CSE 101 Homework 8

Brian Masse, Taira Sakamoto, Emily Xie, Annabelle Coles

December 5, 2025

1. **Verifying Heap**

A *3-ary heap* is like a binary heap, except that every internal vertex has three children: *left*, *middle* and *right*. We will use v.left, v.mid, and v.right to denote the three children. We let v.value denote the value stored at v. We consider only the case of complete trees, so all three sub-trees rooted at the three children will have the same size.

Here's an algorithm that verifies whether a balanced complete 3-ary tree has the min-heap property.

IsHeap(r : root of 3-ary heap H)

- IF r is a leaf: return `true`
- IF r.value $>$ r.left.value return `false`
- IF r.value $>$ r.mid.value return `false`
- IF r.value $>$ r.right.value return `false`
- IF IsHeap[r.left] == `false` return `false`
- IF IsHeap[r.mid] == `false` return `false`
- IF IsHeap[r.right] == `false` return `false`
- Return `true`

(a) Give a recurrence for the time taken by IsHeap in terms of the total size, n, of the heap rooted at r. Explain your answer.

The recurrence is:

$$T(n) = 3T(n/3) + O(1)$$

The algorithm performs a constant amount of work at the current node (checking if it is a leaf and performing 3 comparisons of values). It then makes recursive calls on the three children. Since the tree is a complete 3-ary tree, the problem size n is split evenly among the three subtrees. Therefore, there are 3 recursive calls, each on an input of size approximately $n/3$.

(b) Give a time analysis for IsHeap by solving this recurrence, up to order. If you use the Master Theorem, be sure to give the values of the three parameters used, and the case it falls under.

Use the Master Theorem of the form $T(n) = aT(n/b) + f(n)$.

The parameters are:

- $a = 3$ (number of recursive subproblems)
- $b = 3$ (factor by which the input size is reduced)
- $f(n) = O(1)$ (cost of work outside the recursive calls)

Compare $f(n)$ to $n^{\log_b a}$:

$$\log_b a = \log_3 3 = 1$$

Therefore, comparing $f(n) = 1$ to n^1.

Since $f(n) = O(n^{1-\epsilon})$ (for any $0 < \epsilon \leq 1$), this falls under case 1 of the Master Theorem.

The complexity is:

$$T(n) = \Theta(n^{\log_b a}) = \Theta(n)$$

2. **Median of Sorted Lists**

The following algorithm finds the median (nth smallest element) of two sorted lists of distinct integers, $A[1...n]$ and $B[1..n]$. For simplicity, we assume $n = 2^k$ is a power of 2.

MedianSorted(A[1..n],B[1..n])

(a) IF $n = 1$ THEN Return $min(A[1], B[1])$.

(b) IF $A[n/2] < B[n/2 + 1]$

(c) THEN Return $MedianSorted(A[n/2 + 1..n], B[1..n/2])$.

(d) IF $A[n/2] > B[n/2 + 1]$

(e) THEN Return $MedianSorted(A[1..n/2], B[n/2 + 1..n])$.

(f) IF $A[n/2] = B[n/2 + 1]$

(g) THEN Return $A[n/2]$.

Questions

(a) Give a recurrence for the worst-case time taken by MedianSorted in terms of n. Explain your answer.

(b) Give a time analysis for MedianSorted by solving this recurrence, up to order. If you use the Master Theorem, be sure to give the values of the three parameters used, and the case it falls under.

3. **Taxi Stand**

At a taxi stand, there is a line of customers $C_1, ..C_n$ and a line of taxis, $T_1, ...T_m$. Since taxis are licensed to drive in some towns but not others, each can serve some of the customers but not others. We need to assign each customer to a taxi, and the assigned taxis need to be in order, but not all taxis need to be assigned. We want to minimize the position of the last taxi to be assigned.

For example, if we have three customers A, B, C and four taxis, $1, 2, 3, 4$ where 1 can serve A and C, 2 can serve A and C, 3 can serve B and C, and 4 can serve B and C, we could assign A to taxi 2, B to taxi 3 and C to taxi 4.

(a) Fill out the rest of the problem specification with mathematical expressions clearly defining the constraints and objective:

- Instance: Bipartite graph G whose left vertices represent customers and right vertices represent taxis, and an edge (C, t) represents taxi t being able to serve customer C, with orderings $C_1...C_n$ and $t_1..t_m$ on the two sets of vertices, with $m > n$.
- Solution Format: A list of taxis $i_1, i_2, ..i_n$, with taxi i_j assigned to customer j.
- Constraints: Each customer must be assigned to a taxi that can drive them to their destination, i.e, $\exists$ an edge (j, i_j) for every $1 \leq j \leq n$. And each customer must be assigned to a later taxi than the previous customer, i.e., $i_j < i_{j+1}$
- Objective: Minimize the position of the last taxi, i_j

(b) Give a high level greedy strategy that solves this problem.

For custom C_k choose the lowest taxi T_j such that $\exists$ an edge $(C_k, T_j) \in G$ and the previously selected taxi, $i_{k-1} < T_j$. For the first customer, C_1, assign them the lowest taxi such that (C_1, T_j) exists.

(c) Illustrate your strategy on the example given in the problem.

k	i_1	i_2	i_3
1	1	-	-
1	1	3	-
1	1	3	4

(d) Prove your strategy finds the optimal solution, using any technique we've learned. You can state and prove the main lemmas rather than give the complete proof, such as stating and proving a modify-the-solution lemma but skipping the induction argument.

4. **Maximizing inner product**

Say we are given two sequences of integers $x_1, ..x_n; y_1, ..y_m$ with $m \geq n$. We want to find a subsequence of $y_1, ..y_m$ of length n , $y_{i_1}, y_{i_2}, ..y_{i_n}$ with $1 \leq i_1 < i_2 < ... < i_n \leq m$ that maximizes the *inner product* $\sum_{1 \leq j \leq n} x_j * y_{i_j}$.

For example, if x is $3, 2, 4$ and y is $5, 6, 1, 2, 8$, the optimal subsequence is $5, 6, 8$ giving objective $3 * 5 + 2 * 6 + 4 * 8 = 59$.

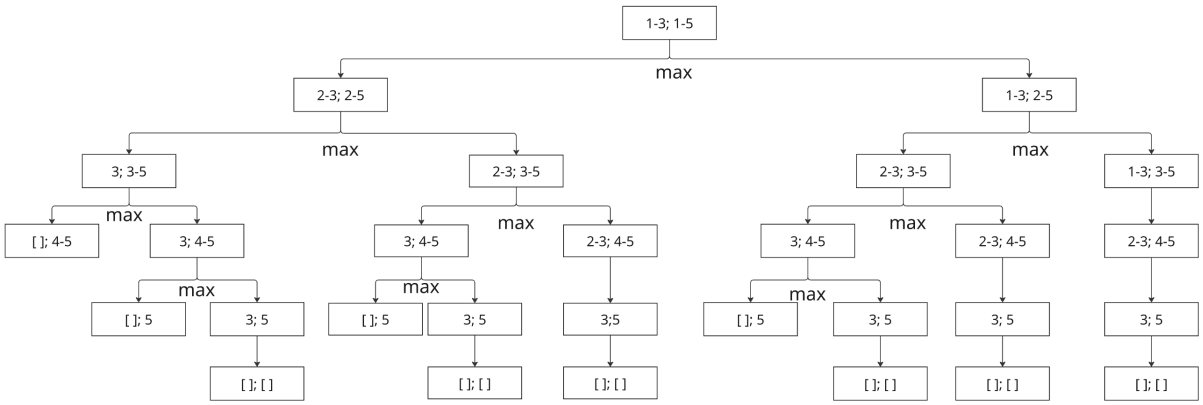
Here's a recursive algorithm to find the maximum inner product with a subsequence:

MIPWS($x_1, ..x_n; y_1, ..y_m$)

- IF $n = 0$ THEN return 0
- IF $n = m$ THEN return $x1 * y_1 + MIPWS(x_2, ..x_n; y_2..y_m)$
- Return $max(x1 * y_1 + MIPWS(x_2, ..x_n; y_2..y_m), MIPWS(x_1..x_n; y_2...y_m)$.

Questions

(a) Show the tree of recursive calls this algorithm makes on the above example. (You can either draw the tree, or list the recursive calls, using indentation to indicate nesting. You can use abbreviations like $2-4, 2-5$ to mean calling the algorithm on $x_2..x_4; y_2..y_5$)



(b) Give a recurrence for the total number of recursive calls this algorithm makes, in terms of $n' = n + m$.

- $T(n) = T(n' - 2) + T(n' - 1) + O(1)$

In the rest of this part, you will give a dynamic programming algorithm for this problem. Your algorithm can be based on the above recursion, or some other case analysis.

(c) Describe precisely the sub-problems that your dynamic programming algorithm solves. Define an array or matrix to hold these answers.

The three subproblems are the function calls in lines two and three:

- $MIPWS(x_2, ..x_n; y_2..y_m)$

- $(x_1 * y_1 + MIPWS(x_2, ..x_n; y_2..y_m)$
- $MIPWS(x_1..x_n; y_2...y_m)$

We will use a 2D matrix called MIP to store the outputs of the subproblems. The matrix will have dimensions $(n+1, m+1)$.

(d) Describe the base cases in terms of your array or matrix.

- The base case is when $n == 0$ when the x array is empty. This refers to the $n+1$ row in the DP output matrix.

(e) Give a formula or pseudo-code saying how you compute larger sub-problems in terms of smaller sub-problems.

- Add previous subproblem output to the multiplication of the current elements and compare to other subproblem outputs.
- Formula: $MIP[i+1][j+1] \leftarrow max(x[i] * y[j] + MIP[i][j], MIP[i][j+1], MIP[i+1][j]))$

(f) Explain the case-analysis that justifies the previous step.

- In the base case, when the array is empty we get zero, which is a given.
- By calculating the max of the subproblems, it is ensured that we find the maximum inner product.
- Since we loop through all of the x-array and the y-array, we check all possible dot products of the elements.
- Since we have already solved the subproblems for i, j and we initialize the MIP matrix with zeros, we can always calculate $MIP[i+1][j+1]$.

(g) Assemble the parts above into a dynamic programming algorithm, described in pseudocode. Be sure to specify the order in which sub-problems are solved, and what the final output is. s Replacing function calls with outputs from matrix:

- $MIP \leftarrow [[0] * (nm1)] * (n+1)$
- FOR $i = 1$ to n do:
- FOR $j = 1$ to m do:
- $MIP[i+1][j+1] \leftarrow max(x[i] * y[j] + MIP[i][j], MIP[i][j+1], MIP[i+1][j])$
- Return $MIP[n][m]$

We solve the subproblems in increasing order from $i = 1$ to $i = n$. We fill the later $MIP[x_{i+1}][y_{j+1}]$ matrix entries using the outputs from the earlier subproblems where we found $MIP[x_i][y_j]$. The final output is the value stored at position $MIP[n][m]$, this is the maximum inner product.

(h) Give a time analysis for your dynamic programming algorithm in terms of n.

- For each index in the DP output array we need to do one addition and comparison. There are two nested for loops that will take $O(n \cdot m)$. Given $m \geq n$, in the worst case $m == n$ so $O(n \cdot m) = O(n^2)$

(i) Show the array or matrix your algorithm produces on the example above.

0	0	0	0	0	0
0	15	18	18	18	24
0	15	27	27	27	34
0	20	39	39	39	59