COMP26120

Worked Solution for Week 1 Example

This document provides a typed-out worked solution to accompany the video *Analysing Algorithms Demonstration*. It can be viewed as a model solution for the kind of answer we might expect to Worksheet 1. However note that (i) I describe three algorithms and in Worksheet 1 you are only asked for two and (ii) I make an effort to provide a lot of detail that may not be necessary in your answer.

Problem Definition

We begin with the following general problem description:

> *You work for an advertising company selling banners. Banners require a frame with four side panels, which come in pairs of the same length. Your supplier has provided a list of the lengths of side panels it provides. When a client requests a banner of a particular area you must check whether you can make a banner of that area. Your job is to write an algorithm for this task. You can assume that any dimension of banner is suitable as long as it is the correct area.*

We need to identify the *inputs* and *outputs* in this problem. As input we get:

- *a list of the lengths of side panels* L; and
- *a particular area* k

and as output we need *check whether you can make a banner of that area* k (implicitly) using the elements of L. The area of a banner can be computed by multiplying two side panel lengths together. This leads to a computational problem.

> *Consider a list of positive integers. We are given a positive integer* k *and wish to find two (not necessarily distinct) numbers,* m *and* n*, in the list whose product is* k*, i.e.* $m \times n = k$*.*

The *not necessarily distinct* part comes from the observation that we are allowed to reuse side panels to create a square banner e.g. we can create a banner of area 25 using two pairs of side panels of length 5.

Test Data. At this stage it is usually sensible to create some sample tests data to test our solutions with. This should aim to cover any edge cases we can identify in the abstract problem. Here's some example test data for this problem:

L	k
$[1, 2, 3]$	6
$[1, 2, 3]$	7
$[5]$	25
$[2, 8, 4]$	16
$[5, 24, 9, 5, 30, 6, 3, 12, 2, 10]$	72

Can you think of any other interesting cases?

Solution 1: The Naive Brute-Force Approach

The naive approach is to look at each pair of numbers in the array and check to see if they are a solution. To avoid being overly naive we should return as soon as we find a solution and we should avoid checking symmetric pairs e.g. if we check $L[i] * L[j]$ we don't need to check $L[j] * L[i]$.

Pseudocode. This solution is nice and short:

```
search(int[] L, int k)

    for i from 0 to length(L)
        for j from i to length(L)
            if L[i] * L[j] = k then return true
    return false
```

Correctness. Let (i, j) be a possible solution. It is a solution if $L[i] \times L[j] = k$. We need to ensure that every possible solution (i, j) for $0 \leq i, j \leq \mathsf{length}(L)$ is checked. Due to the symmetry of solutions ((i, j) is a solution if and only if (j, i) is) we only need to check solutions (i, j) such that $i \leq j$. This is exactly what our nested loops do.

Complexity. The worst case is that the algorithm will return `false` and we need to iterate through all loops fully. Let us label `length(L)` as n. The outer loop will iterate n times. The inner loop will iterate n times (for the first loop), then $n - 1$ times and so on until it iterates once for the final loop. This gives us

$$\sum_{i=1}^{n} n - i + 1$$

iterations, which is equivalent to

$$\sum_{i=0}^{n-1} n - i = \sum_{i=1}^{n} i$$

and we should be familiar with the closed-form expression for the sum that gives us

$$\frac{n(n+1)}{2}$$

From a Big-Oh point of view this is $O(n^2)$ but practically we can see that it is better than n^2 by roughly a factor of 2.

Solution 2: The Sorting Approach

The sorting approach gets us to sort the list first and then search from either end of the list for solution pairs. This is based on the idea that we can rule out many possible solution pairs as the order of values in the list will mean that the solution pair has to be smaller/greater than k.

Pseudocode. I've added an explicit sorting step to the algorithm that was missing in the video.

```
search(int[] L, int k)

    Sort L in-place from smallest to largest
    i:=0; j:= length(L) - 1
    while i ≤ j
        value:= L[i] * L[j]
        if value= k then return true
        if value < k then i++
        else (value > k) and j--
    return false
```

Correctness. We prove the following statement holds on every iteration of the loop:

If (i', j') is a solution then $i' \geq i$ and $j' \leq j$

Alternatively

For all $i' < i$, i' is not part of any solution *and* for all $j' > j$, j' is not part of any solution.

You should be happy that these two statements say the same thing.

It should be clear that if this is true on every iteration and it is also true when $i > j$ (when the loop terminates) then there can be no solution and therefore returning `false` is correct. It is also obvious that whenever we return `true` we are doing so correctly.

We prove the above statement by an inductive argument. Before we start the loop it must be true because all solutions (i', j') lie between the initial i and j. For an arbitrary iteration of the loop we can assume the statement to be true for (i, j) at the start of the loop and we need to show that it still holds at the end of the loop. We have three cases:

1. (i, j) is a solution. Trivially we terminate and do not update (i, j).
2. $L[i] \times L[j] < k$. We show that the above statement holds for $(i + 1, j)$. To do this we just need to show that i cannot be part of any solution (all other values are covered by the statement holding for (i, j)). Because L is sorted we know that $L[j'] \leq L[j]$ for all $j' \leq j$. Therefore, $L[i] \times L[j'] \leq L[i] \times L[j] < k$ for $j' \leq j$ meaning that it is not possible for (i, j') to be a solution for any $j' \leq j$. We don't have to worry about the values that are greater than j since these have already been excluded by our assumption at the start of the loop. Since we have now considered all values, i cannot be part of any solution.
3. $L[i] \times L[j] > k$. This is symmetric to case 2.

Complexity. In the worst case the while loop will iterate n times. We may assume we can sort the list in $O(n \log n)$ which dominates $O(n)$. Therefore, the cost of sorting is the main cost of this approach. If the list were already sorted then the complexity would be $O(n)$. This would help if, for example, we needed to check many k with a single L - we could *amortize* the cost of sorting across the checks.

Solution 3: The Remembering Approach

In the final solution we will iterate through the list once remembering the other half of solutions that we are looking for. We will use a set to do this remembering and there are different ways that we could implement this set to get different complexity guarantees.

Pseudocode.

```
search(int[] L, int k)

    Let S be an empty set
    for i from 0 to length(L)
        a = L[i]
        if a divides k then
            Add a to S
            b = k/a (this has to be an integer)
            if b in S return true
    return false
```

There are two options for implementing the set:

1. We can use a *Hash Set* backed by a *Hash Table*. We will cover how to implement this data structure later in the course. For now we just need to know that there is an amortised argument for addition and containment checks being constant time for hash tables. Space complexity for hash tables is $O(n)$ where n is the number of things added, in this case `length(L)`.
2. If k smaller than `length(L)` then we can do better by using an array of size k for the set and lookup values by index. We would need to handle values $> k$ specially. We could also handle the case of checking for k separately and use an array of size $k/2$. All these operations would give us constant time and in practice could be more efficient than a hash table even if $k >$ `length(L)`.

Correctness. If (a, b) is a solution then a will be in S as a necessarily divides k. If a is in S then when we see b we will compute a from b and find it in S. Thus, if a solution exists we will find it. Conversely, any solution we find is a true solution as at the point we return it we know that $a \times b = k$ (as a divides k and b = k/a) and that both a and b appear in L. Note that this argument also works for solutions of the form (a, a).

Complexity. In the worst case we perform n insertions into S and n containment checks in S. If both operations are constant time then the overall complexity is $O(n)$.

Summary

We have three algorithmic solutions getting increasingly 'better'. We can see that often correctness arguments can be straightforward but sometimes we have to do a bit more work. At the end of the second solution I pointed out that a different but related problem might have a different solution. What would you do if our problem was instead phrased:

> *Consider a list of positive integers. We are given another list of l positive integers and wish to find, for each k in L, two (not necessarily distinct) numbers, m and n, in the list whose product is k, i.e. $m \times n = k$.*

In other words, we now have lots of k and one L. Is our final solution still the best?