Programming Assignment 1
Peter Rasmussen
October 18, 2021

Statement of Integrity: I, Peter Rasmussen, attempted to answer each question honestly and to the best of my abilities. I cited any and all help that I received in completing this assignment.

1. Conditions

(a) Consider a set, P, of points, $(x_1, y_1), \dots, (x_n, y_n)$, in a two-dimensional plane.
(b) A metric for the distance between two points (x_j, y_j) in this plane is the Euclidean distance $\sqrt{(x_i - x_j)^2 + (y_i - y_j)^2}$.

2. Closest Pairs

(a) Construct an algorithm for finding the $m \leq \binom{n}{2}$ closest pairs of points in P. Algorithm inputs are P and m. Return distances between the m closest pairs of points, including their x and y coordinates.

[Analysis is provided on following pages]

2(a)(i). Define your algorithm using pseudocode.

We use the FIND-NEAREST-PAIRS algorithm to return the distances of the m closest pairs of points, including their x and y coordinates. FIND-NEAREST-PAIRS is comprised of the following steps:

1. COMPUTE-DISTANCES: Compute the distance between each point pair
2. HEAP-SORT: Sort the point pairs using heap sort.
3. SELECT-M-POINTS: Select m point pairs with the smallest distances.

Let n be the length of P.

FIND-NEAREST-PAIRS(P, m)	$\boldsymbol{O}(\cdot)$
`P = COMPUTE-DISTANCES(P)`	$O(n^2)$
`HEAP-SORT(P)`	$O(n \lg n)$
`SELECT-M-POINTS(P)`	$O(1)$

We define each sub-algorithm – COMPUTE-DISTANCES, HEAP-SORT, AND SELECT-M-POINTS – in turn.

COMPUTE-DISTANCES

The COMPUTE-DISTANCES function, which calls the COMPUTE-DISTANCE function, computes the distances between each point pair in P.

COMPUTE-DISTANCES(P)	**cost**	**times**
`UNSORTED_P = []`	c1	1
`for i=1 to P.length`	c2	$1+n$
`    for j=i+1 to P.length`	c3	$1+\sum_{j=2}^{n} n-j+1$
`        distance = COMPUTE-DISTANCE(P[i], P[j])`	c4	$\sum_{j=2}^{n} n-j+1$
`        row = [P[i], P[j], distance]`	c5	$\sum_{j=2}^{n} n-j+1$
`        UNSORTED_P.append(row)`	c6	$\sum_{j=2}^{n} n-j+1$
`return UNSORTED_P`	C7	1

The COMPUTE-DISTANCE function computes the distance between a pair of points.

COMPUTE-DISTANCE(p1, p2)	**cost**	**times**
`x1, y1 = p1`	c1	1
`x2, y2 = p2`	c2	1
$d = ((x1 - x2)^2 + (y1 - y2)^2)^{0.5}$	c3	1
`return d`	c4	1

This completes the definition of the COMPUTE-DISTANCES algorithm.

HEAP-SORT

We then sort the list of points using HEAP-SORT (Cormen, Leiserson, Rivest, & Stein, 2009, p. 160), which sorts the points in $O(n \lg n)$ time. HEAP-SORT calls BUILD-MAX-HEAP and MAX-HEAPIFY, which we define further below.

HEAP-SORT(P)	**cost**	**times**
`BUILD-MAX-HEAP(P)`	c1	O(n)
`for i = P.length downto 2`	c2	n
`    exchange P[1] with P[i]`	c3	n-1
`    P.heap-size = P.heap-size – 1`	c4	n-1
`    MAX-HEAPIFY(P, 1)`	c5	O(lg n)

BUILD-MAX-HEAP (CLRS, 2009, p. 157) is defined below.

BUILD-MAX-HEAP(P)

```
P.heap-size = P.length
for i = ⌊P.length/2⌋ downto 1
    MAX-HEAPIFY(P, 1)
```

MAX-HEAPIFY (CLRS, 2009, p. 154), which calls functions LEFT and RIGHT, is defined below.

MAX-HEAPIFY(P, i)	cost	times
`l = LEFT(i)`	c1	1
`r = RIGHT(i)`	c2	1
`if l <= P.heap-size and P[l] > P[i]`	c3	1
`    largest = l`	c4	1
`else largest = i  // cost of 0 in worst case`	0	0
`if r <= P.heap-size and P[r] > P[largest]`	c6	1
`    largest = r`	c7	1
`if largest != i`	c8	1
`    exchange P[i] with P[largest]`	c9	1
`    MAX-HEAPIFY(P, largest)`	c10	T(2n/3)

Finally, the LEFT and RIGHT functions (CLRS, 2009, p. 152) are each defined below.

LEFT(i)

```
return 2i
```

RIGHT(i)

```
return 2i + 1
```

This completes the definition of the HEAP-SORT algorithm.

SELECT-M-POINTS

The SELECT-M-POINTS algorithm is defined below.

SELECT-M-POINTS(P, m)

```
return P[:m]
```

2(a)(ii). Determine the worst-case running time of your algorithm.

The worst-case running time of **FIND-NEAREST-PAIRS(P, m)** is $O(n^2)$, which is explained below.

COMPUTE-DISTANCES

The definition of COMPUTE-DISTANCES in part *2ai* includes the cost and number of times of each line of pseudo code. $T(n)$ is the sum of the costs of each line of pseudo code.

$T(n) = c_1 + c_2(1+n) + c_3\left(1 + \sum_{j=2}^{n} n - j + 1\right) + c_4\left(\sum_{j=2}^{n} n - j + 1\right) + c_5\left(\sum_{j=2}^{n} n - j + 1\right) + c_6\left(\sum_{j=2}^{n} n - j + 1\right) + c_7$

$T(n) = (c_1 + c_2 + c_3 + c_7) + (c_2 n) + (c_3 + c_4 + c_5 + c_6)\left(\sum_{j=2}^{n} n - j + 1\right)$

$T(n) = (c_1 + c_2 + c_3 + c_7) + (c_2 n) + (c_3 + c_4 + c_5 + c_6)\left(\sum_{j=2}^{n} n - \sum_{j=2}^{n} j + \sum_{j=2}^{n} 1\right)$

$T(n) = (c_1 + c_2 + c_3 + c_7) + (c_2 n) + (c_3 + c_4 + c_5 + c_6)\left(n(n-1) - (\frac{1}{2}n(n+1) - 1) + (n-1)\right)$

$T(n) = (c_1 + c_2 + c_3 + c_7) + (c_2 n) + (c_3 + c_4 + c_5 + c_6)\left(n^2 - n - (\frac{1}{2}n^2 + \frac{1}{2}n - 1) + (n-1)\right)$

$T(n) = (c_1 + c_2 + c_3 + c_7) + (c_2 n) + (c_3 + c_4 + c_5 + c_6)\left(\frac{1}{2}n^2 - \frac{1}{2}n\right)$

$T(n) = (c_1 + c_2 + c_3 + c_7) + (c_2 n) + \frac{1}{2}(c_3 + c_4 + c_5 + c_6)(n^2 - n)$

$T(n) = (c_1 + c_2 + c_3 + c_7) + \left(c_2 - \frac{1}{2}(c_3 + c_4 + c_5 + c_6)\right)n + \frac{1}{2}(c_3 + c_4 + c_5 + c_6)n^2$

$T(n) = \frac{1}{2}(c_3 + c_4 + c_5 + c_6)n^2 + \left(c_2 - \frac{1}{2}(c_3 + c_4 + c_5 + c_6)\right)n + (c_1 + c_2 + c_3 + c_7)$

Let $a = \frac{1}{2}(c_3 + c_4 + c_5 + c_6)$, $b = c_2 - \frac{1}{2}(c_3 + c_4 + c_5 + c_6)$, $c = c_1 + c_2 + c_3 + c_7$

$T(n) = an^2 + bn + c$

$T(n) = O(n^2)$

Therefore, the runtime complexity of COMPUTE-DISTANCES is $O(n^2)$.

HEAPSORT

The runtime complexity of HEAPSORT is a function of the runtime complexities of the functions it calls.

The runtime complexity of MAX-HEAPIFY is:

$T(n) = c_1 + c_2 + c_3 + c_4 + c_6 + c_7 + c_8 + c_9 + T(\frac{2n}{3})$

$T(n) = T\left(\frac{2n}{3}\right) + c$

By case 2 of the master theorem (CLRS, 2009, p. 94), $T(n) = \lg n$. Applying the master theorem to this case, $a = 1$, $b = 3/2$, and $f(n) = 1$. $n^{\log_{3/2} 1} = n^0 = 1 = f(n)$, and therefore case 2 applies.

The runtime complexity of BUILD-MAX-HEAP, which calls MAX-HEAPIFY, is not simply $O(n \lg n)$, because each MAX-HEAPIFY runs at a node of variable height in the tree, “and the heights of most nodes are small” (CLRS, 2009, p. 157). The runtime complexity of the algorithm is (CLRS, 2009, p. 159):

$$\sum_{h=0}^{\lceil \lg n \rceil} \left\lceil \frac{n}{2^{h+1}} \right\rceil O(h) = O\left(n \sum_{h=0}^{\lceil \lg n \rceil} \frac{h}{2^h}\right) = O\left(n \sum_{h=0}^{\infty} \frac{h}{2^h}\right) = O(n)$$

We now find the runtime of HEAPSORT. Let $c_1 n$ be the term representing the time complexity of BUILD-MAX-HEAP.

$T(n) \le c_1 O(n) + c_2 n + c_3(n-1) + c_4(n-1) + c_5 O(\lg n)$

$T(n) \le O(n + n \lg n) = O(n \lg n)$

Therefore, the time complexity of HEAPSORT is $O(n \lg n)$.

SELECT-M-POINTS

The time complexity of selecting the first m points of the sorted list is $O(1)$.

FIND-NEAREST-PAIRS

The time complexity of FIND-NEAREST-PAIRS is the sum of the time complexities of each of its algorithms:

- COMPUTE-DISTANCES: $O(n^2)$
- HEAPSORT: $O(n \lg n)$
- SELECT-M-POINTS: $O(1)$

$T(n) = O(n^2) + O(n \lg n) + O(1) = O(n^2 + n \lg n + 1) = O(n^2)$

Therefore, the worst case time complexity of the FIND-NEAREST-PAIRS algorithm is $O(n^2)$.

2(b) Implement the algorithm. Code must have a reasonable, consistent style and documentation. It must have appropriate data structures, modularity, and error-checking.

From the README.md file in PRasmussenAlgosPA1/pa1:

Peter Rasmussen, Programming Assignment 1
This Python program finds the m nearest pairs of n randomly generated, two-dimensional points.

Getting Started
The package is designed to be executed as a module from the command line. The user must specify the input file path and output directory as illustrated below. The PRasmussenAlgosPA1/resources directory provides example input and output files for the user. The PRasmussenAlgosPA1/resources/inputs/default.csv file provides 24 (n, m) combinations.

- `python -m path/to/pa1 -i path/to/in_file.csv -o path/to/out_dir/`

Optionally, the user may specify a file header that is prepended to the outputs. The example below illustrates usage of the optional argument.

- `python -m path/to/pa1 -i path/to/in_file.csv -o path/to/out_dir/ -f "Your Header"`

Finally, the user may specify the random seed used to generate the randomly distributed set of points, as the example below shows.

- `python -m path/to/pa1 -i path/to/in_file.csv -o path/to/out_dir/ -s 777`

A summary of the command line arguments is below.

Positional arguments:
```
    -i, --src          Input File or Directory Pathname
    -o, --dst_dir      Output File or Directory Pathname
```

Optional arguments:
```
    -h, --help         Show this help message and exit
    -f, --file_header  Input custom file header above output file
    -s, --seed         Provide pseudo-random seed
```

Key parts of program

- DataMaker: Class that ingests a set of n-m combinations and, using an optionally provided random seed, generates a random set of points.
- DistanceComputer: Class that computes the distance between each pair of points.
- HeapSort: Sorts the points list by distance.

Features

- Capability to process one or more n-m combinations per run.
- Performance metrics for each run for each algorithm: number of distance comparisons, number of heapifies, and total number of operations (distance comparisons + n number of heapifies).
- Tested on inputs of up to n=1024 and m=523,776.
- Outputs provided as two files: 1) CSV of performance metrics and 2) JSON of echoed inputs and selected outputs.
- Control over randomization by selection of random seed.

Input and Output Files
The resources/inputs directory contains the set of input files. Preprocessed outputs are
in the resources/outputs directory.

Example Output Files

An example of the first few lines of the default.csv file is reproduced below. Each row represents a run. We capture n, m, the number of distance comparisons, the number of heapifies, and the total number of operations in the n, m, dist_comps, n_heapifies, and total_ops columns, respectively.

n	m	dist_comps	n_heapifies	total_ops
2	1	1	0	1
4	6	6	3	9
8	28	28	14	42
16	120	120	60	180

The default.json output echoes the randomized data made by DataMaker and also presents the m-nearest point pairs.

Sources

The heap sort method is adapted from the following source.

Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). Introduction to Algorithms. Cambridge: The MIT Press.

Licensing

2(c) Perform and submit trace runs demonstrating the proper functioning of the code.

The following are screenshots of the first and last 31 lines of a trace run executed using the command below. The full trace run output is available in the `PRasmussenAlgosPA1/resources/outputs/trace_run.txt`.

```
nohup python -m trace --trace --module -m pa1 -i resources/inputs/trace_run.csv -o
resources/outputs &
```

trace_run_output.txt

```
 --- modulename: __main__, funcname: <module>
__main__.py(1): """Peter Rasmussen, Programming Assignment 1, __main__.py
__main__.py(23): import argparse
__main__.py(24): from pathlib import Path
__main__.py(25): import sys
__main__.py(28): from pa1.run import run
__main__.py(31): sys.setrecursionlimit(16000)
__main__.py(35): parser = argparse.ArgumentParser()
 --- modulename: argparse, funcname: __init__
argparse.py(1652):         superinit = super(ArgumentParser, self).__init__
argparse.py(1653):         superinit(description=description,
argparse.py(1654):                   prefix_chars=prefix_chars,
argparse.py(1655):                   argument_default=argument_default,
argparse.py(1656):                   conflict_handler=conflict_handler)
argparse.py(1653):         superinit(description=description,
 --- modulename: argparse, funcname: __init__
argparse.py(1260):         super(_ActionsContainer, self).__init__()
argparse.py(1262):         self.description = description
argparse.py(1263):         self.argument_default = argument_default
argparse.py(1264):         self.prefix_chars = prefix_chars
argparse.py(1265):         self.conflict_handler = conflict_handler
argparse.py(1268):         self._registries = {}
argparse.py(1271):         self.register('action', None, _StoreAction)
 --- modulename: argparse, funcname: register
argparse.py(1309):         registry = self._registries.setdefault(registry_name, {})
argparse.py(1310):         registry[value] = object
argparse.py(1272):         self.register('action', 'store', _StoreAction)
 --- modulename: argparse, funcname: register
argparse.py(1309):         registry = self._registries.setdefault(registry_name, {})
argparse.py(1310):         registry[value] = object
argparse.py(1273):         self.register('action', 'store_const', _StoreConstAction)
```

trace_run_output.txt

```
 --- modulename: __init__, funcname: _format
__init__.py(436):         return self._fmt % record.__dict__
__init__.py(672):         if record.exc_info:
__init__.py(677):         if record.exc_text:
__init__.py(681):         if record.stack_info:
__init__.py(685):         return s
__init__.py(1086):             stream = self.stream
__init__.py(1088):             stream.write(msg + self.terminator)
__init__.py(1089):             self.flush()
 --- modulename: __init__, funcname: flush
__init__.py(1066):         self.acquire()
 --- modulename: __init__, funcname: acquire
__init__.py(902):         if self.lock:
__init__.py(903):             self.lock.acquire()
__init__.py(1067):         try:
__init__.py(1068):             if self.stream and hasattr(self.stream, "flush"):
__init__.py(1069):                 self.stream.flush()
__init__.py(1071):             self.release()
 --- modulename: __init__, funcname: release
__init__.py(909):         if self.lock:
__init__.py(910):             self.lock.release()
__init__.py(956):                 self.release()
 --- modulename: __init__, funcname: release
__init__.py(909):         if self.lock:
__init__.py(910):             self.lock.release()
__init__.py(957):         return rv
__init__.py(1658):             for hdlr in c.handlers:
__init__.py(1662):             if not c.propagate:
__init__.py(1665):                 c = c.parent
__init__.py(1657):         while c:
__init__.py(1666):         if (found == 0):
```

Finally, a screenshot of the log of a successful run is below.

pa1.log

```
2021-10-18 19:17:28,596 - DEBUG - Begin run: src=default.csv, dst_dir=outputs, seed=777
2021-10-18 19:17:28,597 - DEBUG - Read data and check, among other things, that m <= n choose 2
2021-10-18 19:17:28,602 - DEBUG - Iterate over each n, m pair
2021-10-18 19:17:28,603 - INFO - n=2, m=1
2021-10-18 19:17:28,603 - INFO - n=4, m=6
2021-10-18 19:17:28,603 - INFO - n=8, m=28
2021-10-18 19:17:28,604 - INFO - n=16, m=120
2021-10-18 19:17:28,607 - INFO - n=32, m=496
2021-10-18 19:17:28,627 - INFO - n=64, m=2016
2021-10-18 19:17:28,720 - INFO - n=128, m=8128
2021-10-18 19:17:29,065 - INFO - n=256, m=32640
2021-10-18 19:17:30,651 - INFO - n=512, m=130816
2021-10-18 19:17:36,775 - INFO - n=1024, m=523776
2021-10-18 19:18:03,431 - INFO - n=256, m=2
2021-10-18 19:18:05,469 - INFO - n=256, m=4
2021-10-18 19:18:06,685 - INFO - n=256, m=8
2021-10-18 19:18:07,968 - INFO - n=256, m=16
2021-10-18 19:18:09,188 - INFO - n=256, m=32
2021-10-18 19:18:10,562 - INFO - n=256, m=64
2021-10-18 19:18:11,839 - INFO - n=256, m=128
2021-10-18 19:18:13,920 - INFO - n=256, m=256
2021-10-18 19:18:15,108 - INFO - n=256, m=512
2021-10-18 19:18:16,268 - INFO - n=256, m=1024
2021-10-18 19:18:17,435 - INFO - n=256, m=2048
2021-10-18 19:18:18,705 - INFO - n=256, m=4096
2021-10-18 19:18:19,896 - INFO - n=256, m=8192
2021-10-18 19:18:22,849 - INFO - n=256, m=16384
2021-10-18 19:18:24,026 - DEBUG - Write performance outputs to CSV.
2021-10-18 19:18:24,028 - DEBUG - Write set of m closest pairs for given n to JSON.
2021-10-18 19:18:36,090 - DEBUG - Finish.
```

2(d) Perform tests to measure the asymptotic behavior of the program (call this the code's worst case running time).

The asymptotic behavior of the program is $O(n^2)$, as we'll see.

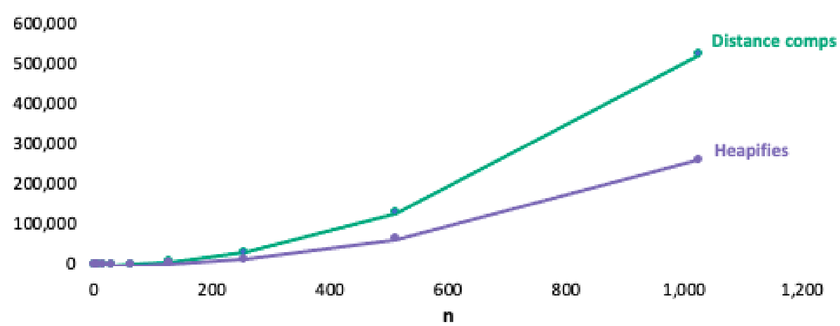
The default input generates multiple n-m combinations to assess the asymptotic behavior of the program[1]. The following table provides a subset of the performance outputs created by the program, and is truncated for the sake of brevity.

Table 1: Subset of program performance tests

n	m	distance comps	heapifies	total operations
2	1	1	0	1
4	6	6	3	9
8	28	28	14	42
16	120	120	60	180
32	496	496	248	744
64	2,016	2,016	1,008	3,024
128	8,128	8,128	4,064	12,192
256	32,640	32,640	16,320	48,960
512	130,816	130,816	65,408	196,224
1,024	523,776	523,776	261,888	785,664

Figure 1 provides, for increasing n, the breakdown of the number of 1) distance comparisons, the key performance

Figure 1: Breakdown of time complexity of program



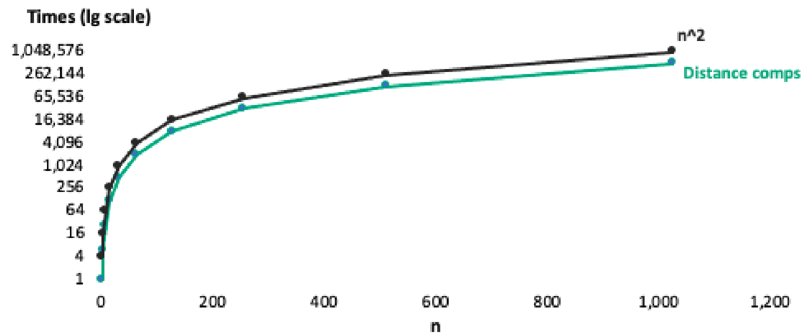
metric of the DistanceComputer algorithm and 2) heapifies, the key performance metric of the HeapSort[2]. We can see that the each of the implemented algorithm runs in super-linear time. How tight of an upper bound can we find for these algorithms, and what is the total runtime complexity of the program?

Figure 2 shows the number of distance comparisons against the n^2 on a logarithmic y-scale. The log scale allows us to see that the number of distance comparisons differs by n^2 by a constant factor. This implies the asymptotic time complexity of the DistanceComputer is $O(n^2)$. It turns out that the number of distance comparisons is exactly half of n^2. This is because the ordering of points in a pair does not matter. For instance, the pair [(1, 0), (0, 1)] is the same as the pair [(0, 1), (1, 0)].

[1] Refer to `PRasmussenAlgosPA1/resources/inputs/default.csv`.

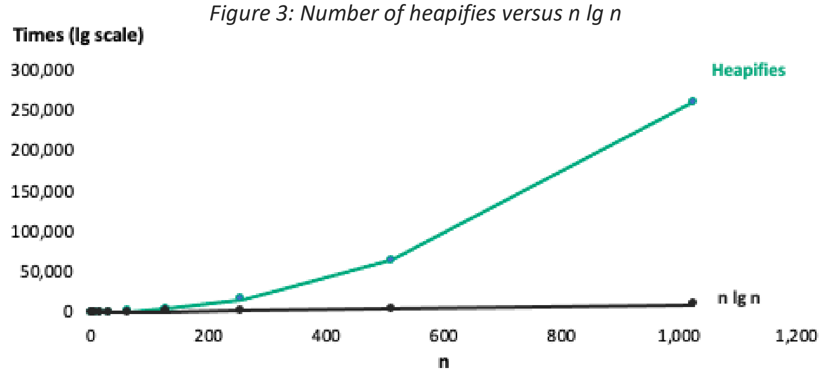
[2] The data used to generate these figures is available in `PRasmussenAlgosPA1/analysis/charts.xlsx`.

Figure 2: Number of distance comparisons versus n²



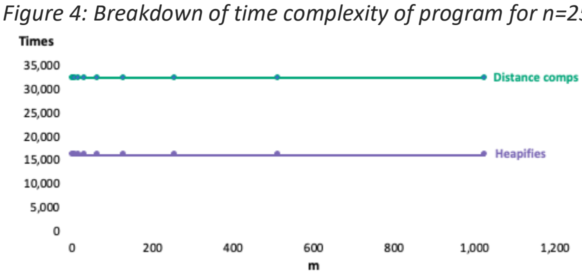
Next, we investigate the number of heapifies versus a plot of $n \lg n$, the upper bound we found for HEAP-SORT. It may appear that this implementation of HEAP-SORT is $O(n^2)$, however recall that the height of the heapifies is not constant, and that important nuance is not captured in the number of heapifies metric. As mentioned in 2(a)(ii), most of the node heights are small.

Figure 3: Number of heapifies versus n lg n



But what about m? As Figure 4 shows, varying m has no effect on runtime. This is because no matter what we must compute all combinations of point pairs, regardless of which m we choose.

Figure 4: Breakdown of time complexity of program for n=256

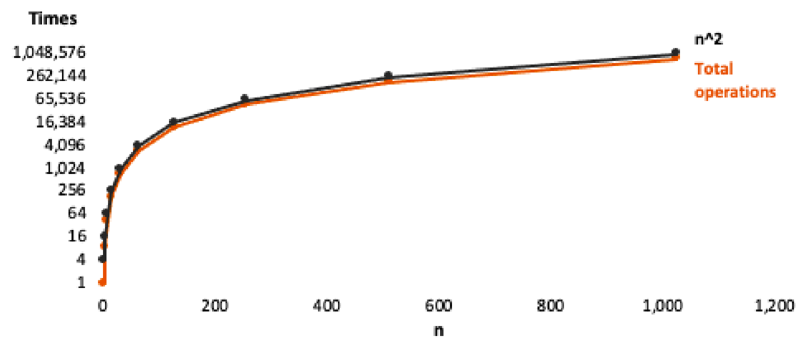


2(e) Analysis comparing algorithm's worst case running time to code's worst case running time.

The pseudo code and program both have $O(n^2)$ running time. However, because of the way that I measured the performance of the program's heap sort, it looks like – for the program – that both the DistanceComputer and HeapSort are both $O(n^2)$. However, this is not the case because the heapify portion of HeapSort runs on varying node heights, and most node heights are small (more nodes tend to be closer to the bottom of the heap than the top). So then, the HeapSort should run in $O(n \lg n)$. Regardless of the running time of HeapSort (which is $O(n \lg n)$), the dominating term in both the algorithm and the program is $O(n^2)$.

Figure 5 shows this clearly. In the worst case, the complexity of the program versus the algorithm is $O(n^2)$. The only separation between the $O(n^2)$ and the total number of operations in the program is a constant factor.

Figure 5: Total complexity of program



It becomes very clear that in the long run, as n increases, it doesn't matter that one of the terms that comprise $T(n)$ is $n \lg n$ rather than n^2. This fact is apparent by the asymptotic behavior of the plot of $n \lg n$ versus the n^2 plot in Figure 3.

3. Retrospective

(a) Now that you have designed, implemented, and tested your algorithm, what aspects of your algorithm and / or code could change and reduce the worst-case running time of your algorithm? Be specific in your response to this question.

From an $O(\cdot)$ perspective, nothing could reduce the worst-case running time of my algorithm and program. This is because the computation of distances *must* occur for all combinations of points, which *forces* the runtime complexity to be at least $O(n^2)$.

However, from the perspective of average runtime, the use of QUICKSORT would have improved running time – albeit quickly and increasingly insignificantly with increasing n – of both algorithm and program. I didn't implement QUICKSORT simply because I was more comfortable working with HEAPSORT. I originally attempted to implement MERGE-SORT, but I was getting recursion depth errors in my program, and so I abandoned that approach for the easier-to-implement (for me, anyway) HEAPSORT.