QUEENSLAND UNIVERSITY OF TECHNOLOGY

Empirical Analysis of an Algorithm

Mitchell Bourne

April 11, 2017

CONTENTS

1 SUMMARY OF ALGORITHM

The pseudo-code of the algorithm in reference is shown in figure 1.1.

```
ALGORITHM BruteForceMedian(A[0..n − 1])
  // Returns the median value in a given array A of n numbers. This is
  // the kth element, where k = ⌊n/2⌋, if the array was sorted.
  k ← ⌊n/2⌋
  for i in 0 to n − 1 do
    numsmaller ← 0   // How many elements are smaller than A[i]
    numequal ← 0     // How many elements are equal to A[i]
    for j in 0 to n − 1 do
      if A[j] < A[i] then
        numsmaller ← numsmaller + 1
      else
        if A[j] = A[i] then
          numequal ← numequal + 1
    if numsmaller < k and k ≤ (numsmaller + numequal) then
      return A[i]
```

Figure 1.1: Algorithm Pseudo-code

The algorithm is designed to return the median of an array of any given numbers denoted by length n. It does this by first finding k which is the midpoint of the arrays length. It then enters the first for loop in which determines the element number in which the next for loop will compare. From here, each element in the array is compared to the selected number of the first loop. Once all variables of the array have been compared to this number, if the amount of smaller value numbers found is less then k; and the amount of smaller value numbers + equal value numbers is less then or equal to k, the function will return the number the original for loop was comparing, or the median of the array. If the loop does not find the median on that test, the outside for loop will advance to the next number of the array, continuing till the median is found.

2 Theoretical Case Efficiency

2.1 Basic Operation

In order to calculate the efficiency of the algorithm we must first identify the 'basic operation' of the algorithm. The basic operation of an algorithm is defined as the process in which will have the most influence on the algorithms running time. In defining this, it is easy to determine the basic operation is that of the two 'if' array comparison statements. This is because the first 'less then' if statement will have to run every iteration of the algorithm, with the 'equal to' statement running when the first is found to be false; heavily influencing the total running time of the algorithm. The final comparison to k statement is disregarded due to its infrequency.

2.2 Best Case Efficiency

In terms of the best case , it is easy to visualize. The algorithm will locate the mean the fastest if it is the first element of the input array; therefore only one iteration of the outer for loop is required, performing the least amount of basic operations possible. The following best case equation is found.

Note: All calculations throughout this section are completed with the assumption the input array contains all unique numbers,in order to make calculation possible.

$$A_{Best} \approx \sum_{n=0}^{n-1} 2n - (\frac{n}{2}) + 1$$

$$2n - (\frac{n}{2}) + 1$$

$$\sigma n$$

The equation is calculated through that of the total count of basic operations, $2n$ which then has the total amount of times the equal loop is not reached, $\frac{n}{2}$ subtracted. With the $+1$ being included because of subtracted basic operation being inclusive of n and not $n-1$.

2.3 Worst Case Efficiency

In terms of worst case efficiency the longest the algorithm will take to find the median will be when it is the last element of the input array. In order to calculate this the sum of each basic operation comparison is calculated, with the amount of times the second comparison

is not reached summed and subtracted. The same assumptions are used and the following function is found.

$$A_{Worst} \approx \sum_{n=0}^{n-1} 2n^2 + (\sum_{n=0}^{n-1} \frac{n^2}{2} - \sum_{n=0}^{n-1} \frac{n}{2})$$

$$2n^2 - ((\frac{n^2}{2}) - \frac{n}{2}))$$

$$\frac{n}{2} + \frac{(3 * (n^2))}{n}$$

$$On^2$$

2.4 Average Case Efficiency

In order to calculate this, the mean average equation was used. It is important to note all possible basic operation counts lay between the best and worst case. Therefore a summation from the best to worst case, with an averaged total will calculate the mean. The following equation is found.

$$\sum_{c=2n-(\frac{n}{2})+1}^{\frac{n}{2}+\frac{(3*(n^2))}{n}} c/(\frac{n}{2} + \frac{(3 * (n^2))}{n}) - (2n - \frac{n}{2} + 1))$$

When the array contains mixed and equivalent numbers the algorithms efficiency will range from On^2 to σn.

3 METHODOLOGY

All tests were completed on a Lenovo Thinkpad Yoga 11e laptop. This was due to computation issues involving time on a quicker computer, with most of the timed outputs nearing close to 0 (creating no notable trend). With this, the data sets are chosen to prove two main points; that the algorithm's efficiency ranges closer to n the less unique the array and that no array will exceed the On^2 nor σn bounds.

In order to produce this data, multiple sets of arrays were created. All experimental data is generated from array sizes 20 to 4980 in steps of 20, running 10 times and averaging at each size; then is output to .txt documents for further extrapolation. This same method was used for all experiments, with further explanation of the data provided in the appropriate sections below.

4 Code Implementation

The following is the C++ implementation of the given pseudo-code.

```
double bruteForceMedian(int array[], double n){
    double k = ceil(n/2);
    for(int i = 0; i <= n-1; i++){
        double numsmaller = 0, numequal = 0;
        for(int j = 0; j <= n-1; j++){
            if(array[j] < array[i]){
                numsmaller++;
            }
            else if(array[j] == array[i]) {
                numequal++;
            }
        }
        if((numsmaller < k) && (k<= (numsmaller + numequal))){
            return array[i];
        }
    }
}
```

With n being calculated by the following:

```
double n = sizeof(array)/sizeof(array[0]);
```

As shown the algorithm in code form is relatively simple; with the only non pseudo-code addition being the calculation of n, which is not included in the algorithm itself in order to keep the algorithm as true to the pseudo-code as possible. The algorithm was tested using the following arrays (1, 4, 3, 5, 9, 7), (9, 3, 2, 5, 6, 6, 7) and (9, 5, 5, 9, 10, 7, 6, 8); with results outputs of 4, 6, and 7 respectively. As can be seen, on odd sized arrays the, the program rounds down the mean where applicable; in order to fix this the ceil() functions was added as requested. The data examples above indicate that the translated code is working correctly.

5 Experimental Basic Operation Count

One method of testing whether our predicted efficiency order of growth is correct is to compare it to that of the experimental basic operation order of growth in the program. This will be done by adding a counter to the relevant sections of the code. These are represented by the number commented code in the following.

```
double bruteForceMedian(int array[], double n){
    double k = ceil(n/2);
    int basic_op = 0; //1//
    for(int i = 0; i <= n-1; i++){
        double numsmaller = 0, numequal = 0;
        for(int j = 0; j <= n-1; j++){
            basic_op = basic_op + 2; //2//
            if(array[j] < array[i]){
                numsmaller++;
                basic_op = basic_op - 1; //3//
            }
            else if(array[j] == array[i]) {
                numequal++;
            }
        }
        if((numsmaller < k) && (k<= (numsmaller + numequal))){
            return basic_op; //4//
        }
    }
}
```

The first addition to the code (1) is the declaration of the basic operation counter; in which resets whenever the a new data set is entered. The next addition (2) adds the maximum amount of basic operations per loop to the counter, doing this every inner for loop. The line inside the if statement (3) removes the count of the 'else if' comparison; this is because if the first statement if found true, the else if statement is never run (subtracting a basic operation). The final addition (4) outputs the basic operation count rather then the mean.

5.1 TEST 1

The first test data was that of unique arrays; these were produced with the following code.

```
int A[5000];
std::ofstream op("output_unique.txt");
int op_count;
for(int i = 20; i < 5000; i = i + 20){
   for(int tests = 0; tests < 10; tests++){
      for(int j = 0; j < i; j++) {
         A[j] = j + 1;
      }
      op_count += bruteForceMedian(A, i);
   }
   cout << op_count;
   op << op_count/10 << "\n";
   op_count = 0;
}
```

This data was then graphed with the addition of its closest trendline, shown with the n^2 trendline in figure 5.1.

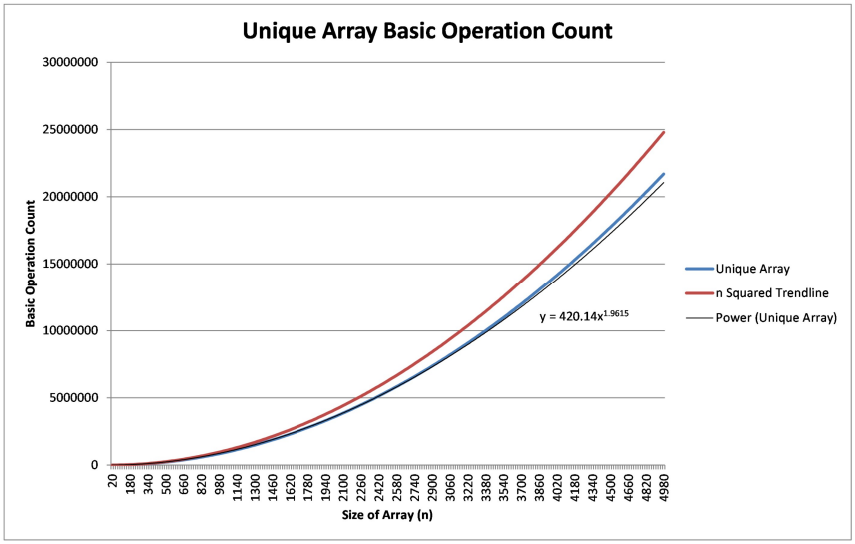

Figure 5.1: Unique Array Data Basic Operation Count

As seen above, the unique data set trendline is close but does not equal that of the upper limit. This is due to the mean always being located in the central array element, meaning no data is added including that of the best or worst case equation. Along with this, the trend line equation does not include the operations in which the number is checking itself;

which will affect the trend heavily at large array sizes, given reason to the larger discrepancy towards the larger array sizes.

5.2 TEST 2

This set of data is produced with randomly generated numbers from 0 to 5000 located anywhere in a given array.

```
int A[5000];
std::ofstream op("output_mixed.txt");
int op_count;
for(int i = 20; i < 5000; i = i + 20){
   for(int tests = 0; tests < 10; tests++){
      for(int j = 0; j < i; j++) {
         A[j] = rand() % 5000;
      }
      op_count += bruteForceMedian(A, i);
   }
   cout << op_count;
   op << op_count/10 << "\n";
   op_count = 0;
}
```

This data was then graphed on with the most accurate associated trend line added, along with n^2 trendline the shown in figure 5.2.

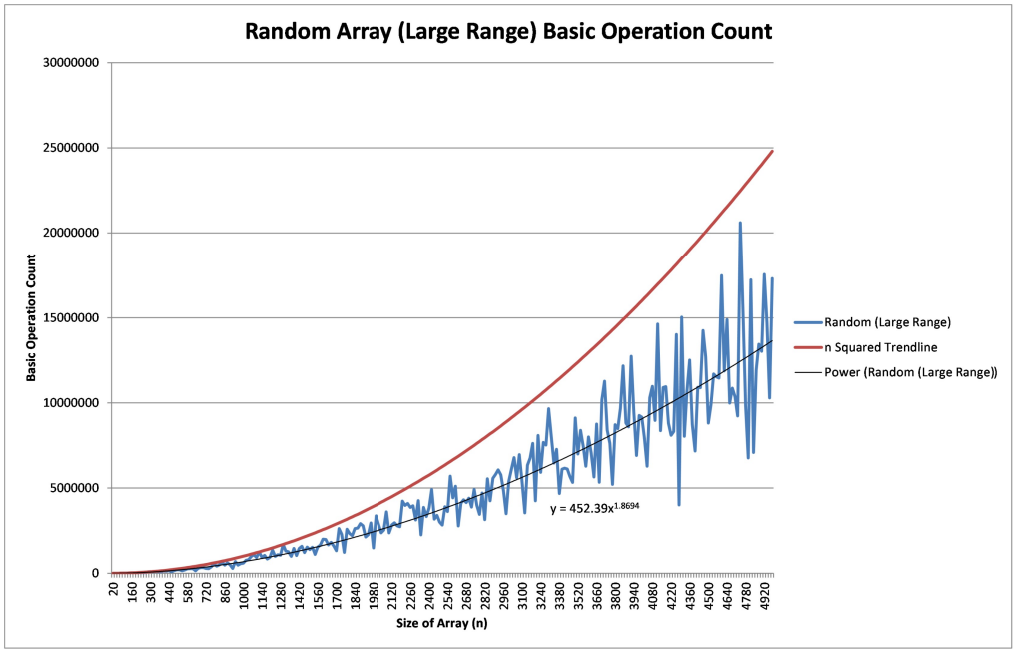

Figure 5.2: Random (Large Range) Basic Operation Count

Discrepancies in this experimental data all all related to the addition of possible matching

elements, causing the mean element to be found quicker.

5.3 TEST 3

The final set of data was produced using any random number from 0 to 50.

```
int A[5000];
std::ofstream op("output_mixed_small.txt");
int op_count;
for(int i = 20; i < 5000; i = i + 20){
   for(int tests = 0; tests < 10; tests++){
      for(int j = 0; j < i; j++){
         A[j] = rand() % 50;
      }
      op_count += bruteForceMedian(A, i);
   }
   cout << op_count;
   op << op_count/10 << "\n";
   op_count = 0;
}
```

As this data is seen to be much smaller then the previous experiments, it was graphed in figure 5.3 with an added trend line.

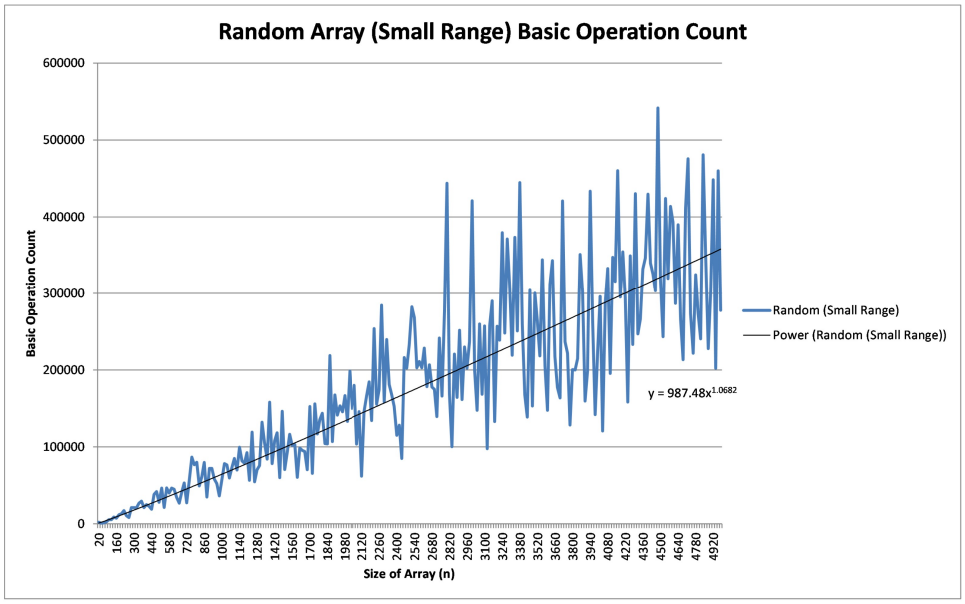

Figure 5.3: Unique Array Data

This data can be expected to have alot more matching elements then any previous data set. Meaning that the mean element will be found the very quickly, causing the data to trend toward the best case, n.

5.4 Comparisons and Findings

As the scales of the data are to widely apart, graphing all data on one graph was not possible unless scaled; instead trend line equations of the experiments above have been compiled in table 5.1.

Table 5.1: Comparison of Experimental Basic Operation Count

Worst Case	On^2
Unique Data Set	$n^{1.9615}$
Mixed Data Set (High Range)	$n^{1.8694}$
Mixed Data Set (Low Range)	$n^{1.0682}$
Best Case	σn

The data does does accurately match that of the theoretical expectation. With the only unexpected result of the unique array being explained by the missing 'equal' basic operation count. Along with the the data follows the expectation of it becoming closer to the best case as the matching array elements increase.Meaning the less unique numbers that are located in the array the closer the mean element will be to the first position; or closer to the best case, n. This can be confirmed by the table 5.1, with the power becoming closer to 1 as the random range decreases.

6 Experimental Execution Time

The execution time of a program should also follow the theoretical trends if the correct basic operation was chosen; as the basic operation by definition is the operation in which takes majority of the run time. To time this, the following was added to the main section of the code with the algorithm itself remaining the same as the original.

```
clock_start = clock();
bruteForceMedian(A, i);
clock_end = clock();
double run_time = (clock_end - clock_start)/CLOCKS_PER_SEC;
```

The added code takes the system clock either side of the algorithm and calculates the difference, scaling it to the given systems speed.

6.1 TEST 1

The first test data was that of full unique arrays; these were produced with the following code.

```
int A[5000];
std::ofstream run_timers("output_unique.txt");
double time;
   for(int i = 20; i < 5000; i = i + 20){
      for(int tests = 0; tests < 10; tests++){
         for(int j = 0; j < i; j++) {
            A[j] = j + 1;
         }
         clock_start = clock();
         bruteForceMedian(A, i);
         double run_time = (clock_end - clock_start)/CLOCKS_PER_SEC;
         time = time + run_time;
      }
      run_timers << time/10 << "\n";
      time = 0;
}
```

In this case, due to different scales, no experimental trendlines could be added, so this data will be placed with the next experimental in figure 6.1 in the section below.

6.2 TEST 2

This set of data is produced with randomly generated numbers from 0 to 5000 located anywhere in a given array.

```
int A[5000];
std::ofstream run_timers("output_mixed.txt");
double time;
   for(int i = 20; i < 5000; i = i + 20){
      for(int tests = 0; tests < 10; tests++){
         for(int j = 0; j < i; j++) {
            A[j] = rand() % 5000;
         }
         clock_start = clock();
         bruteForceMedian(A, i);
         clock_end = clock();
         double run_time = (clock_end - clock_start)/CLOCKS_PER_SEC;
         time = time + run_time;
      }
      run_timers << time/10 << "\n";
      time = 0;
}
```

This data was then graphed, with the previous experiments data with a scaled n^2 trendline shown in figure 6.1.

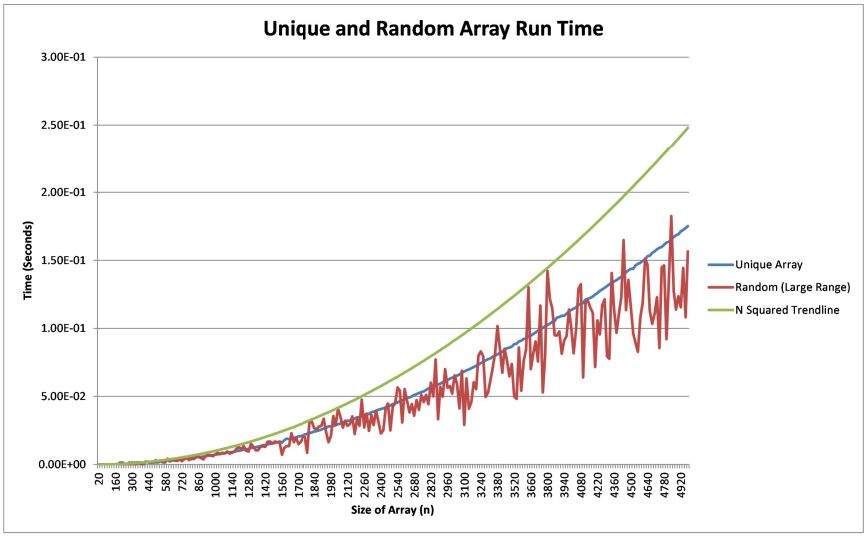

Figure 6.1: Unique Array and Random (Large Range) Execution Time

As shown in figure 6.1, the time data follows an almost identical pattern to that of the basic

operation.

6.3 TEST 3

The final set of data was produced using any random number from 0 to 50.

```
int A[5000];
std::ofstream run_timers("output_mixed_small.txt");
double time;
for(int i = 20; i < 5000; i = i + 20){
   for(int tests = 0; tests < 10; tests++){
      for(int j = 0; j < i; j++) {
         A[j] = rand() % 50;
      }
      clock_start = clock();
      bruteForceMedian(A, i);
      clock_end = clock();
      double run_time = (clock_end - clock_start)/CLOCKS_PER_SEC;
      time = time + run_time;
   }
   run_timers << time/10 << "\n";
   time = 0;
}
```

As this data is seen to be much smaller then the previous experiments, it was graphed alone with an added trend line.

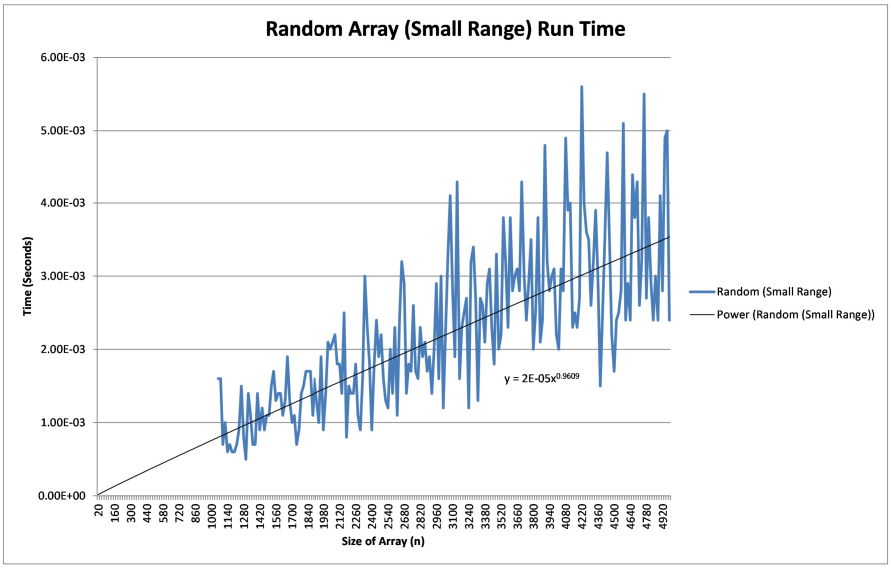

Figure 6.2: Random(Small Range) Execution Time

The data trendline in figure 6.2 is shaped badly at $x^{.9609}$ is due to lower points having such

small execution times, in which were output as 0seconds. Having removed this skewing data to create accurate graphs a new output was found. In saying this, the actual calculated trendline with this data was found to be $x^{1.087}$ in which will be used for comparison.

6.4 Comparisons and Findings

Due to the scale of n being to large for the represented data. Power trendlines were unavailable. Therefore a visual comparison will suffice in terms of execution time referring to that of figure 6.1 and its associated n^2 trendline. Looking at Figure 6.1, you can see that these experiments match the same trends as their basic operation counter parts; both basic operation count and execution time are effected by the same thing. Meaning that the execution time is reduced based on the basic operation completed, in which is related to the amount of matching elements and mean position. The higher the matching elements, the closer to the best case the algorithm will be.

In figure 6.2 it can be seen that the growth ($x^{1.087}$) is very close to that of n. This again proves that the more matching elements in which exists in a given array, the closer it will be to matching the best case scenario (mean located at the first array element).