CS 170: Homework #2

Due on January 26, 2026 at 3:10pm

L. Chen and U. Vazirani, Spring 2026

Zachary Brandt
Collaborators: Michael Murphy (3038760401)

1 The Magical Keys and Locks (Solo Question; 10 points)

In the ancient kingdom of Keylom, there are n magical keys and n enchanted locks, such that each key (and each lock) has a unique size between 1 and n. The unique key of size i opens the unique lock of size i. You cannot directly compare two keys with each other, nor can you compare two locks. The *only* thing you can do is insert any key into any lock and see whether it is too small for the lock's keyhole, too big for it, or if it is a perfect fit.

After a lively lantern festival in the village square, the townsfolk accidentally jumbled all the keys and locks into a single array of $2n$ items, in completely arbitrary order. No one remembers which key matches which lock. Your task is to match each key back to its corresponding lock.

(a) (5 points) Design a *randomized* algorithm that takes this array of $2n$ items as input and returns an array of n key-lock pairs, each of which is a perfect fit. Explain why your algorithm always outputs the correct answer.

(b) (5 points) Give an intuitive argument that your algorithm from part (a) usually runs in time $O(n \log n)$.

You may make the simplifying assumption that the keys and locks are truly magical, in the sense that if you choose a random key from any set of m keys, you always get the key of median (i.e. $\lfloor m/2 \rfloor$th largest) size.

Problem 1 continued on next page...

Part A

Design a *randomized* algorithm that takes this array of $2n$ items as input and returns an array of n key-lock pairs, each of which is a perfect fit. Explain why your algorithm always outputs the correct answer.

Solution

Below is my algorithm description of a randomized algorithm MATCH(ITEMS) presented in pseudocode.

```
1: function MATCH(items)
2:     if items is empty then
3:         return empty list
4:     end if
5:     p ← random item from items
6:     Find the matching pair (k, ℓ) for p by testing p against all items
7:     left ← empty list, right ← empty list
8:     for each item x in items (excluding k and ℓ) do
9:         if x is a key then
10:            Compare x to lock ℓ
11:            Add x to left if smaller, right if larger
12:        else if x is a lock then
13:            Compare x to key k
14:            Add x to left if smaller, right if larger
15:        end if
16:    end for
17:    return MATCH(left) + [(k, ℓ)] + MATCH(right)
18: end function
```

The algorithm MATCH(ITEMS) produces the correct result on an arbitarily ordered array of $2n$ items because each call identifies one correct key-lock pair and partitions the remaining elements without directly comparing two keys with each other or two locks. The subproblems remain valid since there will be an equal number of keys and locks in each subarray with sizes stricly greater or smaller than the pivot key-lock pair.

Problem 1 continued on next page...

Part B

Give an intuitive argument that your algorithm from part (a) usually runs in time $O(n \log n)$.

You may make the simplifying assumption that the keys and locks are truly magical, in the sense that if you choose a random key from any set of m keys, you always get the key of median (i.e. $\lfloor m/2 \rfloor$th largest) size.

Solution

If the key of median size is always chosen when choosing at random from the list of items, then the following recurrence relation describes the runtime of the algorithm:

$$T(n) = 2T(\tfrac{n}{2}) + O(n)$$

At each level of recursion $O(n)$ work is done in finding the corresponding key or lock to the randomly selected median lock or key, respectively. The size of the left and right subarrays partitioned from the original array will then be approximately half of the size of the original items array. The algorithm will then recurse on both on halves and do $O(\frac{n}{2})$ work. Following from Master Theorem, the runtime is $O(n \log n)$ since $\log_b a = d$ where $a = 2$, $b = 2$, and $d = 1$.

2 Maximum Subarray Sum (10 points)

Given an array A of n integers, the maximum subarray sum is the largest sum of any contiguous subarray of A (including the empty subarray). In other words, the maximum subarray sum is:

$$\max_{i \le j} \sum_{k=i}^{j} A[k]$$

For example, the maximum subarray sum of $[-2, 1, -3, 4, -1, 2, 1, -5, 4]$ is 6, the sum of the contiguous subarray $[4, -1, 2, 1]$.

Design an $O(n \log n)$-time divide-and-conquer algorithm that finds the maximum subarray sum. Briefly explain why your algorithm is correct and justify its running time.

Solution

Below is my algorithm description of a divide-and-conquer algorithm MaxSubarraySum(A) presented in pseudocode.

```
1:  function MaxSubarraySum(A)
2:      if A is empty then
3:          return 0
4:      end if
5:      if A has one element then
6:          return max(0, A[0])
7:      end if
8:      mid ← ⌊|A|/2⌋
9:      L ← A[0 ... mid − 1], R ← A[mid ... |A| − 1]
10:     leftMax ← MaxSubarraySum(L)
11:     rightMax ← MaxSubarraySum(R)
12:
13:     maxStart ← −∞, sum ← 0
14:     for i from |L| − 1 down to 0 do
15:         sum ← sum + L[i]
16:         maxStart ← max(maxStart, sum)
17:     end for
18:
19:     maxEnd ← −∞, sum ← 0
20:     for i from 0 to |R| − 1 do
21:         sum ← sum + R[i]
22:         maxEnd ← max(maxEnd, sum)
23:     end for
24:
25:     return max(leftMax, rightMax, maxStart + maxEnd)
26: end function
```

The algorithm MaxSubarraySum(A) produces the correct result on an array of n integers because each call selects the maximum over the maximum subarray sums for the left and right halves of the n integers and the maximum subarray sum of an array that crosses over the two halves. To find the crossover subarray sum, the algorithm scans right to left for the left half to find the maximum contribution from that half, and vice versa for the right half. The subproblems build up and select maximum sums from subarrays that are entirely in each half or crossover.

Problem 2 continued on next page...

The following recurrence relation describes the runtime of the algorithm:

$$T(n) = 2T(\tfrac{n}{2}) + O(n)$$

At each level of recursion finding the crossover subarray sum executes in $O(n)$ time, since the for loops iterate over the entire length of A once. The algorithm partitions A into left and right halves on each recursion. Following from Master Theorem, the runtime is $O(n \log n)$ since $\log_b a = d$ where $a = 2$, $b = 2$, and $d = 1$.

3 Monotone matrices (10 points)

A m-by-n matrix A is *monotone* if $n \geq m$, each row of A has no duplicate entries, and it has the following property: if the minimum of row i is located at column j_i, then $j_1 < j_2 < j_3 \dots j_m$. For example, the following 3-by-6 matrix is monotone (the minimum of each row is bolded), since $j_1 = 1, j_2 = 3, j_3 = 6$:

$$\begin{bmatrix} \mathbf{1} & 3 & 4 & 6 & 5 & 2 \\ 7 & 3 & \mathbf{2} & 5 & 6 & 4 \\ 7 & 9 & 6 & 3 & 10 & \mathbf{0} \end{bmatrix}$$

Give an efficient (i.e., significantly better than $O(mn)$-time) algorithm that finds the minimum in each row of an m-by-n monotone matrix A.

Bound the running time of your algorithm. Note: you might find it easier to bound the work per level of recursion directly rather than writing a formal recurrence relation.

Solution

Algorithm (Right-Scan with Column Pointer):

The key insight is to exploit the monotone property: since minimum column indices are strictly increasing ($j_1 < j_2 < \ldots < j_m$), once we find the minimum of row i at column j_i, we know the minimum of row $i+1$ must be at some column $j_{i+1} > j_i$. We never need to re-examine columns we've already passed.

1. Initialize column pointer $c = 0$
2. For each row $r = 0$ to $m - 1$:
   (a) Find the minimum value in $A[r][c..n-1]$ (from column c to end of row)
   (b) Let j be the column where this minimum is located
   (c) Record $A[r][j]$ as the minimum of row r
   (d) Set $c = j + 1$ (next row's minimum must be to the right)

Correctness: The monotone property guarantees that $j_1 < j_2 < \ldots < j_m$. Therefore, after finding row r's minimum at column j_r, we can safely start searching row $r+1$ from column $j_r + 1$, as its minimum cannot be at or before column j_r. Each row's minimum is correctly identified by scanning the allowed range.

Running Time Analysis:

The column pointer c starts at 0 and can only increase. Each time we find a minimum, we potentially move c rightward. Across all m rows:

- c moves from 0 to at most $n - 1$ (at most n positions total)
- Each row scan examines elements from the current c to the minimum's position
- Total elements examined: $O(n)$ (each column visited at most once by c)
- Processing m rows: $O(m)$

Total operations: $O(m + n)$

This is significantly better than the naive $O(mn)$ approach. For example, if $m = n$, we achieve $O(n)$ instead of $O(n^2)$.

Detailed Example Trace:

Given matrix:

$$\begin{bmatrix} \mathbf{1} & 3 & 4 & 6 & 5 & 2 \\ 7 & 3 & \mathbf{2} & 5 & 6 & 4 \\ 7 & 9 & 6 & 3 & 10 & \mathbf{0} \end{bmatrix}$$

- **Row 0:** $c = 0$. Scan columns $[0, 1, 2, 3, 4, 5]$: values are $[1, 3, 4, 6, 5, 2]$. Minimum is **1** at column $j_1 = 0$. Set $c = 1$.
- **Row 1:** $c = 1$. Scan columns $[1, 2, 3, 4, 5]$: values are $[3, 2, 5, 6, 4]$. Minimum is **2** at column $j_2 = 2$. Set $c = 3$.

  (Note: We skipped column 0 because monotone property guarantees $j_2 > j_1 = 0$)
- **Row 2:** $c = 3$. Scan columns $[3, 4, 5]$: values are $[3, 10, 0]$. Minimum is **0** at column $j_3 = 5$.

  (Note: We skipped columns 0-2 because monotone property guarantees $j_3 > j_2 = 2$)

Total elements examined: $6+5+3 = 14$ (compared to all 18 elements in naive approach). More importantly, we examined each column at most once: columns are visited as c increases from 0 through 5.

4 Werewolves (10 points)

You are playing a party game with n other friends, who each play either as a werewolf or a villager. Your friends know who is a werewolf, but all your friends do. But you know that there are more villagers than there are werewolves. And you also know that villagers always tell the truth, while werewolves can either lie or tell the truth.

Your goal is to identify one player who is definitely a villager. Your elementary query is to pair up two people and ask each whether the other is a villager or werewolf. Your algorithm should work regardless of the behavior of the werewolves.

(a) (5 points) For a given person x, devise an algorithm that returns whether or not x is a villager using $O(n)$ queries.

(b) (5 points) Show how to find a villager in $O(n \log n)$ queries using a divide-and-conquer algorithm.

Problem 4 continued on next page...

Part A

For a given person x, devise an algorithm that returns whether or not x is a villager using $O(n)$ queries.

Solution

A simple algorithm that returns whether or not a given person x is a villager or not is to make $n-1$ queries with pairs consisting of x and all the other players. If more than $\frac{n}{2}$ players say the player x is a villager, then the algorithm returns in the affirmative, i.e., that player x is indeed a villager. Otherwise, the algorithm returns with the opposite result, i.e., that player x is a werewolf.

The algorithm produces the correct result on a given person x because villagers will always tell the truth and there are more villagers than there are werewolves. This is because werewolves may or may not tell the truth. With this algorithm, even if all werewolves were to lie and say that x is a villager when they really aren't, all the villagers would tell the truth and answer in the negative.

The algorithm executes in $O(n)$ time since $n-1$ queries need to be performed before making a decision.

Problem 4 continued on next page...

Part B

Show how to find a villager in $O(n \log n)$ queries using a divide-and-conquer algorithm.

Solution

Below is my algorithm description of a divide-and-conquer algorithm FindVillager(Friends) presented in pseudocode.

```
1: function FindVillager(Friends)
2:     if |Friends| = 1 then
3:         return Friends[0]
4:     end if
5:     mid ← ⌊|Friends|/2⌋
6:     Left ← Friends[0 ... mid − 1]
7:     Right ← Friends[mid ... |Friends| − 1]
8:     x ← FindVillager(Left)
9:     y ← FindVillager(Right)
10:    if IsVillager(x, Friends) then
11:        return x
12:    else
13:        return y
14:    end if
15: end function
```

The algorithm FindVillager(Friends) uses the divide-and-conquer approach by splitting the group into two halves and recursively finding candidate villagers x and y from each half. It then uses the procedure from part (a), IsVillager(x, Friends), to check if x or y is a villager by querying all other friends in the original group and returning the one who is identified as a villager.

The algorithm correctly returns a villager from any group where the majority are villagers. When the friend group is partitioned into two halves, at least one half must contain more villagers than werewolves. This is because if both halves had more werewolves than villagers, then the entire friend group would have more werewolves than villagers, contradicting the starting assumption. Therefore, at each level of recursion, the majority is preserved in at least one of the two subgroups.

When the algorithm reaches the base case of a single person in a group, that person is returned. Working back up through the recursion, each recursive call returns some candidate from its subgroup. At least one of the two candidates x and y must be an actual villager, since at least one of the two subgroups had a villager majority and thus returned a villager. Using IsVillager(x, Friends) from part (a) ensures the villager is returned.

The runtime of the algorithm is in $O(n \log n)$. The following recurrence relation defintes this runtime:

$$T(n) = 2T\left(\tfrac{n}{2}\right) + O(n)$$

At each level of recursion, the algorithm makes two recursive calls on problems of size $\frac{n}{2}$, and then performs queries in $O(n)$ to verify which candidate is a villager using the procedure from part (a). By the Master Theorem, with $a = 2$, $b = 2$, and $d = 1$, $\log_b a = 1 = d$, and therefore $O(n \log n)$ is the runtime.

5 Shaving Logs or Coding (10 points)

For full credit, you should do **one of the following three** questions. (You may solve the others for fun if you want!)

(a) Find a $O(n)$ time algorithm for maximum subarray sum (Question 2), if your solution above was slower.

(b) Find a $O(n)$ query algorithm for werewolves (Question 4), if your solution above was slower.

(c) Implement the quickselect algorithm in a python jupyter notebook called `quick_select.ipynb`. There are two ways that you can access the notebook and complete the problems:

(a) **On Local Machine**: `git clone` (or if you already have it, `git pull`) from the coding homework repo,

`https://github.com/Berkeley-CS170/cs170-sp26-coding`

and navigate to the `hw02` folder. Refer to the `README.md` for local setup instructions.

(b) **On Datahub**: Click on `https://github.com/Berkeley-CS170/cs170-sp26-coding` and navigate to the `hw02` folder if you prefer to complete this question on Berkeley DataHub.

Notes:

- *Submission Instructions*: Please run the last cell to download a zip file and submit it to the gradescope assignment titled "HW02 (Coding)".
- *OH/HWP Instructions*: Designated coding course staff will provide conceptual and debugging help during office hours.
- *Academic Honesty Guideline*: We realize that code for some of the algorithms we ask you to implement may be readily available online, but we strongly encourage you to not directly copy code from these sources. Instead, try to refer to the resources mentioned in the notebook and come up with code yourself. That being said, we **do acknowledge** that there may not be many different ways to code up particular algorithms and that your solution may be similar to other solutions available online.