CS161
Fall 2016

Problem Set Advice

Based on handouts by Tim Roughgarden and Keith Schwartz. Thanks to Michael Kim for editing.

This handout contains information about the problem sets for CS161 – how to approach them, what we're looking for when grading, etc. We hope that this will prove useful as you start working on the first problem set.

General Advice

Here are some general pointers about how to approach the problem sets.

- **Look over the problems early**. Algorithm design takes time, and even simple algorithms can be surprisingly tricky to develop. We suggest reading over all the problems as soon as the problem set goes out so that you will have the time to play around with them over the course of the week.
- **Don't submit your first draft**. When you come up with an algorithm and a correctness / runtime proof for it, your first iteration will likely have some rough edges or parts that ended up not being necessary. We strongly advise against submitting your very first draft of your answers. Taking the time to clean up your proofs and clarify your algorithm will both cement your understanding of the material and help your overall assignment grade.
- **Work on your own before working in a group**. Although you are allowed to work in groups (as specified on the course website), we strongly recommend not doing so until you have already thought about each of the problems and tried out various solutions. The homework problems tend to have solutions that are not particularly complicated but which require some insight to discover. If you immediately start working on the problem sets in a group, you will miss out on the opportunity to come up with these insights on your own.

Citation Policy

When writing up your solutions, you are free to cite any result from lecture, lecture notes, or CLRS that you would like as long as you provide a citation. When citing a result from CLRS, you must not cite results that are given as exercises unless the answer to that exercise is also given. If you consult the Internet, you must include a URL for your source. If you work in a group, please make sure to cite all people you worked with. You must write up your own solutions to each problem.

Describing Algorithms

When writing up an algorithm, you'll need to provide a description of how that algorithm works. At the start of the course, we'll use a lot of pseudocode, but as we progress later on we'll start to get progressively higher level in our descriptions.

As you write up an algorithm, you need to present enough detail so that you can accurately analyze the algorithm's correctness and runtime, but not so much detail that the high-level idea isn't clear. It's perfectly fine to describe an algorithm in plain English as long as there's enough detail to recover how the algorithm works; in fact, we'd prefer this if possible.

You should try to avoid unnecessarily detailed pseudocode unless it's absolutely necessary for your analysis. Low-level detailed pseudocode is hard to understand, so if you do provide pseudocode be sure to provide lots of comments!

Proving Correctness

When designing algorithms, we require that you write a proof of correctness. This should be a rigorous mathematical proof, rather than a general idea about why the algorithm works.

In many cases, we will ask a problem in plain English without providing any mathematical notation. In that case, we suggest introducing symbols or terminology as appropriate when writing the proof. Just make sure that what you have is still easy to read!

Writing a correctness proof doesn't necessarily require going line-by-line through your algorithm and confirming that each step works. If the operation of the algorithm is clear, you can just prove that the general process it follows will work correctly without referring to why each particular step in the algorithm is correct.

Analyzing Runtime

When analyzing the runtime of an algorithm, you do not need to call down to the underlying definition of O, Ω, or Θ notation when proving upper or lower bounds. It's fine to do an informal analysis (e.g. multiplying work done across all iterations by the number of iterations, or accounting for the work done by various code blocks across the entire algorithm runtime). If you want to show that a function runs in time $\Theta(f(n))$, be sure to justify why your bound is tight. One option would be to prove that the function runs in time $O(f(n))$ and $\Omega(f(n))$. Another option would be to do a more precise accounting of the total work done by the function.

In some cases, we will ask you to use the formal definition of O, Ω, or Θ notation in a proof. In that case, be sure to use the formal definitions of these terms.

Sample Problems

To give you a sense for what a good homework solution might look like, we've provided these two sample problems along with solutions that would earn full credit. We suggest taking some time to work through these problems so that you have a sense for how to solve them.

Sample Problem One: Finding a Pairwise Sum

You are given a sorted array A of integers and a target number k. Let n be the length of A. Design an $O(n)$-time algorithm for determining whether there is a pair of values in A sum up to exactly k. Your algorithm should use only O(1) additional space.

Sample Problem Two: Array Partitioning

You are given an array A and a predicate P. Design an algorithm that rearranges the elements of A so that all elements for which P is false appear before all elements for which P is true. Your algorithm should run in $O(n)$ time and use O(1) additional space, where n is the size of the array.

Solution to Sample Problem One:

Our algorithm will be as follows. Set $left = 0$ and $right = n - 1$. Then, repeat the following:

- If $left = right$, return false.
- Otherwise, if $A[left] + A[right] = k$, return true.
- Otherwise, if $A[left] + A[right] > k$, set $right = right - 1$.
- Otherwise (if $A[left] + A[right] < k$), set $left = left + 1$.

Correctness:

Let's define a valid pair to be a pair (i, j) such that $i < j$ and $A[i] + A[j] = k$. To prove correctness, we will show that the following is always true:

(*) If (i, j) is valid pair, then $left \le i < j \le right$.

This statement is true when the algorithm begins, since any array indices i and j with $i < j$ must be in the range $0 \le i < j \le n - 1$, and initially $left = 0$ and $right = n - 1$.

Now suppose (*) is true when the loop starts. If $left = right$, then by (*) we know that no pair exists in the array that can sum to k, because such a pair must satisfy $left \le i < j \le left$, which is impossible. The loop then terminates (because the algorithm returns false), so (*) still holds.

Otherwise, if $left < right$, we consider three cases:

- *Case 1:* $A[left] + A[right] = k$. Then the loop terminates, so (*) still holds.
- *Case 2:* $A[left] + A[right] > k$. Since (*) holds on entry to the loop, we know that if there is a valid pair (i, j), then $left \le i < j \le right$. Since array A is sorted, we know that for any $m \ge right$ that $A[left] + A[m] \ge A[left] + A[right] > k$. Consequently, if there is a valid pair, it cannot involve $A[right]$. Therefore, any valid pair must satisfy $left \le i < j \le right - 1$. Since in this case the algorithm decrements $right$, (*) still holds.
- *Case 3:* $A[left] + A[right] < k$. Using similar logic to Case 2, we can show that if (i, j) is a valid pair, then it must satisfy $left + 1 \le i < j \le right$. Since in this case the algorithm increments $left$, (*) still holds.

Using the fact that (*) is true, we now argue correctness. If the algorithm returns true, then there is a pair that sums to k; namely, it's $(A[left], A[right])$. If the algorithm returns false, it must be the case that $left = right$. By (*), we know that in this case, if there is a solution, it must satisfy $left \le i < j \le right = left$, which is impossible. Thus no solution exists. Therefore, the algorithm returns true iff there is a solution, so the algorithm is correct.

Runtime:

This algorithm runs in time $O(n)$. To see this, note that each iteration does O(1) work, then either increments $left$ or decrements $right$. This means that the quantity $right - left$ decreases by one on each iteration. Since the algorithm starts with $right - left = n$ and terminates when $right - left = 0$, this means that algorithm runs for $O(n)$ iterations. Since each iteration does O(1) work, the total runtime is $O(n)$.

Space Usage:

Since the algorithm only needs to store $left$, $right$, and O(1) temporary variables beyond its parameters, it uses only O(1) additional space.

Solution to Sample Problem Two

Our algorithm is as follows: maintain two indices r and w, both initially 0. Then, repeat the following:

- If $r = n$, the algorithm terminates.
- If $P(A[r])$ is false:
    - Swap $A[r]$ and $A[w]$
    - Set $w = w + 1$
- Set $r = r + 1$

Correctness:

To show correctness, we will prove that the following is always true at the top of the loop:

(*) P is false for all elements in $A[0 \ldots w–1]$ and true for all elements in $A[w, r–1]$

Assuming (*) is always true at the top of the loop, then when the loop terminates (that is, $r = n$), we have that P is false for all elements in $A[0 \ldots w – 1]$ and true for all elements in $A[w, n – 1]$. Thus all elements for which P is false precede the elements for which P is true.

To show that (*) always holds, we proceed by induction. After 0 iterations of the loop, (*) is vacuously true because $A[0 \ldots w–1] = A[0 \ldots -1]$ is empty and $A[w, r–1] = A[0 \ldots -1]$, which is also empty.

Now assume that after some number of iterations that (*) is true. We will prove that (*) is true after the next iteration. Let r_0 be the initial value of r on entry to the loop and w_0 be the initial value of w on entry to the loop. Now, either $P(A[r_0])$ is true or it is false. We consider these independently:

Case 1: $P(A[r_0])$ is true. By (*), we know P is false for all $A[0 \ldots w_0 – 1]$ and true for all $A[w_0 \ldots r_0 – 1]$. Since $P(A[r_0])$ is true, we know that P is true for all $A[w_0 \ldots r_0]$. Since at the end of the loop $w = w_0$ and $r = r_0 + 1$, this means (*) still holds at the end of this iteration.

Case 2: $P(A[r_0])$ is false. By (*), we know P is false for all $A[0 \ldots w_0 – 1]$ and true for all $A[w_0 \ldots r_0 – 1]$. We then swap $A[r_0]$ and $A[w_0]$. Since $A[r_0]$ is false, this now means that P is false for all A$[0 \ldots w_0]$. If $w_0 = r_0$, then it's vacuously true that P is true for all elements in the range $A[w_0 \ldots r_0 – 1]$. Otherwise, if $w_0 \neq r_0$, then since we know that $A[w_0]$ was true on entry to the loop iteration, we now know that P is true for all elements in $A[w_0 + 1, r_0]$. Since at the end of this iteration $w = w_0 + 1$ and $r = r_0 + 1$, this means (*) still holds at the end of this iteration.

Runtime:

To show that this algorithm terminates in O(n) time, note that the loop runs n times, each time doing O(1) work. Therefore, the algorithm runs in O(n) time.

Space Usage:

Since the algorithm only needs to store r, w, and O(1) temporary variables beyond its parameters, it uses only O(1) additional space.