Edinburgh Napier University

SET09122 Algorithms & Data Structures

Lab 7 - Sorting Part A

Dr Simon Wells

1 Aims

At the end of the practical portion of this topic you will be able to:

- Implement a variety of basic sorting algorithms in a language of your choice.
- Use your timing *test harness* for your algorithms
- Create a plot of your algorithm's performance on datasets of various size
- Consider the factors that might affect the performance of your implementation algorithm
- Understand the circumstances that lead to the best, worst, and average performance charateristics for your algorithms

2 Activities

Last week we saw how the performance of searches on our data depends heavily upon the organisation of our data. Particularly, our data frequently needs to be *sorted* to make it efficient to search. If it is sorted then you immediately avoid the situation where you need to traverse the entire collection. In a good situation, you find the data you're searching for. With sorted data, you have a comparator, so you either find your data, or the dta you are comparing against is larger (or smaller) than your search term. If you are searching within a sorted collection of whole numbers for the value '7' and the last comparator was '6' then you know that the next value will either be '7' or else it will be '8' or larger. If the value is '8' or larger then you can stop searching immediately.

Today we are going to be implementing our own versions of some common sorting algorithms. Under normal circumstances, we would use a standard library implementation for our chosen algorithm, however, to really get a feel for how an algorithm works, and to test that you really understand it, implementation is the best way.

There are other situations where you might need to implement sorting algorithms. Perhaps you need to optimise an existing algorithm for a particular dataset, or the language you are using doesn't have a standard implementation. Sorting algorithms are a favourite task in coding interviews & whiteboard exercises. So it is a good idea to get some practise doing this in advce[1].

2.1 Bubble Sort

This is a simple but very inefficient way to sort data. The algorithm is straightforward: Step through the data comparing each adjacent pair of items & swap them if they're in the wrong order. Repeat until sorted. The idea with this algorithm is that during each pass through we end up finding the largest item and this is eventually, after a series of swaps, moved to the final position. This means that the algorithm establishes a partial sort on each iteration so we can check one less position after each iteration. Each pass through also moves larger items up, and smaller items down, the collection, as they are either temporarily the largest, until we find something bigger, or are smaller and so swapped.

Let's take a look at a bubble sort implementation:

```
1  #include <stdio.h>
2  #define SIZE 10
3
4  int main(void)
5  {
6      int arr[SIZE] = {26, 34543, 17, 31, 13, 543, 456, 1, 0, 2};
7      int idx, target, tmp;
8
9      printf("Bubble Sort\n");
10     printf("Before Sorting: \n");
11     for(idx=0; idx<SIZE; idx++)
12     {
13         printf("%d\t", arr[idx]);
14     }
```

[1] Note that implementing algorithms is a really good way to make sure that you understand them. However once you do understand them you should then just use the standard implementation for your chosen language

```
15      printf("\n");
16
17      // The Bubble Sort
18      for(idx=0; idx<SIZE-1; idx++)
19      {
20          for(target=0; target<(SIZE-1)-idx; target++)
21          {
22              if(arr[target] > arr[target+1])
23              {
24                  tmp = arr[target];
25                  arr[target] = arr[target+1];
26                  arr[target+1] = tmp;
27              }
28          }
29      }
30
31      printf("After Sorting: \n");
32      for(idx=0; idx<SIZE; idx++)
33      {
34          printf("%d\t", arr[idx]);
35      }
36      printf("\n");
37
38
39      return(0);
40  }
```

Notice that we have some boilerplate code here to just print our collection before and after. This kind of traversing an array code should be familiar to you by now as a basic C programming skill. The core of the matter is our set of nested loops. Essentially, we are checking through our collection, using an outer iteration and an inner iteration, both iterations start at zero, but the inner iteration stops further from the end on each pass, i.e. the size of the collection that the inner loop processes gets one smaller on each pass through. This is because the items after that position have been sorted into order on previous passes so need not be looked at again. We then have a check and a possible swap before the cycle starts once more.

Our tasks are as follows:

1. Once you have something that works, try increasing the size of the data to be sorted.
2. You might want to investigate using a random method to populate your data structure with random numbers.
3. *Instrument* your code so that you can see the order of your collection after each iteration of sorting. This is just a fancy way of saying, add some printf statements so that you can see the pattern of changes that are made to your collection as the sort proceeds
4. Investigate how the algorithm would work to sort in the opposite order
5. Sketch out, implement, and run a small experiment to get data describing how your sorting algorithm performs. To achieve this you will need to run your algorithm multiple times for different data sets of the same size, record the time taken, then calculate average runtimes. You will then need to repeat this for different sizes of data set. If you plot your data in a spreadsheet with size of data on the x axis and time on the y axis you should be able to see the shape of the curve that is produced. This is similar to the exercise we did last week and your previous test harness should work nicely. You may have to bundle your algorithm up into a function however.
6. Consider the circumstances under which you are likely to get the best, worst, and average performance for this algorithm. What factors affect the performance of your algorithm?

2.2 Selection Sort

This is possibly the simplest sorting method, it's really a toss up between this and Bubble Sort in terms of impementation. Starting with the first element (current) of the array, compare it to every other element (target) in the array. Each time the current element is greater than the target, swap those elements. After one iteration there will have been a number of swaps but the smallest item

in the array will now be in the first position. Now repeat with the second element comparing to the rest of the array, swapping as necessary until the next smallest item is in the seconde position, and so on, iterating through the array until all items are in the correct place. In some ways this is the inverse of the Bubble Sort, on each pass, instead of finding the largest element and moving it to the end, we are finding the smallest (less than curent value) and moving it to the current position. This means that the smallest items sort to the beginning and our algorithm, on each pass, must only sort from the current position to the end. Just like with Bubble Sort the search space gets smaller on each pass, just that the space shortens from different ends in each case.

```c
#include <stdio.h>
#define SIZE 10

int main(void)
{
    int arr[SIZE] = {26, 34543, 17, 31, 13, 543, 456, 1, 0, 2};
    int idx, target, tmp;

    printf("Selection Sort\n");
    printf("Before Sorting: \n");
    for(idx=0; idx<SIZE; idx++)
    {
        printf("%d\t", arr[idx]);
    }
    printf("\n");

    // The Selection Sort
    for(idx=0; idx<SIZE-1; idx++)
    {
        for(target=idx+1; target<SIZE; target++)
        {
            if(arr[idx] > arr[target])
            {
                tmp = arr[idx];
                arr[idx] = arr[target];
                arr[target] = tmp;
            }
        }
    }

    printf("After Sorting: \n");
    for(idx=0; idx<SIZE; idx++)
    {
        printf("%d\t", arr[idx]);
    }
    printf("\n");


    return(0);
}
```

Again, we are iterating over our collection using two loops, an inner and an outer loop. Notice how the start and finish array indexes for each loop differ. In the outer loop we are looking at every item from the beginning to the end minus one, but the inner loop starts at the beginning plus one, but finishes at the end. So we have our loops offset by one from each other, with the inner loop just one step ahead. We then have a check and a swap.

Note the similarities in structure between this and Bubble Sort. It is well worth comparing your implementation for each. Ignore the topping and tailing code for printing the before and after state of the collection. Instead notice the overall shape of the solution, how we iterate over the content, the number of times we do so,how the starting and finishing positions of each iteration are altered, what is compared to what, and how items are moved around. You should notice similarities between the two, but also crucial differences. Recognising the “shape” of a solution, the strategy for getting from starting position to end position is a useful way to understand and remember algorithms.

Our tasks are as follows:

1. Once you have something that works, try increasing the size of the data to be sorted.

2. You might want to investigate using a random method to populate your data structure with random numbers.

3. *Instrument* your code so that you can see the order of your collection after each iteration of sorting. This is just a fancy way of saying, add some printf statements so that you can see the pattern of changes that are made to your collection as the sort proceeds

4. Investigate how the algorithm would work to sort in the opposite order

5. Sketch out, implement, and run a small experiment to get data describing how your sorting algorithm performs. To achieve this you will need to run your algorithm multiple times for different data sets of the same size, record the time taken, then calculate average runtimes. You will then need to repeat this for different sizes of data set. If you plot your data in a spreadsheet with size of data on the x axis and time on the y axis you should be able to see the shape of the curve that is produced. This is similar to the exercise we did last week and your previous test harness should work nicely. You may have to bundle your algorithm up into a function however.

6. Consider the circumstances under which you are likely to get the best, worst, and average performance for this algorithm. What factors affect the performance of your algorithm?

2.3 Insertion Sort

Another simple sorting algorithm that is relatively efficient when dealing with small amounts of data, or data that is most sorted, but is otherwise beaten by more performant algorithms. The algorithm is relatively straightforward: again we're stepping through the array but on each pass through we are comparing the target element with those before it in the array. If an earlier element is greater than the current element then we shift the elements along to make space for the current element in it's proper place.

```
1  #include <stdio.h>
2  #define SIZE 10
3
4  int main(void)
5  {
6      int arr[SIZE] = {26, 34543, 17, 31, 13, 543, 456, 1, 0, 2};
7      int idx, cmp, shift, tmp;
8
9      printf("Insertion Sort\n");
10     printf("Before Sorting: \n");
11     for(idx=0; idx<SIZE; idx++)
12     {
13         printf("%d\t", arr[idx]);
14     }
15     printf("\n");
16
17     // The Insertion Sort
18     for(cmp=1; cmp<SIZE; cmp++)
19     {
20         for(idx=0; idx<cmp; idx++)
21         {
22             if(arr[idx] > arr[cmp])
23             {
24                 tmp = arr[idx];
25                 arr[idx] = arr[cmp];
26
27                 for(shift = cmp; shift > idx; shift--)
28                     arr[shift] = arr[shift-1];
29                 arr[shift+1] = tmp;
30             }
31         }
32     }
33
34     printf("After Sorting: \n");
35     for(idx=0; idx<SIZE; idx++)
36     {
37         printf("%d\t", arr[idx]);
38     }
39     printf("\n");
40
41
```

```
42      return(0);
43  }
```

Use the same approach from the previous two algorithms to see how this works. Notice the implementation pattern again. For the most part we have two loops, an outer and an inner that process the collection. However, after our check, instead of swapping just a single pair of elements as we did in Bubble and Selection Sort, we now move all elements along so that our target element can be inserted into it's final location. This involves an extra inner loop that fires off every time we need to move an element and that causes at least part of the collection to be looped over a third time.

Our tasks are as follows:

1. Once you have something that works, try increasing the size of the data to be sorted.
2. You might want to investigate using a random method to populate your data structure with random numbers.
3. *Instrument* your code so that you can see the order of your collection after each iteration of sorting. This is just a fancy way of saying, add some printf statements so that you can see the pattern of changes that are made to your collection as the sort proceeds
4. Investigate how the algorithm would work to sort in the opposite order
5. Sketch out, implement, and run a small experiment to get data describing how your sorting algorithm performs. To achieve this you will need to run your algorithm multiple times for different data sets of the same size, record the time taken, then calculate average runtimes. You will then need to repeat this for different sizes of data set. If you plot your data in a spreadsheet with size of data on the x axis and time on the y axis you should be able to see the shape of the curve that is produced. This is similar to the exercise we did last week and your previous test harness should work nicely. You may have to bundle your algorithm up into a function however.
6. Consider the circumstances under which you are likely to get the best, worst, and average performance for this algorithm. What factors affect the performance of your algorithm?

2.4 Coursework

Finally, if you're happy with your understanding of the search algorithms we've looked at so far then feel free to use the lab time to work on your coursework assignment.