UPE Tutoring:

CS 180 Midterm Review

Sign-in: https://bit.ly/2XcJNKv

Slides link available upon sign-in

Quick Announcement - CS Town Hall

Date & Time: Wednesday, February 20th, 2019 at 6 PM
Location: Engineering VI, Mong Auditorium

The CS Town Hall is an opportunity for all CS/CSE/CE students to interact with the UCLA Computer Science Department and provide feedback, suggestions or discuss problems. The CS Vice Chair for Undergraduate Studies, Professor Rich Korf, will be there along with a few other CS professors to answer questions and receive suggestions.

Please fill out your comments here so that we can organize the comments into topics: https://goo.gl/forms/uZPbFUH0RUaUuhtA2

Food will be provided! RSVP Link: https://www.facebook.com/events/2321591601406615

Tips:

- I took Gafni's class and the curve is big. The average is ~40% (the lowest being ~10/100, highest being ~80/100), so write anything down. Your intuition will give you at least a few points.
- Read all problems before starting - some are definitely harder than others and the last question might not be the hardest.
- Gafni likes to include terms he mentioned in class, but is not necessarily in the book or in your homework. If you ask him what "partial ordering" is, or "strongly connected components" in the test, he won't tell you.

Topics

- Celebrity Problem
- Time Complexity
- Topological Sort
- Greedy Paradigm
- Interval Scheduling
- Dijkstra's Algorithm
- Counting Inversions & Majority Element
- Dynamic Programming

Celebrity Problem - Background

Consider a group of N people gathered together in this room. We define a famous person as someone that doesn't "know" any of the other N – 1 people, but all other N – 1 people know that person.

We can tell if a person knows another by this method: we pick two people, A and B, and ask Person A if they know Person B. Person A will reply with either a Yes or a No.

Celebrity Problem - Problem Statement

Given the definition of a famous person, the asking method, and a room full of N people, determine which people in the room are famous people.

Celebrity Problem - Naive Approach

- Go to each individual person and confirm whether they are a famous person or not.
  - Requires N - 1 asks to confirm that the individual doesn't know any others
  - Requires N - 1 asks to confirm that all other individuals know this individual

- If we perform this test on everyone in the room, we will perform a total of:
  - $2(N - 1) * N = 2N^2 - 2N$ asks ➔ Time Complexity: $O(N^2)$

Can we do better than this?

Celebrity Problem - Key Observations

- Key Observation 1:

- Key Observation 2:
  - There can at most be one famous person in a group. If one person is famous then everyone else knows that person, but famous people don't know anyone, so all others are disqualified.

Celebrity Problem - New and Improved Solution

- Each time we use the asking method, we eliminate either Person A or Person B. Thus, we can begin the algorithm by picking two arbitrary people in the group, performing an ask, and pitting whoever remains against an arbitrary remaining person, until after N - 1 asks we only have 1 person remaining.
- We can then manually check to see if this person is famous the same way we checked for everyone in the naive approach.

- This gives us a time complexity of:
  - Elimination asks + Final Candidate Check ➔ (N - 1) + 2(N - 1) = 3(N - 1) ➔ O(N)

- Note that even without going in depth into a proof this already matches our intuitions; the algorithm will identify at most 1 famous person.

Time Complexity - Big O and Asymptotic Analysis

Worst case time complexity analysis - in this class usage is very similar to considering time complexity in CS 32.

Mathematically:

- Let T(n) be a function (e.g. worst case runtime of a certain algorithm on input size n)
- Given another function f(n), we say that T(n) is O(f(n)) if, for sufficiently large n, the function T(n) is bounded above by a constant multiple of f(n)
- T(n) = O(f(n)) if there exist constants c > 0 and n_0 >= 0 such that for all n >= n_0:
  - T(n) <= c * f(n)
- Note that c cannot depend on n.
- We say that T is asymptotically upper bounded by f.
- Note that this means any O(n) function is also technically O(nlogn), and so on.

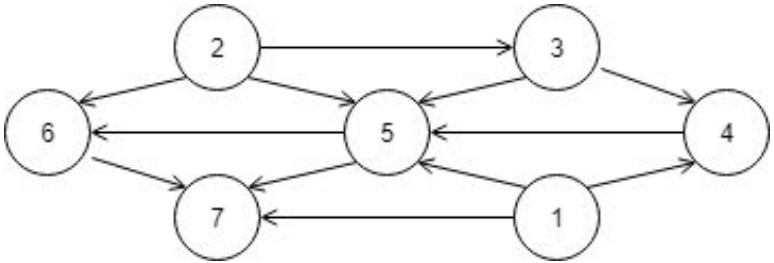
Time Complexity - Common Running Times

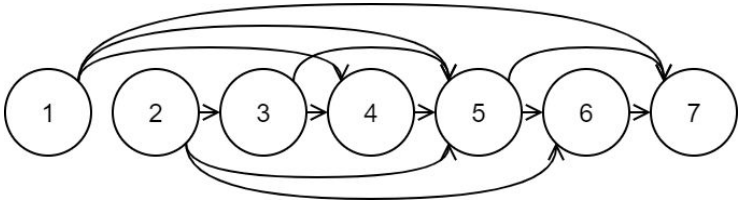
- O(1) - Constant
- O(n) - Linear
- O(nlogn) - Divide and Conquer
- O(logn) - Binary Search
- $O(n^2)$, $O(n^3)$, $O(n^k)$
- Non-polynomial

Topological Sort - Directed Acyclic Graphs

A **directed acyclic graph (left)** is a directed graph that contains no cycles.

A **topological ordering (right)** of a graph G is an ordering of its nodes as v_1, v_2, ..., v_n such that for every edge (v_i, v_j), we have i < j. All edges point “forward” in the ordering. If each node represents a task that must be completed and each edge a dependency, then a topological ordering is an order in which all tasks can be completed safely.





Topological Sort - Proof that Topo Order is DAG

Observation 1:

If a graph G has a topological ordering, then G is a DAG.

Proof by contradiction: Assume that G has a topological ordering v_1, ..., v_n which also has a cycle C. Let v_i be the lowest indexed node on C, and let v_j be the node on C just before v_i, such that (v_j, v_i) is an edge.

By our choice of i, we know j > i, which contradicts our assumption that v_1, ..., v_n is a topological ordering, since j < i would need to be true for (v_j, v_i) to exist as an edge.

Topological Sort - Problem Statement

Does every DAG have a topological ordering? If so, how do we find one efficiently?

Topological Sort - Getting Started

The first node in a topo ordering needs to have no incoming edges.

Observation 2:
In every DAG G, there is a node v with no incoming edges.

Proof by contradiction: Let G be a DAG where every node has at least one incoming edge. Pick any node v, and begin following edges backward from v: since v has at least one incoming edge we can always follow an edge backwards to some node u, and so on.

We can do this indefinitely, since every node has an incoming edge. After doing this n + 1 times, by the Pigeonhole Principle we have visited some node w twice. We can then let C denote the nodes visited between visits of w, which is a cycle, a contradiction.

Topological Sort - Proof of Converse

Observation 3:

If a graph G is a DAG, then it has a topological ordering.

Proof by induction: Claim by induction that every DAG has a topological ordering. This is true for base cases of DAGS with one or two nodes.

Now consider that it is true for DAGs with n nodes. Given a DAG G with n + 1 nodes, we can use Observation 2 to find a node v with no incoming edges, and place it first in our topological ordering since all of its edges point forward. G - {v} is a DAG, since deleting v can't create any cycles. G - {v} has n nodes, so we can apply induction to get its topological ordering. We append that to v. This is an ordering of G where all nodes point forward, so it is indeed a topological ordering. Whew!

Topological Sort - The Algorithm

Below is the algorithm to compute a topological ordering of graph G.

```
To compute a topological ordering of G:
Find a node v with no incoming edges and order it first
Delete v from G
Recursively compute a topological ordering of G−{v}
  and append this order after v
```

It takes O(n) time to find a node v with no incoming edges and deleting it.
The algorithm runs for n iterations, to the time complexity is $O(n^2)$, where n is the number of nodes.

Can we do better?

Topological Sort - Yep

We can do better by tracking two things:

- For each node w, the number of incoming edges w has from active nodes
- The set S of all active nodes in G with no incoming edges from other active nodes

All nodes are active at the start, so initialize our data with one pass through nodes and edges.

Each iteration, we delete a node v from S. Then we go through all nodes w to which v had an edge and decrement 1 from their number of active incoming edges. If, for a node w, it drops to 0, we add w to S.

This way, we track nodes eligible for deletion at all times, with constant work per edge.
Time Complexity: O(m + n), where m is # of edges and n is # of nodes.

Interval Scheduling

- Suppose we have a set of activities $S = \{a_1, ..., a_n\}$. These are activities with start and end times s_i and f_i.
- We want to select the maximum cardinality subset of S, called $S' = \{a_1', a_2' ...\}$ where $[s_i, f_i)$ and $[s_j, f_j)$ do not overlap, and a_i' and a_j' are in S'. This means, $f_i < s_j$. (WLOG)
  - This can be phrased as: max |S'|.

- The solution, as you know, is to sort by end time, and greedily pick the earliest end time intervals that do not overlap.
  - But do you know why?

Interval Scheduling - Optimal Substructure

An **optimal substructure** is a fancy of way of saying “the optimal answer of this larger answer can be constructed from optimal answer of smaller subsets of this answer”.

In this case, denote S_i as the set of activities after ai ends.

We can ask the same question : “What’s the max cardinality of S_i’?”. The answer of this S_i’ can be used to construct our solution for S! *This is an optimal substructure.*

Now, consider the set of intervals that overlap at the beginning, such that no interval can be added before it. If we select the interval of earliest finish time (called f_i), rather than another interval(called f_j), then we can get S_i to be from [f_i, end), and S_j to be from [f_j, end).

Interval Scheduling - Monotonicity

We also need **monotonicity** in our argument to ensure that choosing S_i will be better than S_j.

Monotonicity of a function means if x < y, then f(x) <= f(y) *always*(monotonically increasing), or f(x) >= f(y) *always*(monotonically decreasing). Generalizing this to sets as inputs, mapped to real number outputs, we have $x \subseteq y$, then f(x) <= f(y) for monotonically increasing.

Our function max_nonoverlap() is monotonic. Why? Because the larger set S_i can always include the same intervals as S_j's answer!

Greedy Problems Guideline

Now that you understand Interval Scheduling, what about other greedy problems? There's a boilerplate template that you can follow!

- Determine whether optimal substructure exists *(We can solve for a smaller subset of S).*
- Develop a recursive solution *(Take one of the front intervals, solve for the S_j).*
- Show that if we make a greedy choice, only 1 subproblem remains*(S_i is our subproblem)*, and that it's always safe to make the greedy choice *(monotonicity is your friend).*
- Use the greedy solution, and make it iterative for brownie points.

Dijkstra's Algorithm

Here's another greedy problem. Here's the setup:

Given a graph G = (V, E), a function f : e ➔ R+, where e is in E, and 2 nodes s and t, both in V, we want to find the shortest path between s and t.

A path P = {e_1, e_2, ..., e_n}, where e_1 = (s, v_1), e2 = (v_1, v_2), and e3 = (v_2,v_3)... and e_n = (v_n,t). P⊆E.

In other words, the path connects s to t. The length of the path is the sum of f(e_1) + f(e_2) ... + f(e_n).

We want to find the minimum length of the path, or in other words: $\min_{s,t}$ sum(P).

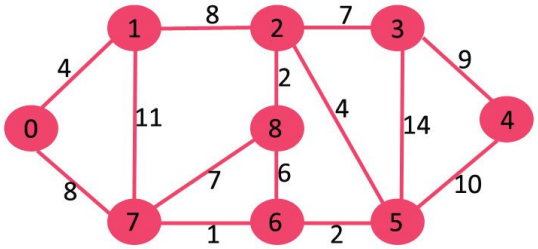
Dijkstra's Algorithm - Algorithm

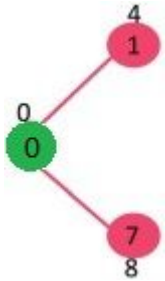
Dijkstra's algorithm is the following:

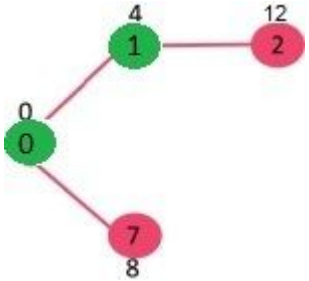
1. Initialization:
   a. distance from s to s = 0, and s to any other vertex = ∞.
   b. We have a set S, which includes nodes we are currently traversing in. S = {(s,0)} initially.
2. In every iteration, we take the node v in (V-S) with lowest distance d_v, add it to S, and relax each of its unvisited neighbors, n, with the shortest path to s, as
   $$(n, \min(d_n, d_v + f(\{v, n\}))).$$
3. When we reach the destination, node t, we terminate and return the value it has inside of S.

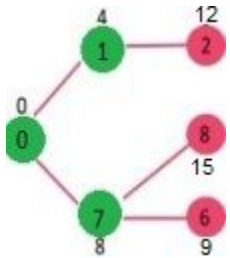
Dijkstra's Algorithm - Example

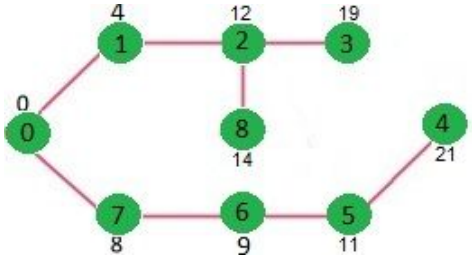
We start with the graph:

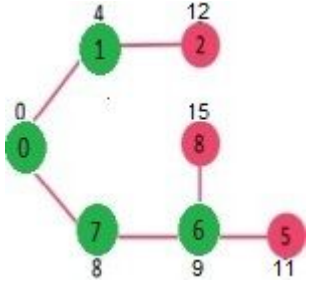












...

Dijkstra’s Algorithm - Optimal Substructure & Monotonicity

What is the optimal substructure?

Dijkstra's Algorithm - Optimal Substructure & Monotonicity

What is the optimal substructure?

The answer is: The shortest path from s to any node v in the optimal path from s to t!
If P* = s ➔ v ➔ t in the optimal answer, then s ➔ v must also be the shortest path from s to v.

Where is the monotonicity?

Dijkstra's Algorithm - Optimal Substructure & Monotonicity

What is the optimal substructure?

The answer is: The shortest path from s to any node v in the path from s to t!
If P* = s ➔ v ➔ t in the optimal answer, then s ➔ v must also be the shortest path from s to v.

Where is the monotonicity?

The monotonicity lies in our assumption of positive length edges. We will never find a path shorter than the optimal by traversing farther in any suboptimal paths, because they will just get longer and longer.

Dijkstra's Algorithm - Formal Proof

We want to prove this inductively with the **invariant properties:**

- **At every inductive step, any element in our finalized explored set S has the correct distance.**

Base case: Starting node s is in our set S. Distance to itself is 0.

Inductive: Suppose we take in a node u into our set S, and suppose to contradict that d(s,u) != d_u. Then this is not the shortest path. Consider the **real shortest path** from s ➔ u then. On this path s ➔ u, there's a "crossing" from S to V\S. The first crossing between a node x in S to a node y in V\S gives us the path: **s ➔$_p$ x ➔ y ➔$_p$ u**. By inductive hypothesis we already know d_x = d(s,x). Look at d_y - it is d_y = d_x + f(x,y) = d(s,y). Why is it not less than? Because if it was, then this path from s➔$_p$x➔y➔$_p$u cannot be the shortest path! Then, we argue that if y != u, by positive weighted edges, d(s,y) <= d(s,u), then our algorithm would've chosen y as the next node. Contradiction.

Counting Inversions - Background

Say you and a friend ranked the same set of n movies. We want to see how similar your tastes are.

We do this by labelling the movies by your ranking, then sorting by your friend's ranking, and seeing how many pairs are "out of order".

Counting Inversions - Problem Statement

Given a sequence of n distinct numbers $\{a_1, \ldots, a_n\}$, find number of **inversions** in the sequence.

An inversion is a case where two indices i and j satisfy $a_i > a_j$ in the sequence.

Ex: {2, 4, 1, 3, 5} has the inversions (4, 1), (4, 3), and (2, 1)

Counting Inversions - Approach

General Approach

- Set m = ceiling(n / 2) and divide the sequence into
  - $\{a_1, ..., a_m\}$ and $\{a_{m+1}, ..., a_n\}$
- Count the inversions in each half, then sort (Divide)
- Count the inversions between elements in each half (Conquer)

Specifics

- Merge-and-Count:
  - Given sorted lists A and B, merge them into a sorted list C and count inversions
  - Regular merge: have pointers into A and B, a_i and b_j, and add the smaller element to C and advance its pointer
  - **Key Property:** Every time we add b_j, it is smaller than all remaining items in A, so all of those are inversions

Counting Inversions - Formal Algorithm

```
Merge-and-Count(A,B)
  Maintain a Current pointer into each list, initialized to
    point to the front elements
  Maintain a variable Count for the number of inversions,
    initialized to 0
  While both lists are nonempty:
    Let a_i and b_j be the elements pointed to by the Current pointer
    Append the smaller of these two to the output list
    If b_j is the smaller element then
      Increment Count by the number of elements remaining in A
    Endif
    Advance the Current pointer in the list from which the
      smaller element was selected.
  EndWhile
```

Counting Inversions - Explanation

.....

Counting Inversions - Explanation

Same as Merge Sort!

The algorithm is the same; we just do an extra calculation while we merge.

Time Complexity: O(NlogN)

Divide and Conquer - Majority (On previous midterm)

Given a sequence of keys x_1,...,x_n, and the only query we can ask is whether x_i = x_j and get the answer yes or no. We would like to find whether there exists a key that constitutes a majority(occurs more than > N/2 times).

a) What will the time complexity of a brute force algorithm be?
b) Design an algorithm with O(n log n) time complexity using divide and conquer.

Majority

a) Is pretty obvious. What's the answer?
b) Hint: start with one element or two elements in your sequence. What is the majority? How can you generalize this?

Majority b)

Divide: For any sequence of length n, divide it into x_1,...,x_{n/2}, and x_{n/2+1},...,x_n. Perform this iteratively until we arrive at every sequence is of length 1.

Conquer: We have length one sequences, when n = 1, pick the single element and say the majority element is of frequency 1. In programming terms, return (x_1, 1).
By the inductive hypothesis we have retrieved the majority of the left side and the majority of the right side, with (x_left, n_left), (x_right, n_right). If x_left = x_right, then it's obvious that the majority is that value, since n_left > n/4, and n_right > n/4, so we return (x_left, n_left + n_right). If x_left != x_right, then we have two elements that could be majority elements. Perform the O(N) algorithm to find the plurality (could be < N/2) in here, and return (x_plurality, n_plurality). At the end, check if n_plurality > n/2 for the original sequence size.

The time recurrence relation is T(N) = T(N/2) * 2 + O(N), so it's O(nlogn).

Dynamic Programming

Dynamic programming is a method for solving complex problems by breaking them down into simpler subproblems

Three important steps:

1. Define the subproblems
2. Write down the recurrence relation
3. Recognize and solve base cases

1D Dynamic Programming - Example

For a given n, find the number of ways to write n as a sum of the numbers 1, 3, and 4 (order matters).

Example:
For n = 5, the answer is 6:

$$5 = 1 + 1 + 1 + 1 + 1$$
$$5 = 1 + 1 + 3$$
$$5 = 1 + 3 + 1$$
$$5 = 3 + 1 + 1$$
$$5 = 1 + 4$$
$$5 = 4 + 1$$

1D Dynamic Programming - Example

Let's go through the three steps...

1. Define the subproblems
   Let dp[i] be the number of ways to write i as a sum of 1, 3, and 4

2. Find the recurrence relation
   dp[i] = ?
   Consider one possible solution: $i = x_1 + x_2 + \ldots + x_n$
   If $x_n = 1$, how many ways are there to sum up to i?
   This is equivalent to dp[i - 1]
   Now consider the other two cases (if $x_n = 3$ or $x_n = 4$)

1D Dynamic Programming - Example

2. Find the recurrence relation

   dp[i] = dp[i - 1] + dp[i - 3] + dp[i - 4]

3. Recognize and solve the base cases

   Our recurrence relation depends on the past 4 elements.

   dp[0] = dp[1] = dp[2] = 1, dp[3] = 2

```
dp[0] = dp[1] = dp[2] = 1; dp[3] = 2;
for(i = 4; i <= n; i++)
    dp[i] = dp[i - 1] + dp[i - 3] + dp[i - 4];
```

2D Dynamic Programming - Knapsack Problem

We are given a set of n items, where each item i is specified with a size s_i and value v_i . We are also given the capacity C of our backpack.

Problem statement for the 0-1 Knapsack Problem:

Find the maximum value of items we can store in our backpack such that each item can only be put in our backpack 0 or 1 times, and the sum of all of the sizes of the items in our backpack cannot exceed the capacity C.

2D Dynamic Programming - Knapsack Problem

1. Define the subproblems:
   Let dp[i][j] be the maximum value we can store in our backpack if we only consider the first i items and have a maximum capacity of j.

2. Find the recurrence relation:
   We either take the current item or we don't.
   dp[i][j] = max(dp[i - 1][j], dp[i - 1][j - s[i]] + v[i])

3. Recognize and solve the base cases:
   dp[i][0] = 0 for all 0 <= i <= n
   dp[0][j] = 0 for all 0 <= j <= C

2D Dynamic Programming - Knapsack Problem

Example where n = 4 and C = 5:
Items (s_i, v_i) are: {(2, 3), (3, 4), (4, 5), (5, 6)}

i \ j	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0	0	3	4	4	7
3	0	0	3	4	5	7
4	0	0	3	4	5	7

Dynamic Programming - Practice Problem

Given unlimited coins of the values 1, 3, and 4, find the minimum number of coins needed to make change for a value v.

Example:

For 6, the minimum number of coins needed is 2 (3 + 3).

Note that the greedy approach, while temping, does not work here!
6 = 4 + 1 + 1 is not the best solution

Dynamic Programming - Practice Problem Solution

Given unlimited coins of the values 1, 3, and 4, find the minimum number of coins needed to make change for a value v.

1. Define subproblems
   Let dp[i] be the minimum number of coins needed to make change for value i
2. Find the recurrence relation
   dp[i] = min(dp[i - 1], dp[i - 3], dp[i - 4]) + 1
3. Recognize and solve the base cases
   dp[0] = 0; dp[1] = 1; dp[2] = 2; dp[3] = 1

Loop from 4 to v. dp[v] is your final answer.

Good luck!

Sign-in https://bit.ly/2XcJNKv
Slides https://bit.ly/2SP9Glq

Questions? Need more help?

- Come up and ask us! We'll try our best.
- UPE offers daily computer science tutoring:
  - Location: ACM/UPE Clubhouse (Boelter 2763)
  - Schedule: https://upe.seas.ucla.edu/tutoring/
- You can also post on the Facebook event page.