CS325 Winter 2013: Implementation 2

Daniel Reichert
Trevor Bramwell

February 25, 2013

1 Theoretical Analysis

1.1 Pseudocode

The following is pseudocode to solve the Maximum Subarray problem.

Brute Force:

```
brute_sum(n)
    size = the size of n
    max = first element in the n array
    for i = 0 .. size+1:
        sums = array of size+1 with each element initialized to n[0]
        for j = i+1 .. size+1:
            if i < j:
                sums[j] = the sum of the elements from i to j-2 plus n[j-1]
                if sums[j] > max_sum:
                    max = sums[j]
    return max
```

Divide and Conquer:

```
divide_conquer(A)
    if length A > 1:
        return A[0]

    if length A = 2:
        return max(A[0], A[1])
```

```
    midpoint := length A / 2

    left := divide_conquer(A[0 ... midpoint])
    right := divide_conquer(A[midpoint ... length A])

    max_left := midpoint element in A
    sum := 0
    for x = midpoint of A ... 0
        sum := sum + x
        max_left := max(sum, max_left)

    max_right := midpoint element in A
    sum := 0
    for y = midpoint of A ... length of A
        sum := sum + y
        max_rigth := max(sum, max_right)

    return max(left, right, (max_left+max_right))
```

Dynamic:

```
dynamic(n)
    cummulative = 0
    high_water = n[0]
    for i = 0 .. size of n
        cummulative = max of (0, cummulative+n[i])
        high_water = max of (cummulative, high_water)
    return high_water
```

2 Asymptotic Run Time Analysis

1. In the brute force algorithm there are no subproblems. With the nested for loops, there is a comparison made between every element in the input, leading to an intutitive asymptotic complexity of $O(n^2)$

2. In the divide and conquer algorithm the problem is branched into 2 subproblems for a size of $n/2$ at each level. Those two subproblems cover the two cases where the maximal subsequence is fully contained within

either half of the array. The case where the maximal subsequence is contained partially in each half of the array can be analyzed in linear time. Which leads to a complexity statement of $T(n) = 2T(n/2) + O(n)$. Per the master theorem, this leads to a complexity of $O(nlogn)$.

3. In the dyanmic programming algorithm the problem is approached from the bottom up. Like all dynamic programming algorithms the complexity can be modeled by the size of the table that is created multiplied by the time it takes to fill in a cell of that table. Since the time to fill in a cell is constant and the table is 1 dimensional, the run time is $c * n$ or $O(n)$

3 Theoretical Correctness

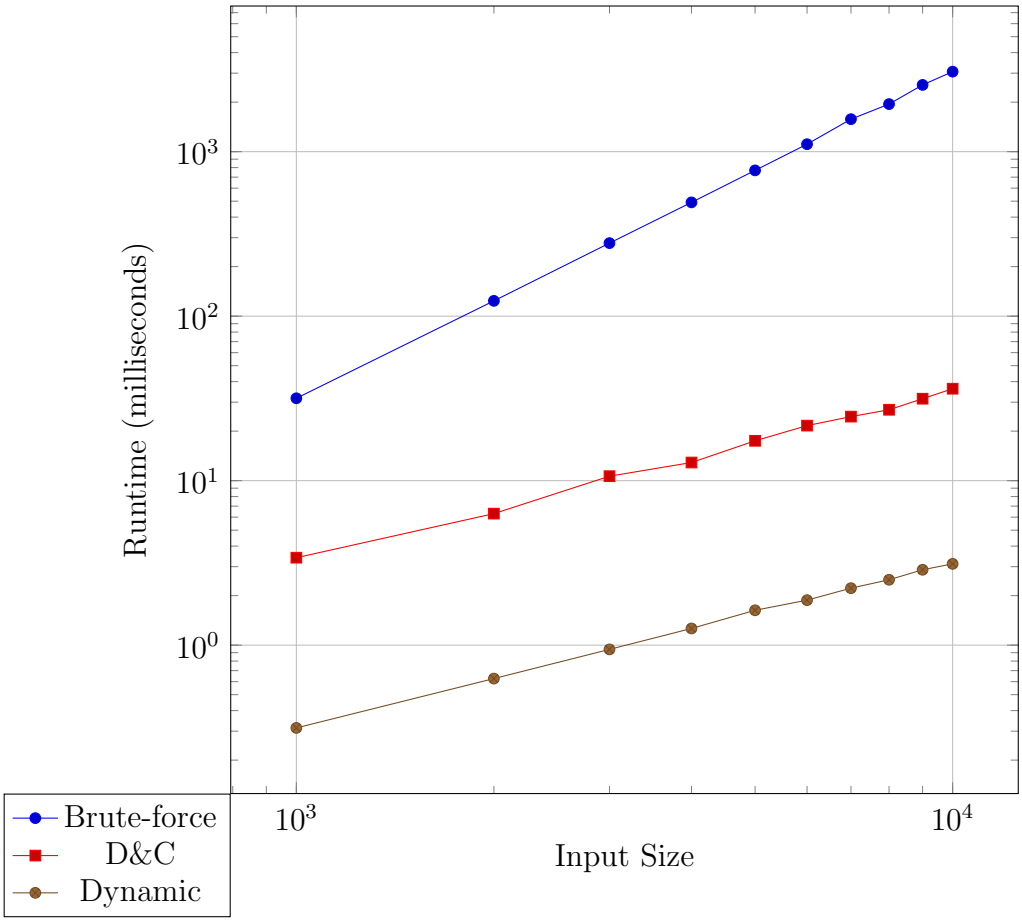
1. In the Divide and conquer algorithm the problem is divided into two halves. The maximal subsequence can be contained in one of three different cases. The first two cases are when the maximal subsequence is contained entirely in one of the two halves. The third case is when the maximal subsequence is partially in both halves of the array. Since the algorithm considers all three of these cases we know that the algorithm will calculate the correct maximum value.

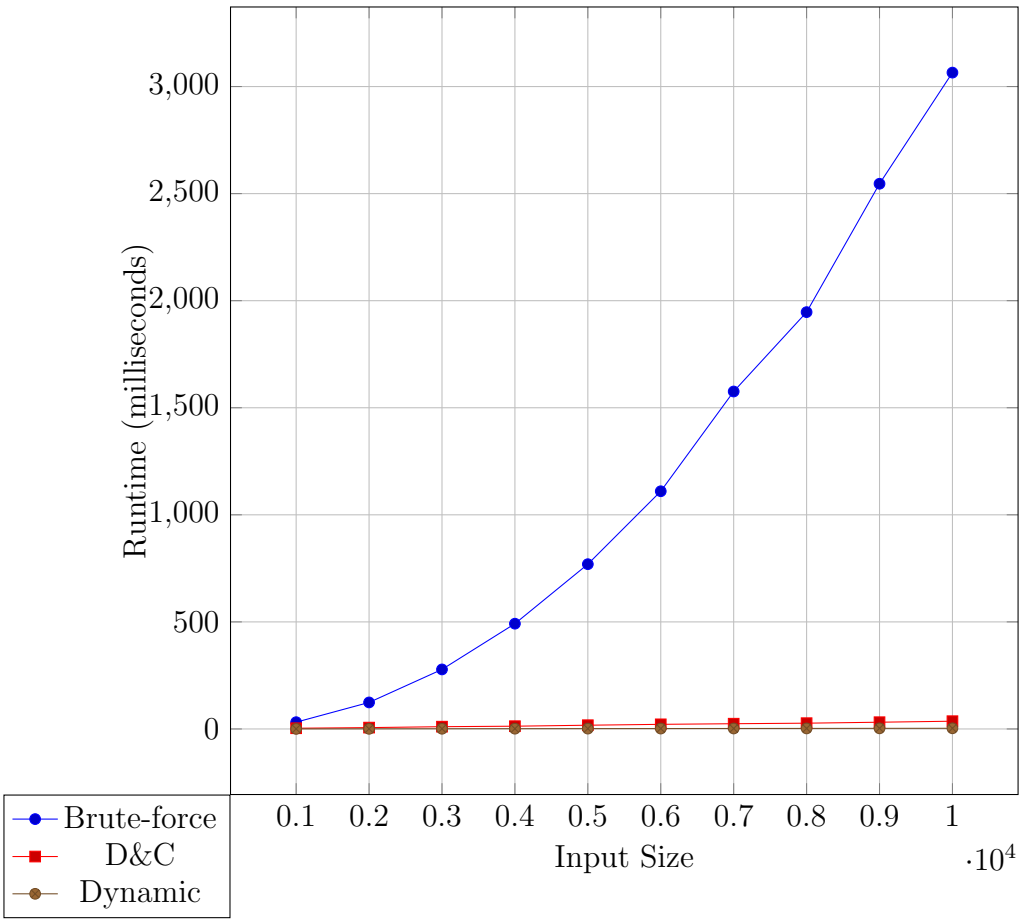
2. In the Dynamic programming algorithm every possible case is considered in a linear fashion. Since the maximum value of all cases that have been considered is continually updated with each new subsequence, we know that the maximum value will be captured.

Empirical Testing

finaltest.txt
25126
38269
14100
30372
23653
28691
17933
28665
8437
14274

Empirical Analysis of Runtime





4 Algorithm Comparison

As can be seen in the graphs above, the three algorithms have clearly different running times. As we explained in the section discussing the run time analysis the each have a different order of complexity so differences in the time taken to run are expected and are amplified as the input size is increased. Beyond the complexity there is also a difference in memory use. Because the divide and conquer algorithm needs to create and compare subarrays it's memory usage is highest.