Topics Covered

- Stable Matching
- Greedy Algorithms
- Dynamic Programming
- Divide and Conquer
- Randomized Algorithms

Stable Matching

Stable Matching Problem

Consider a set $M = \{m_1, \ldots, m_n\}$ of n men and a set $W = \{w_1, \ldots, w\_n\}$ of n women. Each of them has a preference list of the other gender. Now, we want to find a stable matching S between men and women.

A matching is stable if

1) it is perfect
2) there is no instability
   a) Instability: We match (m, w') and (m', w) but m prefers w to w' and w prefers m to m'.

Example: Is it stable?

	1st	2nd	3rd
m1	w1	w2	w3
m2	w2	w1	w3
m3	w1	w2	w3

	1st	2nd	3rd
w1	m2	m1	m3
w2	m1	m2	m3
w3	m1	m2	m3

Example: Is it stable?

	1st	2nd	3rd
m1	w1	w2	w3
m2	w2	w1	w3
m3	w1	w2	w3

	1st	2nd	3rd
w1	m2	m1	m3
w2	m1	m2	m3
w3	m1	m2	m3

No!

m_1 prefers w_2 to w_3 and w_2 prefers m_1 to m_2

Gale-Shapley Algorithm

- Every man proposes to women in the order he prefers.
- Women wait for a proposal. (Suppose m proposes to w)
  - If w is free, w becomes engaged to m.
  - Else w is currently engaged to m', w becomes engaged only if w prefers m' to m.
- Keep running this algorithm when there is a man who is free and hasn't proposed to every woman.
- (Refer to textbook page 6 for more details and proof)
- Q: Is it possible that there are still unmatched men or women after this algorithm finishes?

Quick Properties of G-S

- It always returns a perfect, stable matching
- If the same people propose, the stable matching returned is the same matching
- The proposers are always matched to their best valid match
- The respondents are always matched to their worst valid match

Example Problem

Consider a version of the stable matching problem in which men and women can be indifferent between certain options. As before we have a set M of n men and a set W of n women. Assume each man and each woman ranks the members of the opposite gender – but now we allow ties in the ranking. We will say that w prefers m to m' if m is ranked higher than m' on her preference list (they are not tied).

We define that a **strong instability** in a perfect matching S consists of a man m and a woman w, such that m and w prefer each other to their partners in S. Find an O(n^2) algorithm that is guaranteed to find a perfect matching with no strong instability.

Hint: Reduce this problem to the vanilla stable matching problem.

Solution

Reduce input of P to a stable matching problem.

Men: ✔

Women: ✔

Preferences: We break the ties lexicographically - that is if a man m is indifferent between two women w_i and w_j then w_i appears on m's preference list before w_j if i < j and if j < i w_j appears before w_i.

Time complexity: O(n^2)

Solution

Run Gale-Shapley algorithm and get a stable matching.

Time complexity: O(n^2)

Solution

We claim that the stable matching found by Gale-Shapley algorithm is a perfect matching with no strong instability in the modified stable matching problem.

Suppose we have a strong instability in the modified problem. It means there exist m and w such that they prefer each other to their partners. It implies an instability in the instance of the vanilla stable matching problem we generated. However, Gale-Shapley algorithm is guaranteed to find a stable matching. By contradiction, there is no strong instability in the modified stable matching problem.

Greedy Algorithms

Algorithms We've Covered

- Interval Scheduling
- Minimum Spanning Tree
  - Definitely be familiar (and know how to run) Kruskal and Prims

Proof Techniques to know

- Greedy Stays Ahead
- Exchange Argument
- Minimax

Interval Scheduling

Interval Scheduling (Section 4.1) Setup and Greedy Rule

Given a set of ***n*** requests, as well as start time s(i) and finish time f(i) for any request i, schedule as many requests as possible

What is the optimal greedy rule we should use? Earliest Finish Time

Interval Scheduling Algorithm

R = set of requests, A = {}
sort R by increasing finish time
while R is not empty:
    choose the next request *i* ∈ R
    add *i* to A
    delete requests from R that conflict with *i*
output A

Runs in **O(n log n)**. Sorting by finishing time (using merge or quicksort) takes time **O(n log n)**. In an additional **O(n)** time, we construct the array S[1..n] where S[i] contains the value s(i). Now, we select requests by processing in order of increasing f(i). We always select the first interval, then we reach the first interval j for which s(j) ≥ f(1). In general, if the most recent interval we've chosen ended at f, we iterate through subsequent intervals until we reach first request, j, for which s(j) ≥ f. This is done through one pass of intervals which is **O(n)**. Thus, overall time is **O(n log n).**

Intuitive Explanation: Get the resource available asap after completing one task.

Interval Scheduling Proof: Greedy Stays Ahead

- What are we trying to prove for this problem?
  - The size of the output set of our greedy algorithm is equal to the size of an optimal set of intervals.

- How to prove?
  - Compare your greedy algorithm solution to an optimal solution and show how it **"stays ahead"**
  - **Need to define what it means to "stay ahead"**
  - **Induction!**

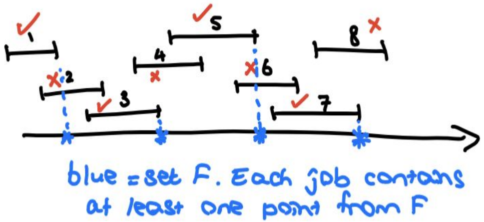
Proof (cont)

- For the Interval Scheduling problem, what does it mean for our greedy algorithm to "stay ahead"
  - Let i_k and j_k be the k^{th} interval in the greedy algorithm set and optimal set of requests, respectively.
  - Want to show that for $f(i_k) \leq f(j_k)$ for all k
  - Can show this using induction
    - Detailed proof on page 120 in the textbook

Interval Scheduling Proof: Minimax

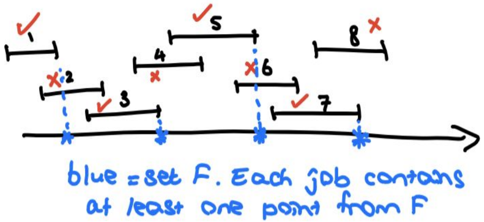
- Suppose we select *m* out of *n* jobs, and we want to show that no more than *m* jobs can be selected.
- Claim: Let *F* = {*f*(*i*) : job *i* selected}. Then the interval for every job *j* in the input contains some element of *F*.

job 2 contains f(1), job 4 contains f(3), job 6 contains f(5), job 7 contains f(8).



Interval Scheduling Proof: Minimax

- Suppose we select *m* out of *n* jobs, and we want to show that no more than *m* jobs can be selected.
- Claim: Let $F = \{f(i) : \text{job } i \text{ selected}\}$. Then the interval for every job *j* in the input contains some element of *F*.
  job 2 contains f(1), job 4 contains f(3), job 6 contains f(5), job 7 contains f(8).
- Proof:
  (a) if job *j* was selected, then f(*j*) is in by *F* by definition.
  (b) if job *j* was not selected, then it was eliminated due to conflict with an "earlier" element of *F*:
  To show this formally, you can examine the way we iterated through the intervals in the example specification of the algorithm: we picked an interval to include (category a), and then passed on all the intervals that conflicted with its finish time (category b).



Finishing up Minimax

- Claim: Let $F = \{f(i) : \text{job } i \text{ selected}\}$.
  Then, the interval for every job in the input contains some element of F. Proved!
- How does F restrict the size of any non-overlapping subset of intervals?

Finishing up Minimax

- Claim: Let $F = \{f(i) : \text{job } i \text{ selected}\}$.
  Then, the interval for every job in the input contains some element of F. Proved!
- How does F restrict the size of any non-overlapping subset of intervals?
  F is of size m, so if the set G had $m + 1$ jobs, two of them would contain the same point in F. Conflict!

Finishing up Minimax

- Claim: Let $F = \{f(i) : \text{job } i \text{ selected}\}$.
  Then, the interval for every job in the input contains some element of F. Proved!
- How does F restrict the size of any non-overlapping subset of intervals?
  F is of size m, so if the set G had $m + 1$ jobs, two of them would contain the same point in F. Conflict!
- Hence, there are no non-overlapping subsets of intervals of size greater than m.
- We found an overlapping subset of size m, so this shows optimality.

Minimum Spanning Tree

Minimum Spanning Tree (Section 4.5) Setup

Given an (undirected!) graph **G = (V,E)** with costs for each edge, find a subset of the edges T⊆E so that the graph (V, T) is connected **and** the total cost of the edges in T is as small as possible

- Goals:
  a. Find a spanning tree, T of G.
     - **Spanning tree**? All nodes in V are connected through some set of edges in T.
     - Note that T won't have cycles. Why?
  b. Find the cheapest spanning tree

Greedy algorithm #1 - Kruskal's Algorithm


```
sortedE <- sort E by increasing cost (smaller cost edges earlier in list)
T = {}
for edge, e,  in sortedE:
        Add e to T if it does not create a cycle
output T
```


Runs in O(|E| log |V|). To achieve this, we need to use the Union-Find data structure. If you want more details on this, see Section 4.6.

Greedy algorithm #2 - Prim's Algorithm

Select a root node, *s*, from which to build the spanning tree.

Let S = {s}

Let T = {}

while S ≠ V:

Select *e* = *(u, v)* where *e* is the min-cost edge such that *u* ∈ S and v ∈ V \ S

Add *e* to T

Add *v* to S

Return T

Runs in O(|E| log |V|). This runtime is achieved by using a heap-based priority queue

Key concepts for proof

- Cut property -Let *S* be a nonempty subset of nodes not equal to *V*. Let *e* = *(v,w)* be the minimum-cost edge with one end in *S* and the other in *V* - *S*. **Every MST contains *e*!**

- Exchange argument - used to prove cut property
  - More details on page 145 in the textbook
  - Summary: If *e* (defined above) is not in our spanning tree *T*, find an edge *e'* in *T* such that T - {*e'*} U {*e*} is still a spanning tree and c(e')>c(e). **Exchange** *e'* with *e* to get *T'* so we get cost of *T'* < cost of *T*.

- Cut property used for optimality proof for Kruskal, Prim

Minimum Spanning Tree - Optimality

Cut Property:

(4.17 from K&T) Assume that all edge costs are distinct. Let S be any subset of nodes that is neither empty nor equal to all of V, and let edge e = (v, w) be the minimum- cost edge with one end in S and the other in V − S. Then every minimum spanning tree contains the edge e.

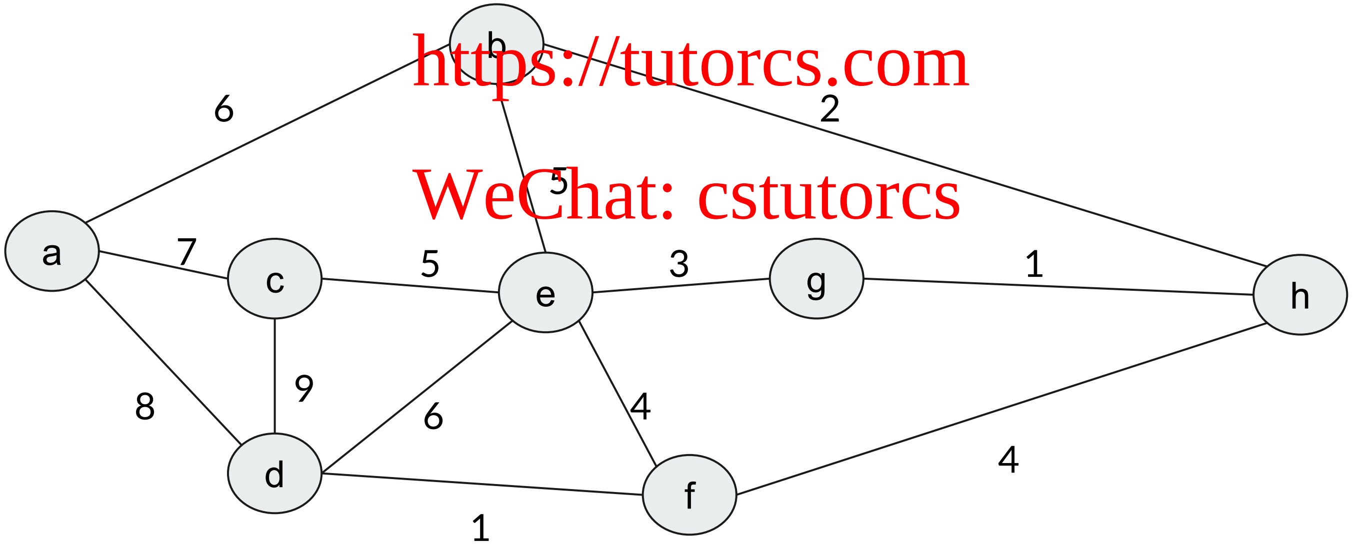
In K&T, the proofs of Kruskal's and Prim's algorithms both rely on the cut property. They argue at that each iterative selection is guaranteed to be in the MST with the cut property.

Minimum Spanning Tree - Cycle Property

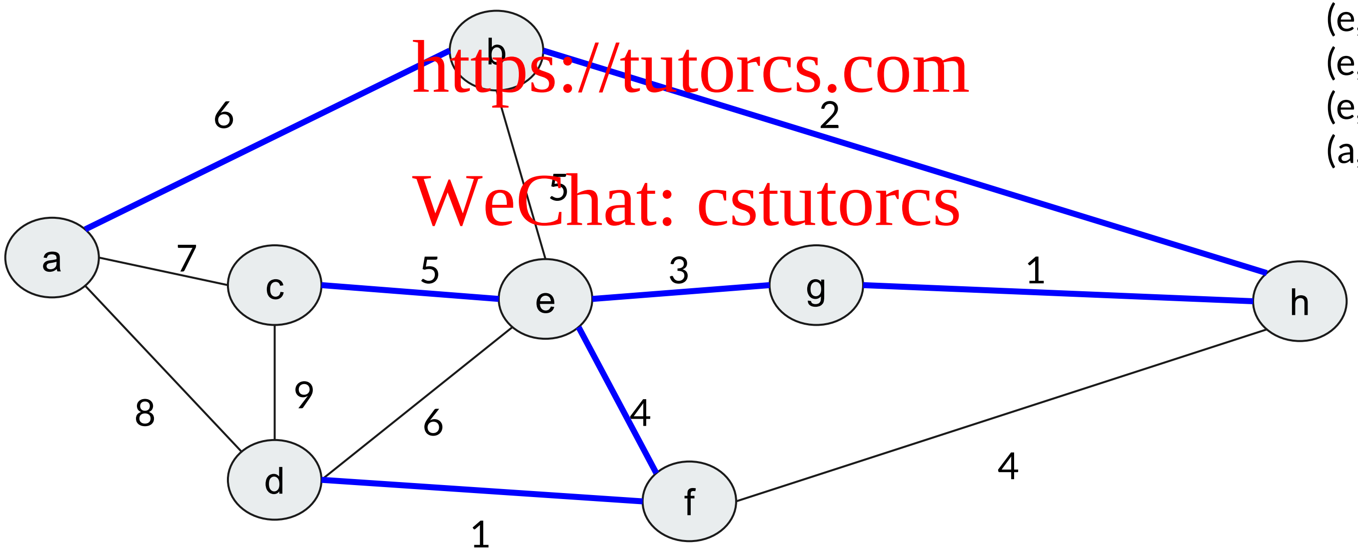
(4.20) *Assume that all edge costs are distinct. Let C be any cycle in G, and let edge $e = (v, w)$ be the most expensive edge belonging to C. Then e does not belong to any minimum spanning tree of G.*

Used to show how an edge is not in the minimum spanning tree

MST example: Find an MST using Kruskal and Prim's algorithm (start at node e)

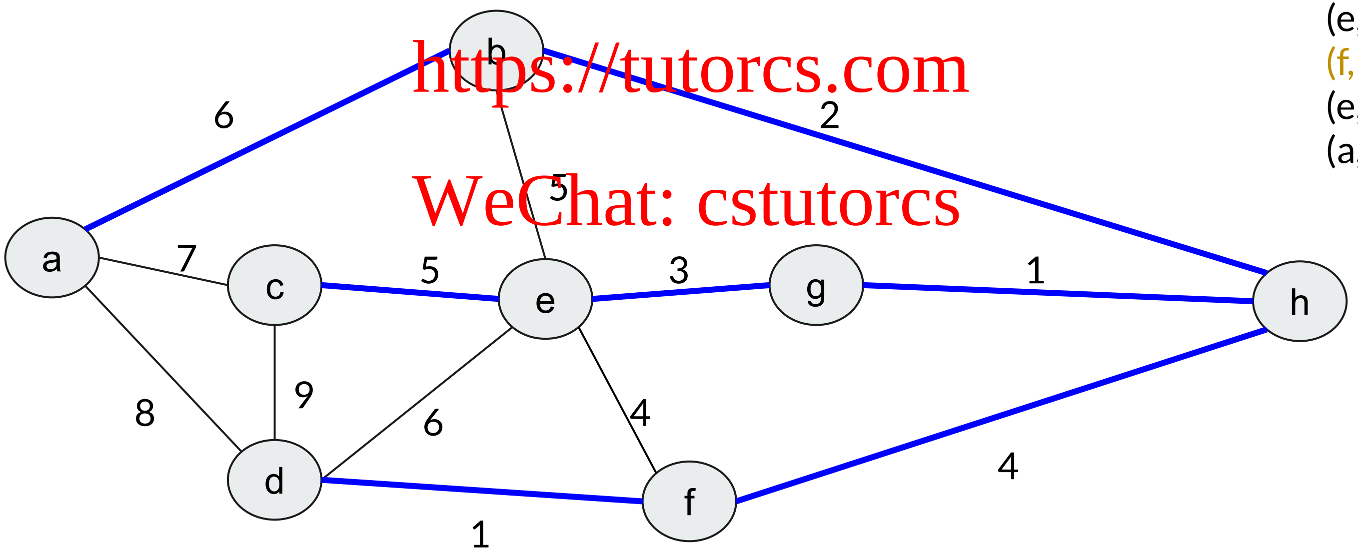


Kruskal's solution #1



Order
(g, h)
(d,f)
(b, h)
(e, g)
(e, f)
(e, c)
(a, b)

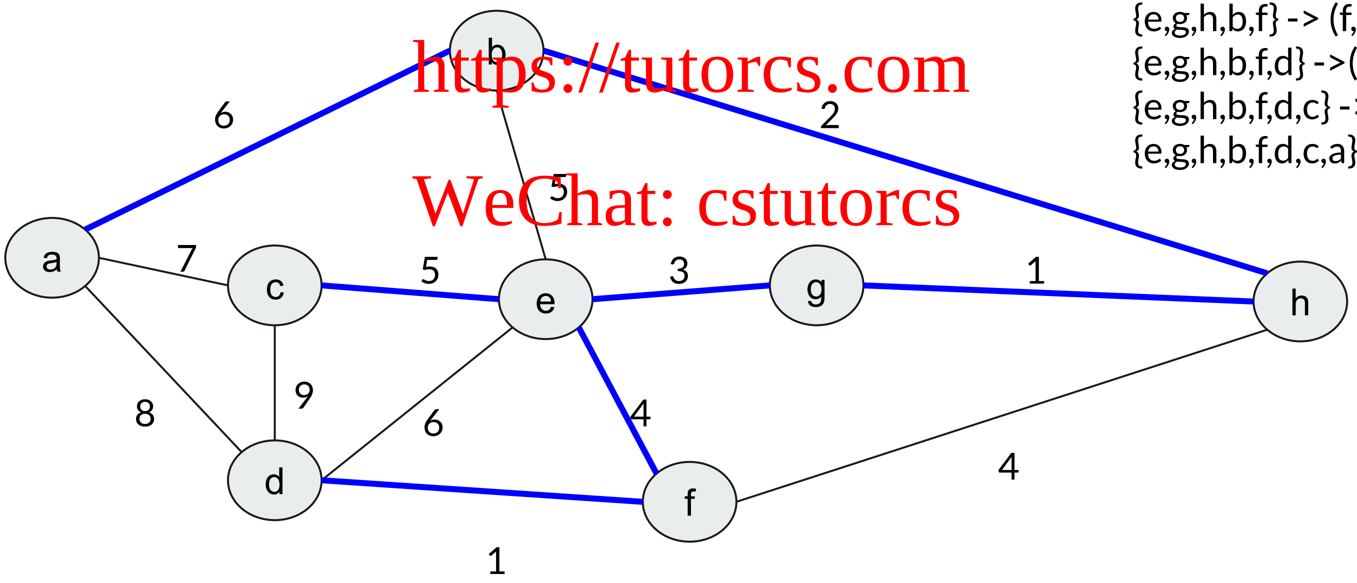
Kruskal Solution #2



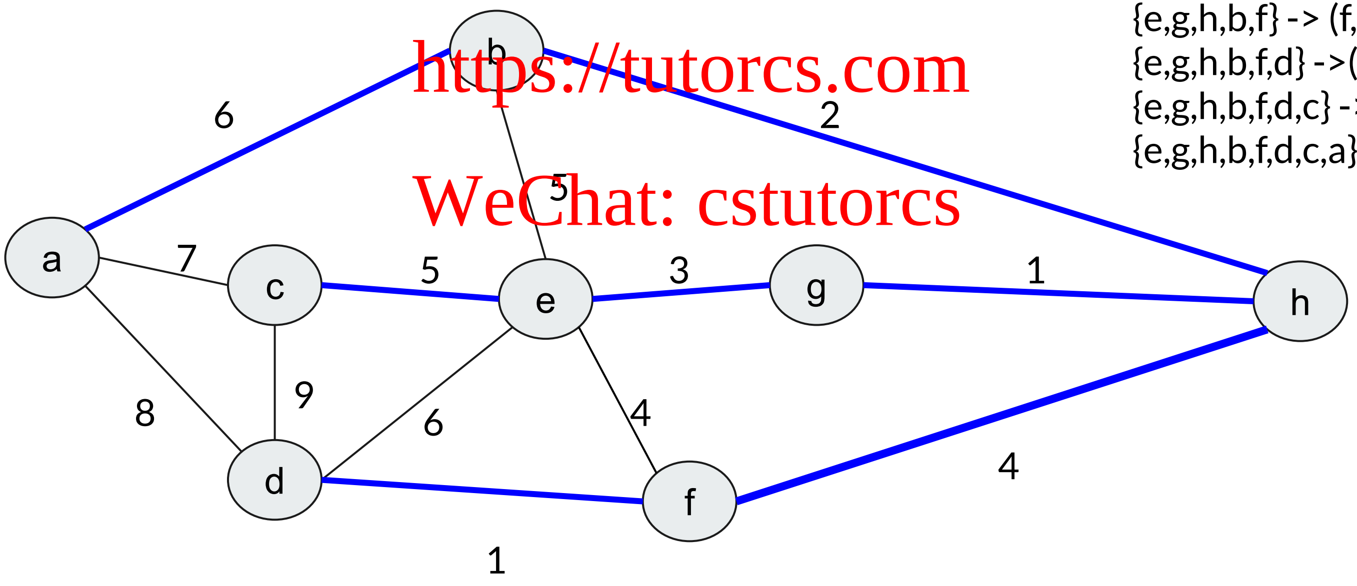
Order
(g, h)
(d,f)
(b, h)
(e, g)
(f, h)
(e, c)
(a, b)

Prim's solution #1

{e} -> (e,g)
{e,g} -> (g, h)
{e,g,h} -> (h,b)
{e,g,h,b}. -> (e, f)
{e,g,h,b,f} -> (f, d)
{e,g,h,b,f,d} ->(e,d)
{e,g,h,b,f,d,c} -> (b,a)
{e,g,h,b,f,d,c,a} Done



Prim's solution #2



{e} -> (e,g)
{e,g} -> (g, h)
{e,g,h} -> (h,b)
{e,g,h,b}. -> (h, f)
{e,g,h,b,f} -> (f, d)
{e,g,h,b,f,d} ->(e,d)
{e,g,h,b,f,d,c} -> (b,a)
{e,g,h,b,f,d,c,a} Done

Practice MST Question

You are given a weighted undirected graph G and a minimum weight spanning tree T for this graph. The graph G has n vertices and m edges. Suppose a new edge (v, w) is added to the graph. Let G_0 denote this new graph. Design an algorithm with running time O(n) to compute a minimum weight spanning tree T_0 of the graph G_0. You may assume that all edge weights are distinct.

Solution

- How is T disrupted if we add (v, w) to it? How can we derive T_0 from this version?

Solution

- If we add (v, w) to T, there must be a cycle, since T now has *n* edges. By the cycle property, the maximum cost edge in this cycle cannot be in the true MST (T_0), so we remove it. We are left with the needed minimum spanning tree T_0

Greedy Algorithm Tips

Tips for designing greedy algorithms

- First, make sure that the problem isn't actually a DP problem!
- What are some metrics that can be used to fuel the greedy rule?
  - e.g. earliest deadline, smallest interval, etc.
- Does it look similar to any of the problems you've encountered this semester?
  - Maybe try a reduction
- Proofs
  - Exchange argument
    - Examples: Section 4.2, Section 4.5 for cut-property
  - Greedy stays ahead (induction)
    - Examples: Section 4.1

General Guideline for Greedy Stays Ahead

1. **Define the solution generated by your algorithm and an optimal solution (could be one of many) to compare it with**
   - Example: Interval Scheduling; A={a_1, a_2, … , a_i} is our solution and an optimal is O = {o_1, o_2, … , o_m}
2. **Define some kind of metric for which your greedy algorithm stays ahead of an optimal solution.**
   - Example: the last finish time of the first *k* jobs in your solution ends before the last finish time of the first *k* jobs in the optimal
   - The metric should also mention the way you are ordering the elements of O.
3. **Use induction to prove greedy stays ahead**
   - Base case
     - Example: r = 1, $f(a_1) \leq f(o_1)$
   - Inductive step
     - Inductive hypothesis: Assume that statement holds for previous step (weak induction)
     - Example: for *r* > 1, assume inductive hypothesis holds for *r* - 1, prove statement holds for *r*.
4. **Use what you've proved about greedy staying ahead about any arbitrary optimal solution to prove that with respect to the metric you've chosen, your solution is optimal**
   - This step should follow naturally from the proof above
   - Example: Your output A in interval scheduling contains same amount of requests as an optimal set

Greedy Stays Ahead Example (K&T Problem 3 in chapter 4)

You are consulting for a trucking company that does a large amount of business shipping packages between New York and Boston. The volume is high enough that they have to send a number of trucks each day between the two locations. **Trucks have a fixed limit W on the maximum amount of weight they are allowed to carry.** Boxes arrive at the New York station one by one, and **each package i has a weight w_i**. The trucking station is quite small, so **at most one truck can be at the station at any time**. Company policy requires that **boxes are shipped in the order they arrive**; otherwise, a customer might get upset upon seeing a box that arrived after his make it to Boston faster. At the moment, the company is using a simple greedy algorithm for packing: **they pack boxes in the order they arrive, and whenever the next box does not fit, they send the truck on its way**.

But they wonder if they might be using too many trucks, and they want your opinion on whether the situation can be improved. Here is how they are thinking. Maybe one could decrease the number of trucks needed by sometimes sending off a truck that was less full, and in this way allow the next few trucks to be better packed.

Prove that, for a given set of boxes with specified weights, the greedy algorithm currently in use actually minimizes the number of trucks that are needed. Your proof should follow the type of analysis we used for the Interval Scheduling Problem: it should establish the optimality of this greedy packing algorithm by identifying a measure under which it "stays ahead" of all other solutions.

Solution

1. **Define the greedy solution and an optimal solution**
   Let A be the total number of trucks our greedy solution uses and O be the total number of trucks an optimal solution uses

2. **Define the metric for which the greedy solution stays ahead.**
   For the first *k* trucks, let our greedy solution pack boxes $b_1, b_2, \ldots, b_m$ while the optimal solution we chose packs boxes $b_1, b_2, \ldots, b_n$. Note that the ordering constraint defined by the problem must be satisfied. That is, if box b_i is sent before box b_j, then b_i came to the company first. Our metric is that $m \geq n$. That is, for a set number of trucks, *k*, our greedy solution packs at least as many boxes as the optimal.

Solution (cont)

3. **Use induction to prove greedy stays ahead**

Base Case: $k = 1$. Our greedy algorithm packs as many boxes as possible into the first truck so this is true.
Inductive hypothesis: Assume for $k-1$ trucks, our greedy solution fits m boxes, the optimal fits n boxes, and $m \geq n$.
Inductive step: need to prove that for k trucks, our greedy solution fits m' boxes and the optimal fits n boxes and $m' \geq n'$.

For the kth truck, the optimal packs in boxes [n+1,..,n']. Since know that $m>=n$ using the inductive hypothesis. This means that our greedy solution can at least pack in boxes [m+1,...,n'] and potentially more since there could still be space as we've packed less total boxes (since n'-(n+1) ≥ n'-(m+1)). This means that if the greedy solution packs boxes [m+1,..,m'], there is a lower bound on m' of n' so m' ≥ n'.

4. **Use what you've proved about greedy staying ahead about any arbitrary optimal solution to prove that with respect to the metric you've chosen, your solution is optimal**

If the total number of trucks our greedy solution uses to pack all boxes is N trucks, our proof shows that we pack at least as many boxes as optimal. Since we actually pack all boxes, this means that N is the smallest number of trucks needed to pack all the boxes.

General Guidelines for Exchange Argument

1. **Define the solution generated by your algorithm and an optimal solution (could be one of many) to compare it with**
   - Example: Let A be the matchings your algorithm chooses. Let O be **an** optimal set of matchings (there could be many optimals!)
2. **Assume that your greedy solution and the optimal solution you choose differ in some way and make sure to specify how. For example:**
   - *O* could include an element A does not include, vice versa.
   - There could be an *inversion* which is a pair of elements that are ordered in reverse order in A and *O*.
3. **"Exchange" elements in O to make it more similar to your solution, A, while preserving the optimality of O**
   - Note the order! You should look to make O more similar to A while preserving optimality, not the other way around!
   - Need to consider all different cases of swapping to show optimality is preserved.
4. **Conclusion: By doing these swaps any time you find differences between O and A, we can eventually eliminate all differences between O and A without worsening the optimality of the solution. This shows that your greedy solution is as good as any optimal solution so it is optimal**

Exchange Argument Example (K&T Problem 7 in chapter 4)

Summary: You have a single large supercomputer as well as an essentially unlimited supply of high-end PCs. Additionally, there is a computation that is broken up in *n* distinct jobs labeled $J_1, J_2, \ldots, J_n$. Each job consists of two stages: first it needs to be preprocessed on the supercomputer and then it needs to be finished on the PCs. For each job J_i, the job needs p_i seconds of time on the supercomputer followed by f_i seconds of time on a PC.

There are at least *n* PCs so the finishing of the jobs can be performed fully in parallel. However, the supercomputer can only work on a single job at at time so you need to figure out an order in which to feed the jobs to the supercomputer. As soon as a job is done on the supercomputer, it is immediately handed off to a PC for finishing; at the same time, the next job is fed into the supercomputer.

A *schedule* is an ordering of the jobs for the supercomputer and the *completion time* of the schedule is the earliest time at which all jobs have finished processing on the PCs.

Give a polynomial-time algorithm that finds a schedule with as small a completion time as possible. Prove that it is optimal.

Solution - Polynomial algorithm

First, we note that since the supercomputer can only work on one job at a time, passing in different orderings of jobs does not affect the **preprocessing time**. Rather, the ordering only affects the total **finishing time.** This intuitively means that when the last job is preprocessed at the supercomputer, we want it to finish as fast as possible.

Let our greedy schedule be: Pass in jobs in order of decreasing **finishing** time so we do the jobs with the largest finishing first.

Solution - Proof of Optimality

1. **Define the solution generated by your algorithm and an optimal solution (could be one of many) to compare it with**
   a. Let A be our greedy solution and let O be any arbitrary optimal schedule.
2. **Assume that your greedy solution and the optimal solution you choose differ in some way and make sure to specify how.**
   a. We define the difference to be an inversion. That is, *O* must contain jobs J_x and J_y such that J_x runs before J_y but $f_x < f_y$. Note that our greedy schedule would have scheduled J_y first before J_x.

Solution - Proof of Optimality (cont)

3. **"Exchange" elements in O to make it more similar to your solution, A, while preserving the optimality of O**
    a. Exchange jobs J_x and J_y so that J_y runs before J_x in *O*. Call this new schedule *O'*. Need to show that *O'* is still optimal.

First, note that the finishing time of all jobs other than J_x and J_y do not change. This is as jobs before J_x and J_y are obviously not affected by this swap. Jobs after J_x and J_y are also not affected as *preprocessing* time of these two jobs combined do not change so the job immediately following J_x and J_y in O and O' starts preprocessing at the same time.

Job J_y is now scheduled earlier so it will finish earlier as it gets handed off to a PC earlier in *O'* than in *O*.

Job J_x is scheduled later. However, the combined preprocessing time of J_x and J_y does not change after the swap so we know that J_x is handed off to the PCs in O' at the same time J_y is handed off to PCs in O. Additionally, $f_x < f_y$ which means Job J_x must finish earlier in O' than J_y in O so the overall time is reduced.

4. **Conclusion**
    - If we do the swap in 3 for all pairs of inversions. We can convert *O* to *A* without increasing the completion time which shows that *A* is optimal.

General Guidelines for Minimax

- Bird's eye view: we are treating this as an optimization problem, by (1) proving a bound and (2) showing that this bound is tight with the output of our algorithm

- Don't forget to mention (2)
- This method is less commonly used

- Some situations when it might be useful:
  - You have trouble writing down an inductive hypothesis to compare ALG to OPT
  - There's a naïve bound on the optimal value, and you want to use this as much as possible
- See *Interval Partitioning*, page 122

Target Sum

Given an integer t and n sets of integers $S_1, S_2, \ldots, S_n$: find the smallest value m such that for $i = 1, \ldots, m$, there exists a sequence of $a_i \in S_i$ with $\sum_i a_i \geq t$.

e.g. $n = 5$, $t = 9$, $S_1 = \{2, 3, 1, \underline{4}\}$, $S_2 = \{-11, \underline{-3}\}$, $S_3 = \{\underline{5}, 2\}$, $S_4 = \{\underline{5}, 4, 3, 1\}$, $S_5 = \{2,3\}$.

ans. $m = 4$, use underlined elements to get a sum of $11 \geq 9$.

Solution - polynomial time algorithm

Initialize $s = 0$, $i = 0$.

while $s < t$:

  increment i by 1

  increment s by max S_i

output i

Target Sum: Proof of Correctness

Minimax: what is the bound on m? why is this bound tight?

Target Sum: Proof of Correctness

Claim: Suppose m_{ALG} is the output of our algorithm. Then, the first m_{ALG} - 1 elements of any sequence sum to less than t.

Proof: Since we summed the maximal elements of S_1 through S_i, this bounds the sum of any valid sequence of numbers a_j for $1 \leq j \leq i$.

Hence, at least m_{ALG} elements are necessary to reach the target. Finally, m_{ALG} elements are sufficient because our algorithm only terminates when $s \geq t$.

In this case, minimax is just “greedy stays ahead” in disguise.

Dynamic Programming

Dynamic Programming (Overview)

Key Idea

- Break a problem into *subproblems*
  - Typically a constrained version of the original problem applied to a small scope of the original input
- Solve larger and larger subproblems to gradually build up to a solution to the original problem

Characteristics

- *Base Case*: How the smallest subproblem is solved
- *Recurrence*: How to solve a larger subproblem using answers to smaller ones
- *Memoization*: Storing the answers to subproblems to use later (and avoid duplicate computation)

DP vs Greedy Algorithms

- Both break a problem down into a series of decisions
- Greedy Algorithms

  - Each time a decision has to be made, choose the best option according to some heuristic based on currently available information
  - Once a decision has been made don't go back on it

- Dynamic Programming
  - Enumerate every possible preceding decision, not just the locally optimal one, and look for the optimal "continuation" of each
  - To make this work in polynomial time, store and reuse values of optimal continuations rather than recomputing them
- When to use a greedy vs dynamic programming?
  - If it's possible for what looks like the "best" option currently to end up being wrong later on, greedy won't work
  - Try out different greedy heuristics and come up with counterexamples

Solving DP Problems (1)

Step 1: Choose appropriate subproblem

- Defined on some reduced version of the original input to the problem
- Desirable Characteristics
  - For a non-base case input, it can be answered efficiently using the answer to the same subproblem on an even smaller version of the input (using a recurrence)
  - The original problem can be answered efficiently using the answer to subproblem
- Need to re-word "Selection" problems as "Numeric" problems
  - Ex. Knapsack:
    - Original Problem: What is the *subset* of S with max value and total weight ≤ W
    - "Numeric" Version: What is the *max value* of a subset of S with total weight ≤ W
  - Bellman Ford: What is the shortest path… → What is the *length* of the shortest path…
- The answer to the problem doesn't have to be a number: Can be a boolean, etc. (HW 3 Question 2)

Examples/Types of Subproblems

The same as the original problem (but on a smaller portion of the input)

- Ex. Weighted Interval Scheduling

    - Original Problem: Find the maximum total weight of a non-overlapping subset of intervals out of intervals {1 … n}
    - Subproblem: OPT[j] = The maximum total weight of a non-overlapping subset of intervals out of intervals {1 … j}

    - OPT[j] can be found efficiently if we already know OPT(k) for all k < j ✅
        - OPT[j] = max(w_j + OPT(p(j)), OPT(j - 1))
    - The answer to the problem can be found efficiently using some subproblem ✅ (it's just OPT[n])
- Typically the first idea to consider
- May need to define the subproblem on more than one parameter to properly specify the scope of the input to be considered
    - (Look at: HW 4, RNA Secondary Structure, String Interleaving (HW 3 Question 2))

Examples/Types of Subproblems

Adding a variable to the subproblem

- Ex. Knapsack

    - Original Problem: Find the max total value of a subset of items with total weight ≤ W chosen from items {1 … n}
    - Subproblem: OPT[i, w] = Max total value of a subset of items chosen from items {1 … i} given that the total weight of the set does not exceed weight limit w

- Trying to have a subproblem which only narrows the portion of input to be considered (restricting to a specific subset of the items) is too general to solve
- Typically if there is some concept of a “capacity” in the problem asking you to select items, adding a variable can be useful to keep track of progress towards exhausting that capacity within the subproblem
- See Also: Bellman-Ford algorithm (used to solve Shortest Path problem)
    - Subproblem: OPT[v, ℓ] is min cost path from s to v using ≤ ℓ edges

Examples/Types of Subproblems

Almost the same as the original problem, but with an added constraint

- Ex. Min Cost File Configuration (Textbook Problem 6-12)
  - Original Problem: Find the minimum total cost of a valid selection of servers from {1 … n}
  - Subproblem: OPT[j] = The minimum total cost of a valid selection of servers from {1 … j} *given that server j is included*
- Useful when the original problem is almost good enough as a subproblem, but a small amount of information is missing to be able to derive a recurrence
- HW 3 Problem 1 is a good example of a problem that combines this type of subproblem with the type where you need to add a variable like knapsack
  - Subproblem: OPT[i, ℓ] = Max value using inputs {1, …, i} given that input i is included using at most ℓ capacity
- Variant: Multiple Subproblems with different added constraints are used to solve each other are solved simultaneously
  - Textbook Problem 6-4

Solving DP Problems (2)

Step 2: Define Base Case(s) and Recurrence

- Base case: specifies the answer to the subproblem on the "smallest" or "first" input(s) to it
    - Base case doesn't depend on the answer to the subproblem on other inputs
    - Typically something like defining OPT[0] or OPT[1]
        - Ex. Weighted Interval Scheduling: OPT[0] = 0
    - If the subproblem uses multiple parameters to define the scope of input considered, base case typically is when the input is restricted to small size
        - Ex. RNA Secondary Structure: OPT[i, j] where i ≥ j - 4 = 0
        - HW 4: 0 if considering a subsection of the bar with one square
    - There can be multiple base cases
        - If your subproblem is 2-Dimensional (i.e. there are two parameters to the subproblem), ex. OPT[i, j], you might need to specify OPT[i, 0] and OPT[0, j]
            - Ex. HW 3 Problem 1

Solving DP Problems (2)

Step 2: Define Base Case(s) and Recurrence

- Recurrence: specifies how to solve the subproblem using the answer to smaller or previous subproblems (using mathematical relationship)

- Typically break the answer to the subproblem into possible cases
  - Ex. for selection problems, commonly cases are either choose a particular object or don't choose it
    - Weighted Interval Scheduling: $OPT[j] = \max(v_j + OPT[p(j)], OPT[j - 1])$
    - Knapsack: $OPT[i, w] = \max(OPT[i - 1, w], v_i + OPT[i - 1, w - w_i])$ if $w < w_i$ else $OPT[i - 1, w]$

Solving DP Problems (3)

Step 3: Determine the order the subproblems will be built up

- Iterative Approach
  - Simply iterate over entries on your DP table with for loops and fill them up using your recurrence relation
  - Useful when there is a very obvious order of dependencies in your DP table
    - ex. OPT[n] depends on OPT[n-1], OPT[n-1] depends on OPT[n-2], ... →
      - Compute OPT[1], then OPT[2], then OPT[3], ... (ex. Weighted Interval Scheduling)
    - In two dimensions, you can fill the table row by row or column by column (ex. Bellman-Ford)
  - Simplistic, easy to write out and follow along
- Recursive/Top Down Approach
  - DP Table is filled with recursive calls, starting from the final subproblem you want to answer
  - Can always be used, even if there's no obvious order of dependencies (i.e. useful when iterative approach fails)
  - To use memoization/avoid duplicate computation, you must be sure to
    a) Store the result of each function call in your OPT table before returning from the call
    b) Check if the input is a base case or the result has already been computed for it and return the result if so at the beginning of the call

Solving DP Problems (4)

Step 4: Determine how to get the answer to the original problem using answer to subproblems

- Likely as simple as retrieving the answer to the subproblem for a specific input: ex. OPT[n]
  - Can also be $\max_j$(OPT[j]) for example

- Getting the answer for a “Selection” problem which was previously re-written as a “Numeric” problem:
  - Backtracking can be made easier by Including in your OPT table which element’s subproblem was used to compute each subproblem while it’s being filled
  - E.g.: Hiring costs programming assignment

Solving DP Problems (5, 6)

Step 5: Proof of Correctness

- Must prove that subproblem was solved correctly (base cases/recurrence) and answer to problem was derived from subproblem correctly

- Inductively prove correctness for problems of larger size
  - Base case of induction proof: Base case of your DP alg
  - Induction Hypothesis: Your subproblem

  - Induction Step: Prove that the subproblem is solved (induction hypothesis holds) assuming it was correct/held on smaller subproblems according to recurrence
- If it’s obvious, a few sentences explanation can suffice

Step 6: Runtime Analysis

Solving DP Problem Process Overview

- Subproblem
- Base cases/Recurrence
- The order subproblems will be built up
- How to get answer to original question
- Proof of correctness
- Runtime analysis

DP Sample Problem: Machine Scheduling (Textbook 6-10)

You have two machines, A and B, and you want to determine a schedule to run them from time 1 to n. Machine A gives value a_i if run at time i and machine B gives value b_i. Only one machine can be run at a time, and to switch which machine is running requires waiting for one unit of time. Determine the maximum value possible that can be achieved (all $a_i, b_i > 0$)

Ex.

	Minute 1	Minute 2	Minute 3	Minute 4
A	10	1	1	10
B	5	1	20	20

Running only A gives 10 + 1 + 1 + 10 = 22
Running only B gives 5 + 1 + 20 + 20 = 46
Optimal Schedule: A, -, B, B gives 10 + 0 + 20 + 20 = 50

Subproblem

First Idea: OPT[i] = Max possible value for a schedule from times 1 to i, OR
Issue: When trying to determine what OPT[i] is, we don't know which machine was run last to give OPT[i - 1], so we don't know if we need to wait to use a certain machine

Second Idea: Include which machine was run last to give the schedule of optimal value within OPT

OPT[i] = (Max possible value for a schedule from times 1 to i, The last machine run to give that value) i.e. stored in a tuple

OPT[i] = The entire schedule of max value from times 1 to i (less efficient than only storing the value)

Issue: Even if we do know which machine was run last to derive a particular value of OPT, we don't know if it would have been better to not have used that machine last (to avoid having to wait). Ex. a = [1, 10]; b = [2, 1]. For i = 1, the optimal schedule starts with B, but for i = 2, it starts with A

We need to know how well we can do if a specific machine was run last to derive a recurrence → Add constraint to our subproblem to tell us this

Final Idea: OPT[i] = (

Max value for a schedule from times 1 to i where machine A is used at time i,

Max value … where machine B is used at time i

)

Base Case/Recurrence

Base Case:

OPT[0] = (0, 0), OPT[1] = $(a_1, b_1$

Recurrence:

To get Max value for 1, … i assuming we end on A, there are two cases:

we use A on t_{i-1}, or we use B on t_{i-2} and wait

OPT[i] = (

a_i + max(OPT[i - 1][0], OPT[i - 2][1],

b_i + max(OPT[i - 1][1], OPT[i - 2][0]

)

Order to build up subproblems

OPT[i] depends on OPT[i-1] and OPT[i-2]

- Clear order of dependency, so we can fill the table from $i = 0 \rightarrow n$ iteratively


```
FILL_OPT():
    Set OPT[0] and OPT[1] according to base cases
    for i from 2 to n:
        Set OPT[i] using recurrence
```

How to get the actual optimal schedule?

We can compute the actual optimal schedule as well, apart from the optimal maximum value.

Use backtracking:

	Minute 1	Minute 2	Minute 3	Minute 4
A	10	1	1	10
B	5	1	20	20

Schedule:
?, ?, ?, ?,

50 = 20 + (11 or 30?)

j↓ i →	0	1	2	3	4
0 (A)	0	10	11	12	22
1 (B)	0	5	6	30	50

How to get the answer to the original question?

The original question asked for the *schedule*, but we only computed the optimal *value* of the optimal schedule

Use backtracking:

	Minute 1	Minute 2	Minute 3	Minute 4
A	10	1	1	10
B	5	1	20	20

Schedule:
?, ?, ?, B,

30 = 20 + (10 or 7?)

j↓ i →	0	1	2	3	4
0 (A)	0	10	11	12	22
1 (B)	0	5	6	30	50

How to get the answer to the original question?

The original question asked for the *schedule*, but we only computed the optimal *value* of the optimal schedule

Use backtracking:

	Minute 1	Minute 2	Minute 3	Minute 4
A	10	1	1	10
B	5	1	20	20

Schedule:
?, -, B, B,

10 = base case

j↓ i →	0	1	2	3	4
0 (A)	0	10	11	12	22
1 (B)	0	5	6	30	50

How to get the answer to the original question?

The original question asked for the *schedule*, but we only computed the optimal *value* of the optimal schedule

Schedule:
A, -, B, B,

Use backtracking:

	Minute 1	Minute 2	Minute 3	Minute 4
A	10	1	1	10
B	5	1	20	20

j↓ i →	0	1	2	3	4
0 (A)	0	10	11	12	22
1 (B)	0	5	6	30	50

How to get the answer to the original question?

The original question asked for the *schedule*, but we only computed the optimal *value* of the optimal schedule

Alternatively: Saving pointers while computing OPT table

	Minute 1	Minute 2	Minute 3	Minute 4
A	10	1	1	10
B	5	1	20	20

Schedule:
A, -, B, B,

j↓ i →	0	1	2	3	4
0 (A)	0, -	10, -	11; from 1,0	12; from 2,0	22; from 3,0
1 (B)	0, -	5, -	6; from 1,1	30; from 1,0	50; from 3,1

Proof of Correctness/Runtime

Induction Hypothesis: OPT[i] = (

Max value for a schedule from times 1 to i where machine A is used at time i,

Max value for a schedule from times 1 to i where machine A is used at time i

)

Base Cases:

i = 0 → (0, 0) as no machines have been scheduled; i = 1 → (a_1, b_1) as no machines have been run before this

Induction Step:

For machine A to be run at time i, either machine A was run at time i-1 or there was a wait at time i-1 and machine B was run at time i-2. By the induction hypothesis we know OPT stores the optimal values for the preceding schedules in both of these cases. Same argument for B

Thus, Max(OPT[n][0], OPT[n][1]) gives max possible value of a schedule, and our backtracking returns a schedule with this value

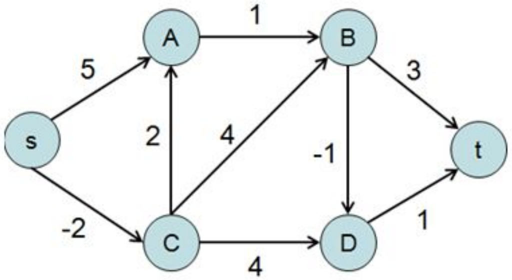
Runtime: O(n). There are O(n) iterations of the loop and each takes constant time, and O(n) constant time steps in backtracking

Bellman-Ford Algorithm

- Used to find shortest paths between any two points in a graph using DP
    - NO NEGATIVE COST CYCLES

- Subproblems: M[i,v] = cost of minimum cost path from v to t (symmetrically, could also do from s to v by reversing all edge directions), Uses an n x n table
- Final value: M[n-1, s]

- Base cases: M[0,t] = 0, M[0,v] = infinity for all other vertices

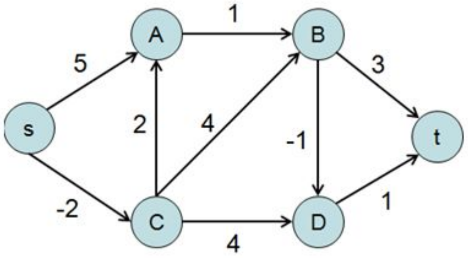


Bellman-Ford Algorithm

- Recurrence: You either get the minimum cost path using less than i edges or using i edges coming from either of its neighbors

  Opt(i,v) = min({Opt(i-1,v), min{c_(vw) + Opt(i-1,w)}}, w is any edge with an edge from v
- Backtracking: Start from vertex t, Do backtracking(n-1,t) for the answer

  For backtracking(i, v): If v = t, return set, If Opt(i-1, v) = Opt(i,v), do backtrack(i-1,v)
  Else, for all edges uv from u to v, see if Opt(i-1, u) =Opt(i,v) - c_(vw),
  then add the edge from u to v into the final solution set,
  then do backtrack(i-1,w)
- O(n^2) backtracking, O(n^3) => O(nm) DP



Divide and Conquer

General Approach

- Given two (or more) smaller pieces to the problem, can I combine them somehow to get a solution to the overall problem?

- Make sure to have/consider a base case
- Classic example: mergesort

- Proof: induction on the size of the input to the problem (or size of whatever you are breaking down)

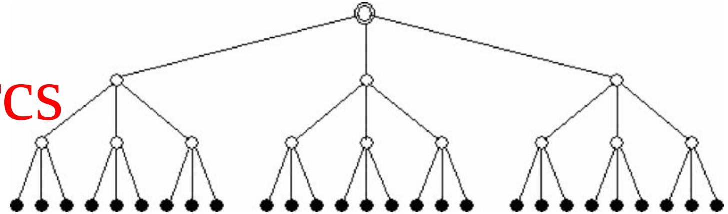
Runtime

Runtime

- Typically: $T(n) = qT(n/p) + O(n)$
  - i.e: Each call performs q calls that are a factor of p smaller and also performs linear time operations

Master Theorem

- $q < p \rightarrow O(n)$
- $q = p \rightarrow O(n \log n)$
  - Ex. Merge sort ($p = q = 2$)
- $q > p \rightarrow O(n\text{^}(\log_p q))$
  - Ex. Fast Integer Multiplication ($p = 2$, $q = 3$)



You do not need to memorize the Master Theorem!

If you are curious about the proof of the Master Theorem for more general forms, this link is very helpful.

Integer Multiplication

- Initial algorithm:
  - $A * B = (A_1 B_1) * 10^{2k} + (A_0 B_1 + A_1 B_0) * 10^k + A_0 B_0$, where A is divided into A_0, A_1, and B is divided into B_0, B_1 digits evenly (number of digits n = 2k is a power of 2)
    - Note that there are only four individual products we need to compute
  - Adding these numbers together is bounded by the number of digits, so it's O(n)
  - Total runtime is thus T(n) = 4T(n/2) + O(n)
  - Total runtime is O(n^2) by Master Theorem

Integer Multiplication

- Better algorithm:
  - Note that we can compute $A_0 B_1 + A_1 B_0$ in terms of $A_0 B_0$ and $A_1 B_1$
    - $A_0 B_1 + A_1 B_0 = (A_0 + A_1) * (B_0 + B_1) - A_1 B_1 - A_0 B_0$
    - Reduces the number of extra multiplications by one!
  - Need to compute $A_0 + A_1$ and $B_0 + B_1$, do the subtraction, and add the numbers in the end, so still O(n)
  - Total runtime is thus T(n) = 3T(n/2) + O(n)
  - Total runtime is $O(n^{\log_2 3})$ by Master Theorem

Counting Inversions

- An inversion in a sequence of numbers is when two indices i and j fulfill the property that $i < j$ but element x at position i is greater than element y at position j
- For a sequence $a_1, a_2, \ldots, a_n$, find a divide and conquer solution to count the number of inversions in a sequence in O(nlogn) time
- Hint: can sorting help in any way?

Solution

- Set m = ceil(n/2) and divide the sequence

- Suppose we have the two halves A and B sorted and had the number of inversions for each
- Just merge A and B in sorted order while simultaneously counting the inversions as you sort ($a > b$ where $a \in A$ and $b \in B$)

    - When an element from A is added, then there is no inversion
    - When an element from B is added, then there is an inversion for each element in A left
- So, sort recursively sort the left and right, and then merge
    - Add together the inversions from left sort, right sort, and merge

Solution, cont.

- Recurrence: $T(n) = 2T(n/2) + O(n)$
  - Dividing into two parts and two recursive calls
  - Merging step is $O(n)$
  - Overall runtime is $O(n\log n)$

Randomized algorithms

Key Idea

- Use randomization within algorithm to solve problems more efficiently than deterministic ones

- Typically evaluated through "Average Case" analysis (rather than "Worst Case" analysis) and probabilities of certain behavior (ex. correct/incorrect)

- Reasoning about probabilities also required to analyze performance

Probability Concepts to Know

(Probability recitation slides go into a lot more detail)

- If A and B are independent, P(A and B) = P(A) * P(B)
- The expected value of a random variable X can be thought of as a weighted mean of the different values X can take.

- If X can be written as the sum of random variables $Y_1 + Y_2 + \ldots$, $E(X) = E(Y_1) + E(Y_2) + \ldots$
  - Does not require $Y_1, Y_2, \ldots$ to be independent!

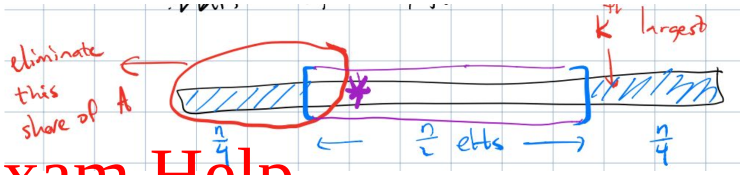
- If you can model a random variable as a distribution covered in class, you can use the E(X) formula covered in class
  - $X \sim Geom(p)$: $E(X) = 1/p$
    - Number of coin flips needed to get a heads ($p = ½$)
  - $X \sim Binom(n, p)$: $E(X) = np$
    - Number of heads in 5 coin flips ($n = 5$, $p = ½$)

Median Finding

- Goal: Find the k'th largest element in a list L of n numbers

- Idea 1:
    - Choose a pivot x in L (ex. First element)
    - Partition L into elements greater than x (L^+)and less than x (L^-) - Takes (O(n) time)
    - Recurse on whichever of L^+ and L^- contains the median (determined by looking at sizes of L^+, L^-, and previous partitions)

    - Worst case $O(n^2)$ time: could choose min element as splitter each time, and next set size decreases by 1



Median Finding

- Goal: Find the k'th largest element in a list L of n numbers
- Idea 2 (Randomization): Randomly choose pivot, only recurse once we've found a "good" pivot
  - Randomly choose an pivot x in L
  - Partition L into elements greater than x (L^+)and less than x (L^-)
  - If the chosen pivot is not contained in the middle 50% of the list, go back to step 1
  - Recurse on whichever of L^+ and L^- contains the median (determined by looking at sizes of L^+, L^-, and previous partitions)
  - Theoretically could run forever
  - In expected case
    - Probability of pivot falling is ½, so the expected number of pivots needed per recursive step is 2
    - At most ¾ of the list is passed to the recursive call
    - Recurrence is $T(n) \le T(3n/4) + 2n$, so runtime is $O(n)$ by Master Theorem

Practice Problem (Chapter 13, Problem 8, simplified)

For a graph G = (V, E), let an induced subgraph of G be one which contains some subset of vertices X and all edges with both endpoints in X

Describe a randomized algorithm which when given a graph G and number k ≤ |V|

- Finds some subset of k vertices that form an induced subgraph with high expected density
  - i.e. Expected number of edges in the induced subgraph is ≥ |E| • (k choose 2)/(|V| choose 2) edges)
- Runs in O(|V|) time

Practice Problem, Solution

- Solution: The algorithm just chooses k vertices at random and returns them
- Why does this work?
  - Probability that a single randomly sampled pair of vertices are connected by an edge in G?

Practice Problem, Solution

- Solution: The algorithm just chooses k vertices at random and returns them
- Why does this work?
  - Probability that a single randomly sampled pair of vertices are connected by an edge in G
    - There are m edges in the graph, and n choose 2 pairs of vertices → |E|/(|V| choose 2)
  - Expected number of edges in induced subgraph of k randomly chosen vertices?

Practice Problem, Solution

- Solution: The algorithm just chooses k vertices at random and returns them
- Why does this work?
  - Probability that a single randomly sampled pair of vertices are connected by an edge in G
    - There are m edges in the graph, and n choose 2 pairs of vertices → |E|/(|V| choose 2)
  - Expected number of edges in induced subgraph of k randomly chosen vertices:
    - For any pair of vertices n_i and n_j among the k chosen vertices, let X_{ij} be an indicator random variable which is 1 if there's an edge between n_i and n_j and 0 otherwise
    - $E[X_{ij}]$ = Probability of an edge between a randomly chosen pair = |E|/(|V| choose 2)
    - Total Number of Edges = ΣX_{ij}
    - E[Total Number of Edges] = $E[\Sigma X_{ij}] = \Sigma E[X_{ij}]$ = (k choose 2) • $E[X_{ij}]$ = |E| • (k choose 2)/(|V| choose 2)

Questions?