**A model solution for an algorithmic problem**

# Maximum sum subarray problem

**Problem:** Given an array $A$ storing $n$ numbers, compute the subarray of $A$ whose sum is maximum among all possible subarrays of $A$.

**A model Solution:**

Let $S_i$ denote the sum of the largest-sum subarray ending at (and including) $A[i]$. Suppose $S_k$ is maximum among all $S_i$'s. If $A[j, .., k]$ be the subarray corresponding to $S_k$, then $A[j, .., k]$ is indeed the subarray of $A$ achieving maximum sum. So to solve our problem, it suffices if we compute $S_i$ for each $i$. Let us develop some insight into $S_i$'s.

It follows from definition that $S_0 = A[0]$. We now state the following Lemma which establishes a relationship between $S_i$ and $S_{i-1}$ for any $i \geq 1$.

**Lemma 1.** *If $S_{i-1} \leq 0$ then $S_i = A[i]$; otherwise $S_i = S_{i-1} + A[i]$.*

*Proof.* The key insight to prove the lemma lies in the following relationship between the subarrays associated with $S_i$ and $S_{i-1}$. Notice that the subarray associated with $S_{i-1}$ if appended with $A[i]$ gives us a subarray of $A$ which terminates at $A[i]$. Hence, $S_i \geq S_{i-1} + A[i]$.

Now $S_{i-1} < 0$ implies that every subarray terminating at $A[i-1]$ has a negative sum. In other words, it is futile to append any subarray terminating at $i-1$ with $A[i]$ to get a better sum. Hence $S_i = A[i]$ in this case. Likewise, if $S_{i-1} > 0$, then by definition, the subarray corresponding to $S_{i-1}$ is the subarray of maximum sum terminating at $A[i-1]$. This subarray when appended with $A[i]$ will give us a subarray terminating at $A[i]$ and having maximum sum. This is exactly the subarray corresponding to $S_i$. Hence $S_i = S_{i-1} + A[i]$ in this case. □

Algorithm 1 computes maximum sum subarray based on the above lemma.

**Data**: Input will be an array $A[0..n-1]$. However, the algorithm will use additional arrays $S[0..n-1]$ and $L[0..n-1]$ such that $S[i]$ will store $S_i$ and $L[i]$ is the index of the leftmost element of the subarray associated with $S_i$.
**Result**: The sum of the maximum sum subarray along with its leftmost and rightmost indices.

```
1  S[0] ← A[0];
2  L[0] ← 0;
3  for i ← 1 to n − 1 do
4  |  if S[i − 1] ≤ 0 then
5  |  |  S[i] ← A[i];
6  |  |  L[i] ← i;
7  |  else
8  |  |  S[i] ← S[i − 1] + A[i];
9  |  |  L[i] ← L[i − 1];
10 |  end
11 end
12 Scan array S to compute the maximum element of S;
13 If S[k] is maximum, output S[k], L[k], k.
```

**Algorithm 1:** Algorithm for finding Maximum-Sum Sub Array

**Proving correctness of the algorithm:**
We need to prove that Algorithm 1 is correct. It follows from the discussion preceding Lemma 1 that it suffices if we can show that Algorithm 1 correctly computes $S_i$'s.

**Lemma 2.** *At the end of the "for" loop in Algorithm 1, $S[i]$ stores $S_i$ for each $i < n$.*

*Proof.* We give a proof by induction of $i$. In particular, we show that assertion $\mathcal{A}(i)$, defined below is true for all $i < n$.
$\mathcal{A}(i)$ : At the end of $i$th iteration, $S[i] = S_i$.
The base case ($i = 0$) is trivially true since we explicitly set $S[0]$ to $A[0]$ (which is indeed $S_0$) in the first statement of the algorithm. Hence $\mathcal{A}(0)$ holds true.

Let us prove that $\mathcal{A}(j)$ is true for some $j > 0$, assuming, as induction hypothesis, that $\mathcal{A}(i)$ is true for all $i < j$. Consider the beginning of $j$th iteration. Firstly note that by induction hypothesis, $S[j-1] = S_{j-1}$. Now Lemma 1 states that $S_j$ is $A[j]$ if $S_{j-1} < 0$ and $S_{j-1} + A[j]$ otherwise. Since $S[j-1]$ stores $S_{j-1}$, it follows from the "if" statement executed during $j$th iteration of "for" loop that $S[j]$ stores $S_j$.

Hence by principle of mathematical induction, $S[i]$ stores $S_i$ for each $i < n$. □

**Remember:** It may appear that Lemma 2 or the proof of correctness of Algorithm 1 is unnecessary since Algorithm 1 seems *obviously* correct. But the proof is needed for each algorithm irrespective of how simple it looks. This is due to the following reasons.

- A formal proof of correctness (or sometimes analysis) of an algorithm is important to rectify some errors. For example, in the binary search algorithm, it leads to serious problem if you replace the statement $Left \leftarrow mid - 1$ by $Left \leftarrow mid$.
- In future, you will find many algorithms whose proof of correctness will not be obvious. However, using mathematical induction, it may turn out to be easy provided you formulate the assertion carefully. In that spirit, the simple proof by induction in Lemma 2 will serve as a useful tool for you.
- If you study the proof of Lemma 2 carefully, you will realize that the proof by induction given in Lemma 2 is nothing but the formal way which you will use to convince yourself about the correctness of Algorithm 1. Isn't it ? It is always good to express your thoughts explicitly in words when it comes to analysis of an algorithm. It may give you a better insight into the algorithm as well.

**Analysis of Time and Space complexity the Algorithm**

Algorithm 1 executes "for" loop $n-1$ times, and in each iteration it spends $O(1)$ time. Thereafter, it spends a total of $O(n)$ time in the "for" loop. Thereafter, it scans array $S$ to find the maximum element. This also takes $O(n)$ time. So overall time complexity of the algorithm is $O(n)$. The algorithm uses two additional arrays $S$ and $L$ in addition to a few variables. Hence, the algorithm uses $O(n)$ extra space.

**Note:** We can suitably modify Algorithm 1 so that it uses just a few extra variables instead of the extra arrays $S$ and $L$. In that case this algorithm runs in $O(n)$ time and takes only $O(1)$ extra space. Do it as an exercise.