Birla Institute of Technology & Science, Pilani
2nd Semester 2016-17 - CS F211 – Data Structures and Algorithms

---

Lab 5 – 14th Jan. 2017
Topics – Sorting and Practical Performance of Sorting Algorithms

1. Measuring the size of the (call) stack

(Call) Stack size in a C program can be measured by a simple hack:

Declare a local variable, say bot, in function *main* in your program. At any point in time, consider a local variable, v, in the current function: abs(&v – &bot) i.e. the absolute difference between address of v (which is on the top of stack frame) and that of bot (which is on the bottom of stack frame) gives the approximate size of the stack (at this point in program execution).

Usually, stack size growth is interesting with recursive programs/procedures, and with multiple recursive calls in a recursive procedure, stack will grow and shrink. You can compute the current stack size using the hack mentioned and update a global variable maxStkSize (if current size exceeds the maximum).
[Note: While implementing this, clearly think and understand what you are doing, including these specific aspects: (i) What is the type of &v for any variable v? (ii) Is &v a byte address or a word address? (iii) Why is it necessary to compute absolute difference? (iv) What is the type of function abs in *math* library of C? End of Note.]

2. Recursive vs. Iterative Implementation of QuickSort

Exercise 1:

The Quicksort procedure derived from a divide-and-conquer design is simple (*Version 1 – qs*):

```
qs(ls,lo,hi) {  if (lo<hi) { p=part(Ls,lo,hi,pivot(Ls,lo,hi)); qs(ls,lo,p-1); qs(ls,p+1,hi); } }

// Assumptions:
// (i) pivot returns the position (i.e. index) of the pivot element;
//  (ii) Postcondition for part: Ls[p] is the pivot; (Ls[j]<=Ls[p] for lo<=j<p) and (Ls[j]>Ls[p] for p<j<=hi)
```

Implement a simple quick sort algorithm on an array of student records, where each record must contain the following:

Student:
- Name : single word (at most 20 characters)
- Marks : unsigned integer

You must sort the records based on marks of the students. Use the last element as the pivot. Also implement a partial quick sort algorithm as per the instructions given in the following table.

Key	Function	Input Format	Description
0	readData	0 M	Indicates that next M lines will contain data to be read. Each line will contain student's name followed by marks, separated by space.
1	partialQuickSort	1 L	Call quicksort but stop partitioning when size of larger list

			falls below L, and print the whole partially-sorted array. Print each name and marks (space separated) in a new line.
2	quicksort	2 1	Call complete quick sort and print final result. The second argument (1) indicates that this is simple version of quick sort. If it is 2, it has to call version 2 of quick sort explained in Exercise 2; If it is 3, then it has to call version 3 of quick sort explained in Exercise 3; and version 4 if it is 4.

Note:

You should call partialQuickSort and quickSort on separate copies of input list. A sample input list is given below.

Sample Input File:
0 12
MAR 9937
MAY 30344
NOV 31441
AUG 17078
APR 24489
JAN 22172
DEC 23239
JUL 23072
FEB 9718
JUN 29687
OCT 31011
SEP 29969
1 5
2 1
-1

Exercise 2:

Controlling depth of recursion (*Version 2 – qs2*):

Eliminate redundant calls by checking whether the list size is non-trivial (i.e. at least 2) before each call. Always expand smaller of the two lists first i.e. decide which of the two recursive calls to make based on the size of the list. [Note: Once partitioned, qs can be applied on sub-lists independently! End of Note.] Name this procedure qs2.

Measure the time taken and (maximum) stack size used by qs as well as by qs2. Repeat this for different input sizes and compare the two sets of values.

Exercise 3:

Eliminating Tail Recursion (*Version 3 – qs3*):

(i) Eliminate the tail recursive call from qs2 manually. Name this procedure qs3.
(ii) Eliminate the tail recursive call from qs2 using gcc –foptimize-sibling-calls. [Verify this with man pages before proceeding.].

Measure the time taken and (maximum) stack size used by qs3 as well as the compiler-optimized version of qs2. Repeat this for different input sizes and compare the two sets of values. If qs3 performs much worse than the compiler-optimized version compare the assembly code so as to tune your code. [Assembly code can be obtained by compiling with –S option of gcc which will generate a .s file.]

Exercise 4:

Eliminating Recursion using explicit Stack (*Version 4 – qs4*):

Eliminate the remaining recursive call from qs3 using an explicit stack. If this affects the order in which sublists are sorted, and thereby impacting your order control code in (a) fix it. Let this be named qs4.

Measure the time taken and (maximum) stack size used by qs4. In this case stack size is the size of the explicit stack your have introduced which may be kept track while pushing / popping elements. as well as the compiler-optimized version of qs2. Repeat this for different input sizes and compare the values with that of qs3.