Eric Timmerman, Stephanie Ison, Geoffrey Corey
October 23, 2013
CS 325 Proj 2

1 Run-time Analysis

1.1 Pseudo-code

1.1.1 Algorithm 1

```
algorithm1 (num_list, num_list2):
    result = algorithm1_helper(num_list)
    result2 = algorithm1_helper(num_list2)
    total = result[0] - result2[0]
    return total


algorithm1_helper (num_list):
    min = sys.maxint
    total = 0
    for i in range(len(num_list) - 1, -1, -1):
        total = total + num_list[i]
        if (abs(total) < min):
            min = abs(total)
            left = i
            right = len(num_list)
    results = (min, left, right)
    return results
```

1.1.2 Algorithm 2

```
algorithm2 (num_list, num_list2):
    num_list = algorithm2_helper (num_list)
    num_list2 = algorithm2_helper (num_list2)
    num_list.sort()
    num_list2.sort()
    min = num_list[0] - num_list2[0]
    return min

algorithm2_helper (num_list):
    list = []
    total = 0
    for i in range(len(num_list) - 1, -1, -1):
        total = total + num_list[i]
        list.append(total)
    return list
```

1.1.3 Algorithm 3

```
algorithm3 (num_list):
    half = len(num_list)
    half = half / 2
    first_half = num_list[: half]
    second_half = num_list[half: ]
    min_left = algorithm34_helper(first_half)
```

```
    min_right = algorithm34_helper(second_half)
    second_half.reverse()
    algorithm1 (first_half, second_half)

algorithm34_helper(half):
    if len(half) <= 1:
        return 0
    left = half[ : len(half) / 2]
    right = half[len(half) / 2 : ]
    left_min = algorithm34_helper(left)
    right_min = algorithm34_helper(right)
    cross_min = min(right) + max(left)
    return min(left_min, right_min, cross_min)
```

1.1.4 Algorithm 4

```
algorithm4 (num_list):
    half = len(num_list)
    half = half / 2
    first_half = num_list[: half]
    second_half = num_list[half: ]
    min_left = algorithm34_helper(first_half)
    min_right = algorithm34_helper(second_half)
    second_half.reverse()
    algorithm2 (first_half, second_half)
```

1.2 Asymptotic Analysis

1.2.1 Algorithm 1

1 for loop ran twice n + n = 2n = O(n)

1.2.2 Algorithm 2

sort log n
sort log n = 2 log n = O (n + log n)

1.2.3 Algorithm 3

n log n x 2 = 2 O(n log n) + O(n)

1.2.4 Algorithm 4

O(n log n) + o(n)

2 Proofs of Correctness

2.0.5 Algorithm 3

```
If we assume that Algorithm 1 is correct. And assume that Algorithm 2 is not correct than for all values

of x algorithm 1 y doesn't equal algorithm 2's y.

Given Array1 [6, -7, -6] and Array2 [10, -2, 15]

Array1 sums [-7, -13, -6] Array2 sums [23, 13, 15]
```

Algorithm 1 would return -13 as the min of Array1 and find 13 of Array2 the Sum Closest to Zero = 0

Algorithm 2 would combine and sort the arrays into NewArray [-13, -7, -6, 13, 15, 23] then compares

NewArray[0] + NewArray [Array1Size+1] which would return the Sum 0. Algorithm 1 and 2 have

returned the same result which is a contradiction to the condition of Algorithm 2 being not correct thu

it must be correct.

2.0.6 Algorithm 4

Base Case: Array A of length = 1, that is the closest to zero value.

Inductive Hypothesis: For size array N the subarray of size k such that 0 < k < n is contained entirely

the first half, second half, or a combination of the suffix of the first half and a prefix of the secon

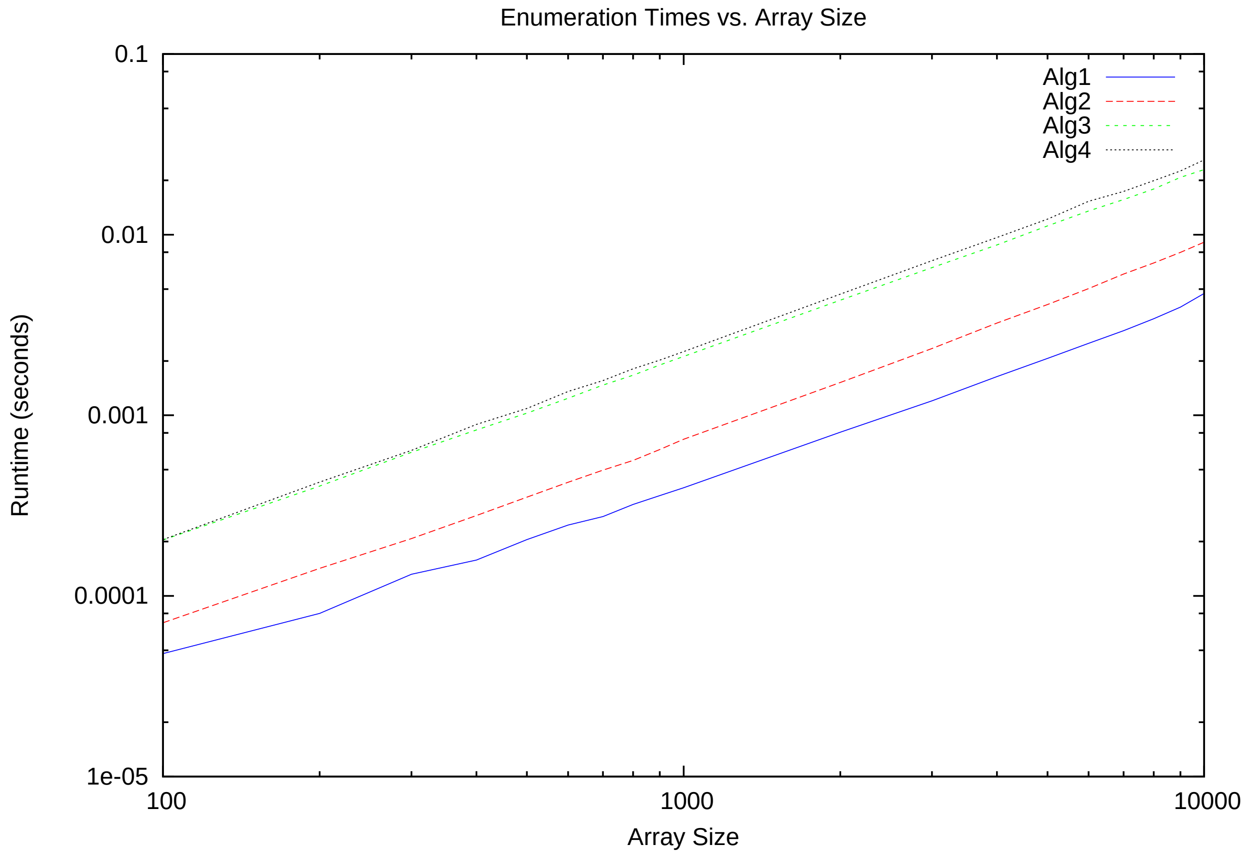
Inductive Step:

By the Inductive Hypothesis if array of size n+1 there will still be a subarray of size k such that k <

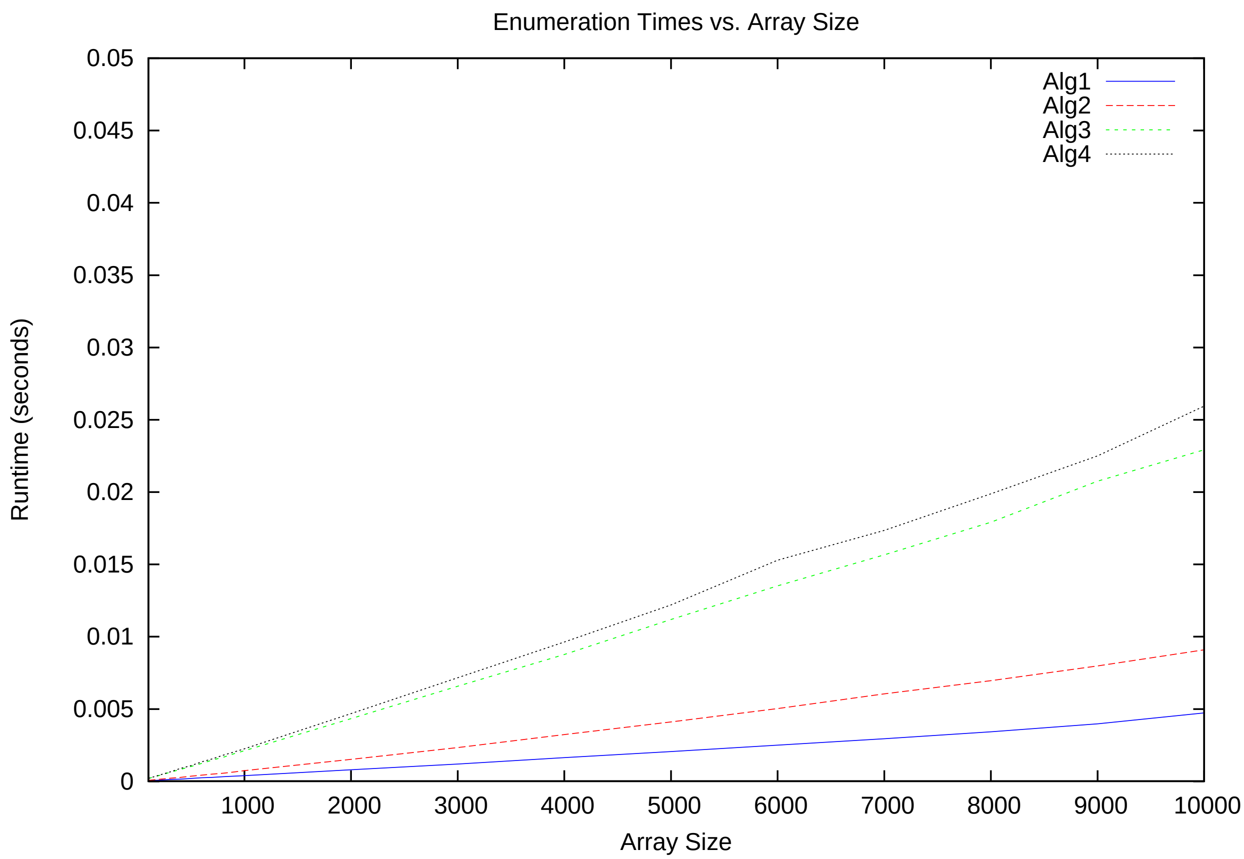
k+1 is still less than or equal to n, which is still a subarray.

3 Experimental Analysis

Size	Alg1	Alg2	Alg3	Alg4
100	0.000048	0.000071	0.000204	0.000205
200	0.000080	0.000142	0.000406	0.000427
300	0.000132	0.000208	0.000624	0.000640
400	0.000158	0.000279	0.000829	0.000891
500	0.000205	0.000352	0.001029	0.001090
600	0.000247	0.000426	0.001244	0.001356
700	0.000275	0.000497	0.001471	0.001556
800	0.000321	0.000562	0.001669	0.001808
900	0.000359	0.000646	0.001898	0.002022
1000	0.000397	0.000736	0.002118	0.002253
2000	0.000806	0.001521	0.004337	0.004686
3000	0.001202	0.002341	0.006573	0.007170
4000	0.001639	0.003239	0.008775	0.009639
5000	0.002064	0.004109	0.011191	0.012194
6000	0.002505	0.005031	0.013523	0.015289
7000	0.002942	0.006053	0.015657	0.017358
8000	0.003422	0.006966	0.017911	0.019886
9000	0.003974	0.007976	0.020761	0.022506
10000	0.004731	0.009098	0.022931	0.025944



Enumeration Times vs. Array Size
Runtime (seconds)
Array Size
Alg1
Alg2
Alg3
Alg4
0.1
0.01
0.001
0.0001
1e-05
100
1000
10000



Enumeration Times vs. Array Size
Runtime (seconds)
Array Size
Alg1
Alg2
Alg3
Alg4
0.05
0.045
0.04
0.035
0.03
0.025
0.02
0.015
0.01
0.005
0
1000
2000
3000
4000
5000
6000
7000
8000
9000
10000

4 Extrapolation and interpretation

4.1 Algorithms 3 and 4, largest dataset in 1 hour?

4.1.1 Algorithm 3

Using 2 data points from the timing results in algorithm 3 (0.000204s, 100 and 0.022931s, 10000) and plugging them into the equation:

$$Slope = m = \frac{log(\frac{F_1}{F_0})}{log(\frac{x_1}{x_0})} \quad (1)$$

resuls in $m = 1.0254$ as a log slope for the runtime of Algorithm 4. Using this value and the previous equation(2), we plug in 1 know data point (0.000204 seconds for an array of size 100) and the known time in 1 hour and solve for the array size.

$$1.0254 = \frac{log(\frac{3600}{0.000204})}{log(\frac{x}{100})}$$

and we get that $x = 1,167,260,000$.

*used `http://en.wikipedia.org/wiki/Log-log_plot` for help understanding log slope.

4.1.2 Algorithm 4

Using 2 data points from the timing results in algorithm 4 (0.000205s, 100 and 0.025944s, 10000) and plugging them into the equation:

$$Slope = m = \frac{log(\frac{F_1}{F_0})}{log(\frac{x_1}{x_0})} \quad (2)$$

resuls in $m = 1.0254$ as a log slope for the runtime of Algorithm 4. Using this value and the previous equation(2), we plug in 1 know data point (0.000204 seconds for an array of size 100) and the known time in 1 hour and solve for the array size.

$$1.04738 = \frac{log(\frac{3600}{0.000205})}{log(\frac{x}{100})}$$

and we get that $x = 825,713,000$.

*used `http://en.wikipedia.org/wiki/Log-log_plot` for help understanding log slope.

4.2 Experimental runtime and discrepancies

4.2.1 Algorithm 3

Our asymptotic analysis said that algorithm 3 should be at least $O(nlogn)$, but the runtime graph showed a graph that represented a more $C_1 * n + C_2$ runtime. This could be due to how we implemented the algorithm as well as optimizations taken by the python interpreter.

4.2.2 Algorithm 4

As with Algorithm 3, our asymptotic analysis said that Algorithm 4 should be dominated by by $O(nlogn)$ but the runtime graph represented a more linear runtime ($C_1 * n + C_2$). We very well could have done the algorithm incorrectly or the python interpreter could have introduced some optimizations that are unknown to us.