Homework 2: CS 652 Advanced Algorithms

Devin Koehl

Fall Semester 2018

Question 1:

1. What is the recurrence for the running time of the closest-pair problem using the techniques a) brute force and b) divide and conquer ?

Answer:

We are given n points in a plane and must find a pair with the smallest distance between them. d = $\sqrt{(x_1 - x_2)^2 + (y_2 - y_1)^2}$

a) If we use a brute force technique, there are N points and we compare every N point to N-1 points. This would require N*N-1 comparisons of each point. The notation

$$\binom{n}{k} = \frac{n!}{k!(n-k)!}$$

reads "n choose k" and denotes the number of k-combinations of an n-set. (Source: Cormen, page 1185 on Binomial Coefficients).

Now we want 2 combinations (since we want a pair of points) of the n-set.

$$\binom{n}{2} = \frac{n!}{2!(n-2)!}$$

Applying this to our problem, we want 2 combinations of the n-set. This gives us the overall running time of $O(n^2)$.

b) If we use the divide and conquer approach, the algorithm the recurrence relationship is as follows (source: Corman text 33.4, page 1039):

1. Divide: Find the vertical line that that bisects the point set P into two sets, P_L (L=left) and P_R (R=right)

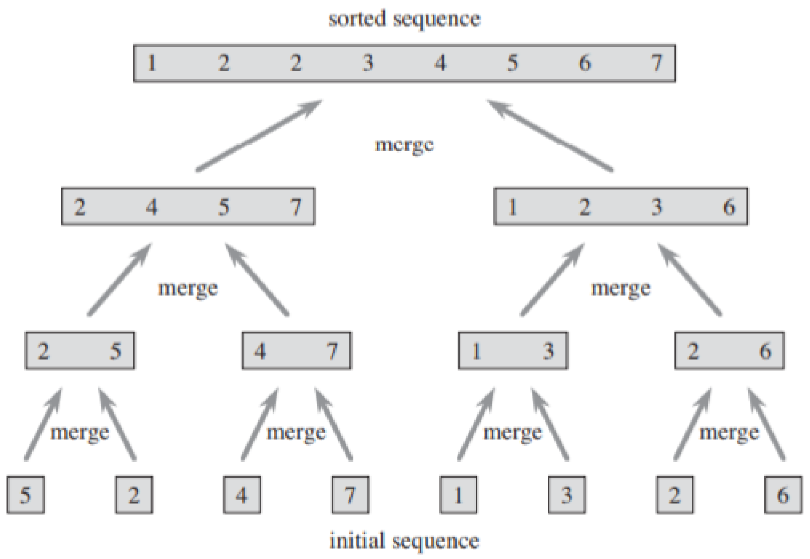
2. Conquer: Having divided P into P_L and P_R, make two recursive calls, one to find the closest pair of points in P_L and the other to find the closest pair of points in P_R

3. Combine: The closest pair is either the pair with distance δ found by one of the recursive calls, or it is a pair of points with one point in P_L and the other in P_R.

Overall recurrence relationship: T(n) = $2T(\frac{n}{2})$+O(n). This recurrence relationship shows we make two recursive calls each time splitting the plane in half each each time. The merge routine takes O(n) time. If we use the Master Method in the Cormen text (source: page 95) with a=2, b=2, and c=1 we have $n^{log_b^a}$. Filling in our values we have $n^{log_2^2}$ which yeilds 1. This is case II of the Master Method in the Cormen text which tells us that our time complexity will be O(nlogn)

Question 2:
Suppose Mergesort routine is applied on the following input arrays: 5, 3, 8, 9, 1, 7, 0, 2, 6, 4. Fast forward to the moment after the two outermost recursive calls complete, but before the final Merge step. Thinking of the two 5-element output arrays of the recursive calls as a glued-together 10-element array, which number is the 7th position ?

Answer: Recall from Cormen text the process of merge sort. Source: Cormen, page 35.



The graphic shows the final step before the final merge. Both sides of our split array are sorted. This means in one half of the array we have [1 3 5 8 9] and [0 2 4 6 7]. If we glued this together like the question asks we would have [1 3 5 8 9 0 2 4 6 7]. Our 7th element position is 2.

Question 3:
Compute 1201*2430 applying the divide and conquer algorithm of Karatsuba's method. Exit the recursion when number of digits n = 1, i.e. computing product of two numbers using the traditional method when the number of digits is 1

Answer: (Source: Coursera, Divide and Conquer Algorithm Course from Stanford University)

The steps to solving this algorithm:

Find the values of a, d, and e, where a=$x_h y_h$, d = $x_l y_l$ and e =$(x_h + x_l)(y_h + y_l) - a - d$. Lastly, we compute xy = ar^n+e$r^{\frac{n}{2}} + d$. We will recurse each time we must find a product until we reach our base case. Take our values 1201 and 2430 and apply this approach. First time through, we have a=12*24 and d = 01*30. Now e = (12+01) (24+30) - a - d. Now we must recurse to find 12*24, 01*30, and 13*54.

Starting with our first product of 12*24, we have a=1*2 = 2. d = 2*4 = 8, and e = (1+2)(2+4)-2-8, which gives is 8. Now we take our xy expression and apply it here (n in the expression corresponds to the number of digits we are recursing on, so here n=2). 2*10^2+8*10+8 = 288.

Now we take this approach to our second subproblem. (01*30). Here we have a=0*3 = 0, d = 1*0 = 0. e = (0+1)(3+0)-a-d = 3. Now take our xy expression where we have n=2. 0*10^2+3*10+0 = 30.

Lastly, we take our expression 13*54 and apply the same method. a=1*5 = 5, d = 3*4 = 12, and e=(1+3)(5+4)-a-d= 19. Apply our xy expression. 5*10^2+19*10+12 = 702.

Finally, we combine all of these steps together to produce the final result. We can fill in the overall numbers with what we have computed above, we can solve for 708-288-30 = 384.

Now we have all the pieces and use our method to compute xy = ar^n+e$r^{\frac{n}{2}} + d$. We found the solution to subproblem a = 288. Solution to d = 30, and solution to e = 384. Now n = 4 since we have four digits. 288*10^4+384*10^2+30 = 2,918,430.

Question 4:
Apply Strassen's algorithm to compute

$$\begin{pmatrix} 1 & 0 & 2 & 1 \\ 4 & 1 & 1 & 0 \\ 0 & 1 & 3 & 0 \\ 5 & 0 & 2 & 1 \end{pmatrix} \begin{pmatrix} 0 & 1 & 0 & 1 \\ 2 & 1 & 0 & 4 \\ 2 & 0 & 1 & 1 \\ 1 & 3 & 5 & 0 \end{pmatrix}$$

Exiting the recursion when n = 2, i.e. computing the products of 2 x 2 matrices by the brute force method.

Answer:

Now a_{00} =

$$\begin{pmatrix} 1 & 0 \\ 4 & 1 \end{pmatrix}$$

a_{01} =

$$\begin{pmatrix} 2 & 1 \\ 1 & 0 \end{pmatrix}$$

a_{10} =

$$\begin{pmatrix} 0 & 1 \\ 5 & 0 \end{pmatrix}$$

a_{11} =

$$\begin{pmatrix} 3 & 0 \\ 2 & 1 \end{pmatrix}$$

b_{00} =

$$\begin{pmatrix} 0 & 1 \\ 2 & 1 \end{pmatrix}$$

b_{01} =

$$\begin{pmatrix} 0 & 1 \\ 0 & 4 \end{pmatrix}$$

b_{10} =

$$\begin{pmatrix} 2 & 0 \\ 1 & 3 \end{pmatrix}$$

b_{11} =

$$\begin{pmatrix} 1 & 1 \\ 5 & 0 \end{pmatrix}$$

Now take these and calculate the following:

$m_1 = (a_{00} + a_{11})$*$(b_{00}+b_{11})$
$m_2 = (a_{10} + a_{11})$*b_{00}
$m_3 = a_{00}$*$(b_{01} - b_{11})$
$m_4 = a_{11} * (b_{10} - b_{00})$
$m_5 = (a_{00} + a_{01}) * b_{11}$

$m_6 = (a_{10} - a_{00}) * (b_{00} + b_{01})$
$m_7 = (a_{01} - a_{11}) * (b_{10} + b_{11})$

$m_1 =$

$$\begin{pmatrix} 4 & 8 \\ 20 & 14 \end{pmatrix}$$

$m_2 =$

$$\begin{pmatrix} 2 & 4 \\ 2 & 8 \end{pmatrix}$$

$m_3 =$

$$\begin{pmatrix} -1 & 0 \\ -9 & 4 \end{pmatrix}$$

$m_4 =$

$$\begin{pmatrix} 6 & -3 \\ 3 & 0 \end{pmatrix}$$

$m_5 =$

$$\begin{pmatrix} 8 & 3 \\ 10 & 5 \end{pmatrix}$$

$m_6 =$

$$\begin{pmatrix} 2 & 3 \\ -2 & -3 \end{pmatrix}$$

$m_7 =$

$$\begin{pmatrix} 3 & 2 \\ -9 & -4 \end{pmatrix}$$

$c_{00} = m_1 + m_4 - m_5 + m_7 =$

$$\begin{pmatrix} 5 & 4 \\ 4 & 5 \end{pmatrix}$$

$c_{01} = m_3 + m_5$

$$\begin{pmatrix} 7 & 3 \\ 1 & 9 \end{pmatrix}$$

$c_{10} = m_3 + m_5 + m_2 + m_4$

$$\begin{pmatrix} 8 & 1 \\ 5 & 8 \end{pmatrix}$$

$c_{11} = m_1 + m_3 - m_2 + m_6$

$$\begin{pmatrix} 3 & 7 \\ 7 & 7 \end{pmatrix}$$

Now combine all of answers and

$$\begin{pmatrix} c_{00} & c_{01} \\ c_{10} & c_{11} \end{pmatrix} = \begin{pmatrix} 5 & 4 & 7 & 3 \\ 4 & 5 & 1 & 9 \\ 8 & 1 & 3 & 7 \\ 5 & 8 & 7 & 7 \end{pmatrix}$$

Question 5:
Define the recursive depth of Quicksort as the maximum number of successive recursive calls it makes before hitting the base case - equivalently, the largest level of its recursion tree. In randomized Quicksort, the recursion depth is a random variable, depending on the pivots chosen. What is the maximum and minimum possible recursion depths of randomized Quicksort (in Big-Oh notation)?

Answer:

Source: Cormen, page 175 on Quick sort.

The worst case behavior for quick is when our partioning routine produces one subproblem with n-1 elements and one with 0 elements. If we assume the unbalanced partitioning arises in each recursive call, the partioning costs O(n) time. The recursive call on the array of size 0 is constant time so we have T(0) = O(1). Overall recurrence is:

T(n) = T(n-1) + T(0) + O(n)

T(n) = T(n-1) + O(n)

The best case is where the partition produces two subproblems each of size $\frac{n}{2}$. Quicksort runs much faster. The recursion tree in this case can have between 1 and 2h nodes. The height would be logn. The recurrence relationship is

T(n) = 2T $(\frac{n}{2})$ + O(n)

If we use the Master Method in the Cormen text (source: page 95) with a=2, b=2, and c=1 we have $n^{log_b^a}$. Filling in our values we have $n^{log_2^2}$ which yeilds 1. This is case II of the Master Method in the Cormen text which tells us that our time complexity will be O(nlogn)

Overall, the minimum depth is O(lgn) while the maximum depth would be O(n).

Question 6:

Suppose the running time T(n) of an algorithm is bounded by the recurrence

T(1) = 1 and T(n) $\leq$ T($\sqrt{n}$)+1 for n > 1. Which of the following is the smallest correct upper bound on the asymptotic running time of the algorithm: (a) O (1) (b) O(log log n) (c) O(log n) (d) O ($\sqrt{n}$)

Answer:

(b) O(log log n). This was a little tricky for me and I admit I had to use Wolfram Alpha and some Math Stack Exchage to get deeper into the Math. First take

T(n) $\leq$ T($\sqrt{n}$)+1 and rewrite the square as power $\frac{1}{2}$. We will have T(n) = T($n^{\frac{1}{2}}$) + 1 = T($n^{\frac{1}{4}}$) + 2 = T($n^{\frac{1}{8}}$) + 3 = T($n^{\frac{1}{2^k}}$) + k. This will continue until the base case of ($n^{\frac{1}{2^k}}$). Solving for k yields log n = 2^k. Taking the log of 2^k and solving for k, k = log log n. The answer is (b)

Question 7:

You are given an array A of n distinct elements, such that its entries are in increasing order up until its maximum element, after which its elements are in decreasing order. Give an algorithm to compute the maximum element of the array A that runs in logarithmic time.

Answer:

Right off the bat, I know we would use binary search to do this if we want logn time. If we used linear search, it would run in O(n) time because it would have to traverse the list comparing each element to find the maximum. (Source: Cormen page 799). If we implore binary search, we can split our array into two subarrays. We could have three cases: The maximum is the middle number, the maximum is less than the middle number, the maxiumum is greater than the middle number

1. Create method findMaximum(int arr[ ], int low, int high)

2. Calculate the middle element, we will use this. middle = low + (low + high)/2

3. Account for the base case where low is equal to high. This means we have only one element present in the array

4. Now if the base case is not met, it is because we have more than one element. Compare the second element to the first, if it's greater than the first maximum arr[low] < arr[high]

5. Compare the middle element to both adjacent elements. If it is greater

we know that the middle is the maximum. return arr[middle]

6. Compare the middle element to the next element. If it is greater than the next element yet smaller than the previous element, we know that the maximum lies in our left subarray that goes from low-middle-1. Recursively call our method on the subarray findMaximum(arr,low,middle-1)

7. Now for the last case, it must be in the right subarray which goes from middle+1 to high.

Binary search has the recurrence relationship T(n) = T(n/2) + 1. If we use the Master Method in the Cormen text (source: page 95) with a=1, b=2 and c=1 we have $n^{log_b^a}$. Filling in our values we have $n^{log_2^1}$ which yeilds 0. n^0 is 1. This is case II of the Master Method in the Cormen text which tells us that our time complexity will be O(logn).

Question 8:
You are given a sorted (from smallest to largest) array A of n distinct integers which can be positive, negative or zero. Design the fastest algorithm you can for deciding if there is an index i such that A[i] = i.

Answer:

I would again use binary search for this type of problem. The algorithm would be similar to what we have already seen from binary search:

1. While low and high do not meet

2. Calculate the middle (low + high) / 2

3. if array[middle] < middle, then high = middle - 1. This correspondings to the case if the index we are searching for is less than the middle. We now will recursively search the first half of the array from low to the middle-1.

4. Else if (array[middle] > middle), this corresponds to the case where the index we are searching for is greatter than the middle. We will now recursively search the first half of the array from middle+1 to high.

5. Else return middle, this is the case where we find the value that is equal to the middle

6. If we do not find the value, return -1

Binary search has the recurrence relationship T(n) = T(n/2) + 1. If we use the Master Method in the Cormen text (source: page 95) with a=1, b=2 and c=1 we have $n^{log_b^a}$. Filling in our values we have $n^{log_2^1}$ which yeilds 0. n^0 is 1. This is case II of the Master Method in the Cormen text which tells us that our time complexity will be O(logn).

Binary search has the recurrence relationship T(n) = T(n/2) + 1. If we use the Master Method in the Cormen text (source: page 95) with a=1, b=2 and c=1 we have $n^{log_b^a}$. Filling in our values we have $n^{log_2^1}$ which yeilds 0. n^0 is 1. This is case II of the Master Method in the Cormen text which tells us that our time complexity will be O(logn).

Question 9:

We define k-shift as taking the last k entries of an array A and moving it to the front of the array. For example 1-shift, 2-shift, 3-shift, 4-shift, and 5-shift operation on the array A [1,2,3,4,5] will respectively yield the arrays [5,1,2,3,4],[4,5,1,2,3], [3,4,5,1,2], [2,3,4,5,1], [1,2,3,4,5].

Given array A (size n) is sorted in increading order and some and some randomly chosen k-order shift ($1 \leq k \leq n$) is applied to the array A. Devise an algorithm to find an element in the array A in logarithmic time.

Answer:

We can use binary search on this problem even though a k-shift has been applied. Although our array is shifted, it is key to note that one half of the array will be sorted. Let us take [2 3 4 5 6 7 8] and do a 4-shift. Now we have [4 5 6 7 8 2 3]. Note that finding the middle here (0+6)/2 = 3. The array corresponding to elements before the middle [4 5 6] is sorted while the array correspondings to elements after the middle [8 2 3] is unsorted. This gives us an advantage. Now we can use binary search in O(lgn) time.

1. While low and high do not meet

2. Calculate the middle (low + high) / 2

3. if array[middle] < middle, then high = middle - 1. This is the first half of the array that is already sorted.

4. Else if (array[middle] > middle), this corresponds to the case where the index we are searching for is greatter than the middle. Now we can recursively sort this side of the array.

5. Else return middle, this is the case where we find the value that is equal to the middle

6. If we do not find the value, return -1

Binary search has the recurrence relationship T(n) = T(n/2) + 1. If we use the Master Method in the Cormen text (source: page 95) with a=1, b=2 and c=1 we have $n^{log_b^a}$. Filling in our values we have $n^{log_2^1}$ which yeilds 0. n^0 is 1. This is case II of the Master Method in the Cormen text which tells us that our time complexity will be O(logn).

Question 10:

Design a recursive algorithm for computing 2^n for any non-negative integer n that is based on the formula $2^n - 1 + 2^n - 1$. Set up the recurrence relation for the number of additions made by the algorithm and solve it. Draw a tree of recursive calls for this algorithm and count the number of calls made by the algorithm. Is this a good algorithm for soving his program?

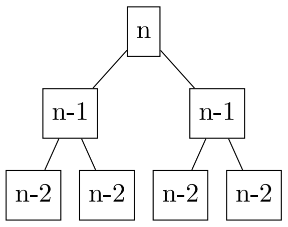
Answer:

This would be a case of a simple algorithm using recursion. We must account for the base case, which would be if we have 2^0 which is 1. After this, it would recursively call itself n-1 times.
Sudocode would be something like:

```
public static int power(int n)
if(n == 0)
return 1
else
return power(n-1) + power(n-1)
```

To set-up the recurrence relationship, we need to plug in values into our equation
If we n=0, we have our base case.
newline If n = 1, then we get 1
newline If n = 2, then 1 + (1 + 1) = 3.
If n=3, then 1 + (1 + (1 + 1) + (1 + (1 + 1)) = 7
From this we can see that each time our value is an odd number, giving us the recurrence relationship T(N) = $2^n - 1$

Recursion Tree



This will continue all the way until we get to our base case of 1. Each node corresponds to a recursive call to the power function. This is not the fatest way to calculate this. Algorithms with powers run in O(n^2) time. A simplier approach would be O(n) and just multiplying 2 with itself n times in a for loop.

Question 11:

The following algorithm seeks to compute the number of leaves in a binary tree. The input is a binary tree T, and the output is the number of leaves in T. Is this algorithm correct? If it is, prove it; if it not, make appropiate corrections.

Answer:

This algorithm is incorrect because we not accounting for if our left AND right nodes are null. Currently, we are saying "If the tree is empty, then return 0". Yet what if we have a tree with a single node? This means that the single node should be counted. If we rewrite this
if T is empty then return 0;
if T.left == null AND T.right == null then return 1
else return LeafCounter(T_left) + LeafCounter(T_right);

We now have accounted for if we have a tree with no nodes, a single node with no leaves, and then a tree with multiple leaves.

Question 12:

Design an algorithm to reaarange elements of a given array of n real numbers so that all its negative eements precede all its positive elements. Your algorithm should be both time efficient and space efficient.

Answer:

If we invoke the partition process of quicksort, we can disect our array into three sections. Lets use the variables m,first, and last. M in this case

1. A section of the array will have elements A[0...m-1] that will always be negative

2. A section of the array A[m...n] which will contain elements that have not been scanned

3. A[n+1....length-1] which will contain positive values

pseuocode:

1. First we will have a method called arrangeArray that will take an array and a and a length public static void(int array[], int length)

2. Next we will loop over the length of our array and where we have elements less than 0, swap them.

if array[i] < 0 (this corresponds to the negative portition of the array)

if i =! j, if SWAP elements. (this will put the elements in the front of the array in our first partition).

This algorithm runs in O(n) time because we will have a single for loop to loop over the length of the array.

Question 12:

Given two integers x and n where n is non-negative, efficiently calculate the value of the power function pow(x,n). Example: pow(-2,10) = 1024. How many recursive calls are you invoking?

Answer:

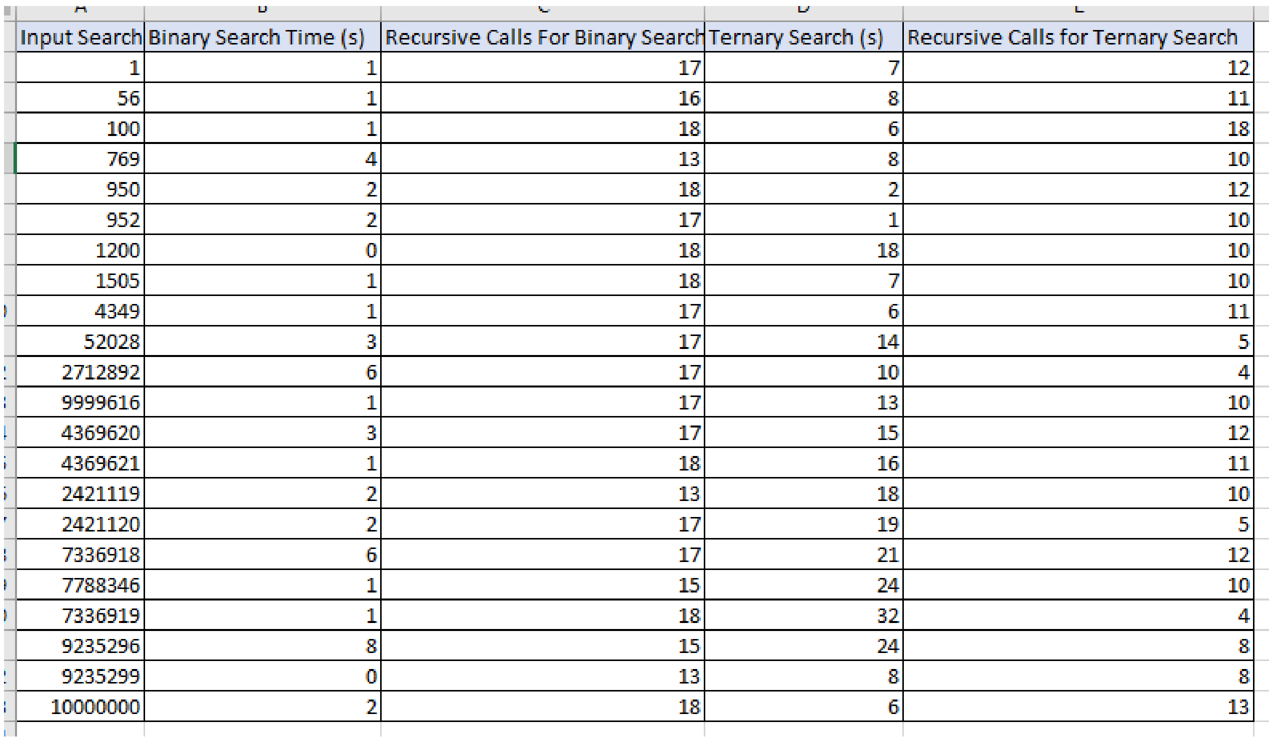
See Python files. I made a global variable that counts the recursion. It counted 10 times the method recursed in order to get (-2,10) = 1024.

Question 13:

Implement binary search and ternary search. Input to these functions would be an array A of length n and a target number x. Find attached a dataset containing 100000 numbers in a sorted array. Run binary and ternary search on the given dataset for the following numbers: 1, 56, 100, 769, 950, 952, 1200, 1505, 4349, 52028, 2712892, 9999616, 4369620, 4369621, 2421119, 2421120, 7336918, 7788346, 7336919, 9235296, 9235299, and 10000000. For each of the target numbers, report, if they have been found in the dataset, and the number of recursive calls both binary and ternary search had to make. Also, plot the running time for both search operations for all the target numbers.

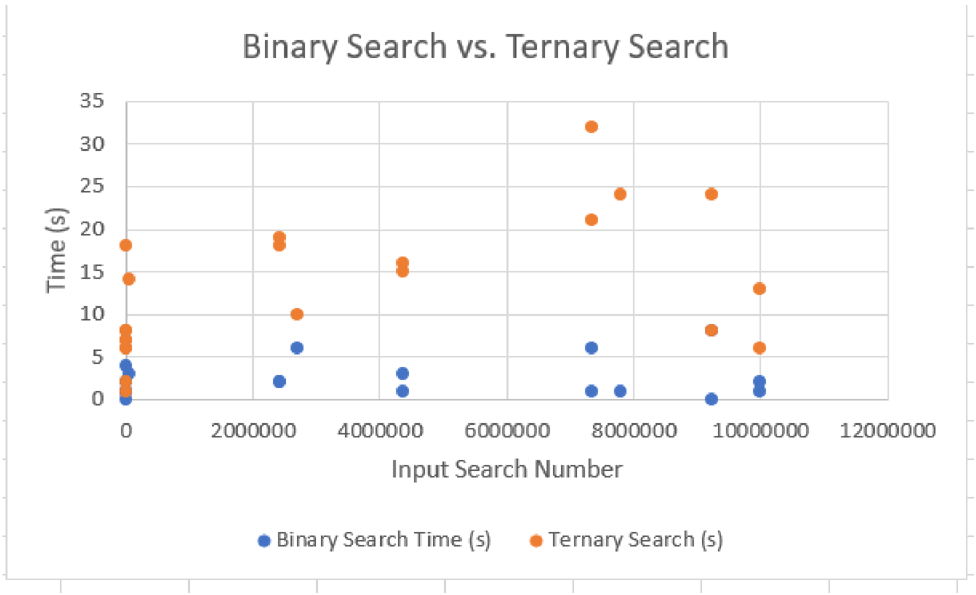
Answer:

After running and plotting each, here is the table:



Input Search	Binary Search Time (s)	Recursive Calls For Binary Search	Ternary Search (s)	Recursive Calls for Ternary Search
1	1	17	7	12
56	1	16	8	11
100	1	18	6	18
769	4	13	8	10
950	2	18	2	12
952	2	17	1	10
1200	0	18	18	10
1505	1	18	7	10
4349	1	17	6	11
52028	3	17	14	5
2712892	6	17	10	4
9999616	1	17	13	10
4369620	3	17	15	12
4369621	1	18	16	11
2421119	2	13	18	10
2421120	2	17	19	5
7336918	6	17	21	12
7788346	1	15	24	10
7336919	1	18	32	4
9235296	8	15	24	8
9235299	0	13	8	8
10000000	2	18	6	13

Plotting this we see the following:



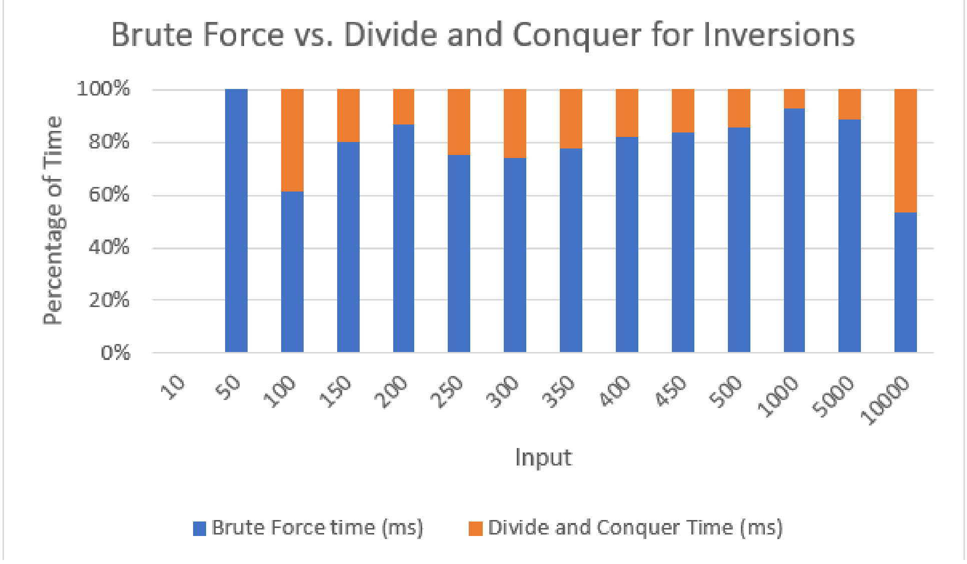
From the scatter plot we can see searching for small values between binary and ternary search seem similar, yet the higher values we must search for, binary search does much better than ternary search.

Question 13:

Let a1,a2,a3....an be a permutation of 1,2,3,...n. (a_i,a_j) is an inversion of i < j and ai ¿ aj. Your task is to implement both the brute force method (O(n^2) and the divide and conquer method O(nlgn) for counting the number of inversions for a given array A of size n, that contains a permutation of 1,2,3...n.
You are provided with 14 input files. These files contain random permutation of 1,2,3...n. The filename represents the number of entries in each of the files. Use these as input for your program. Plot the running time of both methods (brute force and divide and conquer) on these 14 datasets. Also, report the number of inversions in all datasets.

Please note I plotted the time in microseconds so I could see more of a difference.

Input Search	Brute Force time (ms)	Divide and Conquer Time (ms)	Number of Inversions
10	0	0	23
50	171	0	667
100	1582	999	2814
150	3989	997	6037
200	6969	1071	10633
250	8976	2991	16628
300	14258	4985	22636
350	18018	5095	29353
400	32141	6990	42671
450	36133	6874	53486
500	48195	7940	63795
1000	176673	13294	248362
5000	725483	90559	6219990
10000	221951	195865	25054124


Brute Force vs. Divide and Conquer for Inversions
Percentage of Time
Input
Brute Force time (ms)
Divide and Conquer Time (ms)