CMSC 351: Maximum Contiguous Sum

Justin Wyss-Gallifent

September 8, 2022

1 Introduction

Given a list of integers find the contiguous sublist with maximum sum and return that maximum sum.

Example 1.1. For example in the list given here:

```
[-9, 3, 1, 1, 4, -2, -8]
```

There are many contiguous sublists each with a sum, for example:

- `[-9,3,1]` has sum -5
- `[1,1,4,-2]` has sum 4
- ...and so on...

Out of all of these the maximum sum is 9 arising from the sublist `[3,1,1,4]`.

You'll see this algorithm with certain restrictions, for example insisting that the maximum sum must be positive, but we'll ignore any restrictions.

1.1 Why it's Interesting

Beyond any real-world applications solving this problem is a great approach to many of the various approaches we'll be seeing through the course, and so doing it early is like putting out a little taste of a few things.
One real-world application would be if a list A contains the daily increase and decrease information for a stock and we wished to find the set of days over which the stock experienced the largest net growth. In the introductory example if each entry contains the profit (or loss) correspnding to a day of the week then we can say that the period of maximum profit was four days long and yielded a profit of 9.

2 Extremely Stupid Brute Force

2.1 Introduction

The brute force approach would be to check all possible sums. We test all possible start indices and all possible end indices and we calculate each sum.

2.2 Pseudocode with Time Complexity Notes

Here is our pseudocode with choice of time assignments. To explore a bit more detail rather than just time complexity we have preserved some time assignments that we could otherwise dismiss.

```
\\ PRE: A is a list of length n.
max = A[0]                          $c_1$
for i = 0 to n-1                    $n$ times
    for j = i to n-1                $n-1-i+1$ times
        sum = 0                     $c_1$
        for k = i to j              $j-i+1$ times
            sum = sum + A[k]        $c_1$
        end
        if sum > max:             ⎫
            max = sum             ⎬ $c_2$
        end                       ⎭
    end
end
\\ POST: max is the maximum sum.
```

The worst-case total time requirement and time complexity can then be calculated as a nested sum:

$$T(n) = c_1 + \sum_{i=0}^{n-1} \sum_{j=i}^{n-1} \left[c_1 + \left[\sum_{k=i}^{j} c_1 \right] + c_2 \right] = ... = \Theta(n^3)$$

This approach is obviously sloppy and the time complexity of $\Theta(n^3)$ is atrocious. Note that for the best-case even if the single conditional is never true the body of the inner `for` loop is still taking nonzero time and nothing is any better. Thus the best-case, and then the average case, are also $\Theta(n^3)$.

3 Brute Force Method

3.1 Introduction

The brute force method above is pretty brutish and we can be slightly kinder. We start with each index and compute sums from that index onwards. If any given sum is larger than the maximum we update the maximum.
It's basically the same as extreme brute force except we're using the second loop to take care of the sum instead of introducting a third loop.

3.2 Pseudocode with Time Complexity Notes

Here is our pseudocode with choice of time assignments:

```
\\ PRE: A is a list of length n.
max = A[0]                        $c_1$
for i = 0 to n-1                  $n$ times
    sum = 0                       $c_1$
    for j = i to n-1              $n - 1 - i + 1$ times
        sum = sum + A[j]          $c_1$
        if sum > max              ⎫
            max = sum             ⎬ $c_2$
        end                       ⎭
    end
end
\\ POST: max is the maximum sum.
```

The total worst-case time complexity can then be calculated as a nested sum:

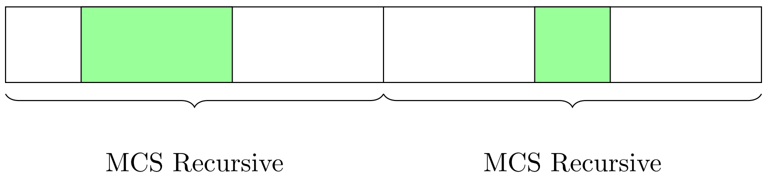
$$\begin{aligned}
T(n) &= c_1 + \sum_{i=0}^{n-1} \left[c_1 + \sum_{j=i}^{n-1} (c_1 + c_2) \right] \\
&= c_1 + \sum_{i=0}^{n-1} \left[c_1 + (c_1 + c_2) \sum_{j=i}^{n-1} 1 \right] \\
&= c_1 + \sum_{i=0}^{n-1} [c_1 + (c_1 + c_2)(n - 1 - i + 1)] \\
&= c_1 + \sum_{i=0}^{n-1} [c_1 + (c_1 + c_2)n] - \sum_{i=0}^{n-1} (c_1 + c_2)i \\
&= c_1 + [c_1 + (c_1 + c_2)n] \sum_{i=0}^{n-1} 1 - (c_1 + c_2) \sum_{i=0}^{n-1} i \\
&= c_1 + [c_1 + (c_1 + c_2)n]\,(n) - (c_1 + c_2) \left(\frac{(n-1)(n)}{2} \right) \\
&= c_1 + c_1 n + (c_1 + c_2)n^2 - \frac{1}{2}(c_1 + c_2)n^2 + \frac{1}{2}(c_1 + c_2)n \\
&= \frac{1}{2}(c_1 + c_2)n^2 + \frac{1}{2}(3c_1 + c_2)n + c_1 \\
&= \Theta(n^2)
\end{aligned}$$

This is perhaps the default go-to method because it's very easy to understand. However $\Theta(n^2)$ time is less than ideal. Of course for small n it's fine, and therefore arguably useful, however.
Using a similar argument to extreme brute force we have best- and average-case time complexity also $\Theta(n^2)$.

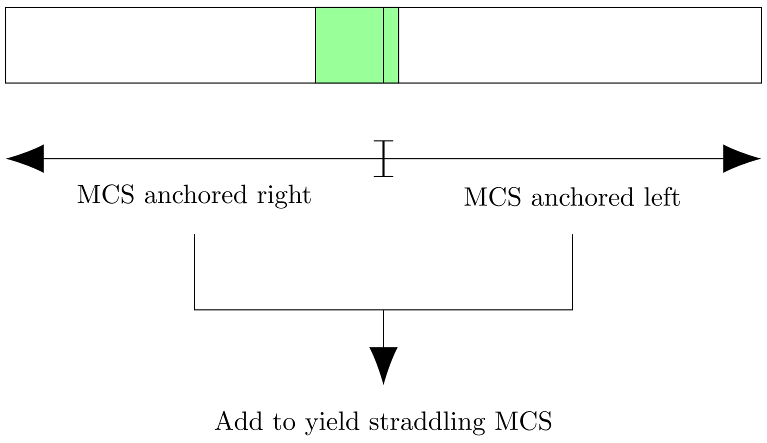
4 Divide-and-Conquer

4.1 Introduction

Interestingly we can apply a divide-and-conquer approach using a recursive algorithm. Basically we split the list in half and then do three things: We find the MCS on the left side (via a recursive call), on the right side (via a recursive call), and the one that straddles the dividing line (not a recursive call). The bottom of the recursion occurs when the list is length 1.
The MCS straddling the middle is calculated by taking the MCS which extends from the middle to the left and the MCS which extends from the middle to the right and adding those.
Graphically the two recursive MCS calls look like:



And the straddling MCS looks like:



We then take the maximum of these three to get the overall maximum.

4.2 Pseudocode with Time Complexity Notes

For the time complexity we'll drop all the constant times that we safely can. We keep the c_1 for the `return` because it's the only thing in the conditional body and the c_2 in the iteration bodies. The c_3 covers all the `else` time other than the body of the `for` loops. We don't worry about the body of the two other `if` statements because they are constant time and this is accounted for with the c_2.

```
\\ PRE: A is a list of length n.
function mcs(A,L,R)
    if L == R
        return(A[L])                    $c_1$
    else
        C = (L+R) // 2                  $c_3$
        Lmax = mcs(A,L,C)
        Rmax = mcs(A,C+1,R)
        \\ Calculate the straddle max
        Lhmax = A[C]
        Lhsum = 0
        for i = C downto L              $C - L + 1$ times
            Lhsum = Lhsum + A[i]        }
            if Lhsum > Lhmax            }
                Lhmax = Lhsum           } $c_2$
            end                         }
        end
        Rhmax = A[C+1]
        Rhsum = 0
        for i = C+1 to R                $(C - L + 1) + (R - (C + 1) + 1) = R - C + 1$ times
            Rhsum = Rhsum + A[i]        }
            if Rhsum > Rhmax:           }
                Rhmax = Rhsum           } $c_2$
            end                         }
        end
        Smax = Lhmax + Rhmax
        \\ Return the overall max
        return(max([Lmax,Rmax,Smax]))
    end
end
\\ POST: max is the maximum sum.
```

In what follows we're glossing over various floor and ceiling functions which occur during the division process. This glossing over is fairly common in algorithm analysis and in general has no impact on the result in any meaningful complexity way.

Let's say that the original call is recursion depth 0. At that depth there is $2^0 = 1$ list of length n. We have a c_3 assignment and then two for loops which together iterate n times for a total of $c_2 n$ time.

At recursion depth 1 there are $2^1 = 2$ lists of length $n/2^1$. For each we have a c_3 assignment and two for loops which together iterate $n/2^1$ times for a total of $c_2(n/2)$ times. All together then the time is $2(c_3 + c_2(n/2^1) = 2c_3 + c_2 n$.

At recursion depth 2 there are $2^2 = 4$ lists of length $n/2^2$. For each we have a c_3 assignment and two for loops which together iterate $n/2^2$ times for a total of $c_2(n/2^2)$ times. All together then the time is $4(c_3 + c_2(n/2^2) = 4c_3 + c_2 n$.

This continues until the recursion length of the lists are 1. If k is the depth then this occurs when $n/2^k = 1$ or $k = \lg 2$. In such a case there are n lists of length 1 and the algorithm takes c_1 for each for a total of $c_1 n$.

Thus in total the time is:

$$\begin{aligned} c_1 n + \sum_{i=0}^{\lg n - 1} \left(2^i c_3 + c_2 n\right) &= c_1 n + c_3 \sum_{i=0}^{\lg n-1} 2^i + c_2 n \sum_{i=0}^{\lg n - 1} \\ &= c_1 n + c_3 \left(2^{\lg n} - 1\right) + c_2 n \lg n \\ &= c_1 n + c_3(n-1) + c_2 n \lg n \\ &= \Theta(n \lg n) \end{aligned}$$

Alternately this may be approached using the Master Theorem (we'll see later). If each call requires time $T(n)$ satisfying:

$$T(n) = 2T(n/2) + \Theta(n)$$

where the $\mathcal{O}(n)$ arises when calculating the maximum contiguous sublist crosses the middle. The Master Theorem then tell us that $T(n) = \mathcal{O}(n \log n)$.

4.3 Straddling Sum Example

To clarify the straddling sum consider the list $[4, -2, 5, 1, 2, 3, -1]$ with $l = 0$ and $r = 6$. We have $c = (6 + 0)/2 = 3$ and so we calculate the left-hand max anchored at index $c = 3$:

4	-2	5	1	2	3	-1	
			1				Lhsum = 1
		5	1				Lhsum = 6
	-2	5	1				Lhsum = 4
4	-2	5	1				Lhsum = 8

Thus we have Lhmax = 8.
And the right-hand max anchored at index $c + 1 = 5$:

4	-2	5	1	2	3	-1	
				2			Rhsum = 2
				2	3		Rhsum = 5
				2	3	-1	Rhsum = 4

Thus we have Lhmax = 5.
In total then smax = $8 + 5 = 13$, the total max straddling the center.

Note 4.3.1. The is the first case we've seen where it's evident why divide-and-conquer confers an advantage with regards to time complexity.
It's extremely common that divide-and-conquer leads to the introduction of a logarithmic factor in the time complexity and that this logarithmic factor replaces something larger.

5 Kadane's Algorithm

5.1 Introduction

Kadane's Algorithm is a sneaky way of solving the problem in $\mathcal{O}(n)$. The basic premise is based on the idea of dynamic programming.
In dynamic programming the idea is to use previous knowledge as much as possible when iterating through a process.

5.2 Mathematics

Theorem 5.2.1. Define M_i as the maximum contiguous sum ending at and including index i for $0 \leq i \leq n-1$. Then we have $M_0 = A[0]$ and $M_i = \max(M_{i-1} + A[i], A[i])$.

Proof. It's clear that $M_0 = A[0]$ since $A[0]$ is the only contiguous sum ending at index 0.
Consider M_i for some $1 \leq i \leq n-1$.
Denote by C_i the set of contiguous sums ending at index i and denote by C_{i-1} the set of contiguous sums ending at index $i-1$.
Then we have:

$$\begin{aligned} C_i &= \{x + A[i] \,|\, x \in C_{i-1}\} \cup \{A[i]\} \\ M_i = \max(C_i) &= \max\left(\{x + A[i] \,|\, x \in C_{i-1}\} \cup \{A[i]\}\right) \\ &= \max\left(\{x + A[i] \,|\, x \in C_{i-1}\}, A[i]\right) \\ &= \max\left(M_{i-1} + A[i], A[i]\right) \end{aligned}$$

$\mathcal{QED}$

What this means is that we can progress through the list, calculating the maximum contiguous sum ending at and including index $i = 1, .., n-1$ using the previous maximum contiguous sum ending at and including index $i-1$.
More programatically: We start by setting `mcs=A[0]`. Well then go through the list from `i=0,...,n-1`. At each step we take the maximum of `mcs` and `mcs+A[i]`. We compare this with an ongoing overall maximum and keep track of the largest of these.

5.3 Pseudocode with Time Complexity Notes

Here is the pseudocode:

```
\\ PRE: A is a list of length n.
maxoverall = A[0]
maxendingati = A[0]
for i = 1 to n-1
    maxendingati = max(maxendingati+A[i],A[i])
    maxoverall = max(maxoverall,maxendingati)
end
\\ POST: maxoverall is the maximum contiguous sum.
```

Note that the use of the `max` commands can be replaced by conditionals. We're just being compact here.
The single iteration of the loop makes it clear that the time complexity is $\Theta(n)$.

Note 5.3.1. Worth noting is that it's not uncommon to have algorithms in which there is a $\mathcal{O}(n)$ solution which requires some rather ingenuous consideration of the solution.

5.4 Example Walk-Through

Consider the list:

$$[2, 3, -4, 5, 1, 2, -8, 3]$$

I'll use M to indicate the maximum overall contiguous sum and Mi to indicate the maximum contiguous sum ending at and including index i.
We initiate $M = 2$ and $Mi = 2$.
Then we iterate:

$\Rightarrow$ When $i = 1$ we get $Mi = \max(2 + 3, 3) = 5$ and $M = \max(2, 5) = 5$.
$\Rightarrow$ When $i = 2$ we get $Mi = \max(5 - 4, -4) = 1$ and $M = \max(5, 1) = 5$.
$\Rightarrow$ When $i = 3$ we get $Mi = \max(1 + 5, 5) = 6$ and $M = \max(5, 6) = 6$.
$\Rightarrow$ When $i = 4$ we get $Mi = \max(6 + 1, 1) = 7$ and $M = \max(6, 7) = 7$.
$\Rightarrow$ When $i = 5$ we get $Mi = \max(7 + 2, 2) = 9$ and $M = \max(7, 9) = 9$.
$\Rightarrow$ When $i = 6$ we get $Mi = \max(9 - 8, -8) = 1$ and $M = \max(9, 1) = 9$.
$\Rightarrow$ When $i = 7$ we get $Mi = \max(1 + 3, 3) = 4$ and $M = \max(9, 4) = 9$.

Thus the maximum contiguous sum is 9.

6 Comment

It's worth noting that even though Kadane's Algorithm is fastest it is arguably not the most clear.

7 Thoughts, Problems, Ideas

1. Fill in the details of the extreme brute force time complexity calculation for the case where all the constants are 1:

$$T(n) = 1 + \sum_{i=0}^{n-1} \sum_{j=i}^{n-1} \left[1 + \left[\sum_{k=i}^{j} 1 \right] + 1 \right] = (\text{fill in!}) = \Theta(n^3)$$

2. Fill in the details of the naïve method time complexity calculation for the case where all the constants are 1:

$$T(n) = 1 + \sum_{i=0}^{n-1} \left[1 + \sum_{j=i}^{n-1} (1+1) \right] = (\text{fill in!}) = \Theta(n^2)$$

3. Modify the pseudocode for each of the four variations so that an additional variable `maxlen` is passed and so that the sublist whose sum is returned may not have a length longer than this.

4. In the divide-and-conquer algorithm we divide the list in half in the line `c=(l+r)//2`. What effect does it have if this line is replaced by, for example, `c=(l+r)//3`? Explain

5. Modify the pseudocode for Kadane's Algorithm so that it finds the maximum nonnegative sum. If every element in the list is negative return `false` instead of the sum.

6. Modify the pseudocode for Kadane's Algorithm so that it returns the length of the sublist which provides the maximum contiguous sum.

7. What is a useful loop invariant for Kadane's Algorithm? Prove the requirement of the Loop Invariant Theorem for this loop invariant.

8. Modify the pseudocode for the naïve method so that it finds the maximum sum of a rectangular subarray within an $n \times m$ array. Assuming that all constant time complexities take time 1 calculate the $\mathcal{O}$ time complexity of your pseudocode.

9. Using the same approach as divide-and-conquer write an algorithm which returns the length of the longest string of `1`s in list of `0`s and `1`s.

10. Write a clear explanation of how the divide-and-conquer approach could be modified to find the maximum sum of a rectangular subarray within an $n \times m$ array. Pseudocode isn't necessary.

11. What are your thoughts on modifying Kadane's Algorithm to find to find the maximum sum of a rectangular subarray within an $n \times m$ array?

12. In Kadane's Algorithm consider the line:

```
maxoverall=max(maxoverall,maxendingati)
```

In a worst-case scenario what is the maximum number of times this will change the value of `maxoverall`? Explain.

13. Suppose your list A has the property that the maximum contiguous sum should only examine sublists of length 5 or less. Modify each of the algorithms in order to take this into account.

14. Modify each algorithm so that instead of finding the maximum contiguous sum a particular value is passed and the algorithm returns True if there is a contiguous sum equal to that value and False if not.

15. Modify the previous problems algorithm so that it returns the number of continguous sums which equal that value.

8 Python Code with Output

8.1 Brute Force

Code:

```
import random
A = []
for i in range(0,7):
    A.append(random.randint(-10,10))
n = len(A)

print(A)

max = A[0]
for i in range(0,n):
    for j in range(i,n):
        sum = 0
        for c in range(i,j+1):
            sum = sum + A[c]
        if sum > max:
            max = sum
        print('Max of ' + str(A[i:j+1]) + ': ' + str(max))

print('Overall max: ' + str(max))
```

Ouput:

```
[-9, 3, 1, 1, 4, -2, -8]
Max of [-9]: -9
Max of [-9, 3]: -6
Max of [-9, 3, 1]: -5
Max of [-9, 3, 1, 1]: -4
Max of [-9, 3, 1, 1, 4]: 0
Max of [-9, 3, 1, 1, 4, -2]: 0
Max of [-9, 3, 1, 1, 4, -2, -8]: 0
Max of [3]: 3
Max of [3, 1]: 4
Max of [3, 1, 1]: 5
Max of [3, 1, 1, 4]: 9
Max of [3, 1, 1, 4, -2]: 9
Max of [3, 1, 1, 4, -2, -8]: 9
Max of [1]: 9
Max of [1, 1]: 9
Max of [1, 1, 4]: 9
Max of [1, 1, 4, -2]: 9
Max of [1, 1, 4, -2, -8]: 9
Max of [1]: 9
Max of [1, 4]: 9
Max of [1, 4, -2]: 9
Max of [1, 4, -2, -8]: 9
Max of [4]: 9
Max of [4, -2]: 9
Max of [4, -2, -8]: 9
Max of [-2]: 9
Max of [-2, -8]: 9
Max of [-8]: 9
Overall max: 9
```

8.2 Naïve Method

Code:

```
import random
A = []
for i in range(0,7):
    A.append(random.randint(-10,10))
n = len(A)
print(A)

max = A[0]
for i in range(0,n):
    sum = 0
    for j in range(i,n):
        sum = sum + A[j]
        if sum > max:
            max = sum
        print('Max of ' + str(A[i:j+1]) + ': ' + str(max))

print('Overall max: ' + str(max))
```

Output:

```
[-2, -1, 6, -1, 6, 3, 2]
Max of [-2]: -2
Max of [-2, -1]: -2
Max of [-2, -1, 6]: 3
Max of [-2, -1, 6, -1]: 3
Max of [-2, -1, 6, -1, 6]: 8
Max of [-2, -1, 6, -1, 6, 3]: 11
Max of [-2, -1, 6, -1, 6, 3, 2]: 13
Max of [-1]: 13
Max of [-1, 6]: 13
Max of [-1, 6, -1]: 13
Max of [-1, 6, -1, 6]: 13
Max of [-1, 6, -1, 6, 3]: 13
Max of [-1, 6, -1, 6, 3, 2]: 15
Max of [6]: 15
Max of [6, -1]: 15
Max of [6, -1, 6]: 15
Max of [6, -1, 6, 3]: 15
Max of [6, -1, 6, 3, 2]: 16
Max of [-1]: 16
Max of [-1, 6]: 16
Max of [-1, 6, 3]: 16
Max of [-1, 6, 3, 2]: 16
Max of [6]: 16
Max of [6, 3]: 16
Max of [6, 3, 2]: 16
Max of [3]: 16
Max of [3, 2]: 16
Max of [2]: 16
Overall max: 16
```

8.3 Divide-and-Conquer

Code:

```
import random

A = []
for i in range(0,10):
    A.append(random.randint(-10,10))
n = len(A)
print(A)

def mcs(A,l,r,indentlevel):
    print(indentlevel*'.' + 'Finding mcs in A='+str(A[l:r+1]))
    if l == r:
        print(indentlevel*'.' + 'It is '+str(A[l]))
        return(A[l])
    else:
        c = (l+r) // 2
        lhmax = A[c]
        lhsum = 0
        for i in range(c,l-1,-1):
            lhsum = lhsum + A[i]
            if lhsum > lhmax:
                lhmax = lhsum
        rhmax = A[c+1]
        rhsum = 0
        for i in range(c+1,r+1):
            rhsum = rhsum + A[i]
            if rhsum > rhmax:
                rhmax = rhsum
        cmax = lhmax + rhmax
        print(indentlevel*'.' + 'Straddle max is '+str(cmax))
        lmax = mcs(A,l,c,indentlevel+2)
        rmax = mcs(A,c+1,r,indentlevel+2)
        omax = max([lmax,rmax,cmax])
        print(indentlevel*'.' + 'It is '+str(omax))
        return(omax)

print(mcs(A,0,len(A)-1,0))
```

Output:

```
[-1, 2, 9, -3, 5, -8, 10, -6, 1, 0]
Finding mcs in A=[-1, 2, 9, -3, 5, -8, 10, -6, 1, 0]
..Finding mcs in A=[-1, 2, 9, -3, 5]
....Finding mcs in A=[-1, 2, 9]
......Finding mcs in A=[-1, 2]
........Finding mcs in A=[-1]
........It is -1
........Finding mcs in A=[2]
........It is 2
......It is 2
......Finding mcs in A=[9]
......It is 9
....It is 11
....Finding mcs in A=[-3, 5]
......Finding mcs in A=[-3]
......It is -3
......Finding mcs in A=[5]
......It is 5
....It is 5
..It is 13
..Finding mcs in A=[-8, 10, -6, 1, 0]
....Finding mcs in A=[-8, 10, -6]
......Finding mcs in A=[-8, 10]
........Finding mcs in A=[-8]
........It is -8
........Finding mcs in A=[10]
........It is 10
......It is 10
......Finding mcs in A=[-6]
......It is -6
....It is 10
....Finding mcs in A=[1, 0]
......Finding mcs in A=[1]
......It is 1
......Finding mcs in A=[0]
......It is 0
....It is 1
..It is 10
It is 15
15
```

8.4 Kadane's Algorithm

Code:

```
import random

A = []
for i in range(0,20):
    A.append(random.randint(-10,10))

n = len(A)
print(A)

maxoverall = A[0]
maxendingati = A[0]
for i in range(1,n):
    maxendingati = max(maxendingati+A[i],A[i])
    maxoverall = max(maxoverall,maxendingati)
    print('Best sum ending with index i='+str(i)+' is '+str(maxoverall))

print(maxoverall)
```

Output:

```
[4, -1, 0, 4, -7, 2, -3, -2, 3, 1, 7, 3, -5, 2, -3, 2, 9, -5, 3, -2]
Best sum ending with index i=1 is 4
Best sum ending with index i=2 is 4
Best sum ending with index i=3 is 7
Best sum ending with index i=4 is 7
Best sum ending with index i=5 is 7
Best sum ending with index i=6 is 7
Best sum ending with index i=7 is 7
Best sum ending with index i=8 is 7
Best sum ending with index i=9 is 7
Best sum ending with index i=10 is 11
Best sum ending with index i=11 is 14
Best sum ending with index i=12 is 14
Best sum ending with index i=13 is 14
Best sum ending with index i=14 is 14
Best sum ending with index i=15 is 14
Best sum ending with index i=16 is 19
Best sum ending with index i=17 is 19
Best sum ending with index i=18 is 19
Best sum ending with index i=19 is 19
19
```