Project 2: Greedy versus Exhaustive – Group 10 CPSC 335.

https://github.com/CSUF-CPSC-335-Bein-SU21/project-2-cpsc-335-project-2-group-10

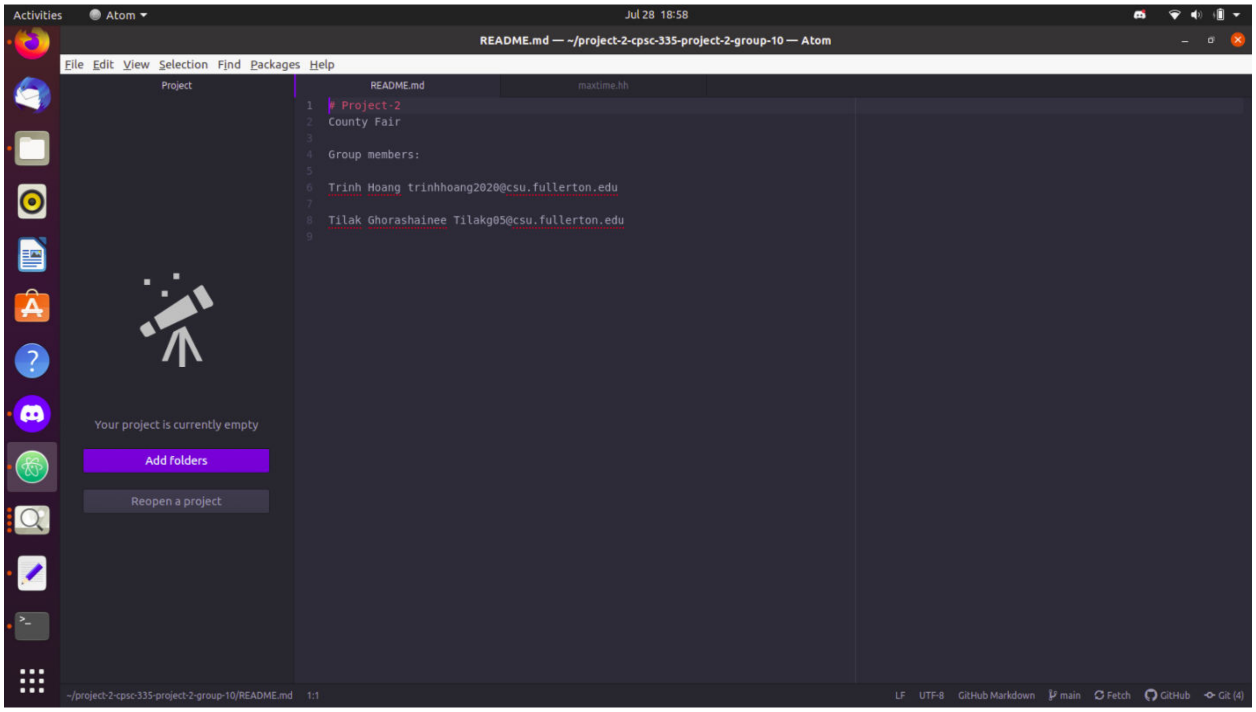
I. Group Members:

1. Name: Trinh Hoang
   Email: trinhhoang2020@csu.fullerton.edu

2. Name: Tilak Ghorashainee
   Email: Tilakg05@csu.fullerton.edu

II. Two full-screen screenshots:

README.md screenshot


Activities
Atom
Jul 28 18:58
README.md — ~/project-2-cpsc-335-project-2-group-10 — Atom
File Edit View Selection Find Packages Help
Project
README.md
maxtime.hh
Your project is currently empty
Add folders
Reopen a project
~/project-2-cpsc-335-project-2-group-10/README.md 1:1
LF UTF-8 GitHub Markdown main Fetch GitHub Git (4)

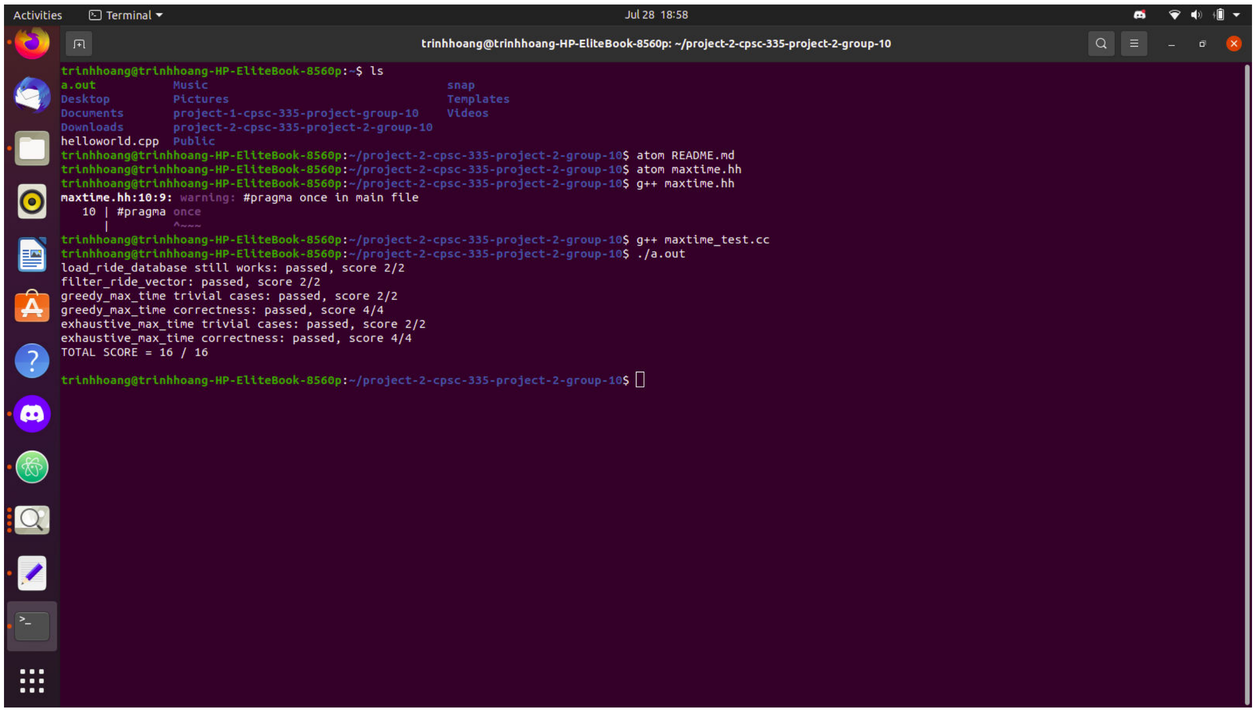
```
# Project-2
County Fair

Group members:

Trinh Hoang trinhhoang2020@csu.fullerton.edu

Tilak Ghorashainee Tilakg05@csu.fullerton.edu
```

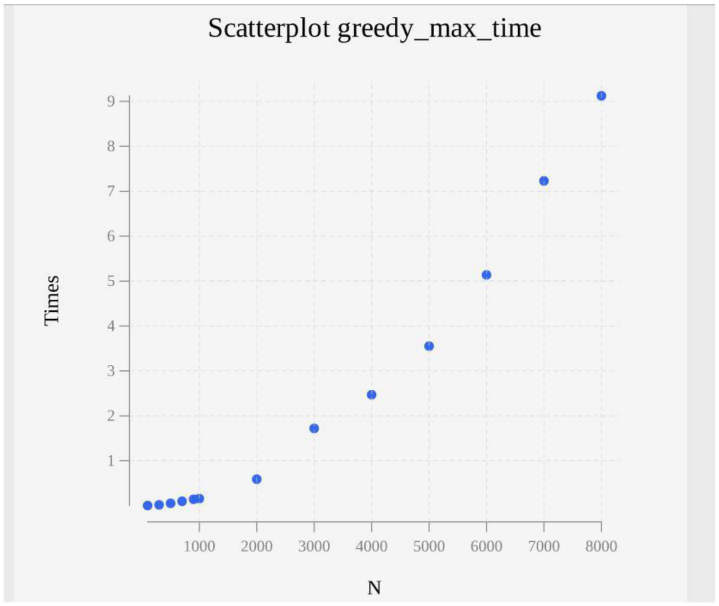
Code executing screenshot

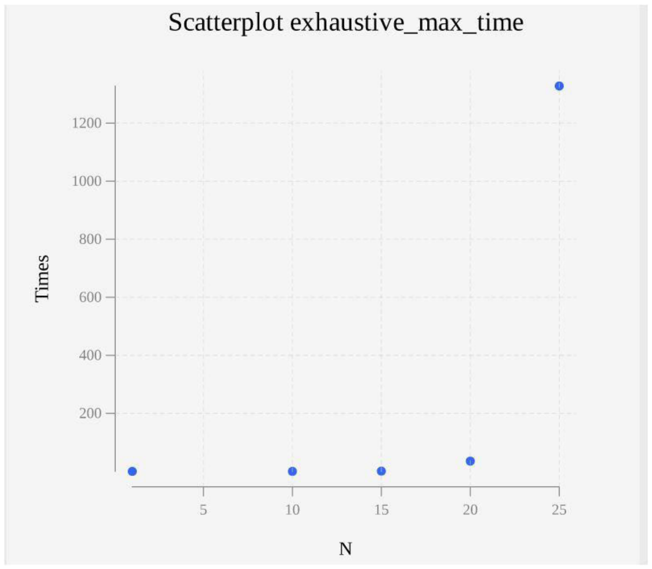

Activities
Terminal
Jul 28 18:58
trinhhoang@trinhhoang-HP-EliteBook-8560p: ~/project-2-cpsc-335-project-2-group-10

```
trinhhoang@trinhhoang-HP-EliteBook-8560p:~$ ls
a.out           Music                                  snap
Desktop         Pictures                               Templates
Documents       project-1-cpsc-335-project-group-10    Videos
Downloads       project-2-cpsc-335-project-2-group-10
helloworld.cpp  Public
trinhhoang@trinhhoang-HP-EliteBook-8560p:~/project-2-cpsc-335-project-2-group-10$ atom README.md
trinhhoang@trinhhoang-HP-EliteBook-8560p:~/project-2-cpsc-335-project-2-group-10$ atom maxtime.hh
trinhhoang@trinhhoang-HP-EliteBook-8560p:~/project-2-cpsc-335-project-2-group-10$ g++ maxtime.hh
maxtime.hh:10:9: warning: #pragma once in main file
   10 | #pragma once
      |         ^~~~
trinhhoang@trinhhoang-HP-EliteBook-8560p:~/project-2-cpsc-335-project-2-group-10$ g++ maxtime_test.cc
trinhhoang@trinhhoang-HP-EliteBook-8560p:~/project-2-cpsc-335-project-2-group-10$ ./a.out
load_ride_database still works: passed, score 2/2
filter_ride_vector: passed, score 2/2
greedy_max_time trivial cases: passed, score 2/2
greedy_max_time correctness: passed, score 4/4
exhaustive_max_time trivial cases: passed, score 2/2
exhaustive_max_time correctness: passed, score 4/4
TOTAL SCORE = 16 / 16

trinhhoang@trinhhoang-HP-EliteBook-8560p:~/project-2-cpsc-335-project-2-group-10$ 
```

III. Scatter plot





In this above graph, there have a difference between the greedy and exhaustive runtimes. The greedy algorithm runs sufficiently faster than the exhaustive algorithm. In the two graph results we realize how fast the exhaustive algorithm would explode in time complexity. A jump from 20 to 25 resulted in an increase from 35 seconds to 1327 seconds. This was a huge increase from such a small change in the value of n runtimes. We were expecting it to be more gradual and not really begin to consume vast amount of time resources until the value of n was a lot larger.

IV. A brief proof argument for the time complexity of your two algorithms

a. Analyze your greedy algorithm code

```
std::unique_ptr<RideVector> greedy_max_time
(
const RideVector& rides,
double total_cost)
{
// TODO: implement this function, then delete the return statement below
RideVector todo = rides;                                    // 1 step
std::unique_ptr<RideVector> result(new RideVector);  // 0 step
double result_cost = 0;                                     // 1 step
while (!todo.empty())                                       // (nlogn +1) times
   {
auto current = todo[0];                                     // 1 step
auto iterate = todo.begin();                                // 1 step
auto current_iterate = iterate;                             // 1 step
for (auto ride : todo)                                      // ride.size times
{
if (ride->rideTime()/ride->cost() > current->rideTime()/current->cost())  // 3 steps
{
current = ride;                                                          // 1 step
current_iterate = iterate;                                               // 1 step
}
iterate++;                                                               // 1 step
}
todo.erase(current_iterate);                                    // 1step
if (result_cost + current->cost() <= total_cost)                // 2 steps
{
result->push_back(current);                                     // 1 step
result_cost += current->cost();                                 // 2 steps
}
}
return result;                                                  // 0 step
}
```

SC = 1 + 1 + (nlogn +1)*(1 +1 +1 + ride_size *(3 + max(1+1,0) + 1 +2 + max(1+2,0)
      = 2 + (nlogn +1) (3 + ride_size*(6) + 6)
      = 2 + (nlogn + 1) (9 + ride_size*6)

So, time complexity belongs to O(nlogn)

b. Analyze your exhaustive optimization algorithm code

```
std::unique_ptr<RideVector> exhaustive_max_time
(
const RideVector& rides,
int total_cost)
{
// TODO: implement this function, then delete the return statement below

  const int n = rides.size();                                  // 1 step
  double check_r_c;                                            // 0 step
  double check_t_c;                                            // 0 step
  double best_r_c;                                             // 0 step
  double best_t_c;                                             // 0 step
  double best_c_c;                                             // 0 step

  std::unique_ptr<RideVector> best = std::unique_ptr<RideVector>(new RideVector);   // 0 step
  std::unique_ptr<RideVector> candidate;                                            // 0 step
  for (uint64_t bits = 0; bits < std::pow(2, n); bits++) {                    // takes 2^n times
    candidate = std::unique_ptr<RideVector>(new RideVector);           // 1 step
    for (int j = 0; j < n; j++)                                                         // (n + 1) times
{
      if (((bits >> j) & 1) == 1)                                                        // 3 steps
{
        candidate->push_back(rides[j]);                                          // 1 step
      }
    }
    sum_ride_vector(*candidate, check_r_c, check_t_c);                 // O(n)
    if (check_r_c <= total_cost) {                                                  // 1 step
      sum_ride_vector(*best, best_c_c, best_t_c);                          // O(n)
      if (best->empty() || check_t_c > best_t_c) {                        // 2 steps
        best = std::move(candidate);                                             // 1 step
      }
    }
  }

return best;                                                                                   0 step
}
```

SC = 1 + 2^n * [1 + (n+1)*(3 + max(1,0)) + O(n) +1 + max (O(n),0) + 2 + max(1,0)]

= 1 +2^n * [1 = (n+1) * [1 + (n + 1)*5 + O(n) + 4 + O(n)]

= 1 + 2^n * (2O(n) + 5n +10)

So time complexity belongs to O(2^n)

Answers to the following questions, using complete sentences.

a. **Is there a noticeable difference in the performance of the two algorithms? Which is faster, and by how much? Does this surprise you?**

Ans: There is a considerable difference between the greedy and exhaustive runtimes. Moreover, the greedy algorithm runs sufficiently faster than the exhaustive algorithm. The results were surprising. We didn't realize how fast the exhaustive algorithm would explode in time complexity. A jump from 20 to 25 resulted in an increase from 35 seconds to 1327 seconds. This was a huge increase from such a small change in the value of n runtimes. We were expecting it to be more gradual and not really begin to consume vast amount of time resources until the value of n was a lot larger.

b. **Are your empirical analyses consistent with your mathematical analyses? Justify your answer.**

Ans: Yes, our empirical analysis is consistent with our mathematical analysis because within our mathematical analysis we concluded that the runtime of the greedy algorithm is 0(n log n) while our exhaustive algorithm's runtime complexity was 0(2^n) which is much slower, and the differences in the empirical analysis were much slower in the exhaustive chart. Which makes the two forms of analysis entirely consistent.

c. **Is this evidence consistent or inconsistent with hypothesis 1? Justify your answer.**

Ans: Yes, exhaustive search algorithms can produce correct outputs, but are not entirely feasible to implement, due to the exponential increase in runtime when higher values of n are added.

d. **Is this evidence consistent or inconsistent with hypothesis 2? Justify your answer.**

Ans: Yes, exponential algorithms are extremely slow, and are not entirely practical to use because of the n runtimes behavior when higher values of n are input into the original function. The increases were so much, that the amount of time it takes waiting for it to run make running it basically infeasible, or pointless.