**CS/ECE-374-B: Algorithms and Models of Computation, Spring 2024**

# Midterm exam 2 – March 26, 2024

---

- You can do hard things! Grades do matter, but not as much as you may think, but then life is uncertain anyway, so what.
- **Don't cheat.** The consequence for cheating is far greater than the reward. Just try your best and you'll be fine.
- **Please read the entire exam before writing anything.** There are 6 problems and most have multiple parts.
- This is a closed-book exam. At the end of the exam, you'll find a multi-page cheat sheet. *Do not tear out the cheat sheet!* No outside material is allowed on this exam.
- You should write your answers legibly and in the space given for the question. Overly verbose answers will be penalized.
- Scratch paper is available on the back of the exam. *Do not tear out the scratch paper!* It messes with the auto-scanner.
- **You have 75 minutes (1.25 hours) for the exam.** Manage your time well. *Do not spend too much time on questions you do not understand and focus on answering as much as you can!*
- We know that this exam is **shorter in length** compared to the first one. Make sure you *use the time well to think, be precise, and show as much work as possible*.

---

Name: ______________________________

NetID: ______________________________

Date: ______________________________

## Problem 1 [10 points]

For each of the following statements, answer if it is `True` or `False`. Use the table at the bottom to mark you choices.

i. Recursion is a special case of reduction that includes reducing the problem into smaller instances of itself.

ii. Merging two sorted arrays each of size $n$ into a single sorted array requires a minimum $O(n \log n)$ time.

iii. If topological sort exists for a graph then that graph has a cycle.

iv. If for some two nodes, the pre-post numbering intervals in DFS are disjoint then it means that the graph is disconnected.

v. Checking if a sequence is increasing or not can be achieved in a minimum of $O(n)$ time.

vi. The node with maximum post numbering in DFS is in a sink strongly connected component of the original graph.

vii. Every directed acyclic graph has either a source or a sink but not both.

viii. The asymptotic runtime of Merge Sort depends on the number of splits one makes of the original array.

ix. If a graph has a cycle then there is a back-edge in its DFS.

x. Decreasingly sorted pre numberings in DFS give a topological sort of the given directed acyclic graph.

**Table 1.**

| Statement | Your choice |
|---|---|
| i. | |
| ii. | |
| iii. | |
| iv. | |
| v. | |
| vi. | |
| vii. | |
| viii. | |
| ix. | |
| x. | |

## Problem 2 [15 points]

Solve the following recurrence relations *exactly*, i.e, obtain a closed form formula for $f(n)$ without order terms/bounds.

Useful formula: $\sum_{k=0}^{n} ar^k = \frac{a(1-r^{n+1})}{1-r}$.

a. $f(n) = f(n-1) + n2^n,\ n > 0$ and $f(0) = 3$.

b. $f(n) = 2f(n-1) + 1,\ n > 1$ and $f(1) = 1$.

c. $f(n) = 2f(\frac{n}{2}) + n,\ n > 1$ and $f(1) = 1$.

## Problem 3 [20 points]

Longest palindromic subsequence (LPS) of a sequence is defined as a subsequence of maximum length that is also a palindrome. For example, given the sequence `BANANA`, an LPS is `ANANA` and has length 5.

Write a dynamic programming algorithm to obtain the length of an LPS of a given sequence by providing the following.

- **Recurrence and short English description (in terms of the parameters):**

- **Memoization data structure and evaluation order:**

- **Return value:**

- **Time Complexity:**

## Problem 4 [20 points]

Consider the graph in Figure 1.

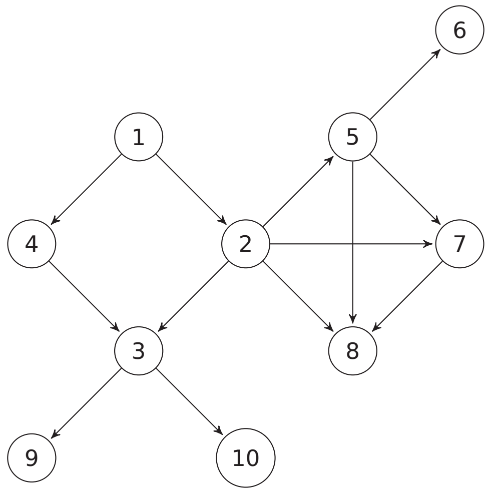



**Figure 1.** Graph.

Perform DFS starting from vertex 1 while breaking ties in the numeric order, i.e., the node with a smaller numeric label is visited first in tying situations and answer the following.

a. What is the DFS traversal order, i.e., the order in which you visit different vertices.

b. What are the DFS pre and post numberings of different vertices.

**Table 2.**

| Node | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|---|
| Pre numbering | | | | | | | | | | |
| Post numbering | | | | | | | | | | |

c. Draw the DFS spanning tree.

d. Classify the non-tree edges into forward, backward, and cross edges. You may draw these non-tree edges in the DFS spanning tree using different edge-styles, colors, etc.

e. Using DFS or otherwise, obtain a topological sort of the given graph.

## Problem 5 [20 points]

a. Consider the following recursive equation.

$$A(u,v) = \sum_{i=1}^{\min(u,v)} u\, A(u+v-i, i-1), \;\; A(0,v) = v, \text{ and } A(u,0) = u.$$

Analyze the runtime of the memoized implementation of the above recursion to compute $A(n,n)$. Show your work.

b. What is the relation between edit distance and longest common subsequence? Explain in detail.

c. Analyze the runtime of the median-of-median algorithm with group size as 3 instead of the usual 5.

d. Provide a logarithmic time algorithm to count the number of instances of a given number in a sorted list.

## Problem 6 [15 points]

Consider the problem of multiplying $n$, $m$-digit numbers, $m \geq n$. One simple strategy to solve this problem is to use Karatsuba's algorithm $n-1$ times, i.e., multiply first number with the second number, then their product with the third number, and so on. Answer the following.

a. Analyze the runtime of the above simple strategy.

b. Provide a more time-efficient divide-and-conquer recursive algorithm to solve given problem.

c. Analyze the runtime of your algorithm.

*This page is for additional scratch work!*

# ECE 374 B Algorithms: Cheatsheet

## 1 Recursion

### Simple recursion

**Definitions**

- **Reduction:** solve one problem using the solution to another.
- **Recursion:** a special case of reduction - reduce problem to a *smaller* instance of *itself* (self-reduction).
  - Problem instance of size $n$ is reduced to *one or more* instances of size $n-1$ or less.
  - For termination, problem instances of small size are solved by some other method as *base cases*

**Tower of Hanoi**

Arguably the most famous example of recursion. The goal is to move $n$ disks one at a time from the first peg to the last peg.

Pseudocode: Tower of Hanoi

**Hanoi** ($n$, src, dest, tmp):
    **if** ($n > 0$) **then**
        **Hanoi** ($n-1$, src, tmp, dest)
        Move disk $n$ from src to dest
        **Hanoi** ($n-1$, tmp, dest, src)

### Recurrences

Suppose you have a recurrence of the form $T(n) = rT(n/c) + f(n)$.

The *master theorem* gives a good asymptotic estimate of the recurrence. If the work at each level is:

| | | |
|---|---|---|
| Decreasing: | $rf(n/c) = \kappa f(n)$ where $\kappa < 1$ | $T(n) = O(f(n))$ |
| Equal: | $rf(n/c) = f(n)$ | $T(n) = O(f(n) \cdot \log_c n)$ |
| Increasing: | $rf(n/c) = Kf(n)$ where $K > 1$ | $T(n) = O(n^{\log_c r})$ |

Some useful identities:

- Sum of integers: $\sum_{k=1}^{n} k = \frac{n(n+1)}{2}$
- Geometric series closed-form formula: $\sum_{k=0}^{n} ar^k = \frac{1-r^{n+1}}{1-r}$
- Logarithmic identities: $\log(ab) = \log a + \log b$, $\log(a/b) = \log a - \log b$, $a^{\log_c b} = b^{\log_c a}$ $(a, b, c > 1)$.

### Backtracking

*Backtracking* is the algorithm paradigm involving guessing the solution to a single step in some multi-step process and recursing backwards if it doesn't lead to a solution. For instance, consider the longest increasing subsequence (LIS) problem. You can either check all possible subsequences:

Pseudocode: LIS - Naive enumeration

**algLISNaive**($A[1..n]$):
    $maxmax = 0$
    **for** each subsequence $B$ of $A$ **do**
        **if** $B$ is increasing and $|B| >$ max **then**
            $max = |B|$
    **return** max

On the other hand, we don't need to generate every subsequence; we only need to generate the subsequences that are increasing:

Pseudocode: LIS - Backtracking

**LIS_smaller**($A[1..n], x$):
    **if** $n = 0$ **then return** $0$
    max = **LIS_smaller**($A[1..n-1], x$)
    **if** $A[n] < x$ **then**
        max $= \max\{$max, $1 +$ **LIS_smaller**$(A[1..(n-1)], A[n])\}$
    **return** max

### Divide and conquer

*Divide and conquer* is an algorithm paradigm involving the decomposition of a problem into the same subproblem, solving them separately and combining their results to get a solution for the original problem.

**Sorting algorithms**

| Algorithm | Runtime | Space |
|---|---|---|
| Mergesort | $O(n \log n)$ | $O(n \log n)$<br>$O(n)$ (if optimized) |
| Quicksort | $O(n^2)$<br>$O(n \log n)$ if using MoM | $O(n)$ |

**Karatsuba's algorithm**

We can divide and conquer multiplication like so:

$$bc = 10^n b_L c_L + 10^{n/2}(b_L c_R + b_R c_L) + b_R c_R.$$

We can rewrite the equation as:

$$\begin{aligned} bc = b(x)c(x) = (b_L x + b_R)(c_L x + c_R) = (b_L c_L)x^2 \\ + ((b_L + b_R)(c_L + c_R) - b_L c_L - b_R c_R)\, x \\ + b_R c_R, \end{aligned}$$

Its running time is $O(n^{\log_2 3}) = O(n^{1.585})$.

### Linear time selection

The *median of medians* (MoM) algorithms give a element that is larger than $\frac{3}{10}$'s and smaller than $\frac{7}{10}$'s of the array elements. This is used in the linear time selection algorithm to find element of rank $k$.

Pseudocode: Quickselect with median of medians

**Median-of-medians** ($A$, $i$):
    sublists = [A[j:j+5] **for** j $\leftarrow 0, 5, \ldots,$ len($A$)]
    medians = [**sorted** (sublist)[**len** (sublist)/2]
            **for** sublist $\in$ sublists]

    // Base case
    **if len** (A) $\leq$ 5 **return sorted** (a)[i]

    // Find median of medians
    **if len** (medians) $\leq$ 5
        pivot = **sorted** (medians)[**len** (medians)/2]
    **else**
        pivot = **Median-of-medians** (medians, **len**/2)

    // Partitioning step
    low = [j **for** j $\in$ A **if** j < pivot]
    high = [j **for** j $\in$ A **if** j > pivot]

    k = **len** (low)
    **if** i < k
        **return Median-of-medians** (low, i)
    **else if** i > k
        **return Median-of-medians** (low, i-k-1)
    **else**
    **return** pivot

## Dynamic programming

*Dynamic programming* (DP) is the algorithm paradigm involving the computation of a recursive backtracking algorithm iteratively to avoid the recomputation of any particular subproblem.

### Longest increasing subsequence

The longest increasing subsequence problem asks for the length of a longest increasing subsequence in a unordered sequence, where the sequence is assumed to be given as an array. The recurrence can be written as:

$$\textit{LIS}(i,j) = \begin{cases} 0 & \text{if } i = 0 \\ \textit{LIS}(i-1,j) & \text{if } A[i] \geq A[j] \\ \max \begin{cases} \textit{LIS}(i-1,j) \\ 1 + \textit{LIS}(i-1,i) \end{cases} & \text{else} \end{cases}$$

Pseudocode: LIS - DP

```
LIS-Iterative(A[1..n]):
    A[n+1] = ∞
    for j ← 0 to n
        if A[i] ≤ A[j] then LIS[0][j] = 1

    for i ← 1 to n − 1 do
        for j ← i to n − 1 do
            if A[i] ≥ A[j]
                LIS[i, j] = LIS[i − 1, j]
            else
                LIS[i, j] = max {LIS[i − 1, j],
                                 1 + LIS[i − 1, i]}
    return LIS[n, n + 1]
```

### Edit distance

The edit distance problem asks how many edits we need to make to a sequence for it to become another one. The recurrence is given as:

$$\mathrm{Opt}(i,j) = \min \begin{cases} \alpha_{x_i y_j} + \mathrm{Opt}(i-1,j-1), \\ \delta + \mathrm{Opt}(i-1,j), \\ \delta + \mathrm{Opt}(i,j-1) \end{cases}$$

**Base cases:** $\mathrm{Opt}(i,0) = \delta \cdot i$ and $\mathrm{Opt}(0,j) = \delta \cdot j$

Pseudocode: Edit distance - DP

```
EDIST(A[1..m], B[1..n])
    for i ← 1 to m do M[i, 0] = iδ
    for j ← 1 to n do M[0, j] = jδ

    for i = 1 to m do
        for j = 1 to n do
```

$$M[i][j] = \min \begin{cases} COST\,[A[i]]\,[B[j]] \\ \qquad + M[i-1][j-1], \\ \delta + M[i-1][j], \\ \delta + M[i][j-1] \end{cases}$$

# 2 Graph algorithms

## Graph basics

A graph is defined by a tuple $G = (V, E)$ and we typically define $n = |V|$ and $m = |E|$. We define $(u, v)$ as the edge from $u$ to $v$. Graphs can be represented as **adjacency lists**, or **adjacency matrices** though the former is more commonly used.

- *path*: sequence of *distinct* vertices $v_1, v_2, \dots, v_k$ such that $v_i v_{i+1} \in E$ for $1 \leq i \leq k-1$. The length of the path is $k-1$ (the number of edges in the path).
  *Note:* a single vertex $u$ is a path of length 0.
- *cycle*: sequence of *distinct* vertices $v_1, v_2, \dots, v_k$ such that $(v_i, v_{i+1}) \in E$ for $1 \leq i \leq k-1$ and $(v_k, v_1) \in E$. A single vertex is not a cycle according to this definition.
  *Caveat:* Sometimes people use the term cycle to also allow vertices to be repeated; we will use the term *tour*.
- A vertex $u$ is *connected* to $v$ if there is a path from $u$ to $v$.
- The *connected component* of $u$, $\mathrm{con}(u)$, is the set of all vertices connected to $u$.
- A vertex $u$ can *reach* $v$ if there is a path from $u$ to $v$. Alternatively $v$ can be reached from $u$. Let **rch**$(u)$ be the set of all vertices reachable from $u$.

## Directed acyclic graphs

Directed acyclic graphs (dags) have an intrinsic ordering of the vertices that enables dynamic programming algorithms to be used on them.
A *topological ordering* of a dag $G = (V, E)$ is an ordering $\prec$ on $V$ such that if $(u, v) \in E$ then $u \prec v$.

**Pseudocode: Kahn's algorithm**

**Kahn**($G(V, E)$,$u$):
toposort$\leftarrow$empty list
**for** $v \in V$:
  in$(v) \leftarrow |\{u \mid u \to v \in E\}|$
**while** $v \in V$ that has in$(v) = 0$:
  Add v to end of toposort
  Remove $v$ from $V$
  **for** $v$ in $u \to v \in E$:
    in$(v) \leftarrow$ in$(v) - 1$
**return** toposort

**Running time:** $O(n + m)$

- A dag may have multiple topological sorts.
- A topological sort can be computed by DFS, in particular by listing the vertices in decreasing post-visit order.

## DFS and BFS

**Pseudocode: Explore (DFS/BFS)**

**Explore**($G$,$u$):
  **for** $i \leftarrow 1$ to $n$:
    Visited$[i] \leftarrow$ False
  Add $u$ to ToExplore and to $S$
  Visited$[u] \leftarrow$ True
  Make tree $T$ with root as $u$
  **while** B is non-empty **do**
    Remove node $x$ from B
    **for** each edge $(x, y)$ in $Adj(x)$ **do**
      if Visited$[y]$ = False
        Visited$[y] \leftarrow$ True
        Add $y$ to B, $S$, $T$ (with $x$ as parent)

Note:

- If B is a queue, *Explore* becomes BFS.
- If B is a stack, *Explore* becomes DFS.

---

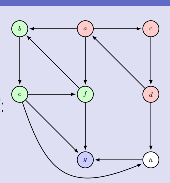
**Pre/post numbering**

Pre and post numbering aids in analyzing the graph structure. By looking at the numbering we can tell if a edge $(u, v)$ is a:

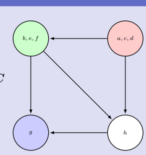
- *Forward edge*: pre$(u) <$ pre$(v) <$ post$(v) <$ post$(u)$
- *Backward edge*: pre$(v) <$ pre$(u) <$ post$(u) <$ post$(v)$
- *Cross edge*: pre$(u) <$ post$(u) <$ pre$(v) <$ post$(v)$

## Strongly connected components

- Given $G$, $u$ is *strongly connected* to $v$ if $v \in$ rch$(u)$ *and* $u \in$ rch$(v)$.
- A *maximal* group of $G$: vertices that are all strongly connected to one nother is called a strong component.



$G^{SCC}$



**Pseudocode: Metagraph - linear time**

**Metagraph**($G(V, E)$):
  Compute rev$(G)$ by brute force
  ordering $\leftarrow$ reverse postordering of $V$ in rev$(G)$
    by **DFS**(rev$(G)$, $s$) for any vertex $s$
  Mark all nodes as unvisited
  **for** each $u$ in ordering **do**
    **if** $u$ is not visited and $u \in V$ **then**
      $S_u \leftarrow$ nodes reachable by $u$ by **DFS**$(G, u)$
      Output $S_u$ as a strong connected component
      $G(V, E) \leftarrow G - S_u$

## Shortest paths

**Dijkstra's algorithm:**
Find minimum distance from vertex $s$ to **all** other vertices in graphs *without* negative weight edges.

**Pseudocode: Dijkstra**

**for** $v \in V$ **do**
  $d(v) \leftarrow \infty$
$X \leftarrow \varnothing$
$d(s, s) \leftarrow 0$
for $i \leftarrow 1$ to $n$ **do**
  $v \leftarrow \arg\min_{u \in V - X} d(u)$
  $X = X \cup \{v\}$
  **for** $u$ in Adj$(v)$ **do**
    $d(u) \leftarrow \min\{(d(u),\ d(v) + \ell(v, u))\}$
**return** $d$

**Running time:** $O(m + n\log n)$ (if using a Fibonacci heap as the priority queue)

---

**Bellman-Ford algorithm:**
Find minimum distance from vertex $s$ to **all** other vertices in graphs *without* negative cycles. It is a DP algorithm with the following recurrence:

$$d(v, k) = \begin{cases} 0 & \text{if } v = s \text{ and } k = 0 \\ \infty & \text{if } v \neq s \text{ and } k = 0 \\ \min \begin{cases} \min_{uv \in E}\{d(u, k-1) + \ell(u, v)\} \\ d(v, k-1) \end{cases} & \text{else} \end{cases}$$

**Base cases:** $d(s, 0) = 0$ and $d(v, 0) = \infty$ for all $v \neq s$.

**Pseudocode: Bellman-Ford**

**for** each $v \in V$ **do**
  $d(v) \leftarrow \infty$
$d(s) \leftarrow 0$

**for** $k \leftarrow 1$ to $n - 1$ **do**
  **for** each $v \in V$ **do**
    **for** each edge $(u, v) \in$ in$(v)$ **do**
      $d(v) \leftarrow \min\{d(v), d(u) + \ell(u, v)\}$

**return** $d$

**Running time:** $O(nm)$

---

**Floyd-Warshall algorithm:**
Find minimum distance from *every* vertex to *every* vertex in a graph *without* negative cycles. It is a DP algorithm with the following recurrence:

$$d(i, j, k) = \begin{cases} 0 & \text{if } i = j \\ \infty & \text{if } (i, j) \notin E \text{ and } k = 0 \\ \min \begin{cases} d(i, j, k-1) \\ d(i, k, k-1) + d(k, j, k-1) \end{cases} & \text{else} \end{cases}$$

Then $d(i, j, n-1)$ will give the shortest-path distance *from* $i$ *to* $j$.

**Pseudocode: Floyd-Warshall**

**Metagraph**($G(V, E)$):
  **for** $i \in V$ **do**
    **for** $j \in V$ **do**
      $d(i, j, 0) \leftarrow \ell(i, j)$
      (* $\ell(i, j) \leftarrow \infty$ if $(i, j) \notin E$, 0 if $i = j$ *)

  **for** $k \leftarrow 0$ to $n - 1$ **do**
    **for** $i \in V$ **do**
      **for** $j \in V$ **do**
        $d(i, j, k) \leftarrow \min \begin{cases} d(i, j, k-1), \\ d(i, k, k-1) + d(k, j, k-1) \end{cases}$
  **for** $v \in V$ **do**
    **if** $d(i, i, n-1) < 0$ **then**
      **return** "$\exists$ negative cycle in $G$"

  **return** $d(\cdot, \cdot, n-1)$

**Running time**: $\Theta(n^3)$