EC 504 – Fall 2023 – Homework 2

Due Friday, Sept 29, 11:59PM, 2023. Submit the results in the directory `/projectnb/ec504rb/students/yourname/HW2` **on your SCC account.** `HW2` **should only include the source code, the makefile, the standard input and output files describe below. It must compile and run successfully using the makefile.**

NOTE: In addition to the required coding part in Section 1 below, the is an optional open ended part in Section 2 to start to analize peformance and present results in graphical form. Help will be given class but this can also be done collaboratively in small group with others students, who might be part of your project team.

1 Searching a Sorted List

Data basis are built so that more efficient algorithms can be applied to retrieve information. Searching a sorted or well ordered list is an example. The sort may require more time but it can be amortized if many subsequent data searches are needed.

The code you need for this exercise is in GitHUB in the file HW1code. It will compile using by typing `make -k`. To run it just type `\find Sorted100K` for example. I will spill data to the terminal and make out file called `Sorted100K.txt_out` But this is wrong! You have to add new functions. The code is set up to select `NoOfKeys = 100` keys to give you a sense of average efficiency. You can test it with a smaller number but set it back to 100 for you solution. I also added a small into put file `Sorted16.txt` as easy way to test your solution.

1. Searching a sorted array is a classic divide and conquer algorithms. Given a value called the `key` you search for a match in an array `int a[N]` of N objects by searching sub-arrays iteratively. Starting with `left = 0` and `right = N-1` the array is divides at the middle `m = (right + left)/2`. The routine, `int findBisection(int key, int *a, int N)` returns either the index position of a match or failure, by returning `m = -1`. First write a function for bisection search. The worst case is `O(log N)` of course.

2. If the data has some structure you may beat the bisection method. As an example next write a second function, `int findDictionary(int key, int *a, int N)` to find the `key` faster, using what is called, **Dictionary search**. This is based on the assumption of an almost uniform distribution of number of in the range of `min = a[0]` and `max = a[N-1]`. Dictionary search makes a better educated guess for the search for the value of `key` in the interval between `a[left]` and `a[right]` using the fraction change of the value, $0 \leq x \leq 1$:

   ```
   x = double(key - a[left])/(double(a[right]) - a[left]);
   ```

to estimate the new index,

```
m =  int(left + x * (right - left)); // bisection uses x = 1/2
```

Write the function `int findDictionary(int key, int *a, int N)` for this. For a uniform sequence of numbers this is with **average** performance: `(log(log( N))`, which is much faster than `log(N)` bisection algorithm.

3. Implement your algorithm as a C/C++ functions. On the class GitHub there is the main file that reads input and writes output the result. You only write the required functions. Do not make any changes to the `infield` reading format. Place your final code in your directory HW2. There are 3 input files
`Sorted100.txt , Sorted100K.txt, Sorted1M.txt` for $N = 10^2, 10^5, 10^6$ respectively. Notice how much faster dictionary is than bisection. This is because the input sorted list were generated as uniform random set of integers.

`(1)The code should run for each file on the command line.`

2 Optional Analysis of Performance Scaling

To understand performance, it is usually much better to plot and fit performance as function of size. Instructions will be given in future lectures and exercises. But if you wish you can try it on your own to get a head start. Already the program pick `NoOfKeys = 100` keys so you can get better estimate by taking the average. The you can graph the performance as function of N to see this scaling behavior much more fun and illuminating than a bunch of numbers! To extend the range of sizes there is a code `makeSortedList.cpp` on in the CodeBucket on GitHUB that can generate more sorted input lists any size N. For example you might choose at least 6 sizes: $N = 10^2, 10^4, 10^5, 10^6, 10^7, 10^8$. Then collect on Time and OpCount and graph them as function of N. Any hack is ok to get some rough graphs. You will find there is an art to measuring these slow growth rates! If you want to automate the size you can integrate `makeSortedList.cpp` as subroutine into your code.

A simple and powerful approach is to use the basic Unix tool: verb! gunplot! On the EC504 GitHUB in **GeneralComputerInforation/Plotting_and_Fitting** there is information on how to use gnuplot and even in **LittleGnuplot.pdf** instructions on how to install gnuplot in your laptop. Gnuplot can be done on the SCC through the SCC OnDemand interface. Other graphing and fitting options are fine. I am currently experimenting with simple ways to import C arrays into a `Jupyter notebook` and/or `Mathematica` to suggest more powerful methods. Everyone can share ideas on Slack.