Closest Pair of Points: Brute Force vs Divide and Conquer

COT 6405 || Dr. Mihaela Cardei

Programming Project || Matthew Acs

This notebook contains the report, source code, and outputs in one deliverable. The sections in the table of contents make it easy to quickly navigate the different sections. By combing the report and code, the assignment is completer and more self-contained.

Problem Definition

This project explores the problem of finding the closest pair of points. Given n points (p1, p2, …, pn) in a plane, the objective is to find the points pi and pj that are the closest to one another. Brute force (Θ(n^2)) and divide and conquer (Θ(nlgn)) algorithms will be used to solve the problem to demonstrate the runtime of different approaches. The algorithms will return the index of the closest two points given the set of points. The approaches will be tested using a randomly generated set of test points with a cardinality of 1000, 2000, 3000, …, 10000. For each input size, the average of 10 iterations will be recorded.

The closest points problem has several real-world applications. One application of the problem is for controlling air traffic. If the closest points problem is generalized to more dimensions, the problem can be concerned with finding the pair of points in three dimensions that are the closest. It is important for air traffic controllers to be notified of the two airplanes in an airspace that are the closest to each other for controllers to effectively monitor situations that may involve near collisions and high traffic conditions. Essentially, the closet points problem could be used in situations that involve collision detection and avoidance.

Another application is in analyzing complex datasets such as in genetics. A high-dimensional dataset may be projected into a lower dimensionality using PCA. Closest point analysis can then be used to find the pair of data points that are the closest to one another. This can be used to discover patterns and similarities in highly complex data such as genomes. For instance, two people with the most similar genome can be determined this way as well as the two most similar genes within a segment of the genome of two species.

Pseudocode

The main idea behind the brute force algorithm is to find the distance between every possible pair of points and identify the two points that minimize this distance. This requires two nested loops that traverse the set of points, making it simple, but inefficient.

```
BruteForceClosestPoints(P)
-------------------------
d_min = inf
for i = 1 to n-1:
    for j = i + 1 to n:
        d = sqrt( (xi-xj)^2 + (yi-yj)^2) )
        if d < d_min:
            d_min = d
            index_1 = i
            index_2 = j
return index_1, index_2
```

On the other hand, the divide and conquer algorithm divides the original problem of finding the two closest points into smaller subproblems that are solved recursively. The original problem is divided into finding the two closest points in the left half of the plane and in the right half of the plane. The problem is recursively solved by dividing the plane and making recursive calls until the base case is reached in which there are only three or less points. Then these solutions are combined to find the two closest points in the entire plane. As part of the combine step, the case in which the two closest points bridge the divide between the left and right is also considered.

```
Closest-Pair(P)
---------------
construct Px and Py          // O(nlgn)
(p0*, p1*) = Closest-Pair-Rec(Px, Py)

Closest-pair-Rec(Px, Py)
-----------------------
if |P|<=3
    find the closest pair by measuring all pairwise distances
construct Qx, Qy, Rx, Ry.         // O(n)
(q0*, q1*) = Closest-Pair-Rec(Qx, Qy)
(r0*, r1*) = Closest-Pair-Rec(Rx, Ry)
delta = min( d((q0*, q1*)), d((r0*, r1*)) )
x* = maximum x-coordinate of a point in set Q
L = {(x, y): x = x*}
S = points in P within distance delta of L
construct Sy          // O(n)
for each point s in Sy          // O(n)
    compute the distance from s to each of the next 15 points in Sy
let s, s’ be the pair with the minimum distance
```

```
if d(s, s') < delta
    return (s, s')
else if d((q0*, q1*)) < d((r0*, r1*))
    return (q0*, q1*)
else
    return (r0*, r1*)
```

Runtime Analysis

The brute force algorithm contains an inner for loop that executes from j=i+1 to n and an outer for loop that executes from i=1 to n-1. Since the instructions in the inner for loop take a constant time, it follows that the nested loops execute Θ(n^2) times. The equations below show how this runtime can be derived.

$$\Theta(\sum_{i=1}^{n-1}\sum_{j=i+1}^{n} 1) = \Theta\left(\sum_{i=1}^{n-1} n-i\right) = \Theta(\,(n-1)+(n-2)+\cdots+1)$$

$$= \Theta(1+2+3+4+\cdots+(n-1)) = \Theta\left(\frac{(n-1)((n-1)+1)}{2}\right) = \Theta(n^2)$$

∴ The runtime of the brute force algorithm is Θ(n^2).

On the other hand, the divide and conquer algorithm utilizes a recurrence and a sorting algorithm before the recurrence. The sorting algorithm is used twice to create Px and Py, and this sorting algorithm has a time complexity of Θ(nlgn). The recurrence can be solved using the master theorem. The algorithm recurses on 2 subproblems with a size n/2 and the divide and combine steps take O(n) due to the construction of Qx, Qy, Rx, Ry, the construction of Sy, and computing the distance from s to each of the next 15 points in Sy. Therefore, the runtime can be written as T(n) = 2T(n/2) + Θ(n).

$$T(n) = 2T\left(\frac{n}{2}\right) + \Theta(n)$$
$$n \; vs \; n^{log_2^2} = n^1 = n$$
$$n = \Theta(n) => Case\ II\ of\ the\ Master\ Theorem$$
$$\Theta(nlgn)$$

∴ The runtime of the divide and conquer algorithm is Θ(nlgn) due to the time complexity of the recurrence and sorting algorithm being Θ(nlgn).

Import Dependencies

The code below imports all of the necessary python packages.

```
import math
import copy
import numpy
import random
import numpy as np
from statistics import mean
from datetime import datetime
import matplotlib.pyplot as plt
from prettytable import PrettyTable
```

Define Supporting Functions

The code below defines supporting functions that are used in the algorithms, driver code, and algorithm testing.

```
## Randomly generates n unique points with integer
## coordinates in a domain of [0, 10*n] and returns
## their x and y values as two separate lists
## -----------------------------------------------
def create_points(n):
  pool = list(range(0,10*n))
  sample = random.sample(pool,2*n)
  x = sample[:len(sample)//2]
  y = sample[len(sample)//2:]
  return x, y

## Plots a set of points given their x and y
## values as two separate lists
## -----------------------------------------------
def plot_points(x,y):
  plt.plot(x,y,'ro')
  plt.show()

## Plots two curves given their x and y values
## as two separate lists along with the plot
## title, axis labels, and legend.
## -----------------------------------------------
def plot_graph(x1,y1,x2,y2,legend1,legend2,xaxis,yaxis,title):
  plt.plot(x1,y1)
  plt.plot(x2,y2)
  plt.legend([legend1, legend2], loc ="upper left")
  plt.xlabel(xaxis)
  plt.ylabel(yaxis)
  plt.title(title)
  plt.show()

## Prints the coordinates two points given
## the lists x, y and the indices of the two points.
## -----------------------------------------------
def print_closest_points(x, y, index_1, index_2):
  print("Closest Points:")
  print("---------------")
  print("(" + str(x[index_1]) + ", " + str(y[index_1]) + ")")
  print("(" + str(x[index_2]) + ", " + str(y[index_2]) + ")")
  print("---------------")

## Returns time since the Unix epoch.
## -----------------------------------------------
def get_time():
  return(datetime.now().timestamp())
```

Algorithm 1 Brute Force

The code below defines algorithm 1, brute force.

```
## The brute force algorithm find the
## distance between every pair of points
## and returns the pair that minimizes the distance.
## ---------------------------------------------
def ALG1(n, x, y):
  d_min = math.inf
  for i in range(n-1):        ## Two loops together are O(n^2)
    for j in range(i+1, n):
      distance = math.dist([x[i], y[i]], [x[j], y[j]])
      if distance < d_min:
        d_min = distance
        index_1 = i
        index_2 = j
  return index_1, index_2
```

Algorithm 2 Divide and Conquer

The code below defines algorithm 2, divide and conquer.

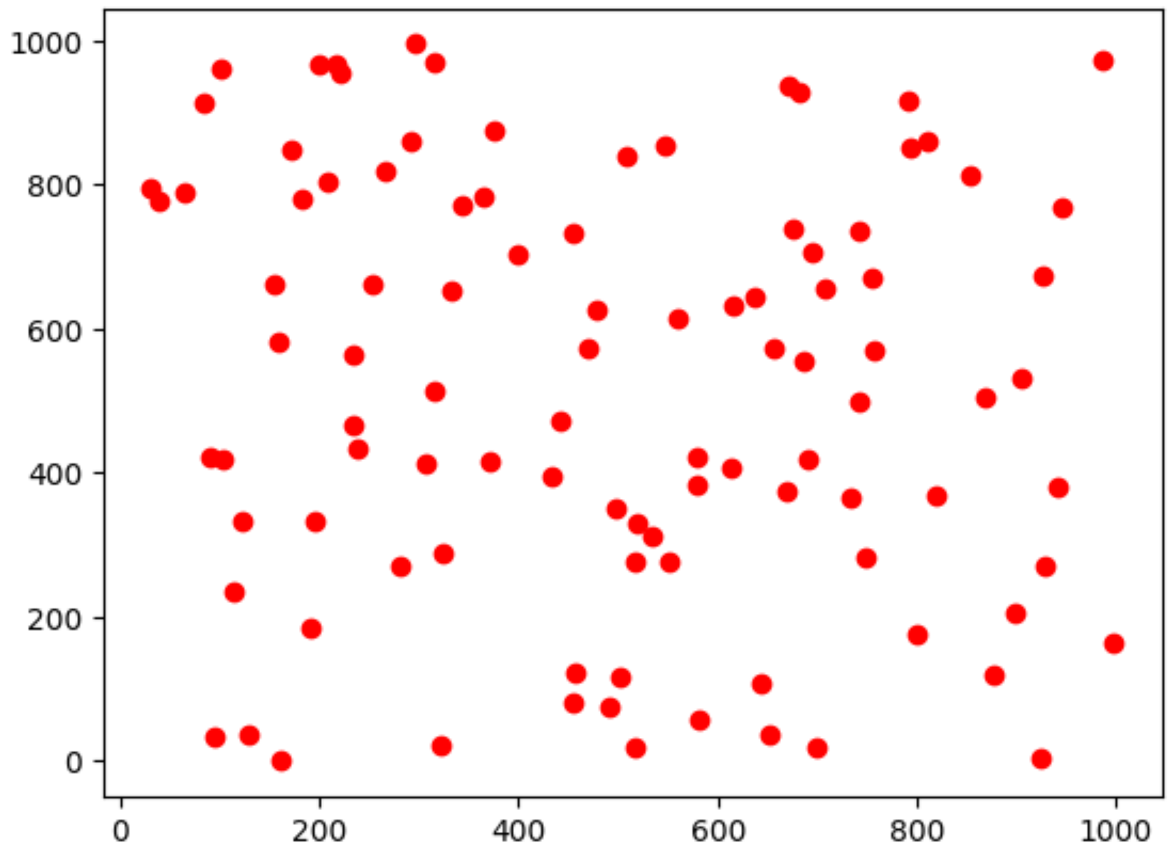
```
## Initial ALG2 function call. Constructs px
## and py using heapsort and initiates the
## first recursive call. Returns the indices
## of the two closest points in x and y.
# -----------------------------------------------
def ALG2(n, x, y):
  px = numpy.argsort(x, kind="heapsort") ## O(nlgn) due to heapsort
  py = numpy.argsort(y, kind="heapsort") ## O(nlgn) due to heapsort
  index_1, index_2 = ALG2_rec(n, n, x, y, px, py)
  return index_1, index_2

## ALG2 recursive call. Contains the base
## case, divide, conquer (recursive calls),
##  and combine steps characteristic of
## divide and conquer algorithms. Returns
##  the indices of the two closest points
## in the subproblem it was called on.
# ---------------------------------------------
def ALG2_rec(z, n, x, y, px, py):

  ## Base Case
  ## *********
  ## Find closest pair by measuring
  ## all pairwise distances
  if n <= 3:
    distance = []
    for i in range(n-1):
      for j in range(i+1, n):
        distance.append(math.dist([x[px[i]], y[px[i]]], [x[px[j]], y[px[j]]]))

    min_base = min(distance)
    if min_base == distance[0]:
      return  px[0], px[1]
    if min_base == distance[1]:
      return  px[0], px[2]
    if min_base == distance [2]:
      return px[1], px[2]
  ## *********
  ## END Base Case

  ## Divide
  ## *********
  ## Construct Qx, Qy, Rx, Ry
  Q = math.ceil(n/2)
  R = math.floor(n/2)

  Qx = [0] * Q
  Rx = [0] * R
  flag = [0] * z

  for i in range(n): # O(n)
    if i < Q:
      Qx[i] = px[i]
    else:
      Rx[i-Q] = px[i]
      flag[px[i]] = 1

  Qy = [0] * Q
  Ry = [0] * R
  j=0
```

```
k=0

for i in range(n): # O(n)
  if flag[py[i]] == 0:
    Qy[j] = py[i]
    j = j+1
  else:
    Ry[k] = py[i]
    k = k+1
## *********
## END Divide

## Recursion (Conquer)
## *********
q0, q1 = ALG2_rec(z, Q, x, y, Qx, Qy)
r0, r1 = ALG2_rec(z, R, x, y, Rx, Ry)
## *********
## END Recursion

## Combine
## *********
distance_q = math.dist([x[q0], y[q0]], [x[q1], y[q1]])
distance_r = math.dist([x[r0], y[r0]], [x[r1], y[r1]])
# Compute delta as the minimum of the left and right subproblem solutions
delta = min(distance_q, distance_r)
x_prime = x[Qx[Q-1]]

# Construct Sy in O(n)
Sy = []
for i in range(n):
  if abs(x[py[i]] - x_prime) < delta:
    Sy.append(int(py[i]))

# Compute distance from s to each of the next 15 points in Sy
# for each point s in Sy. O(n)
loop_min = min(15, len(Sy))
min_distance = math.inf
for i in range(len(Sy)):
  for j in range(loop_min):
    if i != j:
      distance_s = math.dist([x[Sy[i]], y[Sy[i]]], [x[Sy[j]], y[Sy[j]]])
      if distance_s < min_distance and distance_s != 0:
        min_distance = distance_s
        s1 = Sy[i]
        s2 = Sy[j]
# Return the best solution
if min_distance < delta:
  return s1, s2
elif distance_q < distance_r:
  return q0, q1
else:
  return r0, r1
## *********
## End Combine
```

Algorithms 1 and 2 Test

The code below tests algorithms 1 and 2 on a set of 100 points. The points are displayed in a graph and then the outputs and runtimes of the algorithms are displayed.

```
## Creates 100 points and plots them
n = 100
x, y = create_points(n)
plot_points(x, y)
```



```
## Tests ALG1 on the set of 100 points
time1 = get_time() * 1000000
index_1, index_2 = ALG1(n, x, y)
time2 = get_time() * 1000000
runtime = time2-time1

print("ALG1")
print("")
print("Runtime: " + str(runtime) + " microseconds")
print_closest_points(x, y, index_1, index_2)
```

```
ALG1

Runtime: 2472.0 microseconds
Closest Points:
---------------
(91, 423)
(103, 420)
---------------
```

```
## Tests ALG2 on the set of 100 points
time1 = get_time() * 1000000
index_1, index_2 = ALG2(n, x, y)
time2 = get_time() * 1000000
runtime = time2-time1

print("ALG2")
print("")
print("Runtime: " + str(runtime) + " microseconds")
print_closest_points(x, y, index_1, index_2)
```

```
ALG2

Runtime: 1181.0 microseconds
Closest Points:
---------------
(103, 420)
(91, 423)
---------------
```

Main Driver Code

The code below is the main program driver. It utilizes the previously defined functions to run the experiments on the specified n values for the specified number of iterations. It also calculates the values necessary for the tables and graphs.

```
## VALUES SELECTED: n Values selected for experiments
N_VALUES = [1000, 2000, 3000, 4000, 5000, 6000, 7000, 8000, 9000, 10000]

N2_VALUES = np.square(N_VALUES)
NLGN_VALUES = N_VALUES * np.log2(N_VALUES)

## RUNNING TRIALS: m runs for each input value selected
ITERATIONS = 10


t_ALG1 = [[],[],[],[],[],[],[],[],[],[]]
t_ALG2 = [[],[],[],[],[],[],[],[],[],[]]
t_avg_ALG1 = []
t_avg_ALG2 = []
```

```
for n in N_VALUES:
  index = int((n/1000)-1)

  for j in range(ITERATIONS):

    ## IMPLEMENT ALL ALGORITHMS WITH
    ## EXACTLY THE SAME INPUT:
    ## Data points created and then
    ## used for both algorithms
    x, y = create_points(n)

    time1 = get_time() * 1000000
    index_1, index_2 = ALG1(n, x, y)
    time2 = get_time() * 1000000
    runtime = time2-time1
    t_ALG1[index].append(runtime)

    time1 = get_time() * 1000000
    index_1, index_2 = ALG2(n, x, y)
    time2 = get_time() * 1000000
    runtime = time2-time1
    t_ALG2[index].append(runtime)

  ## COMPUTE RT AVERAGE: average runtimes computed
  t_avg_ALG1.append(mean(t_ALG1[index]))
  t_avg_ALG2.append(mean(t_ALG2[index]))
```

Tables for Computing C

The constant C is computed using the tables below. The maximum ratio (empirical/theoretical) is stored as the constant. The constant is then multiplied by the theoretical runtime to find the predicted runtime.

```
## Values for tables and graphs are calculated.
ratio_ALG1 = t_avg_ALG1 / N2_VALUES
c1 = max(ratio_ALG1)
predicted_ALG1 = c1 * N2_VALUES

ratio_ALG2 = t_avg_ALG2 / NLGN_VALUES
c2 = max(ratio_ALG2)
predicted_ALG2 = c2 * NLGN_VALUES


## Generates a table that is specific to the instance
## that was just executed
x = PrettyTable()

x.add_column("n",N_VALUES)
x.add_column("Theoretical RT N^2",N2_VALUES)
x.add_column("Empirical RT (microseconds)", t_avg_ALG1)
x.add_column("Ratio (empirical/theoretical)", ratio_ALG1)
x.add_column("Predicted RT", predicted_ALG1)

print("Table ALG1")
print(x)
print("")
print("C1 = " + str(c1))
print("Predicted RT = C1 * Theoretical RT")
```

```
    Table ALG1
    +-------+--------------------+-----------------------------+-----------------
    |   n   | Theoretical RT N^2 | Empirical RT (microseconds) | Ratio (empirical
    +-------+--------------------+-----------------------------+-----------------
    |  1000 |      1000000       |           132014.5          |           0.1320
    |  2000 |      4000000       |           603154.3          |          0.15078
    |  3000 |      9000000       |          1289018.2          |      0.143224244
    |  4000 |      16000000      |          2267266.9          |         0.141704
    |  5000 |      25000000      |          3567766.8          |      0.142710671
    |  6000 |      36000000      |          5263028.6          |      0.146195238
    |  7000 |      49000000      |          7246388.6          |      0.147885481
    |  8000 |      64000000      |          9487624.7          |        0.1482441
    |  9000 |      81000000      |          12355558.0         |      0.152537753
    | 10000 |     100000000      |          15346921.3         |      0.153469213
    +-------+--------------------+-----------------------------+-----------------

    C1 = 0.15346921300000002
    Predicted RT = C1 * Theoretical RT
```

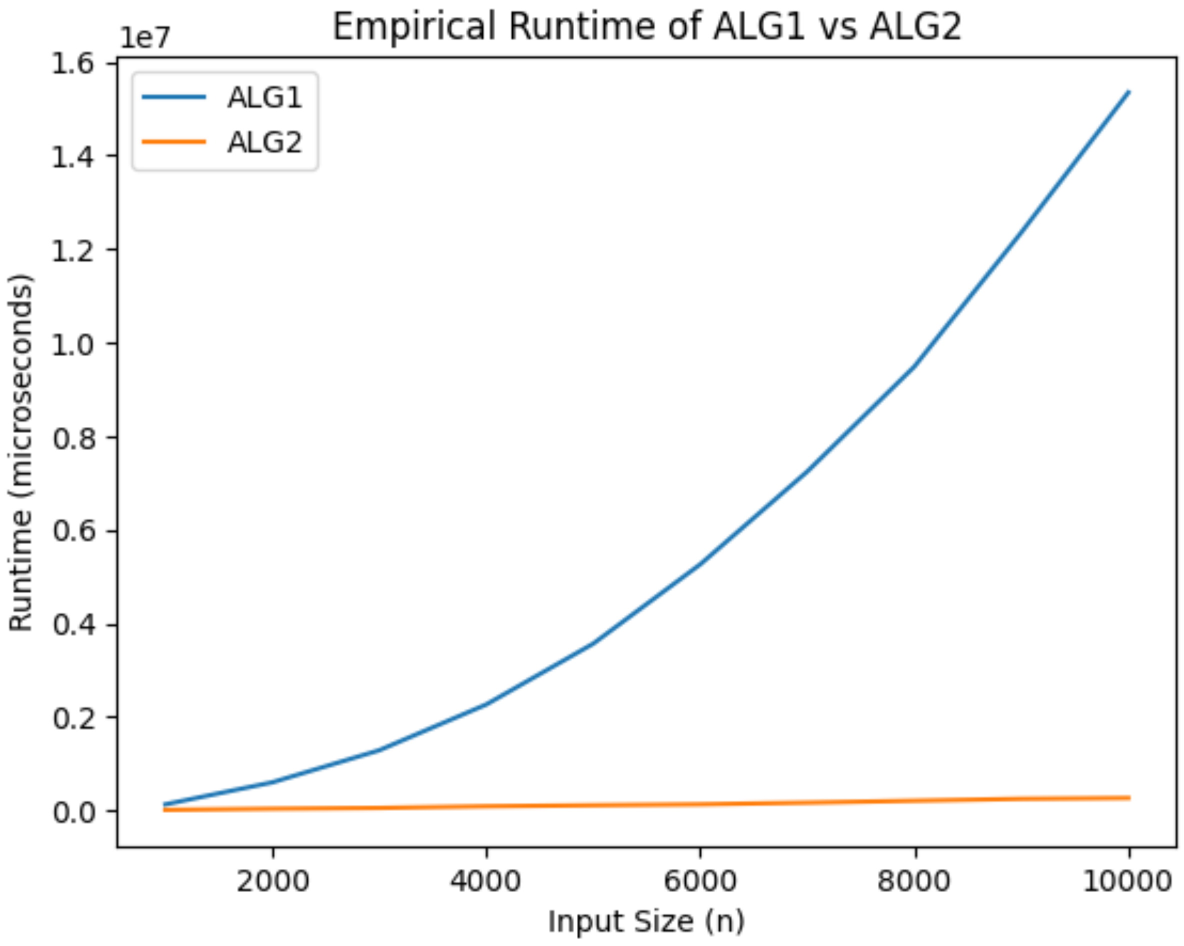
```
## Generates a table that is specific to the instance
## that was just executed
x = PrettyTable()

x.add_column("n",N_VALUES)
x.add_column("Theoretical RT NLGN",NLGN_VALUES)
x.add_column("Empirical RT (microseconds)", t_avg_ALG2)
x.add_column("Ratio (empirical/theoretical)", ratio_ALG2)
x.add_column("Predicted RT", predicted_ALG2)

print("Table ALG2")
print(x)
print("")
print("C2 = " + str(c2))
print("Predicted RT = C2 * Theoretical RT")
```

```
Table ALG2
+-------+---------------------+-----------------------------+----------------
|   n   | Theoretical RT NLGN | Empirical RT (microseconds) | Ratio (empirica
+-------+---------------------+-----------------------------+----------------
|  1000 |  9965.784284662088  |           15243.6           |       1.5295936
|  2000 |  21931.568569324176 |           39180.7           |       1.7864978
|  3000 |  34652.24035614973  |           55975.5           |       1.6153501
|  4000 |  47863.13713864835  |           91392.3           |       1.9094506
|  5000 |  61438.56189774724  |           113254.2          |       1.8433732
|  6000 |  75304.48071229945  |           133735.7          |       1.7759328
|  7000 |  89411.97444703784  |           168628.3          |       1.8859699
|  8000 |  103726.2742772967  |           210370.7          |       2.0281331
|  9000 |  118221.3835749396  |           251235.9          |       2.1251307
| 10000 |  132877.1237954945  |           268357.8          |       2.0195936
+-------+---------------------+-----------------------------+----------------

C2 = 2.1251307707860105
Predicted RT = C2 * Theoretical RT
```

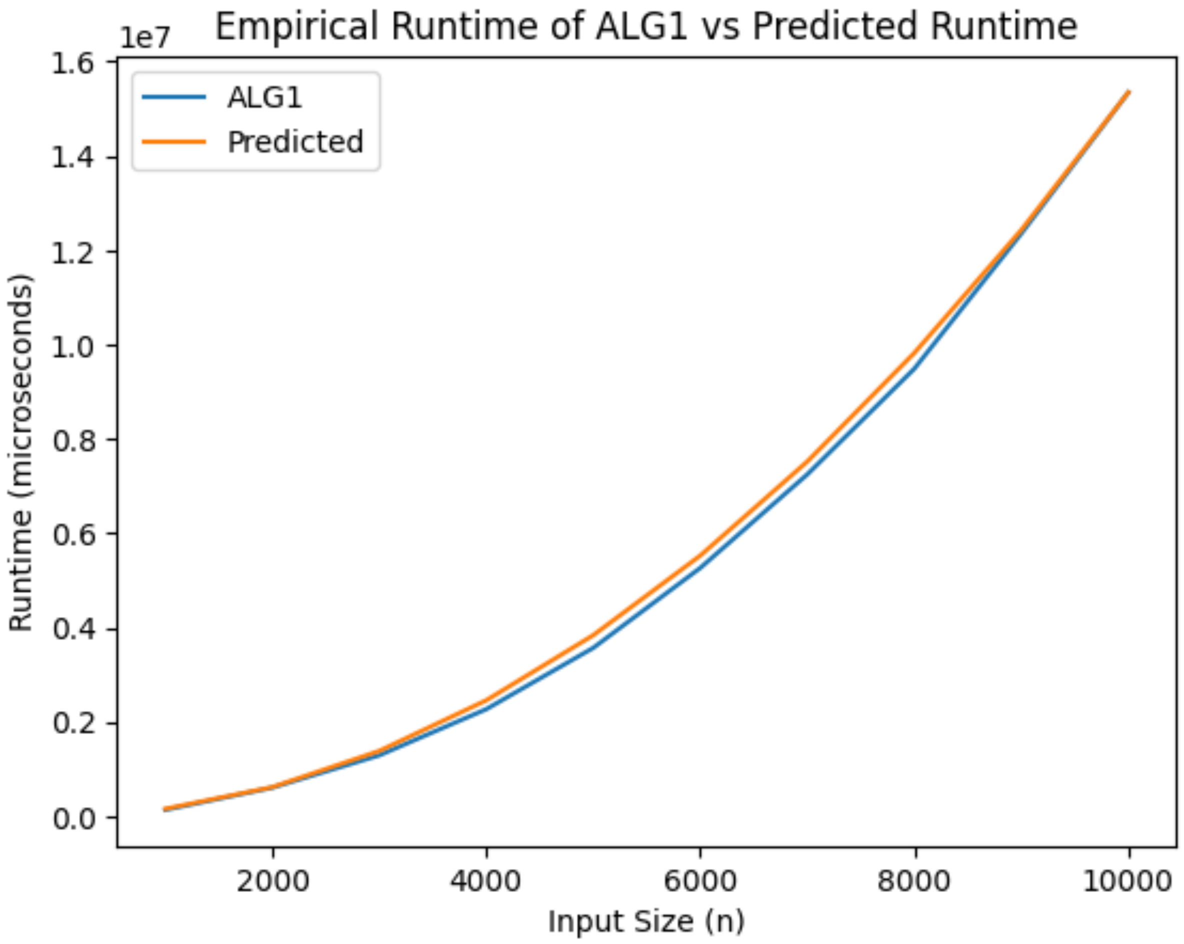
Runtime Graphs

The graphs below compare the runtimes of ALG1 and ALG2 as well as the runtimes of the algorithms with their respective predicted runtimes.

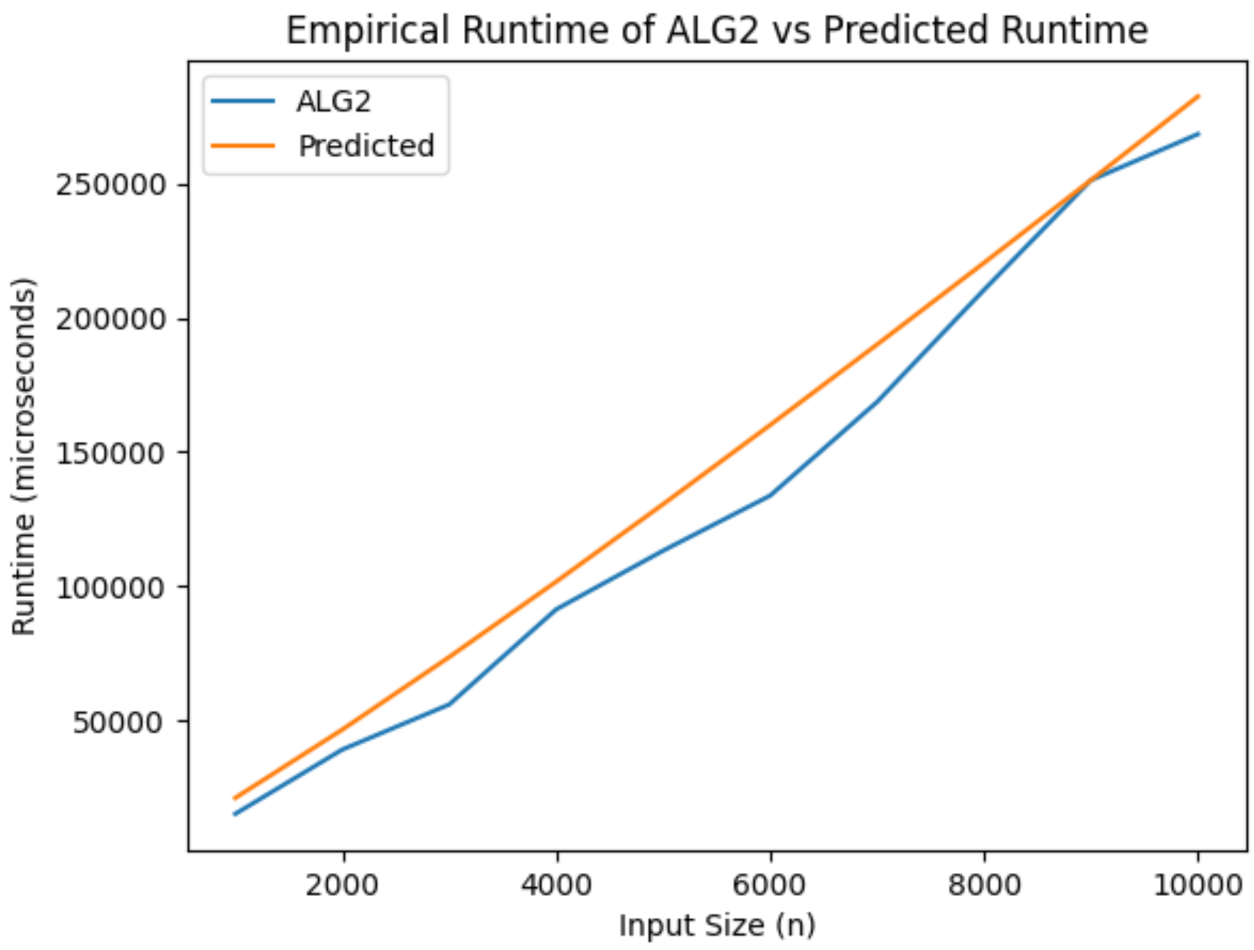
```
## Generates a graph that is specific to the instance
## that was just executed
plot_graph(N_VALUES, t_avg_ALG1, N_VALUES, t_avg_ALG2, 'ALG1', 'ALG2', "Input Size
```



```
## Generates a graph that is specific to the instance
## that was just executed
plot_graph(N_VALUES, t_avg_ALG1, N_VALUES, predicted_ALG1, "ALG1", "Predicted","In
```



```
## Generates a graph that is specific to the instance
## that was just executed
plot_graph(N_VALUES, t_avg_ALG2, N_VALUES, predicted_ALG2, "ALG2", "Predicted","In
```



Conclusions

First and foremost, the results showed that the brute force algorithm is significantly slower than the divide and conquer algorithm. Pseudocode analysis results in a runtime of n^2 for the brute force algorithm and a runtime of nlgn for the divide and conquer algorithm. This implies that at larger n values, the brute force algorithm will take much longer to run than the divide and conquer algorithm. For an input of n = 10,000, the brute force algorithm takes around 60 times longer to run than the divide and conquer algorithm in the experiments, verifying the runtime analysis. This can clearly be seen in the graph of ALG1 vs ALG2. As the input increase ALG1's runtime increases much faster than ALG2's runtime. ALG2's runtime is also much smaller than ALG1's runtime in the graph.

Additionally, the results show that the theoretical and empirical results are consistent. After using the constant and theoretical runtimes to calculate the predicted runtimes and plotting the empirical runtimes against the predicted runtimes, it can be seen that they are similar to one another. They follow the same general curve, and the values are similar. This shows that the theoretical and empirical results are consistent, verifying the pseudocode runtime analysis against experimental runtime results. This is because the difference between the theoretical runtime and the empirical runtime is the constant that is based on the processor speed, hardware, instruction speed, and other factors. Thus, by taking the ratio and finding the constant, the theoretical runtime can be directly compared to the empirical runtime by calculating the predicted runtime. If the predicted and empirical runtimes are similar and have a similar trend, then they are clearly consentient with one another.

Finally, I can conclude that time complexity is a very important criteria to keep in mind when developing algorithms that have very large input sizes. ALG1 took a very long time to compile, which may become intractable for large enough input sizes. On the other hand, ALG2 remained at a more reasonable runtime despite an increase in input size. For very large input sizes, ALG2 may compute a solution in a realistic time frame while ALG1 may become intractable.

Project Demo

A demonstration of the solution and code is linked below:

https://youtu.be/GoKYRP9AR6Q

References

- Cover photo by T.H. Chia on Unsplash
- Algorithm Design, by Jon Klainberg and Eva Tardos, Chapter 5.4
- https://matplotlib.org/
- https://pypi.org/project/prettytable/
- https://numpy.org/