CS 325 Project 1: Max Sum Subarray

Nathan Stauffer, Daniel Olivas, Joanna Lew
Project Group 7

Spring 2017

Theoretical Run-time Analysis

Give pseudo-code for each of the four algorithms and an analysis of the asymptotic running-times of the algorithms.

(a) **Algorithm 1: Enumeration**

The idea for this algorithm was to loop over every possible subarray using a double for-loop, and get the sum of the subarray by adding the elements with another for-loop. We kept a saved "maximum" sum that with which we compared each subarray's sum.

The psuedocode is provided as follows:

```
ALGO1_ENUM(A[0 ... n - 1]){
   for (i = 0; i < n; i++){              // loop over the whole array
      for (j = i; j < n; j++){           // loop to get a subarray
         tempSum = 0;
         for (k = i to j; k++)           // loop to get sum of subarray
              tempSum = tempSum + A[k]

          if (tempSum > maxSum){         // compare sum of subarray
             maxSum = tempSum            // with saved maximum
             save subarray
          }
      }
   }
   return maxSum and subarray
}
```

The algorithm runs two for-loops over the span of the entire array, and one for-loop over the span of the current subarray. Since the worst case for the subarray is to be size n, where n is the size of the array, the run-time must be $n \cdot n \cdot n = O(n^3)$. Alternatively, the run-time can be calculated mathematically as follows, where $N = n - 1$:

$$\sum_{i=0}^{N}\sum_{j=i}^{N}\sum_{k=i}^{j}\Theta(1) = \sum_{i=0}^{N}\sum_{j=i}^{N}(-i+j+1)\,\Theta(1)$$

$$= \sum_{i=0}^{N}\frac{1}{2}(i-N-2)(i-N-1)\,\Theta(1)$$

$$= \frac{1}{6}(N+1)(N^2+5N+6)\,\Theta(1)$$

$$= \Theta(N^3)\,\Theta(1) = \Theta(N^3) = \Theta(n^3)$$

(b) **Algorithm 2: Better Enumeration**

The idea for this algorithm was to build on Algorithm 1. Instead of using a for-loop to calculate the sum of the subarray, one can keep a saved running sum. This improves run-time since instead of using $\sum_{k=i}^{j} A[k]$, which is O(n), we add $A[k]$ to the saved sum, which is $O(1)$. The psuedocode is as follows:

```
ALGO2_BETTER_ENUM(A[0 ... n - 1]){
   for (i = 0; i < n; i++){              // loop over the whole array
    tempSum = 0;
      for (j = i; j < n; j++){           // loop to get a subarray
           tempSum = tempSum + A[k]      // add A[k] to running saved sum

           if (tempSum > maxSum){        // compare sum of subarray
              maxSum = tempSum           // with saved maximum
              save subarray
           }
      }
   }
   return maxSum and subarray
}
```

Since there are two for-loops over the span of the entire array, the run-time must be $O(n^2)$. We can also show this mathematically.

$$\sum_{i=0}^{N}\sum_{j=i}^{N}\Theta(1) = \sum_{i=0}^{N}(-i+N+1)\,\Theta(1)$$

$$= \frac{1}{2}(N+1)(N+2)\,\Theta(1)$$

$$= \frac{1}{2}(n)(n+1)\,\Theta(1)$$

$$= \frac{1}{2}(n^2+n)\,\Theta(1)$$

$$= \Theta(n^2)\,\Theta(1) = \Theta(n^2)$$

(c) **Algorithm 3: Divide and Conquer**

The idea for this algorithm was to recursively find the maximum sum. If we split the array into two halves, the maximum sum subarray must either be contained entirely in the first half, entirely in the second half, or made of a combination of both the first half and the second half. If the maximum sum subarray is a combination, then since it must be continguous, it must contain the suffix (latter part) of the first half and the prefix (former part) of the second half.

The base case for this algorithm was arrays with one element must return that element as the maximum sum subarray. Arrays with more than one element recursively called itself, passing "up" to the parent function the maximum sum of the left, right, and cross subarrays. It, then, chose the largest of the three to find the maximum sum subarray. The psuedocode is provided as follows:

```
ALGO3_DIV&CONQ_MAIN(A[0 ... n - 1]){
   if (A is empty array)           // loop over the whole array
      return 0 as max sum, no sum subarray

   // make some variables to pass "up"
   Declare maax, lmax, rmax, sum    // maax = max sum subarray
   ALGO3_DIV&CONQUER_HELPER(A[0 ... n-1], begin, end, maax, lmax, rmax, sum)
   return maax
}

ALGO3_DIV&CONQ_HELPER(A[0 ... n-1], begin, end, &maax, &lmax, &rmax, &sum){
   // base case: 1 element, return it as max left, right, cross
    if (begin == end)
       maax, lmax, rmax, sum = A[begin]

    // all other arrays
    else {
       mid = (begin + end) / 2
       maax1, lmax1, rmax1, sum1      // variables for left half subarray
       maax2, lmax2, rmax2, sum2      // variables for right half subarray

       // recursively call on left half, right half
       // split in two each time until base case reached
       ALGO3_DIV&CONQ_HELPER(A, 0, mid, maax1, lmax1, rmax1, sum1)
       ALGO3_DIV&CONQ_HELPER(A, mid+1, end, maax2, lmax2, rmax2, sum2)

       // the max of subarray is either max of left, max of right
       // or suffix of left + prefix of right
       maax = largest(maax1, maax2, rmax1 + lmax2)

       // the max of the left subarray is either
       // the max of the left subarray's left subarray, or a sum
       lmax = largest(lmax1, sum1 + lmax2)
```

```
        rmax = largest(rmax2, sum2 + rmax1)

        // sum over the entire subarray is left sum + right sum
        sum = sum1 + sum2
    }
    // maax, lmax, rmax, sum are returned by reference
}
```

For the algorithm's base case (array of no elements or a single element), it simply returns a value. Therefore, $T(1) = \Theta(1)$.

For all other cases, since the algorithm splits the array in half each time and performs the recursion twice, the run-time can be represented as $T(n) = 2\,T(n/2) + \Theta(1)$.

Master Method: $T(n) = aT(\frac{n}{b}) + f(n)$
where $a \geq 1$, $b > 1$, and $f(n) > 0$.

Case 1: if $f(n) = O(n^{log_b a - \epsilon})$ for some $\epsilon > 0$, then $T(n) = \Theta(n^{log_b a})$
Case 2: if $f(n) = \Theta(n^{log_b a})$, then $T(n) = \Theta(n^{log_b a}\, lg\, n)$
Case 3: if $f(n) = \Omega(n^{log_b a + \epsilon})$ for some $\epsilon > 0$, and
if $a\, f(n/b) \leq c\, f(n)$ for some $c < 1$ and all sufficiently large n, then
$T(n) = \Theta(f(n))$

From the equation given above, $a = 2$, $b = 2$, $f(n) = x$, where x is some constant.
Since $log_b a = log_2 2 = 1$, $n^{log_b a} = n^1 = n$
Then, $f(n) = O(n^{log_b a - \epsilon}) = O(n^{1-\epsilon})$, where $\epsilon = 1$
Therefore, $T(n) = \Theta(n^{log_b a}) = \Theta(n)$

(d) **Algorithm 4: Linear-time**

The idea for this algorithm was to loop through the array once, keeping a running sum. If the sum was less than zero, reset it back to zero, and mark the next element in the array to be the start of a potential max sum subarray. Otherwise, extend the subarray to the right and add the element to the running sum. If the sum exceeds the global max, update the global max.

```
ALGO4_LINEAR(A[0 ... n - 1]){
  for (i = 0; i < n; i++){              // loop over the whole array
      tempSum = tempSum + A[i]          // add element to running sum

      if (tempSum <= 0){
         tempSum = 0;
         set subarray endpoints to next element
      }
      else
         extend subarray to right by 1

      if (tempSum > maxSum)
         maxSum = tempSum;
  }
```

```
    return maxSum and subarray;
}
```

The algorithm uses one for-loop that spans the entire array, which means its run-time is $\Theta(n)$. We can show this mathematically.

$$\sum_{i=0}^{N} \Theta(1) = N\,\Theta(1) = \Theta(N) = \Theta(n)$$

Testing

We tested the above four algorithms using the following vectors. We tried to test cases where the maximum sum subarray was the first element, had only one element, or had more than one element. We also tested cases where the maximum sum subarray had 0, 1, or multiple negative numbers.

(a) $v = \{31, -41, 59, 26, -53, 58, 97, -93, -23, 84\}$
$max = \{59, 26, -53, 58, 97\} = 187$

We wanted to test against a case where there were positives and negatives in the vector, and in the maximum sum subarray. We also wanted to test a case where the maximum sum subarray had more than one element.

(b) $v = \{31\}$
$max = \{31\} = 31$

We wanted to test against a case where there was only a single element in the vector.

(c) $v = \{-21, 12\}$
$max = \{12\} = 12$

We wanted to test against a case where there were multiple elements in the vector, but the maximum sum subarray was a single element.

(d) $v = \{12, -21\}$
$max = \{12\} = 12$

We wanted to test against a case where there were multiple elements in the vector, and the maximum sum subarray was the first element in the vector.

(e) $v = \{-5, 1, 2, 3, -1, -1, 4, 5, -7\}$
$max = \{1, 2, 3, -1, -1, 4, 5\} = 13$

We wanted to test against a case where the maximum sum subarray had multiple elements and contained more than one negative number.

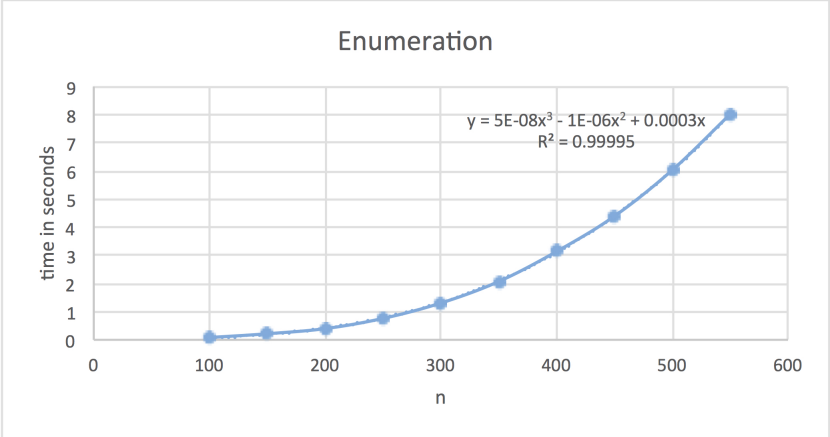
(f) $v = \{1, 2, 3, 4, 5\}$
$max = \{1, 2, 3, 4, 5\} = 15$

We wanted to test against a case where the maximum sum subarray spanned the length of the entire original array.

Experimental Analysis

(a) **Algorithm 1: Enumeration**

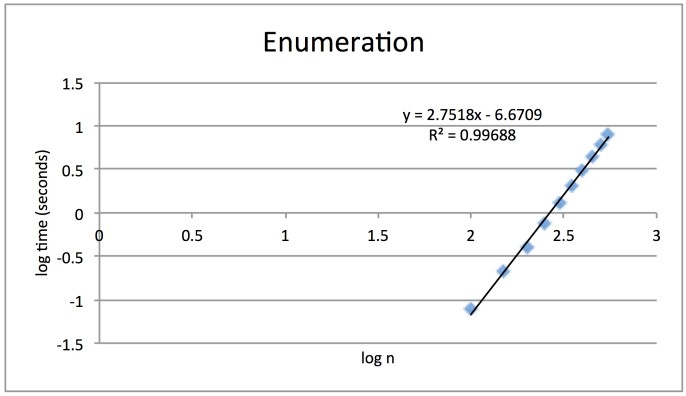
We calculated the average running time for algorithm 1 for 10 sizes between [100, 550] incrementing 50 each time. Below is a plot of the time vs array size.



Using Excel's regression analysis, we determined the best-fit curve to be $T(n) = (5 \times 10^{-8})\, n^3 - (1 \times 10^{-6})\, n^2 + (3 \times 10^{-4})\, n$. The run time is therefore a polynomial function with n^3 as its largest term.

The exponential asymptotic run-time is $\Theta(n^3)$, which matches the theoretical asymptotic run-time.

We can use a log-log plot to check the experimental run-time.



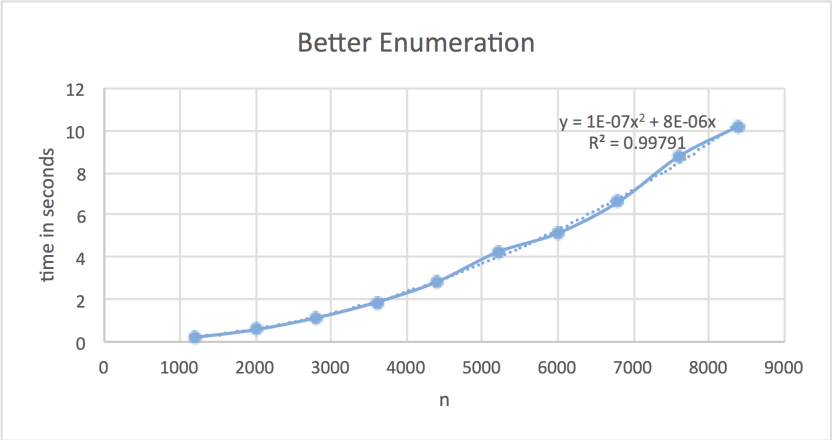
Using Excel's linear regression, the slope of the log-log function is 2.75. Therefore, our experimental run-time according to the log-log plot should be $O(n^{2.75})$. This value is close to our best-fit curve. Error was likely introduced when taking the log_{10} of n and time, especially since the values are relatively small and close together.

Using the equation provided by Excel's regression:
$T(n) = (5 \times 10^{-8})\, n^3 - (1 \times 10^{-6})\, n^2 + (3 \times 10^{-4})\, n$
We can determine, the largest input size for the algorithm that can be solved in 5 seconds, 10 seconds and 1 minute.

Time (seconds)	Array Size
5	453
10	574
60	1 054

(b) **Algorithm 2: Better Enumeration**

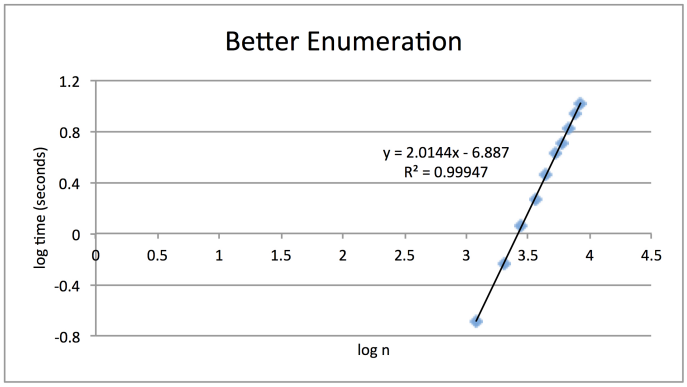
We calculated the average run-time for algorithm 2 for 10 sizes between [1200, 8400], incrementing 800 each time. Below is a plot of the time vs array size.



Using Excel's regression analysis, we determined the best-fit curve to be $T(n) = (1 \times 10^{-7})\, n^2 + (8 \times 10^{-6})\, n$. The run time is therefore a quadratic equation.

The exponential asymptotic run-time is $\Theta(n^2)$, which matches the theoretical asymptotic run-time.

We can use a log-log plot to check the experimental run-time.



Using Excel's linear regression, the slope of the log-log function is 2.01. Therefore, our experimental run-time according to the log-log plot should be $O(n^{2.01})$. This value is very similar to the value we derived from the best-fit curve.

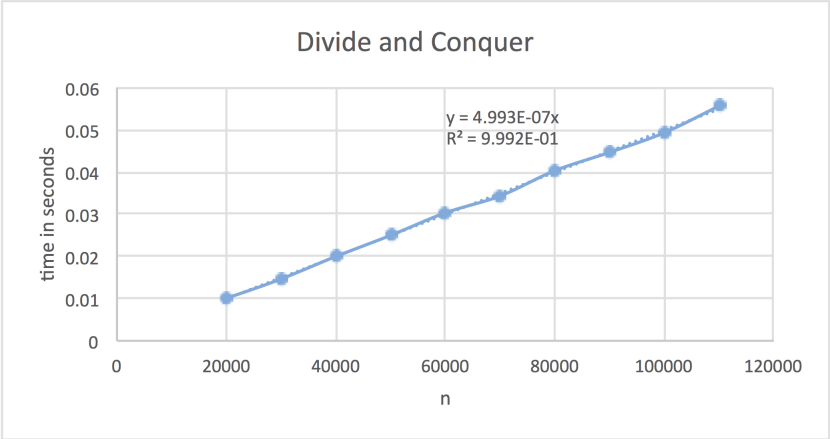
Using the equation provided by Excel's regression:
$T(n) = (1 \times 10^{-7})\, n^2 + (8 \times 10^{-6})\, n$

We can determine, the largest input size for the algorithm that can be solved in 5 seconds, 10 seconds and 1 minute.

Time (seconds)	Array Size
5	7 031
10	9 960
60	24 455

(c) **Algorithm 3: Divide and Conquer**

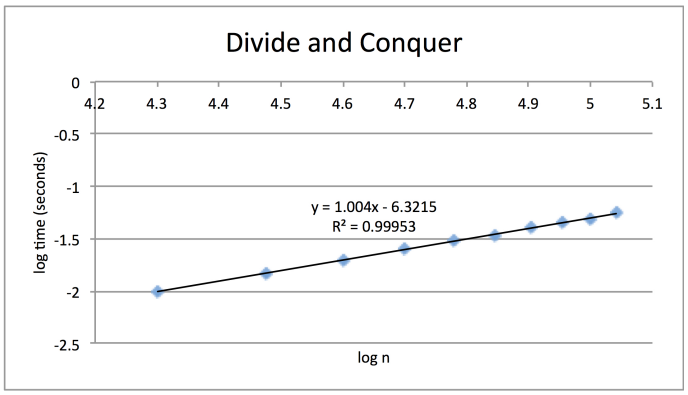
We calculated the average run-time for algorithm 3 for 10 sizes between [20,000, 110,000], incrementing 10,000 each time. Below is a plot of the time vs array size.



Using Excel's regression analysis, we determined the best-fit curve to be $T(n) = (4.99 \times 10^{-7})\, n$. The run time is therefore a linear equation.

The exponential asymptotic run-time is $\Theta(n)$, which matches the theoretical asymptotic run-time.

We can use a log-log plot to check the experimental run-time.



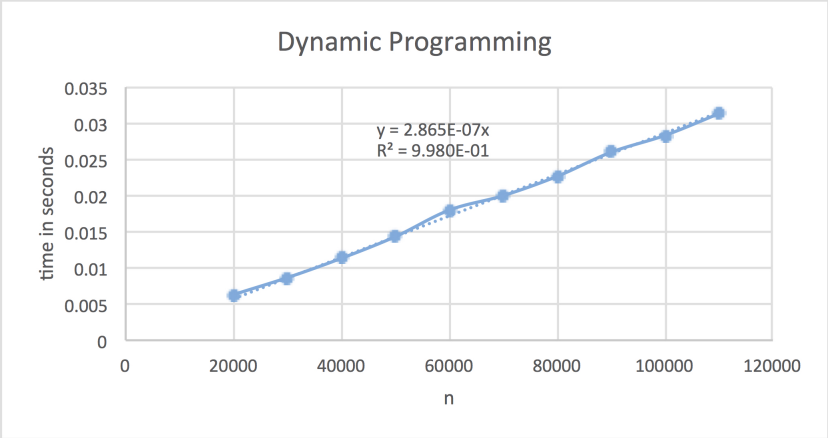
Using Excel's linear regression, the slope of the log-log function is 1.00. Therefore, our experimental run-time according to the log-log plot should be $O(n)$. This value matches the value we derived from the best-fit curve.

Using the equation provided by Excel's regression:
$T(n) = (4.99 \times 10^{-7})\, n$
We can determine, the largest input size for the algorithm that can be solved in 5 seconds, 10 seconds and 1 minute.

Time (seconds)	Array Size
5	1.0×10^7
10	2.0×10^7
60	1.2×10^8

(d) **Algorithm 4: Linear Time**

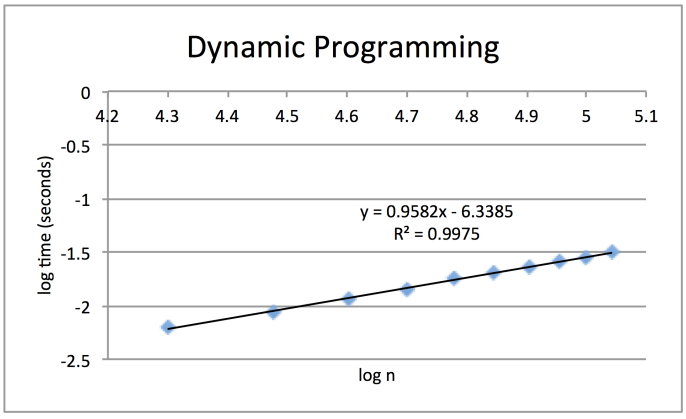
We calculated the average run-time for algorithm 4 for 10 sizes between [20,000, 110,000], incrementing 10,000 each time. Below is a plot of the time vs array size.



Using Excel's regression analysis, we determined the best-fit curve to be $T(n) = (2.87 \times 10^{-7})\, n$. The run time is therefore a linear equation.

The exponential asymptotic run-time is $\Theta(n)$, which matches the theoretical asymptotic run-time.

We can use a log-log plot to check the experimental run-time.



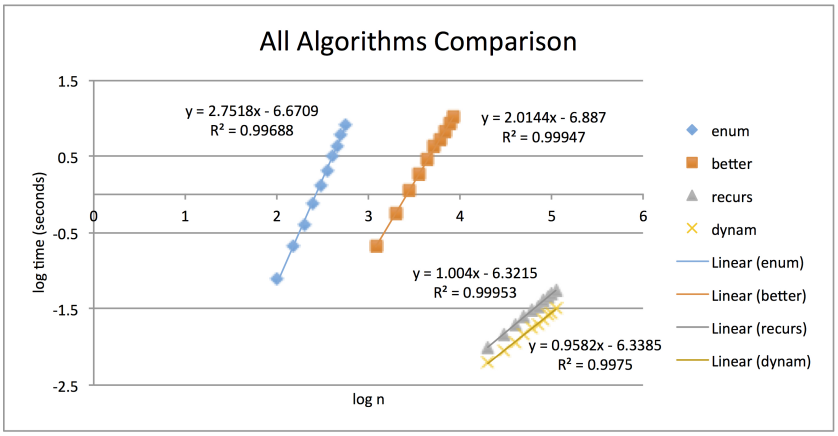
Using Excel's linear regression, the slope of the log-log function is 0.96. Therefore, our experimental run-time according to the log-log plot should be $O(n^{0.96})$. This value is very similar to the value we derived from the best-fit curve.

Using the equation provided by Excel's regression:
$T(n) = (5 \times 10^{-7})\, n$
We can determine, the largest input size for the algorithm that can be solved in 5 seconds, 10 seconds and 1 minute.

Time (seconds)	Array Size
5	1.74×10^7
10	3.48×10^7
60	2.09×10^8

Algorithm Comparison

We can visually compare the experimental run-times of the algorithms by plotting them on the same graph.



The log-log plot above shows the various run-times by slope. The steeper (larger slope) a line is, the greater the degree of the run-time is. Therefore, the run-time with the largest slope (algorithm 1: blue) would run the slowest, which is reinforced by its theoretical and experimental run-time of $O(n^3)$. The recursive algorithm (grey) and dynamic programming (yellow) algorithm would run the fastest, since their lines increase much slower (smaller slope). This is also shown by their theoretical and experimental run-time of $O(n)$.