DMFS - Problem Set 1

pwn274

February 2022

Contents

1 Algorithm 1

```
for i := 1 upto n {
    B[i] := 1
    for j := 1 upto i {
        B[i] := B[i] * A[j]
    }
}
```

1.1 Explain in plain language what the algorithm does

The algorithm takes an two arrays (A and B) and set every element in the second array B to be the product of all elements in A with indices below the element in B.

1.2 Provide an asymptotic analysis of the running time as a function of the array size n

The outer for loop runs for n iterations. The inner for loop runs $\sum_{i=0}^{n} i$ times, which is an average of $n/2$ iterations. The total number of loops made is therefore bounded by:

$$\mathcal{O}(n \cdot n/2) \approx \mathcal{O}(n^2) \tag{1}$$

1.3 Can you improve the code to run faster while retaining the same functionality?

I sure can:

```
B[1] := A[1]
for i := 2 upto n {
    B[i] := A[i] * B[i]
}
```

Which runs in $O(n)$ and is therefore the fastest possible (up to a factor), as we need to utilize at least one `for`-loop of size n to fully fill B.

2 Algorithm 2

```
for i := 1 upto n {
    if (A[i] < 1 or A[i] > n)
    return "failure"
}
i := 1
found := -1
while (i <= n and found < 0) {
    for j := 1 upto n {
        B[j] := false
    }
    j := i
    while (B[j] == false) {
        B[j] := true
        j := A[j]
    }
    if (A[A[j]] == j)
        found := j
    i := i + 1
}
return found
```

2.1 Explain in plain language what the algorithm above does

The algorithm starts out by checking whether all elements in A would correspond to legal indices in B, returning 'failure' if not.

If A is legal, the elements of A is used as indices for randomly bouncing around in B, flipping the elements from false to true until an element is touched twice. It then check whether the last used element in A is equal to its own index. If it is, it is returned, if not, a new starting point is chosen and we rinse and repeat. If no such element is found in this way, -1 is returned.

In particular, when does it return a positive value, and, if it does, what is the meaning of this value?

2.2 Provide an asymptotic analysis of the running time as a function of the array size n

In the worst case, the outer `while`-loop runs for n iterations. The same is true for the inner `while`- and `for`-loops. Combining all the contributions and approximating as we have learned, we get a worst-case running time of:

$$\mathcal{O}(n + n * (n + n)) = (2n^2 + n) \approx \mathcal{O}(n^2) \tag{2}$$

2.3 Can you improve the code to run faster while retaining the same functionality? How much faster can you get the algorithm to run?

I am slightly unsure what the B-array is even supposed to be doing. If all we want is a potential element in A that is equal to its own index when used to index itself, I would do it as follows:

```
for i := 1 upto n {
    if (A[i] < 1 or A[i] > n)
        return "failure"
    elif (A[A[i]] == i)
        return i
}
return -1
```

which runs in $\mathcal{O}(n)$ and finds the same type of elements as the original pseudocode.

If the random jumping is required and we want the exact same functionality, it will require running the same (worst-case) n-long loop for each possible starting point, making it impossible to beat $\mathcal{O}(n^2)$ as far as I can see.

3 Algorithm 3

We have the following code snippet:

```
search (A, lo, hi)
    if (A[lo] >= A[hi])
        return "failure"
    else if (lo + 1 == hi)
        return lo
    else
        mid = floor ((lo + hi) / 2))
        if (A[mid] > A[lo])
            search (A, lo, mid)
        else
            search (A, mid, hi)
```

Starting from

```
search (A, 1, n)
```

3.1 Explain in plain language what the algorithm above does.

This is a recursive binary search, which halves the search space until the smallest element is found. If the list is unsorted, the algorithm returns 'failure'. I have no idea why this is useful, as finding the lowest element in a sorted list amounts to simply taking the lowest index, which I can do by hand in $\mathcal{O}(1)$ time.

3.2 Provide an asymptotic analysis of the running time as a function of the array size n

For each iteration, the array size is about cut in half. As I can clearly recognize a recursive binary search, it is in no way surprising that it will have an average running time of $O(\log(n))$. If we are looking at worst-case running times, there will be n recursive calls, giving a complexity of $\mathcal{O}(n)$.

3.3 Final questions

Could it be the case that recursive calls of the algorithm also return "failure", or would it be sufficient to check just once before making the first recursive call?

There is no need for the "failure" check to be each iteration as it only amounts to checking whether the list is sorted and the list is never mutated. Doing it the way it is written makes it so it only has to make $\mathcal{O}(\log n)$ checks, but the results are much more dependant on the original array.

If we get the additional guarantee that all elements in the array are distinct, could we remove the "failure" check completely, since we would be guaranteed to never have this answer returned anyway? What about if we get the additional guarantee that the array is sorted in increasing order? What if both of these extra guarantees apply?

If the list contains at least two elements, it is sorted, and no two elements are the same, $lo \leq hi$ would not trigger, no matter the contents of the array. If the elements are non-unique, the $lo == hi$ could trigger and if the list is unsorted, the $lo > hi$ could trigger.

A I like python

For fun, I wrote a python snippet this morning to visualize:

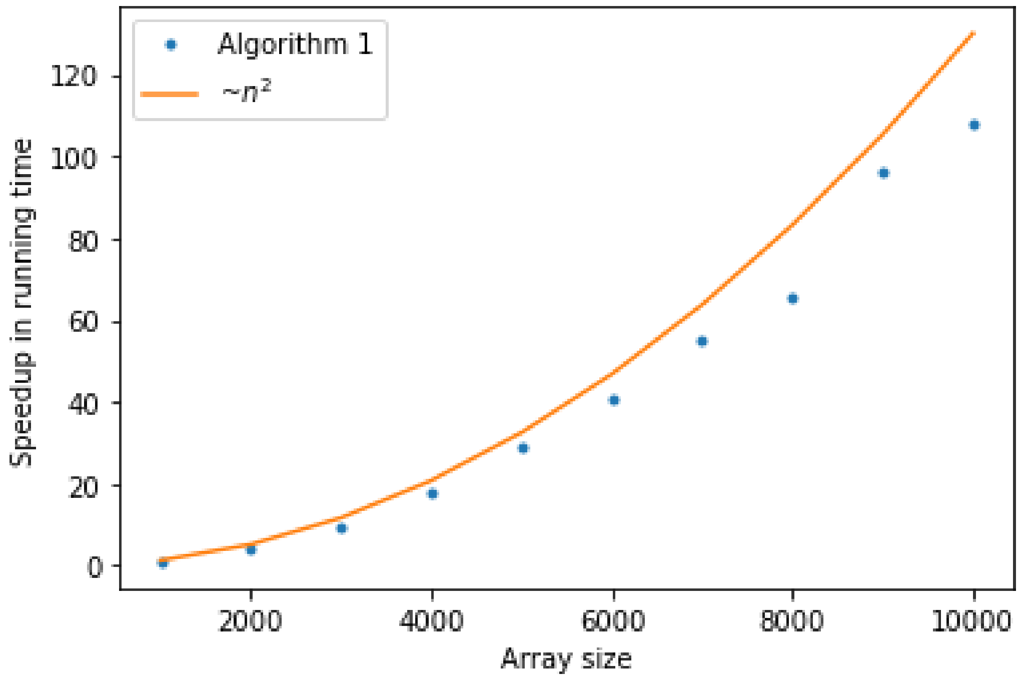

Figure 1: I am an ex-physicist, so empirical evidence has a special place in my heart

The n^2 is scaled down by a constant factor, but visually, you could convince me, that it will remain an upper bound.