ECE 374 B: Algorithms and Models of Computation, Spring 2022

Midterm 2 – November 01, 2022

- **You will have 75 minutes (1.25 hours) to solve 4 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 bring (sharpened) ***pencils and an eraser*** to take your exam with, unless you are *absolutely sure* you will not need to erase. We will *not* provide any additional scratch paper if you write in pen and make mistakes, nor will we provide pencils and erasers.
- 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 (4 questions) - 26 points

Answer the following questions. Briefly justify your answers, but a complete proof is not required.

(a) **[6 points]** What is the value of the following summation?

$$\sum_{k=0}^{n-1} 2^k = ?$$

(b) **[6 points]** I formulated the solution to a particular question using the following recurrence:

$$f(x,y) = f(x,y-1) + f(x-1,y-1)$$
$$\textbf{Base:} \qquad f(x,1) = 1 \qquad f(1,y) = 1$$

Using memoization, what is the optimal runtime of this algorithm?

(c) **[7 points]** I'd like to use the median-of-medians (MoM) algorithm but I don't want to write a function that finds the median value in a list of 5 values. Instead I break the input area into lists of 3 values, and choose the median of medians pivot that way. Evaluate the running time of this resulting algorithm. Does the running time increase or decrease relative to the original running time of MoM?

Hint: the original MoM can be found in the cheatsheet

(d) **[7 points]** For any number of vertices n, describe a graph with n vertices that has the maximum number of topological sorts. Recall that a topological sort for a directed acyclic graph G is an ordering of the vertices of G such that for every edge $a \to b \in G$, a comes before b in the sequence.

2 Recursion - 20 points

I need to use the Depth-First-Search (DFS) algorithm but I have no standard library files that implement stacks/queues. Instead of writing my own stack structure, I decide it'll be easier to use the system stack to implement DFS. In other words, I'll implement a recursive algorithm. Write the recursive version of Depth-First-Search.

3 DP problem - 28 points

(a) **Largest Subsequence Product [12 Points]** You are given as input an array of integers $A[1..n]$. For a given subsequence A' of size m, the *subsequence product* is $\prod_{j \in 1..m} A'[j]$. Give a dynamic programming that finds the largest subsequence product.

For example, given $A = [-1, -2, -1, 3]$, the largest product of a subsequence is 6.

Hint: linear time is possible.

(b) **Outputting the subsequence. [4 points]** Describe how to modify your algorithm to output a subsequence that has the largest product. For example, given A above, you could output $A' = [2, -1, 3]$. Analyze the runtime of the resulting algorithm.

Hint: Your answer must describe what additional information must stored in the memo table. It must also give iterative or recursive pseudocode to output the result from the table.

(c) **[Largest Subsequence Dot Product [12 points]** You are given a pair of same size input arrays of integers, $A[1..n], B[1..n]$. Given a pair of subsequences A', B' of the same size m, the *subsequence dot product* is $\sum_{j \in 1..m} A'[j] * B'[j]$.

For example, given $A = [-1, 2, -1], B = [2, -1, 1]$, the largest dot product is 5.

Give a dynamic programming solution to find the largest subsequence dot product.

Hint: quadratic time is possible.

4 Graph algorithms - 26 points

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

(a) **[12 points]** Derive an efficient algorithm to find all the source vertices in a directed graph and give its time complexity. Recall that a source vertex has no incoming edges.

(b) **[14 points]** Give an efficient algorithm that determines if a particular weighted directed graph has a negative cycle and give its time complexity. (Hint: Remember we're asking "if a negative cycle *exists*", not "if a negative cycle exists at ir reachable from vertex s")

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

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

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 ← 0, 5, . . . , len(A)]
  medians = [sorted(sublist)[len(sublist)/2]
          for sublist ∈ sublists]

  // Base case
  if len(A) ≤ 5 return sorted(a)[i]

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

  // Partitioning step
  low = [j for j ∈ A if j < pivot]
  high = [j for j ∈ 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] = ∞
  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 { COST[A[i]][B[j]] + M[i−1][j−1],
                      δ + M[i−1][j],
                      δ + M[i][j−1] }
```

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, $\mathrm{con}(u)$, 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 algoritm

```
TP-sort-list = []
Count in-degree for each vertex
while $v \in V$ that has $in(v) = 0$:
  Add v to TP-sort-list
  Remove $v$ from $V$
  for $v$ in $(u, v) \in E$:
    $in(v) \leftarrow in(v) - 1$
return TP-sort-list
```

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($G$,$u$):
  array Visited$[1..n]$
  Initialize: Visited$[i] \leftarrow$ False for $i = 1, \ldots, n$
  Bag data structure: B
  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
        Add $y$ to $S$
        Add $y$ to $T$ with $x$ as its 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:

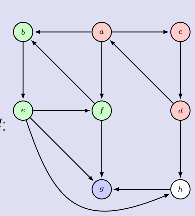
- *Forward edge*: pre(u) < pre(v) < post(v) < post(u)
- *Backward edge*: pre(v) < pre(u) < post(u) < post(v)
- *Cross edge*: pre(u) < post(u) < pre(v) < post(v)

Strongly connected components

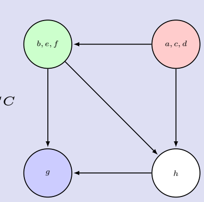
- Given a directed graph G, u is *strongly connected* to v if $v \in$ rch(u) *and* $u \in$ rch(v).
- A group of vertices that are all strongly connected to one-another is called a strongly connected component.

Example of Meta-graph:

G:



G^{SCC}



Pseudocode: Meta-Graph - Linear Time

```
do $(G^{rev})$ and output vertices in decreasing post order.
Mark all nodes as unvisited
for each $u$ in the computed order do
  if $u$ is not visited then
    $(u)$
    Let $S_u$ be the nodes reached by $u$
    Output $S_u$ as a strong connected component
    Remove $S_u$ from $G$
```

Shortest paths

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

Pseudocode: Dijkstra

```
Initialize for each node v, $d(s, v) = \infty$
Initialize $X = \varnothing$, $d(s, s) = 0$
for $i = 1$ to $n$ do
  Let $v$ be such that $d(s, v) = \min_{u \in V - X} d(s, u)$
  $X = X \cup \{v\}$
  for each $u$ in Adj$(v)$ do
    $d(s, u) = \min\{d(s, u),\ d(s, v) + \ell(v, u)\}$
```

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:

$$d(v, k) = \min \begin{cases} \min_{u \in V}(d(u, k-1) + \ell(u, v)). \\ d(v, k-1) \end{cases}$$

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

Pseudocode: Bellman-Ford

```
for each $u \in V$ do
  $d(u) \leftarrow \infty$
$d(s) \leftarrow 0$

for $k = 1$ to $n - 1$ do
  for each $v \in n$ do
    for each edge $(u, v) \in in(v)$ do
      $d(v) = \min\{d(v), d(u) + c(u, v)\}$

for each $v \in V$ do
    $Dist(s, v) \leftarrow d(v)$
```

Running time: $O(nm)$

Floyd-Warshall algorithm:
Find minimum distance between any two vertices i and j. It is a DP algorithm with the following recurrence:

$$d(i, j, k) = \min \begin{cases} d(i, j, k-1) \\ d(i, k, k-1) + d(k, j, k-1) \end{cases}$$

Base cases: $dist(i, j, 0) = \ell(i, j)$ if $(i, j) \in E$, otherwise ∞

Pseudocode: Floyd-Warshall

```
for $i \leftarrow 1$ to $n$ do
  for $j \leftarrow 1$ to $n$ do
    $d(i, j, 0) \leftarrow \ell(i, j)$
 (* $\ell(i, j) \leftarrow \infty$ if $(i, j) \notin E$, 0 if $i = j$ *)

for $k \leftarrow 1$ to $n$ do
  for $i \leftarrow 1$ to $n$ do
    for $j \leftarrow 1$ to $n$ 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 $i \leftarrow 1$ to $n$ do
  if $d(i, i, n) < 0$ then
    Output "$\exists$ negative cycle in $G$"
```

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