CS 325

Group Assignment 1

Ujjval Kumaria, Vijay Vardhan Tadimeti, Aashwin Vats

Algorithm 1: Enumeration

Pseudocode

```
FUNCTION max_sub_array(array):
    global_max <- float('-inf')
    n <- len(array)
    maximum <- 0
    FOR i in range(0, n):
        FOR j in range(i, n + 1):
            current_sum <- 0
            FOR k in range(i, j):
                current_sum = current_sum + array[k]
                IF current_sum > maximum:
                    maximum <- current_sum
    RETURN maximum
```

Theoretical run-time analysis

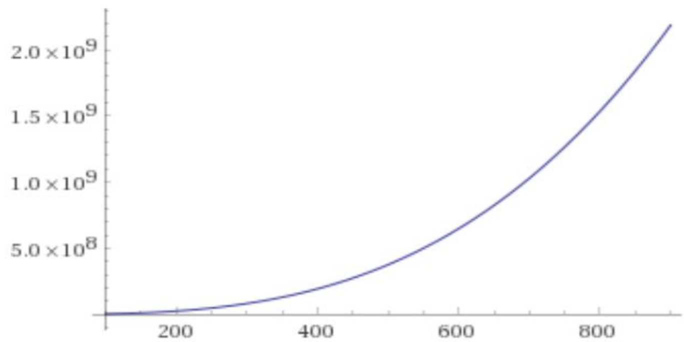
The run time for this algorithm is $O(n^3)$ since there are 3 for loops. The sum can be represented as $\sum_{i=1}^{n} . \sum_{j=i}^{n} . \sum_{k=j}^{n} . 3k + 2$. Following is the graph for the theoritical equation that we represented by doing run-time analysis on the above pseudocode.

Input interpretation:

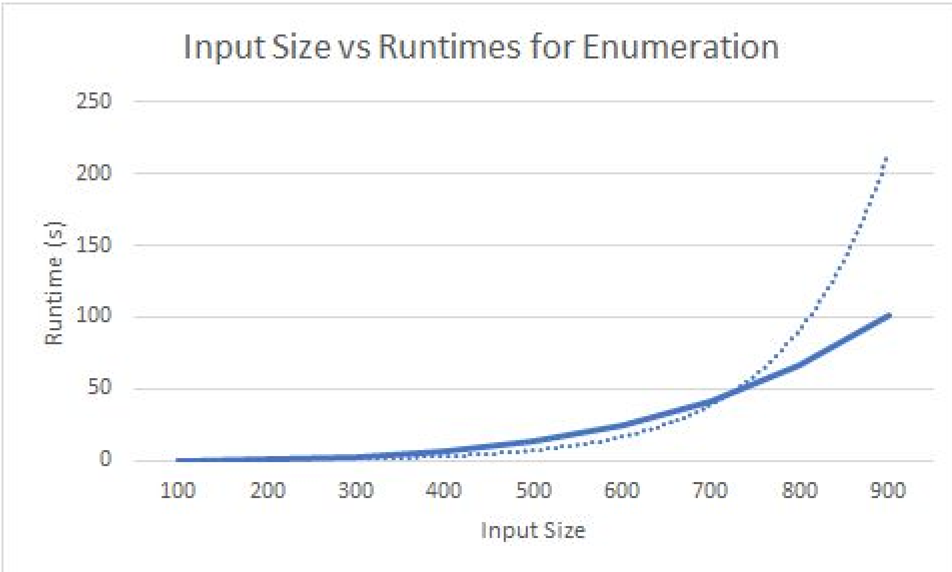
plot	$3k^3 + 2$	$k = 100$ to 900

Enlarge | Data | Customize | Plaintext | Interactive

Plot:



Experimental run-time analysis



Algorithm 2: Better enumeration

Pseudocode

```
FUNCTION max_sub_array_better(array):
    max <- float('-inf')
    FOR i in range(len(array) + 1):
        FOR j in range(len(array) + 1):
            iter_max <- sum(array[i:j])       (sum elements from i to j)
            IF iter_max > max:
                max <- iter_max
    RETURN max
```

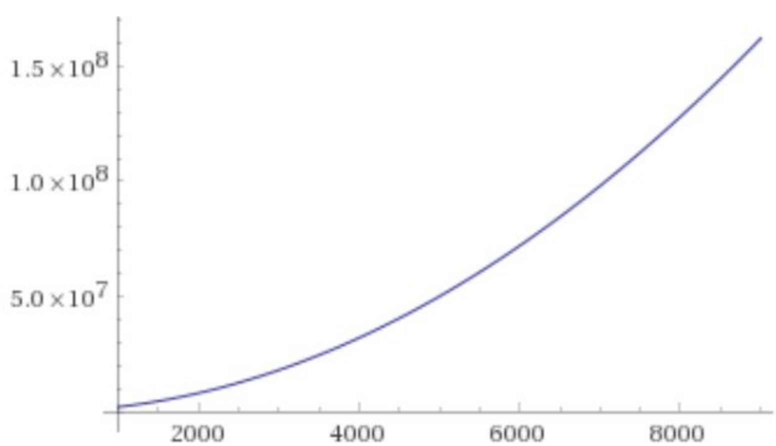
Theoretical run-time analysis

The run time for this algorithm is O(n^2) as can be clearly observed from the 2 for loops. The sum can be represented as $\sum_{i=1}^{n} . \sum_{j=i}^{n} . 2j + 3$. Following is the graph for the theoritical equation that we represented by doing run-time analysis on the above pseudocode.

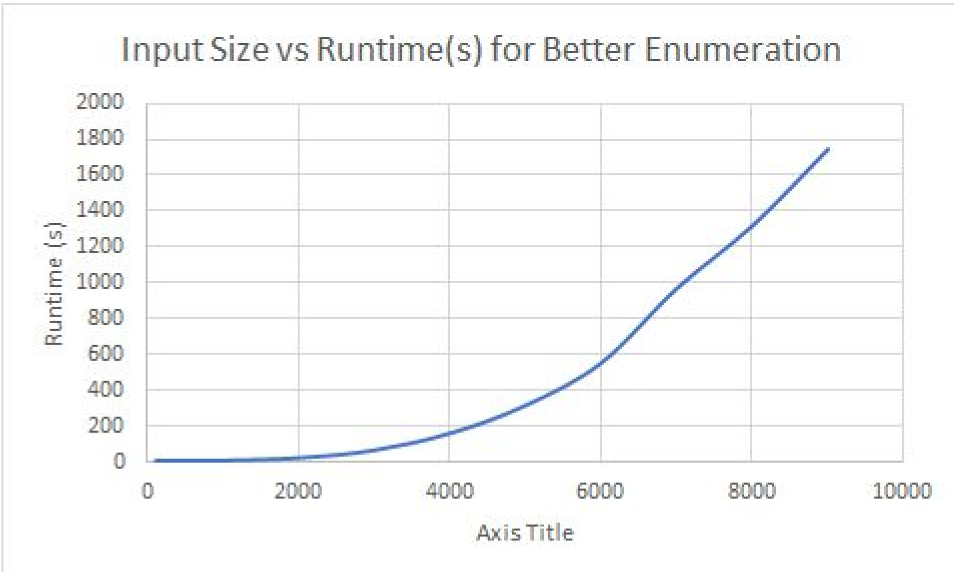
Input interpretation:

plot	$2 j^2 + 3$	$j = 1000$ to 9000

Plot:



Experimental run-time analysis



Algorithm 3: Dynamic programming

Pseudocode

```
FUNCTION maxSubArraySum(a, size):
    max_so_far <- ((-maxsize) - 1)
    max_ending_here <- 0
    start <- 0
    end <- 0
    index <- 0

    FOR i in range(0, size):
        max_ending_here += a[i]
        IF max_so_far < max_ending_here:
            max_so_far <- max_ending_here
            start <- index
            end <- i
        IF max_ending_here < 0:
            max_ending_here <- 0
            index <- i + 1

    IF max_so_far < 0:
        max_so_far <- 0
    RETURN max_so_far
```

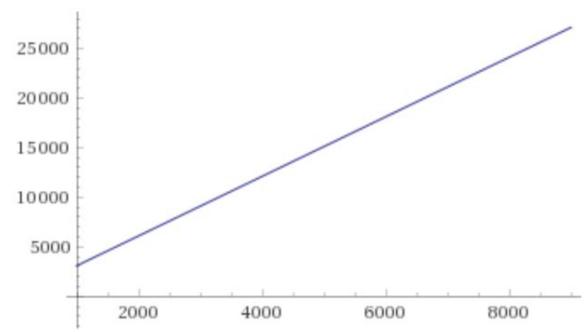
Run-time analysis

The run time for the DP implementation of this problem gives an asymptotic running time of O(n). The sum can be represented as $\sum_{i=1}^{n} . 3i + 2$. Following is the graph for the theoretical equation that we represented by doing run-time analysis on the above pseudocode.

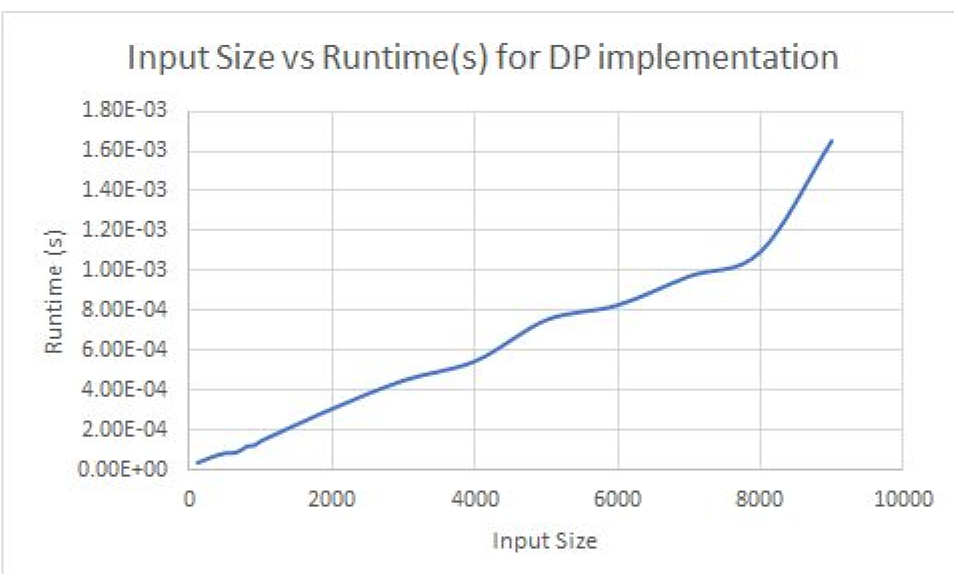
Input interpretation:

plot	$3n + 2$	$n = 1000$ to 9000

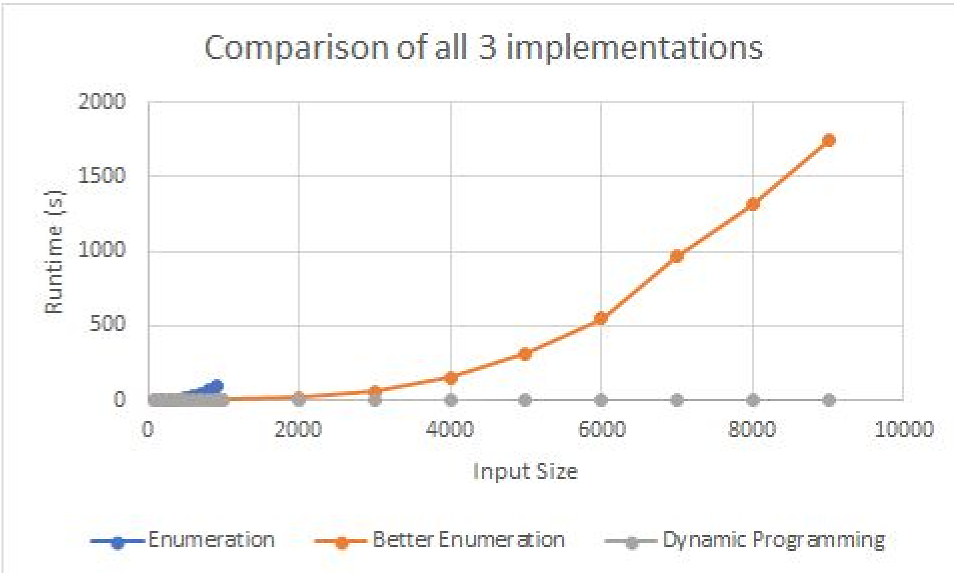
Plot:



Experimental run-time analysis



Comparison of all implementations



From the plots seen above, it is clearly observable that the experimental runtimes were mostly consistent with the theoretical ones. It can be seen that the most efficient algorithm out of all three was the one where we used dynamic programming, giving a linear time complexity. Since the experimental time results for Algo 1 would have taken much longer to compute, we only computed it for length of arrays from 100 to 900(as can be seen in the comparison graph).