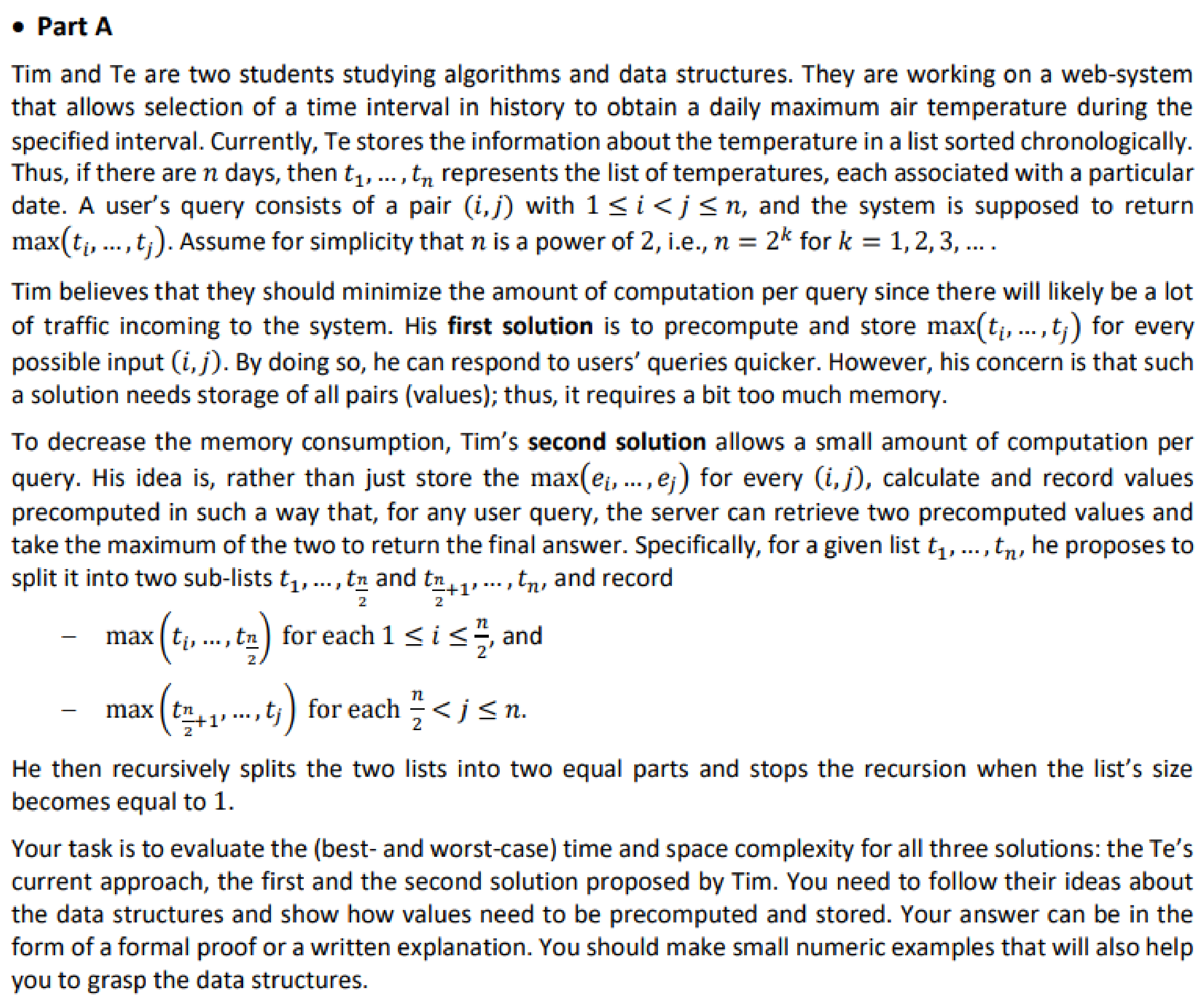


- **Part A**

Tim and Te are two students studying algorithms and data structures. They are working on a web-system that allows selection of a time interval in history to obtain a daily maximum air temperature during the specified interval. Currently, Te stores the information about the temperature in a list sorted chronologically. Thus, if there are n days, then $t_1, \dots, t_n$ represents the list of temperatures, each associated with a particular date. A user's query consists of a pair (i, j) with $1 \leq i < j \leq n$, and the system is supposed to return $\max(t_i, \dots, t_j)$. Assume for simplicity that n is a power of 2, i.e., $n = 2^k$ for $k = 1, 2, 3, \dots$.

Tim believes that they should minimize the amount of computation per query since there will likely be a lot of traffic incoming to the system. His **first solution** is to precompute and store $\max(t_i, \dots, t_j)$ for every possible input (i, j). By doing so, he can respond to users' queries quicker. However, his concern is that such a solution needs storage of all pairs (values); thus, it requires a bit too much memory.

To decrease the memory consumption, Tim's **second solution** allows a small amount of computation per query. His idea is, rather than just store the $\max(e_i, \dots, e_j)$ for every (i, j), calculate and record values precomputed in such a way that, for any user query, the server can retrieve two precomputed values and take the maximum of the two to return the final answer. Specifically, for a given list $t_1, \dots, t_n$, he proposes to split it into two sub-lists $t_1, \dots, t_{\frac{n}{2}}$ and $t_{\frac{n}{2}+1}, \dots, t_n$, and record

- $\max\left(t_i, \dots, t_{\frac{n}{2}}\right)$ for each $1 \leq i \leq \frac{n}{2}$, and
- $\max\left(t_{\frac{n}{2}+1}, \dots, t_j\right)$ for each $\frac{n}{2} < j \leq n$.

He then recursively splits the two lists into two equal parts and stops the recursion when the list's size becomes equal to 1.

Your task is to evaluate the (best- and worst-case) time and space complexity for all three solutions: the Te's current approach, the first and the second solution proposed by Tim. You need to follow their ideas about the data structures and show how values need to be precomputed and stored. Your answer can be in the form of a formal proof or a written explanation. You should make small numeric examples that will also help you to grasp the data structures.

A : For Te's system, the list ti.....tj for each query(i,j).

The time complexity will be O(j – i + 1) = O(n) per query. For his best/worst case.

The space complexity : O(1).

For Tim's first solution, precompute and store the maximum for all possible pairs (i,j).

Total pairs is n(n-1) / 2.

Precompute using dynamic: max(i,j) = max(max(i,j –1) , tj).

Time complexity will be : precomputation O(n^2).

Query : O(1).

Space complexity: O(n^2).

For Tim’s second solution, let recursively split the list and store partial maxima.

Precomputation:

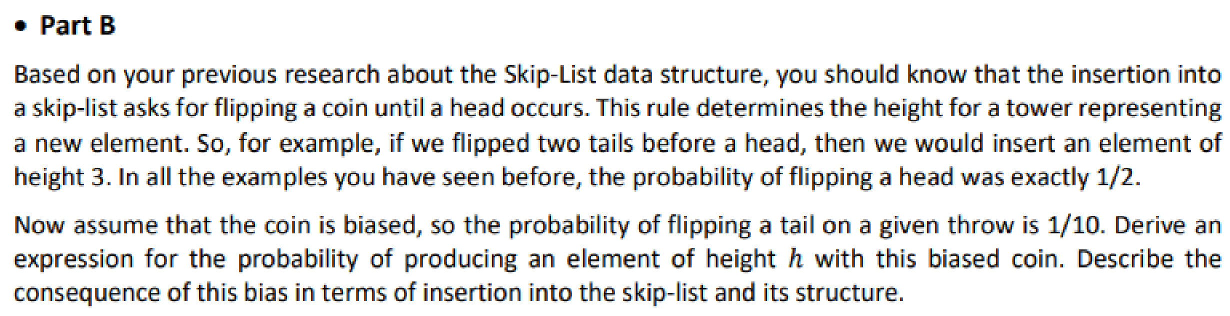
Each recursion level splits the list into halves.

At level l, store 2^l intervals of size n/2^l.

Total nodes : 2n-1.

Space complexity : O(n log n).

Time complexity : O(log n) per query.



- **Part B**

Based on your previous research about the Skip-List data structure, you should know that the insertion into a skip-list asks for flipping a coin until a head occurs. This rule determines the height for a tower representing a new element. So, for example, if we flipped two tails before a head, then we would insert an element of height 3. In all the examples you have seen before, the probability of flipping a head was exactly 1/2.

Now assume that the coin is biased, so the probability of flipping a tail on a given throw is 1/10. Derive an expression for the probability of producing an element of height h with this biased coin. Describe the consequence of this bias in terms of insertion into the skip-list and its structure.

A:

Probability of Height h

Original skip-List: Height h occurs with probability (1/2)^h

Biased Coin:

Continue to increase the number of layers each time you throw a T and stop when you throw an H.

Probability of tail = 1/10, head= 9/10 .

To generate height h, you need to throw T （h−1） times in succession, followed by 1 throw of H.

P(h)=(1/10)^h-1 * (9/10).

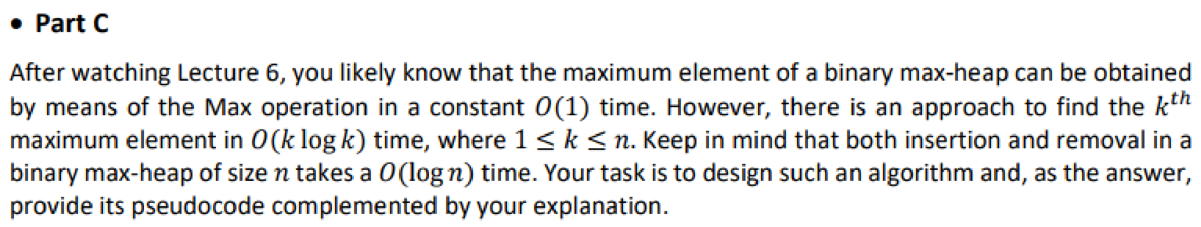
Consequences of Bias:

1. Higher Average Height: Elements are more likely to have taller towers.

Compared to a standard skip list (expected height 2), the biased coin results in a significantly lower average element height.

2. Reduced space overhead: Fewer layers mean fewer pointers per element, reducing memory usage.

3. Search efficiency may decrease: The number of high-level nodes decreases, resulting in lower-level nodes to be traversed during search, and the time complexity may approach O(n).



- **Part C**

After watching Lecture 6, you likely know that the maximum element of a binary max-heap can be obtained by means of the Max operation in a constant $O(1)$ time. However, there is an approach to find the k^{th} maximum element in $O(k \log k)$ time, where $1 \leq k \leq n$. Keep in mind that both insertion and removal in a binary max-heap of size n takes a $O(\log n)$ time. Your task is to design such an algorithm and, as the answer, provide its pseudocode complemented by your explanation.

A:

Data structure: The main heap is stored in an array, with indexes starting from 1, satisfying: The left child of node i is 2i. The right child of node i is 2i + 1.

Use the auxiliary maximum heap (AuxHeap) to store tuples (value, index), sorted by value.

Initialization: Insert the root node of the main heap (index 1, value heap[1]) into AuxHeap.

```
for i = 1 to k:

 if AuxHeap is empty:

break

(current_val, current_idx) = AuxHeap.extract_max()

result.append(current_val)
```

Loop k times:

a. Extract the maximum value: Extract the current maximum value from AuxHeap.

b. Insert child nodes: Calculate the left child node index 2i and the right child node index 2i + 1. If the index does not exceed the heap size n, insert the value and index of the child node into AuxHeap.

Bounds check: Before inserting a child node, verify whether the index is valid.

```
// insert left（if exist）
left_idx = 2 * current_idx
if left_idx <= n:
AuxHeap.insert( (heap[left_idx], left_idx) )


// insert right（if exist）
right_idx = 2 * current_idx + 1
 if right_idx <= n:
AuxHeap.insert( (heap[right_idx], right_idx) )
```

```
return result[k-1]
```

Each operation: extract_max and insert has a time complexity of O(log K), because the size of the auxiliary heap is at most O(k).

Total time: O(K log K).

References:

1. https://www.geeksforgeeks.org/segment-tree-efficient-implementation/
2. https://mitpress.mit.edu/9780262530910/introduction-to-algorithms/