CS 161
Spring 2017

Problem Set 1
Due: April 14, 2017, 3pm

Please answer each of the following problems. Refer to the course webpage for the **collaboration policy,** as well as for **helpful advice** for how to write up your solutions.

Note on notation: On a few problems in this homework set, we use "big-O" notation to state the problem. We will see this in lecture on Monday, but you should still be able to get a start on the homework over the weekend. For all of the problems where O appears, it is fine to to take the definition of "$O(n)$" to mean "grows (at most) roughly linearly in n:" for example, $100 \cdot n$ grows roughly linearly with n, so $100 \cdot n = O(n)$. Similarly, $100 \cdot n + 100 = O(n)$. But n^2 does *not* grow roughly linearly with n, it grows much faster, so $n^2 \neq O(n)$. And $\sqrt{n}$ grows much more slowly than n, so we would still say $\sqrt{n} = O(n)$.

1. **What do you want from this course?** (4 points) What skills do you hope to learn, what topics do you think well cover that will stick with you five, or ten years down the road? Write *at least three sentences* describing how you expect to benefit from taking this class. The reason for this question is twofold. First, it will help me understand what you want. Second, keep your answer to this question in mind throughout the quarter as motivation if the going gets tough!

   SOLUTION: We don't get to answer this one for you!

2. **New friends.** (8 points) Each of n users spends some time on a social media site. For each $i = 1, \ldots, n$, user i enters the site at time a_i and leaves at time $b_i \geq a_i$. You are interested in the question: how many distinct pairs of users are on the site at the same time? (Here, the pair (i, j) is the same as the pair (j, i)).

   Example: Suppose there are 5 users with the following entering and leaving times:

User	Enter time	Leave time
1	1	4
2	2	5
3	7	8
4	9	10
5	6	10

   Then, the number of distinct pairs of users who are on the site at the same time is three: these pairs are $(1, 2)$, $(4, 5)$, $(3, 5)$.

   (a) (3 pts) Given input $(a_1, b_1), (a_2, b_2), \ldots, (a_n, b_n)$ as above, there is a straightforward algorithm that takes about[1] n^2 time to compute the number of pairs of users who are on the site at the same time. Give this algorithm and explain why it takes time about n^2.

[1] Formally, "about" here means $\Theta(n^2)$, but for now any informal understanding you have of this is fine.

SOLUTION: Our algorithm will run as follows: initialize variable $count$ to 0. For every user i, we check every user $j \neq i$:

- If $a_i \leq a_j \leq b_i$ or $a_j \leq a_i \leq b_j$, then user i and user j are on the site at the same time. Increment $count$ by 1.

Finally, we return $count/2$, which counts the number of distinct pairs of users who are ever on the site at the same time. We divide $count$ by 2 because we double counted each pair.
Running time analysis: For each user $i = 1, ..., n$, we iterate over all other users ($\Theta(n)$ of them) to check for the above inequality which can be done in constant time. Therefore, the algorithm runs in $\Theta(n^2)$.

(b) (5 pts) Give an $O(n \log(n))$-time algorithm to do the same task and analyze its running time. (**Hint:** consider sorting relevant events by time).

SOLUTION: Initialize variable $count$ and variable $usersOnsite$ to 0. Note that $usersOnSite$ keeps track of the number of users on the site when we examine at time t_i. First, we produce a combined list l of entry and exit times. l has $2n$ tuples. The first element in each tuple is the entry/exit time and the second element is a binary indicating "entry" or "exit". Thus, user i can be split into $(a_i, \text{“}enter''\text{)}$ and $(b_i, \text{“}exit''\text{)}$. Next, we sort list l by the first element in each tuple using MergeSort. For each tuple p in the sorted list l, we check:

- If $p[1]$ is "enter":
  - If $usersOnSite \geq 1$, increment $count$ by the value of $usersOnSite$.
  - Increment $usersOnSite$ by 1.
- If $p[1]$ is "exit", decrement $usersOnSite$ by 1.

Return count.
Running time analysis: Initialize list l takes $\Theta(n)$ time, since we are iterating over all user entry and exit times, generating 2 tuples for each user. Sorting the list using MergeSort takes $\Theta(n \log(n))$ time. Iterating through the sorted list l takes $\Theta(n)$ time, since there are a total of $2n$ tuples and each iteration takes constant time to execute. Overall, the algorithm takes $\Theta(n \log(n))$ time.

3. **Proof of correctness.** (6 points) Consider the following algorithm that is supposed to sort an array of integers. Please provide a proof that this algorithm is correct.

```
# Sorts an array of integers.
Sort(array A):
    for i = 1 to A.length:
        minIndex = i
        for j = i + 1 to A.length:
            if A[j] < A[minIndex]:
                minIndex = j
```

```
        Swap(A[i], A[minIndex])

# Swaps two elements of the array.  You may assume this function is correct.
Swap(array A, int x, int y):
    tmp = A[x]
    A[x] = A[y]
    A[y] = tmp
```

SOLUTION: We use two loop invariants:

Outer loop: At the beginning of the ith iteration, elements 1 through $i-1$ are sorted in increasing order, and all of the elements i through $A.length$ are at least as large as all of the elements in $1..i-1$.

Inner loop: At the beginning of the jth iteration, minIndex is the index of the smallest element in the range $[i, j-1]$.

Outer loop

Initialization: At the beginning of the first iteration, there are no elements in the range $[1, 0]$, so they are vacuously sorted in increasing order. The second clause of the loop invariant is also vacuously true.

Maintenance: Suppose the invariant holds before iteration i. We show that it holds before iteration $i+1$. If the inner loop invariant holds, then at the end of the inner loop, minIndex is the index of the smallest element in the array range $[i, A.length]$. Then the Swap command ensures that the smallest element in the range $[i, A.length]$ gets placed in A[i]. By the previous invocation of the loop invariant, we know that $A[1] \leq A[2] \leq \cdots \leq A[i-1]$, and $A[i]$ is at least as large as $A[i-1]$. Furthermore, from what we are arguing now, we know $A[i]$ is at least as small as all the elements in $A[i+1], \ldots, A[A.length]$. This maintains the invariant for the next iteration.

Termination: At the beginning of the $(A.length+1)$st iteration, elements 1 through $A.length$ are sorted in increasing order, so the array is sorted.

Inner loop

Initialization: At the beginning of iteration $i+1$, minIndex is i, which is the index of the smallest element in the range $[i, i]$.

Maintenance: Suppose the invariant holds prior to iteration j. We show that it holds prior to iteration $j+1$. At the beginning of the jth iteration, minIndex is the index of the smallest element in the range $[i, j-1]$. If $A[j] < A[minIndex]$, then j is the index of the smallest element in the range $[i, j]$, so setting minIndex = j is correct. If $A[j]$ is not less than $A[minIndex]$, then $minIndex$ is the index of the smallest element in the range $[i, j]$, so it is correct to leave minIndex alone. This maintains the invariant for the next iteration.

Termination: At the beginning of the $(A.length+1)$st iteration, minIndex is the index of the smallest element in the range $[i, A.length]$, which is the condition we needed for our proof of the outer loop invariant.

4. **Needlessly complicating the issue.** (12 points)

(a) (3pts) Give a linear-time (that is, an $O(n)$-time) algorithm for finding the minimum of n values (which are not necessarily sorted).

SOLUTION: In order to find the minimum of n values, we can iterate through all the values, keeping track of the minimum we've seen so far. Each time we look at a new value, we compare it to the minimum we've seen. If the new value is lower, we replace the minimum we've seen so far with this new value. We can also write this in pseudocode:

Algorithm 1: `simpleFindMinimum`

Input: List $A = [a_1, \ldots, a_n]$ of n items
Output: $\min_i\{a_1, \ldots, a_n\}$
$min = \infty$
for $a_i \in A$ **do**
  if $a_i < min$ **then**
    $min = a_i$
return min

(b) (3pts) Argue that any algorithm that finds the minimum of n items must take $\Omega(n)$ time in the worst case.

SOLUTION: In order to find the minimum of n values, we must look at every value. If there were some value we decided not to look at, that value could contain the minimum and we could end up with the wrong answer. This means we must do some work for every single one of the n values, which tells us our algorithm must take $\Omega(n)$ time in the worst case.

Now consider the following recursive algorithm to find the minimum of a set of n items.

Algorithm 2: `findMinimum`

Input: List $A = [a_1, \ldots, a_n]$ of n items
Output: $\min_i\{a_1, \ldots, a_n\}$
if *n=1* **then**
  return _____
$A_1 = A[0 : n/2]$
$A_2 = A[n/2 : n]$
return $\min($`findMinimum`$(A_1),$ `findMinimum`$(A_2)\ )$

(c) (3pts) Fill in the blank in the pseudo-code: what should the algorithm return in the base case? Briefly argue that the algorithm is correct with your choice.

SOLUTION: We can fill in the blank:

Algorithm 3: `findMinimum`

Input: List $A = [a_1, \ldots, a_n]$ of n items
Output: $\min_i\{a_1, \ldots, a_n\}$
if *n=1* **then**
  return -- a_1 --
$A_1 = A[1 : \lfloor n/2 \rfloor]$
$A_2 = A[\lfloor n/2 \rfloor + 1 : n]$
return $\min($`findMinimum`(A_1), `findMinimum`(A_2) $)$

We must now argue that this algorithm is correct. We can do this by strong induction. First, as a base case, if $n = 1$, then there is only a single item and that item must be the minimum. Thus our algorithm returns the correct value in this case.

Now, let us assume that for all $m \leq n$ the algorithm works on arrays of length m. Then we know that the two internal calls we make to **findMinimum** will return the correct minimum. The minimum of our full array is the smaller of the minimums on each side, which is exactly what we compute and return. Thus the algorithm is correct.

(d) (3pts) Analyze the running time of this recursive algorithm. How does it compare to your solution in part (a)?

SOLUTION: We can analyze the runtime of this recursive algorithm by imagining a tree! At the top of the tree is the original call, and then we split into two calls each half the size, and then each of those splits into two more calls. At the ith level of our tree, there will be 2^i nodes and each will represent a problem of size $n/2^i$. Our levels start with $i = 0$, and we know that that the lowest level must have $i = \log_2(n)$, since that will give us our base case. At each node, we do only a constant amount of work (in particular, comparing the minimums of the two sub-problems, which takes time $O(1)$, and finding the subarrays, which also takes time $O(1)$ so long as you just index into a static array and don't copy things over), and so we only have to count the number of nodes total. This gives us the following equation:

$$\sum_{i=0}^{\log_2(n)} 2^i = \frac{1 - 2^{\log_2(n)}}{1 - 2} = n - 1$$

Therefore we know that the overall runtime is $O(n)$. This is the same as what happened in part (a). We know we couldn't have done better!

5. **Recursive local-minimum-finding.** (12 points)

(a) Suppose A is an array of n integers (for simplicity assume that all integers are distinct). A *local minimum* of A is an element that is smaller than all of its neighbors. For example, in the array $A = [1, 2, 0, 3]$, the local minima are $A[1] = 1$ and $A[3] = 0$.

i. (2 points) Design a recursive algorithm to find a local minimum of A, which runs in time $O(\log(n))$.

ii. (2 points) Prove formally that your algorithm is correct.

iii. (2 points) Formally analyze the runtime of your algorithm.

Algorithm 4: findLocalMin

Input: Array A of length n
if $n = 1$ **then**
  return $A[1]$
if $n = 2$ **then**
  return $\min(A[1], A[2])$
$c = \lceil n/2 \rceil$
if $A[c]$ *is a local min* **then**
  return $A[c]$
if $A[c-1] < A[c+1]$ **then**
  return *findLocalMin*$(A[1..c-1])$
else
  return *findLocalMin*$(A[c+1..n])$

SOLUTION: The pseudocode is given in Algorithm 4.

Proof of correctness: We prove correctness by induction, with the following inductive hypothesis.

Inductive hypothesis. Any recursive call of size n returns a local minimum.

Base case: If $n = 2$ or $n = 1$, then our algorithm correctly returns the local minimum by construction.

Induction step: Suppose that the hypothesis holds for $\lfloor n/2 \rfloor$.

Let $c = \lceil n/2 \rceil$ be the middle index. If c is a local minimum (that is, if $T[c] < T[c-1], T[c+1]$), then our algorithm returns that and it is correct. If the middle element is not the local minimum then at least one of $T[c-1]$ or $T[c+1]$ is smaller than $T[c]$. If $T[c-1] < T[c]$, then consider that side of the array, $T[1..c-1]$. By induction, our recursive call will return a local minimum of $T[1..c-1]$. The only problem would be if this local min of $T[1..c-1]$ is not a local min of $T[1..n]$. The only way this would happen is if the local min of $T[1..c-1]$ that is returned occurs at $T[c-1]$. But $T[c-1] < T[c]$, so if $T[c-1]$ is a local min of $T[1..c-1]$, then it is also a local min of $T[1..n]$. The same argument works on the other side.

Conclusion By induction, we conclude that the inductive hypothesis holds for all n. When n is the size of the original problem, it states that the algorithm above returns a local min of $T[1..n]$.

Running Time Analysis: The recursion for the algorithm will be $T(n) = T(n/2)+c$ where c is some constant. Using the Master theorem, we get $T(n) = O(\log(n))$. Without the Master theorem, we can also analyze this directly by imagining a "tree." This is a degenerate tree because each leaf only has one child. So the top node corresponds to a call of the algorithm of size n, the next of size $n/2$, and so on. There is a constant amount of work done at each level, because there is one problem per level, and each level does c work. There are $\log(n)$ levels, so this is $\log(n)$ work.

(b) Let G be a square $n \times n$ grid of integers. A *local minimum* of A is an element that is smaller than all of its direct neighbors (diagonals don't count). For example, in the grid

$$G = \begin{bmatrix} 5 & 6 & 3 \\ 6 & 1 & 4 \\ 3 & 2 & 3 \end{bmatrix}$$

some of the local minima are G[1][1] = 5 and G[2][2] = 1. You may once again assume that all integers are distinct.

i. (2 points) Design a recursive algorithm to find a local minimum in $O(n)$ time.

ii. (2 points) We are not looking for a formal correctness proof, but please explain why your algorithm is correct.

iii. (2 points) Give a formal analysis of the running time of your algorithm.

SOLUTION: Algorithm:

Base cases: If the matrix is of size < 5, just loop over all the elements and return a local minimum.

Otherwise, if the size of the matrix is greater than 5, consider the middle row, the middle column and boundary of the matrix (first and last rows and columns), find the minimum element on the window described above, if it is a local minimum, return.

If not, there is at least one neighbor of that element which is smaller than that element, recurse on the quadrant containing that element (excluding the elements on the window which we already considered).

Runtime: Since we are recursing on one quadrant of size $n/2$-by-$n/2$, and it takes $O(n)$ time to find the minimum element on the cross (and possibly check the last row/column), the runtime recurrence is $T(n) = T(n/2) + O(n)$. This is $O(n)$ time by the master theorem.

Informal explanation:

Base case: Our algorithm is correct for matrices of size < 5 because our algorithms loops over all the elements to check for local minimum.

Let us assume our algorithm works correctly for matrices of size $< n$

For matrices of size n, If minimum element on the window of the matrix is a local minimum, we return it, therefore, it is correct.

If not, we recurse on a quadrant chosen according to the above algorithm. If the local minimum of the submatrix will also be the local minimum of the complete matrix, we will be good since our algorithm finds a local minimum of the submatrix.

So, now, the only thing left to argue is that the local minimum of the submatrix we recurse on is also the local minimum of the complete matrix. Now, neighbor (let us say "a") of the minimum element on the window was smaller than it, so "a" is smaller than elements on the window. In the next recursion call, "a" is part of the window, so the minimum element on the window again will be smaller than "a" and this will keep repeating.

So, eventually doing this recursion, we reach the base case matrices and the invariant that we have is the neighbor in the previous iteration which is part of this quadrant is smaller than all elements on the boundary.

So, when we are on our base case matrix, if base case matrix is of size 1, then by our argument above, it will only contain the neighbour from the previous iteration which will be smaller than all elements on the boundary, hence it will be local minimum.

If, our base matrix is a 2x2 matrix, then also the global minimum of the 2x2 matrix will be smaller than all the values on the boundary and hence, that will be a local minimum.

If our base case is a 3x3 matrix, the global minimum of that 3x3 matrix will be smaller than the neighbour element which is smaller than all the boundary elements and hence, global minimum will be a local minimum.

Same thing happens for matrices of size 4x4.