Additional Practice Problems — Solutions

SCIS 313: Data Structures and Algorithm Analysis

Prof. Antonio Khalil Moretti Spelman College

Problem 1 — Code Analysis and Recurrences

Part (a)

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

The outer loop runs for $i = 1, 2, 4, 8, \ldots$ until $i > n$, so $k = \lfloor \log_2 n \rfloor + 1$ iterations. For each value of $i = 2^m$, the inner loop runs 2^m times. Total:

$$\sum_{m=0}^{\lfloor \log_2 n \rfloor} 2^m = 2^{\lfloor \log_2 n \rfloor + 1} - 1 = \Theta(n).$$

Answer: $\Theta(n)$.

Intuition: The inner loop does $1 + 2 + 4 + \cdots + n/2 + n$ work in the geometric series, which sums to $2n - 1$.

Part (b)

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

For a fixed i, the inner loop runs $j = 1, 1 + i, 1 + 2i, \ldots$ until $j > n$, executing $\lceil n/i \rceil \approx n/i$ times. Summing over all i:

$$\sum_{i=1}^{n} \frac{n}{i} = n \sum_{i=1}^{n} \frac{1}{i} = n \cdot \Theta(\log n) = \Theta(n \log n).$$

This is the *harmonic series*: $\sum_{i=1}^{n} 1/i = \Theta(\log n)$.

Answer: $\Theta(n \log n)$.

Part (c) — Three-Way Merge Sort

(i) Recurrence. `mergeSort3` makes $a = 3$ recursive calls, each on a subproblem of size $n/3$.

The `merge3` step costs $O(n)$, so $d = 1$.

$$T(n) = 3T(n/3) + O(n), \qquad T(1) = O(1).$$

(ii) Three-Case Theorem. $a = 3$, $b = 3$, $d = 1$. Compute $b^d = 3^1 = 3$. Since $a = 3 = b^d$, **Case 1** applies.

$$T(n) = O(n^1 \log n) = \Theta(n \log n).$$

(iii) Does splitting into more parts help? No — splitting into 3 parts instead of 2 gives the same $\Theta(n \log n)$ bound. Intuitively, splitting more reduces the depth of recursion ($\log_3 n$ levels instead of $\log_2 n$), but increases the work per level (merging 3 parts instead of 2 still costs $O(n)$). These effects cancel out and the asymptotic bound is unchanged.

To actually improve the bound you would need to reduce the *combine cost* below $O(n)$ — which is not possible for comparison-based merging since you must examine every element. Splitting into more parts helps only in contexts where the bottleneck is something other than the merge, such as disk I/O in external sorting (B-way merge sort).

Problem 2 — Heaps and Hash Tables

Part (a) — Min-Heaps

Array (1-indexed): $A = [3, 9, 5, 12, 11, 8, 7]$.

(i) Heap check and fix. Check each parent–child pair (parent must be $\leq$ both children):

- $i = 1$: $A[1] = 3$. Children: $A[2] = 9$, $A[3] = 5$. $3 \leq 9$ and $3 \leq 5$. ✓
- $i = 2$: $A[2] = 9$. Children: $A[4] = 12$, $A[5] = 11$. $9 \leq 12$ and $9 \leq 11$. ✓
- $i = 3$: $A[3] = 5$. Children: $A[6] = 8$, $A[7] = 7$. $5 \leq 8$ and $5 \leq 7$. ✓

A is already a valid min-heap. No swaps needed.

(ii) Insert 2. Append: $A = [3, 9, 5, 12, 11, 8, 7, 2]$ (index 8). Sift up: parent(8) $= 4$, $A[4] = 12 > 2$. Swap: $A = [3, 9, 5, 2, 11, 8, 7, 12]$. parent(4) $= 2$, $A[2] = 9 > 2$. Swap: $A = [3, 2, 5, 9, 11, 8, 7, 12]$. parent(2) $= 1$, $A[1] = 3 > 2$. Swap: $A = [2, 3, 5, 9, 11, 8, 7, 12]$. Root reached. ✓

(iii) Extract-min. Remove $A[1] = 2$. Move last to root: $A = [12, 3, 5, 9, 11, 8, 7]$. Sift down from $i = 1$: children $A[2] = 3$, $A[3] = 5$. Min child $3 < 12$. Swap $A[1], A[2]$: $A = [3, 12, 5, 9, 11, 8, 7]$. Sift down from $i = 2$: children $A[4] = 9$, $A[5] = 11$. Min child $9 < 12$. Swap $A[2], A[4]$: $A = [3, 9, 5, 12, 11, 8, 7]$. Children of $i = 4$: indices 8, 9 out of range. Stop. ✓

(iv) Finding k smallest elements. *Using a min-heap:* Build a min-heap of all n elements in $O(n)$, then call `extract-min` k times at $O(\log n)$ each. Total: $O(n + k \log n)$.

Using a max-heap of size k: Maintain a max-heap of the k smallest elements seen so far. For each new element, if it is smaller than the heap's maximum, replace the maximum with it. After processing all n elements, the heap contains the k smallest. Each of the n elements requires at most one `insert`/`extract-max` pair: $O(n \log k)$ total. When $k \ll n$, $\log k \ll \log n$, so this is faster in practice and also avoids building a heap over all n elements.

Part (b) — Hash Tables

Keys: $5, 19, 26, 12, 33, 7, 40$. $h(k) = k \bmod 7$.

Hash values: $h(5) = 5$, $h(19) = 5$, $h(26) = 5$, $h(12) = 5$, $h(33) = 5$, $h(7) = 0$, $h(40) = 5$.

(i) Separate chaining.

Slot	Chain
0	7
1	(empty)
2	(empty)
3	(empty)
4	(empty)
5	$5 \to 19 \to 26 \to 12 \to 33 \to 40$
6	(empty)

(ii) Linear probing.

Slot	Key	Notes
0	7	$h = 0$, empty
1	19	$h = 5$, slots 5,6,0 full $\rightarrow$ 1 empty
2	26	$h = 5$, slots 5–1 full $\rightarrow$ 2 empty
3	12	$h = 5$, slots 5–2 full $\rightarrow$ 3 empty
4	33	$h = 5$, slots 5–3 full $\rightarrow$ 4 empty
5	5	$h = 5$, empty
6	40	$h = 5$, slot 5 full $\rightarrow$ 6 empty

(iii) Collision analysis. All six keys $5, 19, 26, 12, 33, 40$ hash to slot 5 under $h(k) = k \bmod 7$ — a severe clustering problem since $5, 12, 19, 26, 33, 40$ form an arithmetic sequence with common difference 7.

With table size 11 and $h(k) = k \bmod 11$: $h(5) = 5$, $h(12) = 1$, $h(19) = 8$, $h(26) = 4$, $h(33) = 0$, $h(40) = 7$. No two keys collide. This illustrates that choosing a **prime** table size reduces collisions when keys have arithmetic structure, since a prime modulus is less likely to share factors with the key spacing.

Problem 3 — BFS and DFS

Graph: A–B, A–D, B–C, B–E, C–E, C–F, D–G, E–G. Source: A.

Part (a) — BFS

(i) BFS queue table.

Step	Dequeue	Enqueue	Queue after	Notes
0	—	A	$[A]$	Source, $d(A) = 0$
1	A	B, D	$[B, D]$	$d(B) = 1$, $d(D) = 1$
2	B	C, E	$[D, C, E]$	$d(C) = 2$, $d(E) = 2$
3	D	G	$[C, E, G]$	$d(G) = 2$
4	C	F	$[E, G, F]$	E already enqueued; $d(F) = 3$
5	E	—	$[G, F]$	B, C, G already visited
6	G	—	$[F]$	D, E already visited
7	F	—	$[\,]$	Done

(ii) Distances. $d(A) = 0$, $d(B) = 1$, $d(C) = 2$, $d(D) = 1$, $d(E) = 2$, $d(F) = 3$, $d(G) = 2$.
(iii) BFS tree edges: A–B, A–D, B–C, B–E, D–G, C–F. Non-tree edges: C–E (E already discovered), E–G (G already discovered).

Part (b) — DFS

(i) DFS stack table.

Counter	Event	Vertex	Push / Pop	Stack after
1	Discover	A	Push A	$[A]$
2	Discover	B	Push B	$[A, B]$
3	Discover	C	Push C	$[A, B, C]$
4	Discover	E	Push E	$[A, B, C, E]$
5	Discover	G	Push G	$[A, B, C, E, G]$
6	Discover	D	Push D	$[A, B, C, E, G, D]$
7	Finish	D	Pop D	$[A, B, C, E, G]$
8	Finish	G	Pop G	$[A, B, C, E]$
9	Finish	E	Pop E	$[A, B, C]$
10	Discover	F	Push F	$[A, B, C, F]$
11	Finish	F	Pop F	$[A, B, C]$
12	Finish	C	Pop C	$[A, B]$
13	Finish	B	Pop B	$[A]$
14	Finish	A	Pop A	$[\,]$

Notes on key decisions: At step 3, B's neighbors in alphabetical order are A (visited), C, E. Go to C first. At step 4, C's neighbors are B (visited), E, F. Go to E. At step 5,

E's neighbors are B (visited), C (visited), G. Go to G. At step 6, G's neighbors are D, E (visited). Go to D. D's only neighbor is A (visited) and G (visited). Finish D.

(ii) Discovery and finish times.

Vertex	A	B	C	D	E	F	G
$d[v]$	1	2	3	6	4	10	5
$f[v]$	14	13	12	7	9	11	8

(iii) Edge classification. Tree edges: A–B, B–C, C–E, E–G, G–D, C–F. Back edges: C–B? No, B is an ancestor of C, so C–B is a back edge. Wait — in undirected DFS every non-tree edge connecting a vertex to an ancestor is a back edge. B–E: when B is revisited from E, B is an ancestor of E, so B–E is a **back edge**, revealing the cycle $B \to C \to E \to B$. Similarly A–D: D is discovered through G, so when A's neighbor D is checked, D is already finished — this is a **cross edge** in directed DFS, but in undirected DFS all non-tree edges are back edges.

Summary: Tree edges as above. Back edges: B–E (cycle B–C–E–B) and A–D (already finished when A checks it). **Cycle present:** B–C–E–B.

Problem 4 — Fractional Knapsack

Capacity $W = 15$. Items: A(4kg, \$20), B(6kg, \$18), C(3kg, \$12), D(5kg, \$15), E(7kg, \$14).

(a) Value-to-weight ratios, sorted.

Item	A	B	C	D	E
v_i/w_i	5.00	3.00	4.00	3.00	2.00

Sorted order (highest to lowest ratio): A (5.00), C (4.00), B (3.00), D (3.00), E (2.00).

(b) Greedy execution.

- Take all of A (4 kg, \$20). Remaining capacity: $15 - 4 = 11$ kg.
- Take all of C (3 kg, \$12). Remaining capacity: $11 - 3 = 8$ kg.
- Take all of B (6 kg, \$18). Remaining capacity: $8 - 6 = 2$ kg.
- Take 2/5 of D (2 kg, $\$15 \times 2/5 = \$6$). Remaining capacity: 0 kg.

(c) Total value. $\$20 + \$12 + \$18 + \$6 = \$56$. This is optimal — the greedy algorithm is provably optimal for fractional knapsack because taking the highest ratio item first maximizes value density. Any deviation would replace a higher-ratio fraction with a lower-ratio one, reducing total value.

(d) 0/1 knapsack with greedy. Applying the same greedy order but taking whole items only: Take A (4 kg), C (3 kg), B (6 kg) — total 13 kg, \$50. Only E (7 kg) remains and doesn't fit. Final: \$50.

Can we do better? Try $A + C + D$: $4 + 3 + 5 = 12$ kg, $\$20 + \$12 + \$15 = \$47$. Worse. Try $A + B + D$: $4 + 6 + 5 = 15$ kg, $\$20 + \$18 + \$15 = \$53$. Better than greedy's \$50!

So the greedy solution (\$50) is **not optimal** for 0/1 knapsack. The optimal solution is $\{A, B, D\}$ with total value \$53.

(e) Why greedy fails for 0/1 knapsack. In the fractional version, you can always fill remaining capacity exactly by taking a fraction of the next item, so the greedy ratio ordering wastes nothing. In the 0/1 version, taking a whole item may leave a gap of remaining capacity that cannot be filled efficiently. Greedy commits to high-ratio items without considering whether lower-ratio items might combine better to fill the knapsack completely. Dynamic programming is needed to consider all combinations.

Problem 5 — Dijkstra's Algorithm

Graph: $S \to A(2)$, $S \to B(6)$, $A \to C(3)$, $A \to B(4)$, $B \to C(2)$, $C \to T(5)$, $B \to T(7)$. Source: S.

(a) Distance table.

Step	dist[S]	dist[A]	dist[B]	dist[C]	dist[T]
Initial	0	∞	∞	∞	∞
Extract S	0	2	6	∞	∞
Extract A	0	2	6	5	∞
Extract C	0	2	6	5	10
Extract B	0	2	6	5	10
Extract T	0	2	6	5	10

Key updates: After extracting S: $A \leftarrow 2$, $B \leftarrow 6$. After extracting A: $C \leftarrow 2+3=5$; B: $2+4=6$, no improvement. After extracting C: $T \leftarrow 5+5=10$. After extracting B: T: $6+7=13$, no improvement; C: already finalized.

(b) Shortest path $S \to T$: distance **10**. Path: $S \xrightarrow{2} A \xrightarrow{3} C \xrightarrow{5} T$.

(c) Shortest path tree parents.

Vertex	S	A	B	C	T
Parent	—	S	S	A	C

Tree edges: $S \to A$, $S \to B$, $A \to C$, $C \to T$.

(d) Why extraction finalizes distance. When vertex u is extracted, dist[u] is the smallest distance among all unextracted vertices. Any alternative path to u must pass through some other unextracted vertex v with dist[v] $\geq$ dist[u]. Since all edge weights are non-negative, extending that path through v can only increase the cost: dist[v]$+w(v,\ldots) \geq$ dist[u]. Therefore no future path can improve dist[u], and it is final.