# Crib for 4M26 Algorithms and Data Structures 2024/2025

Version: AT/4

## Question 1

### a)

#### i)

For a given function $g(n)$,
$\Omega(g(n)) = \{f(n) : \text{there exists positive constants c and } n_0 \text{ such that } 0 \leq cg(n) \leq f(n) \text{ for all } n \geq n_0\}$

#### ii)

Consider some function $f \in O(n^2)$. Then by this statement $f$ can be any function smaller than $n^2$, such as $n$, etc. Hence we learn nothing about the lower bound. Further, since the statement says at least. there is also no information about the upper bound. Therefore the statement is meaningless.
We can write lower bound as $f \in \Omega(n^2)$.

### b)

#### i)

```
for k in range(p, r + 1):
    if L[i] <= R[j]:
        A[k] = L[i]
        i += 1
    else:
        A[k] = R[j]
        j += 1
```

#### ii)

- **Invariant:** At the start of each iteration of the loop, the subarray $A[p : k]$ contains the $k-p$ smallest elements of $L[0 : i]$ and $R[0 : j]$ in sorted order. $L[i]$ and $R[j]$ are the smallest elements of their respective arrays not yet copied to $A$.
- **Maintenance:** If $L[i] \leq R[j]$, then $L[i]$ is the smallest element not yet in $A$. After assigning $A[k] = L[i]$ and incrementing $i$ and $k$, the subarray $A[p \dots k]$ now contains the $k - p + 1$ smallest elements, preserving the invariant for the next iteration.

### c)

$T(n) = 2T(n/2) + \Theta(n)$:

#### i)

- $a = 2, b = 2, f(n) = \Theta(n)$.
- Critical value: $n^{\log_b a} = n^{\log_2 2} = n^1$.
- Since $f(n) = \Theta(n^1)$, this is **Case 2** of the Master Theorem.
- Result: $T(n) = \Theta(n \log n)$.

**ii)**

- **Level 0:** $c(8)$
- **Level 1:** $c(4) + c(4) = c(8)$
- **Level 2:** $c(2) + c(2) + c(2) + c(2) = c(8)$
- **Level 3:** $8 \times T(1) = \Theta(8)$
- **Total:** $\sum_{i=0}^{\log_2 8} c(8) = c(8) \times 4$ levels.

# Question 2

## a)

### i)

Average is $O(1)$. Worst case is $\Theta(n)$.
Worst case when all keys map to the same entry.

### ii)

With $K$ slots and $N$ data points, load factor: $\alpha = N/K$. It is the average number of elements stored in a chain.
Storage is $\Theta(K + N)$. Partial marks for $\Theta(N)$, as this would be the case if $\Theta(K + N) = \Theta(N)$.

### ii)

$m = 7, h(k) = k \bmod 7$. Keys: $19, 26, 13, 48, 17, 6$.
**Final Table:** [0: empty], [1: empty], [2: empty], [3: 17], [4: empty], [5: 19, 26], [6: 13, 48, 6].

## b)

### i)

Greedy choice: If a problem has the greedy choice property, then there exists an optimal solution that can be constructed by making a locally optimal (greedy) choice first.

The greedy choice property guarantees that making the greedy choice does not rule out an optimal solution.

### ii)

Frequencies: A: 8, B:24, C:23, D:13, E:18, F:25, G:1

1. Merge G(1) and A(8) $\rightarrow$ GA(9).
2. Merge GA(9) and D(13) $\rightarrow$ GAD(22).
3. Merge E(18) and GAD(22) $\rightarrow$ EGAD(40).
4. Merge C(23) and B(24) $\rightarrow$ CB(47).
5. Merge F(25) and EGAD(40) $\rightarrow$ FEGAD(65).
6. Merge CB(47) and FEGAD(65) $\rightarrow$ CBFEGAD(112).

### iii)

A: 11101
B: 01
C: 00
D: 1111
E: 110
F: 10
G: 11100

**iii)**

Huffman coding is greedy because at each step it makes the locally optimal choice of merging the two nodes with the lowest frequencies. It does not look ahead to see how this affects future merges; it simply assumes that reducing the cost of the subtree is the best path to the overall minimum-weight tree.

# Question 3

## a)

Correctness: swap the values properly, check for valid ranges
Style: precompute ranges (i.e. don't use the accessor), use zip, swap in place

```
def swap_rows(self, i: int, j: int):
    assert 0 <= i < self.rows
    assert 0 <= j < self.rows
    i_start = i * self.columns
    i_end = i_start + self.columns
    j_start = j * self.columns
    j_end = j_start + self.columns
    for ik, jk in zip(range(i_start, i_end), range(j_start, j_end)):
        self.values[ik], self.values[jk] = self.values[jk], self.values[ik
            ]
```

## b)

Correctness: correct computation, deal with the start offset properly, check for valid ranges
Style: precompute ranges, use zip

```
def add_rows(self, i: int, j: int, scale: float, start=0):
    assert 0 <= i < self.rows
    assert 0 <= j < self.rows
    assert 0 <= start < self.columns
    i_start = i * self.columns
    i_end = i_start + self.columns
    j_start = j * self.columns
    j_end = j_start + self.columns
    for ik, jk in zip(range(i_start + start, i_end), range(j_start + start
        , j_end)):
        self.values[ik] += s * self.values[jk]
```

## c)

The main loop has to execute $min(m, n)$ times. Each iteration requires:

1. Computing the maximum pivot, which needs $m - h$ comparisons
2. If a pivot is not found, then one increment
3. Otherwise:
   (a) Swapping two rows requiring $n$ swaps
   (b) For $m - h - 1$ rows:
      i. One floating-point division, one floating-point negation
      ii. $n - k$ floating-point multiplications and subtractions
4. Back substitution loop executes $m$ times. Each iteration requires:
   (a) One floating-point comparison
   (b) If the value at A[p, p] is zero, nothing more
   (c) Otherwise, for $p - 1$ rows:
      i. One floating-point division, one floating-point negation
      ii. $n - c$ floating-point multiplications and subtractions

Given this analysis, the the upper bound is $O(n^3)$. Responses do not need to be this detailed, but the answer has to be justified.

## d)

Create a matrix like the one below:

$$\begin{matrix} 0 & 0 & 0 & 1 & 1 \\ 1 & 1 & 0 & 0 & 5 \\ 0 & 0 & 1 & 1 & 3 \\ 0 & 1 & 0 & 0 & 2 \end{matrix}$$

in which the first four columns encode the button sets (each row corresponds to a counter label, $A = 0$, $B = 1$, etc.) and the final column encodes the counter values. Then, use Gauss-Jordan elimination to solve the linear system. In this instance, the solution is unique and consists of non-negative integers, so the sum of the final column gives the minimum number of button presses.

# Question 4

## a)

The evacuation problem can be modelled as a directed graph with cities represented as vertices and one-way highways represented as directed edges (5%). Each highway is represented by an edge that stores both a capacity attribute (people per hour) and a travel-time attribute (edge weight) (5%). Cities that contain evacuation centres are represented as a specified subset of vertices and are treated as destinations in subsequent computations (5%).
(Total: 15%)

## b)

The problem can be solved by performing a graph traversal (e.g., using depth-first search or breadth-first search) to test reachability to an evacuation centre (for example, by reversing all highway directions and traversing from evacuation-centre cities, or by traversing forward from each city).
The description should make clear that the traversal systematically explores cities reachable via directed highways and checks whether every city can reach at least one evacuation centre (10%). Representing the graph as an adjacency list, the traversal runs in $O(|V| + |E|)$ time, since each city and highway is visited at most once. Alternative answer: using an adjacency matrix, the traversal runs in $O(|V|^2)$ (5%).
(Total: 15%)

## c)

### i)

**Input:** Graph $G = (V, E)$ with non-negative edge weights $w(u, v)$; Source vertex $s$
**Output:** Shortest distance $d[v]$ from $s$ to every vertex $v$

```
    INITIALISE-SINGLE-SOURCE(G, s)
 1: for each vertex v ∈ V[G] do
 2:     d[v] ← ∞
 3:     π[v] ← NIL
 4: end for
 5: d[s] ← 0

    RELAX(u, v, w)
 6: if d[v] > d[u] + w(u, v) then
 7:     d[v] ← d[u] + w(u, v)
 8:     π[v] ← u
 9: end if

    DIJKSTRA(G, w, s)
10: INITIALISE-SINGLE-SOURCE(G, s)
11: S ← ∅
12: Q ← V[G]
13: while Q ≠ ∅ do
14:     u ← EXTRACT-MIN(Q)
15:     S ← S ∪ {u}
16:     for each vertex v ∈ Adj[u] do
17:         RELAX(u, v, w)
18:     end for
19: end while
20: return d
```

(Total: 30%)

### ii)

$O(|V|^2)$ using min-priority queue without heap, or $O((|V| + |E|)log|V|)$ with binary heap, or $O(|E| + |V|log|V|)$ with Fibonacci heap (3%). The key assumption is that all edge weights must be non-negative (2%).
(Total: 5%)

## d)

### i)

Ford-Fulkerson algorithm, or Edmonds-Karp algorithm (5%).
The problem can be transformed into a network flow problem by introducing a super-source connected to all cities and a super-sink connected from all evacuation-centre cities, with each highway represented as a directed edge with capacity equal to its evacuation capacity (10%).
A maximum flow is then computed from the super-source to the super-sink to obtain the maximum total number of people per hour that can be evacuated (5%).
(Total: 20%)

**ii)**

Runtime is $O(|E|F)$ for Ford-Fulkerson algorithm or $O(|V||E|^2)$ for Edmonds-Karp algorithm (5%).
(Total: 5%)

## e)

After computing the maximum flow, the most critical highways can be identified by examining the residual graph and finding edges that cross from vertices reachable from the source to vertices that are not reachable. These edges form a minimum cut (7%). Identifying these highways requires a single graph traversal of the residual network and therefore runs in $O(|V| + |E|)$ time using adjacency list (or $O(|V|^2)$ using adjacency matrix) (3%).
(Total: 10%)