ECE 374 B: Algorithms and Models of Computation, Fall 2023
Midterm 2 – October 31, 2023

• You will have 75 minutes (1.25 hours) to solve 6 problems. Most have multiple parts. Don’t spend too much time on questions you don’t understand and focus on answering as much as you can!

• No resources are allowed for use during the exam except a multi-page cheatsheet and scratch paper on the back of the exam. Do not tear out the cheatsheet or the scratch paper! It messes with the auto-scanner.

• You should write your answers completely in the space given for the question. We will not grade parts of any answer written outside of the designated space.

• Please use a dark-colored pen unless you are absolutely sure your pencil writing is forceful enough to be legible when scanned. We will take off points if we have difficulty reading the uploaded document.

• Incorrect algorithms will receive a score of 0, but slower than necessary but correct algorithms will always receive some points, even brute force ones. Thus, you should prioritize the correctness of your submitted algorithms over speed; you will receive more points that way. On the other hand, submit the fastest algorithms that you know are correct; faster algorithms will receive more points.

• Any recursive backtracking algorithm or dynamic programming algorithm given without an English description of the recursive function (i.e., a description of the output of the function in terms of their inputs) will receive a score of 0.

• Any greedy algorithm or a modification of a standard graph algorithm given without a proof of correctness will receive a score of 0.

• Any algorithms written in actual code instead of pseudocode will receive a score of 0.

• For problems with a graph given as input, you may assume the graph is simple (i.e., it has no self-loops or parallel edges).

• Only algorithms referenced in the cheat sheet may be referred to as a “black box”. You may not simply refer to a prior lab/homework for the solution and must give the full answer.

• Unless explicitly mentioned, a runtime analysis is required for each given algorithm.

• Don’t cheat. If we catch you, you will get an F in the course.

• Good luck!

Name: __________________________________________

NetID: __________________________________________

Date: __________________________________________
1 Short answer (2 questions) - 20 points

Answer the following questions. You may briefly (no more than 2 sentences) justify your answers, but a complete proof is not required.

(a) Give a tight asymptotic upper-bound for the following recurrences:

(i) 
\[ A(n) = 2A(n-2) + n \quad A(0) = A(1) = A(2) = 1 \]

(ii) 
\[ B(n) = 3B(n/2) + n^2 \quad B(0) = B(1) = 1 \]

(b) Give the recurrence that describes the following program. What is the asymptotic, upper-bound, of the running time?

```
foolishness(n)
    sum=0
    if n = 1,
        return sum = 1
    for i from 1 to n^3:
        sum += 4*i
    return foolishness (n-1)
```

Recurrence:

Running time:
2 Short answer II (5 questions) - 30 points

Answer the following questions. You may briefly (no more than 2 sentences) justify your answers, but a complete proof is not required. For the following graph problems, use the notation $G = (V, E)$, $n = |V|$ and $m = |E|$.

(a) Suppose we implemented a vanilla version of QuickSort where the pivot was always chosen from the first element of the array. Under what input array will the QuickSort algorithm result in a running time of $O(n^2)$?

(b) In the longest increasing subsequence algorithm (found in the cheat sheet), we defined a recurrence $LIS(i, j)$. Give an English description (no more than 2 sentences) of what $LIS(i, j)$ represents. Note: what's in the cheat sheet does not constitute a english description for the recurrence.
(c) Given a graph with \( n \) vertices and no edges, how many topological sorts will this graph have?

(d) You are given a graph \( (G = (V, E)) \) and two vertices \( a \) and \( b \). Write a algorithm that determines if both \( a \) and \( b \) are part of the same strongly connected component. What is the run-time of this algorithm?

(e) In the Bellman-Ford algorithm (found in the cheat sheet), we defined a recurrence \( d(v, k) \). Give an English description (no more than 2 sentences) of what \( d(v, k) \) represents. Note: what’s in the cheat sheet does not constitute a english description for the recurrence.
3 Recursion - 10 points

Suppose we are given an array \( A[1..n] \) of \( n \) distinct integers, which could be positive, negative, or zero, sorted in increasing order so that \( A[1] < A[2] < \cdots < A[n] \). Describe a fast algorithm that either computes an index \( i \) such that \( A[i] = n - i + 1 \) or correctly reports that no such index exists.

(You cannot simply reference a prior lab/homework. You must give the actual solution.)
4 Finding outliers - 10 points

The ECE374 staff has just completed grading midterm 2. They have a list of $S[1...n]$ of unsorted exam scores and are searching for outliers on either end. The interquartile range is defined by $Q_3 - Q_1$, that is the 75th percentile score – 25th percentile score. Specifically, an outlier is someone who scores $> Q_3 + 1.5 \times IQR$ (interquartile range) or someone who scores $< Q_1 - 1.5 \times IQR$. Give an efficient $O(n)$ program to (1) compute the interquartile range, and (2) output all the outlier scores. A solution that uses hashing or sorting will be awarded 0 points.
5 Dynamic programming (1 questions) - 15

You are given an integer value $x$ and an array $A$ where each element of the array represents a coin denomination. Describe a dynamic programming algorithm that returns the number of ways to make change for $x$.

Example: $A = [1, 2, 3]$ and $x = 5$. Output is 5 ($\{1, 1, 1, 1\}, \{1, 1, 1, 2\}, \{1, 1, 3\}, \{1, 2, 2\}, \{2, 3\}$).

(You cannot simply reference a prior lab/homework. You must give the actual solution.)

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

Memoization data structure and evaluation order:

Return value:

Time Complexity:
6 Graph algorithm - 15 points

You are given two weighted, directed acyclic graphs \( G = (V, E_G) \) and \( H = (V, E_H) \) that have the same vertices but different edges. Every edge is assigned an integer value that could be positive or negative.

You are given two nodes \( s \) and \( t \). The length of a path in each graph is defined as \( \ell_G(s, t) \) for \( G \) and \( \ell_H(s, t) \) for \( H \). You need to find the path that exists in both \( G \) and \( H \) with the minimum combined length (in other words, find minimum value of \( \ell_G(s, t) + \ell_H(s, t) \)). (Hint: topological sort is your friend)

(You cannot simply reference a prior lab/homework. You must give the actual solution.)

Note: These details don’t necessarily matter but might be useful to some. Both graphs are given as adjacency matrices where 0 indicates no edge and 1 indicates an edge. The weights are given by a weight matrix \( w_G[i, j] \) or \( w_H[i, j] \) for \( G \) and \( H \) respectively. Remember pseudo-code is one of the worst ways to describe your algorithm(s); be clear and succinct.
This page is for additional scratch work!
1 Recursion

Simple recursion

- **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).

**Definitions**

- 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.

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.

**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)
```

**Towers of Hanoi**

Disks one at a time from the first peg to the last peg.

```
Hanoi(n):
  if \( n > 0 \) then
    Hanoi(n-1)
    Move disk n from src to dest
    Hanoi(n-1)
```

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: \( r f(n/c) = \alpha f(n) \) where \( \alpha < 1 \)
  \( T(n) = O(f(n)) \)
- Equal: \( r f(n/c) = f(n) \)
  \( T(n) = O(f(n) \cdot \log_c n) \)
- Increasing: \( r f(n/c) = K f(n) \) where \( K > 1 \)
  \( T(n) = O(n^{\log_c r}) \)

Some useful identities:

- Sum of integers: \( \sum_{k=0}^{n} k = \frac{n(n+1)}{2} \)
- Geometric series closed-form formula: \( \sum_{k=0}^{n} a k = \frac{a_1 - a_1 r^{n+1}}{1-r} \)
- Geometric series closed-form formula: \( \sum_{k=0}^{n} b k = \frac{b_a - b_a r^{n+1}}{1-r} \)
- Logarithmic identities: \( \log (a b) = \log a + \log b \), \( \log (a/b) = \log a - \log b \), \( a^{\log_b c} = \frac{a^{\log_b c}}{a^0} \), \( a > 0, b > 1 \), \( \log_a b = \log_c b / \log_c a \).

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:

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

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

```
algLISNaive(A[1..n], x):
  if n = 0 then return 0
  max = algLISNaive(A[1..n-1], x)
  if A[i] < x then
    max = max { max + 1, algLISNaive(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.

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Runtime</th>
<th>Space</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mergesort</td>
<td>( O(n \log n) )</td>
<td>( O(n \log n) )</td>
</tr>
<tr>
<td>Quicksort</td>
<td>( O(n^2) ) if using MoM</td>
<td>( O(n) )</td>
</tr>
</tbody>
</table>

We can divide and conquer multiplication like so:

\[
b_c = 10^n b_L c_L + 10^{n/2} (b_L c_R + b_R c_L) + b_R c_R.
\]

We can rewrite the equation as:

\[
b_c = b(x)c(x) = (b_L x + b_R)(c_L x + c_R) = (b_L c_L) x^2 + (b_L + b_R)(c_L + c_R) x + b_R c_R - b_L c_R x.
\]

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 \( \frac{2}{3} \)’s and smaller than \( \frac{1}{3} \)’s of the array elements. This is used in the linear time selection algorithm to find element of rank \( k \).

```
Median-of-medians(A, i):
  subs = [A[j] for j < i; ...; len(A)]
  medians = \{ median(sublist) for sublist in subs \}
  pivot = \{ \}

  // Impe case
  if len(A) <= 5 return sorted(A)
  // Find med of med
  if len(medians) <= 5
    pivot = sorted(medians[len(medians)//2])
  else
    pivot = Median-of-medians(medians, len(medians)//2)

  // Partitioning step
  low = \{ \}
  high = \{ j \}
  for j in A if j > pivot
  k = len(low)
  if k < i return Median-of-medians(low, i)
  else if k > i return Median-of-medians(high, 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 an unordered sequence, where the sequence is assumed to be given as an array. The recurrence can be written as:

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

Pseudocode: LIS - DP

\[
\text{LIS-Iterative}(A[1..n]):
\]
\[
A[n + 1] = \infty
\]
\[
\text{for } j \leftarrow 0 \text{ to } n
\]
\[
\quad \text{if } A[j] \leq A[i] \text{ then } LIS[0][j] = 1
\]
\[
\text{for } i \leftarrow 1 \text{ to } n - 1
\]
\[
\quad \text{for } j \leftarrow i \text{ to } n - 1
\]
\[
\quad \text{if } A[i] \geq A[j] \text{ then } LIS[i, j] = LIS[i - 1, j]
\]
\[
\quad \text{else}
\]
\[
\quad \quad LIS[i, j] = \max\{LIS[i - 1, j], 1 + LIS[i - 1, i]\}
\]
\[
\text{return } LIS[n, n + 1]
\]

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:

\[
\text{Opt}(i, j) = \begin{cases} 
\alpha_{x[i], y[j]} + \text{Opt}(i - 1, j - 1) & \text{if } A[i] \neq A[j] \\
\delta + \text{Opt}(i - 1, j) & \text{if } A[i] = A[j]
\end{cases}
\]

Base cases: \(\text{Opt}(0, 0) = 0\) and \(\text{Opt}(i, 0) = \delta \cdot i\)

Pseudocode: Edit distance - DP

\[
\text{EDIST}(A[1..m], B[1..n]):
\]
\[
\text{for } i \leftarrow 1 \text{ to } m \text{ do } M[i, 0] = i \delta
\]
\[
\text{for } j \leftarrow 1 \text{ to } n \text{ do } M[0, j] = j \delta
\]
\[
\text{for } i \leftarrow 1 \text{ to } m \text{ do }
\]
\[
\quad \text{for } j \leftarrow 1 \text{ to } n \text{ do }
\]
\[
\quad M[i][j] = \min\{COST[A[i]][B[j]] + M[i - 1][j - 1],
\]
\[
\quad \quad \delta + M[i][j - 1],
\]
\[
\quad \quad \delta + M[i - 1][j]
\]

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 \(v_1, v_2, \ldots, v_k\) such that \(v_i, v_{i+1} \in E\) for \(1 \leq i < k - 1\). 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 \(v_1, v_2, \ldots, v_k\) 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 \(u\)**, \(\text{con}(u)\), 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 \(\text{rch}(u)\) 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 $u \prec v$.

**Directed acyclic graphs (dags)**

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

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**

**Explore**

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

**Pre and post numbering**

- Forward edge: $\text{pre}(u) < \text{pre}(v) < \text{post}(v) < \text{post}(u)$
- Backward edge: $\text{pre}(v) < \text{pre}(u) < \text{post}(u) < \text{post}(v)$
- Cross edge: $\text{pre}(u) < \text{post}(u) < \text{pre}(v) < \text{post}(v)$

**Strongly connected components**

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

**Strongly connected components**

**Metagraph**

Compute $\text{rev}(G)$ by brute force ordering $\leftarrow$ reverse post-ordering of $V$ in $\text{rev}(G)$ by DFS($\text{rev}(G)$, $s$) 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 strongly connected component $G(V, E) \leftarrow G - S_u$

**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 $u \prec v$.

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.

**Directed acyclic graphs**

Running time: $O(n^3)$

**DFS and BFS**

**Explore**

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

**Pre and post numbering**

- Forward edge: $\text{pre}(u) < \text{pre}(v) < \text{post}(v) < \text{post}(u)$
- Backward edge: $\text{pre}(v) < \text{pre}(u) < \text{post}(u) < \text{post}(v)$
- Cross edge: $\text{pre}(u) < \text{post}(u) < \text{pre}(v) < \text{post}(v)$

**Strongly connected components**

- Given $G$, $u$ is strongly connected to $v$ if $u \in \text{rch}(v)$ and $v \in \text{rch}(u)$.
- A maximal group of connected vertices that are all strongly connected to one another is called a strongly connected component.

**Strongly connected components**

**Metagraph**

Compute $\text{rev}(G)$ by brute force ordering $\leftarrow$ reverse post-ordering of $V$ in $\text{rev}(G)$ by DFS($\text{rev}(G)$, $s$) 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 strongly connected component $G(V, E) \leftarrow G - S_u$

**Strongly connected components**

**Metagraph**

Compute $\text{rev}(G)$ by brute force ordering $\leftarrow$ reverse post-ordering of $V$ in $\text{rev}(G)$ by DFS($\text{rev}(G)$, $s$) 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 strongly 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 weights.

**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:

**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:

**Pseudocode: Dijkstra**

```plaintext
for v \in V do
    d(v) \leftarrow \infty
X \leftarrow \emptyset

for i \leftarrow 1 to n do
    v \leftarrow arg min_{u \in V \setminus X} d(u)
    X \leftarrow X \cup \{v\}
    for u in Adj(v) do
        d(u) \leftarrow min(d(u), d(v) + \ell(u, v))

return d
```

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

**Pseudocode: Bellman-Ford**

```plaintext
for each v \in V do
    d(v) \leftarrow \infty

for k \leftarrow 1 to n - 1 do
    for each edge (u, v) \in E do
        d(v) \leftarrow min(d(v), d(u) + \ell(u, v))

return d
```

Running time: $O(nm)$

**Pseudocode: Floyd-Warshall**

```plaintext
for i \in V do
    for j \in V do
        for k \in V do
            d(i, j, k) \leftarrow min\left\{d(i, j, k - 1) + \ell(j, k), \begin{cases} d(i, j) & \text{if } i = j \land (i, j) \in E \land k = 0 \text{ and } \ell(i, j) = \infty \land (i, j) \notin E, \text{ or } i = j \land s \neq e \land \ell(s, e) = 0, \text{ or } s \neq e \land \ell(s, e) = 0 \end{cases} \right. \right.

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

**Pseudocode: Floyd-Warshall**

```plaintext
for i \in V do
    for j \in V do
        for k \in V do
            d(i, j, k) \leftarrow min\left\{d(i, j, k - 1), d(i, k, k - 1) + \ell(k, j, k - 1) \right\}

for v \in V do
    if d(i, j, n - 1) < 0 then
        return "exists negative cycle in G"

return d
```

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

**Pseudocode: Floyd-Warshall**

```plaintext
for i \in V do
    d(i) \leftarrow 0
    for k \in V do
        d(i) \leftarrow min\left\{d(i), d(k) + \ell(k, i) \right\}

return d
```

Running time: $O(mn)$