CS325 Project 1

Aaron Fifer

Spencer Moran

Seth Weston

Group 32

1. Theoretical Run Time Analysis
2. **BruteForceMethod**

for (int i = 0; i <= size-2; i++){

sum = 0;

for (int j = i; j <= size - 1; j++){

sum = sum + a[j];

if (sum > ret){

MaxSumStart = i;

MaxSumEnd = j;

ret = sum;

}}}}

The brute force method tests every possible sub array against the current best sum, returning the best sum at completion. Thus, the method makes operations times. .

1. **ImprovedBruteForceMethod**

int ret = a[0];

int MaxSumStart = 0, MaxSumEnd = 0;

for( int i = 1; i < size; i++){

int sum = 0;

int j = i;

int tempStart = MaxSumStart;

while(j>=tempStart){

sum += a[j];

if ( sum > ret ){

ret = sum;

MaxSumStart = j;

MaxSumEnd = i;

}

j--;

}}}

The improved brute force method makes use of memorization to enable calculation of previously solved pairs in constant time. This removes a number of redundant calls, improving run time. .

1. **Divide and Conquer**

maxSumTuple maxCrossing (int values[], int left, int right, int middle) {

int leftSum = 0;

int sum = 0;

maxSumTuple max;

for (int i = middle; i >= left; i--) {

sum += values[i];

if (sum > leftSum) {

leftSum = sum;

max.left = i;

}}

int rightSum = 0;

sum = 0;

for (int i = middle + 1; i <= right; i++) {

sum += values[i];

if (sum > rightSum) {

rightSum = sum;

max.right = i;

}}

max.sum = leftSum + rightSum;

return max;

}

maxSumTuple divideConquer(int values[], int left, int right) {

if (left == right) {

maxSumTuple max;

max.sum = values[left];

max.left = left;

max.right = right;

return max;

}

int middle = (left + right) / 2;

maxSumTuple leftSum = divideConquer(values, left, middle);

maxSumTuple rightSum = divideConquer(values, middle + 1, right);

maxSumTuple crossing = maxCrossing(values, left, right, middle);

if (leftSum.sum >= rightSum.sum && leftSum.sum >= crossing.sum) {

return leftSum;

} else if (rightSum.sum >= leftSum.sum && rightSum.sum >= crossing.sum) {

return rightSum;

} else {

return crossing;

}

}

Implementation of this algorithm required creation of the maxSumTuple struct, which tracked the maximum sum and indices for the maximum subarray. The algorithm evaluates the left and right sides, and then compares these sides against a cross of the sides and selects the largest sum for the final value. The algorithm must run in at least time since every element must be evaluated, and the calculations require an additional time. .

1. **Linear Time DP**

std::ofstream output;

output.open("results", std::ios::app);

int maxInitIndex = 0; //starting index for the max sub array sum

int maxEndIndex = 0; //last index for the max sub array sum

int curInitIndex = 0; //current initial index for sub array whose sum is being computed

int maxSum = 0;

int curSum = 0;

for (int i = 0; i < valuesLen; i++) {

curSum = curSum + values[i];

if (curSum > maxSum) {

maxSum = curSum;

maxInitIndex = curInitIndex;

maxEndIndex = i;

} else if (curSum <= 0) {

//if all the values are negative, curSum will be set

//to the least negative value

if (values[i] > maxSum) {

maxSum = values[i];

}

//set curSum to 0 and curInitIndex to i + 1 to start

//computing the sum of a new sub array

curSum = 0;

curInitIndex = i + 1;

}}}

The linear time algorithm works by parsing through the array and adding each element to the sum of the last element. It then checks this sum against the maximum sum and if it exceeds it, marks the index and moves on. If the current sum dips below zero, then the sum is reset, and the algorithm begins working on the next positive sub array. In the case where all values are negative, the max will be the least negative value. This continues until the entire array has been processed. Each of these operations takes place in constant time, making the algorithm run in linear time.

1. Testing
2. Experimental Analysis

This section lists the data collected from the timing analysis of each algorithm. All run times are listed in seconds. The line best fit equation is presented in each of the graphs of the data using regression functions in excel.

1. Brute Force

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  | Brute Force | |  |  |  |  |
| n=100 | n=200 | n=500 | n = 1000 | n=2000 | n=5000 | n=10000 | n=20000 | n=50000 | n=1000000 |
| 2.00E-05 | 5.30E-05 | 0.000314 | 0.001489 | 0.005105 | 0.032017 | 0.133363 | 0.531272 | 3.3681 | 13.2832 |
| 1.80E-05 | 6.20E-05 | 0.000388 | 0.001518 | 0.005294 | 0.03276 | 0.132925 | 0.541575 | 3.32885 | 13.2687 |
| 1.70E-05 | 6.30E-05 | 0.000395 | 0.001452 | 0.004989 | 0.032542 | 0.130558 | 0.534111 | 3.31621 | 13.3264 |
| 1.60E-05 | 6.20E-05 | 0.000352 | 0.001343 | 0.00491 | 0.034062 | 0.131184 | 0.519187 | 3.29927 | 13.3021 |
| 1.60E-05 | 6.90E-05 | 0.000412 | 0.00137 | 0.005179 | 0.035344 | 0.134863 | 0.52589 | 3.33833 | 13.5319 |
| 1.70E-05 | 6.70E-05 | 0.00035 | 0.001399 | 0.005358 | 0.033945 | 0.140684 | 0.526301 | 3.34994 | 13.3201 |
| 1.80E-05 | 6.40E-05 | 0.000423 | 0.001486 | 0.005322 | 0.032465 | 0.141604 | 0.529129 | 3.30668 | 13.3379 |
| 1.80E-05 | 6.20E-05 | 0.00039 | 0.001374 | 0.005287 | 0.032296 | 0.138743 | 0.529723 | 3.30983 | 13.3526 |
| 1.60E-05 | 6.70E-05 | 0.000382 | 0.001407 | 0.005694 | 0.034539 | 0.132262 | 0.522947 | 3.36832 | 13.3437 |
| 2.60E-05 | 6.20E-05 | 0.000373 | 0.001487 | 0.004973 | 0.035327 | 0.133398 | 0.53646 | 3.33432 | 13.51 |
| Avg | Avg | Avg | Avg | Avg | Avg | Avg | Avg | Avg | Avg |
| **1.82E-05** | **6.31E-05** | **0.000378** | **0.001433** | **0.005211** | **0.03353** | **0.134958** | **0.52966** | **3.331985** | **13.35766** |

Table 1: Brute Force Timing Data

Figure 1: Brute Force Run Time vs Number of Elements

Using regression equation

We find the number of elements that the algorithm can work for in 5, 10, and 60 seconds.

Figure 2: Brute Force Run Time vs Number of Elements log log plot

1. Improved Brute Force

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  | Improved Brute Force | | |  |  |  |
| n=100 | n=200 | n=500 | n = 1000 | n=2000 | n=5000 | n=10000 | n=20000 | n=50000 | n=1000000 |
| 8.00E-06 | 4.00E-05 | 0.000137 | 0.000618 | 0.005182 | 0.009162 | 0.080605 | 0.295142 | 1.42131 | 9.95867 |
| 1.40E-05 | 2.70E-05 | 0.000151 | 0.001058 | 0.004064 | 0.024849 | 0.076248 | 0.223009 | 0.965965 | 6.85034 |
| 8.00E-06 | 2.20E-05 | 0.000205 | 0.000781 | 0.002704 | 0.021468 | 0.044611 | 0.279406 | 1.25897 | 4.72566 |
| 8.00E-06 | 4.20E-05 | 0.000187 | 0.000828 | 0.002633 | 0.022205 | 0.066709 | 0.322016 | 2.03356 | 10.2935 |
| 1.20E-05 | 4.10E-05 | 0.000148 | 0.000642 | 0.00163 | 0.012647 | 0.118127 | 0.253288 | 2.08973 | 6.83575 |
| 1.10E-05 | 4.30E-05 | 0.000235 | 0.000802 | 0.002878 | 0.013625 | 0.075705 | 0.280478 | 1.26384 | 8.04637 |
| 1.30E-05 | 3.30E-05 | 0.00019 | 0.000637 | 0.002046 | 0.01635 | 0.038793 | 0.173169 | 2.56585 | 6.93253 |
| 1.20E-05 | 4.20E-05 | 0.000226 | 0.001159 | 0.002563 | 0.011407 | 0.101194 | 0.304477 | 1.95178 | 4.94376 |
| 6.00E-06 | 4.50E-05 | 0.000282 | 0.000927 | 0.003061 | 0.014643 | 0.072252 | 0.134293 | 1.58763 | 5.38412 |
| 7.00E-06 | 4.20E-05 | 0.000225 | 0.000864 | 0.003305 | 0.01084 | 0.066611 | 0.183326 | 1.2177 | 13.6168 |
| Avg | Avg | Avg | Avg | Avg | Avg | Avg | Avg | Avg | Avg |
| **9.90E-06** | **3.77E-05** | **0.000199** | **0.000832** | **0.003007** | **0.01572** | **0.074086** | **0.24486** | **1.635634** | **7.75875** |

Table 2: Improved Brute Force Timing Data

Figure 3: Improved Brute Force Run Time vs Number of Elements

Using regression equation

We find the number of elements that the algorithm can work for in 5, 10, and 60 seconds.

Figure 4: Improved Brute Force Run Time vs Number of Elements log log plot

1. Divide and Conquer

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  | Divide and Conquer (3) | | |  |  |  |
| n=1000 | n=2000 | n=5000 | n=10000 | n=20000 | n=50000 | n=100000 | n=200000 | n=500000 | n=1000000 |
| 7.70E-05 | 0.000174 | 0.000444 | 0.000795 | 0.001736 | 0.005128 | 0.008957 | 0.02083 | 0.05074 | 0.108285 |
| 8.40E-05 | 0.000169 | 0.00041 | 0.000853 | 0.001693 | 0.004969 | 0.009459 | 0.021699 | 0.05016 | 0.105968 |
| 8.40E-05 | 0.000183 | 0.000382 | 0.000821 | 0.00194 | 0.0045 | 0.008918 | 0.022058 | 0.050968 | 0.104339 |
| 7.00E-05 | 0.000188 | 0.000396 | 0.000837 | 0.001907 | 0.005288 | 0.009003 | 0.021638 | 0.049365 | 0.101969 |
| 7.10E-05 | 0.000198 | 0.000405 | 0.000825 | 0.001909 | 0.004996 | 0.008989 | 0.020692 | 0.049536 | 0.10073 |
| 7.00E-05 | 0.000186 | 0.000381 | 0.000848 | 0.001737 | 0.00451 | 0.009206 | 0.020485 | 0.049613 | 0.101952 |
| 7.00E-05 | 0.000171 | 0.000384 | 0.000809 | 0.001632 | 0.005171 | 0.009348 | 0.019843 | 0.049926 | 0.10174 |
| 7.50E-05 | 0.000176 | 0.000383 | 0.000822 | 0.001731 | 0.004296 | 0.009241 | 0.018716 | 0.04927 | 0.100748 |
| 7.00E-05 | 0.000189 | 0.000384 | 0.000842 | 0.00169 | 0.005246 | 0.009179 | 0.019295 | 0.049569 | 0.100396 |
| 7.00E-05 | 0.000187 | 0.000418 | 0.000887 | 0.001627 | 0.005183 | 0.008907 | 0.01899 | 0.048761 | 0.103655 |
| Avg | Avg | Avg | Avg | Avg | Avg | Avg | Avg | Avg | Avg |
| **7.41E-05** | **0.000182** | **0.000399** | **0.000834** | **0.00176** | **0.004929** | **0.009121** | **0.020425** | **0.049791** | **0.1029782** |

Table 3: Divide and Conquer Timing Data

Figure 5: Divide and Conquer Run Time vs Number of Elements

Using regression equation

We find the number of elements that the algorithm can work for in 5, 10, and 60 seconds.

Figure 6: Divide and Conquer Run Time vs Number of Elements log log plot

1. Dynamic Programming

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  | Dynamic Programming (4) | | |  |  |  |
| n=1000 | n=2000 | n=5000 | n=10000 | n=20000 | n=50000 | n=100000 | n=200000 | n=500000 | n=1000000 |
| 8.00E-06 | 1.30E-05 | 4.60E-05 | 7.20E-05 | 0.000123 | 0.000385 | 0.000659 | 0.001225 | 0.003392 | 0.006541 |
| 1.00E-05 | 1.60E-05 | 3.80E-05 | 7.70E-05 | 0.000123 | 0.000308 | 0.000697 | 0.001234 | 0.003182 | 0.006525 |
| 1.40E-05 | 2.10E-05 | 3.40E-05 | 6.20E-05 | 0.000163 | 0.000352 | 0.000613 | 0.001277 | 0.003115 | 0.006407 |
| 8.00E-06 | 1.50E-05 | 3.20E-05 | 6.30E-05 | 0.000164 | 0.000455 | 0.000613 | 0.001261 | 0.004152 | 0.006605 |
| 9.00E-06 | 1.30E-05 | 3.20E-05 | 6.20E-05 | 0.000133 | 0.000381 | 0.000613 | 0.00126 | 0.003939 | 0.007233 |
| 7.00E-06 | 1.40E-05 | 3.20E-05 | 6.20E-05 | 0.000135 | 0.000416 | 0.001192 | 0.001251 | 0.004309 | 0.007457 |
| 7.00E-06 | 1.40E-05 | 3.10E-05 | 6.30E-05 | 0.000122 | 0.000417 | 0.00076 | 0.001272 | 0.003161 | 0.006336 |
| 7.00E-06 | 1.20E-05 | 3.10E-05 | 6.30E-05 | 0.000123 | 0.000426 | 0.000705 | 0.001284 | 0.003342 | 0.007711 |
| 6.00E-06 | 1.20E-05 | 3.10E-05 | 7.40E-05 | 0.000171 | 0.0004 | 0.000756 | 0.001258 | 0.003105 | 0.006685 |
| 7.00E-06 | 1.20E-05 | 3.10E-05 | 6.10E-05 | 0.000122 | 0.000407 | 0.000801 | 0.00141 | 0.003262 | 0.00639 |
| Avg | Avg | Avg | Avg | Avg | Avg | Avg | Avg | Avg | Avg |
| **8.30E-06** | **1.42E-05** | **3.38E-05** | **6.59E-05** | **1.38E-04** | **3.95E-04** | **7.41E-04** | **1.27E-03** | **3.50E-03** | **6.79E-03** |

Table 4: Dynamic Programming Timing Data

Figure 7: Dynamic Programming Run Time vs Number of Elements

Using regression equation

We find the number of elements that the algorithm can work for in 5, 10, and 60 seconds.

Figure 8: Dynamic Programming Run Time vs Number of Elements log log plot