2

A warm-up!

"Everything should be made as simple as possible, but not simpler."
Albert Einstein

Let us consider the following problem, surprisingly simple in its statement but not that much for what concerns the design of its optimal solution.

Problem. *We are given the performance of a stock at* NYSE *expressed as a sequence of* day-by-day differences *of its quotations. We wish to determine the best buy-&-sell strategy for that stock, namely the pair of days $\langle b, s\rangle$ that would have maximized our revenues if we would have bought the stock at (the beginning of) day b and sold it at (the end of) day s.*

The specialty of this problem is that it has a simple formulation, which finds many other useful variations and applications. We will comment on them at the end of this lecture, now we content ourselves by mentioning that we are interested in this problem because it admits a sequence of algorithmic solutions of increasing sophistication and elegance, which imply a significant reduction in their time complexity. The ultimate result will be a linear-time algorithm, i.e. linear in the number n of stock quotations. This algorithm is *optimal* in terms of the number of executed steps, because all day-by-day differences must be looked at in order to determine if they must be included or not in the optimal solution, actually, one single difference could provide a one-day period worth of investment! Surprisingly, the optimal algorithm will exhibit the simplest pattern of memory accesses— it will execute a single scan of the available stock quotations— and thus it will offer a *streaming behavior*, particularly useful in a scenario in which the granularity of the buy-&-sell actions is not restricted to full-days and we must possibly compute the optimal time-window *on-the-fly* as quotations oscillate. More than this, as we commented in the previous lecture, this algorithmic scheme is optimal in terms of I/Os and *uniformly* over all levels of the memory hierarchy. In fact, because of its streaming behavior, it will execute n/B I/Os independently of the disk-page size B, which may be thus unknown to the underlying algorithm. This is the typical feature of the so called *cache-oblivious algorithms* [4], which we will therefore introduce at the right point of this lecture.

This lecture will be the prototype of what you will find in the next pages: a simple problem to state, with few elegant solutions and challenging techniques to teach and learn, together with several intriguing extensions that can be posed as exercises to the students or as puzzles to tempt your mathematical skills!

Let us now dig into the technicalities, and consider the following example. Take the case of 11 days of exchange for a given stock, and assume that $D[1, 11] = [+4, -6, +3, +1, +3, -2, +3, -4, +1,$

$-9, +6]$ denotes the day-by-day differences of quotations of that stock. It is not difficult to convince yourself that the gain of buying the stock at day x and selling it at day y is equal to the sum of the values in the sub-array $D[x, y]$, namely the sum of all its fluctuations. As an example, take $x = 1$ and $y = 2$, the gain is $+4 - 6 = -2$, and indeed we would loose 2 dollars in buying the morning of the first day and selling the stock at the end of the second day. Notice that the starting value of the stock is not crucial for determining the best time-interval of our investment, what is important are its variations. In other words, the problem stated above boils down to determine the sub-array of $D[1, n]$ which maximizes the sum of its elements. In the literature this problem is indeed known as the *maximum sub-array sum* problem.

Problem Abstraction. *Given an array $D[1, n]$ of positive and negative numbers, we want to find the sub-array $D[b, s]$ which maximizes the sum of its elements.*

It is clear that if all numbers are positive, then the optimal sub-array is the entire D: this is the case of an always increasing stock price, and indeed there is no reason to sell it before the last day! Conversely, if all numbers are negative, then we can pick the one-element window containing the largest (negative) value: if you are imposed to buy this poor stock, then do it in the day it looses the smallest value and sell it soon! In all other cases, it is not at all clear where the optimum sub-array is located. In the example, the optimum spans $D[3, 7] = [+3, +1, +3, -2, +3]$ and has gain $+8$ dollars. This shows that the optimum neither includes the best exploit of the stock (i.e. $+6$) nor it consists of positive values only. The *structure* of the optimum sub-array is not simple but, surprisingly enough, not very complicated as we will show in Section 2.3.

2.1 A cubic-time algorithm

We start by considering an inefficient solution which translates in pseudo-code the formulation of the problem given above. This algorithm is detailed in Figure 2.1, where the pair of variables $<b_o, s_o>$ identifies the current sub-array of maximum sum, whose value is stored in `MaxSum`. Initially `MaxSum` is set to the dummy value $-\infty$, so that it is immediately changed whenever the algorithm executes Step 8 for the first time. The core of the algorithm examines all possible sub-arrays $D[b, s]$ (Steps 2-3) computing for each of them the sum of their elements (Steps 4-7). If a sum larger than the current maximal value is found (Steps 8-9), then `TmpSum` and its corresponding sub-array are stored in `MaxSum` and $<b_o, s_o>$, respectively.

Algorithm 2.1 The cubic-time algorithm

```
1:  MaxSum = −∞
2:  for (b = 1; b ≤ n; b++) do
3:      for (s = b; s ≤ n; s++) do
4:          TmpSum = 0
5:          for (i = b; i ≤ s; i++) do
6:              TmpSum+ = D[i];
7:          end for
8:          if (MaxSum < TmpSum) then
9:              MaxSum = TmpSum; b_o = b; s_o = s;
10:         end if
11:     end for
12: end for
13: return ⟨MaxSum, b_o, s_o⟩;
```

The correctness of the algorithm is immediate, since it checks all possible sub-arrays of $D[1,n]$ and selects the one whose sum of its elements is the largest (Step 8). The time complexity is cubic, i.e. $\Theta(n^3)$, and can be evaluated as follows. Clearly the time complexity is upper bounded by $O(n^3)$ because we can form no more than $\frac{n^2}{2}$ pairs <b,s> out of n elements,[1] and n is an upper-bound to the cost of computing the sum of each sub-array. Let us now show that the time cost is also $\Omega(n^3)$, so concluding that the time complexity is strictly cubic. To show this lower bound, we observe that $D[1,n]$ contains $(n-L+1)$ sub-arrays of length L, and thus the cost of computing the sum for all of their elements is $(n-L+1)\times L$. Summing over all values of L, we would get the exact time complexity. But here we are interested in a lower bound, so we can evaluate that cost just for the subset of sub-arrays whose length L is in the range $[n/4, n/2]$. For each such L, we have that $n-L+1 > n/2$ and $L \geq n/4$, so the cost above is $(n-L+1)\times L > n^2/8$. Since we have $\frac{n}{2} - \frac{n}{4} + 1 > n/4$ of those Ls, the total cost for analysing that subset of sub-arrays is lower bounded by $n^3/32 = \Omega(n^3)$.

It is natural now to ask ourselves how much fast in practice is the designed algorithm. We implemented it in Java and tested on a commodity PC. As n grows, its time performance reflects in practice its cubic time complexity, evaluated in the RAM model. More precisely, it takes about 20 seconds to solve the problem for $n = 10^3$ elements, and about 30 hours for $n = 10^5$ elements. Too much indeed if we wish to scale to very large sequences (of quotations), as we are aiming for in these lectures.

2.2 A quadratic-time algorithm

The key inefficiency of the cubic-time algorithm resides in the execution of Steps 4-7 of the pseudo-code in Figure 2.1. These steps re-compute from scratch the sum of the sub-array $D[b,s]$ each time its extremes change in Steps 2-3. But if we look carefully at the **for**-cycle of Step 3 we notice that the size s is incremented by one unit at a time from the value b (one element sub-array) to the value n (the longest possible sub-array that starts at b). Therefore, from one execution to the next one of Step 3, the sub-array to be summed changes from $D[b,s]$ to $D[b,s+1]$. It is thus immediate to conclude that the new sum for $D[b,s+1]$ does not need to be recomputed from scratch, but can be computed *incrementally* by just adding the value of the new element $D[s+1]$ to the current value of `TmpSum` (which inductively stores the sum of $D[b,s]$). This is exactly what the pseudo-code of Figure 2.2 implements: its two main changes with respect to the cubic algorithm of Figure 2.1 are in Step 3, that nulls `TmpSum` every time b is changed (because the sub-array starts again from length 1, namely $D[b,b]$), and in Step 5, that implements the incremental update of the current sum as commented above. Such small changes are worth of a saving of $\Theta(n)$ additions per execution of Step 2, thus turning the new algorithm to have quadratic-time complexity, namely $\Theta(n^2)$.

More precisely, let us concentrate on counting the number of additions executed by the algorithm of Figure 2.2; this is the prominent operation of this algorithm so that its evaluation will give us an estimate of its total number of steps. This number is[2]

$$\sum_{b=1}^{n}(1+\sum_{s=b}^{n}1) = \sum_{b=1}^{n}(1+(n-b+1)) = n\times(n+2) - \sum_{b=1}^{n} b = n^2 + 2n - \frac{n(n-1)}{2} = O(n^2).$$

This improvement is effective also in practice. Take the same experimental scenario of the previous section, this new algorithm requires less than 1 second to solve the problem for $n = 10^3$

[1] For each pair $< b, s >$, with $b \leq s$, $D[b,s]$ is a possible sub-array, but $D[s,b]$ is not.

[2] We use below the famous formula, discovered by the young Gauss, to compute the sum of the first n integers.

Algorithm 2.2 The quadratic-time algorithm

```
1:  MaxSum = −∞;
2:  for (b = 1; b ≤ n; b++) do
3:      TmpSum = 0;
4:      for (s = b; s ≤ n; s++) do
5:          TmpSum += D[s];
6:          if (MaxSum < TmpSum) then
7:              MaxSum = TmpSum; b_o = b; s_o = s;
8:          end if
9:      end for
10: end for
11: return ⟨MaxSum, b_o, s_o⟩;
```

elements, and about 28 minutes to manage 10^6 elements. This means that the new algorithm is able to manage more elements in "reasonable" time. Clearly, these timings and these numbers could change if we use a different programming language (Java, in the present example), operating system (Windows, in our example), and processor (the old Pentium IV, in our example). Nevertheless we believe that they are interesting anyway because they provide a concrete picture of what it does mean a theoretical improvement like the one we showed in the above paragraphs on a real situation. It goes without saying that the life of a coder is typically not so easy because theoretically-good algorithms many times hide so many details that their engineering is difficult and big-O notation often turn out to be not much "realistic". Do not worry, we will have time in these lectures to look at these issues in more detail.

2.3 A linear-time algorithm

The final step of this lecture is to show that the maximum sub-array sum problem admits an elegant algorithm that processes the elements of $D[1, n]$ is a streaming fashion and takes the *optimal* $O(n)$ time. We could not aim for more!

To design this algorithm we need to dig into the structural properties of the optimal sub-array. For the purpose of clarity, we refer the reader to Figure 2.1 below, where the optimal sub-array is assumed to be located at two positions $b_o \leq s_o$ in the range $[1, n]$.

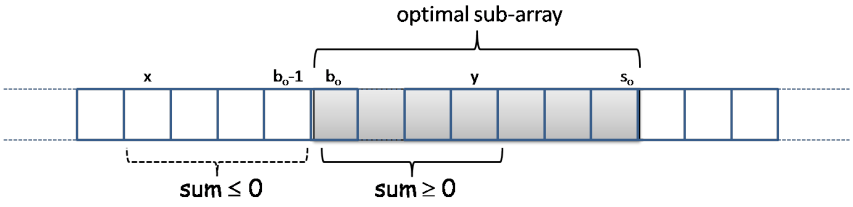

FIGURE 2.1: An illustrative example of Properties 1 and 2.

Let us now take a sub-array that starts before b_o and ends at position $b_o - 1$, say $D[x, b_o - 1]$. The sum of the elements in this sub-array cannot be positive because, otherwise, we could merge it with the (adjacent) optimal sub-array and thus get the longer sub-array $D[x, s_o]$ having sum *larger than*

the one obtained with the (claimed) optimal $D[b_o, s_o]$. So we can state the following:

Property 1. The sum of the elements in a sub-array $D[x, b_o - 1]$, with $x < b_o$, cannot be (strictly) positive.

Via a similar argument, we can consider a sub-array that is a prefix of the optimal $D[b_o, s_o]$, say $D[b_o, y]$ with $y \leq s_o$. This sub-array cannot have negative sum because, otherwise, we could drop it from the optimal solution and get a shorter array, namely $D[y + 1, s_o]$ having sum larger than the one obtained by the (claimed) optimal $D[b_o, s_o]$. So we can state the following other property:

Property 2. The sum of the elements in a sub-array $D[b_o, y]$, with $y \leq s_o$, cannot be (strictly) negative.

We remark that any one of the sub-arrays considered in the above two properties might have sum equal to zero. This would not affect the optimality of $D[b_o, s_o]$, it could only introduce other optimal solutions being either longer or shorter than $D[b_o, s_o]$.

Let us illustrate these two properties on the array $D[1, 11] = [+4, -6, +3, +1, +3, -2, +3, -4, +1, -9, +6]$. Here the optimum sub-array is $D[3, 7] = [+3, +1, +3, -2, +3]$. We note that $D[x, 2]$ is always negative (Prop. 1), in fact for $x = 1$ the sum is $+4 - 6 = -2$ and for $x = 2$ the sum is -6. On the other hand the sum of all elements in $D[3, y]$ is positive for all prefixes of the optimum sub-array (Prop. 2), namely $y \leq 7$. We also point out that the sum of $D[3, y]$ is positive even for some $y > 7$, take for example $D[3, 8]$ for which the sum is 4 and $D[3, 9]$ for which the sum is 5. Of course, this does not contradict Prop. 2.

Algorithm 2.3 The linear-time algorithm

```
1:  MaxSum = −∞
2:  TmpSum = 0; b = 1;
3:  for (s = 1; s ≤ n; s++) do
4:      TmpSum += D[s];
5:      if (MaxSum < TmpSum) then
6:          MaxSum = TmpSum; b_o = b; s_o = s;
7:      end if
8:      if (TmpSum < 0) then
9:          TmpSum = 0; b = s + 1;
10:     end if
11: end for
12: return ⟨MaxSum, b_o, s_o⟩;
```

The two properties above lead to the simple Algorithm 2.3. It consists of one unique **for**-cycle (Step 3) which keeps in `TmpSum` the sum of a sub-array ending in the currently examined position s and starting at some position $b \leq s$. At any step of the **for**-cycle, the candidate sub-array is extended one position to the right (i.e. s++), and its sum `TmpSum` is increased by the value of the current element $D[s]$ (Step 4). Since the current sub-array is a candidate to be the optimal one, its sum is compared with the current optimal value (Step 5). Then, according to Prop. 1, if the sub-array sum is negative, the current sub-array is discarded and the process "restarts" with a new sub-array beginning at the next position $s + 1$ (Steps 8-9). Otherwise, the current sub-array is extended to the right, by incrementing s. The tricky issue here is to show that the optimal sub-array is checked in Step 5, and thus stored in $< b_o, s_o >$. This is not intuitive at all because the algorithm is checking n sub-arrays out of the $\Theta(n^2)$ possible ones, and we want to show that this (minimal) subset of

candidates actually contains the optimal solution. This subset is *minimal* because these sub-arrays form a *partition* of $D[1, n]$ so that every element belongs to one, and only one checked sub-array. Moreover, since every element must be analyzed, we cannot discard any sub-array of this partition without checking its sum!

Before digging into the formal proof of correctness, let us follow the execution of the algorithm over the array $D[1, 11] = [+4, -6, +3, +1, +3, -2, +3, -4, +1, -9, +6]$. Remember that the optimum sub-array is $D[3, 7] = [+3, +1, +3, -2, +3]$. We note that $D[x, 2]$ is negative for $x = 1, 2$, so the algorithm surely zeroes the variable `TmpSum` when $s = 2$ in Steps 8-9. At that time, b is set to 3 and `TmpSum` is set to 0. The subsequent scanning of the elements $s = 3, \ldots, 7$ will add their values to `TmpSum` which is always positive (see above). When $s = 7$, the examined sub-array coincides with the optimal one, we thus have `TmpSum` = 8, so Step 5 stores the optimum location in $< b_o, s_o >$. It is interesting to notice that, in this example, the algorithm does not re-start the value of `TmpSum` at the next position $s = 8$ because it is still positive (namely, `TmpSum` = 4); this means that the algorithm will examine sub-arrays longer than the optimal one, but all having a smaller sum, of course. The next re-starting will occur at position $s = 10$ where `TmpSum` $= -4$.

It is easy to realize that the time complexity of the algorithm is $O(n)$ because every element is examined just once. More tricky is to show that the algorithm is correct, which actually means that Steps 4 and 5 eventually compute and check the optimal sub-array sum. To show this, it suffices to prove the following two facts: (i) when $s = b_o - 1$, Step 8 resets b to b_o; (ii) for all subsequent positions $s = b_o, \ldots, s_o$, Step 8 never resets b so that it will eventually compute in `TmpSum` the sum of all elements in $D[b_o, s_o]$, whenever $s = s_o$. It is not difficult to see that Fact (i) derives from Property 1, and Fact (ii) from Property 2.

This algorithm is very fast in the same experimental scenario mentioned in the previous sections, it takes less than 1 second to process millions of quotations. A truly scalable algorithm, indeed, with many nice features that make it appealing also in a hierarchical-memory setting. In fact, this algorithm scans the array D from left to right and examines each of its elements just once. If D is stored on disk, these elements are fetched in internal memory one page at a time. Hence the algorithm executes n/B I/Os, which is *optimal*. It is interesting also to note that the design of the algorithm does not depend on B (which indeed does not appear in the pseudo-code), but we can evaluate its I/O-complexity in terms of B. Hence the algorithm takes n/B optimal I/Os independently of the the page size B, and thus subtly on the hierarchical-memory levels interested by the algorithm execution. Decoupling the use of the parameter B between algorithm design and algorithm analysis is the key issue of the so called *cache-oblivious algorithms*, which are a hot topic of algorithmic investigation nowadays. This feature is achieved here in a basic (trivial) way by just adopting a scan-based approach. The literature [4] offers more sophisticated results regarding the design of cache-oblivious algorithms and data structures.

2.4 Another linear-time algorithm

There exists another optimal solution to the maximum sub-array sum problem which hinges on a different algorithm design. For simplicity of exposition, let us denote by $Sum_D[y', y'']$ the sum of the elements in the sub-array $D[y', y'']$. Take now a selling time s and consider all sub-arrays that end at s: namely we are interested in sub-arrays having the form $D[x, s]$, with $x \leq s$. The value $Sum_D[x, s]$ can be expressed as the difference between $Sum_D[1, s]$ and $Sum_D[1, x - 1]$. Both of these sums are indeed *prefix*-sums over the array D and can be computed in linear time. As a result, we can rephrase our maximization problem as follows:

$$\max_s \max_{b \leq s} \mathrm{Sum}_D[b, s] = \max_s \max_{b \leq s} (\mathrm{Sum}_D[1, s] - \mathrm{Sum}_D[1, b - 1]).$$

We notice that if $b = 1$ the second term refers to the empty sub-array $D[1, 0]$; so we can assume that $Sum_D[1, 0] = 0$. This is the case in which $D[1, s]$ is the sub-array of maximum sum among all the sub-arrays ending at s (so no prefix sub-array $D[1, b-1]$ is dropped from it).

The next step is to pre-compute all prefix sums $P[i] = Sum_D[1, i]$ in $O(n)$ time and $O(n)$ space via a scan of the array D: Just notice that $P[i] = P[i-1] + D[i]$, where we set $P[0] = 0$ in order to manage the special case above. Hence we can rewrite the maximization problem in terms of the array P, rather than Sum_D: $\max_{b \le s}(P[s] - P[b-1])$. The cute observation now is that we can decompose the max-computation into a max-min calculation over the two variables b and s

$$\max_s \max_{b \le s}(P[s] - P[b-1]) = \max_s(P[s] - \min_{b \le s} P[b-1]).$$

The key idea is that we can move $P[s]$ outside the inner max-calculation because it does not depend on the variable b, and then change a max into a min because of the negative sign. The final step is then to pre-compute the minimum $\min_{b \le s} P[b-1]$ for all positions s, and store it in an array $M[0, n-1]$. We notice that, also in this case, the computation of $M[i]$ can be performed via a single scan of P in $O(n)$ time and space: set $M[0] = 0$ and then derive $M[i]$ as $\min\{M[i-1], P[i]\}$. Given M, we can rewrite the formula above as $\max_s(P[s] - M[s-1])$ which can be clearly computed in $O(n)$ time given the two arrays P and M. Overall this new approach takes $O(n)$ time and $O(n)$ extra space.

As an illustrative example, consider again the array $D[1, 11] = [+4, -6, +3, +1, +3, -2, +3, -4, +1, -9, +6]$. We have that $P[0, 11] = [0, +4, -2, +1, +2, +5, +3, +6, +2, +3, -6, 0]$ and $M[0, 10] = [0, 0, -2, -2, -2, -2, -2, -2, -2, -2, -6]$. If we compute the difference $P[s] - M[s-1]$ for all $s = 1, \ldots, n$, we obtain the sequence of values $[+4, -2, +3, +4, +7, +5, +8, +4, +5, -4, +6]$, whose maximum (sum) is $+8$ that occurs (correctly) at the (final) position $s = 7$. It is interesting to note that the left-extreme b_o of the optimal sub-array could be derived by finding the position $b_o - 1$ where $P[b_o - 1]$ is minimum: in the example, $P[2] = -2$ and thus $b_o = 3$.

Algorithm 2.4 Another linear-time algorithm

```
1:  MaxSum = −∞; b_o = 1;
2:  TmpSum = 0; MinTmpSum = 0;
3:  for (s = 1; s ≤ n; s++) do
4:      TmpSum += D[s];
5:      if (MaxSum < TmpSum − MinTmpSum) then
6:          MaxSum = TmpSum − MinTmpSum; s_o = s; b_o = b_tmp;
7:      end if
8:      if (TmpSum < MinTmpSum) then
9:          MinTmpSum = TmpSum; b_tmp = s + 1;
10:     end if
11: end for
12: return ⟨MaxSum, b_o, s_o⟩;
```

We conclude this section by noticing that the proposed algorithm executes three passes over the array D, rather than the single pass of Algorithm 2.3. It is not difficult to turn this algorithm to make *one-pass* too. It suffices to deploy the associativity of the min/max functions, and use two variables that inductively keep the values of $P[s]$ and $M[s-1]$ as the array D is scanned from left to right. Algorithm 2.4 implements this idea by using the variable `TmpSum` to store $P[s]$ and the variable `MinTmpSum` to store $M[s-1]$. This way the formula $\max_s(P[s] - M[s-1])$ is evaluated incrementally for $s = 1, \ldots, n$, thus avoiding the two passes for pre-calculating the arrays P and

M and the extra-space needed to store them. One pass over D is then enough, and so we have re-established the nice algorithmic properties of Algorithm 2.3 but with a completely different design!

2.5 Few interesting variants$^{\infty}$

As we promised at the beginning of this lecture, we discuss now few interesting variants of the maximum sub-array sum problem. For further algorithmic details and formulations we refer the interested reader to [1, 2]. Note that this is a challenging section, because it proposes an algorithm whose design and analysis are sophisticated!

Sometimes in the bio-informatics literature the term "sub-array" is substituted by "segment", and the problem takes the name of "maximum-sum segment problem". In the bio-context the goal is to identify segments which occur inside DNA sequences (i.e. strings of four letters A, T, G, C) and are *rich* of G or C nucleotides. Biologists believe that these segments are biologically significant since they predominantly contain genes. The mapping from DNA sequences to *arrays of numbers*, and thus to our problem abstraction, can be obtained in several ways depending on the objective function that models the *GC-richness* of a segment. Two interesting mappings are the following ones:

- Assign a penalty $-p$ to the nucleotides A and T of the sequence, and a reward $1-p$ to the nucleotides C and G. Given this assignment, the sum of a segment of length l containing x occurrences of C+G is equal to $x - p \times l$. Biologists think that this function is a good measure for the CG-richness of that segment. Interestingly enough, all algorithms described in the previous sections can be used to identify the CG-rich segments of a DNA sequence in linear time, according to this objective function. Often, however, biologists prefer to define a cutoff range on the length of the segments for which the maximum sum must be searched, in order to avoid the reporting of extremely short or extremely long segments. In this new scenario the algorithms of the previous sections cannot be applied, but yet linear-time optimal solutions are known for them (see e.g. [2]).
- Assign a value 0 to the nucleotides A and T of the sequence, and a value 1 to the nucleotides C and G. Given this assignment, the density of C+G nucleotides in a segment of length l containing x occurrences of C and G is x/l. Clearly $0 \leq x/l \leq 1$ and every single occurrence of a nucleotide C or G provides a segment with maximum density 1. Biologists consider this as an interesting measure of CG-richness for a segment, provided that a cutoff range on the length of the searched segments is imposed. This problem is more difficult than the one stated in the previous item, nevertheless it posses optimal (quasi-)linear time solutions which are much sophisticated and for which we refer the interested reader to the pertinent bibliography (e.g. [1, 3, 5]).

These examples are useful to highlight a *dangerous trap* that often occurs when abstracting a real problem: apparently small changes in the problem formulation lead to big jumps in the complexity of designing efficient algorithms for them. Think for example to the density function above, we needed to introduce a cutoff lower-bound to the segment length in order to avoid the trivial solution consisting of *single* nucleotides C or G! With this "small" change, the problem results more challenging and its solutions sophisticated.

Other subtle traps are more difficult to be discovered. Assume that we decide to circumvent the single-nucleotide outcome by searching for the the *longest* segment whose density is *larger than* a fixed value d. This is, in some sense, a complementary formulation of the problem stated in the second item above, because maximization is here on the segment length and a (lower) cut-off is imposed on the density value. Surprisingly it is possible to *reduce* this density-based problem to a

sum-based problem, in the spirit of the one stated in the first item above, and solved in the previous sections. Algorithmic reductions are often employed by researchers to re-use known solutions and thus do not re-discover again and again the "hot water". To prove this reduction it is enough to notice that:

$$\frac{\text{Sum}_D[x,y]}{y-x+1} = \sum_{k=x}^{y} \frac{D[k]}{y-x+1} \geq t \iff \sum_{k=x}^{y} (D[k]-t) \geq 0.$$

Therefore, subtracting to all elements in D the density-threshold t, we can turn the problem stated in the second item above into the one that asks for the *longest segment that has sum larger or equal than* 0. Be careful that if you change the request from the *longest segment* to the *shortest one* whose density is larger than a threshold t, then the problem becomes trivial again: Just take the single occurrence of a nucleotide C or G. Similarly, if we fix an upper bound S to the segment's sum (instead of a lower bound), then we can change the sign to all D's elements and thus turn the problem again into a problem with a lower bound $t = -S$. So let us stick on the following general formulation:

Problem. *Given an array $D[1,n]$ of positive and negative numbers, we want to find the* longest *segment in D whose sum of its elements is* larger or equal than *a fixed threshold t.*

We notice that this formulation is in some sense a complement of the one given in the first item above. Here we maximize the segment length and pose a lower-bound to the sum of its elements; there, we maximized the sum of the segment provided that its length was within a given range. It is nice to observe that the structure of the algorithmic solution for both problems is similar, so we detail only the former one and refer the reader to the literature for the latter.

The algorithm proceeds inductively by assuming that, at step $i = 1, 2, \ldots, n$, it has computed the longest sub-array having sum larger than t and occurring within $D[1, i-1]$. Let us denote the solution available at the beginning of step i with $D[l_{i-1}, r_{i-1}]$. Initially we have $i = 1$ and thus the inductive solution is the empty one, hence having length equal to 0. To move from step i to step $i+1$, we need to compute $D[l_i, r_i]$ by possibly taking advantage of the currently known solution.

It is clear that the new segment is either inside $D[1, i-1]$ (namely $r_i < i$) or it ends at position $D[i]$ (namely $r_i = i$). The former case admits as solution the one of the previous iteration, namely $D[l_{i-1}, r_{i-1}]$, and so nothing has to be done: just set $r_i = r_{i-1}$ and $l_i = l_{i-1}$. The latter case is more involved and requires the use of some special data structures and a tricky analysis to show that the total complexity of the solution proposed is $O(n)$ in space and time, thus turning to be optimal!

We start by making a simple, yet effective, observation:

FACT 2.1

If $r_i = i$ then the segment $D[l_i, r_i]$ must be strictly longer than the segment $D[l_{i-1}, r_{i-1}]$. This means in particular that l_i occurs to the left of position $L_i = i - (r_{i-1} - l_{i-1})$.

The proof of this fact follows immediately by the observation that, if $r_i = i$, then the current step i has found a segment that improves the previously known one. Here "improved" means "longer" because the other constraint imposed by the problem formulation is boolean since it refers to a lower-bound on the segment's sum. This is the reason why we can discard all positions within the range $[L_i, i]$, in fact they originate intervals of length shorter or equal than the previous solution $D[l_{i-1}, r_{i-1}]$.

Reformulated Problem. *Given an array $D[1,n]$ of positive and negative numbers, we want to find at every step the* smallest *index $l_i \in [1, L_i)$ such that $Sum_D[l_i, i] \geq t$.*

We point out that there could be many such indexes l_i, here we wish to find the *smallest* one because we aim at determining the *longest* segment.

At this point it is useful to recall that $\text{Sum}_D[l_i, i]$ can be re-written in terms of prefix-sums of array D, namely $\text{Sum}_D[1, i] - \text{Sum}_D[1, l_i - 1] = P[i] - P[l_i - 1]$ where the array P was introduced in Section 2.4. So we need to find the smallest index $l_i \in [1, L_i)$ such that $P[i] - P[l_i - 1] \geq t$. The array P can be pre-computed in linear time and space.

It is worth to observe that the computation of l_i could be done by scanning $P[1, L_i - 1]$ and searching for the *leftmost* index x such that $P[i] - P[x] \geq t$. We could then set $l_i = x + 1$ and have been done. Unfortunately, this is inefficient because it leads to scan over and over again the same positions of P as i increases, thus leading to a quadratic-time algorithm! Since we aim for a linear-time algorithm, we need to spend constant time "on average" per step i. We used the quotes because there is no *stochastic* argument here to compute the average, we wish only to capture syntactically the idea that, since we want to spend $O(n)$ time in total, our algorithm has to take constant time *amortized* per steps. In order to achieve this performance we first need to show that we can avoid the scanning of the whole prefix $P[1, L_i - 1]$ by identifying a *subset* of *candidate positions* for x. Call $C_{i,j}$ the candidate positions for iteration i, where $j = 0, 1, \ldots$. They are defined as follows: $C_{i,0} = L_i$ (it is a dummy value), and $C_{i,j}$ is defined inductively as the *leftmost minimum* of the sub-array $P[1, C_{i,j-1} - 1]$ (i.e. the sub-array to the left of the current minimum and/or to the left of L_i). We denote by $c(i)$ the number of these candidate positions for the step i, where clearly $c(i) \leq L_i$ (equality holds when $P[1, L_i]$ is decreasing).

For an illustrative example look at Figure 2.2, where $c(i) = 3$ and the candidate positions are connected via leftward arrows.

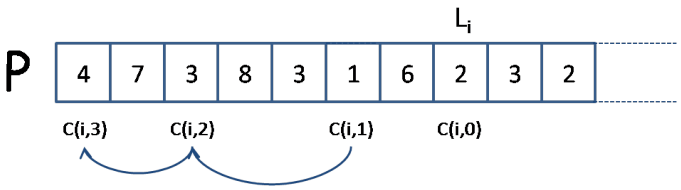

FIGURE 2.2: An illustrative example for the candidate positions $C_{i,j}$, given an array P of prefix sums. The picture is generic and reports only L_i for simplicity.

Looking at Figure 2.2 we derive three key properties whose proof is left to the reader because it immediately comes from the definition of $C_{i,j}$:

Property a. The sequence of candidate positions $C_{i,j}$ occurs within $[1, L_i)$ and moves leftward, namely $C_{i,j} < C_{i,j-1} < \ldots < C_{i,1} < C_{i,0} = L_i$.

Property b. At each iteration i, the sequence of candidate values $P[C_{i,j}]$ is increasing with $j = 1, 2, \ldots, c(i)$. More precisely, we have $P[C_{i,j}] > P[C_{i,j-1}] > \ldots > P[C_{i,1}]$ where the indices move leftward according to Property (a).

Property c. The value $P[C_{i,j}]$ is smaller than any other value on its left in P, because it is the leftmost minimum of the prefix $P[1, C_{i,j-1} - 1]$.

It is crucial now to show that the index we are searching for, namely l_i, can be derived by looking only at these candidate positions. In particular we can prove the following:

FACT 2.2

At each iteration i, the largest index j^* such that $Sum_D[C_{i,j^*} + 1, i] \geq t$ (if any) provides us with the longest segment we are searching for.

By Fact 2.1 we are interested in segments having the form $D[l_i, i]$ with $l_i < L_i$, and by properties of prefix-sums, we know that $\text{Sum}_D[C_{i,j} + 1, i]$ can be re-written as $P[i] - P[C_{i,j}]$. Given this and Property (c), we can conclude that all segments $D[z, i]$, with $z < C_{i,j}$, have a sum *smaller* than $\text{Sum}_D[C_{i,j} + 1, i]$. Consequently, if we find that $\text{Sum}_D[C_{i,j} + 1, i] < t$ for some j, then we can discard all positions z to the left of $C_{i,j} + 1$ in the search for l_i. Therefore the index j^* characterized in Fact 2.2 is the one giving correctly $l_i = C_{i,j^*} + 1$.

There are two main problems in deploying the candidate positions for the efficient computation of l_i: (1) How do we compute the $C_{i,j}$s as i increases, (2) How do we search for the index j^*. To address issue (1) we notice that the computation of $C_{i,j}$ depends only on the position of the previous $C_{i,j-1}$ and *not* on the indices i or j. So we can define an auxiliary array $LMin[1, n]$ such that $LMin[i]$ is the leftmost position of the minimum within $P[1, i - 1]$. It is not difficult to see that $C_{i,1} = LMin[L_i]$, and that according to the definition of C it is $C_{i,2} = LMin[\, LMin[L_i]\,] = LMin^2[L_i]$. In general, it is $C_{i,k} = LMin^k[L_i]$. This allows an incremental computation:

$$LMin[x] = \begin{cases} 0 & \text{if } x = 0 \\ x - 1 & \text{if } P[x-1] < P[\, LMin[x-1]\,] \\ LMin[x-1] & \text{otherwise} \end{cases}$$

The formula above has an easy explanation. We know inductively $LMin[x - 1]$ as the leftmost minimum in the array $P[1, x - 2]$: initially we set $LMin[0]$ to the dummy value 0. To compute $LMin[x]$ we need to determine the leftmost minimum in $P[1, x - 1]$: this is located either in $x - 1$ (with value $P[x - 1]$) or it is the one determined for $P[1, x - 2]$ of position $LMin[x - 1]$ (with value $P[LMin[x - 1]]$). Therefore, by comparing these two values we can compute $LMin[x]$ in constant time. Hence the computation of all candidate positions $LMin[1, n]$ takes $O(n)$ time.

We are left with the problem of determining j^* efficiently. We will not be able to compute j^* in constant time at each iteration i but we will show that, if at step i we execute $s_i > 1$ steps, then we are advancing in the construction of the longest solution. Specifically, we are extending the length of that solution by $\Theta(s_i)$ units. Given that the longest segment cannot be longer than n, the sum of these extra-costs cannot be larger than $O(n)$, and thus we are done! This is called *amortized argument* because we are, in some sense, charging the cost of the expensive iterations to the cheapest ones. The computation of j^* at iteration i requires the check of the positions $LMin^k[L_i]$ for $k = 1, 2, \ldots$ until the condition in Fact 2.2 is satisfied; in fact, we know that all the other $j > j^*$ do not satisfy Fact 2.2. This search takes j^* steps and finds a new segment whose length is *increased* by at least j^* units, given Property (a) above. This means that either an iteration i takes constant time, because the check fails immediately at $LMin[L_i]$ (so the current solution is not better than the one computed at the previous iteration $i - 1$), or the iteration takes $O(j^*)$ time but the new segment $D[L_i, r_i]$ has been extended by j^* units. Since a segment cannot be longer than the entire sequence $D[1, n]$, we can conclude that the total extra-time cannot be larger than $O(n)$.

We leave to the diligent reader to work out the details of the pseudo-code of this algorithm, the techniques underlying its elegant design and analysis should be clear enough to approach it without any difficulties.

References

[1] Kun-Mao Chao. Maximum-density segment. In Ming-Yang Kao, editor, *Encyclopedia of Algorithms*. Springer, 2008.

[2] Kun-Mao Chao. Maximum-scoring segment with length restrictions. In Ming-Yang Kao, editor, *Encyclopedia of Algorithms*. Springer, 2008.

[3] Chih-Huai Cheng, Hsiao-Fei Liu, and Kun-Mao Chao. Optimal algorithms for the average-constrained maximum-sum segment problem. *Information Processing Letters*, 109(3):171–174, 2009.

[4] Rolf Fagerberg. Cache-oblivious model. In Ming-Yang Kao, editor, *Encyclopedia of Algorithms*. Springer, 2008.

[5] Takeshi Fukuda, Yasuhiko Morimoto, Shinichi Morishita, and Takeshi Tokuyama. Mining optimized association rules for numeric attributes. *Journal of Computer System Sciences*, 58(1):1–12, 1999.