ECE 374 B: Algorithms and Models of Computation, Spring 2023
Midterm 2 – April 04, 2023

- **You will have 75 minutes (1.25 hours) to solve 5 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).
- 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) - 22 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 bound for the following recurrences :

(i)

$$A(n) = 2A\left(\frac{n}{2}\right) + n^3 \qquad A(0) = A(1) = 1$$

(ii)

$$B(n) = B(n-2) + n^2 \qquad B(0) = B(1) = 1$$

(b) We developed a new type of algorithm to sort a set of (non-numerical) elements. It sorts a set of size n elements by dividing the sort into nine sub-sorts of size $n/3$, recursively solving each sub-sort, and then combining the nine solutions in $O(n^2)$ time. What is the asymptotic running time of this algorithm?

2 Short answer II (4 questions) - 28 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) How many strongly connected components does a directed acyclic graph (DAG) have?

(b) In the Floyd-Warshall (found in the cheat sheet), we defined a recurrence $d(i, j, k)$. **Give an English description (no more than 2 sentences) of what $d(i, j, k)$ represents.**

Note: what's in the cheat sheet does not constitute a english description for the recurrence.

(c) Given n vertices, what is the minimum number of edges one would need to create a graph with exactly one topological sort.

(d) Your friend says he discovered a better way of calculating the shortest path in graphs with negative weight edges. All we need to do is find the minimum edge weight $w^* = \min\{w(u,v) | (u,v) \in E\}$ and add it to all the other edges in the graph $\hat{w} = w(u,v) - w^*$.

Now that the edges are all positive weight, you can use Djikstra and find the shortest path. **Does this method of re-weighting work?** Either prove the correctness of the method or provide a counter example (and briefly explain the counter example).

Circle one: Yes(re-weighting works) No (re-weighting does not work)

3 Finding a plurality - 15 points

Given an arbitrary array $A[1..n]$, describe an algorithm to determine in $O(n)$ time whether A contains more than $n/4$ copies of any value. **Do not use hashing, or radix sort, or any other method that depends on the precise input values.**

4 Dynamic programming - 15 points

A common subsequence of three strings X, Y, Z is a string that is a subsequence of each of X, Y, and Z. Describe a DP algorithm that returns the length of the longest common subsequence of $X[1..n]$, $Y[1..n]$, and $Z[1..n]$ by providing the following.

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

Memoization data structure and evaluation order:

Return value:

Time Complexity:

5 Graph algorithms (2 questions) - 20 points

For the graph problems, assume graphs are represented by adjacency lists that contain information about outgoing edges only – that is, for each vertex u in the graph, you know $\text{Out}(u)$, which stores outgoing edges from vertex u.

Assume you had a directed acyclic graph with one edge marked as **important**. A *important path* is a path that contains this one important edge.

Assume all the edges have the same weight $\ell(e) = 1$.

(a) Describe an algorithm that finds the shortest *important* path (**not just path length**) from s to t.

(continued from previous page)

(b) Describe an algorithm that finds all the vertices that can reach t using an *important* path.

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*

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.

Tower of Hanoi

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: $\sum_{k=0}^{n} ar^k = a\frac{1-r^{n+1}}{1-r}$
- Logarithmic identities: $\log(ab) = \log a + \log b$, $\log(a/b) = \log a - \log b$, $a^{\log_c b} = b^{\log_c a}$ $(a, b, c > 1)$, $\log_a b = \log_c b / \log_c a$.

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:

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	$O(n \log n)$	$O(n \log n)$ $O(n)$ (if optimized)
Quicksort	$O(n^2)$ $O(n \log n)$ if using MoM	$O(n)$

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:

Karatsuba's algorithm

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

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:

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

Pseudocode: LIS - DP

```
LIS-Iterative($A[1..n]$):
    $A[n+1] = \infty$
    for $j \leftarrow 0$ to $n$
        if A[i] $\leq$ A[j] then $LIS[0][j] = 1$

    for $i \leftarrow 1$ to $n-1$ do
        for $j \leftarrow i$ to $n-1$ do
            if $A[i] \geq A[j]$
                $LIS[i,j] = LIS[i-1,j]$
            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 $i \leftarrow 1$ to $m$ do $M[i,0] = i\delta$
    for $j \leftarrow 1$ to $n$ do $M[0,j] = j\delta$

    for $i = 1$ to $m$ do
        for $j = 1$ to $n$ do
            $M[i][j] = \min \begin{cases} COST[A[i]][B[j]] + M[i-1][j-1], \\ \delta + M[i-1][j], \\ \delta + M[i][j-1] \end{cases}$
```

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 $v_1, v_2, \ldots, v_k$ 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 $v_1, v_2, \ldots, v_k$ 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, 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←empty list
for $v \in V$:
    in$(v)$ ← $|\{u \mid u \to v \in E\}|$
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$(v)$ ← 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$[i]$ ← False
    Add $u$ to ToExplore and to $S$
    Visited$[u]$ ← 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$[y]$ ← 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.

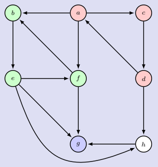
Pre/post numbering

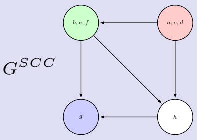
Pre and post numbering aids in analyzing the graph structure. By looking at the numbering we can tell if a edge (u, v) is a:

- *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.





Pseudocode: Metagraph - linear time

```
Metagraph($G(V, E)$):
    Compute rev$(G)$ by brute force
    ordering ← 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
            $S_u$ ← 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 "∃ negative cycle in $G$"

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

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