Algorithms and Datastructures (HS2024)
Week 3

Rui Zhang

October 7, 2024

Contents

1 Revision Theory

This time, we have Maximum Subarray Sum (MSS) and two other topics that are probably only covered in the next exericise sheet. Focus on MSS first (come back to this summary once the summary is finished, otherwise revise the professor's notes), and then take a look at Ω and Θ to prepare for next week.

1.1 Maximum Subarray Sum

Given an array of n numbers representing the change in price of a stock across multiple days, where each entry of the array stands for the change in price after that day, we want to calculate the perfect time to buy a stock and then sell the stock (to make the most profit). This problem corresponds to calculating the maximum sum of a continuous subarray. That is why this problem is called Maximum Subarray Sum.

Here, we consider the number of additions performed as the elementary operations.

1.1.1 Naive Algorithm

Again, we start off with the most simple solution. Let us say a subarray ranges from the i-th position to the j-th position:

$$a_1, a_2, a_3, a_4, a_5, a_6, \ldots, a_n 0$$
$$= 2, 4, 1, 6, 34, 59, \ldots, 10$$

In this example, if $i = 2, j = 5$, then the subarray would be

$$a_2, a_3, a_4, a_5$$
$$= 4, 1, 6, 34$$

And its sum would be $S_{i,j} = S_{2,5} = a_2 + a_3 + a_4 + a_5 = 4 + 1 + 6 + 34$. Now, we need to calculate the subarray sum for all possible subarrays, so for all combinations of i and j. How many combinations are there? There are n possibilities for i and $n - i + 1$ corresponding possibilites for j for each i we fix (those possibilities are $j = i, i+1, i+2, i+3, \ldots, n-1, n$). For the first possibility ($i = j = 1$), we only have one element as the subarray, so there aren't any additions. For all further possibilites of j, we can just take the last possibility $j - 1$, and add the a_j to the previous sum: $S_{i,j} = S_{i,j-1} + a_j$. So in total, for each possibility of i, we have $(n - i + 1) - 1 = n - i$ many additions (the $n - i + 1$ comes from the number of possibilities for j and the -1 comes from not needing an addition for the first possibility).

Thus, in total we have:

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

number of additions. That is in $\Theta(n^2)$.

1.1.2 Recursive Algorithm

A faster algorithm consists of splitting the array into two halves and solving the MSS Problem for those two halves recursively. But we have a problem. If we do this, we will find the maximum subarray sum of both halves. But what if for the full array, the maximum subarray sum is not a sum of a subarray that is strictly in the left or right half, but is the sum of a subarray which starts in the first half ($i \leq n/2$) and ends in the second half ($j > n/2$)? We can find this subarray sum by finding the largest subarray sum that begins somewhere in the left partition (at i) and ends at $n/2$ and adding it to the sum of the largest subarray sum that begins at $n/2+1$ and ends somewhere in the right partition (at j). Doing this is just applying the previous naive algorithm but with one variable fixed and the other iterating over every possibility. From the previous section, we had a total of $n-i$ additions. This time, since we only have half of the array and we go through all $n/2$ possibilites for the non-fixed variable (i for the left half and j for the right half), we have $n/2-1$ additions per halve. That's $n-1$ additions in total. The advantage this idea has over the naive approach is that this number $n-1$ is linear, whereas the naive approach has quadratic complexity.

This recursive approach leads to a runtime function of

$$A(n) = \underbrace{2 * A(n/2)}_{\text{solving problem for the 2 halves recursively}} + \underbrace{n-1}_{\text{number of additions for the middle part}}$$

where $A(n)$ is the number of additions to execute this algorithm.

Solving this equation and finding a closed formula, assuming the n is a power of two to make our lives easier (as done in the lecture) yields:

$$A(n) = n\log_2(n)$$

Which is indeed an improvement.

1.1.3 DP Algorithm

We have not talked about dynamic programming yet, do not worry. But this final algorithm is actually the first example of a DP Algorithm you see in the lecture, without it being explicitly mentioned as such.

The idea behind DP is to apply recursion and deriving the solution for a certain problem size n from the solution for the same problem with smaller problem size $m < n$ (so recursion), but to also save the result of this previous computation (solution for $m < n$) in some data structure for reuse. The dynamic programming part of this algorithm is very subtle and trivial and also only consists of one dimension, but soon, you will encounter DP for up to 4 dimensions, so revising this approach is worth the time.

The idea is to calculate the number $R_j = \max_{i \leq j} S_{i,j}$ for all possiibilities of j. This number reflects **the maximum of all subarray sums that end in j and that start somewhere (i)**,

Now, with this definition in hand, we have can observe the following: to calculate R_k, we have to include a_k, the corresponding subarray must end at position a_k. This means that the only options we have is to

1. either only take this a_k

2. or add this a_k to the maximum subarray sum that ends just at a_{k-1} (because all other subarrays ending in a_{k-1} that is not maximal will not result in the maximum sum)

But this is great is it not? For the second option, we can just add a_{k-1} to R_{k-1}! So now we just have to take the maximum of both options, which is only a matter of checking if R_{k-1} is negative or not. If it is, then it is better to just take a_k. If it is ≥ 0, then we may add the current element a_k to R_{k-1} to receive a sum that is at least larger than or equal to a_k itself.

Now, all we have to do is initialize R_1 to be a_k and iterate through k until we reach $k = n$. Then, since the maximum subarray does not have to end at a specific fixed spot j, we just have to take the maximum of all R_k's and receive the solution:

$$\max(R_1, \ldots, R_n, 0)$$

where the zero is added in case all of the subarray sums are negative, in which case we just don't buy / sell at all and receive a total profit of 0.

In total, we are doing at most $O(n)$ additions.

1.1.4 Can we do better (Than O(n))?

Sadly, not. If we were to not read all of the n elements, then we will miss at least 1 element in the array that has not been read. The value of this array element may then vary. Depending on this element, our solution may change significantly.

In other words, not reading any element in the array could result in an incorrect algorithm. Just the number of reads as elementary operations would limit the best performance of this algorithm to be $O(n)$ then.

1.2 Ω-Notation

1.2.1 Motivation

Until now, we have always been making statements about the worst case scenario. Such as "This algorithm runs in $O(n^2)$", which means that the algorithm has a runtime dependent on n - let that be $f(n)$ - which grows asymptotically slower than n^2, even in the worst case scenario. In those sort of situations, it makes most sense to use Big-O-Notation, since, as we have learned, $O(g)$ is the set of functions f that grow asymptotically slower than g.

But, what if we want to prove that a certain algorithm has a best case scenario runtime of so and so? For instance, what if we wanted to make the claim that a comparison based sorting algorithm cannot be faster than in the order of $log(n)$ (we will see this later in the lecture)? Let $f(n)$ be the function denoting the runtime of an arbitrary comparison based sorting algorithm. We cannot write $f(n) \leq O(log(n))$, because what if this arbitrary algorithm is terrible and $f(n) = n^{10000}$? Then this mathematical statement would be plain wrong. Therefore, we need a complementary concept.

1.2.2 Definition

From the motivation, it should become clear what kind of definition $\Omega(f)$ should have. It should be the "opposite of Big-O". It should probably also be a set and should contain all functions that grow asymptotically faster. We would use the notation

$$g \geq \Omega(f)$$

instead of

$$g \in \Omega(f)$$

The formal definition is left as an exercise.

1.3 Θ-Notation

1.3.1 Motivation

We have talked about the worst-case scenario (Big-O). We have talked about the best-case scenario (Ω).

If the worst-case scenario and the best-case scenario are both the same, then we use the Θ-Notation.

1.3.2 Definition

Let f, with $f : \mathbb{N} \to \mathbb{R}^+$, be some function representing the runtime of an algorithm. We say

$$f = \Theta(g) \overset{\text{def.}}{\Leftrightarrow} f \leq O(g) \ \wedge \ f \geq \Omega(g)$$

Again, the formal definition of $\Theta(f)$ for arbitrary functions f is left as an exercise.

2 Exercises

2.1 Exercise Sheet 3 - Priority List

Again, top is most important, bottom is least important.

1. 3.3, 3.1 (these are bonus questions and also questions that often appear in exams)
2. 3.2 (good practice for developing algorithms)
3. 3.5 (it is still important for you to know the idea behind iterative squaring.)

2.2 Exercise Sheet 1 - Feedback

I have tried my absolute best this time to give as many points as possible! In general, please look through your solutions once more and make sure that every detail and calculation is correct. Our grading scheme is very strict, same as in the exam: Small mistakes in calculations that still yield the same correct results will still result in a deduction of points... As far as I can see, most of you have understood all of the concepts very well, but you have to work on your precision.

Additionally, many of you have decided to combine the formula from "to show" along with the actual proof. Formally speaking, this is quite dubious because we do not want to use anything from the part that we want to show as a step in the proof. So just copy the formula on the left again for the "proof" part. It does not save you that much time and is much more clear and clean.

Beware of backwards proofs as well. In general, proving a statement from left to right may seem harder at first, but writing it down as as such will reduce the number of mistakes made.

3 Supplementary Exercises

3.1 Ω-Notation

Based on the motivation of the Ω-Notation and the definition of Big-O, formally define $\Omega(f)$ for arbitrary functions f, with $f : \mathbb{N} \to \mathbb{R}^+$.

3.2 Θ-Notation

Based on the definition of Θ-Notation from above, answer the following questions:

1. if $f = \Theta(g)$, then what can we say about the relation between f and g in terms of asymptotic growth rate?
2. give a formal definition of $\Theta(f)$ for arbitrary functions f, $f : \mathbb{N} \to \mathbb{R}^+$.

3.3 Extra Exam Exercises

We did not have any time this week to go through further exercises, so here are the ones I prepared. Again, they can be found on the VIS Exams page.

For the following, prove or disprove:

1. $\frac{1}{2}n^2 - \sum_{i=1}^{n-1} i \leq O(n)$
2. Suppose $a_1 = 1 and a_{i+1} \leq O(a_i)$ for all i. Then $a_n \leq O(n)$
3. $\sum_{i=1}^{n} \log(i!) = \Theta(n^2 \log(n))$ For this, answer the following:
   (a) Intuition
   (b) What is it that we have to show to prove this statement? (if it holds)

(c) Proof / Counter Example