

# EECS 281 - Project 2A: Stock Market Simulation



*Due Friday, October 11, 2024 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.

## Project Overview

There are **two** parts to Project 2. Part A is an emulation of an electronic stock exchange market. Part B requires you to make multiple implementations of the priority queue abstract data type.

### Part A Goals

- Gain experience using a priority queue.
- Become more proficient with object-oriented design.
- Become more proficient using the STL, especially `std::priority_queue<>`.

### Part B Goals

- Implement multiple versions of the priority queue and compare their performances.
- Learn a data structure (the pairing heap) from a paper. This type of task is something that programmers have to do in their careers.
- Gain experience writing templated classes and subclasses.

## Command Line Interface

Your program market should take the following case-sensitive command-line options that will determine which types of output to generate. There are no arguments required or allowed for any of the output options. Details about each output mode are under the Output Details section.

- `-v`, `--verbose`

An optional flag that indicates verbose output should be generated

- `-m`, `--median`

An optional flag that indicates median output should be generated

- `-i` , `--trader_info`

An optional flag that indicates that the trader details output should be generated

- `-t` , `--time_travelers`

An optional flag that indicates that time travelers' output should be generated

## Legal Command Line Examples

```
1 ./market < infile.txt > outfile.txt
2 ./market --verbose --trader_info < infile.txt
3 ./market --verbose --median > outfile.txt
4 ./market --time_travelers
5 ./market --trader_info --verbose
6 ./market -vmit
```

## Illegal Command Line Examples

```
./market -v -q
```

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.

## Market Logic


Your program market will receive a series of “**orders**,” or intentions to buy or sell shares of a certain stock. An order consists of the following information:

- Timestamp - the timestamp that this order comes in
- Trader ID - the trader who is issuing the order
- Stock ID - the stock that the trader is interested in
- Buy/Sell Intent - whether the trader wants to buy or sell shares
- Price Limit - the max/min amount the trader is willing to pay/receive per share
- Quantity - the number of shares the trader is interested in

As each order comes in, your program should see if the new order can be matched with any previous orders. A new order can be matched with a previous order if:

- Both orders are for the same stock ID
  - Trader ID does not matter; traders are allowed to buy from themselves.
- The buy/sell intentions are different. That is, one trader is buying and the other trader is selling.
- The selling order's price limit is less than or equal to the buying order's price limit.

A buyer will always try to buy for the lowest price possible, and a seller will always try to sell for the highest price possible. Functionally, this means that whenever a possible match exists, it will **always** occur at the price of the *earlier* order. When trying to match orders, use priority queues to identify the *lowest-price* seller and the *highest-price* buyer.

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

If the SELL order arrived first, then the price of the match will be at the price listed by the seller. If the BUY order arrived first, the match price will be the price listed by the buyer.

In the event of a tie (e.g. the two cheapest sellers are offering for the same price), **always match using the order that arrived earliest.**

## Simple Example

Consider the following series of orders:

1. Trader 1 wants to buy 10 shares of stock 2 for up to \$100 each
2. Trader 2 wants to sell 20 shares of stock 2 for at least \$10 each
3. Trader 3 wants to buy 10 shares of stock 2 for up to \$1 each

(see [Input](#) below for actual input formatting)

Here is an explanation of what should happen:

1. Trader #1 enters the market with the intent to buy 10 shares of stock #2 for up to \$100/share
  - There are no other orders in the market yet, so trader #1 posts his order to the stock exchange to wait for a matching order.
2. Trader #2 enters the market with the intent to sell 20 shares of stock #2 for as low as \$10 per share
  - Trader #2 sees the order left by trader #1, and decides to first sell 10 shares of the stock to trader #1 for \$100 each.
  - Even though she would have been happy selling for \$10/share, she wants to make as much money as possible and takes advantage of the fact that trader #1 is willing to pay more.
  - After selling her 10 shares, she still has 10 shares she wants to get rid of. So, she now posts an order to sell her 10 leftover shares for \$10 per share to wait for a matching order.
3. Trader #3 enters the market with the intent to buy 10 shares of stock #2 for only \$1/share.
  - Trader #3 sees the order left by trader #2 to sell for \$10 each, but is not willing to pay that much.
  - As a result, Trader #3 posts an order to wait for a matching order.

## Input

Input will arrive from standard input ( `cin` ). There are two input formats, trade list (TL) and pseudorandom (PR). All numeric values within an input file will always fit within a 4-byte integer variable ( `int` ).

## Input File Header

The first four lines of input will always look the same, with no syntax errors, additions, or omissions, regardless of input format:

Common header format for all input files

```
1  COMMENT: <COMMENT AS ONE OR MORE CHARACTERS AND/OR SPACES>
2  MODE: <INPUT_MODE_AS_TWO_CHARACTERS>
3  NUM_TRADERS: <NUM_TRADERS_AS_A_POSITIVE_INTEGER>
4  NUM_STOCKS: <NUM_STOCKS_AS_A_POSITIVE_INTEGER>
```

<COMMENT...> is a string terminated by a newline, which should be ignored. Each input file will have exactly one line of comments. You should add a meaningful comment your test files to explain their purpose.

<INPUT\_MODE...> will either be the string "TL" or "PR" (without the quote marks). TL indicates that the rest of input will be in the trade list format, and PR indicates that the rest of input will be in pseudo-random format. Details for these input formats will be explained shortly.

<NUM\_TRADERS> and <NUM\_STOCKS>, respectively, will tell you how many traders and stocks will exist. These are integers.

## Trade List Input Mode

If <INPUT\_MODE...> is TL, the rest of the input file will be a series of lines in the following format:

```
<TIMESTAMP> <BUY/SELL> T<TRADER_ID> S<STOCK_NUM> $<PRICE> #<QUANTITY>
```

Each line represents a unique order. For example, the line:

```
0 BUY T1 S2 $100 #50
```

reads:

*"At timestamp 0, trader 1 is willing to buy 50 shares of stock 2 for up to \$100/share."*

Each line which describes an order will be well-formatted, meaning that it might have invalid values, but not an unreadable format. For example, there will always be a `T` before the `<TRADER_ID>`, always be an `S` before the `<STOCK_NUM>`, etc. The things you expect to be integers will always be integers (not strings, not floating-point values with a decimal point).

When you want to read a trade, DO NOT `getline()` the entire line, copy it to a stringstream, then extract what you want one piece at a time: that's processing the same input three times. Instead, use `>>` to extract exactly what you need. Starting out, a trade needs some type of integer (the timestamp), a string ( `BUY` or

SELL ), a char for the T before the trader number, some type of integer for the trader number, another char for the S , etc.

## TL Input Errors

You will always find a number where you expect to find one (though they might be invalid values). To avoid unexpected losses of large amounts of money, you must check for each of the following:

- `<TIMESTAMP>` is non-negative (no one can trade before the market opens)
- Timestamps are non-decreasing
  - e.g. 0 cannot come after 1, but there can be multiple orders with the same timestamp
- `<TRADER_ID>` is in the range `[0, <NUM_TRADERS>)`
  - e.g. if `<NUM_TRADERS>` is 5, then valid trader IDs are 0, 1, 2, 3, 4
- `<STOCK_ID>` is in the range `[0, <NUM_STOCKS>)`
  - ibid.
- `<PRICE>` and `<QUANTITY>` are both positive

If you detect an input error 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 Mode

If `<INPUT_MODE>` is PR, the rest of input will consist of these three lines:

Pseudorandom generator value format for PR input files

```
1  RANDOM_SEED: <RANDOM_SEED>
2  NUMBER_OF_ORDERS: <NUM_ORDERS>
3  ARRIVAL_RATE: <ARRIVAL_RATE>
```

**`<RANDOM_SEED>`** — A number integer used to initialize the random seed.

**`<NUMBER_OF_ORDERS>`** — The number of orders to generate.

**`<ARRIVAL_RATE>`** — The average number of orders per timestamp.

All three of these values are unsigned integers.

***PR input will always be correctly formatted.***

## Using P2random.h

In the project folder, we provide a pair of files to generate the orders in PR mode. This pseudorandom order generator (PROG) is to make sure that the generation of pseudo-random numbers is consistent across platforms. The class P2random contains the following function:

`PR_init()` initializes the pseudorandom order generator

```

1 void P2random::PR_init(std::stringstream& ss, unsigned int seed,
2 unsigned int num_traders, unsigned int num_stocks,
3 unsigned int num_orders, unsigned int arrival_rate);

```

`P2random::PR_init()` will set the contents of the stringstream argument ( `ss` ) so that you can use it just like you would use `cin` for TL mode.

You may find the following C++ code helpful in reducing code duplication:

#### Useful code snippet for eliminating duplicate code

```

1  ...
2  // First read Input File Header (mode, num_traders, num_stocks)
3  ...
4
5  // Create a stringstream object in case the PROG is used
6  stringstream ss;
7
8  if (mode == "PR") {
9      // TODO: Read PR mode values from cin (seed, num_orders, arrival_rate)
10     ...
11     // Initialize the PROG and populate ss with orders
12     P2random::PR_init(ss, seed, num_traders, num_stocks, num_orders, rate);
13 } // if ..mode
14
15 // Call the function with either the stringstream produced by PR_init()
16 // or cin
17 if (mode == "PR")
18     processOrders(ss); // This is a separate function you must write
19 else
20     processOrders(cin);
21
22 ...
23 }
24
25 // TODO: Create a separate function as referenced above, accepting a stream
26 //       reference variable, to which you will pass cin or a stringstream
27 //       that is populated by PR_init()
28 void processOrders(istream &inputStream) {
29     // Read orders from inputStream, NOT cin
30     while (inputStream >> var1 >> var2 ...) {
31         // process orders
32         ...
33     } // while ..inputStream
34 } // processOrders()
35

```

## Spec Example Inputs

The following two input files are in different modes, but produce the same buy/sell orders; thus they generate the same output.

Spec Example, Transaction List Input Mode: p2-stocks/spec-input-TL.txt

```
1  COMMENT: Spec Example, TL mode generating 12 orders.
2  MODE: TL
3  NUM_TRADERS: 3
4  NUM_STOCKS: 2
5  0 SELL T1 S0 $100 #44
6  1 SELL T2 S0 $56 #42
7  2 BUY T1 S0 $73 #19
8  2 BUY T2 S0 $34 #50
9  2 SELL T2 S1 $86 #23
10 2 SELL T0 S0 $20 #39
11 2 BUY T1 S1 $49 #24
12 2 SELL T1 S1 $83 #45
13 3 SELL T0 S1 $64 #22
14 3 SELL T2 S1 $6 #19
15 3 SELL T0 S1 $42 #37
16 4 SELL T1 S0 $10 #44
```

Spec Example, Pseudorandom Input Mode: p2-stocks/spec-input-PR.txt

```
1  COMMENT: Spec Example, PR mode generating the same 12 orders.
2  MODE: PR
3  NUM_TRADERS: 3
4  NUM_STOCKS: 2
5  RANDOM_SEED: 104
6  NUMBER_OF_ORDERS: 12
7  ARRIVAL_RATE: 10
```

## 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:

```
Processing orders...
```

### Summary Output

After all input has been read and all possible trades completed, the following output should always be printed without any preceding newlines before any optional end of day output:

```
1  ---End of Day---
2  Trades Completed: <TRADES_COMPLETED><NEWLINE>
```

**<TRADES\_COMPLETED>** is the total number of trades completed over the course of the trading day. The number of shares traded doesn't matter; any seller and any buyer making a trade of matched orders adds one to this count.

## Verbose Output

If and only if the `--verbose/-v` option is specified on the command line (see [Command Line Interface](#)), whenever a trade is completed you should print on a single line:

```
Trader <BUYING_TRADER_NUM> purchased <NUM_SHARES> shares of Stock <STOCK_NUM> from Trader
<SELLING_TRADER_NUM> for $<PRICE>/share
```

### Verbose Output Example

Given the following list of orders:

```
1  0 SELL T1 S0 $125 #10
2  0 BUY T2 S0 $1 #100
3  0 SELL T3 S0 $100 #10
4  0 SELL T4 S0 $80 #10
5  0 BUY T5 S0 $200 #4
```

**No trades** are possible until the 5th order comes in. When the 5th order comes in, you should print:

```
Trader 5 purchased 4 shares of Stock 0 from Trader 4 for $80/share
```

## Median Output

If and only if the `--median/-m` option is specified on the command line, at the times detailed in the Market Logic section, your program should print the current median match price for all stocks in ascending order by stock ID. **If no trades have been made for a given stock, it does not have a median, and thus nothing should be printed for that stock's median.** In the case that a median does exist, you should print:

```
Median match price of Stock <STOCK_ID> at time <TIMESTAMP> is $<MEDPRICE>
```

If there are an even number of trades, take the average of the middle-most and **use integer division** to compute the median. **If a particular timestamp in the input file produces no new trades, you still print a median for that timestamp (limited by the rules above);** see [Detailed Algorithm](#).

### Median Output Example

Given the following transactions:



```
Trader 5 purchased 4 shares of Stock 9 from Trader 4 for $80/share
Trader 2 purchased 1 shares of Stock 9 from Trader 10 for $50/share
```

The median match price for Stock 9 after these two transactions is  $$(80 + 50 / 2)$  or \$65. If the timestamp changed and the `--median/-m` option was specified, you should print:

```
Median match price of Stock 9 at time 0 is $65
```

The median match price only considers transactions, and does not consider the quantity traded in each trade. You must keep a running median, and do it efficiently! Simply sorting the values every time you need the middle value(s) will take too much time.

## Trader Info Output

If and only if the `--trader_info/-i` option is specified on the command line, **following the summary output**, you should print the following line without any preceding newlines.

```
---Trader Info---
```

This is followed by a line printed for every trader in ascending order (0, 1, 2, etc.):

```
Trader <TRADER_ID> bought <NUMBER_BOUGHT> and sold <NUMBER_SOLD> for a net transfer of
$<NET_VALUE_TRADED>(newline)
```

These numbers consider orders across all stocks.

## Trader Info Example

```
1  ---Trader Info---
2  Trader 0 bought 0 and sold 2 for a net transfer of $166
3  Trader 1 bought 63 and sold 0 for a net transfer of $-5359
4  Trader 2 bought 24 and sold 85 for a net transfer of $5193
```

The numbers bought and sold, and net value traded include all stocks that the trader happened to trade in. The total number bought by all traders should equal the total number sold by all traders, and the total of all the net transfers should be 0. This shows that our simulation is a zero-sum game.

## Time Travelers Output

In time-travel trading, we want to find the ideal times that a time traveler theoretically could have bought shares of a stock and then later sold that stock to maximize profit per share.

If and only if the `--time_travelers/-t` option is specified on the command line, you should print the following line without any preceding newlines:

```
---Time Travelers---
```

This is followed by time traveler's output for every stock in ascending order (0, 1, 2, etc.):

A time traveler would buy Stock <STOCK\_ID> at time <TIMESTAMP1> for \$<PRICE1> and sell it at time <TIMESTAMP2> for \$<PRICE2>

<TIMESTAMP1> will correspond to an actual sell order that came in during the day, and <TIMESTAMP2> will correspond to an actual buy order that came after the sell order that maximizes the time traveler's profit.

- When calculating the results for time traveler trading, the only factors are the time and price of orders that happened throughout the day, quantity is not considered.
- If there would be more than one answer that yields the optimal result, you should report the buy/sell pair with the lowest <TIMESTAMP1> and <TIMESTAMP2> . This implies the first selection when ties occur for either of the buy/sell prices, and the earliest matched pair if multiple matches return the same amount of profit.

If there are no valid buy/sell order pairs (none exist, or none result in a profit) you should print:

A time traveler could not make a profit on Stock <STOCK\_ID>

## Time Traveler Output Example

Suppose you had these orders:

```
1  0 SELL T1 S0 $10 #10
2  0 BUY  T2 S0 $20 #10
3  0 SELL T1 S0 $30 #10
4  0 BUY  T1 S0 $40 #10
```

The most profitable time traveler trades of stock 0 would be to buy shares for \$10, and then later sell them for \$40. *\*Notice that an actual matched pair of orders need not occur (the \$10 shares are no longer available when the \$40 buyer arrives). You should report only the most profitable \*hypothetical trade, regardless of which trades actually happen.*

Your time traveler must work in  $\Theta(n)$  time ( $\Theta(1)$  per buy/sell order), using  $\Theta(1)$  memory per stock.

## Detailed Algorithm

Following these steps in order will help guarantee that your program prints the correct output at the proper times. Details of the output options are covered in the [Output Details](#) section.

Initialize `CURRENT_TIMESTAMP` to 0, its value is updated throughout the run of the program, and even though the first order may not arrive at timestamp 0, no valid orders can arrive before that time.

1. Print [Startup Output](#)
2. Read the next order from input
3. If the new order's `TIMESTAMP != CURRENT_TIMESTAMP` a. If the `--median/-m` option is specified, print the median price of all stocks that have been traded on at least once by this point in ascending order by stock ID ([Median Output](#)) b. Set `CURRENT_TIMESTAMP` to be the new order's `TIMESTAMP` .

4. Add the new order to the market
5. Make all possible matches in the market that now includes the new order (observing the fact that a single incoming order may make matches with multiple existing orders) a. If a match is made and the `--verbose/-v` option is specified, print [Verbose Output](#) b. Remove any completed orders from the market c. Update the share quantity of any partially fulfilled order in the market
6. Repeat steps 2-5 until there are no more orders
7. If the `--median/-m` flag is set, output final timestamp median information of all stocks that were traded that day ([Median Output](#))
8. Print [Summary Output](#)
9. If the `--trader_info/-i` flag is set print [Trader Info Output](#)
10. If the `--time_travelers/-t` flag is set print [Time Travelers Output](#)

## Full Spec Example

Given the Spec Example input in either format (TL input shown below):

p2-stocks/spec-input-TL.txt

```
1  COMMENT: Spec example, TL mode generating 12 orders.
2  MODE: TL
3  NUM_TRADERS: 3
4  NUM_STOCKS: 2
5  0 SELL T1 S0 $100 #44
6  1 SELL T2 S0 $56 #42
7  2 BUY T1 S0 $73 #19
8  2 BUY T2 S0 $34 #50
9  2 SELL T2 S1 $86 #23
10 2 SELL T0 S0 $20 #39
11 2 BUY T1 S1 $49 #24
12 2 SELL T1 S1 $83 #45
13 3 SELL T0 S1 $64 #22
14 3 SELL T2 S1 $6 #19
15 3 SELL T0 S1 $42 #37
16 4 SELL T1 S0 $10 #44
```

the output when run with command line options: `--verbose` , `--median` , `--trader_info` , and `--time_travelers` would be:

Full Spec Example Output: p2-stocks/spec-output-all.txt

```
1  Processing orders...
2  Trader 1 purchased 19 shares of Stock 0 from Trader 2 for $56/share
3  Trader 2 purchased 39 shares of Stock 0 from Trader 0 for $34/share
4  Median match price of Stock 0 at time 2 is $45
5  Trader 1 purchased 19 shares of Stock 1 from Trader 2 for $49/share
6  Trader 1 purchased 5 shares of Stock 1 from Trader 0 for $49/share
```

```

7 Median match price of Stock 0 at time 3 is $45
8 Median match price of Stock 1 at time 3 is $49
9 Trader 2 purchased 11 shares of Stock 0 from Trader 1 for $34/share
10 Median match price of Stock 0 at time 4 is $34
11 Median match price of Stock 1 at time 4 is $49
12 ---End of Day---
13 Trades Completed: 5
14 ---Trader Info---
15 Trader 0 bought 0 and sold 44 for a net transfer of $1571
16 Trader 1 bought 43 and sold 11 for a net transfer of $-1866
17 Trader 2 bought 50 and sold 38 for a net transfer of $295
18 ---Time Travelers---
19 A time traveler would buy Stock 0 at time 1 for $56 and sell it at time 2 for $73
20 A time traveler could not make a profit on Stock 1

```

## 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 orders 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 [Market Logic](#) if you have multiple sell orders for a stock, which order has the highest priority if a buyer arrives? In the same way, buy orders 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 sell orders and a different functor for a priority queue containing buy orders.

## 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: 0E04A31E0D60C01986ACB20081C9D8722A1899B6
```

### 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 in Part B, including but not limited to `priority_queue<>`, `make_heap()`, `push_heap()`, `pop_heap()`, `sort_heap()`, `partition()`, `partition_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 valid Transaction List input files, and may have no more than 30 orders** 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`, `t`, or `i`.
  - This tells the autograder which command-line option to test with.

For example, `test-0-m.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: 0E04A31E0D60C01986ACB20081C9D8722A1899B6
```
- The Makefile must also have this identifier (in the first TODO block).
- 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](http://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.

**For Part A** (stock market 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 the 281 autograder at:

<https://eecs281ag.eecs.umich.edu/>. 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.









