Edinburgh Napier University

SET09122 Algorithms & Data Structures

Lab 8 - Sorting Part B

Dr Simon Wells

1 Aims

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

- Implement and understand some more sorting algorithms.
- 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

In one sense, this week's activities complete the basic sorting portion of the module by examining the merge and quick sorts. However, this shouldn't be the end of journey into sorting algorithms. There are many more approaches that have already been discovered, and knowledge of at least some of them should help in your career[1].

2.1 Merge Sort

This is a relatively efficient, general purpose sorting algorithm that produces *stable* sorts.Mergesort is an example of a divide & conquer algorithm[2] and was invented by John von Neumann[3]

Our basic algorithms is quite simple:

1. Repeatedly divide our unsorted data until each portion only contains a single element[4].
2. Repeatedly merge the elements, producing larger collections, comparing and placing in sorted order until all elements are back in the collection. This collection will now be sorted.

Let's take a look at a merge sort implementation (mergsort.c):

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  #define SIZE 10
5
6  void mergesort(int*, int, int);
7  void merge(int*, int, int, int);
8
9  int main(void)
10 {
11     int data[SIZE] = {9, 8, 79, 63, 7, 45, 17, 22, 10, 3};
12     int i;
13
14     printf("Mergesort\n");
15     printf("Before sorting...\n");
16
17     for(i=0; i<SIZE; i++)
18     {
19         printf("%d\t", data[i]);
20     }
21     printf("\n");
22
23     _mergesort(data, 0, SIZE-1);
24
25     printf("After sorting...\n");
26
27     for(i=0; i<SIZE; i++)
28     {
29         printf("%d\t", data[i]);
```

[1] or just be of interest; you are interested in computers and computing aren't you?
[2] remember those from our lectures?
[3] remember him from our lectures? He did an awful lot of interesting computer-related stuff.
[4] Because a list containing a single element is already sorted

```
30      }
31      printf("\n");
32
33      return(0);
34  }
35
36  void _mergesort(int data[], int left, int right)
37  {
38      if (left < right)
39      {
40          int middle = left+(right-left)/2;
41
42          _mergesort(data, left, middle);
43          _mergesort(data, middle+1, right);
44
45          merge(data, left, middle, right);
46      }
47  }
48
49  void merge(int data[], int left, int middle, int right)
50  {
51      int left_idx, right_idx, merged_idx;
52      int left_size = middle - left + 1;
53      int right_size =  right - middle;
54
55      int *tmp_left = (int*) calloc(left_size, sizeof(int));
56      int *tmp_right = (int*) calloc(right_size, sizeof(int));
57
58      for (left_idx = 0; left_idx < left_size; left_idx++)
59          tmp_left[left_idx] = data[left + left_idx];
60      for (right_idx = 0; right_idx < right_size; right_idx++)
61          tmp_right[right_idx] = data[middle + 1+ right_idx];
62
63      left_idx = 0;
64      right_idx= 0;
65      merged_idx = left;
66      while (left_idx < left_size && right_idx < right_size)
67      {
68          if (tmp_left[left_idx] <= tmp_right[right_idx])
69          {
70              data[merged_idx] = tmp_left[left_idx];
71              left_idx++;
72          }
73          else
74          {
75              data[merged_idx] = tmp_right[right_idx];
76              right_idx++;
77          }
78          merged_idx++;
79      }
80
81      while (left_idx < left_size)
82      {
83          data[merged_idx] = tmp_left[left_idx];
84          left_idx++;
85          merged_idx++;
86      }
87
88      while (right_idx < right_size)
89      {
90          data[merged_idx] = tmp_right[right_idx];
91          right_idx++;
92          merged_idx++;
93      }
94      free(tmp_left);
95      free(tmp_right);
96  }
```

Notice that we have broken our solution down into three functions, our main function, which contains some boilerplate code that should be familiar to you from last week. All we are doing here is setting up an array and storing some data, Integers, into it. After that we print out our data in its unsorted form then call our mergsort function, passing in both the data array an the start and end indexes of the data that we want to be sorted within the array. After that we print out the sorted state of the data array. We pass in the indexes of the data within the array to sort,

because this means that if we call the same mergesort function from elsewhere in our code, with different indices, then we can indicate that we want to sort different sub-sections of the array. This is a useful part of making our function work recursively.

We then have two functions which together comprise the mergesort implementation. The first function, mergesort, does our division of the data into smaller and smaller chunks, then the second function performs the merge of the small chunks to reassemble them in sorted order

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 Quick Sort

This is another sorting algorithm. It is considered efficient and was invented by Tony Hoare[5]. Again, we have a form of divide & conquer, rather than dealing with the entire collection at one, we repeatedly find ways to partition the data into smaller units until we have a trivially sorted unit which we can reassembel. The important part is that at each partition step, we choose a position in the partitioned data, called a *pivot* and sort around that position, e.g. compare and sort lower values to one side and higer to the other.Eventually we end up with a series of sorted positions which can be concatenated[6] onto each other to reassemble the original dataset but now in sorted order.

An algorithm for quick sort looks something lke this:

1. Pick an element from the dataset. This element is the *pivot*.
2. Perform a *partitioning* operation: Reorder the dataset so that elements greater than the pivot are to one side of it, and items less than the pivot are to the other side of it[7]. Once complete, the pivot value will be in it's final sorted position.
3. Repeat step 2, applying the partition operation to each separate collection formed respectively from the values smaller than, and greater than the pivot.

[5] Another computer scientist, like John von Neumann, who did important, foundational, research into computer science

[6] A fancy name for linking things together

[7] Equal values can be to either side of the pivot, just be consistent

An interesting point of the algorithm is that after the partition operation, due to the sort that is done, the pivot value actually ends up in it's final place in the sorted collection. Also note that step 2, the partition, performs both a split and a sort operation, e.g. compare each value in the collection to the pivot, but the sort is quite coarse grained, at each partition step it's only interested in getting values to the right side of the pivot, not ensuring that those values are sorted after they are moved. It's the recursive aspect of reapplyingthe partition operation that leads to the final sort of all elements.

Let's take a look at our quick sort implementation (quicksort.c):

```
1  #include <stdio.h>
2
3  #define SIZE 10
4
5  void quicksort(int*, int, int);
6  int partition(int*, int, int);
7
8  int main(void)
9  {
10     int data[SIZE] = {9, 88, 79, 63, 7, 45, 17, 22, 1, 3};
11     int i;
12
13     printf("Quicksort\n");
14     printf("Before sorting...\n");
15
16     for(i=0; i<SIZE; i++)
17     {
18         printf("%d\t", data[i]);
19     }
20     printf("\n");
21
22     quicksort(data, 0, SIZE-1);
23
24     printf("After sorting...\n");
25
26     for(i=0; i<SIZE; i++)
27     {
28         printf("%d\t", data[i]);
29     }
30     printf("\n");
31
32     return(0);
33 }
34
35 void quicksort(int data[], int lower, int upper)
36 {
37     int pivot;
38     if(upper > lower)
39     {
40         pivot = partition(data, lower, upper);
41         quicksort(data, lower, pivot-1);
42         quicksort(data, pivot+1, upper);
43     }
44 }
45
46 int partition(int data[], int lower, int upper)
47 {
48     int pivot = data[upper];
49     int idx = lower-1;
50     int temp;
51
52     for (int j=lower; j<upper; j++)
53     {
54         if(data[j] <= pivot)
55         {
56             idx++;
57             temp=data[idx];
58             data[idx] = data[j];
59             data[j] = temp;
60         }
61     }
62
63     temp=data[idx+1];
64     data[idx+1] = data[upper];
65     data[upper] = temp;
```

```
66
67      return idx+1;
68  }
```

Note that the overall implementation has a similar structure to our merge sort approach, Two functions, one the divides the data in some way, and another function that reassembles the data into a whole, sorted collection. Note that the our data isn't really divided up in this case, we are just using starting and ending indexes over various ranges within our data and swapping data around in place. The choice of value to use for the pivot need not be fixed. Choosing the leftmost element[8] can lead to worst case behaviour if applied to sorted data. Other pivot selection strategies include, choosing a random index, or the middle index from the partition, or the median value from the set of the first, middle, and last values in the partition.

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 Coursework

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

[8] this was originally the default in early versions of the algorithm