# CSE 101 Homework 4

Kreshiv Chawla, Brian Masse, Taira Sakamoto, Emily Xie, Annabelle Coles

October 31, 2025

1. **Maximum consecutive sum**

Here is an algorithm that, given an array of integers (not necessarily positive) $A[1..n]$, finds the maximum sum of a consecutive subarray, $\max_{1 \le I \le J \le n} \sum_{K=I}^{J} A[K]$.

```
(a) Initialize $MaxSumEndingAt[1..n]$.
(b) $MaxSumEndingAt[1] = A[1]$
(c) For $J = 2$ to $n$ do:
(d)     $MaxSumEndingAt[J] = \max(A[J],\ A[J] + MaxSumEndingAt[J-1])$
(e) $MaxConsSum = MaxSumEndingAt[1]$.
(f) For $J = 2$ to $n$ do:
(g)     $MaxConsSum = \max(MaxConsSum,\ MaxSumEndingAt[J])$
(h) Return $MaxConsSum$
```

(a) For each $J \in \{1, \dots, n\}$,

$$\text{MaxSumEndingAt}[J] = \max_{1 \le i \le J} \sum_{k=i}^{J} A[k]$$

This means it stores the largest sum of any non-empty subarray that ends at position $J$. From the algorithm, this can also be written as:

$$\text{MaxSumEndingAt}[J] = \max(A[J],\ A[J] + \text{MaxSumEndingAt}[J-1]).$$

(b) **Loop invariant:** Before each loop iteration at index $J$ ($J \ge 2$), MaxSumEndingAt$[t]$ holds the correct value for all $t \in \{1, \dots, J-1\}$. After computing line 4 for $J$, the invariant also holds for index $J$.

**Proof:**

- *Base case ($J = 2$):* This is the state before the first loop iteration. Line 2 sets MaxSumEndingAt$[1] = A[1] = \max_{1 \le i \le 1} \sum_{k=i}^{1} A[k]$.
- *Inductive step:* Assume the invariant holds up to $J-1$. The best subarray ending at $J$ either:
  i. starts fresh at $A[J]$, or
  ii. continues from the best subarray ending at $J-1$.

  Therefore,

  $$\text{MaxSumEndingAt}[J] = \max(A[J],\ A[J] + \text{MaxSumEndingAt}[J-1]),$$

  which matches the formula, so the invariant holds for $J$.

(c) The goal is to find:

$$\text{Ans} = \max_{1 \le I \le J \le n} \sum_{k=I}^{J} A[k].$$

Since each MaxSumEndingAt$[J]$ gives the best sum ending at $J$,

$$\text{Ans} = \max_{1\le J\le n} \text{MaxSumEndingAt}[J].$$

The second loop keeps track of this maximum:

$$\text{MaxConsSum} = \max(\text{MaxConsSum}, \text{MaxSumEndingAt}[J]).$$

Initially (line 5), MaxConsSum = MaxSumEndingAt[1]. At the end (line 8), $J = n$ so MaxConsSum $= \max_{1\le J\le n}$ MaxSumEndingAt$[J]$ = Ans.

Hence, the algorithm is correct.

(d) The first loop (lines 3–4) runs $n-1$ times, each taking constant time. The second loop (lines 6–7) also runs $n-1$ times. Therefore, the total time is $\Theta(n)$.

2. **Maximum within sliding window**

You are given an array $A[1..n]$ of integers, and an integer $1 \leq k \leq n$. You wish to compute $MaxInWindow[I] = max_{I \leq J \leq min(I+k,n)} A[J]$ for all $1 \leq I \leq n$. By answering the questions below, you'll develop an efficient algorithm for this problem. A helpful abstraction will be to consider a window set at each $I$, $WS[I] = \{(J, A[J]) | I \leq J \leq min(I+k, n)\}$.

(a) In terms of the window set at $I$, what is $MaxInWindow[I]$?
$MaxInWindow[I] \;=\; \max\{\, v \mid (J, v) \in WS[I] \,\}$.

(b) What is the difference between $WS[I]$ and $WS[I+1]$?
The window set $WS[I+1]$ differs from $WS[I]$ by one element leaving and one new element entering the window. Specifically, $(I, A[I])$ is removed from the set and $(I+k+1, A[I+k+1])$ is added, given that $(I+k+1 \leq n)$.

$$WS[I{+}1] \;=\; \big(WS[I] \setminus \{(I, A[I])\}\big) \;\cup\; \begin{cases} \{(I{+}k{+}1, A[I{+}k{+}1])\}, & \text{if } I{+}k{+}1 \leq n, \\ \emptyset, & \text{otherwise.} \end{cases}$$

(c) Based on your answers to the first two questions, what data structure operations would we want a data structure for $WS$ to support?

- insert(x) - In order to add the new element entering the window
- delete(x) - In order to delete the element exiting the window
- get_max() - To get the current maximum value within the window

(d) What data structure could we use for $WS$?
Data structures that support the needed operations include:

- Balanced Binary Search Trees (BST or RST)
- Double Ended Queue
- Priority Queue

(e) What is the maximum size of this data structure?
At most $k{+}1$ elements (the window is $[I, \min(I{+}k, n)]$ inclusive).

(f) How long do the different data structure operations take?
*BST:*

- insert $= O(\log k)$
- delete $= O(\log k)$
- get_max $= O(1)$ if we keep a pointer to the largest (or $O(\log k)$ to query the max).

*Plain Dequeue (without monotone maintenance):*

- push/pop at ends = $O(1)$
- get_max requires scanning = $O(k)$

*Max-Heap:*

- get_max = $O(1)$
- insert = $O(\log k)$ ($O(k)$ in worst case)
- delete = $O(\log k)$ ($O(k)$ in worst case)

(g) Give pseudo-code for an algorithm to solve this problem using data structure operations.

```
max_in_window():
    max_arr = [1...n]
    heap = initialize_max_heap( (i, A[i]) for i in range 0...k )

    for i in 1...n:
        max[i] = heap.peek()

        heap.remove(key=i)
        if (i + 1 + k <= n):
            heap.insert((i+1+k, A[i+1+k]))
```

(h) Give a time analysis for your algorithm, using answers to previous questions.

- The loop runs for each index of the array.
- The heap is initialized in $O(logk)$
- heap.peek runs in $O(1)$
- heap.remove / heap.insert both run in $O(logk)$ in the average case.

$$\begin{aligned}\text{Time Complexity} &\in O(logk + n(2 \cdot logk)) \\ &\in O(n \cdot logk)\end{aligned}$$

In the worst case (insert/delete on the heap run in $O(k)$). Time complexity $\in O(n \cdot k)$

3. **Subway stops**

Underneath a city road, there are $n$ subway stops $s_1, ...s_n$ and $k$ subway lines. Each subway line $k$ stops at some of the stops, in the same order, with line $k$ stopping at $s_{i_{k,1}}, ...s_{i_{k,t_k}}$, where $i_{k,1} < i_{k,2}... < i_{k,t_k}$. You want to go from $s_1$ to $s_n$ making as few transfers between lines as possible.

For example, there might be five stops, and three lines. The first line might stop at stops $1, 3$, the second at $2, 4, 5$, and the third at $2, 3, 4$. Then we could board on line 1, take it to stop 3, switch to line 3, take it to stop 4, switch to line 2 and take it to the end.

Below, you will describe how to use a graph algorithm from class to solve the problem.

(a) What graph will you use to solve the problem? Be sure to specify the set of vertices in your graph, the set of edges, whether the edges are directed or undirected, and what weights edges have, if any.

- This will be a directed graph. Each node $S_{i,k}$ represents a subway station $i$ reached by subway line $k$
- There will be unweighted (weight $= 0$), directed edges representing getting from one subway station to another via line $k$
- There will be weighted (weight $= 1$), undirected edges representing switching subway line at a single station. For example, these edges will go from vertex $S_{i,k_1}$ to $S_{i,k_2}$

(b) How will you create the graph from the information given? What format will you use for the graph? How long does it take to create the graph?

To create the graph from the given information:

- Create a grid of k rows by n columns of verticies. Label each $S_{row,col}$
- Draw unweighted, directed edges from $S_{i_1,k}$ to $S_{i_2,k}$ (collection of all stations reachable by line k) for every rail line k.
- Draw weighted (weight $= 1$), undirected edges between the rail stops at each specific station
- Create a single node at the start and end of the graph, with directed, unweighted edges to all $(S_i, k)$ stops. This will serve as the start and end goals of the algorithm.
- Remove any unused rail stops. (a rail line that is not used at station $i$

Given the answers in the following 2 questions, you must draw at most $n \cdot k + 2$ nodes, $(n-1) \cdot k$ rail connections, $2 \cdot (k-1) \cdot n$ rail line changes, and $2k$ edges from the first/last nodes to the first/last stations. Asymptotically, it will take $O(nk)$

(c) How many vertices does your graph have, at most? Give this in terms of $n$ and $k$.

$$|V| \leq n \cdot k + 2$$

(d) How many edges does your graph have, at most? Give this in terms of $n$ and $k$

- Directed Edges. If each rail line connects to every stations, there are at most $(n-1) \cdot k$ edges

- Undirected Edges. To connect every rail line at every station requires 2 directed edges, so there are at most $2(k-1) \cdot n$ edges
- Extra Edges. If there are k rail line connections at the first and last stations, then the first and last nodes will have $k$ edges to and from them respectively.

$$|E| \leq (n-1) \cdot k + 2(k-1) \cdot n + 2 \cdot k$$

(e) How do paths in your graph relate to ways of transfering? What is the relationship between the length of paths and the number of transfers?

Unweighted edges represent no transfers. Subway line transfers have a weight of 1. Thus the weigh of the paths from $S_1$ to $S_n$ is the number of transfers made during that trip.

(f) What algorithm from class will you run on the graph? Be sure to specify all inputs to this algorithm, and say how you use the results.

The modified Dijkstras algorithm should be used to find the shortest weighted path between the first and last subway station. This will automatically minimize the number of transfers.

The algorithm would assume the goal is to get from station $S_1$ to station $S_n$. It would run the modified Dijkstras algorithm from class on the first node and try to reach the last node.

The fully weighted and directed graph would be input (either in adjacency list or matrix format). The result would either be the length of the shortest trip, or an array of previous nodes showing the shortest path.

(g) What is the total time complexity of using this algorithm from class to solve the subway transfer problem? This should be given in terms of $n$ and $k$.

The algorithm uses the same Dijkstras algorithm we used in class, and has the same time complexity: $O(|RemoveMin||V| + |ReduceKey||E|)$

For this problem specifically:

$$\begin{aligned} & O(|RemoveMin| \cdot (nk) + |ReduceKey| \cdot (nk)) \\ \in \; & O((nk) \cdot (|RemoveMin| + |ReduceKey|)) \end{aligned}$$

Depending on the underlying implementation of the min-heap, the exact time complexity would vary.

4. **Vertex costs**

Say we are given a graph $G$ where both vertices and edges have positive integer weights and two vertices $u$, $v$. We want to find a path from $u$ to $v$ that minimizes the total weights of both edges and vertices along the path. Give an efficient algorithm to solve this problem. You can use any algorithm from class as given, but need to relate the correctness guarantee proved for that algorithm to correctness for this new problem.

Can use the following slightly modified Dijkstras:

```
Modified_djikstras():
    dist = [ set all nodes to infinity ]
    prev = [ set all nodes to None ]
    dist[s] = weight[s]

    H = initialize_min_heap(dist)

    while H is not empty:
        u = H.removeMin()
        for edge (u, v) in E:
            potential_dist = dist[u] + length(u, v) + weight[v]
            if potential_dist < dist[v]:
                dist[v] = potential_dist
                H.update_heapify(v, potential_dist)
                prev[v] = u
```

The modification simply takes into account the weight of nodes when deciding whether to call ‘reduceKey.‘

*Time Analysis and Efficiency*

The modification uses the same loop / logic as the base djikstras algorithm, and thus shares time complexity.

Specifically, each node $|V|$ is removed from the heap (removeMin) once. And each edge is checked $|E|$ once. The priorityQueue is implemented with a mean heap, which has the following properties:

- removeMin: $= O(logn)$
- decreaseKey: $= O(logn)$

Thus, the total time complexity is:

$$O(log(n) \cdot (|V| + |E|))$$

*Proof of Correctness*

The proof of correctness is largely the same as the proof of correctness for the base djikstras algorithm, since loop, data structure, and operations are the same. Vertex weights can be thought of as being added onto all incoming verticies.

Specifically, for correctness, use the loop invariant: *When a node u is removed from the priorityQueue, dist[u] is the shortest path, including node weights, from s to u.*

- The base case, when s is removed, is trivially correct. Since the shortest path from s to itself is $weight[s]$
- For the general case, assume that a group $u_1, ..u_k$ have been removed and that $dist[u_i]$ is the shortest path from s to $u_i$ We must show that when $u_{k+1}$ is removed, that $dist[u_{k+1}]$ is the shortest path.

  The remained of the proof is the same as we discussed in class: any supposedly shorter path from s to $u_{k+1}$ would need to cross form the group $u_1...u_k$ to outside the explored nodes. However, any such path would necessarily include a positive edge and vertex weight, making it longer. If those edges / verticies have weight 0, then at best we would have found a path of equivalent distance, proving the invariant.

Thus, when each node is removed, we've fond the shortest path. Then, when all nodes are removed, we will have found the shortest path from s to the ending node.