605.621 Foundations of Algorithms
Programming Assignment 1

Hannah Ripley

September 30th, 2024

Closest Pairs (1)

Overview

The Closest Pairs problem takes a list of coordinate points as an input and returns the pair of points with the shortest Euclidean distance.

Euclidean Distance is here defined as $D = \sqrt{(x_i - x_j)^2 + (x_i - x_j)^2}$

I have designed and implemented two algorithmic approaches: a naive, brute-force approach that compares the distance between every two points, and a recursive approach, which divides the list into sublists, determines the closest pair in each sublist, and compares the results of each sublist.

Brute Force Closest Pairs (1.a)

Algorithm Description

The most straight-forward way to find the closest pair of points is to compare the distance of each point with every other point, keeping track of the minimum distance and its associated points.

I've developed the following brute-force algorithm to calculate the closest pair in a list of points:
Taking *Data Set A,* a list of points as input:

1) Initialize the closest pair and minimum distance
2) Iterate over every point in Data Set A
3) Within the first loop, iterate over every point in Data Set A
4) Calculate the distance between the outer point and the inner point
5) Compare the latest distance to the minimum distance, and if the latest distance is smaller:
    - Update minimum distance
    - Update closest pair
6) Return the minimum distance and closest pair

Pseudocode:

```
CLOSEST_PAIR_BRUTE(Data Set A):
  minimum distance = INFINITY (so any actual distance will be smaller)
  closest pair = UNDEFINED (because no comparisons have been made yet)
  For index1 starting at 0 to length of Data Set A:
    For index2 starting at 0 to length of Data Set A:
      skip over index1 == index2 (do not compare a point to itself)
      calculate the distance between the Point at index1 and the Point at index2
      If distance < minimum distance:
        update minimum distance = distance
        update closest pair = Pair(Point at index1, Point at index2)
  Return closest pair, minimum distance
```

Analysis (1.a)

Let running time complexity be defined as the rate at which the number of steps grows as the size of the dataset grows. Here, the number of steps is defined as the number of times the distance is calculated between two points.

This algorithm has a running time complexity of $\Theta(n^2)$.

Proof:

- The algorithm loops over every element in the dataset.
- Within the first loop, the algorithm loops over every element in the dataset again.
- The only circumstance under which the algorithm does not compare two elements is when both loops are iterating over the same element.
  - This is guaranteed to happen because both loops iterate over every element.
- Within the inner loop, the distance is calculated between the element of the inner and outer loop.
- Thus the distance is calculated for every permutation of two elements.
  - i.e. The algorithm calculates both distance(p1, p2) and distance(p2, p1).
- Thus the total number of distance calculations will be exactly $n^2 - n$ for any dataset of size n.
- $\Theta(n^2 - n) = \Theta(n^2)$

Therefore the running time complexity of this algorithm is $\Theta(n^2)$.

There is no circumstance under which, depending on the values of the dataset, the algorithm performs less steps. As such, it is also true that the worst-case time complexity is $O(n^2)$, and the best-case time complexity is $\Omega(n^2)$.

Experiment and Findings (1.b)

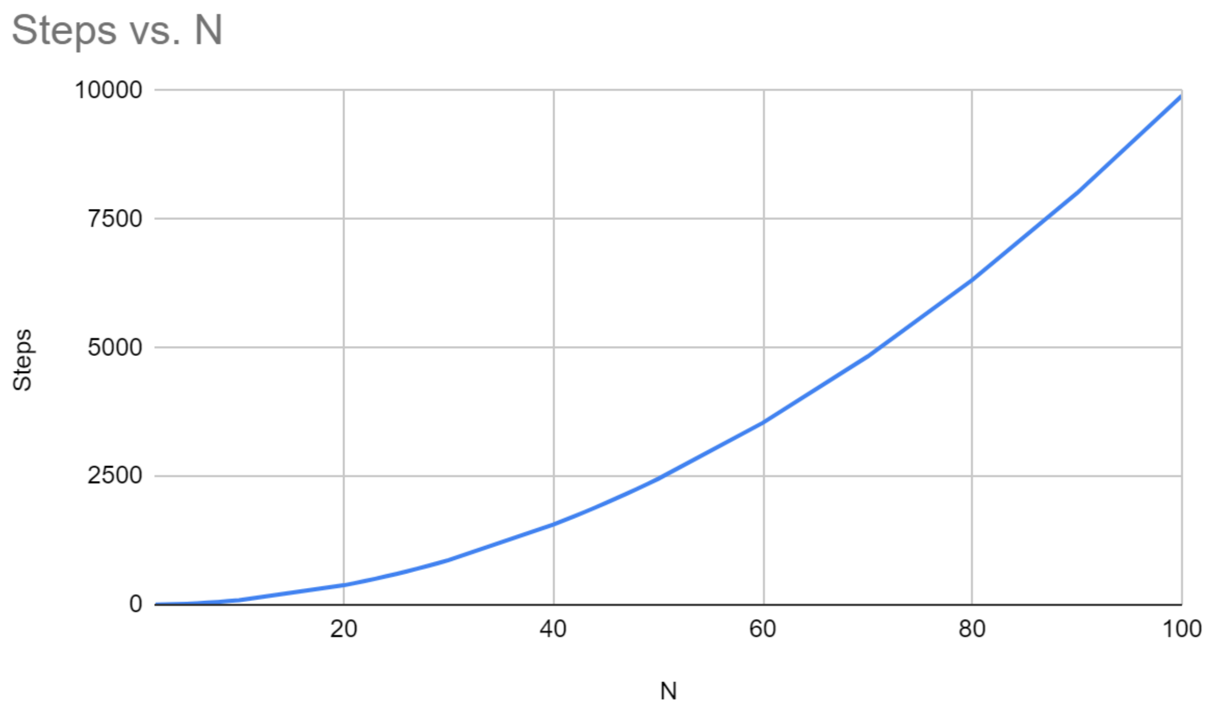
In this experiment, I used the brute force algorithm to calculate the closest pairs for many randomized datasets of points. I utilized a counter in the code to count the number of steps the algorithm required for each dataset. Here, the number of steps is defined as the number of times the distance is calculated between two points.

As input, I generated randomizes lists of points of the following sizes: [2, 3, 4, 5, 6, 7, 8, 9, 10, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50, 60, 70, 80, 90, 100].

For example:

- For size 4, input = [(17,31), (38,88), (48,23), (74,94)]
- For size 7, input = [(5,39), (18,52), (37,36), (68,90), (94,6), (100,84)]

Below, I have graphed the relationship between each dataset's size and the number of steps it required to determine the closest pair.



The result of this experiment confirms my theoretical performance expectations. Because no point calculates the distance to itself, the number of steps is exactly n^2-n for any dataset of size n. It is impossible for there to be any outliers.

Recursive Closest Pairs (1.c)

Algorithm Description

In an attempt to achieve a more efficient running time complexity for determining the closest pair, I have designed an algorithm that recursively identifies the closest pairs for subsets of the data set, comparing and returning the closer pair between subsets as they merge back together.

In designing this algorithm, I considered the following: for a data set of one-dimensional points (i.e. a set of points with the same y-value but different x-values), the closest pair would consist of some two adjacent elements in a sorted list (sorted by x-values).

The closest pair of points in a one-dimensional data set of size n can using the following algorithm:

1) Sort the input data set
2) Initialize the closest pair and minimum distance
3) Iterate over the sorted list (excluding the final element, x_{n-1})
4) For each element $x_i \in$ the sorted list, calculate the distance $|x_i - x_{i+1}|$

5) Compare $|x_i - x_{i+1}|$ to the minimum distance, if smaller:
    - Update minimum distance = $|x_i - x_{i+1}|$
    - Update closest pair = (x_i, x_{i+1})
6) Return the minimum distance and closest pair

Conventional fast sorting algorithms (i.e. merge sort, heapsort, and quicksort) achieve a running time complexity of $\Theta(n \log n)$. For an input data set A, such that all elements have the same y-value, making the data set functionally one-dimensional, it would take, at minimum, a $\Theta(n \log n)$ run time complexity to sort the points. Consequently, I determined it would be impossible for the algorithm to have a better runtime than $\Theta(n \log n)$.

The brute force algorithm has a $\Theta(n^2)$ running time complexity. Consequently, in order for the recursive algorithm to be an improvement over its brute-force counterpart, it must utilize some technique of $\Theta(n \log n)$ running time complexity and no technique of larger than $\Theta(n^2)$ running time complexity.

Due to the difficulty of this problem, I iteratively developed my algorithm until it consistently identified the closest pair and its distance. For the final algorithm, skip to "Iteration Two: Bisecting Pairs."

Iteration One: Sort and Merge

By first using merge sort to sort the data set, I attempted to maintain $\Theta(n \log n)$ runtime complexity and flatten one of the two dimensions of the point set.

This algorithm took heavy inspiration from the merge sort algorithm. In addition to utilizing merge sort to sort the data set, the algorithm performs similar divide-and-conquer strategies on subsets of the data set, recursively comparing closest pairs by distance.

The first iteration of the divide-and-conquer algorithm to calculate the list of closest pairs is as follows: Taking *Data Set A*, a list of points as input:

1. Use merge sort to initially sort the data set by x-coordinate
2. Recursively split the sorted list into two sublists
3. Hit the base case when a sublist contains three or less elements
    a. Calculate the closest pair of the small sublist using the brute force algorithm (as described above)
    b. Return the closest pair and smallest distance for the small sublist
4. Compare the distances returned by two sublists
5. Return the smaller distance and it associated closest pair
6. Recursively compare and return until a final smallest distance and closest pair are determined
7. Return smallest distance and closest pair

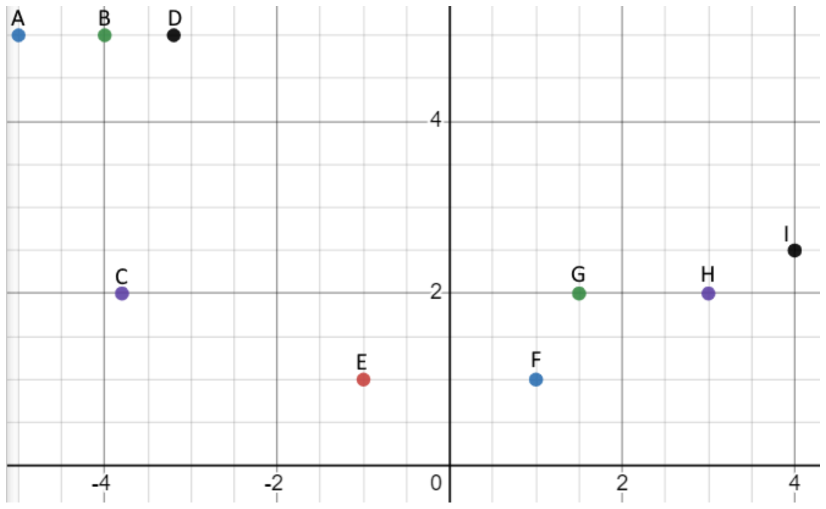
Pseudocode:

```
Use merge sort to sort Data Set A by x-coordinate then
CLOSEST_PAIR_RECURSIVE(Data Set A):
 If length of Data Set A <= 3:
   return CLOSEST_PAIR_BRUTE(Data Set A) (specified above)
 Else:
   left pair,  distance = CLOSEST_PAIR_RECURSIVE(first half of Data Set A)
   right pair, distance = CLOSEST_PAIR_RECURSIVE(second half of Data Set A)
   If left distance < right distance:
     return left pair, left distance
   Else:
     return right pair, right distance
```

Despite achieving the target running time complexity, this algorithm produced incorrect results for the majority of dataset inputs. The distance returned by the algorithm was consistently larger than anticipated. Consequently, I came to the conclusion that this iteration of the algorithm failed to identify some critical relationship(s) between points.

Exercise: Understanding Distance Relationships

In order to better understand potential complications in the algorithm, I created a graph, attempting to identify unintuitive relationships (e.g. non-reciprocal and non-adjacent closest pairs) between points:



I came to several conclusions:

- Points adjacent on the x-axis are not necessarily a closest pair
    - e.g. $x_B < x_C < x_D$, $but\ \Delta(B, D) < \Delta(B, C)$
- The closest pair (Point$_1$, Point$_2$) does not imply the closest pair (Point$_2$, Point$_1$)
    - e.g. $\Delta(A, B) < \Delta(A,\ any\ other\ point),\ but\ \Delta(B, D) < \Delta(B, A)$
- A closest pair may be split across a midpoint
    - e.g. $\Delta(E, F) < \Delta(E, D)$

My primary takeaway from this exercise was that I needed a mechanism for identifying and evaluating potential closest pairs that bisect sublists (i.e. were split between two halves of a list).

Iteration Two: Bisecting Pairs

The second iteration of the divide-and-conquer algorithm to calculate the list of closest pairs is as follows: Taking *Data Set A*, a list of points as input:

1. Use merge sort to initially sort the data set by x-coordinate
2. Recursively split the sorted list into two sublists
3. Hit the base case when a sublist contains three or less elements
   - Calculate the closest pair of the small sublist using brute force algorithm (as described above)
   - Return the closest pair and smallest distance for the small sublist
4. Compare and determine the smaller distance between two returned sublists
5. Find the midpoint of the the combined list of points
6. Iterate over the list of points
7. Calculate the distance between each point and the midpoint
8. If the distance is smaller than the current smallest distance, add it to the list of bisecting points
9. If the bisecting points list is empty, return current the smallest distance and closest pair
10. If the bisecting points list is the same as the full list of points, calculate the closest pair using the brute force algorithm
11. Otherwise, recursively calculate the closest pair and distance in the bisecting points list
12. Compare and determine the smaller distance between the current smallest distance and the bisecting point list's smallest distance
13. Return the smaller distance and it associated closest pair
14. Recursively compare and return until a final smallest distance and closest pair are determined
15. Return smallest distance and closest pair

Pseudocode:

```
Use merge sort to sort Data Set A by x-coordinate then
CLOSEST_PAIR_RECURSIVE(Data Set A):
 If length of Data Set A <= 3:
   return CLOSEST_PAIR_BRUTE(Data Set A) (specified above)
 Else:
   left pair,  distance = CLOSEST_PAIR_RECURSIVE(first half of Data Set A)
   right pair, distance = CLOSEST_PAIR_RECURSIVE(second half of Data Set A)
   If left distance < right distance:
     closest pair, minimum distance = left pair, left distance
   Else:
     closest pair, minimum distance = right pair, right distance
   bisecting points = GET_BISECTING_POINTS(minimum distance, Data Set A)
   If (length of bisecting points = 0):
     return closest pair, minimum distance
   Else If (length of bisecting points = length of A):
     bisecting pair, distance = CLOSEST_PAIR_BRUTE(Data Set A)
   Else:
     bisecting pair, distance = CLOSEST_PAIR_RECURSIVE(bisecting points)
   if bisecting distance < minimum distance:
     return bisecting pair, distance
   else:
     return closest pair, minimum distance

GET_BISECTING_POINTS(minimum distance, Data Set A):
   midpoint = the Point at the middle index of Data Set A
   bisecting points = EMPTY LIST
   For every point in Data Set A:
     calculate the distance between point and midpoint
     if distance < minimum distance:
       add point to bisecting points
     return bisecting points
```

This version of the algorithm achieved the target running time complexity and produced correct results for all randomized test dataset inputs.

Analysis (1.c)

Let running time complexity be defined as the rate at which the number of steps grows as the size of the dataset grows. Here, the number of steps is defined as the number of times the distance is calculated between two points, the number of times closest_pair_recursive is called, the number of times merge_sort is called, and the number of elements merge_sort accesses from a list.

This algorithm has a worst-case time complexity of $O(n^2)$.
Proof:

- Before the main closest pair algorithm runs, the input list is sorted by x-value using merge sort.
    - Merge sort is a well-known sorting algorithm with a time complexity of $\Theta(n \log n)$.
- When the size of the data set is <= 3, the algorithm performs the brute-force version of this algorithm.
    - Because $\Theta(n^2)$ is only used for n <=3, its rate does not grow as n grows.
    - Thus for the purposes of time complexity, we can treat the use of $\Theta(n^2)$ as a linear function cn for n <=3.
- The input list is split in half and recursively runs the recursive closest pair algorithm on each half
- The the total runtime for the recursive calls is T(n) = T(n/2) + T(n/2) + cn = 2T(n/2) + cn
- Using Master theorem, let a=2, b=2, and f(n) = cn
- $\log_2 2 = 1$

- cn = $\Theta(n)$, thus for the recursive calls T(n)= $\Theta(n\ lg\ n)$
- The algorithm iterates over every point once to calculate its distance to the midpoint and determine if it is a bisecting point, contributing n steps to the overall runtime.
  Let all points of the dataset be in the bisecting points list.
- For bisecting points list of size equal to n, the brute-force version of the algorithm is called.
- As stated in the brute-force algorithm analysis section, the brute-force algorithm has a running time complexity of $\Theta(n^2)$.
- Thus the overall worst-case time complexity is $\Theta(n\ lg\ n)$ +$\Theta(n\ lg\ n)$+ $\Theta(n)$ + $\Theta(n^2)$

Therefore the worst-case running time complexity is $O(n^2)$

The algorithm has a best-case time complexity of $\Omega(n\ lg\ n)$.

Proof:

- Before the main closest pair algorithm runs, the input list is sorted by x-value using merge sort.
  - Merge sort is a well-known sorting algorithm with a time complexity of $\Theta(n\ log\ n)$.
- When the size of the data set is <= 3, the algorithm performs the brute-force version of this algorithm.
  - Because $\Theta(n^2)$ is only used for n <=3, its rate does not grow as n grows.
  - Thus for the purposes of time complexity, we can treat the use of $\Theta(n^2)$ as a linear function cn for n <=3.
- The input list is split in half and recursively runs the recursive closest pair algorithm on each half
- The the total runtime for the recursive calls is T(n) = T(n/2) + T(n/2) + cn = 2T(n/2) + cn
- Using Master theorem, let a=2, b=2, and f(n) = cn
- $\log_2 2 = 1$
- cn = $\Theta(n)$, thus for the recursive calls T(n)= $\Theta(n\ lg\ n)$
- The algorithm iterates over every point once to calculate its distance to the midpoint and determine if it is a bisecting point, contributing n steps to the overall runtime.
  Let no point of the dataset be in the bisecting points list.
- The recursive closest pair algorithm immediately returns if the bisecting points list is empty.
- Thus an empty list of bisecting points does not contribute to runtime complexity.
- Thus the overall best-case time complexity is $\Theta(n\ lg\ n)$+$\Theta(n\ lg\ n)$+$\Theta(n)$

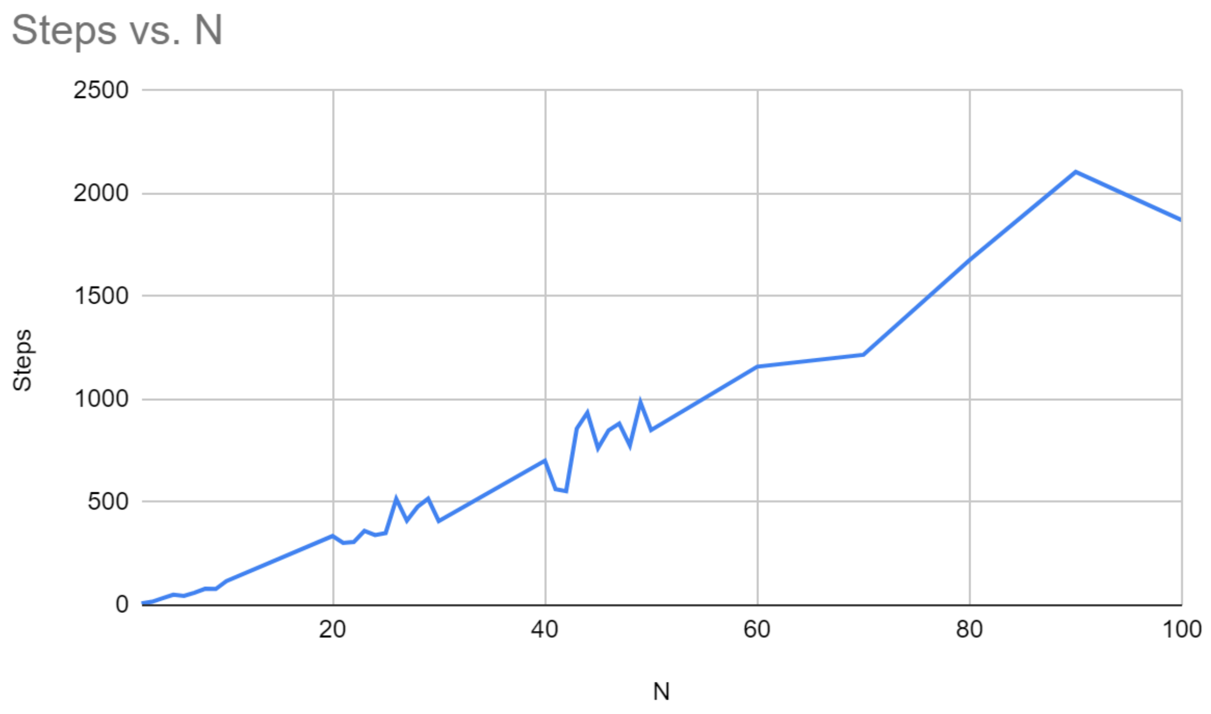
Therefore the best-case running time complexity is $\Omega(n\ lg\ n)$

Experiment and Findings (1.d)

In this experiment, I used the recursive algorithm to calculate the closest pairs for many randomized datasets of points. I utilized a counter in the code to count the number of steps the algorithm required for each dataset. Here, the number of steps is defined as the number of times the distance is calculated between two points, the number of times closest_pair_recursive is called, the number of times merge_sort is called, and the number of elements merge_sort accesses from a list.

As input, I generated randomizes lists of points of the following sizes: [2, 3, 4, 5, 6, 7, 8, 9, 10, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50, 60, 70, 80, 90, 100].

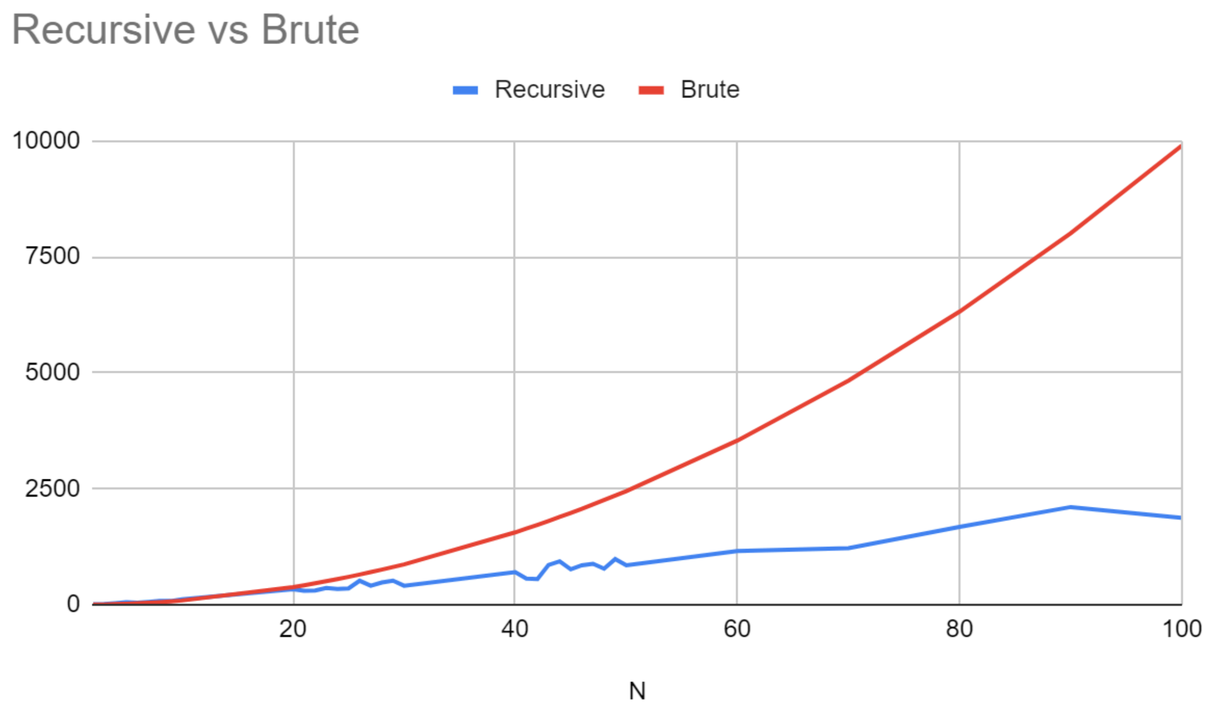
Below, I have graphed the relationship between each dataset's size and the number of steps it required to determine the closest pair.



Unlike the brute force algorithm, the number of steps required to determine the closest pair using the recursive algorithm can vary. It generally follows the trends from my theoretical performance expectations. Because the location of points is randomized, proximity to the midpoint can have a mild (in cases where some sublist of points are all near the midpoint) to substantial (in cases where all points are near the midpoint) impact on performance.

Because no value *n* from the dataset results in n^2 steps, I know that any calls of the brute-force closest pair method for points cluster near the midpoint must be on *sublists* of the dataset, not on the full dataset itself. Unfortunately, because datasets are randomized, it is difficult to determine the size of the sublist and/or if the brute-force method had to be called multiple times for any dataset.

Below are the brute and recursive experiment results, plotted on the same graph:



For datasets of size > 20, the recursive algorithm consistently performs better than the brute force algorithm. The only circumstance under which the brute force algorithm could perform better is if all the points in the dataset were close to the midpoint, requiring the algorithm to run both the recursive algorithm for the left and right halves of the sublist, as well as the brute force algorithm. For the majority of datasets, however, the performance of the recursive algorithm is considerably better.

Additional Optimizations (1.e)

Regarding the brute-force algorithm, which is utilized both independently and by the recursive closest pair algorithm, there is little I could do to improve the running time complexity. If everything must be compared to everything, the algorithm must be $\Theta(n^2)$.

I can, however, cut the runtime in half. My brute-force algorithm calculates the distance between any two points twice, once for the outer loop and once for the inner loop, i.e. it calculates both distance(p1, p2) and distance(p2, p1). Instead of iterating i=0->n for the outer loop and j=0->n for the inner loop, I could just iterate i=0->n-1 and j=i+1->n. That way, once an element has had its distance to every point calculated, it isn't calculated again.

Regarding my recursive algorithm, I suspect I could make substantial changes to improve the worst-case time complexity.

I suspect that, while my experiments demonstrated that the worst-case scenario is highly unlikely, there may be a way for me to eliminate calls to the brute-force algorithm for the bisecting points list entirely.

Admittedly I'm not sure how to assure that the bisecting points list doesn't cause an infinite loop when points are clustered near the midpoint. In my research of alternative implementations of the closest pairs algorithm, I found that some implementations did additional sorting on y-values for bisecting points lists. My own efforts to replicate this method successfully were not fruitful. Regardless, any means which removed the inclusion of the brute-force algorithm could improve the worst-case running time complexity from O(n^2) to $O(n\ lg\ n)$.

- This is under the assumption that typical use of the third recursive closest pairs call for the bisecting points list would add an additional $O(n\ lg\ n)$, bring the total worst-case running time from $\Theta(n\ lg\ n)$ +$\Theta(n\ lg\ n)$ + $\Theta(n^2)$ to $\Theta(n\ lg\ n)$ + $\Theta(n\ lg\ n)$ + $\Theta(n\ lg\ n)$

Deterministic Turing Machine (2)

Overview

For this exercise, I created a structure that emulates a Deterministic Turing Machine (DTM), comprised of:

- A mechanism for tracking the current state
- An array of characters, emulating a tape
- A mechanism for tracking the current tape position, emulating a read-write head
- A structure representing a turing transition function
- A mechanism for transitioning between positions on the tape

The transition function I implemented performs unary subtraction on an input string. The machine takes a string formatted <one or more 1s>#<one or more 1s>, calculates the difference between operands, and returns a string formatted <input string>#<difference, expressed as zero or more 1s> as the output.

Algorithm Description

In developing a deterministic turing machine (DTM) ruleset for unary subtraction, I quickly came to the realization that the head would have to move back and forth between the operands to calculate the difference. I required some method of identifying each operand, as well as the boundaries of the tape.

My algorithm for unary subtraction can be described as so:

1) Transition the tape forward to the first "1" of the second operand
    - Mark "1" as "X"
2) Transition the tape backward to the last "1" of the first operand"
    - Mark "1" as "X"
3) Move forward and backward across the tape until either operand has been completely converted to "X"s
4) Move backward to before the first operand
5) Move forward across the tape until a "1" is encountered
    - Mark "1" as "X"
6) Move forward across the tape until the second "#" after the second operand
    - Mark "#" as "1"
7) Move backward to before the first operand
8) Move forward across the tape until a "1" is encountered
    - Mark "1" as "X"
9) Move forward across the tape until the first "#" after the current result
    - Mark "#" as "1"
10) Repeat until both operands have been completely converted to "X"s
11) Move backward to before the first operand
12) Move forward until an "X" is encountered
    - Mark "X" as "1"
13) Continue until all "X"s have been converted into "1"s

14) Halt

With additional time, I feel confident that I could condense the transition ruleset. I did not, however, feel that it had a substantial impact on the total number of tape transitions, and as such, I leave that action for future investigation.

Formal 6-Tuple for Unary Subtraction (2.a)

I define the formal 6-tuple for unary subtraction below.
$M = (\Gamma, Q, s, \delta, \Sigma, b)$ where:

- Γ is the complete alphabet of symbols used,
  - {1, #, X, Y, S}
- Q is the set of DTM states,
  - {START, q0, q1, q2, q3, q4, q5, q6, q7, q8, q9, q10, q11, q12, q13, q14, q15, q16, q17, q18, q_Y, q_N}
- s is the set of tape head directions,
  - {left, right, halt}
- $\Sigma \subset \Gamma$ is the set of input symbols,
  - {1, #}
- $b \in (\Sigma \subset \Gamma)$ is the blank symbol, #
- δ is the transition function, defined by the following table:

Q-{q_A, q_R}	**1**	**#**	**X**	**Y**	**S**
START	{q0, S, 1}	{q_N, #, 0}	{q_N, X, 0}	{q_N, Y, 0}	{q_N, S, 0}
q0	{q0, 1, 1}	{q1, #, 1}	{q_N, X, 0}	{q_N, Y, 0}	{q_N, S, 0}
q1	{q3, X, -1}	{q_N, #, 0}	{q2, X, 1}	{q_N, Y, 0}	{q_N, S, 0}
q2	{q3, X, -1}	{q6, #, -1}	{q2, X, 1}	{q_N, Y, 0}	{q_N, S, 0}
q3	{q3, X, -1}	{q4, #, -1}	{q_N, X, 0}	{q_N, Y, 0}	{q_N, S, 0}
q4	{q5, X, 1}	{q7, #, 1}	{q4, X, -1}	{q_N, Y, 0}	{q7, X, 1}
q5	{q_N, 1, 0}	{q1, #, 1}	{q5, X, 1}	{q_N, Y, 0}	{q_N, S, 0}
q6	{q_N, 1, 0}	{q7, #, -1}	{q6, X, -1}	{q_N, Y, 0}	{q_N, S, 0}
q7	{q7, 1, -1}	{q8, Y, 1}	{q7, X, -1}	{q_N, Y, 0}	{q7, 1, -1}
q8	{q9, X, 1}	{q_N, #, 0}	{q8, X, 1}	{q_N, Y, 0}	{q_N, S, 0}
q9	{q9, 1, 1}	{q11, #, 1}	{q9, X, 1}	{q_N, Y, 0}	{q_N, S, 0}
q10	{q11, X, 1}	{q14, #, -1}	{q10, X, 1}	{q_N, Y, 0}	{q_N, S, 0}
q11	{q11, 1, 1}	{q12, #, 1}	{q11, X, 1}	{q_N, Y, 0}	{q_N, S, 0}

q12	{q_N, 1, 0}	{q13, X, -1}	{q12, X, 1}	{q_N, Y, 0}	{q_N, S, 0}
q13	{q_N, 1, 0}	{q14, #, -1}	{q13, X, -1}	{q_N, Y, 0}	{q_N, S, 0}
q14	{q11, X, 1}	{q15, #, -1}	{q14, X, -1}	{q_N, Y, 0}	{q_N, S, 0}
q15	{q9, X, 1}	{q15, #, -1}	{q15, X, -1}	{q16, #, 1}	{q_N, S, 0}
q16	{q_N, 1, 0}	{q7, #, 1}	{q16, 1, 1}	{q_N, Y, 0}	{q_N, S, 0}
q17		{q18, #, 1}	{q17, 1, 1}		
q18		{q_Y, #, 0}	{q18, 1, 1}		

Analysis

The unary subtraction algorithm has a $\Theta(n)$ space complexity.
Proof:

- In simulation of a DTM's tape mechanism, the algorithm writes over elements of the tape to calculate the unary difference
- The output of the tape must include the input
- Thus the tape must contain at least n cells, where n is the length of the input string
- The algorithm only writes to new blank cells when writing the result of the unary subtraction
- If the machine rejects an input string, no new blank cells are written to
- In order to conform to the input format, each operand must consist of at least one "1"
- In order to conform to the input format, exactly one character of the input string must be "#"
  Let one operand be the string "1"
- The other operand must consist of exactly n-2 "1"s
- Thus, the length of the unary difference can be, at most, n-2-1=n-3
- The DTM also writes a blank symbol to separate the second operand from the difference
- Thus, the total length of the tape when the DTM halts must be between n and n+1+n-3 cells long.
- Thus the unary subtraction algorithm has a $\Theta(2n - 2)$ space complexity

Therefore, the unary subtraction algorithm has a $\Theta(n)$ space complexity.

The unary subtraction algorithm has a $O(n^2)$ worst-case runtime complexity.
Proof:

- The algorithm transitions to the center of the tape
- The algorithm transitions to the first cell of the second operand
- The algorithm moves back and forth across the tape, adding an additional character that must be transitioned across for each "X" that is written
- This can take at most, if operand1=operand2, n-1 back and forth transitions
- Because each transversal adds one additional character, we can approximate the total number of transitions to be the triangle number $\frac{n(n-1)}{2}$

- After all the the “1”s have been converted for one operand, the head transitions back to the beginning of the tape
- The head continues to transition all the way across the original input string
- The head continues to transition until it writes a “1” to the result
- The head continues to transition back and forth until all “1”s have been converted into “X”s and the complete difference has been written.
- The head then moves back to the beginning of the tape again and converts all the “X”s back to “1”s.
- Moving from the beginning of the tape to the end of the tape and then back takes 2n transitions
- The tape is moved back and forth for each non-”#” element in the original input string, across a total of n-1 cells
- Thus, moving back and forth for each element of the input string takes 2n(n-1) transitions
- Additionally, the tape must move from the beginning of the result to the end of the result and then back to the beginning of the result
- The size of this result grows by 1 for each transition into the result second of the tape
- Thus we can express the number of transitions across the result space as the triangle number $\frac{r(r+1)}{2}$ where r is equal to the length of the result, such that $o \leq r \leq n - 3$
- Thus the number of transitions within the result space can be, at most $\frac{(n-3)(n-2)}{2}$
- Then the head transitions across the total length of the tape, at most across 2n-2 cells back to the start of the tape
- Then the head transitions from the start of the tape to the end of the second operand, transitioning exactly n times
- Thus the total number of transitions can be approximated as $\frac{n(n-1)}{2}$ + 2n(n-1) + $\frac{(n-3)(n-2)}{2}$ + (2n-2) + n
- Thus the total number of transitions can be approximated as $\frac{n^2-n}{2}$ + $2n^2$-2n + $\frac{n^2-5n+6}{2}$ + 2n-2 + n

Therefore, the unary subtraction algorithm has a O(n^2) worst-case runtime complexity.

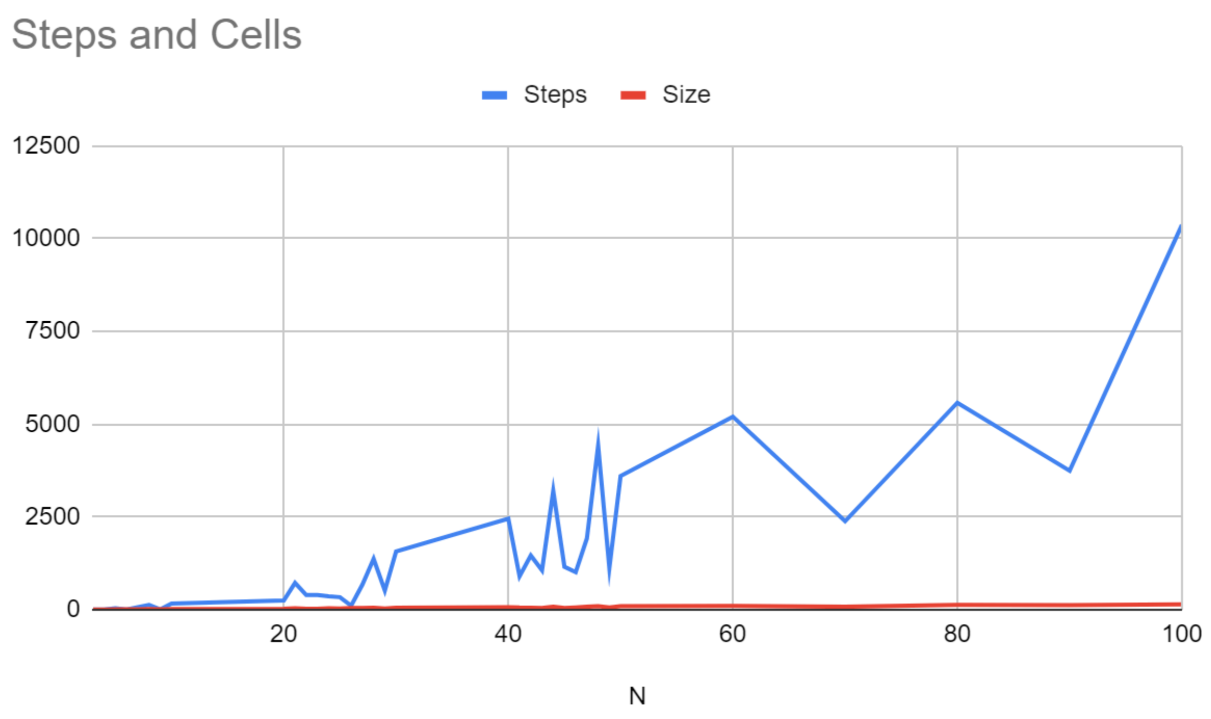
Experiment and Findings

In this experiment, I used the DTM to calculate the unary difference for many randomized datasets of input strings. I utilized a counter in the code to count the number of steps the algorithm required for each dataset. Here, the number of steps is defined as the number of transitions between positions on the tape. I also measured the size of the tape after the machine halted.

As input, I generated randomizes lists of points of the following sizes: [3, 4, 5, 6, 7, 8, 9, 10, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50, 60, 70, 80, 90, 100].

For example:

- For size 7, input = “11#1111”
- For size 10, input = “1111#11111”



Whereas the number of steps increased substantially as the dataset size n increased, the total number of tape cells used never exceeded. From this, I concluded that I had a very efficient utilization of space. The number of transitions grew at the rate I expected. With further optimization, I believe this rate could be substantially reduced.

Additional Optimizations

There are several simple ways the transition function could be modified to improve performance.

- Use “X”s for the left operand, use “Y”s for the right operand
  - This would allow the removal of states that exist exclusively to track the current operand location.
- Don’t traverse the entire tape unless necessary
  - Design transition function rules to mark the outer edges (i.e. left side of the first operand and right side of the second operand) first. As cells are marked, the length the head must traverse will shrink.
  - Then we can transition on the midpoint “#” to a state for retrieving and writing the result, regardless of which operand has more “1”s
- Use “-” for the middle “#”
  - This cell has unique properties. Distinguishing it from the empty cells outside the current tape space could be useful for reducing the number of states.