Programming Assignment Four: Electronic Trading

Out: November 19, 2016; Due: December 7, 2016.

I. Motivation

1. Get familiar with the various data structures provided by STL: priority queues (`std::priority_queue`), hash tables (`std::unordered_map`, `std::unordered_set`, `std::unordered_mutlimap`, `std::unordered_multiset`), and binary search trees (`std::map`, `std::set`, `std::multimap`, `std::multiset`).
2. Get experience in choosing proper data structures.
3. Become more proficient at testing and debugging code.

II. Programming Overview

In this project, you are asked to write a program to help facilitate the trading of equities on an electronic exchange market. The market offers a variety of equities. Any market client can place a **buy** or **sell** order on equity to request that a transaction be executed when matching sellers or buyers become available. Your program should take in **buy** and **sell** orders for a variety of equities as they arrive and match buyers with sellers to execute trades as quickly as possible.

III. Input Format

The input will arrive from standard input (`cin`), not from an ifstream. The input contains a series of orders that will be presented such that orders with lower timestamps always appear first. Orders will be formatted as follows:

```
TIMESTAMP CLIENT_NAME BUY_OR_SELL EQUITY_SYMBOL $PRICE #QUANTITY DURATION
```

on a single line, with all fields separated by one or more spaces/tabs. For example:

```
0 Jack BUY GOOG $100 #50 2
```

The definition of each field is:

1. `TIMESTAMP` - A non-negative integer value corresponding to the time. Its unit is second.
2. `CLIENT_NAME` - The buyer or seller's name. This will be a string that contains only alphanumeric characters and '_'.
3. `BUY_OR_SELL` - The string "BUY" or the string "SELL", corresponding to the type of order.
4. `EQUITY_SYMBOL` - The shorthand name of the equity. This will be a string that contains 1-5 characters that are alphanumeric character, '_', or '.' (examples: C, F, GM, KO, TGT, WMT, AAPL, PZZA, BRK.A, BRK.B)
5. `PRICE` - This is a positive integer. If it is a buy order, this is the **highest** price a buyer is willing to pay for the equity. If it is a sell order, this is the **lowest** price the seller is willing to sell the equity for. Buyers may pay less than their limit price, sellers might get more money than the minimum they ask for. The $ sign will appear in the input before this value.
6. `QUANTITY` - A positive integer representing the number of shares the client wants to buy/sell. The # sign will appear in the input before this value.
7. `DURATION` - An integer value indicating how long the order will stay in the market. There are 3 possible cases. For `DURATION = -1`, the order will never expire and stay forever in the market until it is matched for transaction completely. For `DURATION = 0`, the order is an Immediate Or Cancel (IOC) order which needs to be matched for transaction immediately, partially or completely. Any remaining quantity is canceled. For `DURATION > 0`, the order will expire and exit the market right before `timestamp+DURATION`.

All valid input will be arranged by timestamp (lowest timestamp first). As you read in orders, you should assign all orders a unique ID number, such that the first order you read gets an ID of 0, the second an ID of 1, and so on. These ID numbers ensure that there is only one possible matching of buyers and sellers based on arrival timestamps. They are also useful for tracking and debugging. **You do not need to check for invalid input.**

IV. Specifications

Market Logic

A variable `CURRENT_TIMESTAMP` is maintained throughout the run of the program. It always starts at 0.

Your program must perform as follows:

1. Read the next order from input.
2. If the order's `TIMESTAMP != CURRENT_TIMESTAMP` then:
    a. If the `--median` option is specified, print the median price of all equities that have been traded on at least once by this point in the lexicographical order by `EQUITY_SYMBOL` (see below).
    b. If the `--midpoint` option is specified, print the midpoint price for all equities that have had at least one order placed by this point in the simulation in the lexicographical order by `EQUITY_SYMBOL` (see below).
    c. Set `CURRENT_TIMESTAMP` equal to the order's `TIMESTAMP` .
    d. If any orders have expired, remove them from your data structure.
3. Match the order you just read with the unexpired orders that are stored in your data structure. If the order is not fully filled and it is not an IOC order, you should add the order into your data structure to match with future orders.
4. Repeat previous steps until the end of the day, defined as when there is no more input (and thus no more trades to be performed).
5. Treat the end of day like the timestamp has moved again, and output median and midpoint information as necessary.
6. Print all end of day output.

Orders and Trades

All orders on the market are considered limit orders. A buy limit order expresses the intent to buy at most N shares of a given stock at no more than D dollars per share, where D is called the limit. A sell limit order expresses the intent to sell at most N shares of a given stock at no less than D dollars per share. To facilitate trade execution, an order can be split and matched with several other orders at different execution prices.

Order books and trade execution

For each equity, you should keep track of all current buy and sell orders in a dedicated container data structure called an order book. A trade can be executed for a given equity when the buy order with the highest limit price (and in the event of ties, lowest ID number) has a buy limit price greater than or equal to the sell limit price of the sell order with the lowest limit price (and

in the event of ties, lowest ID number). If this happens, then a trade is executed between the buy order with the highest limit price (and in the event of ties, lowest ID number) and the sell order with the lowest limit price (and in the event of ties, lowest ID number). For a given order book, the order of trade execution is fully determined. You must support this order correctly and ensure high speed of trade execution by optimizing data structures.

For example, given the following orders as input:

```
0 SELLER_1 SELL GOOG $125 #10 -1
0 SELLER_2 SELL GOOG $100 #30 -1
0 SELLER_3 SELL GOOG $100 #15 -1
0 BUYER_1 BUY GOOG $200 #4 -1
0 BUYER_2 BUY GOOG $250 #50 -1
0 SELLER_4 SELL GOOG $60 #20 -1
```

The first trade to be executed would be `BUYER_1` buying 4 of `SELLER_2`'s shares. This is because at this point, the program has not read `BUYER_2`'s order yet (see market logic section) and `SELLER_2` has the lowest sell price. While `SELLER_2` and `SELLER_3` are both selling at the same price, `SELLER_2`'s order arrived first, and will therefore have a lower ID number.

When that first trade is executed, `SELLER_2`'s order must be revised to offer only 26 shares (because 4 had already been traded). The revised order keeps the id of the original order.

Whenever a trade is executed, the match price of the trade is the limit price of the order (buy or sell) with the lower ID number. In this case, `BUYER_1` offered to buy for $200 and `SELLER_2` offered to sell for `$100`; because `SELLER_2` has a lower ID number, the trade will be executed at a match price of `$100` per share.

For another example, suppose the input is as follows:

```
0 BUYER_1 BUY GOOG $100 #10 -1
0 BUYER_2 BUY GOOG $125 #30 -1
0 BUYER_3 BUY GOOG $125 #15 -1
0 SELLER_1 SELL GOOG $120 #4 -1
0 SELLER_2 SELL GOOG $110 #4 -1
```

The first trade to be executed would be `BUYER_2` buying 4 of `SELLER_1`'s shares with a match price of $125. This is because at this point, the program has not read `SELLER_2`'s order yet (see market logic section) and `BUYER_2` has the highest buy price. While `BUYER_2` and `BUYER_3` are both buying at the same price, `BUYER_2`'s order arrived first, and will therefore have a lower

ID number. Since BUYER_2 has a lower ID number than SELLER_1, the trade will be executed at a match price of $125 per share.

Order expiration and IOC orders

Each order comes into the order book with a DURATION at which to stay in the order book. If DURATION = -1, then the order stays indefinitely. If DURATION > 0 then the order should be removed from the order book right before the CURRENT_TIMESTAMP = TIMESTAMP + DURATION. If we had the following sequence of orders:

```
0 SELLER_1 SELL GOOG $125 #10 2
1 BUYER_1 BUY GOOG $200 #5 -1
2 BUYER_2 BUY GOOG $150 #5 -1
```

Then first BUYER_1 would buy 5 shares of GOOG from SELLER_1 for $125, but before BUYER_2 can buy the remaining shares, SELLER_1's order will expire, and so only one trade will happen for this sequence of 3 orders.

If DURATION = 0, then the order is an IOC order which means whatever part of the order can transact should transact, but any remaining quantity should be canceled and not placed in the order book. If we had the following sequence of trades:

```
0 SELLER_1 SELL GOOG $125 #5 -1
0 BUYER_1 BUY GOOG $200 #10 0
0 SELLER_2 SELL GOOG $150 #5 -1
```

BUYER_1 will buy 5 shares of GOOG from SELLER_1 for $125 with its IOC order, but the remaining 5 shares will expire before SELLER_2 will be able to sell its shares. If BUYER_1's order DURATION were changed from 0 to 1, the order sequence looked like:

```
0 SELLER_1 SELL GOOG $125 #5 -1
0 BUYER_1 BUY GOOG $200 #10 1
0 SELLER_2 SELL GOOG $150 #5 -1
```

and all of the orders will transact.

Commission fees

For providing this matching service, the market (and thus your program) also takes a commission fee of 1% from every completed trade from both the buyer and the seller. Suppose in a trade, a buyer purchased 4 shares of equity for $100 per share from a seller. The commission fee is:

`(100·4)/100 = $4` from both the buyer and seller. So the buyer will pay `(100·4)+4 = $404` and the seller will receive `(100·4)−4 = $396`, and the market will earn a commission of `(2·4) = $8`. For these calculations, all values should be integers and therefore all decimals will be truncated. Commissions must be computed exactly as above when executing the trade. To be clear, do not calculate the combined commissions of the buyer and seller in a single arithmetic expression (such as *total_commission = 2 · match_price · num_shares/100*), as this may yield different results when truncating. Instead, first calculate the commission as shown and then multiply the result by 2.

V. Program Arguments

Your program should be named as `market`. It should take the following case-sensitive command-line options:

1. `-v, --verbose`: An optional flag that indicates the program should print additional output information while trades are being executed (see output section for more details).
2. `-m, --median`: An optional flag that indicates the program should print the current median match price for each equity at the times specified in the Market Logic section above (see output section for more details).
3. `-p, --midpoint`: An optional flag that indicates the program should print the current midpoint price for each equity that has had at least one order placed for it at times specified in the Market Logic section above (see output section for more details).
4. `-t, --transfers`: An optional flag that indicates the program should print additional output information at the end of the day to show the net amount of funds transferred by all clients (see output section for more details).
5. `-g, --ttt EQUITY_SYMBOL`: An optional flag that may appear **more than once** with different equity symbols as arguments. This option requests that at the end of the day the program determines what was the best time to buy (once) and then subsequently sell (once) a particular equity during the day to maximize profit. More information is in the output section.

If multiple options are specified that produce output at the end of the program, the output should be printed in the order that they are listed here (e.g., `--transfers` before `--ttt`).

Examples of legal command lines:

- `./market < infile.txt > outfile.txt`
- `./market --verbose --transfers > outfile.txt`
- `./market -v -t > outfile.txt`
- `./market --verbose --median > outfile.txt`
- `./market --transfers --verbose --ttt GOOG --ttt IBM`

We will not be specifically error-checking your command-line handling, but we expect that your program conforms with the default behavior of `getopt_long` (see http://www.gnu.org/software/libc/manual/html_node/Getopt.html#Getopt). Incorrect command-line handling may lead to a variety of difficult-to-diagnose problems.

VI. Output Format

All the outputs are printed through `cout`.

Default

At the end of the day, after all inputs have been read and all possible trades completed, the following output should always be printed before any optional end of day output:

```
---End of Day---
Commission Earnings: $COMMISION_EARNINGS
Total Amount of Money Transferred: $MONEY_TRANSFERRED
Number of Completed Trades: NUMBER_OF_COMPLETED_TRADES
Number of Shares Traded: NUMBER_OF_SHARES_TRADED
```

with the corresponding values being in the output text. The dollar signs should be printed in the output where indicated. The `MONEY_TRANSFERRED` value should not include commissions.

For example, suppose by the end of the day, only two trades have happened: `BUYER_1` purchased 4 shares of equity `EQ_1` from `SELLER_1` for $100 per share and `BUYER_2` purchased 8 shares of equity `EQ_2` from `SELLER_2` for $40 per share. Then the total amount of money transferred is $4 \cdot 100 + 8 \cdot 40 = 720$. The number of completed trades is 2 and the number of shares traded is 4+8=12.

Verbose Option

If and only if the `--verbose` option is specified on the command line, whenever a trade is completed you should print:

```
BUYER_NAME purchased NUMBER_OF_SHARES shares of EQUITY_SYMBOL
from SELLER_NAME for $PRICE/share
```

on a single line. In the following example:

```
0 SELLER_1 SELL GOOG $125 #10 -1
0 SELLER_2 SELL GOOG $100 #10 -1
0 SELLER_3 SELL GOOG $100 #10 -1
0 SELLER_3 SELL GOOG $80 #10 0
0 BUYER_1 BUY GOOG $200 #4 -1
```

you should print:

```
BUYER_1 purchased 4 shares of GOOG from SELLER_2 for $100/share
```

No trades can be executed on an equity for a given `CURRENT_TIMESTAMP` if there is no buyer willing to pay the lowest asking price of any seller. An example of such a scenario is:

```
0 BUYER_1 BUY GOOG $100 #50 -1
0 SELLER_1 SELL GOOG $200 #10 -1
0 BUYER_2 BUY GOOG $150 #3 -1
0 SELLER_2 SELL GOOG $175 #30 -1
```

Median Option

If and only if the `--median` option is specified on the command line, at the times described in the Market Logic section (above), your program should print the current median match price of all completed trades for each equity that were executed in the time interval `[0, CURRENT_TIMESTAMP]`. To be clear, this is the median of the match prices of the trades themselves. This does not consider the quantity traded in each trade. Equities with lexicographically smaller `EQUITY_SYMBOL`s must be printed first. If no matches have been made on a particular equity, do not print a median for it. If there are an even number of trades, take the average of the middle two to compute the median. The output format is:

```
Median match price of EQUITY_SYMBOL at time CURRENT_TIMESTAMP is
$MEDIAN_PRICE
```

Midpoint Option

If and only if the `--midpoint` option is specified on the command line, at the times described in the Market Logic section (above), your program should print the current price midpoint for each equity that has had at least one order placed in the time interval `[0, CURRENT_TIMESTAMP]` in lexicographical order. The midpoint of an equity is an integer average between the highest price of buy orders, and the lowest price of sell orders that are still active for the given equity, which is calculated exactly by `(HighestBuy + LowestSell)/2` for the same equity. The division here is the integer division in C++. The output format is:

```
Midpoint of EQUITY_SYMBOL at time CURRENT_TIMESTAMP is
$MIDPOINT_PRICE
```

If an equity currently has zero buy or zero sell orders, then output a line like the following:

```
Midpoint of EQUITY_SYMBOL at time CURRENT_TIMESTAMP is undefined
```

As an example, if the following sequence of orders were placed:

```
0 PlanetExpress SELL CAR $120 #1 -1
0 BluthCorp SELL CAR $110 #1 -1
0 KrustyKrab BUY CAR $80 #1 -1
0 BluthCorp BUY CAR $105 #1 -1
1 PlanetExpress SELL CAR $80 #2 -1
2 BluthCorp BUY CAR $70 #1 -1
```

The midpoint output would be the following:

```
Midpoint of CAR at time 0 is $107
Midpoint of CAR at time 1 is undefined
Midpoint of CAR at time 2 is $90
```

After time 0, the highest buy order is from `BluthCorp` for `$105`, and the lowest sell order is also from `BluthCorp` for `$110`. The midpoint is `(110 + 105)/2 = 107` (with integer math). At time `1 PlanetExpress` sells to both buy orders, so there are no buy orders at the end of time 1 and therefore the quote is undefined. At time `2, BluthCrop` places a new buy order for `$70`. Now the midpoint is `(70 + 110)/2= 90.`

Transfers Option

If and only if the `--transfers` option is specified on the command line, you should print the following information at the end of the day for each client who placed an order during the run of the program:

```
CLIENT_NAME bought NUMBER_OF_STOCKS_BOUGHT and sold
NUMBER_OF_STOCKS_SOLD for a net transfer of $NET_VALUE_TRADED
```

This should be printed such that clients with lexicographically smaller client names are printed first. The `NET_VALUE_TRADED` does not include commissions.

Time-Travel Trading Option

If and only if the `--ttt` option (Time-Travel Trading) is specified on the command line, you should do the following.

If the `--ttt` option is specified more than once, you should print the results for each `EQUITY_SYMBOL` that is given in the same order that they were given in the command line. In other words, if the command line had first `--ttt MSFT` and then `--ttt IBM`, you would print `MSFT`'s information before `IBM`'s. We will not specify the same equity twice.

In time travel trading, you are a time traveler that wants to find the ideal times that you "could have" bought an equity and then later "could have" sold that equity to maximize profit (or if it is not possible to make a profit, to do this while minimizing losses). Your program will print a `TIMESTAMP1` and a `TIMESTAMP2` corresponding to the times you "could have" placed orders to do this.

What this means is that `TIMESTAMP1` will be the same as some actual sell order that came in during the day, and that `TIMESTAMP2` will be the same as some actual buy order that came in after the sell order (i.e., with a higher ID number). The assumption is that the time traveler "would have" placed those orders immediately after the actual orders.

When calculating the results for time travel trading, the only factors are the time and price of orders that happened throughout the day. Quantity is not considered. One way to think about this is to imagine (only for the purpose of time travel trading) that all orders are for unlimited quantity.

If there would be more than one answer that yields the optimal result, you should prefer the answer with the lowest ID. If during the day there is not at least one actual sell order followed by

at least one actual buy order, then `TIMESTAMP1` and `TIMESTAMP2` should both be printed as `-1`.

The output format is as follows:

```
Time travelers would buy EQUITY_SYMBOL at time: TIMESTAMP1 and
sell it at time: TIMESTAMP2
```

As an example, if the following sequence of orders were placed:

```
0 SELLER_1 SELL GOOG $10 #5 -1
1 BUYER_1  BUY  GOOG $20 #8  -1
2 SELLER_1 SELL GOOG $12 #10 -1
3 BUYER_1  BUY  GOOG $16 #3  -1
4 SELLER_1 SELL GOOG $8 #10 -1
5 BUYER_1  BUY  GOOG $16 #2  -1
6 SELLER_1 SELL GOOG $9 #7 -1
7 BUYER_1  BUY  GOOG $19 #4  -1
```

Then the time-travel trading for `GOOG` should output:

```
Time travelers would buy GOOG at time: 4 and sell it at time: 7
```

A Complete Example

Input:

```
0 PlanetExpress SELL AMD $120 #32 1
0 BadWolfCorp BUY GE $200 #20 8
0 BluthCorp BUY AMD $100 #50 10
1 KrustyKrab BUY AMD $130 #10 7
1 PlanetExpress SELL GE $150 #50 6
1 PlanetExpress BUY NFLX $80 #15 6
3 BluthCorp SELL AMZN $50 #22 -1
4 BadWolfCorp SELL GE $50 #15 -1
4 BadWolfCorp SELL AMZN $100 #30 10
4 KrustyKrab BUY AMZN $130 #12 0
4 BadWolfCorp BUY AMZN $50 #30 5
5 BadWolfCorp SELL AMZN $50 #5 0
5 BluthCorp BUY AMD $150 #25 0
```

```
6 PlanetExpress SELL AMD $80 #100 -1
6 BadWolfCorp BUY AMD $120 #10 1
6 KrustyKrab BUY GE $110 #10 3
```

Output when run with `--verbose`, `--median`, `--midpoint`, `--transfers`, and `--ttt AMZN`:

```
Midpoint of AMD at time 0 is $110
Midpoint of GE at time 0 is undefined
BadWolfCorp purchased 20 shares of GE from PlanetExpress for
$200/share
Median match price of GE at time 1 is $200
Midpoint of AMD at time 1 is undefined
Midpoint of GE at time 1 is undefined
Midpoint of NFLX at time 1 is undefined
Median match price of GE at time 3 is $200
Midpoint of AMD at time 3 is undefined
Midpoint of AMZN at time 3 is undefined
Midpoint of GE at time 3 is undefined
Midpoint of NFLX at time 3 is undefined
KrustyKrab purchased 12 shares of AMZN from BluthCorp for
$50/share
BadWolfCorp purchased 10 shares of AMZN from BluthCorp for
$50/share
Median match price of AMZN at time 4 is $50
Median match price of GE at time 4 is $200
Midpoint of AMD at time 4 is undefined
Midpoint of AMZN at time 4 is $75
Midpoint of GE at time 4 is undefined
Midpoint of NFLX at time 4 is undefined
BadWolfCorp purchased 5 shares of AMZN from BadWolfCorp for
$50/share
Median match price of AMZN at time 5 is $50
Median match price of GE at time 5 is $200
Midpoint of AMD at time 5 is undefined
Midpoint of AMZN at time 5 is $75
Midpoint of GE at time 5 is undefined
Midpoint of NFLX at time 5 is undefined
KrustyKrab purchased 10 shares of AMD from PlanetExpress for
$130/share
BluthCorp purchased 50 shares of AMD from PlanetExpress for
$100/share
BadWolfCorp purchased 10 shares of AMD from PlanetExpress for
$80/share
KrustyKrab purchased 10 shares of GE from BadWolfCorp for
$50/share
```

```
Median match price of AMD at time 6 is $100
Median match price of AMZN at time 6 is $50
Median match price of GE at time 6 is $125
Midpoint of AMD at time 6 is undefined
Midpoint of AMZN at time 6 is $75
Midpoint of GE at time 6 is undefined
Midpoint of NFLX at time 6 is undefined
---End of Day---
Commission Earnings: $258
Total Amount of Money Transferred: $12950
Number of Completed Trades: 8
Number of Shares Traded: 127
BadWolfCorp bought 45 and sold 15 for a net transfer of $-4800
BluthCorp bought 50 and sold 22 for a net transfer of $-3900
KrustyKrab bought 32 and sold 0 for a net transfer of $-2400
PlanetExpress bought 0 and sold 90 for a net transfer of $11100
Time travelers would buy AMZN at time: 3 and sell it at time: 4
```

VII. Hints and Implementation Requirements

We highly encourage the use of the STL (priority queues `(std::priority_queue)`, hash tables `(std::unordered_map, std::unordered_set, std::unordered_mutlimap, std::unordered_multiset)`, and binary search trees `(std::map, std::set, std::multimap, std::multiset)`) for this project, with the exception of two prohibited features: The C++11 regular expressions library and the thread/atomics libraries (which spoil runtime measurements). Do not use other libraries (e.g., boost, pthreads, etc).

VIII. Source Code Files

You have the freedom to define all of your source code files.

IX. Compiling

Since `std::unordered_map` is a new feature of C++ 11, you should add the option `-std=c++11` to the `g++` compiling command. Make sure the version of your `g++` is high enough. Otherwise it may not support that option.

X. Testing

We have supplied an input file called `test.txt` for you to test your market program. It can be found in the Programming-Assignment-Four-Related-Files.zip. To do this test, type the following into the Linux terminal once your program has been compiled:

```
./market --verbose --median --midpoint --transfers --ttt AMZN < test.txt > result.txt
```

The correct output can be found in the file `out.txt`. This is the minimal amount of tests you should run to check your program. Those programs that do not pass this test are not likely to receive much credit. You should also write other different test cases yourself to test your program extensively.

You should also check whether there is any memory leak. For those programs that behave correctly but have memory leaks, we will deduct some points.

XI. Performance Report

In order to test whether you choose proper data structures in this project, you are required to write a performance report. We have provided to you 5 test cases named as `test01.txt`, ..., `test05.txt` in the Programming-Assignment-Four-Related-Files.zip. Your task is to run each of these files with the options `--verbose`, `--median`, `--midpoint`, and `--transfers` and report some values in your output and the runtime of your program.

When you compile your program, you should add the `-O2` option to your `g++` compiling command. This will let the compiler do some optimization to your program.

The command for this test is (using `test01.txt` as an example):

```
./market --verbose --median --midpoint --transfers < test01.txt > out01.txt
```

Note that you do not print the output on the screen, but write it to a file. The runtime is the end-to-end runtime of you program. You can use `clock()` function to get the runtime. See http://www.cplusplus.com/reference/ctime/clock/

The report should only contain a table of the following format:

Testcase #	Time (s)	$Commission earnings	#Completed trades	#Output lines
1				
2				
3				
4				
5				

The unit of the time is second. To count the number of lines in an output file, use the command:

```
wc -l OUT_FILE.txt
```

These test cases will not be used in the online judge system. They are only for performance tests. However, you shouldn't fake your answer! We will randomly choose some students to run their programs and compare the results with their reported results. If we find that the difference is quite large between your reported result and our measurement, we will consider this as an **honor code violation**. Of course, we are aware of the difference between different machines, so we will set a reasonable threshold.

XII. Submitting and Due Date

You should submit all the source code files and a `Makefile` via the online judgment system. The `Makefile` compiles a program named `market`. See announcement from the TAs for more details about the submission. The due time is 11:59 pm on December 7th, 2016.

XIII. Grading

Your program will be graded along four criteria:

1. Functional Correctness (75%)
2. Implementation Constraints (Violation of constraints will cause a deduction of 50%)
3. General Style (5%)
4. Performance (20%)

Functional Correctness is determined by running a variety of test cases against your program, checking your solution using our automatic testing program. We will grade Implementation Constraints to see if you have met all of the implementation requirements and restrictions. In this assignment, we will also check whether your program has memory leak. For those programs that behave correctly but have memory leaks, we will deduct some points. General Style refers to the

ease with which TAs can read and understand your program, and the cleanliness and elegance of your code. Part of your grade will also be determined based on the performance of your algorithm by reading your performance report. Each test case is assigned with 4 points, 1 for the correctness and 3 for the runtime. Correctness is determined by checking the output values you put in the table against the correct values. If it is not correct, you lose all the 4 points. Runtime score has four levels: 0 for exceeding the time limit, 1 for low performance, 2 for medium performance, and 3 for high performance.