Practice Final Examination

SCIS 313: Data Structures and Algorithm Analysis

Prof. Antonio Khalil Moretti Spelman College

Name: _____

Student ID: _____

Instructions: Show all work for full credit. For code analysis, set up and evaluate all sums explicitly using the provided formulas. For recurrences, identify a, b, d; compute b^d; state the case; give the bound. Calculators are not permitted.

Problem	Topic	Points
1	Code Analysis & Recurrences (parts a–b)	20
2	Heaps & Hash Tables (parts a–b)	20
3	BFS / DFS Trace and Analysis	25
4	Greedy Algorithms — Interval Scheduling	20
5	Dijkstra's Algorithm (Guided)	15
	Total	**100**
6	*Extra Credit: Dynamic Programming*	*10*

Reference: Three-Case Recurrence Theorem. For $T(n) = aT(n/b) + O(n^d)$ with $a \geq 1$, $b > 1$, $d \geq 0$:

$$T(n) = \begin{cases} O(n^d \log n) & \text{if } a = b^d \\ O(n^d) & \text{if } a < b^d \\ O(n^{\log_b a}) & \text{if } a > b^d \end{cases}$$

Useful formulas.

$$\sum_{i=1}^{n} i = \frac{n(n+1)}{2} \qquad \sum_{i=1}^{n} i^2 = \frac{n(n+1)(2n+1)}{6}$$

$$\sum_{i=0}^{\infty} r^i = \frac{1}{1-r} \text{ for } |r| < 1 \qquad \sum_{i=0}^{k-1} r^i = \frac{1-r^k}{1-r} \text{ for } r \neq 1$$

Heap indexing (1-indexed): $\text{parent}(i) = \lfloor i/2 \rfloor$, $\text{left}(i) = 2i$, $\text{right}(i) = 2i + 1$.

Problem 1

20 points

Code Analysis and Recurrences.

Part (a)

10 points

Iterative Code Analysis.

Give a tight Θ bound on the number of times `ops++` executes. Show all work: write the exact sum, evaluate it using the provided formulas, and simplify.

```
int ops = 0;
for (int i = 1; i <= n; i++) {
    for (int j = i; j <= n; j++) {
        ops++;
    }
}
```

Part (b)

10 points

Recurrence Relations.

The **fast exponentiation** algorithm computes $\texttt{base}^n$ for a non-negative integer n much more efficiently than the naïve approach of multiplying `base` by itself n times. The key idea is that when n is even, $\texttt{base}^n = (\texttt{base}^{n/2})^2$, so only **one** recursive call of half the size is needed, plus a single multiplication to square the result.

```
// Compute base^exp for non-negative integer exp
long fastPow(long base, int exp) {
    if (exp == 0) return 1;
    if (exp % 2 == 0) {
        long half = fastPow(base, exp / 2);
        return half * half;          // one recursive call, O(1) work
    } else {
        return base * fastPow(base, exp - 1);  // one recursive call, O
            (1) work
    }
}
```

(i) Focus on the **even** case, which dominates when $n = \texttt{exp}$ is a power of 2. How many recursive calls are made? What is the size of each subproblem? What is the cost of the work done outside the recursive call? Write the recurrence $T(n)$ for the running time of `fastPow` on an exponent of size n.

(ii) Apply the Three-Case Recurrence Theorem. Identify a, b, d; compute b^d; state which case applies and why; give the Θ bound.

(iii) A naïve iterative approach computes $\texttt{base}^n$ by multiplying `base` by itself n times, running in $\Theta(n)$. How does `fastPow` compare? This algorithm is used in cryptographic systems such as RSA, where one must compute $m^e \bmod N$ for exponents e that are hundreds of digits long. Explain in 1–2 sentences why the naïve approach would be completely impractical in that context.

Problem 2

20 points

Heaps and Hash Tables.

Part (a)

10 points

Heaps.

Consider the following array (1-indexed): $A = [\,10,\ 7,\ 8,\ 5,\ 3,\ 12,\ 6\,]$.

(i) Is A a valid max-heap? Using the index formulas on the reference sheet, identify every heap-property violation by stating which parent–child pair is out of order. Then perform the minimum number of swaps to fix it, showing the array after each swap.

(ii) Starting from your corrected max-heap, insert the value 11. Show the array after the insertion and after each sift-up swap until the heap property is restored. Use the index formulas to identify which parent to compare against at each step.

(iii) Perform `extract-max` on the heap from part (ii). Show the array after the removal and after each sift-down swap until the heap property is restored. [526 lines total]

(iv) `build-heap` runs in $\Theta(n)$ even though a heap has $\Theta(\log n)$ levels. Explain in 2–3 sentences why this is the case. Then state the runtime of building a heap by inserting elements one at a time and explain why it differs.

Part (b)

10 points

Hash Tables.

Use the hash function $h(k) = k \bmod 7$ and insert the keys 22, 35, 14, 9, 16, 28, 3 in that order into a table of size 7.

(i) Draw the hash table using **separate chaining**. For each key show which slot it hashes to and draw the resulting chains.

(ii) Draw the hash table using **open addressing with linear probing**. For each key show the initial hash slot and, if occupied, the full probe sequence until an empty slot is found.

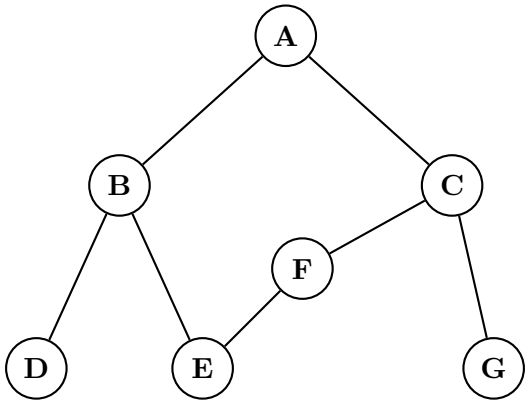
(iii) After all insertions, what is the load factor α? In chaining, what does α represent about expected lookup time? In open addressing, why does performance degrade much more sharply as $\alpha \to 1$?

Problem 3

25 points

BFS / DFS Trace and Analysis.

Consider the following undirected graph G:



(a) **(6 pts)** Run BFS from vertex A, processing neighbors in alphabetical order. List the vertices in the order they are discovered. State the shortest-path distance $d(A, v)$ for every vertex v.

(b) **(5 pts)** Draw the BFS tree produced by your traversal. Label all tree edges.

(c) **(8 pts)** Run DFS from vertex A, processing neighbors in alphabetical order. Record the discovery time $d[v]$ and finish time $f[v]$ for each vertex, using a counter starting at 1 that increments each time a vertex is discovered or finished.

(d) **(3 pts)** Using your DFS result, classify each edge of G as a *tree edge* or a *back edge*. Is there a cycle in G? If so, identify it.

(e) **(3 pts)** The graph has V vertices and E edges stored as an adjacency list. State the runtime of BFS and of DFS. Justify by explaining how many times each vertex and each edge is processed — do not simply state the answer.

Problem 4

20 points

Greedy Algorithms: Interval Scheduling.

You are given a set of activities, each with a start time and a finish time. You want to select the *maximum* number of non-overlapping activities (two activities overlap if one starts strictly before the other finishes). The greedy strategy is: *always select the activity with the earliest finish time that does not conflict with the previously selected activity.*

Activity	A	B	C	D	E	F	G
Start	1	3	0	5	3	5	6
Finish	4	5	6	7	8	9	10

(a) **(3 pts)** Sort the activities by finish time. List them in sorted order. Break ties alphabetically.

(b) **(5 pts)** Apply the greedy algorithm step by step. At each step state which activity you are considering, whether you select or reject it, and why. List the final selected set.

(c) **(3 pts)** A classmate proposes the alternative set $\{C, D, G\}$. Compare its size to the greedy solution. What does this tell you about the quality of the greedy solution?

(d) **(6 pts)** We want to argue the greedy choice is always safe. Complete the exchange argument below.

Let g be the activity with the earliest finish time, and let OPT be any optimal solution that does not include g. OPT must contain some first activity x with $f(x) \geq f(g)$. Define OPT$'$ by replacing x with g in OPT.

(i) Why is OPT$'$ still a valid non-overlapping schedule? Your answer should explicitly reference $f(g)$ and $f(x)$.

(ii) Why does $|\text{OPT}'| = |\text{OPT}|$?

(iii) What do (i) and (ii) together tell you about the greedy choice?

(e) **(3 pts)** State the total runtime of the interval scheduling algorithm including the initial sort. Does the sort or the selection step dominate? Justify.

Problem 5

15 points

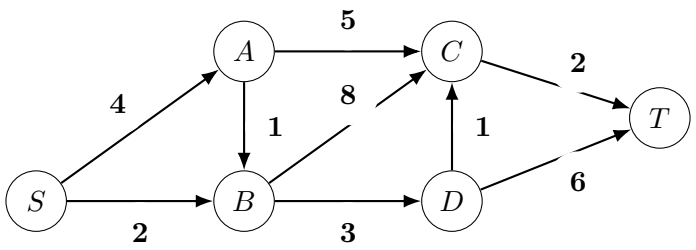
Dijkstra's Algorithm (Guided).

Dijkstra's algorithm finds shortest paths from a source vertex s to all other vertices in a weighted graph with non-negative edge weights. The algorithm is described below for reference.

Dijkstra(G, s):

1. Set $\text{dist}[s] = 0$; $\text{dist}[v] = \infty$ for all $v \neq s$.
2. Insert all vertices into a priority queue Q keyed by dist.
3. While Q is not empty:
   a. Extract the vertex u with minimum $\text{dist}[u]$ from Q.
   b. For each neighbor v of u with edge weight $w(u, v)$:
   if $\text{dist}[u] + w(u, v) < \text{dist}[v]$, update $\text{dist}[v] \leftarrow \text{dist}[u] + w(u, v)$.

Consider the following weighted directed graph with source S:



(a) **(7 pts)** Run Dijkstra's algorithm from source S. Fill in the table showing the dist value for each vertex after each extraction from Q. Write ∞ for unvisited vertices. The first row is provided.

Step (vertex extracted)	dist[S]	dist[A]	dist[B]	dist[C]	dist[D]	dist[T]
Initial (none extracted)	0	∞	∞	∞	∞	∞
Extract S						
Extract B						
Extract A						
Extract D						
Extract C						
Extract T						

(b) **(3 pts)** What is the shortest-path distance from S to T? List the edges on the shortest path.

(c) **(3 pts)** Dijkstra's algorithm is considered a greedy algorithm. What is the "greedy choice" made at each iteration? Why does a vertex never need to be re-extracted once it has been extracted? (*Hint: what property of the edge weights makes this safe?*)

(d) **(2 pts)** If the priority queue Q is implemented as a **binary min-heap**, each `extract-min` costs $O(\log V)$ and each distance update costs $O(\log V)$. Given that `extract-min` is called once per vertex and a distance update can happen once per edge, write the total runtime of Dijkstra's algorithm in terms of V and E.

Problem 6

10 points extra credit

Dynamic Programming: Coin Change.

You have an unlimited supply of coins with denominations $\{1, 4, 6\}$. Given a target amount n, you want to find the *minimum* number of coins that sum to n.

Define $C(k)$ = minimum number of coins needed to make amount k, with $C(0) = 0$.

(a) **(3 pts)** Write the recurrence for $C(k)$. Define your subproblem clearly and state the base case(s).

(b) **(4 pts)** Fill in the table below for $n = 10$. Show your work for at least three entries by writing out which coin choices you considered and which gave the minimum.

k	0	1	2	3	4	5	6	7	8	9	10
$C(k)$	0										

(c) **(3 pts)** A greedy approach — always pick the largest denomination that fits — gives $6+1+1+1+1=5$ coins for $n = 10$. Does your DP solution do better? Explain why greedy fails here but worked for interval scheduling.