COMP26120
Academic Session: 2018-19

Lab Exercise 6: Recursive Sorting and Searching

Duration: 1 lab session

For this lab exercise you should do all your work in your COMP26120/ex6 directory.

Learning Objectives

At the end of this lab you should be able to:

- Implement the quicksort and mergesort recursive sorting algorithms and explain their operation
- Define what it means for a sorting algorithm to be stable and identify stable sorting algorithms
- Analyse the complexity of recursive algorithms
- Design experiments to help understand the practical performance of algorithmic solutions

Introduction

Stability

A sorting algorithm is said to be stable if any two records with equivalent keys retain their relative order in the sorted output. You met this concept previously in Lab 4 when inserting people into the linked list in sorted order. There if two people had the same age or name then the person who was inserted first should have appeared first.

Merge Sort

Merge sort is an efficient, divide and conquer sorting algorithm based on the merge operation.

The idea of merge sort is given as follows:

1. Recursively dividing the input list into equal-sized sublists until the sublists are trivially sorted;
2. Merge the sublists while returning up the call chain.

Figure 1 gives pseudocode for merge sort.

```
// One could implement this recursively but then the length of lists
// that could be merged is limited by the size of the call stack
merge(L1, L2){
    if one of the lists is empty return the other list
    Let Merged be big enough to contain both lists
    l1_index, l2_index, m_index = 0
    while each of the indexes is still valid
      if L1[l1_index] > L2[l2_index] then
          Merged[m_index++] = L1[l1_index++]
      else
          Merged[m_index++] = L2[l2_index++]
    if one of the lists still contains elements copy these into Merged
    return Merged
}

mergeSort(L){
    if |L| ≤ 1
      return L
    Split L into two roughly equal halves L = L_l + L_r
    return merge(mergeSort(L_l), mergeSort(L_r))
}
```

Figure 1: Algorithm for Merge Sort

Quick Sort

Quick sort is an efficient, divide and conquer sorting algorithm based on partitioning.

The idea of quick sort is given as follows:

1. Pick one element called "pivot" from the list.
2. Reorder the list so that all elements smaller than pivot come before the pivot; while those with values greater than pivot come after it. After this partitioning, the pivot is in its final position.
3. Recursively apply step 1-2 to the two sublists divided by pivot.

Figure 2 gives pseudocode for quick sort. Note that this pseudocode is not 'in-place' i.e. it assumes that we create new lists rather than editing the existing lists. What are the memory requirements of doing it like this? How can it be made in-place?

Note that the choice of pivot can be important for deciding the complexity of the algorithm – notice what happens if we make the 'worst' choice of pivot on each step.

Criteria for choosing a sorting algorithm

If you are dazzled by all the sorting algorithms and wonder which one to choose when it comes to real life problems, consider the following properties:

- Time complexity - It is usually $O(n^2)$ for most simple algorithms. If there are huge data sets, you might want to choose a more sophisticated one that has $O(n \log n)$ time complexity on average cases.

```
function quicksort(L){
    if length of L ≤ 1
      return L
    remove an element x from L to use as a pivot
    L_≤ := elements of L less than or equal to x
    L_> := elements of L greater than x
    L_l := quicksort(L_≤)
    L_r := quicksort(L_>)
    return L_l + [x] + L_r
}
```

Figure 2: Algorithm for Quick Sort

- Space complexity - How much auxiliary memory is allowed here?
- Stability - Do we need to preserve the relative order of equivalent keys in output?
- Complexity of algorithm itself - For example, insertion sort, bubble sort are elementary algorithms; while quick sort, merge sort, bucket sort and radix sort are sophisticated ones.

For more information, you might find this helpful.

There is always a trade-off in between evaluation criteria. It is hard to find an algorithm with low time complexity requiring no additional memory at the same time. Or maybe you could be the first person who invents a perfect solution?

Binary Search

You should have met this general idea before. If a list is sorted and you pick a random element of the list then you can always tell if another element should appear before or after that element. This gives us a way of quickly searching the list by recursively halving the list. You should be used to this as a typical example where we get logarithmic complexity – do you remember why?

Amortization

I'm sure the following scenario is familiar to you. You can manually edit some data or write a script to do it for you. The script will take 5-10 minutes to write whilst manually editing will take under 5 minutes. So you manually edit the data. But then you have to repeat this 4 or 5 times; it would have been quicker to write the script in the first place.

This general idea is known as amortization. Fans of xkcd may already be familiar with this concept and you will certainly meet it again when looking at advanced data structures.

Once learning about these ideas (binary search and amortization) it is tempting to apply it everywhere. But sometimes it is not worth the cost. If we have a large *unordered* list and we need to find a single item in it then the cost of first sorting the list and then using binary search will definitely be more expensive than a simple linear search of the list.

Part 1 - Recursive Sort

For part 1 of this exercise you will extend what you did in the previous lab to implement two sorting algorithms - **Merge Sort** and **Quick Sort** - and extend the analysis that you did.

Write C programs for each sorting algorithm separately. Note that if you use code from outside sources for an algorithm, you can only get partial credit for its part. See marking scheme for details.

The input and output format is the same as that in the previous lab.

Extend your *times.txt* and *complexities.txt* tables (you should start with those that you submitted last time and extend them) but add an additional column to *complexities.txt* indicating whether the sorting algorithm is *stable* or not.

Your sorting algorithms should be called *merge.c* and *quick.c*. You should also include your *insertion.c* solution for comparison purposes.

Part 2 - Recursive Search

Note that the majority of the marks for this lab are for part 1. Therefore, if you are running out of time make sure you get part 1 done before moving on to part 2.

We now look at a scenario where we have a large list of strings and a large (but usually not as large) number of queries and the task is to check whether each of the query strings appears in the list.

Input and Output

The first line will contain two integers n ($1 \leq n \leq 10^6$) and m ($1 \leq m \leq 10^4$) where n is the number of strings and m is the number of queries. The second line will contain n space-separated quoted strings. The third line will contain m space-separated quoted queries. No strings will be longer than 25 characters.

The output should be a single line of m space-separated query answers of either *yes* or *no* indicating whether the query string is in the list or not.

Sample Input 1	Sample Output 1
5 2 "Bandit" "Critic" "Bubble" "Lackluster" "Dwindle" "Bubble" "Green-Eyed"	yes no
Sample Input 1	Sample Output 1
3 4 "Shape of You" "Castle on the Hill" "Galway Girl" "Love Yourself" "Everything has Changed" "Little Things" "Shape of You"	no no no yes

Suggested Solution

You can decide to solve the problem in any way you want. However, a simple linear search is unlikely to work as the tests will almost certainly time out. Our suggestion is to first sort the input and then perform a binary search for each query. Other approaches may also solve the problem within the time constraints and anything that produces the correct solution within the time limits is a reasonable solution (as long as it is well written and can be explained).

To modify your implemented sorting algorithms to work with strings you could simply use `strcmp`. Alternatively, you could produce a more generic sorting algorithm using a function pointer to a comparator function as we did in Lab 4.

To help you we have provided a function for reading in quoted strings in a file *string_helper.h* in the usual place. You can assume this is available. **Update: A few people have struggled to get this function working. There are two alternatives. Firstly, you can write your own within run.c. Secondly, a student has submitted an alternative function that is available on the server and in the usual place called *new_string_helper*.**

Your solution should be implemented in a set of files called *search.h*, *search.c*, and *run.c* where *search.c* implements the searching part of your code and *run.c* contains your main function. This organisation allows you to reuse the search code in the next part. Each sorting run will be subject to a time limit of 5 seconds.

Design an Experiment

Finally, you should design an experiment that answers the question *when is it worth using your proposed solution over a simple linear search?* You should answer this question in terms of m and (optionally) consider how this changes with the balance between successful and unsuccessful queries. To get an interesting answer to this question you would need to have used amortization (e.g. pre-sorting) in your solution.

Your proposed experiment should be implemented in a file called *experiment.c*. Your solution should make use of your code in *search.c* and you should use the header file *search.h* to link the two files together.

We have provided a very large input (in the format expected by the previous part) labelled *data.txt* in the usual place. This file has n around one million. You should have a guess at what value of m your solution improves on the linear solution given this value of n and what you know about the complexities of your sorting, binary search, and linear search algorithms.

Your experiment will be executed on *data.txt* (e.g. `./experiment < data.txt`) and you will be asked to explain what your conclusion is to the above question and how your experiment helped you reach this. The *experiment.c* file should contain sufficient comments to explain what the experiment is doing.

Hint: You may also find it useful to look at the `clock` function provided in `time.h`.

Summary

For this part you should submit files *search.h*, *search.c*, *run.c* and *experiment.c*. Your experiment will be subject to a time limit of 30 seconds.

Submission and Marking Scheme

Note that this may be refined to introduce extra cases reflecting special cases if required.

Implementation of sorting algorithms	
Both algorithms are implemented correctly (all test cases pass) without reusing code from the internet	(2)
Both algorithms are complete but a few test cases fail	(1)
Both algorithms are implemented correctly (all test cases pass) but depends in part on code from the internet	(1)
At least one algorithm is complete and correct	(1)
An attempt has been made but it does not compile or fails most tests	(0.5)
No attempt has been made	(0)

Understanding of sorting algorithms	
The student gives an excellent explanation of both algorithms and how their code implements each algorithm.	(2)
The student gives a good explanation of both algorithms and how their code implements each algorithm. However, they may need prompting in some parts.	(1.8)
The student can explain one of the algorithms.	(1)
The student cannot explain how either of the algorithms work	(0)

Quality of code	
The code is of good quality (e.g. appropriate comments, good layout throughout).	(0.5)
The quality of code is good enough but not great (e.g. only a few comments)	(0.25)
The code is not of acceptable quality (e.g. no comments, bad layout or variable names)	(0)

Record of execution time and comparison of complexities	
Correctly filled in both tables and can reflect on what they mean e.g. how the algorithms from this lab compare to the last lab and why there is a difference between worst and best case complexity. Can give a good explanation of why either merge sort or quick sort have a given complexity.	(2)
Correctly filled in both tables and can mostly explain what this means. Can give a good explanation of why either merge sort or quick sort have a given complexity.	(1.8)
Filled in both tables mostly correctly but missing some information. The explanation of what they have done is good.	(1.5)
Filled in both tables correctly but cannot explain what they have done adequately.	(1)
Made an attempt at filling in the tables but it is not complete	(0.5)
Did not attempt	(0)

The recursive search algorithm is complete, correct, and can be explained.	
The tests pass for the search and the student can explain how binary search works and its complexity.	(2)
The tests pass but the student cannot explain the algorithm and its complexity in enough detail.	(1.5)
Some of the tests pass but the test with large input fails as the implementation is not sufficient enough (it times out)	(1)
An attempt was made	(0.5)
Did not attempt	(0)

An appropriate experiment has been designed and explained	
The designed experiment answers the given question, it runs, and can be sufficiently explained by the student	(1.5)
The designed experiment makes sense and is explained well but does not run as expected	(1)
The designed experiment does an okay job at answering the given question but the student's explanation falls short of answering the question properly	(1)
An attempt was made	(0.5)
Did not attempt	(0)