CS 374A Review MT2

Electric Boogaloo

ACM @ UIUC

November 8, 2025

Disclaimers and Logistics

- **Disclaimer:** Some of us are CAs, but we have not seen the exam. We have no idea what the questions are. However, we've taken the course and reviewed many previous exams, so we have **suspicions** as to what the questions will be like.
- This review session is being recorded. Recordings and slides will be distributed on EdStem after the end.
- **Agenda:** We'll quickly review all topics likely to be covered, then go through a practice exam, then review individual topics by request.
- Please let us know if we're going too fast/slow, not speaking loud enough/speaking too loud, etc.
- If you have a question anytime during the review session, please ask! Someone else almost surely has a similar question.
- We'll provide a feedback form at the end of the session.

Recursion

- **Definition:** Reducing the problem to a smaller instance of itself, where eventually we can terminate in a base case.
  - Think: If we have a problem of size n, we want to continuously reduce to a problem smaller than n.
  - Example: Binary search, Karatsuba

Template

```
1: procedure AmazingRecursiveAlgo(n)
2:     if n == [some base case] then
3:         return [value]
4:     else
5:         return AmazingRecursiveAlgo(n/2)
```

- Similar to **induction**!

Recursion: Runtime Analysis

- **General Form:**

$$T(n) = \underbrace{r}_{\text{\# of subproblems}} \cdot \overbrace{T\left(\frac{n}{c}\right)}^{\text{work at each subproblem}} + \underbrace{f(n)}_{\text{work at current level}}$$

  - Describes how the amount of work changes between each level of recursion.
  - We can solve for a **time complexity** that describes the scaling behavior of the algorithm at hand.

Recursion: Runtime Analysis

- **General Form:**

$$T(n) = \underbrace{r}_{\text{\# of subproblems}} \cdot \overbrace{T\left(\frac{n}{c}\right)}^{\text{work at each subproblem}} + \underbrace{f(n)}_{\text{work at current level}}$$

  - Describes how the amount of work changes between each level of recursion.
  - We can solve for a **time complexity** that describes the scaling behavior of the algorithm at hand.
- **Master Theorem**

Master Theorem

$$\begin{aligned}
\text{Decreasing: } r \cdot f(n/c) = \kappa \cdot f(n) \text{ where } \kappa < 1 &\implies T(n) = O(f(n)) \\
\text{Equal: } r \cdot f(n/c) = \kappa \cdot f(n) \text{ where } \kappa = 1 &\implies T(n) = O(f(n) \cdot log_c n) \\
\text{Increasing: } r \cdot f(n/c) = \kappa \cdot f(n) \text{ where } \kappa > 1 &\implies T(n) = O(n^{\log_c r})
\end{aligned}$$

Recursion: Runtime Analysis

- **General Form:**

$$T(n) = \underbrace{r}_{\text{\# of subproblems}} \cdot \overbrace{T\left(\frac{n}{c}\right)}^{\text{work at each subproblem}} + \underbrace{f(n)}_{\text{work at current level}}$$

  - Describes how the amount of work changes between each level of recursion.
  - We can solve for a **time complexity** that describes the scaling behavior of the algorithm at hand.
- **Master Theorem**

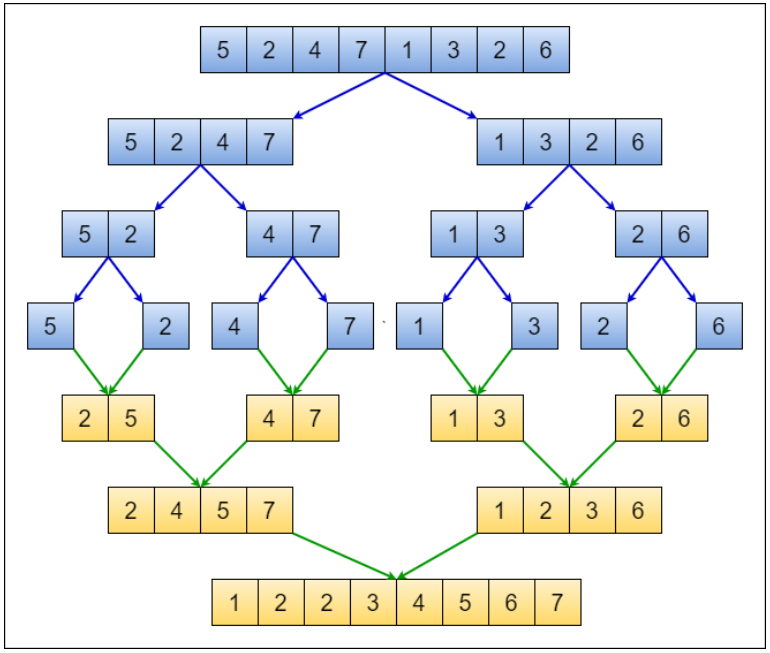
Master Theorem

$$\begin{aligned}
\text{Decreasing: } r \cdot f(n/c) = \kappa \cdot f(n) \text{ where } \kappa < 1 &\implies T(n) = O(f(n)) \\
\text{Equal: } r \cdot f(n/c) = \kappa \cdot f(n) \text{ where } \kappa = 1 &\implies T(n) = O(f(n) \cdot log_c n) \\
\text{Increasing: } r \cdot f(n/c) = \kappa \cdot f(n) \text{ where } \kappa > 1 &\implies T(n) = O(n^{\log_c r})
\end{aligned}$$

  - **Intuition:** If each level contains more work than the level below it, then the root level will dominate. If each level contains the same amount of work, then we have $log_c n$ levels with $f(n)$ work. If each level contains less work than the work below it, then the leaf nodes will dominate.

Divide and Conquer Algos: Merge Sort

- **Purpose:** Sort an arbitrary array.
- **Time Complexity**: $O(n \log n)$
- **Intuition:** Three phases: (a) split the array in half, (b) sort each side, (c) merge the sorted halves by repeatedly comparing smallest elements on each side not yet inserted.

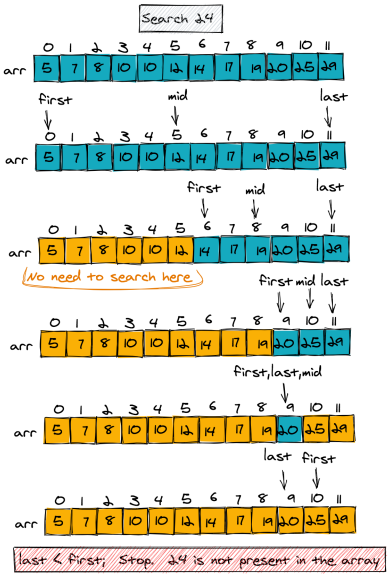


Divide and Conquer Algos: MomSelect

- **Purpose:** Get the r^{th} smallest element in an arbitrary array.
- **Time Complexity**: $O(n)$
- **Intuition**: Pick a pivot P with a value P_V and rearrange the array such that all the elements that are less than P_V are to the left of P and all the elements that are greater than P_V are to the right of P. If the length of the array of elements that are less than P_V is greater than r, then we know that the r^{th} smallest element is to the left of P and we recurse on the left subarray. Otherwise, we know that the r^{th} smallest element is to the right of P and we recurse on the right subarray.

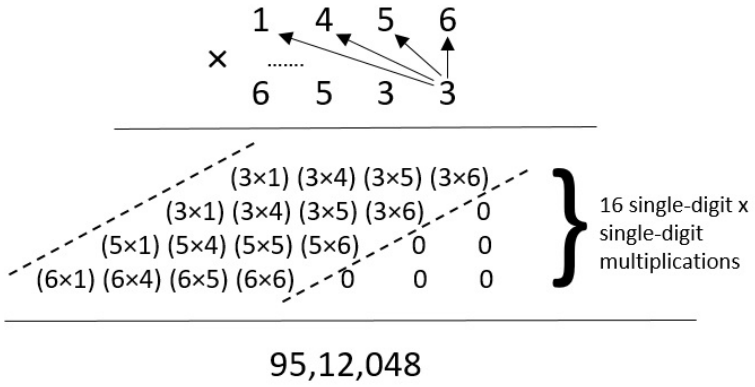
Divide and Conquer Algos: Binary Search

- **Purpose:** Find the existence of an element in a sorted array
- **Time Complexity**: $O(\log n)$
- **Intuition**: Say we are trying to find the value k. Pick the middle element M in the array. If $k > M$, the element must be to the right of M and we recurse on the right. Otherwise, we recurse on the left.



Divide and Conquer Algos: Karatsuba's

- **Purpose:** Multiplication
- **Time Complexity**: $O(n^{\log_2 3}) \approx O(n^{1.585})$
- 3 Phases:
  - **Divide**: Represent x as $x_0 p^n + x_1$, y as $y_0 p^n + y_1$
  - **Recurse**: Calculate $x_0 y_0$, $x_1 y_1$, $(x_0 + x_1)(y_0 + y_1)$
  - **Combine**: Use the three results to calculate our final answer.



Dynamic Programming

- It’s backtracking, but we compute all of the subproblems iteratively.
  - This idea of "writing things down" as to not repeat computation is called **memoization**

Dynamic Programming

- It's backtracking, but we compute all of the subproblems iteratively.
  - This idea of "writing things down" as to not repeat computation is called **memoization**
- Alternatively, you can think about this recursively, except we check our memoization structure to see if we've computed anything before. If we have, we just use the computed result. Otherwise, we compute the subproblem.

Dynamic Programming

- It's backtracking, but we compute all of the subproblems iteratively.
  - This idea of "writing things down" as to not repeat computation is called **memoization**
- Alternatively, you can think about this recursively, except we check our memoization structure to see if we've computed anything before. If we have, we just use the computed result. Otherwise, we compute the subproblem.
- For a DP solution, we need:
  1. English Description
  2. Recurrence
  3. Memoization Structure
  4. Solution Location
  5. Evaluation Order
  6. Runtime

Dynamic Programming

- It's backtracking, but we compute all of the subproblems iteratively.
  - This idea of "writing things down" as to not repeat computation is called **memoization**
- Alternatively, you can think about this recursively, except we check our memoization structure to see if we've computed anything before. If we have, we just use the computed result. Otherwise, we compute the subproblem.
- For a DP solution, we need:
  1. English Description
  2. Recurrence
  3. Memoization Structure
  4. Solution Location
  5. Evaluation Order
  6. Runtime
- **How to solve a DP:**
  - Identify how we can take advantage of a recursive call on a smaller subset of the input space.
  - Identity base cases
  - Identity recurrences (they should cover all possible cases at each step)

Dynamic Programming: Longest Increasing Subsequence

1. **English Description:** $\mathrm{LIS}(i, j) =$ the length of the longest increasing subsequence in $A[i..n]$, such that every element in the subsequence is $> A[j]$. Here, $j = 0$ can be used as a sentinel with $A[0] = -\infty$.
2. **Recurrence:**

$$\mathrm{LIS}(i, j) = \begin{cases} 0 & \text{if } i > n, \\ \mathrm{LIS}(i+1, j) & \text{if } A[i] \leq A[j], \\ \max\left(\mathrm{LIS}(i+1, j),\ 1 + \mathrm{LIS}(i+1, i)\right) & \text{if } A[i] > A[j]. \end{cases}$$

3. **Memoization Structure:** A 2D array $LIS[0..n, 0..n]$, storing all subproblem solutions $\mathrm{LIS}(i, j)$.
4. **Solution Location:** The length of the LIS of the entire array is found at $\mathrm{LIS}(1, 0)$, starting from the first element with the sentinel.
5. **Evaluation Order:** Fill the table in decreasing order of i.
6. **Runtime:** The table has $O(n^2)$ entries, each computed in $O(1)$, so the total runtime is $O(n^2)$.

Graphs

- **Definition:** A set of vertices V connected by a set of edges E. Individual edges are notated as (u, v), where $u, v \in V$.
  - They are usually represented as **adjacency lists** or **adjacency matrices**

Graphs

- **Definition:** A set of vertices V connected by a set of edges E. Individual edges are notated as (u, v), where $u, v \in V$.
  - They are usually represented as **adjacency lists** or **adjacency matrices**
  - **Directed:** Each edge $(u, v) \in E$ now has a direction $u \rightarrow v$

Graphs

- **Definition:** A set of vertices V connected by a set of edges E. Individual edges are notated as (u, v), where $u, v \in V$.
  - They are usually represented as **adjacency lists** or **adjacency matrices**
  - **Directed:** Each edge $(u, v) \in E$ now has a direction $u \to v$
  - **Acyclic:** No cycles.

Graphs

- **Definition:** A set of vertices V connected by a set of edges E. Individual edges are notated as (u, v), where $u, v \in V$.
  - They are usually represented as **adjacency lists** or **adjacency matrices**
  - **Directed:** Each edge $(u, v) \in E$ now has a direction $u \to v$
  - **Acyclic:** No cycles.
- **Path:** A sequence of distinct vertices where each pair of consecutive vertices have an edge

Graphs

- **Definition:** A set of vertices V connected by a set of edges E. Individual edges are notated as (u, v), where $u, v \in V$.
  - They are usually represented as **adjacency lists** or **adjacency matrices**
  - **Directed:** Each edge $(u, v) \in E$ now has a direction $u \to v$
  - **Acyclic:** No cycles.
- **Path:** A sequence of distinct vertices where each pair of consecutive vertices have an edge
- **Cycle:** A sequence of distinct vertices where each pair of consecutive vertices have an edge **and** the first and last vertices are connected.

Graph Algorithms: Traversal

Graph Algorithms: Traversal

- **BFS:**
  - **Purpose:** Reachability, Shortest Path (unweighted graph)
  - **Implementation details:** Add your neighbours to a **queue**, pop from the queue to get next node
  - **Runtime:** $O(V + E)$

Graph Algorithms: Traversal

- **BFS:**
  - **Purpose:** Reachability, Shortest Path (unweighted graph)
  - **Implementation details:** Add your neighbours to a **queue**, pop from the queue to get next node
  - **Runtime:** $O(V + E)$
- **DFS:**
  - **Purpose:** Reachability, toposort
  - **Implementation details:** Add your neighbours to a **stack**, pop from the stack to get next node
  - **Runtime:** $O(V + E)$

Graph Algorithms: Shortest Path

- **Dijkstra's**
  - **Purpose:** SSSP, no negative edges
  - **Implementation:** Visit neighbours in **priority queue**
  - **Runtime:** $O(m \log n)$ (with **Fibonacci Heap**, $O(m + n \log n)$)

Graph Algorithms: Shortest Path

- **Dijkstra's**
  - **Purpose:** SSSP, no negative edges
  - **Implementation:** Visit neighbours in **priority queue**
  - **Runtime:** $O(m \log n)$ (with **Fibonacci Heap**, $O(m + n \log n)$)
- **Bellman-Ford:**
  - **Purpose:** SSSP, yes negative weights. Will detect negative cycles.
  - **Implementation:** Dynamic Programming recurrence:
    - $d(v, k)$ is the shortest-walk distance from s to v using at most k edges
    - $d(v, k) = \min\left(d(v, k-1), \min_{u \to v} d(u, k-1) + \ell(u \to v)\right)$
  - **Runtime:** $O(mn)$

Graph Algorithms: Longest Path

Longest path in a Directed Acyclic Graph (DAG)

- **Purpose:** To find the longest simple path (no repeating vertices) by weight in a graph which is guaranteed to be a DAG[1].
- **Runtime[2]:** $O(V + E)$
- **Returns:** The sum of the weights of the longest path in the DAG.

[1] Finding the longest path in other types of graphs is at least NP-hard.
[2] This is a relatively straight-forward DP on a DAG problem if you wish to derive it.

Graph Problems: General Stuff

How to solve graph problems:

1. Identify type of problem (Reachability, Shortest Path, SCC)
2. Construct new graph
   - Add sources/sinks
   - Add vertices via $V' = V \times \{\text{some set}\}$ (Useful for tracking states)
   - Add vertices via $E' = E \times \{\text{some set}\}$ (Useful for allowing/prohibit certain behaviour)
3. Apply some stock algorithm **(DO NOT MODIFY THE ALGORITHMS - MODIFY THE INPUTS!)**
4. Draw connection between how to result of the algorithm upon the new graph relates to the solution of the original question.

Recurrences and Asymptotics

Give a tight asymptotic bound for each recurrence:

- $T(n) = 4\ T(\frac{n}{2}) + n \log_2 n$

Recurrences and Asymptotics

Give a tight asymptotic bound for each recurrence:

- $T(n) = 4\ T(\frac{n}{2}) + n \log_2 n$
- $T(n) = T(\frac{3n}{4}) + T(\frac{n}{4}) + 5n$

Recurrences and Asymptotics

Give a tight asymptotic bound for each recurrence:

- $T(n) = 4\ T(\frac{n}{2}) + n \log_2 n$
- $T(n) = T(\frac{3n}{4}) + T(\frac{n}{4}) + 5n$
- $T(n) = 9\ T(\frac{n}{3}) + n^2$

Recurrences and Asymptotics

Give a tight asymptotic bound for each recurrence:

- $T(n) = 4\ T(\frac{n}{2}) + n \log_2 n$
- $T(n) = T(\frac{3n}{4}) + T(\frac{n}{4}) + 5n$
- $T(n) = 9\ T(\frac{n}{3}) + n^2$

Group the following functions s.t. f and g are in the same group if $f(x) \sim \Theta(g(x))$, and sort the groups by runtime:

- n^n
- $\log \log n$
- 374^n
- $n!$
- $\log(n + n^{374})$
- $\log n^n$
- $n^{1.000001}$
- 2^n
- $n \log n^5$
- $\frac{1}{\log_n 2}$

Recurrences and Asymptotics

Divide and Conquer

Consider the following (correct!) in-place sorting algorithm:

```
1: procedure StoogeSort(A[1 ... n])
2:     If A[1] > A[n], swap them.
3:     if n ≥ 3 then
4:         StoogeSort the initial 2/3 of A
5:         StoogeSort the final 2/3 of A
6:         StoogeSort the initial 2/3 of A (again)
```

Give a *tight* asymptotic bound on the runtime of StoogeSort, in terms of n.

Divide and Conquer

Divide and Conquer

Suppose you have two arrays: A of length n_1, sorted in ascending order, and B of length n_2, sorted in descending order. You decide to concatenate them to form a new array $C = A \cdot B$ for some application.

All the numbers are distinct. After concatenation, you no longer care about A and B individually, and you don't know n_1 or n_2. You only have C and its size $n = n_1 + n_2$.

For example, if $A = [1, 2, 6, 7]$ and $B = [8, 4, 3]$, then $C = [1, 2, 6, 7, 8, 4, 3]$.

Given a number x, we want to check whether x appears in C.

Divide and Conquer

Divide and Conquer

Suppose we are given a three-dimensional array $A[1 \ldots n, 1 \ldots n, 1 \ldots n]$ of distinct integers. An array element $A[i, j, k]$ is called a *local minimum* if it is smaller than its six immediate neighbors:

$$A[i,j,k] < \min \left\{ \begin{array}{ll} A[i-1,j,k], & A[i+1,j,k], \\ A[i,j-1,k], & A[i,j+1,k], \\ A[i,j,k-1], & A[i,j,k+1] \end{array} \right\}$$

To avoid edge cases, we assume all cells on the boundary of the array have value $+\infty$.

Describe and analyze an algorithm to find a local minimum in A as quickly as possible.

Divide and Conquer

Dynamic Programming

A shuffle of two strings X and Y is formed by interspersing the characters into a new string, keeping the characters of X and Y in the same order. For example, the string BANANAANANAS is a shuffle of the strings BANANA and ANANAS in several different ways.

BANANAANANAS BANANAANANAS BANANAANANAS

Similarly, the strings PRODGYRNAMAMMIINCG and DYPRONGARMAMMICING are both shuffles of the strings DYNAMIC and PROGRAMMING:

PRODGYRNAMAMMIINCG DYPRONGARMAMMICING

Given three strings $A[1..m]$, $B[1..n]$, and $C[1..m+n]$, describe and analyze an algorithm to determine whether C is a shuffle of A and B.

DP

Dynamic Programming

The **StupidScript** language includes a binary operator @ that computes the *average* of its two arguments. For example, the StupidScript code `print(3 @ 6)` would print 4.5, because (3 + 6)/2 = 4.5.
Expressions like 3@7@4 that use the @ operator more than once yield different results when they are evaluated in different orders:

$$(3@7)@4 = 5@4 = 4.5 \quad \text{but} \quad 3@(7@4) = 3@5.5 = 4.25$$

Here is a larger example:

$$\begin{aligned}(((((8@6)@7)@5)@3)@(0@9)) &= 4.5\\ ((8@6)@(7@5))@((3@0)@9) &= 5.875\\ (8@(6@(7@(5@(3@0)))))@9 &= 7.890625\end{aligned}$$

Your goal for this problem is to describe and analyze an algorithm to compute, given a sequence of integers separated by @ signs, the **largest possible** value the expression can take by adding parentheses. Your input is an array $A[1 \ldots n]$ listing the sequence of integers.

For example, if your input sequence is $[3, 7, 4]$, your algorithm should return 4.5, and if your input sequence is $[8, 6, 7, 5, 3, 0, 9]$, your algorithm should return 7.890625. Assume all arithmetic operations (including @) can be performed exactly in $O(1)$ time.

DP

Graphs

It's late at night, and you're walking home. You have a (directed, positive weighted) graph $G = (V, E)$ describing Champaign's road network, with each edge annotated as to whether or not the edge is lit by streetlights. On your way home, you need to stop at a special vertex q. Describe and analyze an efficient algorithm to calculate the shortest $s \to t$ route for a given (s, t) (stopping at q), where at most k edges are unlit.

Graphs

Graphs

Let $G = (V, E)$ be a weighted DAG with n vertices and m edges, where each edge $e \in E$ has a cost $c(e) \in \mathbb{R}$. We are given four vertices $s, t, s', t' \in V$, a path length $k \leq n$, and a threshold L.
We want to find two length-k paths in G,

$$P = \langle s, u_1, u_2, \ldots, u_{k-1}, t \rangle \quad \text{and} \quad Q = \langle s', v_1, v_2, \ldots, v_{k-1}, t' \rangle,$$

such that

$$\sum_{i=0}^{k-1} \left| c(u_i, u_{i+1}) - c(v_i, v_{i+1}) \right| \geq L,$$

where $(u_0, u_1) = (s, u_1)$, $(u_{k-1}, u_k) = (u_{k-1}, t)$, and similarly for v_i.

Graphs

Graphs

We are given a directed graph $G = (V, E)$ with n vertices and m edges, where $m \geq n$. Each edge $e \in E$ has a weight $w(e) \geq 0$, and all weights are distinct. Let s and t be two distinct vertices in V.
For a path π from s to t, we define its **bottleneck value** as

$$b(\pi) = \max_{e \in \pi} w(e),$$

i.e., the weight of the heaviest edge along π.
We want to find a path from s to t that **minimizes the bottleneck value**.

Graphs

Feedback

- Further questions in ACM (Siebel 1104) after this
- Please fill out the feedback form:
  `go.acm.illinois.edu/cs374a_mt2_feedback`