ECE 374 B: Algorithms and Models of Computation, Fall 2024
Midterm 2 – November 5th, 2024

- **You will have 75 minutes (1.25 hours) to solve all the 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! Make sure to check both sides of all the pages and make sure you answered everything. **Time is a factor!** Budget yours wisely.
- *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.
- 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: _____

1 Short answer - 18 points

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

(a) For each of the following recurrences, do the following:

- Provide a **tight asymptotic upper bound**.
- **No partial credit. Draw a square around your final answer.**

(i)

$$A(n) = A(n/2) + A(n/3) + A(n/4) + n^2$$

(ii)

$$B(n) = 2B(n/4) + \sqrt{n}$$

(iii)

$$C(n,m) = C(n/2, m/3) + O(nm)$$

(b) Consider two numbers x and y, where B is the base, and x_1, x_0, y_1, y_0 are integers. How does Karatsuba's algorithm compute xy using three multiplications instead of 4?

Hint: Remember Karatsuba's algorithm broke n-digit integers x and y into two $m = n/2$ digit numbers: $x = x_1 B^m + x_0$ and $y = y_1 B^m + y_0$

Note: This is a short answer (no more than two sentences). Equations are allowed but everything needs to be concise.

2 Short answer II - 12 points

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

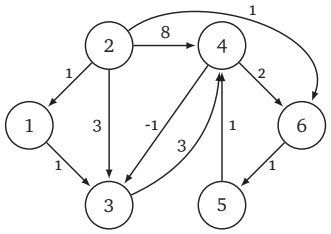
(a) You are given two character sequences $A[0 \ldots n-1]$, $B[0 \ldots m-1]$. **What are the *minimum and maximum* alignment possible between these two sequences**? Assume mismatch cost (α) and insertion/deletion (δ) costs are both equal to 1.

(b) Recall in lecture/discussion we discussed the median of median (linear time selection) algorithm. The algorithm we discussed breaks a array into lists of size five. What if we break the array into lists of size 15. **What is the recurrence and asymptotic running time** for the this modified version of linear time selection

3 Short answer III - 15 points

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

(a) Consider the following graph:



We call the Floys-Warshall algorithm on this graph and fill out the three dimensional $d(i, j, k)$ matrix.

What is the value of $d(2, 4, 3)$?

(b) You have a directed graph $G = (V, E)$ with all positive edge weights. **Describe an algorithm that finds the shortest path between all pairs of vertices.** You can use any of the cheat sheet algorithms as a black box.

4 Dynamic Programming - 10 points

In class we discussed the longest increasing subsequence problem but just to recap: we are given a sequence of n integers and the goal is to find the longest increasing subsequence. We also know that we can find the length of the longest increasing subsequence in polynomial time. But how do we find the actual LIS (the actual values that make up the LIS).

Basically if the input is: $[6,3,5,2,7,8,1]$, the output should be: $[3,5,7,8]$ (The actual subsequence). **Provide an algorithm (or modify the existing LIS algorithm) that returns the longest increasing subsequence values.** I included the LIS code from lectures/labs below so you could save some time and only include the modifications to the original algorithm.

```
LIS-Iterative(A[1..n]):
  A[n+1] = ∞

  int LIS[0..n-1, 0..n]

  for j = 0...n) if A[i] ≤ A[j] then LIS[0][j] = 1

  for i = 1...n-1 do
    for j = i...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]
```

5 Dynamic programming - 15 points

Assume you have a chain that is n links long that you need to sell. However, the value of the chain is not linearly proportional with the number of links. You are given an array $A[1 \ldots n]$ where $A[i]$ stores the price of a chain with n links (you can also assume $A[0] = 0$. no links = no chain = no value).

You look at the prices and realize that multiple smaller chains would be more valuable than the n-link chain you have right now. But you also know that dividing the chain requires you cut (and destroy) a link meaning that every division reduces the total number of links.

Show a dynamic programming algorithm that finds the maximum value you can obtain from a n-link chain assuming you sub-divide it correctly.

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

Memoization data structure and evaluation order:

Return value:

Time Complexity:

6 Graphing Algorithm I - 15 points

In the traveling salesman problem, we are trying to find the path of smallest length that visits every vertex exactly once. For a normal graph this is a very difficult problem but for simple cases, an efficient solution is possible.

Suppose you had a directed acyclic graph (DAG) $G = (V, E)$ with all positive edge weights. Describe a efficient algorithm **that returns the value of** the shortest path that visits every vertex in the graph exactly once. Note that not every DAG has a path that visits every vertex once and so if there is no such path, your algorithm should return -1. The beginning and ending vertices can be any vertices in G.

7 Graphing Algorithm II - 15 points

You are give a directed graph $G = (V, E)$ where every edge weight can be positive or negative and is marked as red or black. A red-black path is a path in the graph where edges alternate between red and black and can start with either a black or red edge.

Give an algorithm that gives you the shortest possible red-black path between vertices s and t.

EXTRA CREDIT (1 pt)

Name the instructors of this course (need both for credit):

Instructor 1 name:

Instructor 2 name:

EXTRA CREDIT (2 pts)

Name a TA and the discussion section time he covers.

TA name:

Discussion section time:

This page is for additional scratch work!

ECE 374 B Algorithms: Cheatsheet

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.

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)
```

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 $O(n^{\log_2 3}) = O(n^{1.585})$.

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
```

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 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.

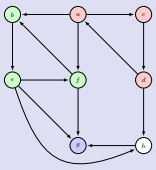
Pre/post numbering

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:

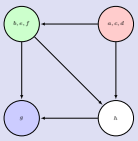
- *Forward edge*: pre(u) < pre(v) < post(v) < post(u)
- *Backward edge*: pre(v) < pre(u) < post(u) < post(v)
- *Cross edge*: pre(u) < post(u) < pre(v) < post(v)

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: $\Theta(n^3)$