ACM40660 Assignment 2

Adam Ralph

Deadline: 3rd November 2019

3. Create a program, numericalIntegration.c/f90, that calculates the area under the curve $f(x) = sin(x)$ from $0 \to \pi$ using the Trapezoidal and Simpson's Rule and Gaussian Integration. Compare with the actual result.

   Trapezoidal rule

$$\int_a^b f(x)dx \sim \frac{b-a}{2N}(f(x_0) + 2f(x_1) + 2f(x_2) + \cdots + 2f(x_{N-1}) + f(x_N))$$

   Where $x_0 = a$ and $x_N = b$ with $N-1$ equidistance points in between.

   Simpson's rule
   If the function is sampled at n equal intervals (n must be even) $\{f_o = f(a), f_1 = f(a+h), f_2 = f(a+2h), \cdots, f_n = f(b)\}$ then,

$$\int_a^b f(x)dx \sim \frac{h}{3}[f(a) + 4(f_1 + f_3 + \cdots) + 2(f_2 + f_4 + \cdots) + f(b)]$$

   Gauss Quadrature

   Using the two point Guass quadrature rule, the integral of sin(x) can be approximated by

$$\int_a^b f(x)dx \sim h\left[f\left(\left(k - \frac{h}{\sqrt{3}}\right)\right) + f\left(\left(k + \frac{h}{\sqrt{3}}\right)\right)\right],$$

   where $h = \frac{b-a}{2}$ and $k = a + h$

   (a) Make each integration method a separate function each in a separate file.

   (b) Create a Makefile that compiles and links the numericalIntegration program. Also add a 'clean' target that removes all object files.

   (c) Allow the user to pick the integration method and the number of intervals from the command line (except Gauss Quadrature).

   (d) Print the calculated value and the actual value to the screen.

   (e) Tabulate the results of all methods for the following interval values: $n = \{2, 8, 16, 64\}$.

   (f) Comment on the results obtained.

4. In this program we are going to test the performance of two searching codes.

   (a) Write a *randomarray(n, max)* function that returns a pointer to an array of size n integers, filled with random values between 0 and *max*.

   (b) Write a *median(n, arr)* function that returns the median of an integer array, without ordering them. The median value means that half the numbers are smaller or equal and half are bigger or equal. If the length n is even, choose the $n/2$ highest value. In this implementation of the median, the return value must be a number within the sequence.

      i. Starting from the first element determine if it is the median.

      ii. If not advance to the next element and check, repeat the process until the median is found.

iii. It is possible to skip elements to make it more efficient.

(c) Write a *mediansort(n, *cycle, arr, sorted)* recursive function (or subroutine) that sorts the size *n* array of integers *arr* in ascending order and saved in *sorted*. Note that *cycle* is defined as a pointer.

i. Find the median of the array and place that value in the middle.

ii. Find the median of the remaining values. This value will be to the left or right of the previous median.

iii. Repeat until the array has been sorted, see below.

1st pass:	cycle=0
(2, 8, 1, 5, 7)	5 is the median
(-, -, 5, -, -)	partially sorted array
2nd pass:	cycle=1
(2, 8, 1, 7)	2 is the median
(-, 2, 5, -, -)	
3rd pass:	cycle=2
(8, 1, 7)	7 is the median
(-, 2, 5, 7, -)	
4th pass:	
(1, 8)	1 is the median
(1, 2, 5, 7, -)	
final pass:	
(8)	8 is the median
(1, 2, 5, 7, 8)	completely sorted array

Notice that when $cycle = 0$ the median is placed in the middle. When $cycle = 1, 2$ the median of the remaining numbers is placed ± 1 away from the middle, *etc*. Having many elements with the same value as the median can lead to problems. Make sure they are distributed either side of the middle. It might be easier to construct an iterative function first and then convert it to a recursive one.

(d) Write a main program that asks the user for an array size and a maximum value. The program should then generate the array using *randomarray()* , print it to the screen, then sort it using *mediansort()* and print it again. Two runs with the exact same inputs should produce the same results.

(e) Write a *search(i, n, arr)* function that returns the smallest index of the size *n* array of integers *arr* containing the value *i* or -1 if it's not present. In order to do it, the function should assume that the array is sorted in ascending order. It will scan the list from the beginning until it finds the correct element or finds an element superior to *i* (the array being sorted, that means the element we're looking for is not present), or reaches the end of the array.

(f) Write a *chopsearch(i, n, arr, amin, amax)* recursive function that returns the smallest index of a subset of an array *arr* of size *n*, containing the value *i* or -1 if it's not present. That is, at least one of the elements $arr[amin] \cdots arr[amax] = i$. Thus if called with *amin=0* and *amax=n-1* (*amin=1* and *amax=n* for fortran) it should produce the same result as the function *search()*.

Binary chop search is like searching a word in the dictionary. You open it in the middle, and depending on whether you're before or after the word you're looking for, you repeat the search on the corresponding half of the dictionary, and so on, until you found the correct page.

In order to do this on an array of numbers, this function should assume that the array is sorted in ascending order. This function is slightly different to the binary sort in that it will take a **random index** x between *amax* and *amin* and compare it to i:

- if $i < arr[x]$, then the function will call itself on the 'left' half of the array, bounded by *min* and *x-1* (or return -1 if that half is empty),
- if $i > arr[x]$, then the function will call itself on the 'right' half of the array, bounded by *x+1* and *max* (or return -1 if that half is empty),
- if $i == arr[x]$, the function should iterate over elements left most of x until it finds the lowest array index containing the value i (the same value can be present multiple times in the array, and we want to return the lowest index at which it can be found).
- Repeat the process until i has been found or $amax == amin$.

(g) Write a *benchmark_naive(n, max, s, mult)* function that will do the following:

- start a timer,
- loop for *mult* times,
  - for every 1000 cycles
    - generate an array through *randomarray()* using n and *max* as parameters
    - (in this case the arrays should be different each time),
    - and sort the array through *mediansort()*,
  - search multiple times for the value s in the array using *search(s, n, arr)*,
- stop the timer,
- deallocate the array,
- calculate and return the elapsed time in any unit of your choice (seconds, milliseconds, clock cycles, etc.). The unit choice doesn't really matter as we're only going to use it for comparisons, so feel free to use whichever is convenient.

(h) Write a *benchmark_chop(n, max, s, mult)* function that will do the following:

- do the same as above, except use *chopsearch(i, n, arr, 0, n-1)*.

(i) Use your two benchmark functions with the following sets of parameters:

- (2000, 10000, 10, 100000)
- (2000, 10000, 5000, 100000)
- (2000, 10000, 9000, 100000)

Provide your results as part of the assignment, either as comments in the code or as output. Comment on the relative performances is this expected or not?). Evaluate the time complexity of the two techniques (sorting and searching), give the best, worst and average case scenarios.

5. Some general points:

(a) the main point is to get the program to do what the question asks (use either FORTRAN or C **not** C++),

(b) but make sure your code conforms to good programming practise and the C standard,

(c) you can use -Wall as a compile option,

(d) nested functions are not standard in C,

(e) comment your code and place any observations as comments,

(f) make sure the work is your own and upload the completed code to BrightSpace.