Practice Final Examination — Solutions

SCIS 313: Data Structures and Algorithm Analysis

Prof. Antonio Khalil Moretti Spelman College

Problem 1 — Code Analysis and Recurrences 20 points

Part (a) — Iterative Code Analysis 10 points

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

For a fixed i, the inner loop runs from $j = i$ to $j = n$, executing $n - i + 1$ times. Summing over all i:

$$\sum_{i=1}^{n}(n - i + 1) = \sum_{k=1}^{n} k \quad (\text{substituting } k = n - i + 1) = \frac{n(n+1)}{2} = \Theta(n^2).$$

Answer: $\Theta(n^2)$.

Part (b) — Recurrence Relations 10 points

(i) Recurrence. In the even case (which determines the dominant behavior), `fastPow` makes $a = 1$ recursive call on a subproblem of size $n/2$. The work outside the recursive call is $O(1)$ (one multiplication), so $d = 0$.

$$T(n) = T(n/2) + O(1), \qquad T(1) = O(1).$$

(ii) Three-Case Theorem. $a = 1$, $b = 2$, $d = 0$. Compute $b^d = 2^0 = 1$. Since $a = 1 = b^d$, **Case 1** applies.

$$T(n) = O(n^0 \log n) = O(\log n) = \Theta(\log n).$$

(iii) Comparison to naive. The naive approach runs in $\Theta(n)$; `fastPow` runs in $\Theta(\log n)$, which is exponentially faster. In RSA, the exponent e may be 2048 bits long, meaning $n \approx 2^{2048}$. The naive method would require 2^{2048} multiplications — more operations than there are atoms in the observable universe. `fastPow` reduces this to about 2048 multiplications, making RSA computationally feasible.

Problem 2 — Heaps and Hash Tables 20 points

Part (a) — Heaps 10 points

Array (1-indexed): $A = [10, 7, 8, 5, 3, 12, 6]$.

(i) Heap check and fix. Check each parent–child pair using the index formulas:

- $i = 1$: $A[1] = 10$. Children at $2i = 2$ ($A[2] = 7$) and $2i + 1 = 3$ ($A[3] = 8$). ✓
- $i = 2$: $A[2] = 7$. Children at 4 ($A[4] = 5$) and 5 ($A[3] = 3$). ✓
- $i = 3$: $A[3] = 8$. Children at 6 ($A[6] = 12$) and 7 ($A[7] = 6$). **Violation**: $12 > 8$.

Swap $A[3]$ and $A[6]$: $A = [10, 7, 12, 5, 3, 8, 6]$. Re-check $i = 1$: $A[1] = 10$, children 7 and 12. **Violation**: $12 > 10$. Swap $A[1]$ and $A[3]$: $A = [12, 7, 10, 5, 3, 8, 6]$. All pairs now satisfy the max-heap property. ✓

(ii) Insert 11. Append: $A = [12, 7, 10, 5, 3, 8, 6, 11]$ (new element at index 8). Sift up: parent(8) = 4, $A[4] = 5 < 11$. Swap: $A = [12, 7, 10, 11, 3, 8, 6, 5]$. parent(4) = 2, $A[2] = 7 < 11$. Swap: $A = [12, 11, 10, 7, 3, 8, 6, 5]$. parent(2) = 1, $A[1] = 12 > 11$. Stop. ✓

(iii) Extract-max. Remove $A[1] = 12$. Move last element to root: $A = [5, 11, 10, 7, 3, 8, 6]$. Sift down from index 1: children at 2 (11) and 3 (10). Max child is $11 > 5$. Swap $A[1]$ and $A[2]$: $A = [11, 5, 10, 7, 3, 8, 6]$. Children of $i = 2$: at 4 (7) and 5 (3). Max child is $7 > 5$. Swap $A[2]$ and $A[4]$: $A = [11, 7, 10, 5, 3, 8, 6]$. Children of $i = 4$: at 8 and 9 (out of range). Stop. ✓

(iv) Why build-heap is $\Theta(n)$. Most nodes are near the bottom of the tree and sift down very few levels. The total work is $\sum_{h=0}^{\log n}(\text{nodes at height } h) \cdot h \approx \sum_{h=0}^{\log n} \frac{n}{2^h} \cdot h = O(n)$ (geometric series). In contrast, inserting elements one at a time requires each new element to sift *up* as far as $\log n$ levels, giving $\sum_{i=1}^{n} \log i = \Theta(n \log n)$.

Part (b) — Hash Tables 10 points

Keys: $22, 35, 14, 9, 16, 28, 3$. $h(k) = k \bmod 7$. Table size 7.

Hash values: $h(22) = 1$, $h(35) = 0$, $h(14) = 0$, $h(9) = 2$, $h(16) = 2$, $h(28) = 0$, $h(3) = 3$.

(i) Separate chaining.

Slot	Chain
0	$35 \to 14 \to 28$
1	22
2	$9 \to 16$
3	3
4	(empty)
5	(empty)
6	(empty)

(ii) Linear probing.

Slot	Key	Notes
0	35	$h(35) = 0$, empty
1	22	$h(22) = 1$, empty
2	14	$h(14) = 0$, slot 0 full $\rightarrow$ 1 full $\rightarrow$ 2 empty
3	9	$h(9) = 2$, slot 2 full $\rightarrow$ 3 empty
4	16	$h(16) = 2$, slots 2,3 full $\rightarrow$ 4 empty
5	28	$h(28) = 0$, slots 0–4 full $\rightarrow$ 5 empty
6	3	$h(3) = 3$, slots 3–5 full $\rightarrow$ 6 empty

(iii) Load factor. $\alpha = 7/7 = 1$. In chaining, α equals the average chain length, so expected lookup is $O(1+\alpha)$. In open addressing, as $\alpha \rightarrow 1$ the table fills up and long *clusters* form: consecutive occupied slots force long probe sequences. Each new insertion extends clusters, making future collisions more likely. This cascading effect does not occur in chaining since overflow simply extends a linked list without blocking other slots.

Problem 3 — BFS / DFS

25 points

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

(a) BFS discovery order and distances.
Queue starts with A.

1. Dequeue A. Enqueue neighbors B, C. Distances: $d(A) = 0$.
2. Dequeue B. Enqueue D, E (not yet seen). $d(B) = 1$.
3. Dequeue C. Enqueue F, G. $d(C) = 1$.
4. Dequeue D. No new neighbors. $d(D) = 2$.
5. Dequeue E. Neighbor F already enqueued. $d(E) = 2$.
6. Dequeue F. Neighbor E already seen. $d(F) = 2$.
7. Dequeue G. $d(G) = 2$.

Discovery order: A, B, C, D, E, F, G. Distances: $d(A) = 0$, $d(B) = 1$, $d(C) = 1$, $d(D) = 2$, $d(E) = 2$, $d(F) = 2$, $d(G) = 2$.

(b) BFS tree edges: A–B, A–C, B–D, B–E, C–F, C–G. (The edge E–F is a *cross edge* in BFS and is not a tree edge.)

(c) DFS times. Counter starts at 1; increments on discover and on finish.

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

DFS visits: discover $A(1)$, discover $B(2)$, discover $D(3)$, finish $D(4)$, discover $E(5)$, discover $F(6)$, finish $F(7)$, finish $E(8)$, finish $B(9)$... wait — B's alphabetical neighbors are D, E; after E finishes, back to B: finish $B(10)$... then C: discover $C(11)$, discover $G(12)$, finish $G(13)$, finish $C(14)$, finish $A(15)$.

Corrected table:

Vertex	$d[v]$	$f[v]$
A	1	16
B	2	11
C	12	15
D	3	4
E	5	8
F	6	7
G	13	14

(d) Edge classification. Tree edges (used in DFS tree): A–B, B–D, B–E, E–F, A–C, C–G. Back edge: F–E would be a back edge if F were discovered inside E's subtree and pointed back—here E–F is a tree edge and F has no other edges. The edge C–F: F is already

finished when C is processed, so C–F is a **cross edge**. No back edge exists in this graph, so **no cycle** is present (this graph is acyclic).

(e) Runtime. Both BFS and DFS run in $\Theta(V + E)$ on an adjacency list. Each vertex is enqueued/discovered exactly once: $O(V)$ total. Each edge (u, v) is examined when u is processed (and again when v is processed in undirected graphs): $O(E)$ total. Together: $\Theta(V + E)$.

Problem 4 — Greedy: Interval Scheduling

20 points

(a) Sorted by finish time: $A(4)$, $B(5)$, $C(6)$, $D(7)$, $E(8)$, $F(9)$, $G(10)$.

(b) Greedy execution.

- Consider A (finish 4). No conflict. **Select** A. Last finish = 4.
- Consider B (start 3, finish 5). $3 < 4$: conflicts. **Reject**.
- Consider C (start 0, finish 6). $0 < 4$: conflicts. **Reject**.
- Consider D (start 5, finish 7). $5 \geq 4$: no conflict. **Select** D. Last finish = 7.
- Consider E (start 3, finish 8). $3 < 7$: conflicts. **Reject**.
- Consider F (start 5, finish 9). $5 < 7$: conflicts. **Reject**.
- Consider G (start 6, finish 10). $6 < 7$: conflicts. **Reject**.

Selected set: $\{A, D\}$, size 2.

(c) The classmate's set $\{C, D, G\}$ has size 3, which is *larger* than the greedy solution's size of 2. Wait—let us recheck: C ends at 6, D starts at 5 ($5 < 6$, overlap!). So $\{C, D, G\}$ is not a valid non-overlapping set. The greedy solution of $\{A, D\}$ is in fact optimal for this instance. The greedy algorithm always finds a *maximum* set, so no valid alternative of size 3 exists here.

(d) Exchange argument. (i) Since $f(g) \leq f(x)$, activity g finishes no later than x. Any activity in OPT that starts after $f(x)$ also starts after $f(g)$, so replacing x with g creates no new conflicts. OPT$'$ is valid. (ii) OPT$'$ is formed by replacing exactly one activity (x) with one activity (g), so $|\text{OPT}'| = |\text{OPT}|$. (iii) Together: there exists an optimal solution containing g. The greedy first choice is always safe — we never lose optimality by choosing g.

(e) Runtime. Sorting takes $O(n \log n)$. The selection scan is a single pass: $O(n)$. Total: $O(n \log n)$. The sort dominates.

Problem 5 — Dijkstra's Algorithm

15 points

Graph: $S \to A(4)$, $S \to B(2)$, $A \to B(1)$, $A \to C(5)$, $B \to C(8)$, $B \to D(3)$, $D \to C(1)$, $D \to T(6)$, $C \to T(2)$.

(a) Distance table.

Step	dist[S]	dist[A]	dist[B]	dist[C]	dist[D]	dist[T]
Initial	0	∞	∞	∞	∞	∞
Extract S	0	4	2	∞	∞	∞
Extract B	0	3	2	10	5	∞
Extract A	0	3	2	8	5	∞
Extract D	0	3	2	6	5	11
Extract C	0	3	2	6	5	8
Extract T	0	3	2	6	5	8

Key updates: After extracting B: A updated to $2+1=3$; C to $2+8=10$; D to $2+3=5$. After extracting A: C updated to $3+5=8$ (better than 10). After extracting D: C updated to $5+1=6$ (better than 8); T to $5+6=11$. After extracting C: T updated to $6+2=8$ (better than 11).

(b) Shortest path $S \to T$: distance **8**. Path: $S \xrightarrow{2} B \xrightarrow{3} D \xrightarrow{1} C \xrightarrow{2} T$.

(c) The greedy choice is: always extract the unvisited vertex with the smallest current dist value. A vertex need not be re-extracted because all edge weights are non-negative: once a vertex u is extracted, any path to u through a later-extracted vertex v would cost at least $\text{dist}[u] + w \geq \text{dist}[u]$. So the first time u is extracted, its distance is already optimal.

(d) `extract-min` is called V times at $O(\log V)$ each: $O(V \log V)$. Distance updates occur at most once per edge, each costing $O(\log V)$: $O(E \log V)$. Total: $O((V+E)\log V)$.

Problem 6 — Extra Credit: Coin Change

10 points

Denominations $\{1, 4, 6\}$, target $n = 10$.

(a) Recurrence. Subproblem: $C(k)$ = minimum number of coins to make amount k.

$$C(k) = 1 + \min_{c \in \{1,4,6\},\, c \le k} C(k - c), \qquad C(0) = 0.$$

$C(k) = \infty$ if $k < 0$ (impossible).

(b) DP table.

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

Sample work: $C(4) = 1 + C(0) = 1$ (use coin 4). $C(6) = 1 + C(0) = 1$ (use coin 6). $C(8) = 1 + \min(C(7), C(4), C(2)) = 1 + \min(2, 1, 2) = 2$ (use coin 4). $C(10) = 1 + \min(C(9), C(6), C(4)) = 1 + \min(3, 1, 1) = 2$ (use $4 + 6$ or $4 + 4 + \ldots$; best is two coins: $4 + 6$).

(c) The DP solution uses **2 coins** ($4+6$), which is better than the greedy's 5 coins. Greedy fails because it commits to the largest coin (6) without considering that two medium coins (4+4, 4+6) might be better. Greedy worked for interval scheduling because the exchange argument showed the first greedy choice is always in *some* optimal solution — that structural guarantee does not hold for coin change with arbitrary denominations.