Maximum sum subarray problem(M(S)SP)

Author:George Vlassis  
Version:1

Definition:Given an n-tuple: A=(a1,a2,…,an) (mathematical equivalent of the well-known array used in programming), with each element being a real number, find S, I and F, where:  
S: maximum sum of all the possible continuous sub-tuples of A  
I(Initial): position of the first element of S, in A  
F(Final): position of the last element of S, in A

Notes:

\*By convention we allow the empty tuple(here denoted as () ) as a solution to the problem. The sum of the empty tuple will be considered 0, and in this case I=F=-1(this will simplify our code)

\* The MSP can be extended to higher dimensions.

Examples:1) A=(-3, 6, -8) => S=(6), I=2, F=2

2) A=(-2.5 , 5, 6.4, 7, -99.32 ) => S=(5, 6.4 , 7), I=2, F=4

3) A=(8, 9, 15, 24, 67.1) => S=A, I=1, F=5

4) A=(-2, -5) => S=(), I=F=-1

5) A=(140, -3, 4, 5, -8, 3) => S=(140, -3, 4, 5), I=1, F=4

History:

Ulf Grenander, a statistician of Brown University, while developing a procedure for the analysis of digitized images came across the 2D version of the MSP. In that version A is not an n-tuple but a matrix, S is the maximum sum of all the possible continuous sub-matrices of A, and we are looking for 4 indexes: the first and last row of S in A, and the first and last column of S in A. S was needed in the maximum likelyhood estimation of a certain kind of pattern in an image. He saw that a brute-force approach needs O(n^6) time, which is a prohibitively large amount . A better algorithm was needed. In an attempt to better understand the structure of the problem, he devised the 1D version. In that version the brute-force algorithm had O(n^3) time complexity, again prohibitively large for common array sizes. He derived a faster algorithm, which needed O(n^2) time, a lot better than O(n^3), but still slower than desired.

Later, he described the problem to the mathematician Michael Ian Shamos of Carnegie-Mellon University who, overnight, designed an O(n\*logn) algorithm. Shamos discussed the algorithm with Jon Bentley (author of the popular coding book “Programming pearls”) and they both agreed that this was probably an asymptotically optimal solution(which means that any other possible solution would be,at best, asymptotically equal). This assumption was further reinforced by the fact that researchers had just proven that several similar problems required at least O(n\*logn) time.

After a few days, while Shamos presented the problem at a seminar, a member from the audience, Joseph(Jay) Born Kadane, a statistician at Carnegie-Mellon University, immediately came up with an O(n) algorithm which, thankfully, has been proven to be asymptotically optimal.

As for the original 2D MSP, it was later proved to be solvable in O(n^3) time by extending Kadane’s algorithm. The upper bound has been further reduced to sub-cubic time, without parallel processing, or O(logn) if O(n^3/logn) processors are available (an unrealistic amount of processors even for supercomputers).

Although Grenander’s initial approach to the image analysis problem has long been abandoned, the MSP remains popular both in academia and in the industry. Research continues to be done, mainly for the 2D version of the problem, and the 1D version has been frequently asked in job interviews from companies such as Facebook, Microsoft or Goldman Sachs.

Algorithm1:This is based on the brute-force/exhaustive/generate and test paradigm. In short, in the brute-force paradigm we systematically create every possible solution to a problem and check if it can indeed be a solution.

The algorithm, implemented in Java, is as follows:  
A screenshot of a cell phone

Description automatically generated

Correctness: Every subarray is compared and thus their maximum will be the solution

Complexity: First loop will be executed n times. Second loop will be executed n-1, n-2, …, 1 times. Third loop will be executed at worst n times. Thus we have O(n^3).

\*The function is located in the Main class(that is the class that has the “main”) and is static so it can be called without a Main instance.

\*The input of the function is an integer array, but this is just for the sake of simplicity. The function can be trivially extended for double array

Algorithm2:This is also based on the brute-force approach, but with an intelligent tweak that reduces the time complexity. Let’s say that we are in second loop above with n=10, I=5, F=7. Via the third loop we find the sum in[5..7]. Then F=8. In the first 3 iteration of the third loop we calculate again in[5..7]. This is unnecessary. We could simply use in[5..7] to get in[5..8] if we did: in[5..8]=in[5..7]+in[8]. In general we do: curSum=curSum+in[F] at the beginning of each second loop in order to immediately calculate the new curSum using the old current sum and the value of the extra element. This way we could eliminate the third loop and speed up our program a lot. Similar observation can be made in many programs, and the general principle is that we should save data to avoid recomputation.

The algorithm, implemented in Java, is as follows:  
A screen shot of a smart phone

Description automatically generated

Correctness: Same as algorithm1

Complexity: Same as algorithm one, but now we have only the first two loops. Thus O(n^2).

Algorithm3:This is again based on the brute-force approach but uses preprocessed datato achieve a speed-up. Let’s say we have an n-array named B, whose every element bi is equal to the sum: bi=a1+a2+…+ai. Then the sum of a subarray with starting index i and ending index j can be calculated as: Sij=(bj-bi)+ ai (the first term is the sum of A[a(i+1)..ai] ). This way we do not need the third loop. Getting B is a trivial task, and thus we again arrive at an algorithm with O(n^2) complexity.

Java implementation:  
A screenshot of a cell phone

Description automatically generated

Correctness: Same as algorithm1 and 2

Complexity: As with algorithm2: T(n)=O(n^2) (T(n) denotes the execution time)

Algorithm4:This algorithm is based on the very frequently used approach divide and rule/conquer. According to this approach, to solve a problem we break it down to similar sub-problems, solve them and then combine their solutions to yield a solution to the original problem. To solve the sub-problems we divide them once again, and this continues until the problems become simple enough to be solved directly. In code this is realized by recursively calling the same function used to solve the initial problem to the sub-problems that constitute it.

Algorithm4 is based on the following observations:   
1) If we divide A in two parts (not necessarily equal), then the max subarray will be on the left part, on the right part, or between those parts.   
2) If the solution is between the two parts it will be the combination of two subarrays, one on the left and one on the right.   
3) The first subarray will be the maximum subarray that starts at some point on the left part and ends at the middle of the two parts. Similarly, the second subarray will be the maximum subarray that starts at the middle of the two parts and ends at some point on the right part.

In short the program divides A in the middle recursively, and in each recursion finds the maximum subarray that crosses the middle of the initial array, the maximum subarray on the left part(by recursion) and the maximum subarray on the right part(by recursion). It then compares these three maximums to decide which one is the real max subarray.

*The rest of the page remains blank so all of the implementation code will be on the same page.*

Java implementation of the recursion:  
A screenshot of a computer

Description automatically generated  
A screenshot of a cell phone

Description automatically generated  
A screenshot of a cell phone

Description automatically generated

Correctness: It is a logical consequence of the description of the algorithm given above and the comments on the actual code.

Complexity: In each recursion we have [M-I+1]+[F-(M+1)+1]=M-I+1+F-M-1+1=F-I+1 iterations (the first term comes from LEFT part determination, while the second term comes from the RIGHT part determination). At worst(in the first recursion) F-I+1=n. The maximum recursion depth d can be calculated as below:

This means that an upper limit of the algorithm will be: O(n\*logn).

\* In order to have same behavior as on the other functions we can add a method as below:  
A close up of a logo

Description automatically generated

Algorithm5:A common approach to algorithmic problem solving is, instead of solving a problem of size n, finding a solution to a problem of smaller size(e.g n-1) and then somehow extending it to solve the original problem. This is the approach we will follow here.

Let’s say we have found the solution for A[a1…ak]. Can we extend it to A[a1…a(k+1)]? The solution to A[a1…a(k+1)] will either contain a(k+1) or it will not. If it does not contain a(k+1), it will be the same with the previous solution. If it does contain it, the solution will be the maximum subarray that starts somewhere in A[a1…a(k+1)] and ends in a(k+1)(empty array is allowed). The solution to A[a1… a(k+1)] is the maximum between the two subarrays mentioned above.

The algorithm, implemented in Java, is as follows:  
A screenshot of a cell phone

Description automatically generated

Correctness: It is a logical consequence of the description of the algorithm given above and the comments on the actual code.

Complexity: The loop iterates for n times, which gives us O(n) time complexity.

\*It is interesting to note that Algorithm5 is not only faster than algorithm4 but also far simpler too. This is a recurring pattern in algorithms: **algorithms that are optimal/very fast are often simpler than slower algorithms**.

Algorithm6:This algorithm gives us all the consecutive(in the sense of descending sum) non-overlapping max subarrays, in the order they appear on the original array. In other words, we get the max subarray, the next greatest non-overlapping subarray, the greatest non-overlapping subarray after that etc. in the order they appear. Empty subarrays or subarrays with negative sum are not allowed in the solution. For example, if   
A=(8,-10,5,30,7,-25,20), then the algorithm would return: (8), (5,30,7), (20).

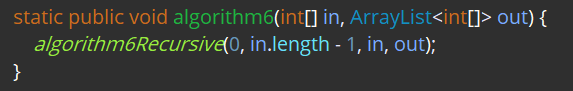
We use any existing MSP algorithm to find the max subarray of an array, and then recursively divide the initial array to two smaller arrays that do not overlap with the max subarray (one on the left and one on the right) and apply the same function to each of the smaller arrays.

The algorithm is implemented using recursion.

Java implementation:  
A screenshot of a computer

Description automatically generatedCorrectness: It is a logical consequence of the description of the algorithm given above and the comments on the actual code.

Complexity: The complexity analysis is quite similar to Quicksort complexity analysis (in fact the whole algorithm reminds us a lot of Quicksort; like we find q in quicksort we find the max subarray and its indexes in Algorithm6, and like we divide the input array in Quicksort in left and right parts according to q we divide the input array in left and right parts according to the indexes of max subarray in Algorithm6. If we use an algorithm of complexity O(n^x) to solve the consecutive MSPs, then the total complexity is O( (n^x)\*n )=O(n^(x+1) ) (the proof is quite similar to Quicksort analysis).

\*Similarly to algorithm4, to make the function more user-friendly we add an additional method:  


Algorithm7:This algorithm functions similarly to algorithm6, with the difference being in the ordering of the max subarrays. The subarrays are ordered from maximum to minimum.

The function checks for the next max subarray again and again “extracting” each solution from the initial subarray, until the next solution is the empty array. Extraction is in practice implemented by setting each element of the initial array that is part of the subarray as -inf( in Java, for nearly all intents and purposes, -inf is the minimum (negative) integer. Since every solution found before the empty array has positive sum, no solution before the empty array can contain one of these “extracted” elements. This means that no element can be “extracted” twice, or equivalently means that all the max subarrays are non-overlapping.

The algorithm is implemented using recursion (in fact tail recursion, which means that the recursive call happens in the last command of the recursive function).

Java implementation:  
A screenshot of a cell phone

Description automatically generated

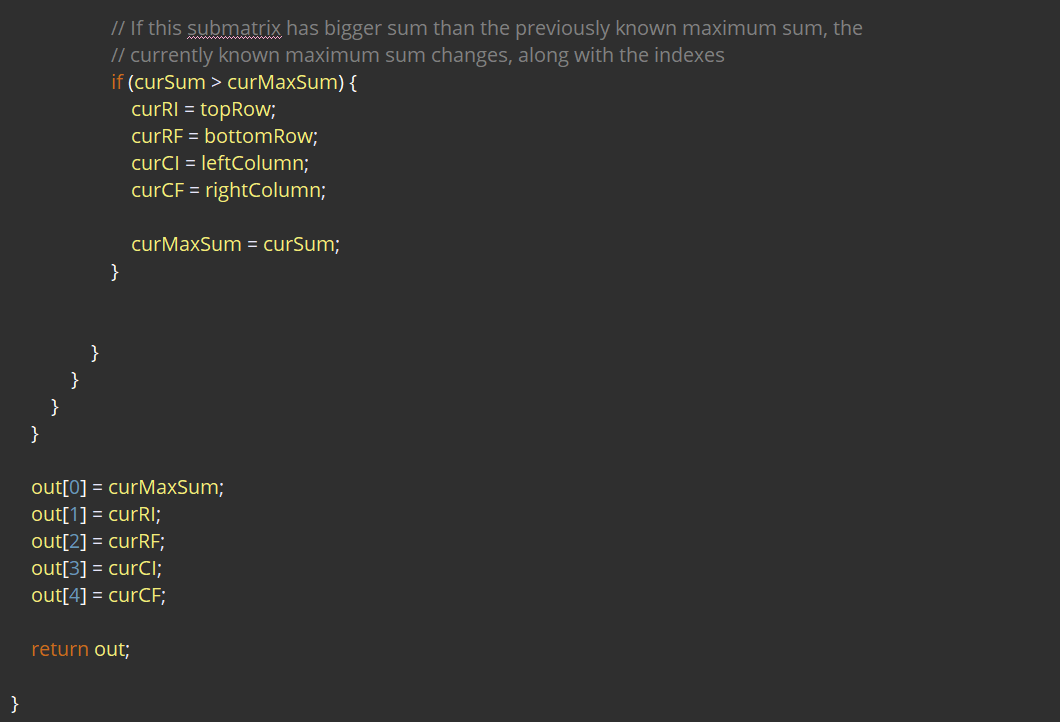
Correctness: It is a logical consequence of the description of the algorithm given above and the comments on the actual code.

Complexity: In the worst case the array will consist of consecutive positive and negative numbers, and all the solutions will be 1-element. This means we will have n/2 solutions and thus n recursions. If we use an algorithm of complexity O(n^x) to solve the consecutive MSPs, then the total complexity is O( (n^x)\*n )=O(n^(x+1) ), same as Algorithm6.

**Algorithm8:**This algorithm solves the 2D MSP via the brute-force approach. Refer to the first paragraph of **History** to see what constitutes a solution

\* Here , similarly to the 1D MSP, we allow the empty matrix as a solution, and in this case we consider: S=0, RI=0, RF=0, CI=0, CF=0 (R stands for row, C for column, I for initial and F for final)

The algorithm, implemented in Java, is as follows:  
A screenshot of a cell phone

Description automatically generated  


Correctness: Every submatrix is compared and thus their maximum will be the solution

Complexity: First loop will be executed R times. Second loop will be executed R-1, R-2, …, 1 times. Third loop will be executed C times. Fourth loop will be executed C-1, C-2, …, 1 times. Fifth loop is executed at worst R times. Sixth loop is executed at worst C times. Thus we have O[(R^3)\*(C^3)].

\*For a square matrix: O[(n^6)] (n=R=C)

Algorithm9:  
In this algorithm we first transform the 2D MSP problem to the 1D and then solve it using any of the 1D MSP algorithms.

Here we see a technique commonly encountered in engineering. When we have to solve a seemingly difficult problem we often try to transform it to another problem, more easily solvable (as we say we transform the problem on another domain). After solving the second problem we can convert our solution to a solution for the first problem. This technique might appear the same with the extension of a solution to a problem of smaller size we saw in Algorithm5. However this is not the case. Here we do not reduce the size of the problem. We just convert the original problem to a problem of similar size but with an easier/known solution.This is the main principle behind derivation/integration taught in school or Fourier/Laplace/Z analysis that is taught to engineers.

The algorithm considers all the possible pairs of CIs and CFs. For each possible pair we find the max submatrix for any possible pair of RIs and RFs. We then find the maximum of all these submatrices, which will of course be the total max submatrix. But how do we find the max submatrix for a specific pair of CIs and CFs? This is far better illustrated than described with words (another pattern we often see in algorithm design).

Let’s assume the original matrix is 5x5 and we want to find out the max submatrix for CI=1 and CF=3. This submatrix will be in the yellow area:  
A picture containing building

Description automatically generated

Some candidates for the solution are the pink submatrices below:  
A picture containing building

Description automatically generatedA picture containing building, shoji

Description automatically generatedA picture containing building, shoji

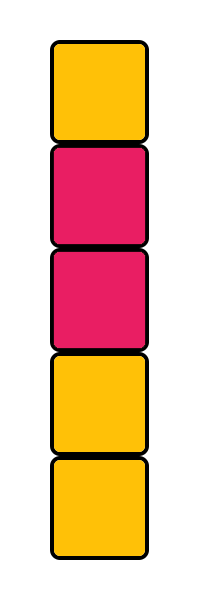
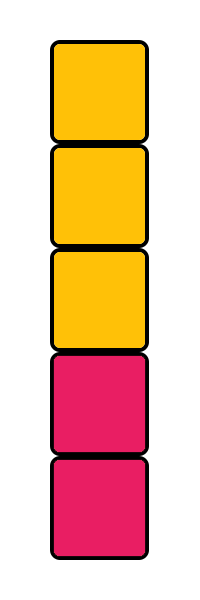
Description automatically generated  
Caution! We look only for submatrices that start at CI and end at CF. This means that for every possible RI and RF all the cells between CI and CF will be considered. In other words, the solution can either contain a whole row or not contain the row at all. Therefore, submatrices like:  
A picture containing building, shoji

Description automatically generatedA picture containing building, shoji

Description automatically generated  
, are not allowed (In case you are wondering, yes, the two submatrices above can be a solution to the total problem. But if they are, they will appear as max submatrices when CI=2, CF=3 and CI=1, CF=3 respectively. We do not consider them here).

Since every row can only exist on a solution as a whole, we reduce (or compress if you prefer) each row to its contribution to the sum. Since, the rows can only be continuous (by this I mean that each row of the solution has to be next to another row of the solution) I have to maximize the sum of a continuous array of numbers, those numbers being the contributions of the rows.

Graphically, the yellow area and the possible solutions illustrated above are now represented as:  
 A picture containing yellow, looking, light

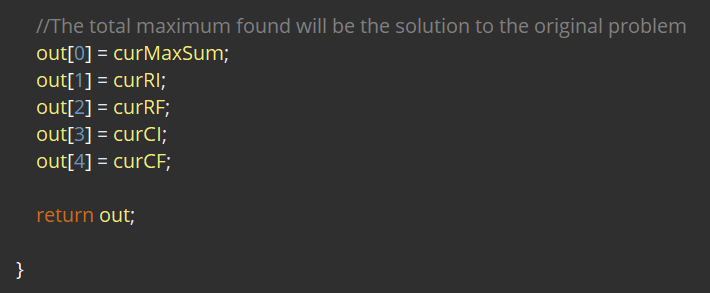
Description automatically generated   A picture containing toiletry

Description automatically generated

Does this remind you of something? It should! It is the 1D MSP, for which we have already developed a lot of algorithms.

*The rest of the page remains blank so all of the implementation code will be on the same page.*

Without further ado, here is the implementation in Java:  
A screenshot of a cell phone

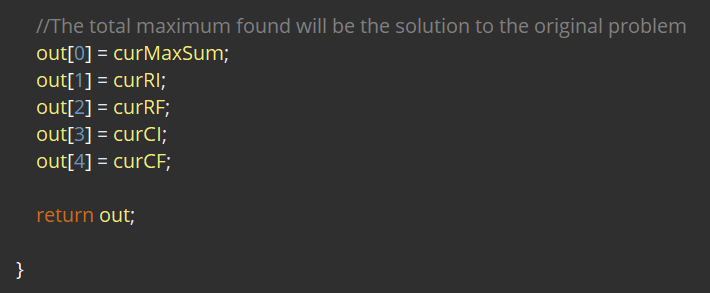
Description automatically generated  


Correctness: It is a logical consequence of the description of the algorithm given above and the comments on the actual code.

Complexity:

Algorithm10:It is more or else the same as Algorithm9, but with an important tweak, inspired by Algorithm2. We will again save state to avoid recomputation.

Java implementation:  
A screenshot of a cell phone

Description automatically generated  


Correctness:

Complexity:  
  
Performance:

Applications:According to my personal, and admittedly limited, research there have not been many real applications of the MSP, neither in industry nor in academia. In a lot of sources it is stated otherwise, but I could not verify those claims. However, the applications mentioned in other sources are indeed interesting and give us ideas about how we could apply the MSP to real life problems.

The applications are divided according to the field of studies they pertain to. Three field of studies will be mentioned: Bioinformatics, signal processing and data science. For each of them I will avoid talking about things I do not fully comprehend (a really dangerous practice) and I will solve a specific problem (implementation code will be given in Java). These problems will illuminate some further details about the applications of the MSP and also act as inspiration.

Bioinformatics:Bionformatics is an [interdisciplinary](https://en.wikipedia.org/wiki/Interdisciplinary) domain that uses software and/or hardware to analyse biological data. Out of the three fields, it is the one for which I found the most real applications, with the caveat that all of them were relatively old. The most important one is protein folding.

To understand protein folding we first have to understand proteins. Proteins are relatively large molecules found on nearly every living organism. Proteins are encoded in genes and are directly or indirectly responsible for most functions within the organisms. A protein’s functions are determined fully by its conformation, but what exactly determines its conformation?

Proteins’ conformation is determined fully by 5 structure levels, corresponding to five degrees of freedom. They are called primary, secondary, tertiary, quaternary and quinary structure. These structures are in turn determined by the protein’s composition. The fundamental building blocks of proteins are molecules called amino acids, and their sequence (1D space) is the primary structure. Some amino acids are grouped together, consisting a chain called peptide. Peptides fold (2D space) giving the protein its secondary structure (some of the most common secondary structures are called α-helix and β-sheet but others exist). Peptides are combined to form a protein. Within the protein there are areas called rhe formation of the protein is the quaternary structure. Finally, the quinary structure refers simply to the surface of the protein.

Thus, it is clear that to discover a protein’s function we have to find its conformation. An important step towards this direction is to find the functional domains of the protein. It has been found that functional domains are characterized by hydrophobicity (tendency to be repelled by water). Also the hydrophobicity of every amino acid that is encoded in proteins in humans has been experimentally determined. Their hydrophobicities (normalized) are as follows:

|  |  |  |  |
| --- | --- | --- | --- |
| **Full name** | **1-letter code** | **3-letter code** | **Index** |
| Alanine | A | Ala | +1.8 |
| Arginine | R | Arg | -4.5 |
| Asparagine | N | Asn | -3.5 |
| Aspartic acid | D | Asp | -3.5 |
| Cysteine | C | Cys | +2.5 |
| Glutamine | Q | Gln | -3.5 |
| Glutamic acid | E | Glu | -3.5 |
| Glycine | G | Gly | -0.4 |
| Histidine | H | His | -3.2 |
| Isoleucine | I | Ile | +4.5 |
| Leucine | L | Leu | +3.8 |
| Lysine | K | Lys | -3.9 |
| Methionine | M | Met | +1.9 |
| Phenylalaline | F | Phe | +2.8 |
| Proline | P | Pro | -1.6 |
| Serine | S | Ser | -0.8 |
| Threonine | T | Thr | -0.7 |
| Trystophan | W | Trp | -0.9 |
| Tyrosine | Y | Tyr | -1.3 |
| Valine | V | Val | +4.2 |

\*The range goes from +4.5 (most hydrophobic) to -4.5 (least hydrophobic- hydrophylic)  
\*Although there are infinitely many proteins, it is projected that the human body contains between 10.000 and several billion different proteins. The RCSB protein data bank today contains around 160.000 proteins with identified structure.   
\*Humans can produce 10 of the 20 amino acids. The others must by supplied by food  
\*In the human body there are actually two more amino acids, [Selenocysteine](https://en.wikipedia.org/wiki/Selenocysteine) and [Pyrrolysine](https://en.wikipedia.org/wiki/Pyrrolysine), that are not encoded in proteins by the genes.

Given the table above and the sequence of amino acids of a protein, we can construct an array that contains the hydrophobicities of each amino acid. We would expect that continuous areas that have the highest sum are functional domains. This problem is nothing more that the 1D MSP.

\*Folding@Home is a really interesting distributed computing project that tackles many bioinformatics problems, protein folding being the main one. Volunteers can offer their computer so it runs calculations for the project while the machine is idling. This leads to a combined computational power of approximately 100 PFLOPS. For a frame of reference, Summit, currently the fastest super computer in the world, has a speed of around 150 PFLOPS.

Problem:

Signal processing:

Problem:

Data science:

Problem:

Further work:1)Another technique that is commonly encountered in algorithm design (or really generally in science) is that when we have to solve a seemingly difficult problem we usually construct a far simpler problem that shares the first’s main attributes. After solving the second problem, we re-examine the first one in order to see if we can apply the techniques learned in the solving of the second one. We have to keep in mind that even if we find an optimal solution for the simpler problem, this does not necessarily mean that the total solution is optimal. Following this approach derive an algorithm similar to Algorithm4 for the 2D MSP.

2) Study Ruzzo–Tompa algorithm

3) Study the 3D MSP. Follow the same approach used when going from 1D to 2D.

4) Study 1D and 2D MSP with length constraints

5) Study the 1D MSP for the first k subarrays

6) Study the max product subarray problem. This may demand a whole new set of notes.

7) Study the Maximum Average Subarray problem. This may demand a whole new set of notes.

8) Other bioinformatics applications

\*Further workis as much a to-do list for me as a calling to the interested reader for help. If you want to contribute to these notes you can either implement something from Further work, implement something you thought of or came across that interests you and is relevant to the MSP, or simply suggest improvements or make corrections. For contact details see below.

Sources:1) Programming Pearls, by Jon Bentley (1st edition) – Column 7 : This set of notes are heavily based on this source.

2) Sequential and Parallel Algorithms for the Generalized Maximum Subarray Problem, by Sung Eun Bae – Chapter 1: This source details perfectly the applications of the MSP and has some more recent progress about the problem (in comparison to source 1).

3) <https://www.interviewbit.com/problems/max-sum-contiguous-subarray/> (Visited: 15/3/2019) : Companies that ask the MSP as an interview question.

4) Simple Method for Displaying the Hydropathic Character of a Protein by Jack Kyte and Russel Doolittle: Hydropathy index for the amino acids that are encoded in proteins in humans

Contact:e-mail: gvlassis@mailbox.org

\*Do not hesitate to contact me if you have any questions