CLOSEST PAIR OPTIMIZATION ON MODERN HARDWARE

By

Jason Bright

A Project Submitted in Partial Fulfillment of the Requirements

for the Degree of

Master of Science

in

Computer Science

University of Alaska Fairbanks

May 2019

Abstract

In this project we examine the performance of several algorithms for finding the closest pair of points out of a given set of points in a plane. We look at four algorithms, including brute force, recursive, non-recursive, and a random expected linear time for numbers of points ranging from one hundred to one billion. In our examination, we find that on average the non-recursive is the fastest, except for limited cases of 100 points for the brute force, and 32 bit spaces for the random expected linear.

Table of Contents

Introduction

Closest Pair Optimization on Modern Hardware

There are many problems involving optimization with modern hardware. Extensive pipelining of code execution, preprocessor rewriting, and other technologies result in variable overhead where different algorithms and variations on algorithms have different execution times, often in unpredictable ways. The difference between no optimization and moderate optimization in the compiler can result in two orders of magnitude difference in code execution.

When considering different algorithms for solving a problem, it is important to realize that selecting the lowest Big-O algorithm is not always the best solution. The overhead for a constant time algorithm, for example, can be such that for smaller problems, even an $O(n^2)$ solution may be superior to an $O(n)$ solution, for time or memory requirements. This is the classic reason why quick sort is so frequently used. While its worst case performance is $O(n^2)$, the expected performance is generally much better at $O(n \log n)$.

The Closest Pair Problem is a classic problem where you are given a set of points and are asked to find the pair of points with the minimum distance between them.

The points are generally described using a grid coordinate system, using two numbers to describe the distance in a 2-dimensional plane along two perpendicular lines. Variants like polar coordinates are possible, where a direction is described using one number, whether radians or degrees, and a distance along the direction's line by another. A third dimension can be added by adding a third number, where the third number describes a distance along a line perpendicular to the plane.

A cost function is described to determine the distance between each pair of points. This is normally Pythagorean, $D=\sqrt{a^2+b^2}$, though for comparison purposes, the square root can be dispensed with, so the function is $D=a^2+b^2$. For different problems, a cost function can be used to provide different numbers for comparing the distance between points. One simple alternative would be simply $D=|a|+|b|$, which is otherwise known as the Manhattan, or taxicab, metric. This metric considers that if one is driving in a city laid out in a grid, then the direct route is rarely available, so the number of blocks north + east, or their inverse, is the actual distance that will be driven. For straight line travel, it will over-estimate distance by up to 41%, but will never give an answer less than the pythagorean straight line distance. As it removes the need for squaring and square roots, in cases where being within 40% of the expected answer at worst is acceptable, it can result in substantial savings in computing effort.

Another alternative example could be that the points describe positions on a globe. Thus, rather than measuring a straight line distance between the points, the cost function would be the arc distance between the two. This is known as the Great-Circle distance, commonly used for calculating aircraft routes and travel distances.

A final alternative would be a cost function that considers actual travel costs between points: a function that calculates a route between the points and assigns a cost on the basis of road speed, stop lights, stop signs, and anticipated road traffic, for example. It could be given by a lookup table that holds the time between points. There are infinitely many possibilities for the cost function.

Algorithms that solve the closest pair problems are useful for optimization, certain heuristics, matching, and other uses.

Project Approach and Methodology

By writing several sorting algorithms and testing them with a number of sizes of test point sets, we can benchmark the performance of the algorithms and determine the size ranges at which each dominates.

Algorithms

Brute Force

This algorithm is written as simply as possible. Using two loops, the outer checks points 0 to n-1, known as ‘i’ and the inner loop i to n. For each pair of points, the distance is checked against the closest pair yet found. If the distance is less, the new pair replaces the old closest pair. The loops continue until finished. This results in n(n-1)/2 calculations

Recursive

This method of finding the closest pair works by sorting the list of points, then splitting the list in half an recursing on the two parts. It recurses until less than four points remain for the comparison, then reverses back. After returning from the recursions, it checks the points between the two sets. If a pair of points at zero distance are found, it ceases and returns. See Closest Pair Problem(Subhash Suri).

Pseudo-code:

```
Sort(testSet) by first element
n = sizeOf(testSet)
closePoint0 = testSet[0]
closePoint1 = testSet[1]
dSmall = dist(p0,p1)
recurseFunction(testSet, 0, n, closePoint0, closePoint1, dSmall)
recurseFunction(& testSet, start, end, & closePoint0, & closePoint1,& dSmall)
    if dSmall == 0
        return
    if (end – start) < 4
        for(i = start; i < end -1; ++i)
            for(j = start+1; j < end; ++j)
                if dist(testSet[i],testSet[j]) < dSmall
                    closePoint0 = testSet[i]
                    closePoint1 = testSet[j]
                    dSmall = dist(testset[i],testSet[j])
            next j
        next I
    else
        midIndex = start + (end - start)/2
        recurseFunction(testSet, start, midIndex, closePoint0, closePoint1, dSmall)
        recurseFunction(testSet, midIndex, end, closePoint0, closePoint1, dSmall)
        for(i=midIndex-1;i >= 0;--i)
            if testSet[midIndex] - testSet[i].first > dSmall
                break
            for(j=midIndex; j < end; ++j)
                if testSet[j].first – testSet[i].first > dSmall
                    break
                if dist(testSet[i],testSet[j]) < dSmall
                    closePoint0 = testSet[i]
                    closePoint1 = testSet[j]
                    dSmall = dist(testset[i],testSet[j])
            next j
        next i
```

Non-Recursive

The non-recursive algorithm(Levitin) works by sorting the list along one axis (x). It then calculates the distance between the first two points. After this, it checks the distance towards each subsequent point until the x distance exceeds that of the smallest total distance yet found. Once the x distance exceeds the smallest distance yet found, the algorithm moves to the next point in the list for the first, and continues checking distances to subsequent points. This is essentially an optimized brute force search.

Pseudo-code:

```
Sort(testSet) by first element
n = sizeOf(testSet)
closePoint0 = testSet[0]
closePoint1 = testSet[1]
dSmall = dist(p0,p1)
i = 0
for(i=0; i < n-1; ++i)
    for(j=i+1; j < n; ++j)
        if (testSet[i][0] + dSmall < testSet[j][0])
            break
        if dist(testSet[i],testSet[j]) < dSmall
            closePoint0 = testSet[i]
            closePoint1 = testSet[j]
            dSmall = dist(closePoint0, closePoint1)
            if dSmall == 0
                return closePoint0, closePoint1
    next j
next i
return closePoint0, closePoint1
```

Reliable Randomized Expected Linear

An algorithm proposed by Dietzfelbinger, Hagerup, and Katajainen. While not a true linear time algorithm, this algorithm is expected to be linear time on average with large enough data sets. It works by sorting points into cells using a linear sorting algorithm that is a variety of bucket sort, based on the x/y distance of the point from zero. The size or width of the bucket is dependent upon the closest distance found by using a random sample of points. Then the closest pair of points is determined for each bucket and its neighboring buckets. In this way, only small subsets of points ever need to be sorted or compared using algorithms that are higher than linear. As only the resulting pairs need to be compared, the expected order is linear; however if the points are not evenly distributed, then all points could end up in one bucket, thus making the order the same as the underlying algorithm for sorting within a bucket, creating a worst case of O(n log n). As an additional optimization, if two points are found having the same coordinates, a distance of zero, the algorithm will halt and return those points.

`Pseudo-code:`

```
Select a set of random points and find closest pair within set
If zero distance found, return pair
Distance between points becomes w
Bucket width set at w
Each point assigned to int(x/w) bucket
Each bucket and its neighbors(-1/+1) are searched for the closest point.
If zero distance found, halt and return pair
return closest pair found from all buckets
```

Experiment details

Hardware:

- Intel Core i7-6700K at 4 GHz
- 32 GB of RAM
- Motherboard: Gigabyte Z170XP-SLI

Software:

- Windows 10
- Cygwin 2.11.2
- Compiler: GCC 7.3.0,
- Code C++11
- Optimization: O3

The maximum reasonable size for N on this machine using 32 or 64 bit integers to remain in RAM is approximately 1 billion. The practical maximum depends upon the algorithm’s memory efficiency, and will therefore be different for each.

Test sets: Points were generated using the std::mt19937_64 pseudorandom number generator in the C++ standard library with known seeds to generate a number of differing sets. Because only the brute force algorithm is stable, not changing the order of the point list, this allows each algorithm to be tested against identical data sets by regenerating the test set for each algorithm. The std::uniform_int_distribution was used for integer test sets, std::uniform_real_distribution was used for floating point test sets. Both distributions were from the C++ standard library.

The range for each experiment was set to the square root of the limit of the variable type in question divided by 2, or $\sqrt{\lim}/2$. This was to remove the need to promote variables to the next larger type, while still preventing over-runs that would lead to inaccurate distance results. Each point was stored in a std::pair structure.

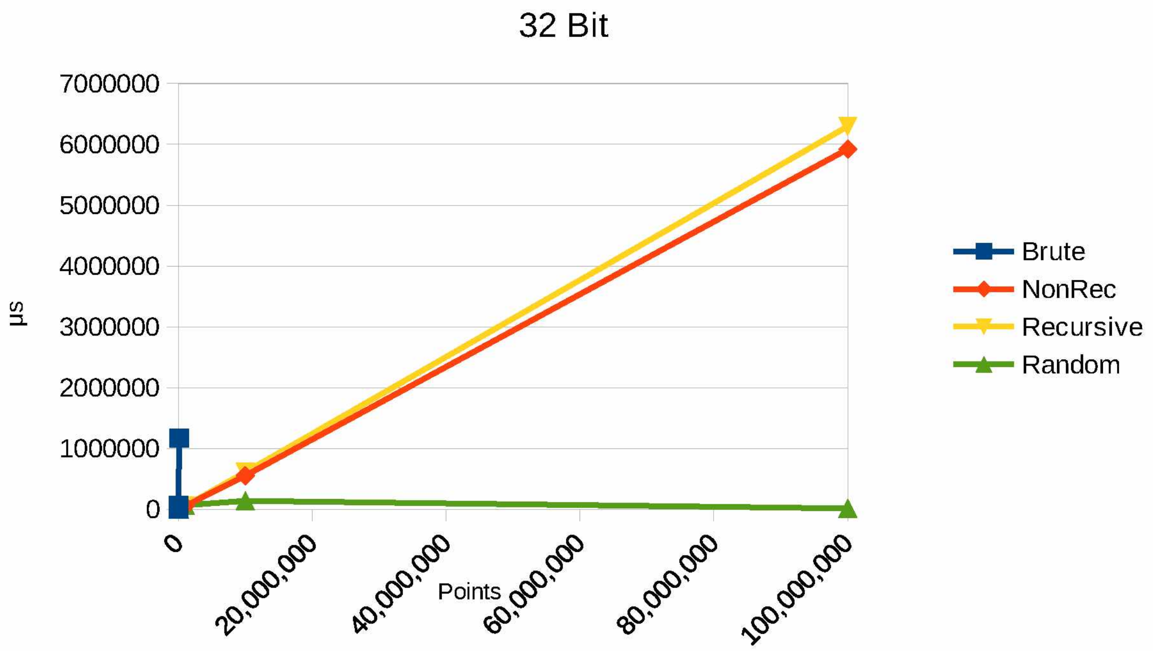
For each data set size, multiple runs were conducted, such that 1 billion points would always be sorted, with the times totaled then averaged, to give more consistent results. Sets under 100 points were not tested because accurate timing could not be collected for sets that small.

All data sets were created before timing starts. After timing starts, each test set was searched using the selected algorithm. Once finished, the timing was stopped and the metrics – time taken, average, maximum and minimum distance were calculated and reported. The data sets were then regenerated for the subsequent algorithm.

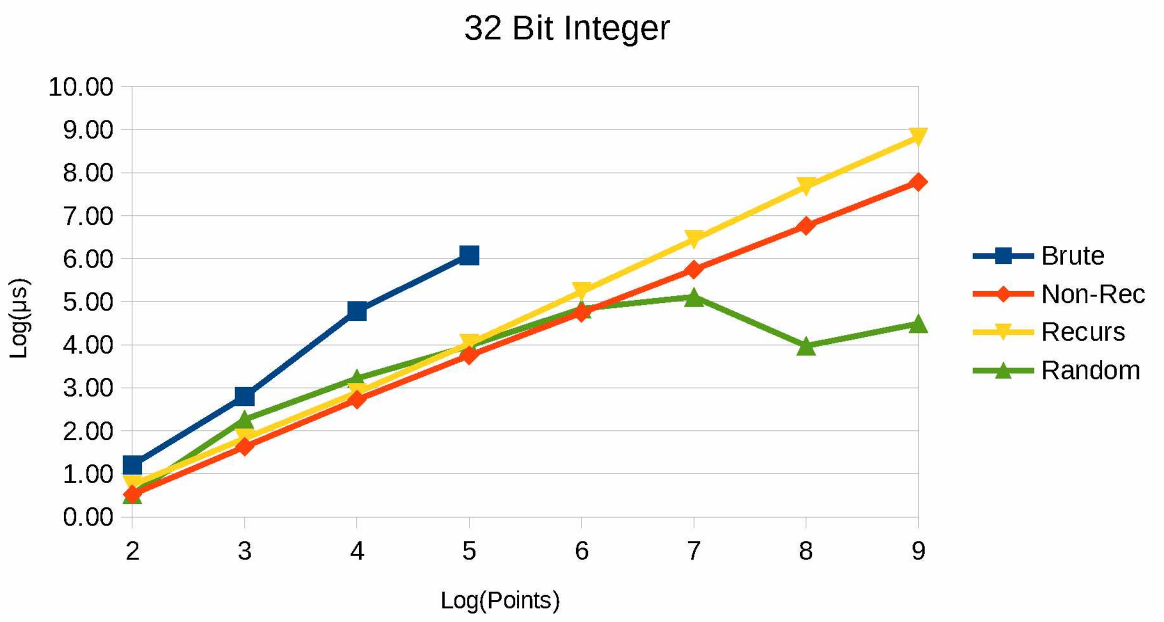
For testing behavior when the closest pair was guaranteed to be at zero distance, no action was necessary for the larger 32 bit data sets, as zero distances were present for all sets of one million or more points. For 64 bit data sets, a zero distance closest pair was guaranteed by selecting two points in the data set by using the std::uniform_int_distribution, then setting one equal to the other, after the data set had been generated.

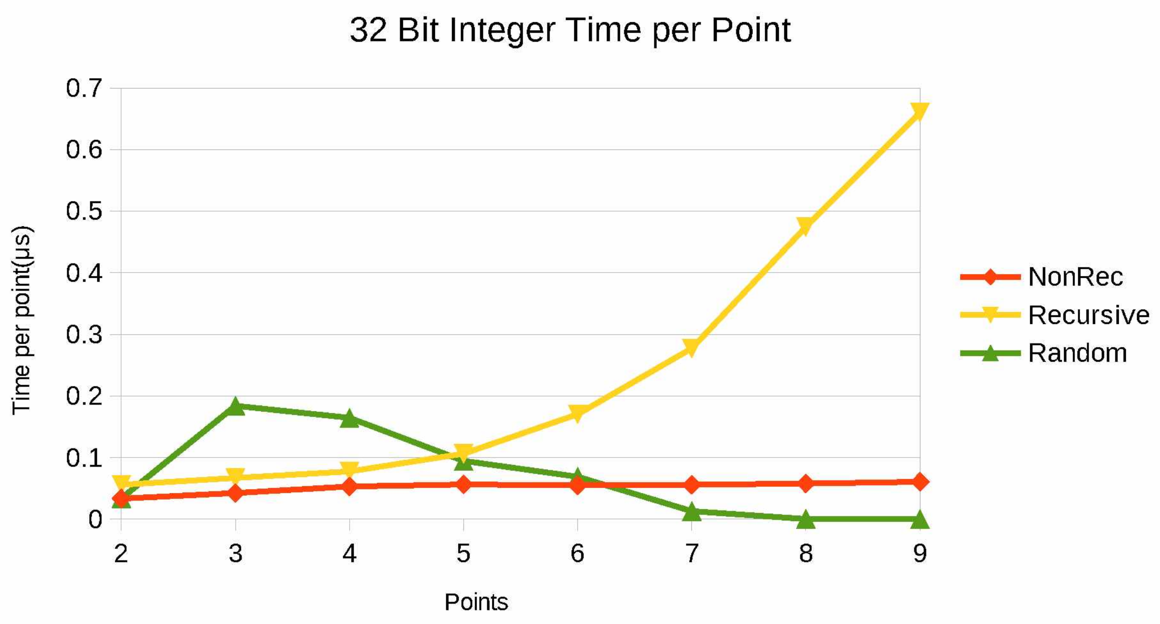
Results

Direct graphing of the results proved less than useful, as times increased so quickly. As such, graphs after this one feature logarithmic scaling for the x axis.

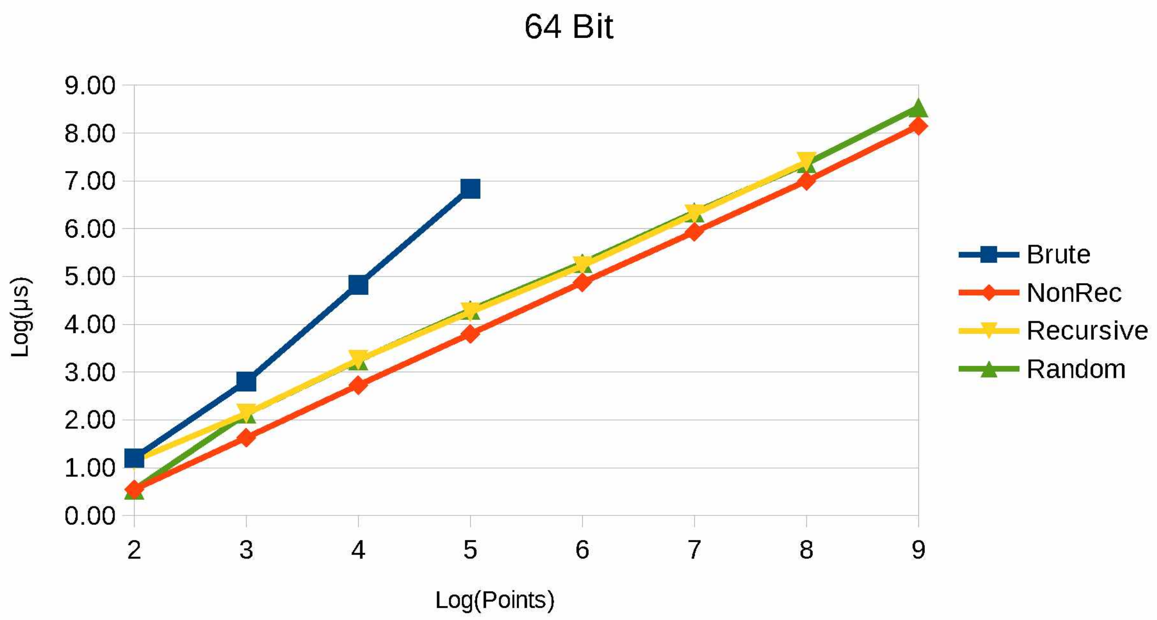


For 32 bit operations, we can see that for under 100 points, the brute force algorithm is actually the fastest, but by 1000 points it is the slowest. The Non-Recursive algorithm maintains a slight lead beyond that until 10^6 points, at which point the random expected linear takes over as the fastest.

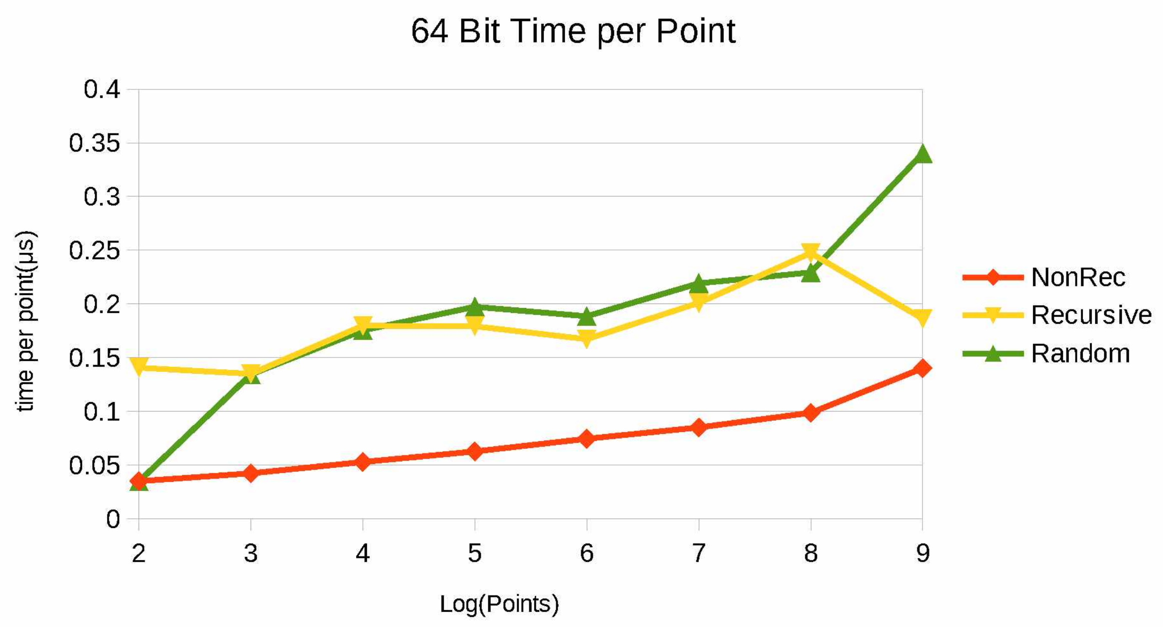




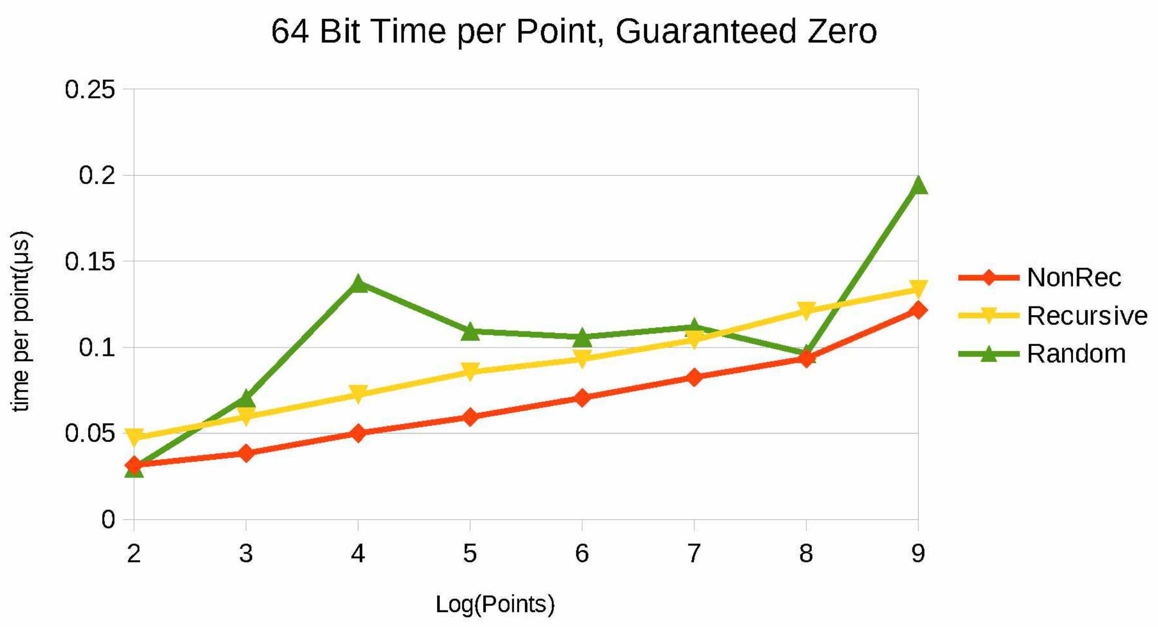
The perforanace of the non-Recursive and random expected linear is the same at 100 and 1,000,000 points. After this, the Reliable Randomized algorithm quickly became the better performing. The recursive algorithm increased faster than would be expected for O(n log n).



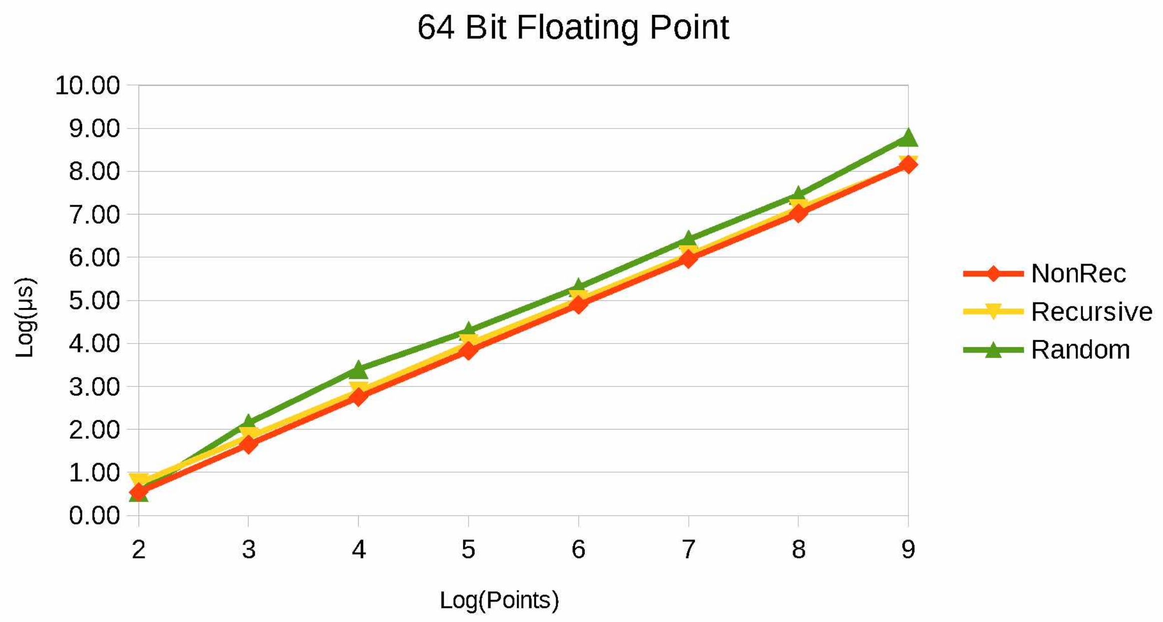
For 64 bit integers, we don’t see the large variances that were seen with 32 bit integers.

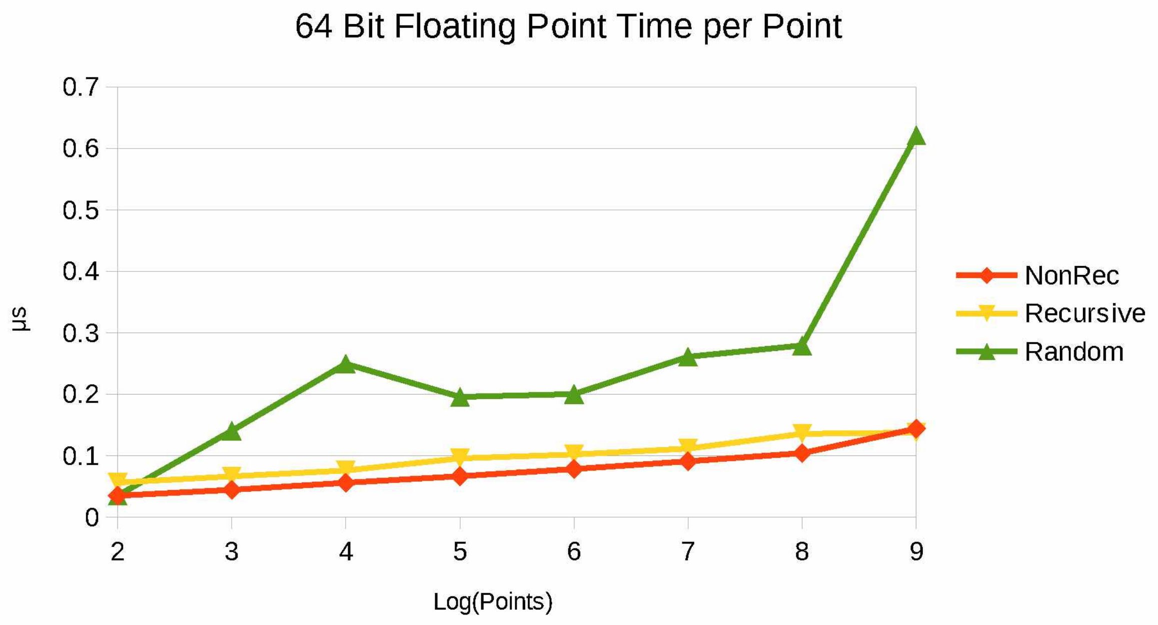


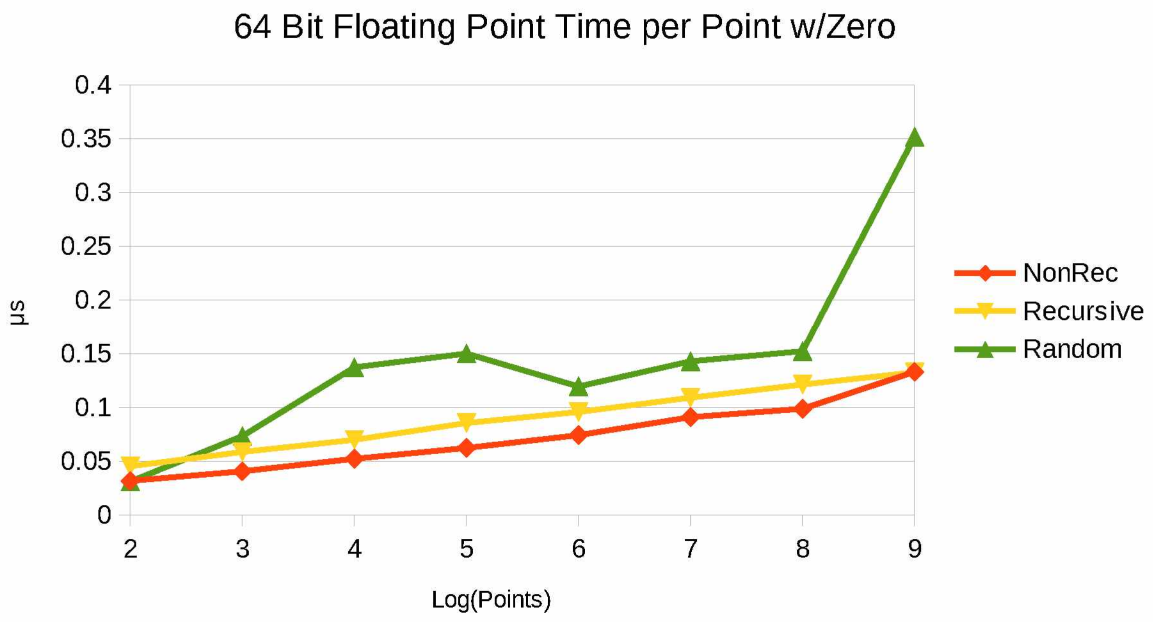
When we look at the time taken per point, we do not see the Reliable Randomized algorithm becoming faster, at least at the test set sizes tested in this experiment. The non-recursive algorithm is the fastest choice over the entire field.



When a zero is guaranteed to be in place, execution time is reduced roughly by half. Noticeably, after peaking, the reliable random algorithm evens out at approximately 0.1 ms per point in the data set until it reaches 10^9 points, at which point the time necessary spikes.







The behavior of 64 bit floats is effectively the same with 64 bit Integers.

Table 1: Minimum Distance, 32 Bit Integer

Points	Average	Max	Min	Trials
1E+02	232.60	1,028.00	0.00	1E+07
1E+03	22.82	105.00	0.00	1E+06
1E+04	2.09	8.00	0.00	1E+05
1E+05	0.01	1.00	0.00	1E+04
1E+06	0.00	0.00	0.00	1E+03
1E+07	0.00	0.00	0.00	1E+02
1E+08	0.00	0.00	0.00	1E+01
1E+09	0.00	0.00	0.00	1E+00

An interesting quirk of the system is the minimum distance between points found for each test set as the size of the test set increased. This value is actually fairly predictable. With 32 bit numbers, at 100 pairs the average distance between points was 233. When this was increased to 1,000, the average decreased an order of magnitude to 23 units. At 100,000 units, the maximum distance found was 1 unit, and the average was 0. This was the point where the random expected caught up with the non-recursive algorithm.

With 64 bit integers, the average distance between the closest points starts at 15 million, and decreases linearly with the number of points, and zero distance pairs are not seen.

64 bit floating point numbers were all in the $10^{144} – 10^{151}$ range. While it decreased substantially as the test set size increased, by a relatively consistent order of magnitude with each order of magnitude the number of points increased, the distances between points was always very large.

Table 2: Minimum Distance, 64 Bit Integer

Points	Average	Max	Min	Trials
1E+02	15,274,100	67,355,600	9,709	1E+07
1E+03	1,520,730	6,835,410	1,753	1E+06
1E+04	151,987	573,547	987	1E+05
1E+05	15,138	52,023	288	1E+04
1E+06	1,533	3,753	68	1E+03
1E+07	147	451	24	1E+02
1E+08	16	29	7	1E+01
1E+09	1	1	1	1E+00

Table 3: Minimum Distance, 64 Bit Float

Points	Average	Max	Min	Trials
1E+02	6.7E+151	3.0E+152	4.3E+148	1E+07
1E+03	6.7E+150	3.0E+151	7.7E+147	1E+06
1E+04	6.7E+149	2.5E+150	2.5E+150	1E+05
1E+05	6.7E+148	2.3E+149	1.0E+147	1E+04
1E+06	6.8E+147	1.7E+148	3.0E+146	1E+03
1E+07	6.5E+146	2.0E+147	1.1E+146	1E+02
1E+08	7.3E+145	1.3E+146	3.3E+145	1E+01
1E+09	5.9E+144	5.9E+144	5.9E+144	1E+00

Conclusion

From the testing, it appears that conventional non-recursive sorting is the best general case scenario. Brute force is sometimes better at 100 point test sets, while recursive is worse than the non-recursive in most scenarios. For test sets of limited area, as seen in the 32 bit, the Random Expected Linear can pull ahead, presumably due to being able to find a pair of zero distance points more efficiently. This is also seen with 64 bit numbers when a zero distance point is included in the test sets, though this took 100 million points, close to the maximum the test machine can handle, before catching up with the non-recursive. The exponential increase in execution time for 32 bit integers using the recursive algorithm was not expected. It appears that there are excessive cache misses and poor optimization on a 64 bit architecture with the algorithm.

It appears that while our implementation of the Random Expected Linear algorithm has too much overhead to be the algorithm of choice in most cases, it did achieve linearity at times, within the limits of our ability to test using sets limited to the memory of the test computer. Further optimization may improve the algorithm to be more useful. At 10^9, or one billion, points the algorithm sharply increases in execution time. Due to the differences from the other algorithms, this is likely due to it hitting memory limitations and needing to cache to disk.

The recursive algorithm would eventually become very close, performance wise, to the non-recursive formula at 32 bits starting at one million points, but it never became faster, leaving no real use for it. The Random Expected Linear algorithm performed somewhat better with 64 bit floating point numbers as opposed to 64 bit integers, while the other algorithms performed essentially identically.

Discussion and future work

The testing here was for uniformly distributed points, sort of like an even star field. Clustered points would probably work the same, but restricting the range of the points more, increasing effective density, might change results.

The algorithms used in this paper could be optimized further. For example, for the recursive algorithm, it does a brute force search once it reduces the number of points in the set down to a set amount. This was set at less than 4 points, however, increasing this number to 10 or even 20 improves the performance.

There are other closest pair algorithms that could be implemented and tested. Continuing on to test the traveling salesman problem would be interesting.

References

Dietzfelbinger, M., Hagerup, T., Katajainen, J., & Penttonen, M. (1997). A Reliable Randomized Algorithm for the Closest-Pair Problem. Journal of Algorithms, 25(1), 19-51. doi:10.1006/jagm.1997.0873

Kamousi, P., Chan, T. M., & Suri, S. (2011). Closest Pair and the Post Office Problem for Stochastic Points. Computational Geometry, 47(2), b, 214-223. doi:https://doi.org/10.1016/j.comgeo.2012.10.010

Levitin, A. (2012). Introduction to the design and analysis of algorithms. Boston: Pearson.

Suri, S. (n.d.). Closest Pair Problem (Tech.). Retrieved from https://sites.cs.ucsb.edu/~suri/cs235/ClosestPair.pdf