Citadel *2020 Apr29*

Chicago session (9am HKT)

Implement an datafeed analyser that reads tick by tick data from a source (may be a file or a live datafeed), then print out the maximum transaction price, minimum transaction price and transaction quantity every integral minute. The protocol is defined as following (it is ASCII text, not binary protocol). Integral minute means from `09:30:01.000` to `09:30:02.000`.

```
012345678901234567890123456 <- index for ease of counting
09:30:01.123 HSBC 45.6 1000
09:30:01.234 HSBC 45.6 5000
09:30:01.378 CASH 0.16 1000
09:30:02.023 ASMP 98.6 4000
```

Implement a solution that can be compiled on spot by coderpad, but no need to run on spot. Define all classes needed.

```
struct timestamp
{
	int h;
	int m;
	int s;
	int ms;

	inline bool is_same_minute(const timestamp& rhs)
	{
		return h == rhs.h && m == rhs.m;
	}
};

struct tick // All are transaction ticks.
{
	tick(const std::string& str) // This is ASCII, not binary, hence no memcpy needed.
	{
		size_t pos0 = str.find(13, " ");
		size_t pos1 = str.find(pos0+1, " ");

		t = timestamp{ str.substr(0,2), str.substr(3,2), str.substr(6,2), str.substr(9,3) };
		symbol = str.substr(13, pos0-13);
		price = std::stod(pos0+1, pos1-(pos0+1));
		size = std::stoi(pos1+1);
	}

	timestamp t;
	std::string symbol;
	double price;
	int size;
};

struct result
{
	double max_price;
	double min_price;
	int quant;
};

// Different data sources, all provide eof() and read_a_tick() functions.
class database_source{};
class datafeed_source{};
class file_source
{
	file_source(std::ifstream& ifs) : ifs(ifs) {}
	bool eof() { return ifs.eof(); }

	tick read_a_tick()
	{
		std::string str;
		std:getline(ifs, str); // Use global function, as ifstream::getline supports c-str only.
		return tick(str);
	}

	std::ifstream& ifs;
};

template<typename SRC> class tick_manager
{
	tick_manager(SRC& source) : source(source) {}

	void run()
	{
		timestamp t;
		while(!source.eof())
		{
			auto x = source.read_a_tick();
```

```
                        // time checking and print result of previous minute
                        if (!t.is_same_minute(x.timestamp))
                        {
                              print_result_map();
                              result_map.clear();
                        }
                        t = x.timestamp;

                        auto iter = result_map.find(x.symbol);
                        if (iter == result_map.end())
                        {
                              result_map[x.symbol] = tick{ x.price, x.price, x.size };
                        }
                        else
                        {
                              if (x.price > iter->second.max_price) iter->second.max_price = x.price;
                              if (x.price < iter->second.min_price) iter->second.min_price = x.price;
                              iter->second.quant += x.size;
                        }
                  }
            }

            void print_result_map() const
            {
                  for(const auto& x:result_map) std::cout << x.first << x.second.max_price << ...
            }

            SRC& source;
            std::unordered_map<std::string, result> result_map;
      };
```

Extension : Add different analysers to the class, so that all of them run together in single thread, some analysers print out every second or every minute, while some print out by dayend. Probably we need inheritance tree of analysers, manager should keep a heterogenous container of analysers :

- manager should register each analyser before run
- manager should invoke each analyser for each parsed tick

```
class analyser
{
      virtual add_one_tick(const tick& x) = 0;
};

class max_min_integral_minute_analyser : public analyser
{
      virtual add_one_tick(const tick& x)
      {
            // if t and x.timestamp aren't in the same minute, display and reset result_map
            // then update max_price, min_price and quant in result_map
      }
};

class moving_average_continuous_second_analyser : public analyser
{
      virtual add_one_tick(const tick& x)
      {
            // keep unordered_map<symbol, std::queue<timestamp, tick>> for each symbol
            // remove all ticks older than 1 sec, then update and print moving average for each new tick
      }
};

template<typename SRC> class tick_manager
{
      tick_manager(SRC& source) : source(source) {}

      void register(std::shared_ptr<analyser> sp_analyser)
      {
            analysers.insert(sp_analyser);
      }

      void run()
      {
            timestamp t;
            while(!source.eof())
            {
                  auto x = source.read_a_tick();
                  for(auto& sp_analyser : analysers) sp_analyser->add_one_tick(x);
            }
      }

      SRC& source;
      std::unordeded_set<std::shared_ptr<analyser>> analysers;
};
```

London session (4pm HKT)
Given a matrix of sorted integers, find two numbers which sum equal to a given target.

```
std::vector<std::vector<int>> mat;
mat.push_back(std::vector<int>{ 1, 4, 6, 8, 9,10,12});
mat.push_back(std::vector<int>{14,17,19,21,23,27,30});
mat.push_back(std::vector<int>{32,36,37,39,41,43,46});
mat.push_back(std::vector<int>{47,49,50,51,52,55,58});
mat.push_back(std::vector<int>{60,61,63,67,69,73,77});

int target = 67;
auto ans = find_target(mat, target); // Hint : ans is an optional type
if (ans)    std::cout << "numbers are " << ans->first << " " << ans->second;
else        std::cout << "numbers cannot be found";
```

First thing that came to my mind is Kadane algorithm, however this is not necessary, we don't need to build a histogram as the numbers are sorted, we should do better than that by making use of the sorted property. Though both Kadane algo and the following solution are *O(N)*, the following solution is faster as no hash map is needed.

```
std::optional<std::pair<int,int>> find_target(const std::vector<std::vector<int>>& mat, int target)
{
	// Two pointers pointing to UL and LR corner (y-coord & x-coord) respectively.
	std::pair<int,int> i(0, 0);
	std::pair<int,int> j(mat.size()-1, mat.back().size()-1);
	while(i!=j)
	{
		auto n = mat[i.first][i.second];
		auto m = mat[j.first][j.second];
		if	(n+m < target) increment(i, mat);
		else if (n+m > target) decrement(j, mat);
		else return std::make_optional(std::make_pair(n, m));
	}
	return std::nullopt;
}

void increment(std::pair<int,int>& i, const std::vector<std::vector<int>>& mat)
{
	++i.second;
	if (i.second == mat[i.first].size())
	{	++i.first;
		i.second = 0;
	}
}

void decrement(std::pair<int,int>& i, const std::vector<std::vector<int>>& mat)
{
	--i.second;
	if (i.second < 0)
	{	--i.first;
		i.second = mat[i.first].size()-1;
	}
}
```

Can we enhance the algorithm so that it can return multiple pairs of numbers? Answer :

```
void find_target(const std::vector<std::vector<int>>& mat, int target, std::vector<std::pair<int,int>>& ans)
{
	// Two pointers pointing to UL and LR corner (y-coord & x-coord) respectively.
	std::pair<int,int> i(0, 0);
	std::pair<int,int> j(mat.size()-1, mat.back().size()-1);

	while(i!=j)
	{
		auto n = mat[i.first][i.second];
		auto m = mat[j.first][j.second];
		if	(n+m < target) increment(i, mat);
		else if (n+m > target) decrement(j, mat);
		else
		{	// move either i or j, move the one with smaller step
			auto i0 = i; increment(i0, mat);
			auto j0 = j; decrement(j0, mat);
			if (mat[i0.first][i0.second]-n < m-mat[j0.first][j0.second]) i = i0;
			else j = j0;
			ans.push_back(std::make_pair(n, m));
		}
	}
}
```

Can we enhance the algorithm so that it can handle unsorted integers? Answer : it becomes Kadane algorithm.

Hong Kong session (5pm HKT)
Given the following header messages for Hong Kong protocol, Japan protocol and Singapore protocol :

```
#pragma pack(push, 1)
struct header_hk
{
	std::int64_t seq_num;
};
struct header_jp
{
	std::int32_t seq_num;
	std::uint8_t msg_type; // different msg_type have separate sequences of seq_num
};
struct header_sg
{
	char timestamp[9]; // format = HHMMSSsss
};
#pragma pack(pop)
```

What is the meaning of directive `#pragma pack(push,1)` and `#pragma pack(pop)`? Compiler tends to lineup data members on 2 bytes or 4 bytes boundaries by introducing zero padding, making it easier and faster for processors to handle. Yet we can disable zero padding by putting structure definition between 2 directives : `#pragma pack(push,1)` and `#pragma pack(pop)`. Now given abstract handler class, implement concrete handler classes (one for each message type) by filling `is_new_packet()`.

```
class handler_base
{
	void process(const std::byte* buffer, std::size_t length)
	{
		num_packets_read += is_new_packet(buffer, length);
	}

	virtual bool is_new_packet(const std::byte* buffer, std::size_t length) = 0;

	std::uint64_t num_packets_read = 0;
};
```

The three message types need different sequence checking mechanisms. Assume that we are using binary protocol. Core function for parsing binary protocol is0 `memcpy(destination, source, size)`. Here is the solution.

```
class handler_hk : public handler_base
{
	virtual bool is_new_packet(const std::byte* buffer, std::size_t length)
	{
		if (length < sizeof(msg.seq_num)) return false;
		memcpy(msg.seq_num, buffer, sizeof(msg.seq_num));
		if (msg.seq_num > latest_seq_num)
		{
			latest_seq_num = msg.seq_num;
			return true;
		}
		else return false;
	}

	header_hk msg;
	std::int64_t latest_seq_num = -1;
};

class handler_jp : public handler_base
{
	virtual bool is_new_packet(const std::byte* buffer, std::size_t length)
	{
		if (length < sizeof(msg.seq_num) + sizeof(msg.msg_type)) return false;
		memcpy(msg.seq_num, buffer, sizeof(msg.seq_num));
		memcpy(msg.msg_type, buffer, sizeof(msg.msg_type));
		auto iter = latest_seq_nums.find(msg.msg_type);
		if (iter == latest_seq_nums.end())
		{
			latest_seq_nums[msg.msg_type] = msg.seq_num;
			return true;
		}
		else if (msg.seq_num > iter->second)
		{
			iter->second = msg.seq_num;
			return true;
		}
		else return false;
	}

	header_jp msg;
	std::unordered_map<std::uint8_t, std::int32_t> latest_seq_nums;
};
```

However this implementation is not optimum. Can we group the two `memcpy` into one? Yes, replaced them by :

```
memcpy(msg, buffer, sizeof(msg));
```

Why is this possible? This implementation works only if compiler does not add zero padding in between data members. This can guaranteed by the directive `#pragma pack(push,1)` and `#pragma pack(pop)`. However copying isn't fast enough, is there any even faster method? Yes, by reinterpret casting … no copying is involved.

```
const header_jp* msg_ptr = reinterpret_cast<const header_jp*>(buffer);
auto iter = latest_seq_nums.find(msg_ptr->msg_type);
if (iter == latest_seq_nums.end())
{
      latest_seq_nums[msg_ptr->msg_type] = msg_ptr->seq_num;
      return true;
}
else if (msg_ptr->seq_num > iter->second)
{
      iter->second = msg_ptr->seq_num;
      return true;
}
else return false;
```

Lets finish the final message.

```
struct timestamp
{
      std::uint16_t h;
      std::uint16_t m;
      std::uint16_t s;
      std::uint16_t ms;
};

bool operator > (const timestamp& lhs, const timestamp& rhs)
{
      if (lhs.h > rhs.h) return true;
      if (lhs.h == rhs.h)
      {
            if (lhs.m > rhs.m) return true;
            if (lhs.m == rhs.m)
            {
                  if (lhs.s > rhs.s) return true;
                  if (lhs.s == rhs.s)
                  {
                        if (lhs.ms > rhs.ms) return true;
                  }
            }
      }
      return false;
};

class handler_sg : public handler_base
{
      virtual bool is_new_packet(const std::byte* buffer, std::size_t length)
      {
            if (length < sizeof(header_sg)) return false;

            timestamp current_time;
            current_time.h = std::stoi(std::string(buffer+0,2));
            current_time.m = std::stoi(std::string(buffer+2,2));
            current_time.s = std::stoi(std::string(buffer+4,2));
            current_time.ms = std::stoi(std::string(buffer+6,3));
            if (current_time > previous_time)
            {
                  previous_time = current_time;
                  return true;
            }
            else return false;
      }

      timestamp previous_time;
};
```

What are the problems with virtual functions? Slow, 2 levels of redirections, possible instruction cache miss. How can we solve it? Typical way to convert runtime polymorphism into compile time polymorphism is CRTP, shown in next page.

```
template<typename HANDLER_IMPL> class handler_base
{
	void process(const std::byte* buffer, std::size_t length)
	{
		num_packets_read += HANDLER_IMPL::is_new_packet(buffer, length);
	}

	typename HANDLER_IMPL::header_type msg;		// for deep copying
	typename HANDLER_IMPL::header_type* msg_ptr;	// for reinterpret casting
	std::uint64_t num_packets_read = 0;
};

class handler_hk : public handler_base<handler_hk>
{
	typedef header_hk header_type;			// link between header and handler
	static bool is_new_packet(const std::byte* buffer, std::size_t length) {...}
};
class handler_jp : public handler_base<handler_jp>
{
	typedef header_jp header_type;			// link between header and handler
	static bool is_new_packet(const std::byte* buffer, std::size_t length) {...}
};
class handler_sg : public handler_base<handler_sg>
{
	typedef header_sg header_type;			// link between header and handler
	static bool is_new_packet(const std::byte* buffer, std::size_t length) {...}
};
```

Other questions for Hong Kong session

How to implement unordered map?

- array of lists of cells for separate chaining — pros : it will not be full, hence no rehashing
- array of cells for probing — pros : contigous in memory and hence cache friendly

If there are many threads, one of them is latency sensitive, what should we do for that thread?

- assign affinity, however scheduler stills assign the cpu to serve other threads
- assign priority, however this is not perfect
- isolate the cpu from scheduler

What are the components in a low latency trading system?

- market datafeed
- order gateway
- position monitoring
- trading strategies
- contingency plan, failover
- database storing all market data and order
- testing framework : unit test, regression test, stress test etc

How to implement throttle management for (1) continuous-second and (2) integral second?

```
struct throttle_checker_for_continuous_second // higher latency
{
	bool is_valid_order(const timestamp& t)
	{
		bool ans = (t - history.front() > one_second_interval);
		history.push(t);
		while (history.size() > THROTTLE) history.pop();
		return ans;
	}
	std::queue<timestamp> history;
};

struct throttle_checker_for_integral_second // lower latency
{
	bool is_valid_order(const timestamp& t)
	{
		if (!same_second(t, latest_t))
		{
			latest_t = t;
			count = 1;
			return true;
		}
		else if (count < THROTTLE)
		{
			++count;
			return true;
		}
		else return false;
	}
	timestamp latest_t;
	unsigned short count = 0;
};
```

Citadel quant questions from the web

Questions

1. What is 89^2?
2. Solve x^x^x^… = 2
3. How many digits are in 7^7?
4. What is the single digit for 2^230?
5. How many times a day does a clock's hands overlap?
6. When is gamma for an option the highest?
7. What is the expected number of dice rolls until we get two 6's in a row?
8. If ABC are integers between 1 and 10, how many different combinations of A, B, and C exist such that A<B<C?
9. You have a deck of 52 cards. What's the probability you draw exactly 1 heart in 2 draws with replacement?
10. What is the probability you draw 2 cards of the same color without replacement from a standard 52-card deck?
11. You can buy nuggets in packs of 5 or 11. What is the max num nuggets you can't buy using packs of 5 or 11?

Solution

1. We have : (90-1)^2 = 8100 – 180 + 1 = 7921

2. We have : x^2 = 2, hence x is 1.4141…

3. Answer is floor(log10(7^7)) + 1. Another method is brute force : 7, 49, 343, 2401 …

4. It forms a cycle 2 > 4 > 8 > 6 > 2 …, lets put them into `array = {2,4,8,6}`, then final answer is `array[(230-1)%4]`.

5. Answer is 23 excluding the starting overlap and the ending overlap.

6. Gamma is max ATM.

7. Build the binomial tree with p = 1/6, hence we have :

$$\begin{aligned} E[N] &= p^2 E[N \mid 6,6] + pqE[N \mid 6,\bar{6}] + qE[N \mid \bar{6}] \\ &= p^2 2 + pq(2 + E[N]) + q(1 + E[N]) \\ &= \frac{2p^2 + 2pq + q}{1 - pq - q} \\ &= \frac{2 + 2\times 5 + 5\times 6}{6\times 6 - 5 - 5\times 6} \\ &= \frac{42}{1} \end{aligned}$$

8. Answer is ${}_{10}C_3$.

9. Answer is 1/4 × 3/4 + 3/4 × 1/4 = 3/8.

10. Answer is (1/4 × 12/51) × 4 = 12/51.

11. This is called Chicken McNugget theorem (or Frobenius coin problem). Given two prime x and y, the greatest integer that cannot be denoted as $nx+my$ is $xy–x–y$, which is $55–5–11 = 39$ in this case. Another method is dynamic programming solution to coin problem. Suppose S, $S+1$, $S+2$, … $S+min(x,y)-1$ are consecutive sums which can be constructed by x and y, then all integers equal to or lying above $S+min(x,y)$ can be denoted as $nx+my$, as we can always consider coin-subproblem by deducting $min(x,y)$ from target sum. Consider multipartite graph in time domain, red indicates new numbers :

$$0 \to 5,11 \to 5,10,11,16,22 \to 5,10,11,15,16,21,22,27,33 \to \ldots \text{ propagate until we get 5 consecutive numbers}$$

The answer is the minimum of those 5 consecutive numbers minus 1.