Bu-Ali Sina University

Report

Maximum Subarray Sum

Mehran Shahidi

supervised by
Dr. Mansori

May 10, 2020

Summary

In this report, I will explain how to solve the Maximum Sub-Array Sum with three different approaches, and in the end, you will see the comparison of those approaches. This report is written in LaTeX. For accessing to the LaTeX files and the source codes you can visit my github repo for the algorithm course.

https://github.com/m3hransh/algorithmt

I also have an algorithm course on youtube feel free to check that out too.

https://www.youtube.com/watch?v=n5rv7pbJpmM&t=256s

Contents

1 introduction

1.1 Defining problem

Let's define our problem(Maximum Sub-Array Sum). You are given a one-dimensional array that may contain both positive and negative integers , find the sum of a contiguous subarray of numbers which has the largest sum. For example, if the given array is {-2, -5, **6, -2, -3, 1, 5**, -6}, then the maximum subarray sum is 7 (see highlighted elements).

2 Brute Force

Our first solution is to check all the possible subarrays. Every subarray is consists of a start and an end. So there are $\binom{n}{2} + n$ possible sub-arrays that is needed to consider (n is for subarray of length 1). This can be implemented with two for-loops. The first for-loop chooses different possible values for the beginning of the sub-array and the second one values for the end of that sub-array. The implementation is in **Listing 1**.

```
def maximum_subarray_1(A):
    '''Return a tuple(range i,j,sum).

    argument:
    A -- list of numbers.
    '''
    max_sum =float('-inf')
    index = (-1,-1)
    for i in range(len(A)):
        temp =0
        for j in range(i, len(A)):
            temp += A[j]
            if temp > max_sum:
                max_sum = temp
                index =(i, j)

    return (index, max_sum)
```

Listing 1: **Brute force implemented of maximum subarray sum**

2.1 Cost Analysis

As you can see, for-loops go through all the possible subarrays, and for each, only it takes $\Theta(1)$ to check if it is the maximum or not. So in total takes $\Theta(n^2)$ to find the maximum subarray.

3 Divide & Conquer

In this approach, first, the input array is divided into two halves. Let's call them the left subarray and the right subarray, and then we solve those smaller instance problems. All possible subarrays are in the left subarray, right subarray, or there is another case that the subarray crosses the midpoint, in this situation we need to solve the problem directly with another help function. You can see different scenarios in **Figure 1**.

(a) (b)

Figure 1: (a) possible locations of subarrays of A[low..high]: entirely in A[low..mid], entirely in A[mid+1..high], (b) or crossing mid.

First, a function is written to solve the situation that the sub-array crosses the middle. The logic is simple, first, start from the middle and go to the beginning to find the index **i** that has the largest sum and then do the same thing for the right sub-array, start from the middle to the end of the array and store the index **j** that has the maximum value. Then the index and the maximum sum will be returned. The code is shown in the Listing 2

```
def find_max_crossing_subarray(A, low, mid, high):
    left_sum = float('-inf')
    temp = 0
    i = mid
    for k in range(mid, low-1, -1):
        temp += A[k]
        if temp > left_sum:
            left_sum = temp
            i = k
    right_sum = float('-inf')
    temp = 0
    j = mid + 1
    for k in range(mid+1, high+1):
        temp += A[k]
        if temp > right_sum:
            right_sum = temp
            j = k

    return ((i,j), left_sum + right_sum)
```

Listing 2: **Find the subarray that has the largest value and crosses the middle**

The divide-and-conquer form is shown in Listing 3. First, the array is divided into two sections. Then the subproblems are solved recursively, and their values compare to each other to return the one that has the largest value.

```
def maximum_subarray_rec(A, low, high):
    if low == high:
        return ((low,low), A[low])
    mid = (low + high) // 2
    left = maximum_subarray_rec(A, low, mid)
    right = maximum_subarray_rec(A, mid+1, high)
    cross = find_max_crossing_subarray(A, low, mid, high)

    return max( left, right, cross, key=lambda x: x[1])
```

Listing 3: **Maximum subarray code using divide and conquer**

3.1 Cost Analysis

As you can see in the base case, it takes $\Theta(1)$ to return the answer. In the recursive steps, the problem is divided into two subproblems with half the size of the original one, so if the run time be $T(n)$, then each of them takes $T(\frac{n}{2})$. For finding maximum subarray that crosses the middle takes $\Theta(n)$ cause we need to

traverse the whole array once and check if the sum is larger than our temp_max or not (takes $\Theta(1)$).

$$T(n) = \begin{cases} \Theta(1) & n = 1 \\ \Theta(n) + T(\frac{n}{2}) & n \geq 1 \end{cases}$$

Using **Master Theorem**, it gives $T(n) = \Theta(n \log(n))$.

4 Greedy

To solve maximum subarray sum with the greedy approach, we need to define another structure. Let's define S_i to be the largest subarray sum that ends at i_{th} element of the array. If the S_{i-1} is known, we can find S_i using following theorem.

Theorem
If S_{i-1} is negetive then S_i is only include $A[i]$ element otherwise $S_i = S_{i-1} + A[i]$.

Proof Let S_{i-1} be the largest subarray that ends at the $(i-1)_{th}$ element. We need to prove in two cases that we can build the S_i. The first case is when $S_{i-1} > 0$. Let $S_i = S_{i-1} + A[i]$ let's consider that S'_i doesn't consist S_{i-1} so it consists another subarray R_{i-1} (ends at $i - 1_{th}$ element) instead of S_{i-1} so $S'_i = R_{i-1} + A[i]$ and $S'_i > S_i$ so we can conclude that $R_{i-1} > S_{i-1}$ that it contradicts our assumption that S_{i-1} is the largest subarray that ends at the $(i-1)_{th}$ element. So $S_i (= S_{i-1} + A[i])$ is the largest subarray if $S_{i-1} > 0$. For other case we can say $S_i = A[i]$ because if $S_{i-1} > 0$ we can conclude $A[i] > S_{i-1} + A[i]$ and there isn't any other option to replace S_{i-1} cause they are also negetive and if they become postive it contradicts our assumption again.

```python
def maximum_subarray_3(A):
    # S_i is the maximum subarray the end in index i
    maximum = float('-inf')
    index = (-1, -1)
    si = 0
    start = 0
    for i in range(len(A)):
        if si <=0:
            si = A[i]
            start = i
        else:
            si += A[i]
        if si > maximum:
            maximum = si
            index = (start, i)

    return (index, maximum)
```

Listing 4: **greedy implementation of the maximum subarray**

4.1 Cost Analysis

The function consists of a for-loop and it traverses n elements in the array and inside of the for-loop the if statements only take $\Theta(1)$. So in total, it takes $\Theta(n)$ to find the largest subarray.

5 Conclusion

Now if we run our code for some randoms inputs, it results in the graphs are shown in Figure 2. As you can see the results are compatible with what we analyzed before. The code of the plotting the graphs is shown in the Listing 5.

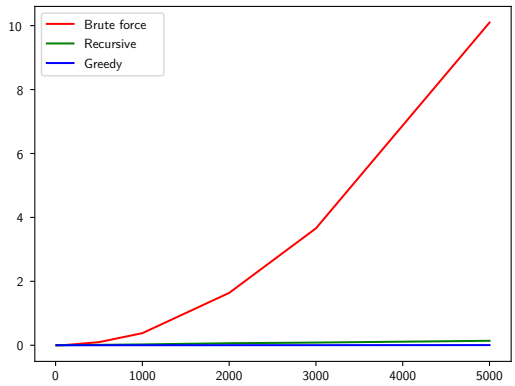
```python
import timeit
from random import randint
import matplotlib.pylab as plt

```

```
from maximum_subarray_sum import maximum_subarray_1, maximum_subarray_2,
    maximum_subarray_3


def func_timer(func, *args, **kwargs):
    '''Take a function and its arguments and return runtime.'''
    def wrap():
        func(*args, **kwargs)
    return timeit.timeit(wrap, number=10)


#Input samples
t = [10,100,500,1000,2000,3000,5000]

brute_force = []
recursive = []
greedy = []

for i in t:
    #creating random array
    A = [randint(-100, 100) for i in range(i)]
    d1 = func_timer(maximum_subarray_1, A)
    d2 = func_timer(maximum_subarray_2, A)
    d3 = func_timer(maximum_subarray_3, A)

    brute_force.append(d1)
    recursive.append(d2)
    greedy.append(d3)

#plotting the runtimes
plt.plot(t, brute_force,'r', label='Brute force')
plt.plot(t, recursive,'g',label='Recursive')
plt.plot(t, greedy,'b',label='Greedy')
plt.legend()
plt.show()
```

Listing 5: Plotting running time of the our three functions on some random inputs

Figure 2: Comparing different algorithms

(a) compare brute-force, divide & conquer and greedy algorithms



(b) compare divide & conquer and greedy