented with only one variable - the index i.
Second, we need a way to transition between states, such as dp[5] and dp[7]. This is called a recurrence relation and can sometimes be tricky to figure out.
Let's say we know dp[0], dp[1], and dp[2]. How can we find dp[3] given this information? Well, since dp[2] represents the length of the longest increasing subsequence that ends with nums[2], if nums[3] > nums[2], then we can simply take the subsequence ending at i = 2 and append nums[3] to it, increasing the length by 1. The same can be said for nums[0] and nums[1] if nums[3] is larger. Of course, we should try to

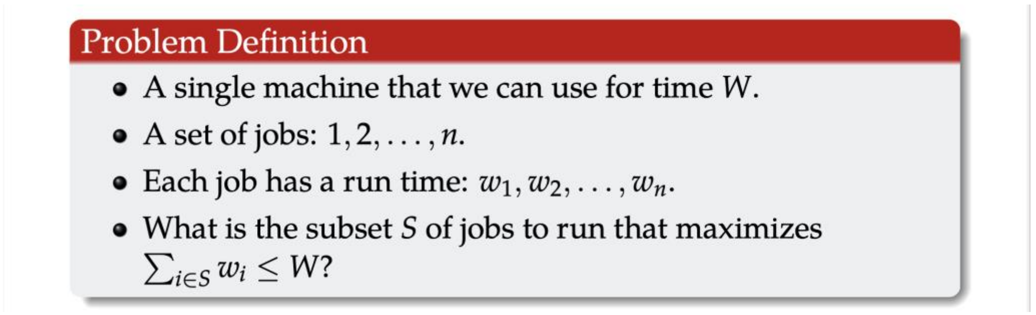
maximize dp[3], so we need to check all 3. Formally, the recurrence relation is: dp[i] = max(dp[j] + 1) for all j where nums[j] < nums[i] and j < i. From nums, find the longest sequence that's smaller than nums[i].
The third component is the simplest: we need a base case. For this problem, we can initialize every element of dp to 1 since every element on its own is technically an increasing subsequence.
(1) Initialize an array dp with length nums.length and all elements equal to 1. dp[i] represents the length of the longest increasing subsequence that ends with the element at index i.
(2) Iterate from i = 1 to i = nums.length - 1. At each iteration, use a second for loop to iterate from j = 0 to j = i - 1 (all the elements before i). For each element before i, check if that element is smaller than nums[i]. If so, set dp[i] = max(dp[i], dp[j] + 1).
(3) Return the max value from dp.
- Predict the winner for coin game



Head or Tail?

- Two players: Assume that Bob will play optimally.
- For Alice's kth turn:
  - Coin array: $C[i..j]$
  - $\max\{c[i] + \text{BobOpt}(c[i+1..j]), c[j] + \text{BobOpt}(c[i..j-1])\}$
- $\text{BobOpt}(c[i..j]) :=$
  $\min\{\text{AliceOpt}(c[i+1..j]), \text{AliceOpt}(c[i..j-1])\}$

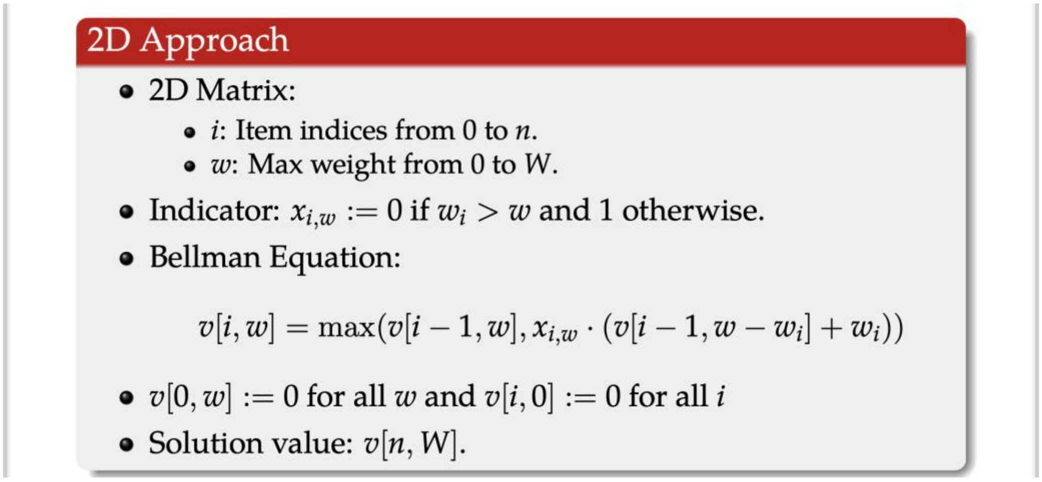
Max{head+what choice Alice get after Bob's next move, tail+what choice Alice get after Bob's next move}
2D array M:
M[i, j] is the maximum value possible for Alice when choosing from c[i..j], assuming Bob plays optimally.
Bellman Equation:
M[i, j] = max{c[i] + min{M[i + 2, j], M[i + 1, j − 1]}, c[j] + min{M[i + 1, j − 1], M[i, j − 2]}}
M[i, i] = c[i] for all i.
M[i, j] = max{c[i], c[j]} for i = j − 1.
Populate i from n − 2 to 1; j from n to 3. Solution: M[1, n]
Runtime: O(n2)
- Subset of jobs



Problem Definition

- A single machine that we can use for time W.
- A set of jobs: $1, 2, \ldots, n$.
- Each job has a run time: $w_1, w_2, \ldots, w_n$.
- What is the subset S of jobs to run that maximizes $\sum_{i \in S} w_i \leq W$?

Max(no ith job, use ith job and wi<w)



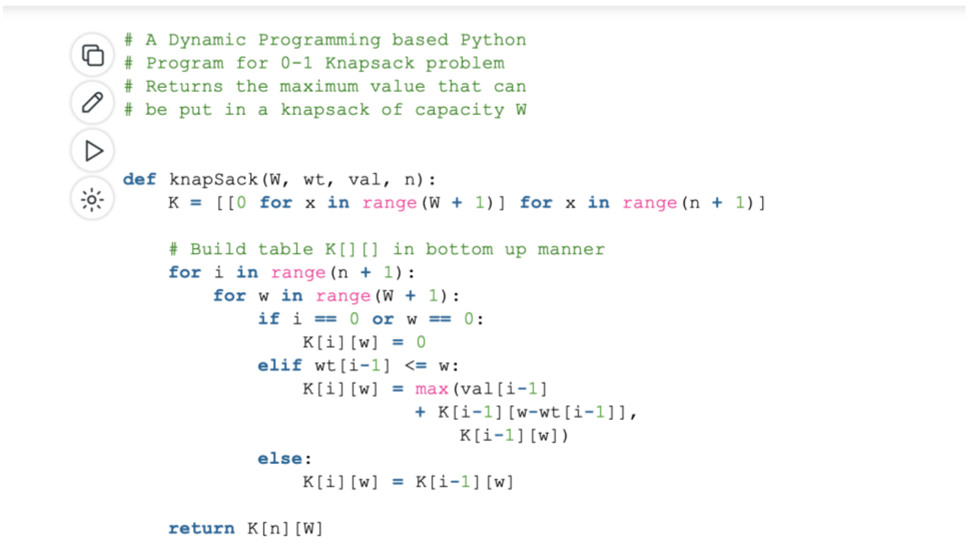
2D Approach

- 2D Matrix:
  - i: Item indices from 0 to n.
  - w: Max weight from 0 to W.
- Indicator: $x_{i,w} := 0$ if $w_i > w$ and 1 otherwise.
- Bellman Equation:

$$v[i,w] = \max(v[i-1,w], x_{i,w} \cdot (v[i-1,w-w_i] + w_i))$$

- $v[0,w] := 0$ for all w and $v[i,0] := 0$ for all i
- Solution value: $v[n,W]$.

- Knapsack Extended:



```
# A Dynamic Programming based Python
# Program for 0-1 Knapsack problem
# Returns the maximum value that can
# be put in a knapsack of capacity W


def knapSack(W, wt, val, n):
    K = [[0 for x in range(W + 1)] for x in range(n + 1)]

    # Build table K[][] in bottom up manner
    for i in range(n + 1):
        for w in range(W + 1):
            if i == 0 or w == 0:
                K[i][w] = 0
            elif wt[i-1] <= w:
                K[i][w] = max(val[i-1]
                          + K[i-1][w-wt[i-1]],
                            K[i-1][w])
            else:
                K[i][w] = K[i-1][w]

    return K[n][W]
```

In a DP[][] table let’s consider all the possible weights from ‘1’ to ‘W’ as the columns and weights that can be kept as the rows.
The state DP[i][j] will denote maximum value of ‘j-weight’ considering all values from ‘1 to ith’. So if we consider ‘wi’ (weight in ‘ith’ row) we can fill it in all columns which have ‘weight values > wi’. Now two possibilities can take place: (1) Fill ‘wi’ in the given column. (2) Do not fill ‘wi’ in the given column. Now we have to take a maximum of these two possibilities, formally if we do not fill ‘ith’ weight in ‘jth’ column then DP[i][j] state will be same as DP[i-1][j]; but if we fill the weight, DP[i][j] will be equal to the value of ‘wi’+ value of the column weighing ‘j-wi’ in the previous row. So, we take the maximum of these two possibilities to fill the current state.
-Edit distance:
m: Length of str1 (first string)
n: Length of str2 (second string)
If last characters of two strings are same, nothing much to do. Ignore last characters and get count for remaining strings. So we recur for lengths m-1 and n-1.
Else (If last characters are not same), we consider all operations on ‘str1’, consider all three operations on last character of first string, recursively compute minimum costs for all three operations and take minimum of three values.

Insert: Recur for m and n-1
Remove: Recur for m-1 and n
Replace: Recur for m-1 and n-1
2D array E, where E[i, j] is the edit distance for A[1..i] and B[1..j].
Solution in E[m, n]

$$E[i,j] = \begin{cases} i, \text{ if } j = 0 \\ j, \text{ if } i = 0 \\ \min\{E[i,j-1]+1, E[i-1,j]+1, \\ \qquad E[i-1,j-1]+A[i] \neq B[j]\}, \text{ otherwise} \end{cases}$$

Set E[0,j] = i; E[i,0] = j; populate from 1 to n, 1 to m. Run time: O(mn)

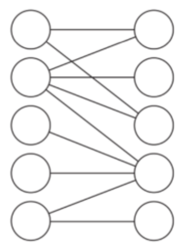
4. Network Flow

Note: I didn't put much tricks here because network flow is very problem-specific. I recommend understand the classical examples in the lecture and go over the questions.

Max flow = minimum cut = Partition of V into sets (A, B) with s ∈ A and t ∈ B.

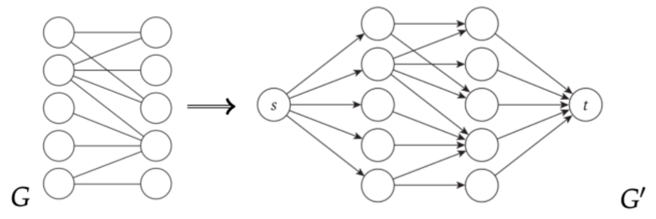
- Cut capacity: $c(A,B) = \sum_{e \text{ out of } A} c_e$

Bipartite:



Definition

- Bipartite Graph $G = (V = X \cup Y, E)$.
- All edges go between X and Y.
- Matching: $M \subseteq E$ s.t. a node appears in only one edge.
- Goal: Find largest matching (cardinality).



- Add source connected to all X.
- Add sink connected to all Y.
- Original edges go from X to Y.
- Capacity of all edges is 1.

Theorem 10

$|M^*|$ in G is equal to the max-flow of G', and the edges carrying the flow correspond to the edges in the maximum matching.

Proof.

- s can send at most 1 unit of flow to each node in X.
- Since $f^{\text{in}} = f^{\text{out}}$ for internal nodes, Y nodes can have at most 1 flow from 1 node in X.

Node Demand:

Flow Network with Demand

- Each node has a demand d_v:
  - if $d_v < 0$: a source that demands $f^{\text{in}}(v) - f^{\text{out}}(v) = d_v$.
  - if $d_v = 0$: internal node $(f^{\text{in}}(v) - f^{\text{out}}(v) = 0)$.
  - if $d_v > 0$: a sink that demands $f^{\text{in}}(v) - f^{\text{out}}(v) = d_v$.
- S is the set of sources $(d_v < 0)$.
- T is the set of sinks $(d_v > 0)$.

Flow Conditions

i. Capacity: For each $e \in E$, $0 \le f(e) \le c_e$.
ii. Conservation: For each $v \in V$, $f^{\text{in}}(v) - f^{\text{out}}(v) = d_v$.

Goal

Feasibility: Does there exist a flow that satisfies the conditions?

Lemma 10

If there is a feasible flow, then $\sum_{v \in V} d_v = 0$.

Proof.

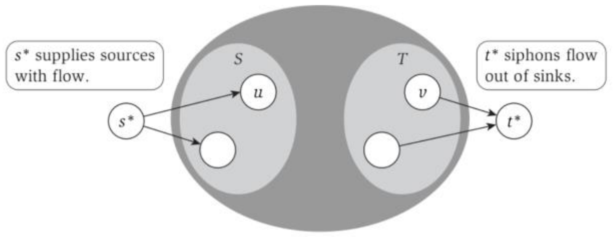
- Suppose that f is a feasible flow, then, by definition, for all v, $d_v = f^{\text{in}}(v) - f^{\text{out}}(v)$.
- For every edge $e = (u, v)$, $f_e^{\text{out}}(u) = f_e^{\text{in}}(v)$. Hence, $f_e^{\text{in}}(v) - f_e^{\text{out}}(u) = 0$.
- $\sum_{v \in V} d_v = 0$.

Corollary 11

If there is a feasible flow, then

$$D = \sum_{v: d_v > 0 \in V} d_v = \sum_{v: d_v < 0 \in V} -d_v$$

Demand of the sinks = -(Demand of the sources)



s^* supplies sources with flow.
S
u
s^*
T
v
t^*
t^* siphons flow out of sinks.

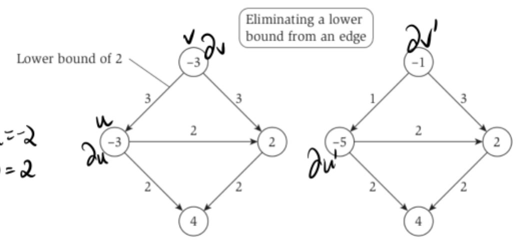
Reduction from G (demands) to G' (no demands)

- Super source s^*: Edges from s^* to all $v \in S$ with $d_V < 0$ with capacity $-d_v$.
- Super sink t^*: Edges from all $v \in T$ with $d_V > 0$ with capacity d_v to t^*.
- Maximum flow of $D = \sum_{v: d_v > 0 \in V} d_v = \sum_{v: d_v < 0 \in V} -d_v$ in G' shows feasibility.

Reduction to Only Demand

$l_{u,v} = 2$

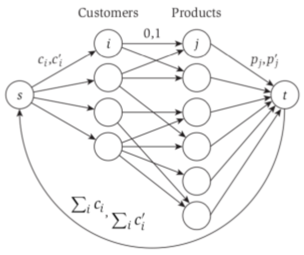
How much flow has to go into that node in order to satisfy the lower bound. $\left\{ \begin{array}{l} L_v = 0-2=-2 \\ L_u = 2-0=2 \end{array} \right.$



Step 1: Reduction from G (demand + LB) to G' (demand)

- Consider an f_0 that sets all edge flows to ℓ_e:
  $L_v = f_0^{\text{in}}(v) - f_0^{\text{out}}(v)$.
  - if $L_v = d_v$: Condition is satisfied.
  - if $L_v \neq d_v$: Imbalance.
- For G':
  - Each edge e, $c'_e = c_e - \ell_e$ and $\ell_e = 0$.
  - Each node v, $d'_v = d_v - L_v$.

- Survey Design:

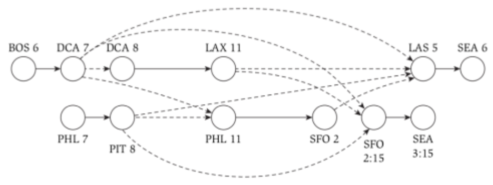


Solution

- Feasibility means it is possible to meet the constraints.
- Edge (i, j) carries flow if customer i asked about product j.
- Flow (t, s) overall # of questions.
- Flow (s, i) # of products evaluated by customer i.
- Flow (j, t) # of customers asked about product j.

-Airplane:

$k = 2$ planes



Reduction

- Units of flow correspond to airplanes.
- Each edge of a flight has capacity $(1, 1)$.
- Each edge between flights has capacity of $(0, 1)$.
- Add node s with edges to all origins with capacity of $(0, 1)$.
- Add node t with edges from all destinations with cap $(0, 1)$.
- Edge (s, t) with a min of 0 and a max of k.
- Demand: $d_s = -k, d_t = k, d_v = 0 \forall v \in V \setminus \{s, t\}$.

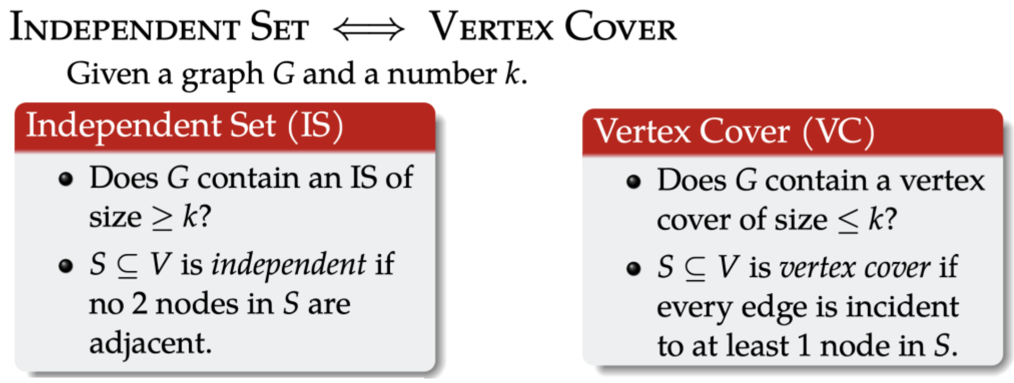
5. NP

Classical Case*: Focus on these: **3-SAT, Independent Set, Vertex Cover, 3D Matching, Set Cover**; Go over others if you have time.

What is 3-SAT(book #21): https://math.stackexchange.com/questions/86210/what-is-the-3-sat-problem

HW11 HW11

What is Independent Set(book #4) and Vertex Cover(book #10):



INDEPENDENT SET $\iff$ VERTEX COVER

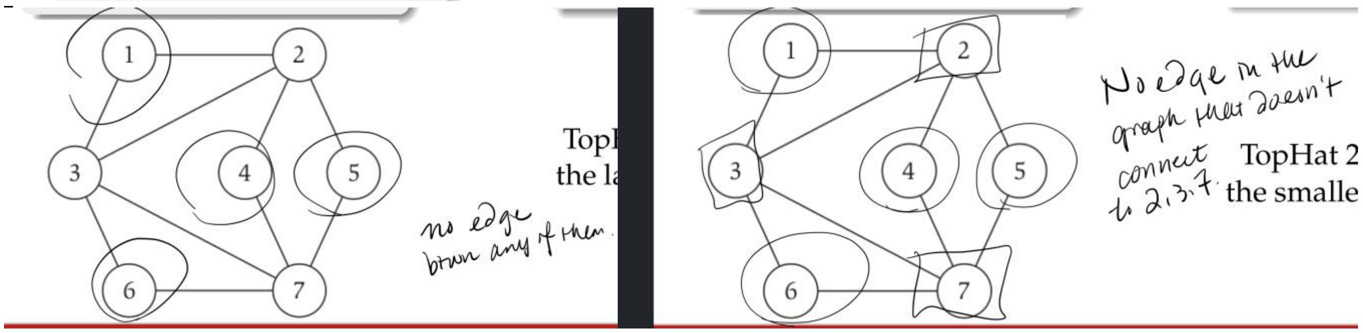
Given a graph G and a number k.

Independent Set (IS)

- Does G contain an IS of size $\geq k$?
- $S \subseteq V$ is *independent* if no 2 nodes in S are adjacent.

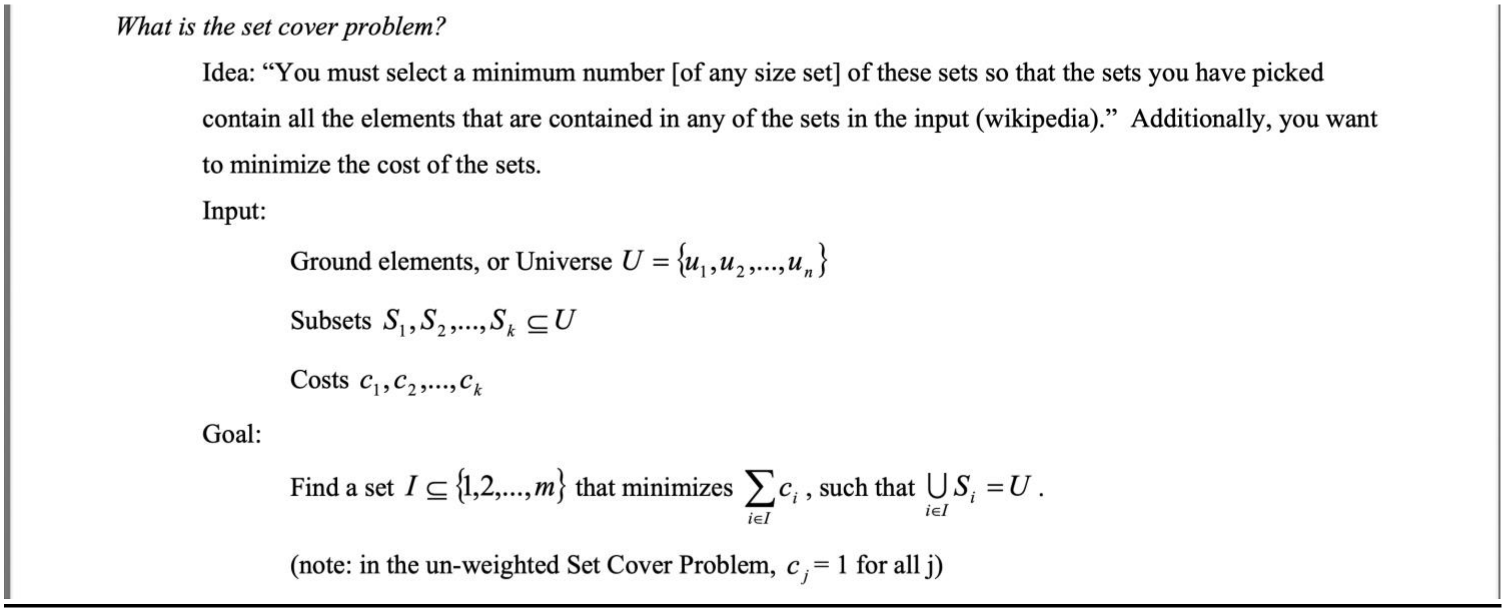
Vertex Cover (VC)

- Does G contain a vertex cover of size $\leq k$?
- $S \subseteq V$ is *vertex cover* if every edge is incident to at least 1 node in S.


1
2
3
4
5
6
7
Top
the la
no edge btwn any of them.
1
2
3
4
5
6
7
No edge in the graph that doesn't connect to 2,3,7.
TopHat 2
the smalle

Independent Set: Goal is to pack as many vertices as possible without violating edge constraints.
Vertex Cover: Goal is to cover all the edges in the graph using as few vertices as possible.

What is Set Cover(book #24) [ignore the cost constraint, we didn't reach this far in class]:



What is the set cover problem?

Idea: "You must select a minimum number [of any size set] of these sets so that the sets you have picked contain all the elements that are contained in any of the sets in the input (wikipedia)." Additionally, you want to minimize the cost of the sets.

Input:

Ground elements, or Universe $U = \{u_1, u_2, \ldots, u_n\}$

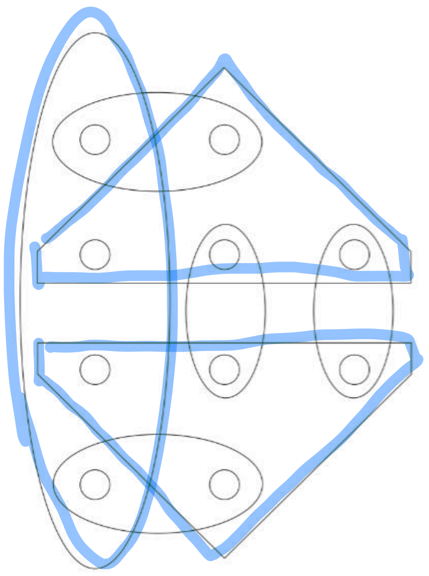
Subsets $S_1, S_2, \ldots, S_k \subseteq U$

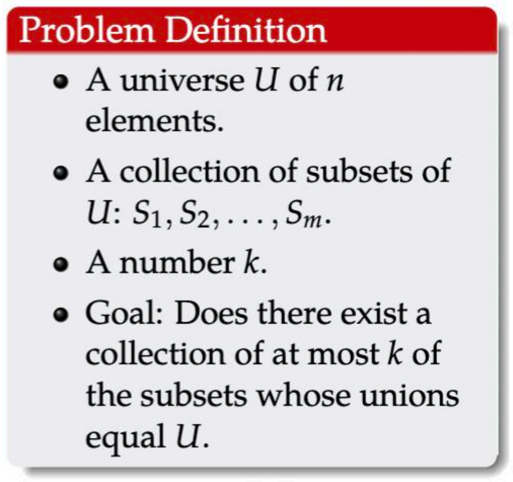
Costs $c_1, c_2, \ldots, c_k$

Goal:

Find a set $I \subseteq \{1,2,\ldots,m\}$ that minimizes $\sum_{i \in I} c_i$, such that $\bigcup_{i \in I} S_i = U$.

(note: in the un-weighted Set Cover Problem, $c_j = 1$ for all j)

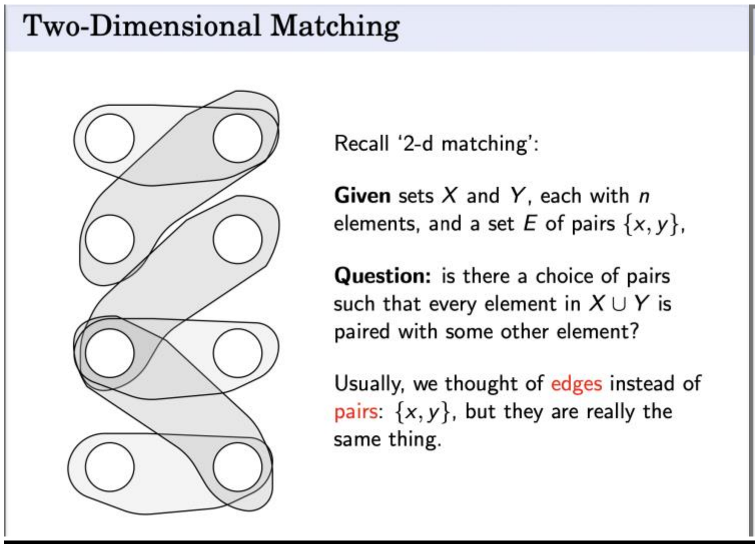




Problem Definition

- A universe U of n elements.
- A collection of subsets of U: $S_1, S_2, \ldots, S_m$.
- A number k.
- Goal: Does there exist a collection of at most k of the subsets whose unions equal U.

What is 3D Matching (https://www.cs.cmu.edu/~ckingsf/bioinfo-lectures/3dm.pdf):



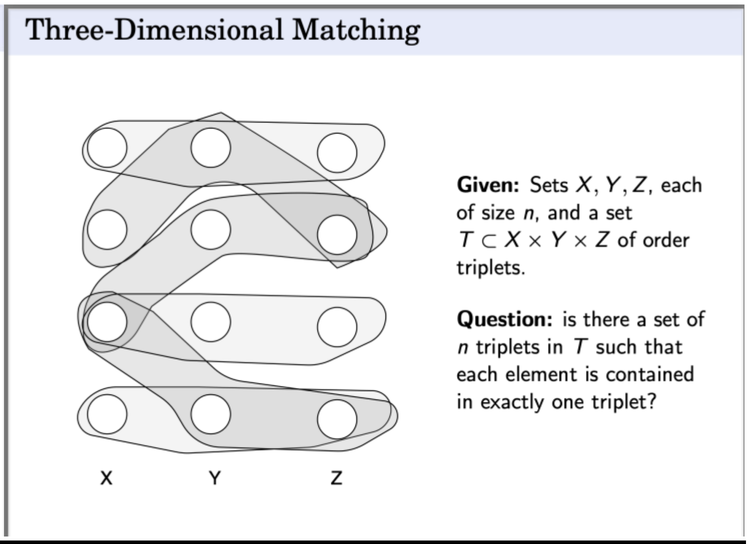
Two-Dimensional Matching

Recall '2-d matching':

Given sets X and Y, each with n elements, and a set E of pairs $\{x, y\}$,

Question: is there a choice of pairs such that every element in $X \cup Y$ is paired with some other element?

Usually, we thought of edges instead of pairs: $\{x, y\}$, but they are really the same thing.



Three-Dimensional Matching

Given: Sets X, Y, Z, each of size n, and a set $T \subset X \times Y \times Z$ of order triplets.

Question: is there a set of n triplets in T such that each element is contained in exactly one triplet?

How to answer this type of questions? Follow this template:

1. The general Your_Problem (YP) is in NP.

Given a set of k __whatever kind of resource you're given in problem__, we can verify that __whatever goal you're having in the problem__ in polynomial time. __Explain how you're gonna verify__. This can be done in time O(km). If ___, the solution is valid, if not, the solution is invalid. Since a proposed solution can be confirmed or rejected in polynomial time, this problem is in NP. pick a Classical_Case ≤ NP-Complete.

2. The general YP is NP-Hard. (**Classical_Case***≤p YP)

An arbitrary instance of Classical_Case is defined on ___A graph/set of clauses/etc.___and an integer k.

(State how to fit the scenario into your problem)

• Let __whatever element in Classical_Case__ be __whatever element__ in YP

•…

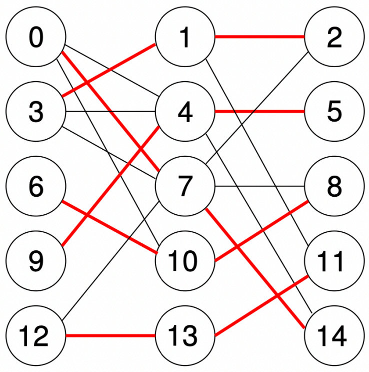
•…

This transformation can be done in polynomial time, since __briefly explain that you are gonna go thru whatever thing and that's polynomial time___. In this way, we transform an arbitrary instance of Classical_Case into an instance of YP.
Proof IFF (if and only if):
(⇒) If we have a k xxx correspond to Classical_Case, then this means that ___explain what does it mean in the classical case scenario___, which means that __why this also applies/works/fulfilles the requirement to our scenario__ by the way we transformed the instance above. So a yes instance for Classical_Case is a yes instance for YP.
(⇐) If we have k xxx that __fulfills your problem statement/requirement___, then the k xxx that correspond to those xxx must ___ (explain why the k works in our scenario can also be applied/fulfilling the Classical_Case). So a yes instance for the YP is a yes instance for Classical_Case. Thus, we have a polynomial time mapping from Classical_Case to YP, in which we have a yes instance for Classical_Case if and only if we have a yes instance for YP. Since Classical_Case is NP-Complete, YP is NP-Hard.

Another Example to 3D matching:

a solution to the example

$X = \{0, 3, 6, 9, 12\}$, $Y = \{1, 4, 7, 10, 13\}$, $Z = \{2, 5, 8, 11, 14\}$.



A solution: $\{(0,7,14), (3,1,2), (6,10,8), (9,4,5), (12,13,11)\}$.