D0012E Lab2 Part 1 Grupp 11:

Elias Grundberg	eligru-3@student.ltu.se	010314
Hjalmar Norén	hjanor-3@student.ltu.se	041029
Pablo Zepada Garcia	pabzep-3@student.ltu.se	021107

2024-12-03

Theory Behind the Algorithm:

This project is composed of two algorithms, one divide and conquer algorithm and a Kagane algorithm as comparison and validation. The algorithm uses an array A = [a_1, a_2,..., 1_n] of non-zero real numbers, our task is to find the maximum sum of any contiguous subarray. How the algorithm achieves this is firstly divide the array into two halves and then recursively compute the maximum subarray for those halves. We use a 4-tuple for calculation:

- The maximum subsequence sum
- The maximum sum for any subsequence starting from the left
- The maximum sum for any subsequence ending from the right
- The total sum of the entire array

Then the maximum subarray sum is returned.

Runtime:

$T(n) = c_1n + c_2n + c_3\frac{n}{2} + c_4\frac{n}{2} + c_5n + c_6n + c_7n + c_8n + c_9n + c_{10}n + c_{11}n + c_{12}n + c_{13}n$

```
def maxRecursive(arr, start, end):                              cost   times
    if start == end:
        return arr[start], arr[start], arr[start], arr[start]   c1     n
    mid = (start + end) // 2                                    c2     n
    left = maxRecursive(arr, start, mid)                        c3     n/2
    right = maxRecursive(arr, mid + 1, end)                     c4     n/2
    max_sum = left[0]                                           c5     n
    if right[0] > max_sum:
        max_sum = right[0]                                      c6     n
    if left[2] + right[1] > max_sum:
        max_sum = left[2] + right[1]                            c7     n
    prefix = left[1]                                            c8     n
    if right[1] + left[3] > prefix:
        prefix = right[1] + left[3]                             c9     n
    suffix = right[2]                                           c10    n
    if left[2] + right[3] > suffix:
        suffix = left[2] + right[3]                             c11    n
        total = left[3] + right[3]array                         c12    n
    return max_sum, prefix, suffix, total                       c13    n
```

Removing unnecessary term our runtime becomes: $T(n) = 2T(\frac{n}{2}) + O(1)$

Using the master theorem this becomes: $T(n) = \Theta(n^{\log_2 2}) = \Theta(n)$

Run cases:

Run Size:	Kadane Results:	Recursive Results:
2^{13} = 8192	0.001000s	0.002520s
2^{18} = 262 144	0.017856s	0.089084s
2^{20} = 1 048 576	0.072213s	0.362340s
2^{24} = 16 777 216	1.183813s	5.667478s
2^{28} = 268 435 456	17.927226s	87.896397s

The reason why our recursive function runs slower than Kadane's Algorithm despite both being of time complexity $\Theta(n)$ is due to the recursion depth of the recursive algorithm being logarithmic because the array is halved at each step.

Code Components:

Kadane Algorithm: This is the most efficient way to find the maximum subarray sum and is used for comparison and validation of the result of our code.

Recursive Algorithm: A divide and conquer algorithm used to divide and compute the maximum subarray sum. We have a base case in which we only have one value in our array but if not the code starts by dividing the code and then calculating it using a 4-tuple.

Main: Runs both Kadane's Algorithm and our Recursive Algorithm, initiates a random array, keeps track of how long each algorithm takes to run and verifies that both algorithms got the same result.

Worst Case: Runs both Kadane's Algorithm and our Recursive Algorithm but initiates the worst possible array which alternates between a very large and very small value, keeps track of how long each algorithm takes to run and verifies that both algorithms got the same result.