EECS 281 - Winter 2023 Project 2: *“Star Wars: Episode X - A New Heap”*

Due Tuesday, February 21 at 11:59pm

⚠ **Important Note:** There are two parts to this project. In your IDE, make a separate “IDE Project” for each part. If you try to create a single project, you run into a problem of having two copies of `main()` , which won’t compile, etc.

Part A: Galactic Warfare Simulation

⚠ **Important Note:** For Part A, you should always use `std::priority_queue<>` , not your templates from Part B.

Overview

A war over the galaxy has broken out between the Jedi and the Sith. There are many planets in the galaxy, and generals will be deploying Jedi and Sith battalions to attack and defend them. The Sith are ruthless and vicious, and vehemently oppose the peacekeeping forces of the Jedi. When battles occur, they are always initiated by the Sith, who will send their strongest battalion to fight the weakest Jedi battalion on any given planet. Battalion strength is measured by the average Force-sensitivity of the troops in the battalion, but even the worst pitched battles will always result an equal number of troops lost on both sides.

Learning Objectives

- Learn how to use the `std::priority_queue<>` class
- Run input files in stream mode through a simulation
- Run other stream-based algorithms in the simulation
    - Calculate a running median

- Identify the broadest range of Force-sensitivity that could theoretically be encountered in a battle

Command Line Options

Your program, `galaxy`, should take the following case-sensitive command-line options that will determine which types of output to generate. See Output Details for information about each output mode.

- `-v`, `--verbose`

  A command line option that indicates verbose output should be generated.

- `-m`, `--median`

  A command line option that indicates median output should be generated.

- `-g`, `--general-eval`

  A command line option that indicates that the general evaluation output should be generated.

- `-w`, `--watcher`

  A command line option that indicates that movie watcher output should be generated.

Legal Command Lines

`./galaxy < infile.txt > outfile.txt`
Read deployments from `infile.txt` and write output to `outfile.txt`, using default output options.
`./galaxy -v < infile.txt > outfile.txt`
Same, but with verbose output messages. `./galaxy --verbose --general-eval > outfile.txt < infile.txt` Run with verbose output and print a general evaluation at the end.
`./galaxy --median < infile.txt`
Calculate and print the median number of troops lost in battles.
`./galaxy --watcher < infile.txt`
Show "movie watcher" output.
`./galaxy -vmgw < infile.txt`
Run with all output options enabled.

We will not be specifically error-checking your command-line handling; however we expect that your program conforms with the default behavior of getopt_long(). Incorrect command-line handling may lead to a variety of difficult-to-diagnose problems.

Battle Simulation

As you read each battalion deployment from input, your program should see if the new battalion can be **matched** with a battalion previously deployed on the planet. If a match occurs, then the two battalions engage in warfare. A new battalion can be matched with a previously deployed battalion if:

- Both deployments are on the same planet
- The troops deployed were on opposite sides of the Force: one deployment was a Jedi deployment and the other was a Sith deployment
- The Jedi Force-sensitivity is **less than or equal** to the Sith Force-sensitivity.

Other considerations:

- The Sith always instigate fights, and they only fight battalions with Force-sensitivity which they are confident that they can overcome
- When a Sith battalion is deployed and starts a battle with a Jedi battalion already on the planet, this is referred to as an “attack”
- When a Jedi battalion is deployed and a Sith battalion already on the planet starts a battle with it, this is referred to as an “ambush”
- General ID does not matter: a Sith general may conceal their alignment, so one general may deploy troops from either side; this allows the confusing but totally plausible situation where a single general sends both Sith and Jedi troops to the same planet to battle each other

If the newly arriving battalion is a Sith battalion, it will **always** choose to attack the least Force-sensitive Jedi battalion on the planet, given that there is a Jedi battalion with lesser Force-sensitivity. If the newly arriving battalion is a Jedi battalion, it will **always** be ambushed by the most Force-sensitive Sith battalion on the planet, given that there is a Sith battalion with greater Force-sensitivity. In the event of a tie in Force-sensitivity, the battalion that was deployed first (came first in the input file) will be sent to battle first.

When a battle occurs, the battalions trade troops one-for-one, regardless of their Force-sensitivity. That is to say that an equal number of troops from both battalions are eliminated. That number is always equal to the number of troops in the smaller battalion. If one of the battalions survives, it remains on the planet for future possible battles. For example, if a Sith battalion with 20 troops battles a Jedi battalion of 30 troops, the Sith battalion is eradicated and the Jedi battalion remains with 10 troops.

In the event that some troops from the newly deployed battalion survives, it is possible that another battle will break out. If the new battalion is Sith, they will then look to attack another Jedi battalion. If the new battalion is Jedi, they may be ambushed again after defeating the first Sith battalion. This happens until no more battles break out: that is, there are no pairs of Jedi and Sith battalions remaining on the planet such that the Sith Force-sensitivity is greater than or equal to the Jedi Force-sensitivity.

Input

Your program will receive as input a series of "**battalion deployments**," or placements of troops on a certain planet. A **battalion deployment** consists of the following information:

- Timestamp — the time that this deployment order is issued
- General ID — the general who is issuing the deployment order
- Planet ID — the planet which the troops are being deployed to
- Jedi or Sith — Whether the troops being deployed are Jedi or Sith (*Flavor note: in Star Wars, the Jedi are the good team using the Light side of the Force, and the Sith are the bad team on the Dark side of the Force. Click the link below for more info on The Force*)
- Force-sensitivity — the average Force-sensitivity of the troops being deployed
- Quantity — the number of troops being deployed

Input will arrive from standard input (cin). There are two input formats, *deployment list* (DL) and *pseudorandom* (PR). All inputs tested the autograder will be correct and properly formed.

Input File Header

The header is the first four lines of any valid input file will always be in the following format:

```
Common header format for all input files
```
```
COMMENT: <COMMENT AS ONE OR MORE CHARACTERS AND/OR SPACES>
MODE: <INPUT_MODE_AS_TWO_CHARACTERS>
NUM_GENERALS: <NUM_GENERALS_AS_A_POSITIVE_INTEGER>
NUM_PLANETS: <NUM_PLANETS_AS_A_POSITIVE_INTEGER>
```

`<COMMENT>` is a string terminated by a newline, which should be ignored. *(You should comment your test files to explain their purpose.)*

`<INPUT_MODE>` will either be the string `"DL"` or `"PR"`. DL indicates that the rest of input will be in the deployment list format, and PR indicates that the rest of input will be in pseudorandom format. Details for these input formats will be explained below.

`<NUM_GENERALS>` and **`<NUM_PLANETS>`**, respectively, will tell you how many generals and planets will exist. These values will be positive integer values that can be read and stored in an unsigned integer type.

Deployment List Input

In Deployment List mode, the remainder of the input file will be a series of lines with each line representing a unique deployment. DL input files will not specify the number of deployments in the file. Each line will be in the following format:

`<TIMESTAMP> <SITH/JEDI> G<GENERAL_ID> P<PLANET_NUM> F<FORCE_SENSITIVITY> # <NUM_TROOPS>`

The line:

```
0 JEDI G0 P1 F100 #44
```

Can be translated as:

"At timestamp 0, General 0 deploys 44 Jedi troops with Force-sensitivity 100 to Planet 1."

DL Input Error Checking

To detect corrupt deployment orders, you must check for each of the following:

- The **`<GENERAL_ID>`** and **`<PLANET_ID>`** integers are in ranges **`[0, <NUM_GENERALS>)`** and **`[0, <NUM_PLANETS>)`** , respectively
    - e.g. if **`<NUM_GENERALS>`** is 5, then valid general IDs are 0, 1, 2, 3, 4
- **`<FORCE_SENSITIVITY>`** and **`<NUM_TROOPS>`** are greater than 0
- Timestamps are non-decreasing
    - e.g. 0 cannot come after 1, but there can be multiple deployments with the same timestamp

If you detect invalid input at any time during the program, print a helpful message to `cerr` and `exit(1)` . **You do not need to check for input errors not explicitly mentioned here.**

Pseudorandom Input

In pseudorandom mode, the file header is followed by exactly three lines, which specify values that are used to create deployments using a pseudorandom number generator (PRNG). Lines 5-7 will be in the following format:

Pseudorandom input file format

```
COMMENT: <COMMENT>
MODE: PR
```

```
NUM_GENERALS: <NUM_GENERALS>
NUM_PLANETS: <NUM_PLANETS>
RANDOM_SEED: <SEED>
NUM_DEPLOYMENTS: <NUM_DEPLOYMENTS>
ARRIVAL_RATE: <ARRIVAL_RATE>
```

`<SEED>` is an integer used to initialize the random seed.

`<NUM_DEPLOYMENTS>` is the number of deployment orders to generate. This value will be a non-negative integer.

`<ARRIVAL_RATE>` is a non-negative integer corresponding to the maximum number of deployments per timestamp (but the actual number could be less, due to random numbers).

Generating deployments with P2random.h

Psuedorandom input is generated using the `P2random` class in the included `P2random.h` file. After initializing the `P2random` object with the seed, number of deployments, and arrival rate found in lines 5-7 of the input file, a `stringstream` object is filled with the randomly generated deployments. A solution that can read both DL and PR input files will choose to read deployments from the `stringstream` object for PR or from `cin` for DL.

Details for how to use the `P2random` class are provided below, with the full implementation available in `P2random.h` . This file is included to make pseudo-random generation uniform across platforms.

PR input will always be correctly formatted

The `P2random` class is initialized with the following function:

`P2random::PR_init() function declaration from P2random.h`

```
void P2random::PR_init(std::stringstream &ss,     uint32_t seed,
                       uint32_t num_generals,     uint32_t num_planets,
                       uint32_t num_deployments, uint32_t arrival_rate);
```

`P2random::PR_init(...)` will fill the stringstream argument, `ss` , with deployments, so that it can later be used with the stream extraction operator `>>` similar to how `cin` is used.

The following C++ code is helpful in reducing code duplication:

`Using P2random and an istream reference variable to read input for PR or DL`

```
stringstream ss;

// inputMode is the "PR" or "DL" from line 2
```

```
if (inputMode == "PR") {  // inputMode is read from line 2 of the header
  // TODO: include number of generals and number of planets read from lines
  // 3-4 of the header and add code to read random seed, number of
  // deployments, and arrival rate from lines 5-7 of the input file
  P2random::PR_init(ss, seed, num_gen, num_planets, num_deploys, rate);
}  // if ..inputMode

// Create a reference variable that is ALWAYS used for reading input.
// If PR mode is on, refer to the stringstream. Otherwise, refer to cin.
// This is a place where the ternary operator must be used: an equivalent
// if/else is impossible because reference variables must be initialized
// when they are created.
istream &inputStream = inputMode == "PR" ? ss : cin;

// Make sure to read an entire deployment in the while statement
while (inputStream >> var1 >> var2 ...) {
  // Process this deployment, use PQs, initiate battles (if possible), etc.
}  // while ..inputStream
```

DL & PR Comparison

The following two input files are in different modes, but generate **the same deployments** when `P2random` is used:

DL mode input file p2-a-new-heap/spec-DL-in.txt

```
COMMENT: DL mode generating some deployments.
MODE: DL
NUM_GENERALS: 3
NUM_PLANETS: 2
0 JEDI G0 P1 F100 #44
1 JEDI G0 P1 F56 #42
2 SITH G0 P1 F73 #19
2 SITH G0 P0 F34 #50
2 JEDI G0 P0 F86 #23
2 JEDI G0 P0 F20 #39
2 SITH G2 P0 F49 #24
2 JEDI G2 P0 F83 #45
3 JEDI G2 P1 F64 #22
3 JEDI G1 P0 F6 #19
3 JEDI G0 P1 F42 #37
4 JEDI G2 P0 F10 #44
```

PR mode input file p2-a-new-heap/spec-PR-in.txt

```
COMMENT: PR mode generating the same sequence of deployments.
MODE: PR
NUM_GENERALS: 3
NUM_PLANETS: 2
RANDOM_SEED: 104
NUM_DEPLOYMENTS: 12
ARRIVAL_RATE: 10
```

Simple Example

Consider the following series of deployments from an input file in DL mode:

```
Three deployments, explained
```

```
0 SITH G1 P2 F100 #10
0 JEDI G2 P2 F10 #20
0 SITH G3 P2 F1 #10
```

Here is what should happen when these deployments are processed:

1. General 1 deploys battalion 1 with Force-sensitivity 100 on Planet 2 with 10 Sith troops
   - There are no other battalions on the planet yet, so they remain on standby
   - 10 Sith troops are on Planet 2
2. General 2 deploys battalion 2 with Force-sensitivity 10 on Planet 2 with 20 Jedi troops
   - Jedi battalion 2 encounters Sith battalion 1: the Sith battalion has a higher Force-sensitivity, so it instigates a fight with (ambushes) the Jedi battalion 2
   - They do battle and both lose 10 troops, since battles exchange troops one-for-one
   - 10 Jedi troops remain on Planet 2
3. General 3 deploys battalion 3 with Force-sensitivity 1 on Planet 2 with 10 Sith troops
   - Sith battalion 3 encounters Jedi battalion 2: the Sith battalion has a lower Force-sensitivity, so it avoids the fight and remains on standby
   - 10 Jedi troops remain on Planet 2, 10 Sith troops remain on Planet 2

Output Details

The output generated by your program will depend on the command line options specified at runtime. With the exception of startup output and the summary output, all output is optional and should not be generated unless the corresponding command line option is specified.

Startup Output

Your program should always print the following line **before** reading any deployments:

`Deploying troops...`

Summary Output

After all input has been read and all possible battles have been fought, the following output should **always** be printed:

```
---End of Day---
Battles: <NUM_BATTLES>[NEWLINE]
```

`<NUM_BATTLES>` is the total number of battles that have happened over the course of the day.

Verbose Output

If and only if the `--verbose/-v` option is specified on the command line (see above), whenever a battle is completed you should print on a single line:

`General <GENERAL_A_NUM>'s battalion attacked <GENERAL_B_NUM>'s battalion on planet <PLANET_NUM>. <NUM_TROOPS_LOST> troops were lost.`

`<GENERAL_A_NUM>` is the number of the general who deployed the Sith troops

`<GENERAL_B_NUM>` is the number of the general who deployed the Jedi troops

`<NUM_TROOPS_LOST>` is defined to be the total number of troops lost from both sides, or the number of Jedi troops lost + the number of Sith troops lost.

Example:

Given the following list of deployments:

```
0 JEDI G1 P0 F125 #10
0 SITH G2 P0 F1 #100
0 JEDI G3 P0 F100 #10
0 JEDI G4 P0 F80 #10
0 SITH G5 P0 F200 #4
```

No battles are possible until the **5th** battalion is deployed. When the 5th battalion is deployed and a battle occurs, you should print:

```
General 5's battalion attacked General 4's battalion on planet 0. 8 troops were lost.
```

Median Output

If and only if the `--median/-m` option is specified on the command line, whenever there is a change in timestamp your program should print the current median troops lost in a battle for each planet, in ascending order, by planet ID. **If no battles have occurred on a specific planet, do not print the median message for that planet**. In the case that battles have occurred, you should print:

```
Median troops lost on planet <PLANET_ID> at time <TIMESTAMP> is <MED_TROOPS_LOST>.
```

`<PLANET_ID>` is the number of the planet that is being summarized

`<TIMESTAMP>` is the timestamp that has just closed, ie. as soon as the first deployment from timestamp 5 arrives, you should print the median through timestamp 4, if there have ever been any battles on that planet

`<MED_TROOPS_LOST>` is the median number of troops lost in all battles on that planet since the beginning of the day, up to and including the timestamp that has just closed

If an even number of battles occur on a planet, take the average of the middlemost two to compute the median. There will always be an even number of troops lost in a battle (the same number of troops from both sides), so the result will always be an integer, even when divided by two.

Example

Given the following battles:

```
General 5's battalion attacked General 4's battalion on planet 9. 8 troops were lost.
General 2's battalion attacked General 7's battalion on planet 9. 2 troops were lost.
```

The median lost troops for planet 9 after these two battles is $((2+8)/2) = 5$. If the timestamp changed, and the `--median` option was specified, your program should print:

```
Median troops lost on planet 9 at time 0 is 5.
```

General Evaluation Output

If and only if the `--general-eval/-g` option is specified on the command line, **following the summary output**, you should print the following line without any preceding newlines:

```
---General Evaluation---
```

Followed by lines in the following format for ***every*** general in ascending order (0, 1, 2, etc.), even if they did not engage in any battles:

```
General <GENERAL_ID> deployed <NUM_JEDI> Jedi troops and <NUM_SITH> Sith troops,
and <NUM_SURVIVORS>/<NUM_DEPLOYED> troops survived.
```

These numbers are troops across all planets. Example:

```
---General Evaluation---
General 0 deployed 40 Jedi troops and 0 Sith troops, and 20/40 troops
survived.
General 1 deployed 0 Jedi troops and 0 Sith troops, and 0/0 troops survived.
General 2 deployed 60 Jedi troops and 30 Sith troops, and 44/90 troops
survived.
```

Movie-Watcher Output

As a fervent Star Wars fan, you want to find which pairs of battling Jedi and Sith deployments would have made for a maximally-exciting movie. For each planet, your job is to find the pairs of Jedi and Sith deployments that would have had the most "exciting" battles. The most "exciting" battle is one that has the greatest difference in Force-sensitivity between the battling parties. A battle can only happen if the Jedi Force-sensitivity is less than or equal to the Sith Force-sensitivity.

You want to find the most exciting possible Sith **attack** and Sith **ambush** for each planet. A Sith **attack** occurs when the Jedi are deployed to the planet first, and the Sith are deployed to that planet afterward to attack the Jedi. A Sith **ambush** occurs when the Sith are deployed to the planet first, and they surprise Jedi that land on the planet afterward. One deployment comes "after"another one if it appears later in the file, even if the timestamps for the two deployments are the same.

For example, suppose you had these deployments:

```
0 JEDI G1 P0 F10 #10
0 SITH G2 P0 F20 #10
0 JEDI G1 P0 F30 #10
0 SITH G1 P0 F40 #10
```

Then the most exciting Sith attack on planet 0 would be the Sith with Force-sensitivity 40 attacking the Jedi with Force-sensitivity 10, but there would be no most-exciting Sith ambush because it's not possible for a the Sith deployment to be paired against a Jedi deployment that came later.

Notice that an actual battle need not happen. In this output, the first and second deployments would be paired in the regular output, not the first and the fourth. You should report only the most exciting hypothetical battle, regardless of which battles actually occurred.

If and only if the `--watcher/-w` option is specified on the command line, you should print the following line without any preceding newlines once at the very end of the program:

```
---Movie Watcher---
```

Followed by a pair of movie watcher's output lines for every planet in ascending order in the following format:

```
A movie watcher would enjoy an ambush on planet <PLANET_ID> with Sith at time <TIMESTAMP1> and Jedi at time <TIMESTAMP2> with a force difference of <NUM>.
A movie watcher would enjoy an attack on planet <PLANET_ID> with Jedi at time <TIMESTAMP1> and Sith at time <TIMESTAMP2> with a force difference of <NUM>.
```

When finding exciting battles, the number of troops is not taken into consideration. If there would be more than one battle with the maximal level of excitement (i.e., difference in Force-sensitivity), you should prefer the battle with the lower **`<TIMESTAMP2>`**. If the second timestamps are equal, then use the lower **`<TIMESTAMP1>`**.

If there are no exciting battles (there are no pairs of Jedi/Sith deployments on a given planet, or none result in the Jedi Force-sensitivity being less than or equal to the Sith Force-sensitivity) you should print a message like one of the following, as appropriate:

```
A movie watcher would not see an interesting ambush on planet <PLANET_ID>.
A movie watcher would not see an interesting attack on planet <PLANET_ID>.
```

Detailed Simulation Algorithm

Following these steps in order will help guarantee that your program prints the correct output at the proper times.

The `CURRENT_TIMESTAMP` starts at 0, and is maintained throughout the run of the program.

1. Print program startup output
2. Read the next deployment from input
3. If the new deployment's TIMESTAMP is not the CURRENT_TIMESTAMP

   a. If the `--median/-m` option is specified, print the median information b. Set `CURRENT_TIMESTAMP` to be the new deployment's `TIMESTAMP`.
4. Instigate all possible fights between the Jedi and Sith battalions

   i. If the `--verbose/-v` option is specified, you should print the details of each completed battle to `stdout/cout`

5. Repeat steps 2-4 until there are no more deployments to be made.
6. Output median information **again** if the `--median` option was specified.
7. Print end-of-day summary output.
8. Output the general evaluation if the `--general-eval` option was specified.
9. Output the movie-watcher's output if the `--watcher` option was specified.

Full Example

Full example input file

```
COMMENT: DL mode generating some deployments.
MODE: DL
NUM_GENERALS: 3
NUM_PLANETS: 2
0 JEDI G0 P1 F100 #44
1 JEDI G0 P1 F56 #42
2 SITH G0 P1 F73 #19
2 SITH G0 P0 F34 #50
2 JEDI G0 P0 F86 #23
2 JEDI G0 P0 F20 #39
2 SITH G2 P0 F49 #24
2 JEDI G2 P0 F83 #45
3 JEDI G2 P1 F64 #22
3 JEDI G1 P0 F6 #19
3 JEDI G0 P1 F42 #37
4 JEDI G2 P0 F10 #44
```

Output when run with `-v`, `-m`, `-g`, and `-w`

```
Deploying troops...
General 0's battalion attacked General 0's battalion on planet 1. 38 troops were lost.
General 0's battalion attacked General 0's battalion on planet 0. 78 troops were lost.
Median troops lost on planet 0 at time 2 is 78.
Median troops lost on planet 1 at time 2 is 38.
General 2's battalion attacked General 1's battalion on planet 0. 38 troops were lost.
Median troops lost on planet 0 at time 3 is 58.
Median troops lost on planet 1 at time 3 is 38.
General 2's battalion attacked General 2's battalion on planet 0. 10 troops were lost.
General 0's battalion attacked General 2's battalion on planet 0. 22 troops were lost.
```

```
Median troops lost on planet 0 at time 4 is 30.
Median troops lost on planet 1 at time 4 is 38.
---End of Day---
Battles: 5
---General Evaluation---
General 0 deployed 185 Jedi troops and 69 Sith troops, and 127/254 troops
survived.
General 1 deployed 19 Jedi troops and 0 Sith troops, and 0/19 troops
survived.
General 2 deployed 111 Jedi troops and 24 Sith troops, and 95/135 troops
survived.
---Movie Watcher---
A movie watcher would enjoy an ambush on planet 0 with Sith at time 2 and
Jedi at time 3 with a force difference of 43.
A movie watcher would enjoy an attack on planet 0 with Jedi at time 2 and
Sith at time 2 with a force difference of 29.
A movie watcher would enjoy an ambush on planet 1 with Sith at time 2 and
Jedi at time 3 with a force difference of 31.
A movie watcher would enjoy an attack on planet 1 with Jedi at time 1 and
Sith at time 2 with a force difference of 17.
```

The `std::priority_queue<>`

The STL `std::priority_queue<>` data structure is an efficient implementation of the binary heap which you will code in Part B. To declare a `std::priority_queue<>` you need to state either one or three types:

1. The data type to be stored in the container. If this type has a natural sort order that meets your needs, this is the only type required.
2. The underlying container to use, usually just a `vector<>` of the first type.
3. The comparator to use to define what is considered the highest priority element.

If the type that you store in the container has a natural sort order (i.e. it supports `operator<()`), the `priority_queue<>` will be a max-heap of the declared type. For example, if you just want to store integers, and have the largest integer be the highest priority:

```
priority_queue<int> pqMax;
```

When you declare this, by default the underlying storage type is `vector<int>` and the default comparator is `less<int>` . If you want the smallest integer to be the highest priority:

```
priority_queue<int, vector<int>, greater<int>> pqMin;
```

If you want to store something other than integers, define a custom comparator.

About Comparators

The functor must accept two of whatever is stored in your priority queue: if your PQ stores integers, the functor would accept two integers. If your PQ stores pointers to units, your functor would accept two pointers to orders (actually two const pointers, since you don't have to modify units to compare them).

Your functor receives two parameters, let's call them a and b. It must always answer the following question: **is the priority of a less than the priority of b**? What does lower priority mean? It depends on your application. For example, refer back to Battle Simulation if you have multiple Jedi deployed on the same planet, which Jedi has the highest priority if a Sith arrives? In the same way, Sith have a different way of determining the highest priority. This means you will need at least two different functors: one for a priority queue containing Jedi and a different functor for a priority queue containing Sith.

When you would have wanted to write a comparison, such as:

```
        if (data[i] < data[j])
```

You would instead write:

```
        if (this->compare(data[i], data[j])
```

Your priority queues must work **in general**. In general, a priority queue has no idea what kind of data is inside of it. That's why it uses `this->compare` instead of `<`. What if you wanted to perform the comparison `if (data[i] > data[j])`? Use the following:

```
        if (this->compare(data[j], data[i])
```

Submitting your solution

Project Identifier

You MUST include the project identifier at the top of all source and header files that you submit:

```
// Project Identifier: AD48FB4835AF347EB0CA8009E24C3B13F8519882
```

Do all of your work (with all needed files, as well as test files) in some directory other than your home directory. This will be your "submit directory".)

Libraries and Restrictions

We highly encourage the use of the STL for part A, with the exception of these prohibited features:

- The thread/atomics libraries (e.g., boost, pthreads, etc) which spoil runtime measurements.
- Smart pointers (both unique and shared).

In addition to the above requirements, you may not use any STL facilities which trivialize your implementation of your priority queues, including but not limited to `priority_queue<>` , `make_heap()` , `push_heap()` , `pop_heap()` , `sort_heap()` , `partition()` , p `artition_copy()` , `stable_partition()` , `partial_sort()` , or `qsort()` . However, you **may** (and probably should) use `sort()` . Your main program (Part A) **must** use `priority_queue<>` , but your PQ implementations (Part B) **must not**. If you are unsure about whether a given function or container may be used, ask on Piazza.

Testing and Debugging

Part of this project is to prepare several test files that expose defects in a solution. Each test file is an input file. We will give your test files as input to intentionally buggy solutions and compare the output to that of a correct project solution. You will receive points depending on how many buggy implementations your test files expose. The autograder will also tell you if one of your test files exposes bugs in your solution. For the first such file that the autograder finds, it will give you the correct output and the output of your program.

Test File Details

Your test files must be Deployment List input files, and may have no more than 30 deployments in any one file. You may submit up to 10 test files (though it is possible to expose all buggy solutions with fewer test files).

Test files should be named in the following format:

```
test-<N>-<OPTION>.txt
```

- **`<N>`** is an integer in the range $[0, 9]$
- **`<OPTION>`** is one (and only one) of the following characters `v` , `m` , `g` , or `w` .
    - This tells the autograder which command-line option to test with.

For example, `test-0-w.txt` and `test-5-v.txt` are both valid test file names.

The tests on which the autograder runs your solution are NOT limited to 30 deployments in a file; your solution should not impose any size limits (as long as memory is available)

Submitting to the Autograder

Do all of your work (with all needed files, as well as test files) in some directory other than your home directory. This will be your “submit directory” Before you turn in your code, be sure that:

- Every source code and header file contains the following project identifier in a comment at the top of the file:

  ```
  // Project Identifier: AD48FB4835AF347EB0CA8009E24C3B13F8519882
  ```

- The Makefile must also have this identifier (in the first TODO block).
- DO NOT copy the above identifier from the PDF! It might contain hidden characters. Copy it from the README file from Canvas instead.
- Your makefile is called `Makefile` . Typing `make -R -r` builds your code without errors and generates an executable file called `galaxy` . (The command-line options `-R` and `-r` disable automatic build rules; these automatic rules do not exist on the autograder).
- Your Makefile specifies that you are compiling with the gcc optimization option `-O3` . This is extremely important for getting all of the performance points, as `-O3` can speed up code by an order of magnitude.
- Your test files are named as described and no other project file names begin with test. Up to 10 test files may be submitted.
- The total size of your program and test files does not exceed 2MB.
- You don’t have any unnecessary files (including temporary files created by your text editor and compiler, etc) or subdirectories in your submit directory (e.g., the .git folder used by git source code management).
- Your code compiles and runs correctly using the g++ compiler. This is available on the CAEN Linux systems (that you can access via login.engin.umich.edu. Even if everything seems to work on another operating system or with different versions of GCC, the course staff will not support anything other than GCC running on CAEN Linux. At the moment, the default version installed on CAEN is 4.8.5, however we want you to use version 6.2.0 (available on CAEN with a command and/or Makefile); this version is also installed on the autograder machines.

For Part A (galactic combat simulation), turn in all of the following files:

- All your .h(pp) and or .cpp files for the project (NOT including your priority queue implementations)
- Your Makefile

- Your test files

For Part B (priority queues), turn in all of the following files:

- Your priority queue implementations: SortedPQ.h, BinaryPQ.h, PairingPQ.h
- Your `Makefile` (actually optional, it includes itself, but we'll replace this with our `Makefile`)

If **any** of your submitted priority queue files do not compile, **no** unit testing (Part B) can be performed.

You must prepare a compressed tar archive (.tar.gz file) of all of your files to submit to the autograder. One way to do this is to have all of your files for submission (and nothing else) in one directory. Our Makefile provides the command `make fullsubmit`. Alternately you can go into this directory and run this command:

```
dos2unix *; tar czf ./submit.tar.gz *.cpp *.h *.hpp Makefile test*.txt
```

This will prepare a suitable file in your working directory.

Submit your project files directly to either of the two autograders at:

https://g281-1.eecs.umich.edu/ or https://g281-2.eecs.umich.edu/. **When the autograders are turned on and accepting submissions, there will be an announcement on Piazza.** The autograders are identical and your daily submission limit will be shared (and kept track of) between them. You may submit up to two times per calendar day, per part (A or B; more per day in Spring). If you use a late day to extend one part, the other part is automatically extended also. For this purpose, days begin and end at midnight (Ann Arbor local time). **We will use your best submission for final grading.** If you would instead like us to use your LAST submission, see the autograder FAQ page, or use this form.

Please make sure that you read all messages shown at the top section of your autograder results! These messages often help explain some of the issues you are having (such as losing points for having a bad Makefile or why you are segfaulting). Also be sure to check whether the autograder shows that one of your own test files exposes a bug in your solution.

Grading

- 60 pts total for part A (all your code, using the STL, but not using your priority queues)
    - 45 pts — correctness & performance
    - 5 pts — no memory leaks
    - 10 pts — student-provided test files to find bugs in our code (and yours!)
- 40 pts total for part B (our main, your priority queues)

  - 20 pts — pairing heap correctness & performance
  - 5 pts — pairing heap has no memory leaks
  - 10 pts — binary heap correctness & performance
  - 5 pts — sorted heap correctness & performance

In your autograder output for Part A, the section named “Scoring student test files” will tell you how many bugs exist, how many are needed to start earning points, earn full points, and earn an extra submit per day (for Part A).

Part B: Priority Queues

The Part B Spec is in a separate document. The solution to Part B will be submitted separately from Part A, and should be developed separately in your development environment, with the files from both solutions stored in separate directories.

Final Notes

If you’re asked about “cool projects” in an interview, note that this is simply a different wrapper on a more “marketable” project: stock market simulation. Sith and Jedi are like buyers and sellers, planets are commodities, and generals are stock market traders!

The “movie watcher mode” is a particular type of algorithm called a “streaming algorithm” where we can only look at each input once, and cannot make a copy of all the input. This is similar to a stock market simulation also, where if we could go back in time, we could determine the optimal point at which to buy and then later sell the same commodity (planet) to achieve the highest profit margin.