

CS 170: Homework #2

Due on January 26, 2026 at 3:10pm

L. Chen and U. Vazirani, Spring 2026

Zachary Brandt

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.

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:    $p \leftarrow$  random item from items
6:   Find the matching pair  $(k, \ell)$  for  $p$  by testing  $p$  against all items
7:    $left \leftarrow$  empty list,  $right \leftarrow$  empty list
8:   for each item  $x$  in items (excluding  $k$  and  $\ell$ ) do
9:     if  $x$  is a key then
10:       Compare  $x$  to lock  $\ell$ 
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( $left$ ) +  $[(k, \ell)]$  + MATCH( $right$ )
18: end function

```

The algorithm MATCH(ITEMS) produces the correct result on an arbitrarily 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 strictly greater or smaller than the pivot key-lock pair.

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:

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

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 halves and do $O\left(\frac{n}{2}\right)$ 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:

$$\max_{i \leq j} \sum_{\ell=i}^j A[\ell]$$

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:    $mid \leftarrow \lfloor |A|/2 \rfloor$ 
9:    $L \leftarrow A[0 \dots mid - 1], R \leftarrow A[mid \dots |A| - 1]$ 
10:   $leftMax \leftarrow \text{MAXSUBARRAYSUM}(L)$ 
11:   $rightMax \leftarrow \text{MAXSUBARRAYSUM}(R)$ 
12:
13:   $maxStart \leftarrow -\infty, sum \leftarrow 0$ 
14:  for  $i$  from  $|L| - 1$  down to 0 do
15:     $sum \leftarrow sum + L[i]$ 
16:     $maxStart \leftarrow \max(maxStart, sum)$ 
17:  end for
18:
19:   $maxEnd \leftarrow -\infty, sum \leftarrow 0$ 
20:  for  $i$  from 0 to  $|R| - 1$  do
21:     $sum \leftarrow sum + R[i]$ 
22:     $maxEnd \leftarrow \max(maxEnd, sum)$ 
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.

The following recurrence relation describes the runtime of the algorithm:

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

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.

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.

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.