High Performance Computing

Homework #3

Due: Monday September 25 2017 before 11:59 PM (Midnight)

Email-based help Cutoff: 5:00 PM on Sun, Sept 24 2017

Maximum Points For This Part: 25

Objective

The objective of this part of the homework is to:

- Review basics of Confidence Intervals (CI) from online presentation at: https://youtu.be/g0Xap1omt90
- Develop a micro-benchmark in C++ to compare the performance of using methods vs. lambdas in algorithms.
- Conduct detailed analysis of the micro-benchmark using profilers
- Develop comprehensive report based from the experiments

Background:

In C++ many standard algorithms such as: `std::sort`, `std::count_if` accept predicates to which you may pass methods or specify lambdas. Recollect that in computer science, a "predicate" is a function that takes zero or more parameters and returns a Boolean value (i.e., `true` or `false`). Predicates that accept one parameter are called unary predicates while predicates that accept two parameters are called binary predicates.

Scientific question:

Is there a performance difference between using a function versus supplying a lambda for a predicate for algorithms in C++?

Hypothesis:

Using a lambda would be faster as lambdas can be effectively inlined (see: https://en.wikipedia.org/wiki/Inline_function) with algorithms (thereby avoiding overhead of a function call) and the compiler can better optimize the resulting assembly code.

Instructions:

This homework requires you to design an experimental benchmark to test the aforementioned hypothesis and conduct experiments to accept/reject the hypothesis. Here are a few tips to help design your benchmark and complete this report:

1. Working with your `std::sort` algorithm would be effective (because of its time complexity arising from the number of times the predicate is used for comparisons). See slides off Canvas for example on using `std::sort` or refer to online documentation.
2. In general use smaller vectors (no more than 500,000 integers) to ensure it fits in cache.
3. Ensure your benchmark runs for about 10 seconds with optimizations (`-O3`) using a trial-and-error process to identify suitable settings.

4. Don't overcomplicate the benchmark. A few lines of code can be more useful than 100s of lines of code.

Name: Charles Mullenix

Apparatus Used (experimental platform)

The experiments documented in this report were conducted on the following platform:

Component	***Details***
CPU Model	26
CPU/Core Speed	1596.000
Main Memory (RAM) size	24,591,648 kB
Operating system used	Linux
Interconnect type & speed (if applicable)	n/a
Was machine dedicated to task (`yes/no`)	Yes (but with minor load from general purpose software and system processes)
Name and version of C++ compiler (if used)	4.9.2
Name and version of Java compiler (if used)	n/a
Name and version of other non-standard software tools & components (if used)	`callgrind` profiler and `kcachegrind` (GUI viewer)

Benchmark Design

My benchmark is going to utilize the std::sort algorithm to sort large vectors of integers in altering orders, essentially reversing the order of the array multiple times. The benchmark will take either "function" or "lambda" as a parameter, defining which technique will be used to pass the predicate to the algorithm.

The only looping that is done is in the method in which we initialize the vector in question, and again to run the tests a specified number of times. The vector initialization would have O(n) runtime, and thus will have n control hazards. That means it must be balanced out by an appropriate number of comparisons made by the sort algorithm (and by extension calls to our comparison code which will likely create its own control hazards). The sort algorithm will make approximately n*logn comparisons (taken from this stack overflow question). That means that one sort would likely be similar in proportion to the vector initialization. I think that that would still have too much extra noise polluting the benchmark, and our results would be inconclusive. Thus, I'll sort eight times per method call.

The other loop is to call the sorting function, and will have two control hazards for each time the test method is called and the sort runs eight times (the loop and the method call). The proportions should be fine.

Experiments and Observations

Briefly describe how the experiments were conducted (indicate commands used and copy-paste the PBS script(s) used to submit jobs and conduct the experiments)

I wrote a bash script to execute the job, and had to repeatedly run the script and test out different numbers of looping/method calling to get the right time proportion.
I ran the following commands to compile and execute the bash script:

```
g++ -g -Wall -std=c++14 -O3 homework3.cpp -o hw3
qsub hw3.bash
```

Here is the script:

```
#!/bin/bash

#PBS -N hw3_03_run
#PBS -l walltime=0:01:40
#PBS -l mem=128MB
#PBS -l nodes=1:ppn=1
#PBS -S /bin/bash
#PBS -j oe

# Change to directory from where PBS job was submitted
cd $PBS_O_WORKDIR

# Run the benchmark once in each configuration
echo "----------------[ Output from perf for function ]---------------------"
perf stat ./hw3 function 85
perf stat ./hw3 function 85
perf stat ./hw3 function 85
perf stat ./hw3 function 85
perf stat ./hw3 function 85
echo "----------------[ Output from perf for lambda ]---------------------"
perf stat ./hw3 lambda 85
perf stat ./hw3 lambda 85
perf stat ./hw3 lambda 85
perf stat ./hw3 lambda 85
perf stat ./hw3 lambda 85

#end of script
```

Then I ran:

```
cat hw3.bash > hw3_profile.bash
```

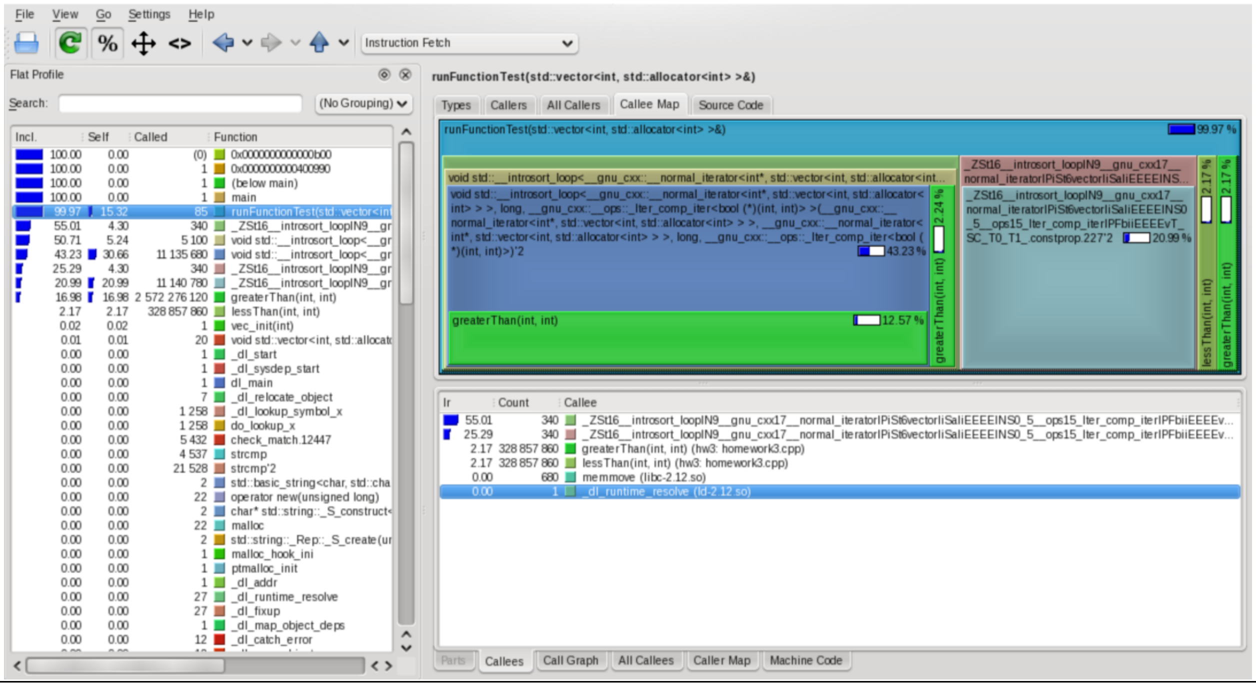
and modified the code to create a job to create the profiler scripts. That one is as follows:

```
#!/bin/bash

#PBS -N hw3_03_profiler
```

```
#PBS -l walltime=0:30:00
#PBS -l mem=128MB
#PBS -l nodes=1:ppn=1
#PBS -S /bin/bash
#PBS -j oe

# Change to directory from where PBS job was submitted
cd $PBS_O_WORKDIR

# Run the benchmark once in each configuration
echo "----------------[ Callgrind for function ]---------------------"
valgrind --tool=callgrind ./hw3 function 85
echo "----------------[ Callgrind for lambda ]---------------------"
valgrind --tool=callgrind ./hw3 lambda 85

#end of script
```

Verification:
A key aspect of the microbenchmark is that most of the runtime is spent in executing operations associated with the lambda. Include screenshots from suitable profiler to provide evidence that the two versions of the benchmark are correctly designed:

Screenshot of profiler at **-O3** optimization level using a **method**:



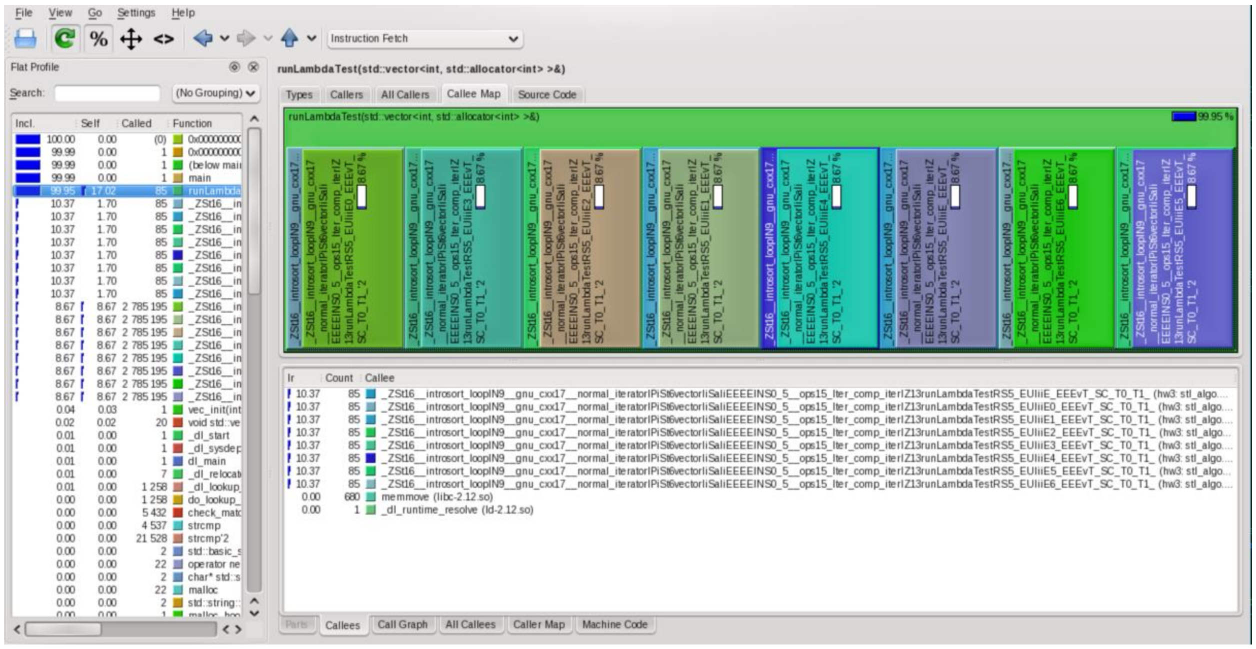
File
View
Go
Settings
Help
Instruction Fetch
Flat Profile
Search:
(No Grouping)
Types
Callers
All Callers
Callee Map
Source Code
runFunctionTest(std::vector<int, std::allocator<int> >&) 99.97 %
void std::__introsort_loop<__gnu_cxx::__normal_iterator<int*, std::vector<int, std::allocator<int...
void std::__introsort_loop<__gnu_cxx::__normal_iterator<int*, std::vector<int, std::allocator<int> > >, long, __gnu_cxx::__ops::_Iter_comp_iter<bool (*)(int, int)> >(__gnu_cxx::__normal_iterator<int*, std::vector<int, std::allocator<int> > >, __gnu_cxx::__normal_iterator<int*, std::vector<int, std::allocator<int> > >, long, __gnu_cxx::__ops::_Iter_comp_iter<bool (*)(int, int)>)'2 43.23 %
greaterThan(int, int) 12.57 %
greaterThan(int, int) 2.24 %
_ZSt16__introsort_loopIN9__gnu_cxx17__normal_iteratorIPiSt6vectorIiSaIiEEEEINS...
_ZSt16__introsort_loopIN9__gnu_cxx17__normal_iteratorIPiSt6vectorIiSaIiEEEEINS0_5__ops15_Iter_comp_iterIPFbiiEEEEvT_SC_T0_T1_.constprop.227'2 20.99 %
lessThan(int, int) 2.17 %
greaterThan(int, int) 2.17 %
Parts
Callees
Call Graph
All Callees
Caller Map
Machine Code

runFunctionTest(std::vector<int, std::allocator<int> >&)

Incl.	Self	Called	Function
100.00	0.00	(0)	0x0000000000000b00
100.00	0.00	1	0x0000000000400990
100.00	0.00	1	(below main)
100.00	0.00	1	main
99.97	15.32	85	runFunctionTest(std::vector<int
55.01	4.30	340	_ZSt16__introsort_loopIN9__gr
50.71	5.24	5 100	void std::__introsort_loop<__gr
43.23	30.66	11 135 680	void std::__introsort_loop<__gr
25.29	4.30	340	_ZSt16__introsort_loopIN9__gr
20.99	20.99	11 140 780	_ZSt16__introsort_loopIN9__gr
16.98	16.98	2 572 276 120	greaterThan(int, int)
2.17	2.17	328 857 860	lessThan(int, int)
0.02	0.02	1	vec_init(int)
0.01	0.01	20	void std::vector<int, std::allocato
0.00	0.00	1	_dl_start
0.00	0.00	1	_dl_sysdep_start
0.00	0.00	1	dl_main
0.00	0.00	7	_dl_relocate_object
0.00	0.00	1 258	_dl_lookup_symbol_x
0.00	0.00	1 258	do_lookup_x
0.00	0.00	5 432	check_match.12447
0.00	0.00	4 537	strcmp
0.00	0.00	21 528	strcmp'2
0.00	0.00	2	std::basic_string<char, std::cha
0.00	0.00	22	operator new(unsigned long)
0.00	0.00	2	char* std::string::_S_construct<
0.00	0.00	22	malloc
0.00	0.00	2	std::string::_Rep::_S_create(ur
0.00	0.00	1	malloc_hook_ini
0.00	0.00	1	ptmalloc_init
0.00	0.00	1	_dl_addr
0.00	0.00	27	_dl_runtime_resolve
0.00	0.00	27	_dl_fixup
0.00	0.00	1	_dl_map_object_deps
0.00	0.00	12	_dl_catch_error

Ir	Count	Callee
55.01	340	_ZSt16__introsort_loopIN9__gnu_cxx17__normal_iteratorIPiSt6vectorIiSaIiEEEEINS0_5__ops15_Iter_comp_iterIPFbiiEEEEv...
25.29	340	_ZSt16__introsort_loopIN9__gnu_cxx17__normal_iteratorIPiSt6vectorIiSaIiEEEEINS0_5__ops15_Iter_comp_iterIPFbiiEEEEv...
2.17	328 857 860	greaterThan(int, int) (hw3: homework3.cpp)
2.17	328 857 860	lessThan(int, int) (hw3: homework3.cpp)
0.00	680	memmove (libc-2.12.so)
0.00	1	_dl_runtime_resolve (ld-2.12.so)

Screenshot of profiler at **-O3** optimization level using a **lambda**:


File View Go Settings Help
Instruction Fetch
Flat Profile
Search:
(No Grouping)
runLambdaTest(std::vector<int, std::allocator<int> >&)
Types Callers All Callers Callee Map Source Code
runLambdaTest(std::vector<int, std::allocator<int> >&)
99.95 %
Parts Callees Call Graph All Callees Caller Map Machine Code

Incl.	Self	Called	Function
100.00	0.00	(0)	0x00000000
99.99	0.00	1	0x00000000
99.99	0.00	1	(below mai
99.99	0.00	1	main
99.95	17.02	85	runLambda
10.37	1.70	85	_ZSt16__in
10.37	1.70	85	_ZSt16__in
10.37	1.70	85	_ZSt16__in
10.37	1.70	85	_ZSt16__in
10.37	1.70	85	_ZSt16__in
10.37	1.70	85	_ZSt16__in
10.37	1.70	85	_ZSt16__in
10.37	1.70	85	_ZSt16__in
8.67	8.67	2 785 195	_ZSt16__in
8.67	8.67	2 785 195	_ZSt16__in
8.67	8.67	2 785 195	_ZSt16__in
8.67	8.67	2 785 195	_ZSt16__in
8.67	8.67	2 785 195	_ZSt16__in
8.67	8.67	2 785 195	_ZSt16__in
8.67	8.67	2 785 195	_ZSt16__in
8.67	8.67	2 785 195	_ZSt16__in
0.04	0.03	1	vec_init(int
0.02	0.02	20	void std::ve
0.01	0.00	1	_dl_start
0.01	0.00	1	_dl_sysdep
0.01	0.00	1	dl_main
0.01	0.00	7	_dl_relocat
0.01	0.00	1 258	_dl_lookup_
0.00	0.00	1 258	do_lookup_
0.00	0.00	5 432	check_matc
0.00	0.00	4 537	strcmp
0.00	0.00	21 528	strcmp'2
0.00	0.00	2	std::basic_s
0.00	0.00	22	operator ne
0.00	0.00	2	char* std::s
0.00	0.00	22	malloc
0.00	0.00	2	std::string::
0.00	0.00	1	malloc_hoo

Ir	Count	Callee
10.37	85	_ZSt16__introsort_loopIN9__gnu_cxx17__normal_iteratorIPiSt6vectorIiSaIiEEEEiNS0_5__ops15_Iter_comp_iterIZ13runLambdaTestRS5_EUliiE_EEEvT_SC_T0_T1_ (hw3: stl_algo....
10.37	85	_ZSt16__introsort_loopIN9__gnu_cxx17__normal_iteratorIPiSt6vectorIiSaIiEEEEiNS0_5__ops15_Iter_comp_iterIZ13runLambdaTestRS5_EUliiE0_EEEvT_SC_T0_T1_ (hw3: stl_algo....
10.37	85	_ZSt16__introsort_loopIN9__gnu_cxx17__normal_iteratorIPiSt6vectorIiSaIiEEEEiNS0_5__ops15_Iter_comp_iterIZ13runLambdaTestRS5_EUliiE1_EEEvT_SC_T0_T1_ (hw3: stl_algo....
10.37	85	_ZSt16__introsort_loopIN9__gnu_cxx17__normal_iteratorIPiSt6vectorIiSaIiEEEEiNS0_5__ops15_Iter_comp_iterIZ13runLambdaTestRS5_EUliiE2_EEEvT_SC_T0_T1_ (hw3: stl_algo....
10.37	85	_ZSt16__introsort_loopIN9__gnu_cxx17__normal_iteratorIPiSt6vectorIiSaIiEEEEiNS0_5__ops15_Iter_comp_iterIZ13runLambdaTestRS5_EUliiE3_EEEvT_SC_T0_T1_ (hw3: stl_algo....
10.37	85	_ZSt16__introsort_loopIN9__gnu_cxx17__normal_iteratorIPiSt6vectorIiSaIiEEEEiNS0_5__ops15_Iter_comp_iterIZ13runLambdaTestRS5_EUliiE4_EEEvT_SC_T0_T1_ (hw3: stl_algo....
10.37	85	_ZSt16__introsort_loopIN9__gnu_cxx17__normal_iteratorIPiSt6vectorIiSaIiEEEEiNS0_5__ops15_Iter_comp_iterIZ13runLambdaTestRS5_EUliiE5_EEEvT_SC_T0_T1_ (hw3: stl_algo....
10.37	85	_ZSt16__introsort_loopIN9__gnu_cxx17__normal_iteratorIPiSt6vectorIiSaIiEEEEiNS0_5__ops15_Iter_comp_iterIZ13runLambdaTestRS5_EUliiE6_EEEvT_SC_T0_T1_ (hw3: stl_algo....
0.00	680	memmove (libc-2.12.so)
0.00	1	_dl_runtime_resolve (ld-2.12.so)

Describe how the data in the above screenshot serve as evidence that your benchmark has been designed correctly:

If you look at the percentages in the Flat profile column, you can see that the majority of the time spent in either case was in the function calling the sorting algorithms (only about 0.02 - 0.03 percent was in the vec_init method).

Beyond that, in the function call profiler screenshot, you can see in the callees tab that over 80 percent of the program's execution time is spent in the sorting algorithm's callee's, with 12 percent of the time spent in the greaterThan method (it's possible that the lessThan method was handled differently by the compiler at the -O3 level). Either way, you can see in the callee's table beneath the graph that these two methods are called over 300 million times each. Considering that we're testing control hazards and their effect on the speed of the code, I think that this would provide substantial evidence of that.

In the same callee's table for the lambda function, you can see that there are eight different sorting methods, each called 85 times. This corresponds to the eight different times the sorting method is called and passed the lambda within each call to runLambdaTest(), a method which is called 85 times. Each of these sort calls take up about 10 percent of the time spent processing, which is exactly what we're looking for with this benchmark.

Observations:

Record the timing information collated from your experiments conducted using your micro-benchmark in the tables below. Note you must calculate 95% CI (Confidence Intervals) from 5 data points (see Appendix at end for 95% CI formulas). **Runtimes should be at least 10 seconds in all cases without requiring any changes to the benchmark**

Using inline function	**Elapsed Time (sec) measured using `/usr/bin/time` command**		
	Opt: `-O0`	**Opt: `-O2`**	**Opt: `-O3`**
Observation #1	94.805	15.632	10.969
Observation #2	94.643	15.323	10.601
Observation #3	94.662	15.295	10.609
Observation #4	94.662	15.322	10.596
Observation #5	94.637	15.499	10.584
Averages±95%CI	**94.682±0.077**	**15.414±0.162**	**10.672±0.185**

Using lambda	**Elapsed Time (sec) measured using `/usr/bin/time` command**		
	Opt: `-O0`	**Opt: `-O2`**	**Opt: `-O3`**
Observation #1	91.014	5.306	5.427
Observation #2	90.999	5.481	5.609
Observation #3	91.082	5.481	5.606
Observation #4	90.975	5.481	5.618
Observation #5	91.121	5.305	5.433
Averages±95%CI	**91.038±0.068**	**5.411±0.107**	**5.539±0.110**

Results and Conclusions

Using the above data develop a report discussing the performance aspects and conclude if you accept or reject the hypothesis. You may also mention data that you may not have recorded (as in peak memory usage) as needed. Indicate what suggestion you would provide to a programmer or software developer based on your experiment.

The above benchmark was designed to test the performance overhead of using a lambda over a traditional method call. The software accomplishes this by passing predicates constructed with each method as arguments to separate identical std::sort calls, which then runs the predicate repeatedly.

I predicted that the lambda would outperform the function consistently. The compiler's ability to inline lambdas avoids the control hazards produced when calling externally defined methods, and allows for a significant improvement in the runtime of the program. My prediction was validated by my observations, which displayed runtime improvements of 64.9% (2.85x) and 48.1% (1.93x) at the O2 and O3 optimization levels respectively.

At the O0 level (without any optimizations), we saw a percentage increase of only 3.8% (1.04x). Without in-lining, the lambdas created the same control hazards as the method calls and their performance became similar.

The percentage CPU for use of the function tests didn't fall below 99% (mean of 99.6), and 91% (mean of 97) for the lambdas. The confidence interval ranges did not exceed 2% of the associated mean. The tight confidence interval ranges and the stability of the CPU usage implies that the benchmark performed consistently, and ran largely unaffected by outside influence. The memory used never exceeded the amount defined in the bash script, which was 128MB.

Clearly in this micro-benchmark, the optimized lambda functions had a strong advantage over their externally defined counterparts.

Conclusion:
With the right optimization levels, lambdas can significantly and consistently outperform externally defined functions by avoiding the control hazards produced by the latter. Lambdas are important to clean code, as they do not force the reader to look for the method definition. They can also be assigned to variables and easily re-used without adding to the API of the program. They are the preferred method of passing a block of code to another method.

Appendix

Copy-paste your style-checked benchmark program source code into the space below:

```
// Copyright 2017 Charles Mullenix

/*
 * File:   homework3.cpp
 * Author: mulle
 *
 * Created on September 23, 2017, 6:03 PM
 */

#include <cstdlib>
#include <string>
#include <vector>
#include <algorithm>

std::vector<int> vec_init(const int VecSize);
```

```
bool lessThan(const int a, const int b);
bool greaterThan(const int a, const int b);
void runLambdaTest(std::vector<int>& vec);
void runFunctionTest(std::vector<int>& vec);

/*
 *
 */
int main(int argc, char** argv) {
    const int numRuns = (argc > 2 ? std::stoi(argv[2]) : 30);
    const int VecSize = 500000;
    std::string cmd = "function";
    std::vector<int> vec = vec_init(VecSize);
    if (argc > 1 && argv[1] == cmd) {
        for (int i = 0; i < numRuns; i++)
            runFunctionTest(vec);
    } else {
        for (int i = 0; i < numRuns; i++)
            runLambdaTest(vec);
    }
    return 0;
}

void runLambdaTest(std::vector<int>& vec) {
    std::sort(vec.begin(), vec.end(), [](int a, int b) { return a > b; });
    std::sort(vec.begin(), vec.end(), [](int a, int b) { return a < b; });
    std::sort(vec.begin(), vec.end(), [](int a, int b) { return a > b; });
    std::sort(vec.begin(), vec.end(), [](int a, int b) { return a < b; });
    std::sort(vec.begin(), vec.end(), [](int a, int b) { return a > b; });
    std::sort(vec.begin(), vec.end(), [](int a, int b) { return a < b; });
    std::sort(vec.begin(), vec.end(), [](int a, int b) { return a > b; });
    std::sort(vec.begin(), vec.end(), [](int a, int b) { return a < b; });
}

void runFunctionTest(std::vector<int>& vec) {
    std::sort(vec.begin(), vec.end(), greaterThan);
    std::sort(vec.begin(), vec.end(), lessThan);
    std::sort(vec.begin(), vec.end(), greaterThan);
    std::sort(vec.begin(), vec.end(), lessThan);
    std::sort(vec.begin(), vec.end(), greaterThan);
    std::sort(vec.begin(), vec.end(), lessThan);
    std::sort(vec.begin(), vec.end(), greaterThan);
    std::sort(vec.begin(), vec.end(), lessThan);
}
```

```
std::vector<int> vec_init(const int VecSize) {
   std::vector<int> v;
   for (int i = 0; i < VecSize; i++)
      v.push_back(i);
   return v;
}

bool lessThan(const int a, const int b) {
   return a < b;
}

bool greaterThan(const int a, const int b) {
   return a > b;
}
```

Formula for 95% CI

Ensure you review presentation at: https://youtu.be/g0Xap1omt90. Assume the five timings you have recorded in Task #1 are t_1, t_2, t_3, t_4, and t_5. First calculate the mean (μ) and standard deviation (σ) using the formulas:

$$\mu = \sum_{1}^{n} t_i/n \quad \textbf{and} \quad \sigma = \sqrt{\left(\sum_{1}^{n}(t_i - \mu)^2\right)/n}$$

Where n =5 (in this specific exercise since we are using only 5 timing values). Now, from the student t-distribution tables, the 95% CI is computed using the formula:

$$95\%\ \text{CI} = (2.776 * \sigma)/\sqrt{n}$$

Note that the number `2.776` (aka the z-value) varies depending on the value of n.

Submission

Once you have completed your report upload the following onto Canvas:

1. This report document (duly filled) and saved as a PDF file with the naming convention `Homework3.pdf`.
2. The benchmark program that you have developed named with the convention `Homework3.cpp`.
3. The PBS scripts you developed to submit jobs on Red Hawk.