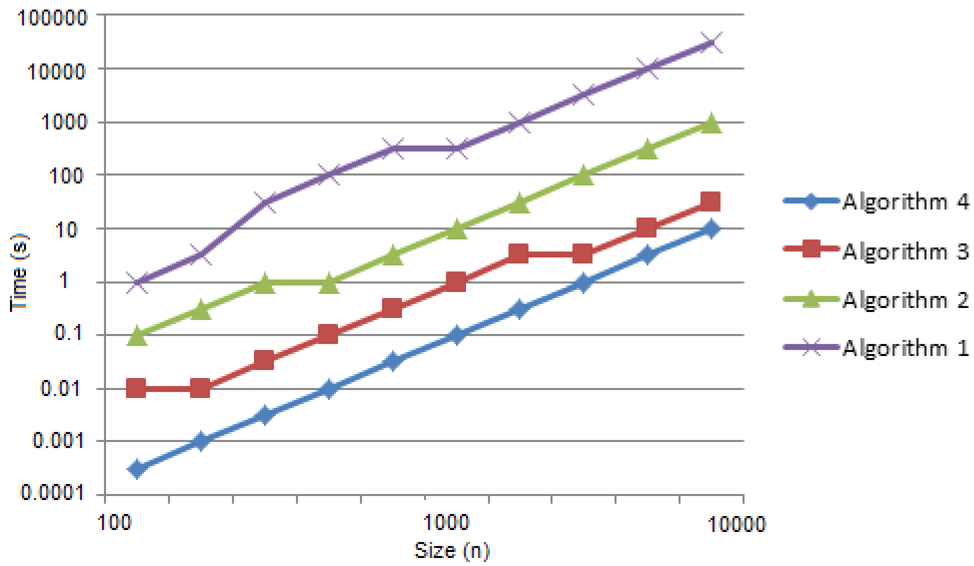
CS 325: PROJECT 1

GROUP 29: JACOB MASTEL, YASH NAIK, CERA OLSON

Date: 26 April 2015.



[H]

FIGURE 1. Average Run Time for All 4 Algorithms

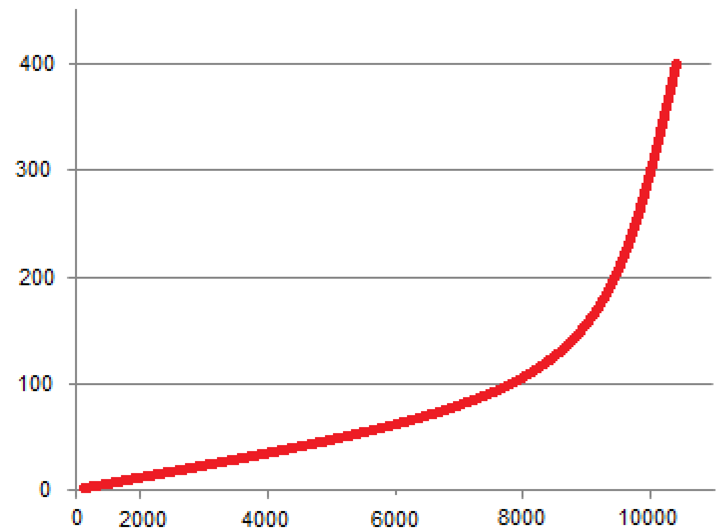
1. ALGORITHM 1: ENUMERATION

1.1. **Theoretical Run-Time Analysis.** Enumeration has a running time of $O(n^3)$. Because of the 3 loops, each run n times gives the algorithm a running time of $n * n * n$.

1.1.1. *Pseudocode.*

```
for i = 1 to n
for j = i to n
for k = i to j
sum += a_k
if  sum > ans then ans = sum
return ans
```

1.2. **Experimental Analysis.**



[H]

FIGURE 2. Algorithm 1 Run Time

n	Time
100	0.029255198
200	0.178765495
300	0.541698859
400	1.337829466
500	2.473190885
600	4.287970855
700	7.054050103
800	10.06102867
900	15.23538047
1000	20.64124558
1100	26.77310262
1200	31.31654893
1300	39.63601268
1400	49.22060823
1500	60.55456953
1600	73.94340321
1700	84.80297324
1800	99.51908661
1900	117.1002546
2000	136.9574863

1.3. **Extrapolation and Interpretation.**

1.3.1. *For each algorithm use the experimental data to estimate a function that models the relationship between running times and input sizes (n). Discuss any discrepancies between the experimental and theoretical running times.* In an $O(n^3)$ algorithm, as n doubles the running times increases 8 fold. This means the running time increases significantly faster than the other versions of this algorithm. Right around n=9000, the algorithm begins to increase even faster than it was already. Of the 4 algorithms, it has the worst efficiency, and the experimental data supports that.

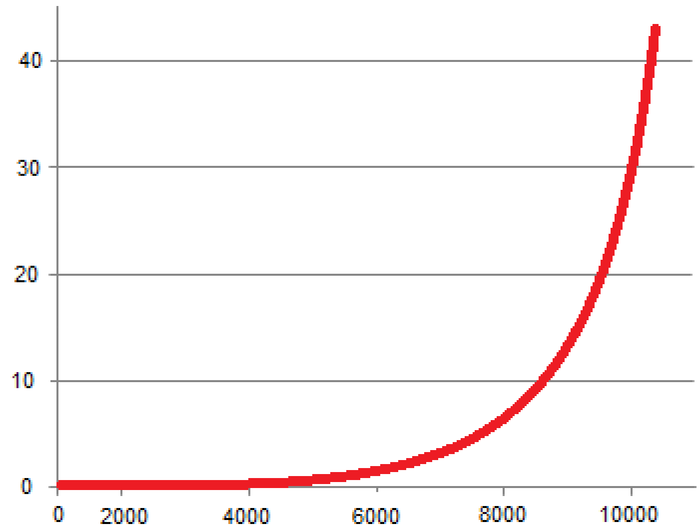
1.3.2. *For each algorithm, what is the size of the biggest instance that you can solve with your algorithm in one hour?* When n doubles, the running time increases eightfold for an $O(n^3)$ algorithm. In one hour this algorithm can run approximately 4,000 values. $(1,000 * 2^2)$.

2. Algorithm 2: Better Enumeration

2.1. **Theoretical Run-Time Analysis.** Because of the reduced number of loops, 2 loops running n times, the running time id $O(n^2)$. This algorithm runs very similarly to the first Algorithm, but it instead skips recalculating the sum each time through. This saves time and makes for a more efficient running time.

2.1.1. *Pseudocode.*

```
for i = 1 to n
for j = i to n
sum += a_j
if  sum > ans then ans = sum
return ans
```



[H]

Figure 3. Algorithm 2 Run Time

2.2. **Experimental Analysis.**

n	Time
100	0.000691457
200	0.002523139
300	0.005414918
400	0.009347594
500	0.017913874
600	0.032127009
700	0.03163984
800	0.037066278
900	0.076525134
1000	0.058382396
2000	0.277541148
3000	0.756019462
4000	1.17740895
5000	1.81821447
6000	2.809200957
7000	3.774356505
8000	4.120086395
9000	5.538109223
10000	6.983674863
20000	28.4940567
30000	67.27082063

2.3. **Extrapolation and Interpretation.**

2.3.1. *For each algorithm use the experimental data to estimate a function that models the relationship between running times and input sizes (n). Discuss any discrepancies between the experimental and theoretical running times.* Using what we know about the algorithm and the experimental data, the data above supports the running time of $O(n^2)$. If you walk through the data, the running times increase by approximately a factor of 4.

2.3.2. *For each algorithm, what is the size of the biggest instance that you can solve with your algorithm in one hour?* When n doubles, the running time increases fourfold for an $O(n^2)$ algorithm. In one hour this algorithm can run approximately 320,000 values. $(10,000 * 2^5)$.

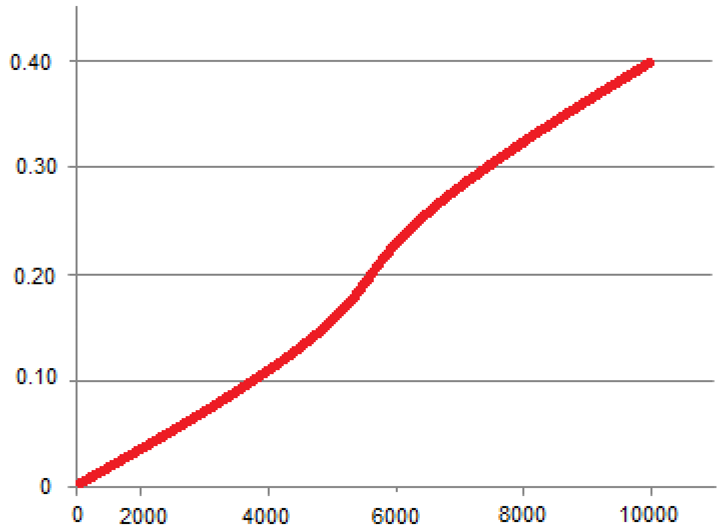
3. Algorithm 3: Divide and Conquer

3.1. **Theoretical Run-Time Analysis.** The original problem size is a power of 2, so all sub sizes are integers. So the run time method of the program will be $O(nlgn)$. $T(n) = 2T(n/2) + O(n)$

```
def algorithm3(A):
        if len(A) < 1:
                return 0

        m = len(A) / 2

        lmax = s = 0
        for i in range(len(A)/2, -1, -1):
                s += A[i]
                if s > lmax:
                        lmax = s

        rmax = s = 0
        for i in range(len(A)/2+1, len(A)):
                s += A[i]
                if s > rmax:
                        rmax = s

        return max(
                algorithm3(A[:len(A)/2]),
                algorithm3(A[(len(A)/2)+1:]),
                lmax + rmax
        )
```

3.2. **Proof of Correctness.** MaxSubarray(A, n) will return the sum of the maximum subarray of an array A of size n.

Proof: Base Case:- If n = 0. Then max = current = 0. Then subarray wont be possible and the max will be zero for array of zero elements. Case n=1: If n=1 meaning that there is one single element then if the value is positive it will be considered as array and max value will be assigned with the value of the array element. Inductive hypothesis:-



[H]

FIGURE 4. Algorithm 3 Run Time

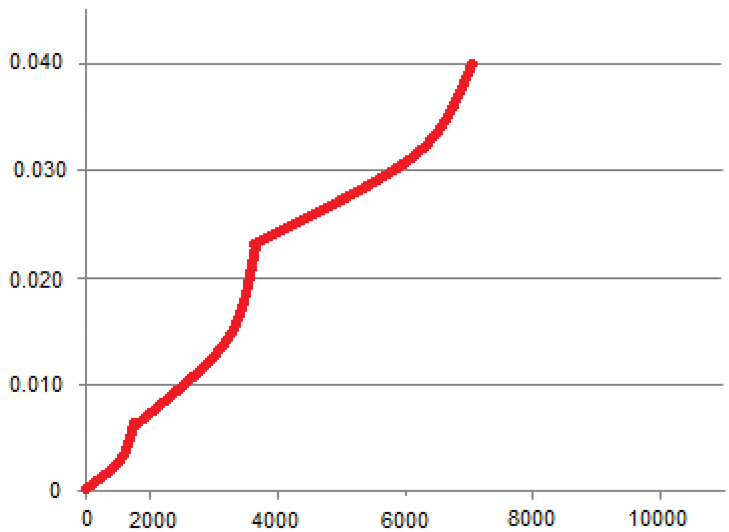
Considering array contain more elements say n. The method will find the sum of the whole array and save it in a variable max. and then it will divide the array in such a way that all the possible combinations of the array will be find and computed such that every loop execution the sum is compared with the max value if sum is greater then the value will be stored. So while n¿1 (meaning that more than one element in the array) the array will recursively divided and computes sum for all possible subarrays and compares with the max which will ultimately considered as the maximum value of a subarray.

3.3. **Experimental Analysis.**

n	Time
100	0.000513281
1000	0.004811525
2000	0.010599179
3000	0.015096591
4000	0.020326933
5000	0.041936171
6000	0.039872297
7000	0.047068976
8000	0.042948396
9000	0.050002995
10000	0.054033207
20000	0.127373954
30000	0.194761927
40000	0.254894853
50000	0.31652922
60000	0.411895206
70000	0.450825678
80000	0.538815784
90000	0.589621596
100000	0.64801116

3.4. **Extrapolation and Interpretation.**

3.4.1. *For each algorithm use the experimental data to estimate a function that models the relationship between running times and input sizes (n). Discuss any discrepancies between the experimental and theoretical running times.* While this graph was a bit less obvious, the general shape indicates a relationship similar to that of the linear running time. The data supports the theoretical running time of O(n log n) but just more than doubling each time n doubles.



[H]

FIGURE 5. Algorithm 4 Run Time

3.4.2. *For each algorithm, what is the size of the biggest instance that you can solve with your algorithm in one hour?* When n doubles, the running time slightly more than doubles for an $O(nlogn)$ algorithm. In one hour this algorithm can run approximately 409,600,000 values. $(100,000 * 2^12)$.

4. Algorithm 4: Linear-Time

4.1. **Theoretical Run-Time Analysis.** Because there is only 1 loop that runs through n times the Linear Time Algorithm runs the fastest with a $O(n)$ running time.

4.1.1. *Pseudocode.*

```
for j = 1 .. n
b = max{b + j, j}
     ans = max{b, ans}
return ans
```

4.2. **Experimental Analysis.**

n	Time
100	0.00014592
1000	0.000610561
2000	0.001227777
3000	0.00182733
4000	0.002412802
5000	0.002993667
6000	0.003615748
7000	0.004194308
8000	0.004854277
9000	0.005397254
10000	0.005966598
20000	0.011977484
30000	0.018128915
40000	0.034239523
50000	0.043462445
60000	0.037134118
70000	0.041755435
80000	0.061413183
90000	0.053687095
100000	0.061596479

4.3. **Extrapolation and Interpretation.**

4.3.1. *For each algorithm use the experimental data to estimate a function that models the relationship between running times and input sizes (n). Discuss any discrepancies between the experimental and theoretical running times.* While the long term graph is a bit rough, the data supports a linear running time. Tracing through the data, the values follow the trend of doubling as n double. The exact data follows this trend in the long run, despite the few bumps.

4.3.2. *For each algorithm, what is the size of the biggest instance that you can solve with your algorithm in one hour?* When n doubles, the running time doubles for an $O(n)$ algorithm. In one hour this algorithm can run approximately 3,276,800,000 values. $(100,000 * 2^15)$.

5. Appendices

5.1. Code.

```
def algorithm1(A):
	max = 0
	for i in range(0, len(A)):
		for j in range(i, len(A)):
			partial = 0
			for k in range(i, j+1):
				partial += A[k]

			if partial > max:
				max = partial

	return max


def algorithm2(A):
	max = 0
	for i in range(0, len(A)):
		sum = 0
		for j in range(i, len(A)):
			sum = sum + A[j]
			if max <= sum:
				max = sum

	return max


def algorithm3(A):
	if len(A) < 1:
		return 0

	m = len(A) / 2

	lmax = s = 0
	for i in range(len(A)/2, -1, -1):
		s += A[i]
		if s > lmax:
			lmax = s
```

```
    rmax = s = 0
    for i in range(len(A)/2+1, len(A)):
        s += A[i]
        if s > rmax:
            rmax = s

    return max(
        algorithm3(A[:len(A)/2]),
        algorithm3(A[(len(A)/2)+1:]),
        lmax + rmax
    )


def algorithm4(A):
    m1 = m2 = 0

    for x in A:
        m1 = max(0, m1 + x)
        m2 = max(m2, m1)

    return m2



# Algorithm Tests

# Needed Python libraries
import csv
import sys
import random
from timeit import Timer
from multiprocessing import Process

# Import our algorithms
from algorithm1 import *
from algorithm2 import *
from algorithm3 import *
from algorithm4 import *

# Global Variables
max_time = 2*60 # 2 minutes
```

```
min_num = -99
max_num = 99

def run_test(Alg):
    f_name = "alg_res{0}.csv".format(Alg)
    with open(f_name, 'wb') as csvfile:
        writer = csv.writer(csvfile)

        for n in range(100, 100001, 100):
            # build a random array of len n
            A = []
            for _ in range(n):
                A.append(random.randint(min_num, max_num))

            # determine which algorithm to call
            # run each set 3 times
            if Alg == 1:
                t = Timer(lambda: algorithm1(A)).timeit(number=
            elif Alg == 2:
                t = Timer(lambda: algorithm2(A)).timeit(number=
            elif Alg == 3:
                t = Timer(lambda: algorithm3(A)).timeit(number=
            elif Alg == 4:
                t = Timer(lambda: algorithm4(A)).timeit(number=

            writer.writerow([n, t])

            # see if we've gone beyond our max time.
            # if we have, break the loop
            if t >= max_time:
                break;

        print 'Algorithm {0} finished'.format(Alg)

def random_tests():
    jobs = []
    for i in range(1,5):
        p = Process(target=run_test, args=(i,))
        jobs.append(p)
        p.start()
```

```
        for p in jobs:
                p.join()

def print_fail(alg, a, expected, returned):
        print "Algorithm {0} failed test {1}".format(alg, a)
        print "  Expected: {0}".format(expected)
        print "  Returned: {0}".format(returned)

# Test each of the algorithms against known arrays and their answers
def validate_algorithms():
        a = {}
        a["a1"] = [1, 4, -9, 8, 1, 3, 3, 1, -1, -4, -6, 2, 8, 19, -10, -11]
        a["a1_sub"] = [8, 1, 3, 3, 1, -1, -4, -6, 2, 8, 19]
        a["a1_ans"] = 34

        a["a2"] = [2, 9, 8, 6, 5, -11, 9, -11, 7, 5, -1, -8, -3, 7 -2]
        a["a2_sub"] = [2, 9, 8, 6, 5]
        a["a2_ans"] = 30

        a["a3"] = [10, -11, -1, -9, 33, -45, 23, 24, -1, -7 -8, 19]
        a["a3_sub"] = [23, 24, -1, -7, -8, 19]
        a["a3_ans"] = 50

        a["a4"] = [31,-41, 59, 26, -53, 58, 97, -93, -23, 84]
        a["a4_sub"] = [59, 26, -53, 58, 97]
        a["a4_ans"] = 187

        a["a5"] = [3, 2, 1, 1, -8, 1, 1, 2, 3]
        a["a5_sub"] = [3, 2, 1, 1]
        a["a5_ans"] = 7

        a["a6"] = [12, 99, 99, -99, -27, 0, 0, 0, -3,10]
        a["a6_sub"] = [12, 99, 99]
        a["a6_ans"] = 210

        a["a7"] = [-2, 1, -3, 4, -1, 2, 1, -5, 4]
        a["a7_sub"] = [4, -1, 2, 1]
        a["a7_ans"] = 6
```

```
    a["a8"] = [-1, -3 -5]
    a["a8_sub"] = []
    a["a8_ans"] = 0

    all_passed = True
    for i in range(1,9):
        a1 = algorithm1(a["a{0}".format(i)])
        a2 = algorithm2(a["a{0}".format(i)])
        a3 = algorithm3(a["a{0}".format(i)])
        a4 = algorithm4(a["a{0}".format(i)])

        if  a1 != a["a{0}_ans".format(i)]:
            print_fail(1, i, a["a{0}_ans".format(i)], a1)
            all_passed = False

        if  a2 != a["a{0}_ans".format(i)]:
            print_fail(2, i, a["a{0}_ans".format(i)], a2)
            all_passed = False

        if  a3 != a["a{0}_ans".format(i)]:
            print_fail(3, i, a["a{0}_ans".format(i)], a3)
            all_passed = False

        if  a4 != a["a{0}_ans".format(i)]:
            print_fail(4, i, a["a{0}_ans".format(i)], a4)
            all_passed = False

    if all_passed == True:
        print 'All tests passed! :)'

def read_input():
    A = []
    with open('MSS_Problems.txt', 'r') as f:
        for line in f:
            arr = line.replace('[','').replace(']','').replace(' ',
            arr = arr.split(',')
            arr = [int(a) for a in arr]
            A.append(arr)

    return A
```

```
def MSS_test():
        A = read_input()

        with open('MSS_Results.txt', 'w') as f:
                for cnt, arr in enumerate(A):
                        f.write('Array {0}:\n'.format(cnt))
                        f.write('    Algorithm 1: {0}\n'.format(algorithm1(arr))
                        f.write('    Algorithm 2: {0}\n'.format(algorithm2(arr))
                        f.write('    Algorithm 3: {0}\n'.format(algorithm3(arr))
                        f.write('    Algorithm 4: {0}\n'.format(algorithm4(arr))

def print_help():
        print "Program argument error!"
        print "  Valid arguments include time_test, alg_test, or MSS_test"

if __name__ == "__main__":
        if len(sys.argv) < 2 :
                print_help()
                sys.exit()

        if sys.argv[1] == "time_test":
                random_tests()
        elif sys.argv[1] == "alg_test":
                validate_algorithms()
        elif sys.argv[1] == "MSS_test":
                MSS_test()
        else:
                print_help()
```