Eric Watson

ECE 331 Homework 6

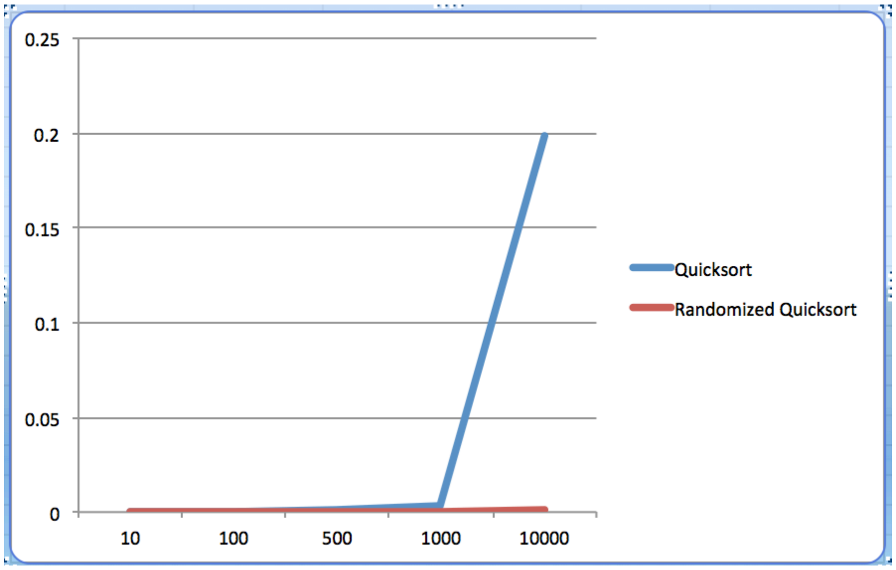
Quicksort versus Randomized Quicksort

This assignment required us to generate c functions to implement both the Quicksort and Randomized Quicksort algorithms, and then compare the execution times of the algorithms when sorting arrays consisting of elements that are already sorted. To implement the functions I followed the pseudo code provided in the textbook very closely. Within my program I would then populate an array with random integers from 0 to 999, and then sort the source array using the insertion sort function that I had implemented in a previous homework assignment. Once this array was sorted, I would then copy the sorted values into a second source array so that the same array could be used as a fair comparison for the two algorithms. Next each array was passed to a corresponding sorting method, and the execution times were measured using the C library function clock(). The results of my tests for various array test sizes are shown in the table below.

Execution Times for Various Array Sizes (In Microseconds)

Array Size	10	100	500	1,000	10,000
Quicksort	6	43	982	3,081	198,594
Randomized Quicksort	2	15	72	102	1,009

Execution Time Comparison Plot
Array Size (x-axis) vs Execution Time in seconds (y-axis)



The running time of Quicksort depends on whether the partitioning is balanced or unbalanced. Thus, its execution time depends on which elements are used for partitioning. If the partitioning is balanced, the algorithm runs asymptotically as fast as Merge sort. However, if the partitioning is unbalanced, it can run asymptotically as slowly as Insertion sort. For this reason, we see that when the values of our array are already sorted in ascending order prior to being passed to Quicksort, then this invokes the worst-case Quicksort performance ($\theta(n^2)$ on an input of n numbers). Looking at the table above we can confirm that the Quicksort algorithm performed significantly slower in this situation.

Despite Quicksort's slow worst-case running time, this sorting algorithm is often the best practical choice for sorting because it is remarkably efficient on the average. It also has the advantage of sorting in place.

The Quicksort and Randomized Quicksort procedures differ only in how they select pivot elements, they and are the same in all other aspects. Using the randomized partition, the expected running time of quicksort is O(n lg n) when element values are distinct. Because we randomly choose the pivot element with this algorithm, rather than choosing the last element in the array (such as in Quicksort), we expect the split of the input array to be reasonably well balanced on average.

Based on the results of this experiment, as well further analysis, I would choose to use Quicksort when there is balanced partitioning. That is, when the values of the array elements are randomized. On the other hand, when there is unbalanced partitioning and the array values are nearly sorted (such as the situation that we tested) then I would use Randomized Quicksort.

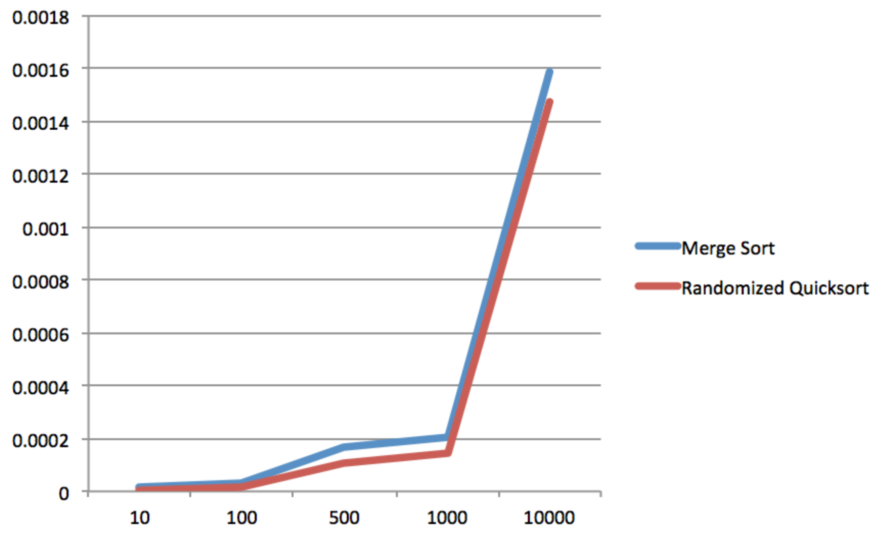
<u>Merge Sort versus Randomized Quicksort</u>

The second part of this assignment required us to compare execution times of the randomized quick sort and merge sort algorithms when sorting randomly populated arrays of various sizes. Once again I began by populate an array with random integers from 0 to 999. Then I copied these values into a second array so that the array to be sorted would be the same for each algorithm, thereby ensuring a valid comparison. Next each array was passed to my merge sort and randomized quick sort functions, using the c library function clock() to measure the execution time of each sorting method. The results of my comparison for various array sizes are displayed in the table below.

Execution Times for Various Array Sizes (In Microseconds)

Array Size	10	100	500	1,000	10,000
Merge Sort	14	32	167	206	1,586
Randomized Quicksort	4	15	108	142	1,477

Execution Time Comparison Plot
Array Size (x-axis) vs Execution Time in seconds (y-axis)



Randomized Quicksort, like merge sort, applies the divide-and-conquer method. The worst case running time of Merge sort is Θ(nlg(n), whereas the worst case for Quicksort is O(n^2). However, when choosing the partitioning pivot element at random, it is very easy to avoid Quicksort's worst case. Therefore, the expected running time of Randomized Quicksort is O(n lg n). We can see in the table above that the Randomized Quicksort algorithm performs slightly faster than Merge Sort for every array size tested. It is also important to make note of the memory usage for these sorting algorithms. Randomized Quicksort uses (log n) memory on average, and (n) in the worst case, where as Merge sort uses a lot of memory(n) in all scenarios. Quicksort requires little additional space and exhibits good cache locality, which can make it faster than merge sort in many cases.

Based on the results of this experiment, in addition to further analysis, I would choose to use Randomized Quick sort when looking for a fast sorting method that requires less memory. I would choose Merge Sort when I have plenty of memory, and need to use a sorting algorithm that is stable, and relatively fast.

The terminal output below demonstrates the full functionality of my program for array size 10. In order to prevent a lengthy terminal output, I chose to print only the first 10 elements of arrays for debugging purposes. In the first part of the assignment I print the original array that is populated with random values. Next I print out the sorted array of these values, which is then copied into a second source array to be passed to the sorting algorithms. After passing the source arrays to Quicksort and Randomized Quicksort, I then print the execution times of the functions, as well as the values of the arrays to ensure that the sorting algorithms performed correctly. Similarly, for the second part of the assignment I print out the original source array of randomized values, which will then be copied into a second source array (not sorted this time!) to be passed to the sorting algorithms. After passing the random-valued arrays to Merge Sort and Randomized Quicksort, I print the execution times of these sorting methods, and then also print arrays to ensure that the values have been sorted correctly.

```
Erics-MacBook-Pro-2:HW06 Watson$ gcc HW06.c
Erics-MacBook-Pro-2:HW06 Watson$ ./a.out
Input Array Size: 10
The first 10 randomly generated values are:
280
190
925
696
113
666
337
814
151
113
The first 10 Sorted values are:
113
113
151
190
280
337
666
696
814
925
Quick Sort Execution Time: 0.000004 seconds
The first 10 Quick-Sorted values are:
113
113
151
190
280
337
666
696
814
925
Randomized Quick Sort Execution Time: 0.000002 seconds
The first 10 Randomized Quick-Sorted values are:
113
113
151
190
280
337
666
696
814
925

PART 2
The first 10 randomly generated values are:
694
980
591
764
268
929
999
158
321
238
Merge Sort Execution Time: 0.000014 seconds
The first 10 Merge-Sorted values are:
158
```

```
238
268
321
591
694
764
929
980
999
Randomized Quick Sort Execution Time: 0.000004 seconds
The first 10 Randomized Quick Sorted Values Are:
158
238
268
321
591
694
764
929
980
999
```

Eric Watson

ECE 331 Homework 6 Source Code (HW06.c)

```c
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <math.h>

void print_first_ten(int A[]){
  int i;
  for (i =0; i < 10; i++){
    printf("%d\n", A[i]);
  }
}

int quick_partition(int A[],int p,int r){
  int j,q,z;
  int x = A[r];               // pivot element
  int i = p-1;
  for (j=p; j < r; j++){
    if (A[j] <= x){           //partition elements less than x
                              // if greater than x then they are left where they are
      i++;
// exchange A[i] with A[j]
      q = A[j];
      A[j] = A[i];
      A[i] = q;
  }
}
  // exchange A[i+1] with A[r]
  z = A[r];
  A[r] = A[i+1];
  A[i+1] = z;
  return (i+1);
}

int randomized_partition(int A[], int p, int r){
  int q;
  int i = (rand() %(r-p)) + p;
  //exchange A[r] with A[i]
  q = A[i];
  A[i] = A[r];
  A[r] = q;

  return quick_partition(A,p,r);
}

void randomized_quicksort(int A[], int p, int r){
  int q;
  if (p<r){
    q = randomized_partition(A,p,r);
    randomized_quicksort(A,p,q-1);
    randomized_quicksort(A, q+1, r);
  }
}

void quicksort (int A[], int p, int r){
  int q;
  if (p < r){
    q = quick_partition(A,p,r);
    quicksort(A,p,q-1);
    quicksort(A,q+1,r);
  }
}

void insertion (int A[], int size){
  int i,j, key;
  for (j= 1; j<size; j++){
    key = A[j];
// insert A[j] into the sorted sequence A[1..j-1]
    i =j-1;
    while (i >=0 && A[i] > key){
      A[i+1] = A[i];
      i = i-1;
    }
    A[i+1] = key;
  }
}

void merge (int B[], int p, int q, int r){
  int i, j, k;
  int n1 = q - p + 1; //length of subarray A[p..q]
  int n2 = r - q;                 //length of subarray A[q+1..r]
// let L[1..n1+1] and R[1..n2+1] be new arrays
  int L[n1+1], R[n2+1];

  for (i=0; i< n1; i++){
    L[i] = B[p+i-1];
  }

  for (j=0; j< n2; j++){
    R[j] = B[q+j];
  }
//       Issue sentinel cards
  L[n1] = 9999999;
  R[n2] = 9999999;

  i = 0;
  j = 0;

  for (k = (p-1); k < r; k++){
    if (L[i] <= R[j]){
      B[k] = L[i];
      i = i + 1;
    }
    else{
      B[k] = R[j];
      j=j+1;
    }
  }
}

void merge_sort(int B[], int p, int r){

  if (p < r){
    int q = floor((p+r)/2);
    merge_sort(B, p, q);
    merge_sort(B, q+1, r);
    merge (B,p,q,r);
  }
}

int main(){

  clock_t begin_quick, end_quick, begin_rand1, end_rand1, begin_rand2, end_rand2,
begin_merge, end_merge;
  double quick_time;
  double rand1_time;
  double rand2_time;
  double merge_time;
  int size,i,j,k,l,m,x,w,z;

  time_t seconds;
  time(&seconds);
  srand((unsigned int) seconds);

  printf("Input Array Size: ");
  scanf("%d", &size);
  int A[size];                      // Quick sort, sorted values
  int B[size];                      // Randomized quick sort, sorted values
  int C[size];                      // Randomized quick sort, unsorted values
  int D[size];                      // Merge sort, unsorted values

// FIRST PART OF ASSIGNMENT: QUICK SORT VS RANDOMIZED QUICK SORT - SORTED VALUES
  for (i=0; i< size; i++){          // Populate A with random integers
    A[i] = rand() % 1000;
  }

//DEBUGGING - Make sure the integers are randomly generated
  printf("The first 10 randomly generated values are:\n");
  print_first_ten(A);

  insertion(A,size);                // sort A
  printf("The first 10 Sorted values are:\n");
  print_first_ten(A);               // Show that Array was sorted properly

  for (z=0; z < size; z++){         // Copy the sorted values of A into B
    B[z] = A[z];
  }

// Now that A and B are sorted, we can pass them to quick sort and randomized quick sort
  begin_quick = clock();            // start clock
  quicksort(A,0, size-1);           // Quick sort the sorted values of A
  end_quick = clock();              // end clock
  quick_time = (double)(end_quick-begin_quick)/ CLOCKS_PER_SEC;
  printf("Quick Sort Execution Time: %f seconds\n", quick_time);
  printf("The first 10 Quick-Sorted values are:\n");
  print_first_ten(A);

  begin_rand1 = clock();            // start clock
  randomized_quicksort(B,0, size-1);// randomized quick sort
  end_rand1 = clock();              // end clock
  rand1_time = (double)(end_rand1-begin_rand1)/ CLOCKS_PER_SEC;
  printf("Randomized Quick Sort Execution Time: %f seconds\n", rand1_time);
  printf("The first 10 Randomized Quick-Sorted values are:\n");
  print_first_ten(B);

// SECOND PART OF ASSIGNMENT: RANDOMIZED QUICK SORT VS MERGE SORT - RANDOM VALUES

  for (i=0; i< size; i++){         //populate C with random integers
    C[i] = rand() % 1000;
  }
  //DEBUGGING - Make sure the integers are randomly generated
  printf("\nPART 2\n");
  printf("The first 10 randomly generated values are:\n");
  print_first_ten(C);

  for (z=0; z < size; z++){       // Copy the random valued integer array into D
    D[z] = C[z];
  }
// Leave array C and D populated with these unordered values

  begin_merge = clock();          //start clock
  merge_sort(C,1,size);           // merge sort
  end_merge = clock();            // end clock
  merge_time = (double)(end_merge-begin_merge)/ CLOCKS_PER_SEC;

  printf("Merge Sort Execution Time: %f seconds\n", merge_time);

//DEBUGGING - Make sure the values are sorted properly
  printf("The first 10 Merge-Sorted values are:\n");
  print_first_ten(C);

  begin_rand2 = clock();           //start clock
  randomized_quicksort(D,0,size-1);// randomized quick sort
  end_rand2 = clock();             // end clock
  rand2_time = (double)(end_rand2-begin_rand2)/ CLOCKS_PER_SEC;
  printf("Randomized Quick Sort Execution Time: %f seconds\n", rand2_time);

//DEBUGGING - Make sure the values are sorted properly
  printf("The first 10 Randomized Quick Sorted Values Are:\n");
  print_first_ten(D);

  return(0);
}
```