Additional Practice Problems

SCIS 313: Data Structures and Algorithm Analysis

Prof. Antonio Khalil Moretti Spelman College

These problems are not graded. Solutions are provided separately.

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.

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

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

$$\sum_{i=1}^{n} \frac{1}{i} \approx \ln n = \Theta(\log n)$$

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

Problem 1 Code Analysis and Recurrences

Part (a)

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

```
int ops = 0;
for (int i = 1; i <= n; i *= 2) {
    for (int j = 0; j < i; j++) {
        ops++;
    }
}
```

Part (b)

Give a tight Θ bound on the number of times `ops++` executes. Show all work.

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

Part (c)

Below is a recursive sorting algorithm. Answer all parts.

```
void mergeSort3(int arr[], int left, int right) {
    if (right - left < 1) return;

    int third = (right - left) / 3;
    int mid1  = left + third;
    int mid2  = left + 2 * third;

    mergeSort3(arr, left,  mid1);   // sort first third
    mergeSort3(arr, mid1 + 1, mid2);  // sort second third
    mergeSort3(arr, mid2 + 1, right); // sort third third

    merge3(arr, left, mid1, mid2, right); // merge three sorted parts:
        O(n)
}
```

(i) How many recursive calls does `mergeSort3` make? What is the size of each subproblem? What is the cost of the work done outside the recursive calls? Write the recurrence $T(n)$ for the running time of `mergeSort3`.

(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) Standard merge sort splits into **2** parts and runs in $\Theta(n \log n)$. This version splits into **3** parts and also runs in $\Theta(n \log n)$. Does splitting into more parts make merge sort faster? Explain why or why not, and state what would need to change to actually improve the asymptotic bound.

Problem 2 Heaps and Hash Tables

Part (a) — Min-Heaps

Consider the following array (1-indexed): $A = [3,\ 9,\ 5,\ 12,\ 11,\ 8,\ 7]$.

(i) Is A a valid **min-heap**? Recall that in a min-heap every parent must be *less than or equal to* both of its children. Using the index formulas on the reference sheet, identify every violation by stating which parent–child pair is out of order. Perform the minimum number of swaps to fix it, showing the array after each swap.

(ii) Starting from your corrected min-heap, insert the value 2. Show the array after the insertion and after each sift-up swap until the heap property is restored.

(iii) Perform `extract-min` on the heap from part (ii). Show the array after the removal and after each sift-down swap until the heap property is restored.

(iv) You are given n numbers and want to find the k smallest ones. Describe a strategy using a min-heap. What is the runtime? Is there a way to do it with a **max-heap** of fixed size k that is more efficient when $k \ll n$? Explain.

Part (b) — Hash Tables

Use the hash function $h(k) = k \bmod 7$ and insert the keys 5, 19, 26, 12, 33, 7, 40 in that order into a table of size 7.

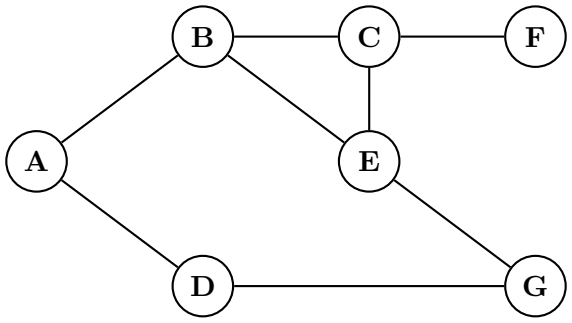
(i) Draw the hash table using **separate chaining**. For each key show which slot it hashes to and draw the resulting chains.

(ii) Draw the hash table using **open addressing with linear probing**. For each key show the initial hash slot and the full probe sequence if needed.

(iii) Two of the keys hash to the same slot under $h(k) = k \bmod 7$. Identify them. Now suppose you change the table size to 11 and use $h(k) = k \bmod 11$. Do those same two keys still collide? What does this suggest about choosing a table size?

Problem 3 BFS and DFS

Consider the following undirected graph G:



Part (a) — Breadth-First Search

Run BFS from vertex A, processing neighbors in alphabetical order.

(i) Fill in the BFS queue table below. At each step write the vertex dequeued, any new vertices enqueued, and the queue contents after the step. *Note: the table has enough rows for the complete traversal — use this as a guide for how many steps to expect.*

Step	Dequeue	Enqueue	Queue after	Notes
0	—	A	$[A]$	Source, $d(A) = 0$
1				
2				
3				
4				
5				
6				
7				

(ii) State the shortest-path distance $d(A, v)$ for every vertex v.

(iii) Draw the BFS tree. Label all tree edges.

Part (b) — Depth-First Search

Run DFS from vertex A, processing neighbors in alphabetical order. The counter starts at 1 and increments each time a vertex is discovered or finished.

(i) Fill in the DFS stack table below. *Note: the table has enough rows for the complete traversal — use this as a guide for how many steps to expect.*

Counter	Event	Vertex	Push / Pop	Stack after
1	Discover	A	Push A	$[A]$
2				
3				
4				
5				
6				
7				
8				
9				
10				
11				
12				
13				
14				

(ii) Fill in the discovery and finish times.

Vertex	A	B	C	D	E	F	G
$d[v]$							
$f[v]$							

(iii) Classify each edge of G as a tree edge or a back edge. Is there a cycle in G? If so, identify it.

Problem 4 Greedy Algorithms: Fractional Knapsack

You have a knapsack with capacity $W = 15$ kg. There are 5 items available, each with a weight and a value. In the **fractional knapsack** problem you may take any fraction of an item — if you take a fraction $f \in [0, 1]$ of item i you gain value $f \cdot v_i$ and use weight $f \cdot w_i$. The goal is to maximize the total value of items in the knapsack.

Item	A	B	C	D	E
Weight (kg)	4	6	3	5	7
Value ($)	20	18	12	15	14

(a) Compute the value-to-weight ratio v_i/w_i for each item. Sort the items from highest to lowest ratio.

(b) Apply the greedy algorithm: take as much as possible of the highest-ratio item, then the next highest, and so on until the knapsack is full. Show each step, stating how much of each item you take and the running total weight and value.

(c) What is the total value of the greedy solution? Is it possible to do better? Justify briefly.

(d) Now consider the **0/1 knapsack** problem, where you must take each item either entirely or not at all (no fractions). Apply the same greedy strategy (sort by ratio, take greedily). Find

a selection of whole items that fits in the knapsack and has strictly greater value than the greedy solution, or explain why the greedy solution remains optimal.

(e) What does part (d) tell you about using a greedy algorithm for 0/1 knapsack? Why does the greedy approach work for the fractional version but not necessarily for the 0/1 version?

Problem 5 Dijkstra's Algorithm

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

Dijkstra(G, s):

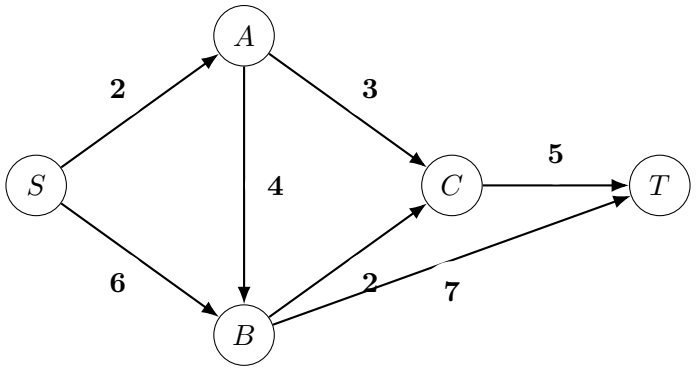
1. Set $\text{dist}[s] = 0$; $\text{dist}[v] = \infty$ for all $v \neq s$.
2. Insert all vertices into a priority queue Q keyed by dist.
3. While Q is not empty:
   a. Extract the vertex u with minimum $\text{dist}[u]$ from Q.
   b. For each neighbor v of u with edge weight $w(u, v)$:
   if $\text{dist}[u]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

How to fill in the distance table. Each row of the table corresponds to one *extraction* from the priority queue. Here is what to do at each step:

1. **Choose which vertex to extract.** Look at all vertices not yet extracted. The one with the smallest current dist value is extracted next. (The row label tells you which vertex this is on the exam.)
2. **Relax all outgoing edges from that vertex.** For each edge $u \to v$ with weight w, compute the *candidate distance* $\text{dist}[u] + w$. If this is strictly less than the current $\text{dist}[v]$, update $\text{dist}[v]$ to the new smaller value. This is called *relaxing* the edge.
3. **Carry forward all other values.** Every cell that was not updated keeps its previous value. Every row must be fully filled in.
4. **A vertex's distance is final once extracted.** Because all edge weights are non-negative, no future path can improve on a distance that has already been extracted. You never go back and change an extracted vertex's distance.

Key terms: *Extraction* means pulling the minimum-distance vertex from the priority queue and processing its edges. *Relaxation* means checking whether a newly discovered path to a neighbor v is shorter than the best path found so far, and updating if so.

Consider the following weighted directed graph with source S:



(a) Run Dijkstra's algorithm from source S. Fill in the distance table, showing dist values after each extraction. Write ∞ for vertices not yet reached. The first row is provided.

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

(b) What is the shortest-path distance from S to T? List the vertices and edges on the shortest path.

(c) For each vertex, trace back which vertex caused its final distance update (its "parent" in the shortest path tree). Draw the shortest path tree.

(d) Dijkstra's algorithm is greedy — it always extracts the vertex with the smallest current distance. Explain in 2–3 sentences why a vertex's distance is guaranteed to be finalized the moment it is extracted from the priority queue, assuming all edge weights are non-negative.

Note: Memoization and Dynamic Programming

What is memoization?

Many problems have a natural recursive structure: to solve a problem of size n, you solve smaller subproblems and combine their results. The difficulty is that a naive recursive solution often solves the *same subproblem many times*. For example, computing the LCS of two strings of length 5 naively requires solving overlapping subproblems exponentially many times.

Memoization is the technique of storing the result of each subproblem the first time it is solved, so that subsequent requests for the same subproblem are answered in $O(1)$ by a table lookup rather than recomputed from scratch.

Dynamic programming is the systematic, bottom-up version of memoization: instead of recursing top-down and caching results, you fill in a table of subproblem answers from smallest to largest, ensuring every subproblem is solved exactly once before it is needed.

What memoization achieves. Both approaches — memoization and DP — reduce an exponential-time recursive algorithm to polynomial time by ensuring each distinct subproblem is solved only once. The savings can be dramatic: naive recursive LCS on strings of length m and n costs $O(2^{m+n})$; the DP table version costs $O(mn)$.

How to recognize a DP problem. Ask two questions:

1. **Optimal substructure:** can the optimal solution to the whole problem be built from optimal solutions to smaller subproblems?
2. **Overlapping subproblems:** does a naive recursive solution solve the same subproblems repeatedly?

If both answers are yes, DP (or memoization) will likely give an efficient solution.

Connection to the extra credit problems. Both coin change and LCS satisfy these two conditions. For LCS, the subproblem is $L[i][j]$ = length of the LCS of the first i characters of X and the first j characters of Y. Each cell depends only on previously computed cells, so filling the table left-to-right, top-to-bottom solves every subproblem exactly once in $O(mn)$ total time.