Largest Subarray

Daiwei Chen Joseph Watts

February 15, 2019

Abstract

Calculating the maximum sum of values within a contiguous subset of an array is a problem This paper explores the impact of using two different algorithms to solve the problem of calculating the maximum sum of values within a contiguous subset of an array. Due to the different complexity of these two algorithms used to solve it, we will see the performance impact over a range of different sized arrays.

1 Background and Related Work

Finding the largest subarray is a problem where we have to find an subarray within a given list L. And make sure that the found subarray has a sum higher than any other possible subarrays within L. This algorithm is important and has some real world uses. One such as computer vision, to find the brightest area within an image, it's basically a "Maximum Subarray" problem on the lumens within the list of pixels.

1.1 Brute Force Algorithm

Algorithm 1 Brute Force

```
function BRUTEFORCE(A)
    n ← len(A)
    maxsum ← A[0]
    for i in 0..n-1 do
        for j in i..n-1 do
            total ← 0
            for k in i..j do
                total ← total + A[k]
            end for
            if maxsum < total then
                maxsum ← total
            end if
        end for
    end for
end function
```

This brute force algorithm works by keeping track of both a start and ending index, and then calculating a sum for all the indexes between.

$$\sum_{i=0}^{n-1}\sum_{j=i}^{n-1}\sum_{k=i}^{j}1$$

$$
\begin{aligned}
&\sum_{i=1}^{n}\sum_{j=1}^{i}\sum_{k=j}^{i+j} 1 \\
&\quad = \sum_{i=1}^{n}\sum_{j=1}^{i} i+1 \\
&\quad = \sum_{i=1}^{n}\sum_{j=1}^{i} i + \sum_{j=1}^{i} 1 \\
&\quad = \sum_{i=1}^{n} \frac{i(i+1)}{2} + i \\
&\quad = \sum_{i=1}^{n} (\frac{1}{2}i^2 + \frac{i}{2}) + \sum_{i=1}^{n} i \\
&\quad = \frac{1}{2}\sum_{i=1}^{n} i^2 + \frac{1}{2}\sum_{i=1}^{n} i + \sum_{i=1}^{n} i \\
&\quad = \frac{1}{2} * \frac{1}{6} n(n+1)(2n+1) + \frac{1}{2}\frac{n(n+1)}{2} + \frac{n(n+1)}{2} \\
&\quad = \frac{1}{12} n(2n^2+3n+1) + \frac{(n^2+n)}{4} + \frac{2(n^2+n)}{4} \\
&\quad = \frac{1}{6}n^3 + \frac{1}{4}n^2 + \frac{1}{12}n + \frac{3(n^2+n)}{4} \\
&\quad = \frac{1}{6}n^3 + \frac{1}{4}n^2 + \frac{1}{12}n + \frac{3n^2+3n}{4} \\
&\quad = \frac{1}{6}n^3 + n^2 + \frac{10}{12}n
\end{aligned}
\tag{1}
$$

The starting index goes from the position of the first element until the last element, while the ending index repeats for all values between the starting index and the position of the last element. A third loop then sums all the values found between these two indexes.

The outer loop keeps track of i, the starting index, and runs the entire length of the array, or n times. The middle loop keeps track j, the ending index, and runs the length between i and the end of the array, n. There is then another inner value, k, which runs between the values of i and j.

Due to these three nested summations, the brute force algorithm runs in $O(n^3)$ time. It should be noted, however, that there is a slightly different version of this brute force algorithm that runs in $O(n^2)$ time by doing the sum as part of the second loop.

1.2 Kadane Algorithm

Algorithm 2 Kadane Algorithm

```
function KADANE(L)
    maxEnding ← L[0]
    maxAlways ← L[0]
    for i 1..len(L) do
        maxEnding ← MAX(L[i], maxEnding + L[i])
        maxAlways ← MAX(maxAlways, maxEnding)
    end for
    return maxAlways
end function
```

Kadane is a very good example of a simple but effective way to write a better algorithm using dynamic programming. It focuses on remembering two very important variables $maxEnding$ and $maxAlways$.

maxAlways will always remember the largest sum you've seen up till now out of all the subarrays. But *maxEnding* will keep track of the largest subset just within that iteration of i. Because it remembers what sort of sums you've checked before, you do not have to check other possible subarrays again. Thus getting rid of the 2 for loops within the brute force algorithm.

$$\sum_{1}^{n} 2 = 2 * n = \Theta(n)$$

This algorithm has a big Θ of n because it only goes through each element of the array once and only once.

2 Experimental Setup

For the experimental setup, we have allowed our users to either input an array to test both algorithms against, or run our own tests. Our own tests included of multiple n values that will be timed. For each n value, we randomly generate n numbers of numbers between -5 and 5 inclusive and place them into a list. We then will time each run of the brute force algorithm and kadane algorithm to go through that list of random numbers. After about 100 trials for each n, we will collecte all the timing data and proceed to find the averge run time for both algorithms for that n. Timing data is collected by using a high definition clock within Rust.

3 Results

Include our graph visualization of our data here. Brute Force and Kadane Timing

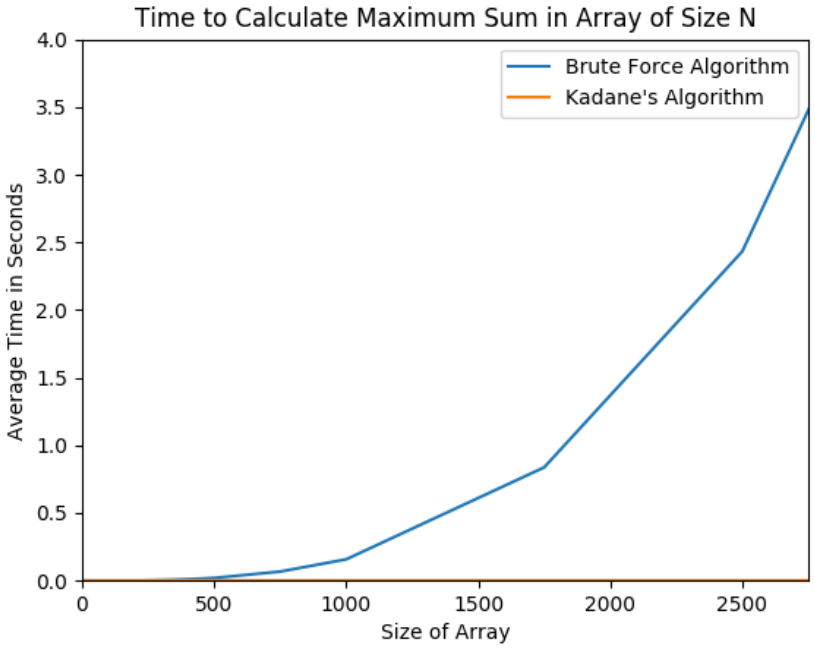

Figure 1: Graph of Brute Force vs Kadane Timings

Figure 2: Table of Brute Force vs Kadane Timings

Size of Array	Brute Force (s)	Kadane (s)
1	0.000000033	0.000000025
5	0.000000085	0.000000036
10	0.000000448	0.000000051
50	0.000030930	0.000000125
100	0.000194367	0.000000184
150	0.000594095	0.000000260
200	0.001366055	0.000000331
250	0.002607847	0.000000402
300	0.004420136	0.000000466
350	0.006950275	0.000000573
400	0.010421218	0.000000639
450	0.014655456	0.000000716
500	0.020030729	0.000000956
750	0.067125954	0.000001231
1000	0.157617270	0.000001779
1750	0.837418008	0.000002682
2500	2.432867191	0.000003601
3500	6.606255604	0.000005064

4 Conclusions

As cleary shown through our graph and time table found in Section 3, Kadane's algorithm is a much more optimized solution to solve the problem of finding the maximum sum of values within an array. A brute forcing algorithm can be simplified down to $O(n^2)$, but our algorithm runs in $O(n^3)$. At the same time Kadane's algorithm runs in $O(n)$ time meaning that it will execute much faster. The amount of time that this algorithm saves is exponential.