BIRLA INSTITUTE OF TECHNOLOGY & SCIENCE, PILANI
First Semester 2010-2011
CS / IS C363 Data Structures and Algorithms
[Section 4 – Thursday, 19th August]
Lab Sheet – 2 [Duration: 165 minutes]

Problem

Brief:

- Implement a procedure that finds the k^{th} smallest element: start by finding the minimum element, say M_1, and then find the smallest element > M_i iteratively, for each i = 2 to k.
- Implement the Partition procedure (as in quicksort) using a pivot selected uniformly randomly. Implement QuickSelect (i.e. select the kth smallest value given k as a parameter) using Partition.
- Compare the performance of these two procedures for different values of k (ranging from 1 to N/2 - in particular try values 1,2, 5, 10, floor(logN), floor(sqrt(N)), N/20, N/5, N/2.)

Objective:
In this lab, comparatively analyze the performance of two methods that find the k^{th} smallest element in a given list. For simplicity, we assume that all the elements in the list are integers, as it will be trivial to extend the algorithm to other types.

Method 1:

For finding the k^{th} smallest element in a list, we can repeatedly find the smallest element k times, while ignoring the elements already found as smallest so far. The design will be as follows: Given a (random access) list L of size n, for iteration j, assume L[0] … L[j-2] is sorted, find the minimum element in L[j-1] … L[n-1] and swap it with L[j-1]. At the end of the routine, k^{th} smallest element should be in L[k-1].

Method 2:

For finding the k^{th} smallest element we can also use procedure QuickSelect that uses Partition:

1. We first partition the given list into two sub lists with the help of some pivot, p. All elements less than p will be in left sub list and all the elements greater than p will be in the second sub list.
2. If p is at the k^{th} location in the list, then p is the k^{th} smallest element.
    - Else, if number of elements in the left sub list is greater than or equal to k, then the k^{th} smallest element is in the left sub list.
    - Else, if number of elements in the left sub list is less than k, then it implies that the k^{th} smallest element is in the right sub list.
3. Repeat steps 1 and 2 till we find the k^{th} smallest element.

Partition Procedure: This procedure must choose a pivot, p, uniformly randomly and use it for partitioning. (The Partition procedure can be referred from text book, or lecture slides.)

Description:

The program must take as input a file containing integers. The number of elements in the list of initegers is determined by the number of elements stored in the file. The program then invokes two methods for computing the k^{th} smallest element in the list using the same set of input parameters. It determines the time taken by each of the method for the same value of k. It then stores the time values

along with the value of k used onto a file for plotting graph. Note that the size of the list of integers must be sufficiently large enough for getting any meaningful time values for comparison.

Step 1:
Create new type definitions: (**`List.h`**)

(a) Create a type definition for the type **`Element`**, which is represented as an integer.
(b) Create a type definition for **`List`** of **`Element`**s, which is a dynamic array.
(c) Create a type definition for the type **`Location`**, which is used to maintain the location of an element.

Step 2:
Create header file for operations: (**`ListOps.h`**)

(d) **`Element findkthsmallest_m1(List list, int list_ size, Element k)`**: This function will find the k^{th} smallest element using method 1 described above and returns it
(e) **`Location doPartition(List list, Location start, Location end):`** This function will uniformly randomly select a pivot from within the **`List`** list[start...end], then partition it into two sub lists as described above. This function returns the location in which pivot element is finally stored.
(f) **`Element quickSelect(List list, int list_ size, Element k):`** This function will find the k^{th} smallest element using method 2 described above and return it.

Step 3:
Create two implementation files; method1.c and method2.c.
In method1.c, implement the function described in (d) above.
In method2.c, implement the fu**n**ction described in (e) and (f) above.

If you wish to implement recursive versions, then you must create separate helper functions in which the recursive versions are implemented; but you must not change the names/prototypes of the functions described above.

Step 4: driver file.
Create a driver file (**`evaluate.c`**) for meeting the objectives of this problem. The problem description above outlines the actions to be performed by the driver file. The names of the input file and output file must be read as command-line arguments. The first parameter will be the input file and the second parameter is the output file. If second parameter is not specified, then the output needs to be printed in standard output. The input file must have only one integer in each line. The output file must be a comma separated file(.csv) containing the fields value of k, time needed by method 1 and time needed by method 2. The driver performs the following steps:

1. The program must take as input, a file containing integers stored in it; with one integer in each line. (Input file must be created by you.)
2. The program must read the integers from this input file, and store the integers in a dynamically created array.
3. for the values of k, ranging from 1 to N/2. (At least the values 1,2, 5, 10, floor(log(N)), floor(sqrt(N)), N/20, N/5,and N/2, must be used.), repeat the steps 4 through 9
4. note current time.(t_0)
5. Call the method1 for computing the kth smallest element.

6. note the current time (t_1). The time for computing step 5 is $t_1 - t_0$.
7. Call the method2 for computing the kth smallest element.
8. note the current time (t_2). The time for computing step 7 is $t_2 - t_1$.
9. Store the time taken for both the methods and the value of k used in an output file.
10. Plot a graph with two curves one for each method with time taken on the y-axis and k on the x-axis.

You may use rand() function to generate random values, and store it in file. Please use a separate c file for this code. (You can also use the script provided in the lab2 direcotory in blade3. Run the script without arguments for usage information.) Refer man rand for details. Appendix 1 describes one of the methods for computing time consumed by a function.

Step 5: Performance evaluation.
Use the output file generated to plot a graph for comparatively evaluating the two methods. You may use xgraph, open office spread sheet, google charts, or any other method. Please name the file containing the plot as **`plot`**, with appropriate extension.

Support files: List.h, ListOps.h, input file(say, input.txt)

Deliverables: method1.c, method2.c, evaluate.c, make file, output file (say, output.txt), plot file.

Appendix 1:

Measuring Execution Time:

```
   #include <sys/time.h>

   int main(int argc,char **argv)
   {           ......
// Declaration of timeval variables which will capture the system time
   long time_start, time_end;
   struct timeval tv;

//  Write all the initialization code here

 //  Time measurement begins here
   gettimeofday(&tv,NULL); /*gets the current system time into variable tv */
   time_start  = (tv.tv_sec *1e+6) + tv.tv_usec;
// calculating start time in seconds

//  Write the code to be analyzed here

   gettimeofday (&tv, NULL);
   time_end = (tv.tv_sec *1e+6) + tv.tv_usec;
 // calculating end time in seconds
// (time_end-time_start) gives the total execution time
   return (0);
}
```