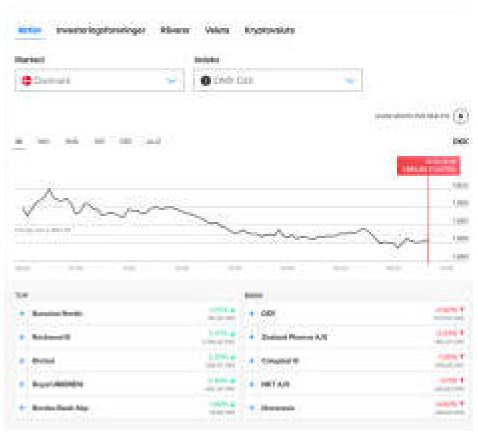
Maximum Sum Problem

Given an array (or list) of numbers, the goal is to find the segment (sub-array) with the largest sum.

For example, in the segment: 6 0 -1 1 2 2 -4 3 5 4 3 5 -1 6 2 7 -6 8 0 9 8 10 12 11 -4 12 6 13 8 14 4 15 The sum of the segment $(-4)+5+3+(-1)+2+(-6)+0+8=7$. The question is: which segment has the largest sum?

Motivation: Stock Analysis

The Maximum Sum problem has applications in stock analysis.



The image displays a fluctuating line graph, typical of stock market data, with a table below it. This kind of data visualization helps in understanding stock performance over time.

Given stock data like:

- +2% 2023.02.20
- −3% 2023.02.21
- +8% 2023.02.22
- −1% 2023.02.23
- −3% 2023.02.24
- +3% 2023.02.25
- +11% 2023.02.26

The question becomes: during which period would it have been best to own the stock?

Percent Calculations

If 1000 kr increases by 3, it becomes $1000 \times 1.03 = 1030$ kr. If 1000 kr decreases by 2, it becomes $1000 \times 0.98 = 980$ kr. If 1000 kr first increases by 3 and then decreases by 2, it becomes $1000 \times 1.03 \times 0.98 (= 1009.40)$ kr.

Using the stock data from before:

- +2% 2023.02.20
- −3% 2023.02.21
- +8% 2023.02.22
- −1% 2023.02.23
- −3% 2023.02.24
- +3% 2023.02.25
- +11% 2023.02.26

The stock changes by a factor of $0.97 \times 1.08 \times 0.99 \times 0.97 \times 1.03$ in the period. The question is: in which period would it have been best to own the stock?

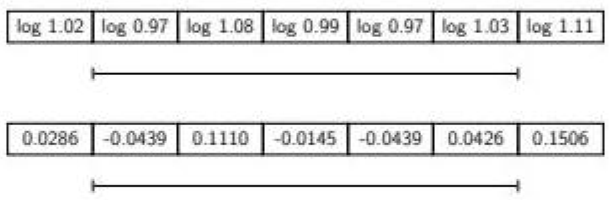
1. 02 0.97 1.08 0.99 0.97 1.03 1.11

The question "Which period was the best to own the stock?" translates to "Which segment has the largest product?"

Maximum Product to Maximum Sum

Logarithms are increasing functions. So, $0.94 \times 1.05 \times 0.99 \leq 0.96 \times 1.03 \times 1.01$ if and only if $log(0.94 \times 1.05 \times 0.99) \leq log(0.96 \times 1.03 \times 1.01)$. Since $log(x \times y) = log(x) + log(y)$, the above holds if and only if $log(0.94) + log(1.05) + log(0.99) \leq log(0.96) + log(1.03) + log(1.01)$.

Therefore, the segment with the largest product in this array: 1.02 0.97 1.08 0.99 0.97 1.03 is the same as the segment with the largest sum in this array: log 1.02 log 0.97 log 1.08 log 0.99 log 0.97 log 1.03 0. 0286 -0.0439 0.1110 -0.0145 -0.0439 0.0426 | 1.11 |



log 1.02	log 0.97	log 1.08	log 0.99	log 0.97	log 1.03	log 1.11
0.0286	-0.0439	0.1110	-0.0145	-0.0439	0.0426	0.1506

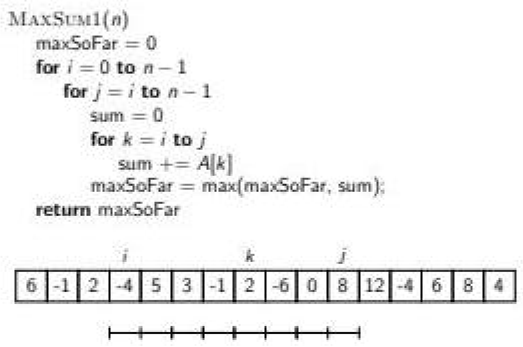
This image displays a table that relates logarithmic expressions to their corresponding numerical values. For example, it shows how "log 1.02" corresponds to "0.0286". This is useful for converting a maximum product problem into a maximum sum problem using logarithms.

Algorithms for MaxSum

The task is to find the sum for all segments: 6 -1 2 -4 5 3 -1 2 -6 0 8 12 -4 6 8 4 i j A natural algorithm follows from the definition: for all i and for all $j \geq i$, compute the sum from i to j.

6 -1 2 -4 5 3 -1 2 -6 0 8 12 -4 6 8 4 i k j

First Algorithm for MaxSum



```
MaxSum1(n)
  maxSoFar = 0
  for i = 0 to n − 1
      for j = i to n − 1
          sum = 0
          for k = i to j
              sum += A[k]
          maxSoFar = max(maxSoFar, sum);
  return maxSoFar
```

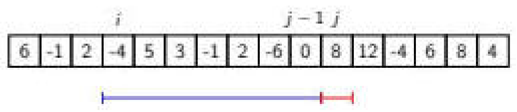
The image displays the pseudocode for `MaxSum1(n)`, an algorithm designed to find the maximum contiguous subarray sum. It uses three nested loops to iterate through all possible subarrays and calculate their sums.

This algorithm is correct because it directly follows the definition of the problem. However, its runtime is $\Theta(n^3)$, similar to Algorithm 3 from the asymptotic analysis examples, due to its triply nested loop structure.

$$(-4) + 5 + 3 + (-1) + 2 + (-6) + 0 = (-1)$$

$$\Downarrow$$

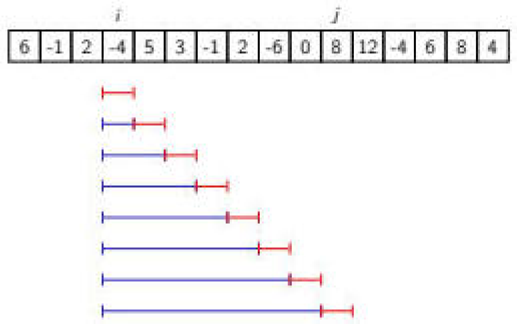
$$(-4) + 5 + 3 + (-1) + 2 + (-6) + 0 + 8 = (-1) + 8 = 7$$



The image shows a mathematical equation alongside a number line, illustrating the addition of integers. For example, $(-4) + 5 + 3 + (-1) + 2 + (-6) + 0 = (-1)$, which transitions to $(-4) + 5 + 3 + (-1) + 2 + (-6) + 0 + 8 = (-1) + 8 = 7$. This visually represents how cumulative sums are calculated over intervals, relevant for understanding the MaxSum problem.

Idea for Improved Algorithm

Algorithm: For each i, calculate sums for increasing j with one new addition per sum.



The image illustrates a series of line segments, likely representing intervals or vectors, with values "i" and "j" indicating endpoints. This visualization is helpful for understanding the concept of iterating through segments of an array to find the maximum sum.

Second Algorithm for MaxSum

```
MaxSum2(n)
  maxSoFar = 0
  for i = 0 to n - 1
    sum = 0
    for j = i to n - 1
      sum += A[j]
      maxSoFar = max(maxSoFar, sum);
  return maxSoFar
```

This algorithm is correct and follows the definition of the problem and the observation above. Its runtime is $\Theta(n^2)$, similar to Algorithm 2 from the asymptotic analysis examples.

New observation: $x1 \leq x2 \iff x1 + 2 \leq x2 + 2$

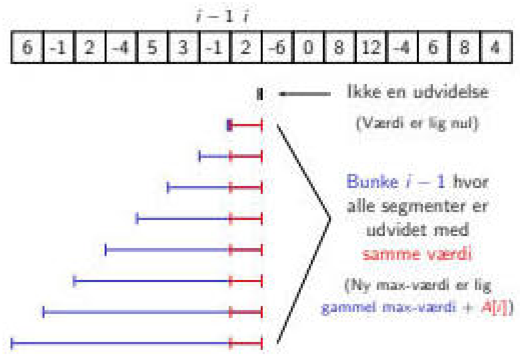
Implications:

$$max x1 + 2, x2 + 2, \ldots, xi + 2 = max x1, x2, \ldots, xi + 2$$

The idea is whether we can look at segments in bunches, such that the new bunch is equal to the old bunch with all segments expanded by the same value.

Idea for Improved Algorithm

Let bunch i be all segments that end at $A[i]$'s right edge. Then bunch i is the same as bunch $i - 1$ with all segments expanded by the same value, plus the empty segment:


$i-1$ i
6 -1 2 -4 5 3 -1 2 -6 0 8 12 -4 6 8 4
Ikke en udvidelse
(Værdi er lig nul)
Bunke $i-1$ hvor alle segmenter er udvidet med samme værdi
(Ny max-værdi er lig gammel max-værdi + $A[i]$)

This image visually represents segments and their values, likely used to explain how segments can be grouped or "bunched" together to optimize the maximum sum calculation.

Third Algorithm for MaxSum

```
MaxSum3(n)
  maxSoFar = 0
  maxEndingHere = 0
  for i = 0 to n - 1
    maxEndingHere = max(maxEndingHere + A[i], 0)
    maxSoFar = max(maxSoFar, maxEndingHere);
  return maxSoFar
```

This algorithm is correct, following the definition of the problem and the new observation above. This ensures that we take the maximum over all segments (since every segment is in a bunch, namely the bunch for the i where the segment ends). The runtime is $\Theta(n)$ since there are n iterations, each taking $\Theta(1)$ time.