CSC 212: Data Structures and Abstractions

Spring 2019

University of Rhode Island

Weekly Problem Set #6

Due Thursday 4/4 at the beginning of class. Please turn in neat, and organized, answers hand-written on standard-sized paper **without any fringe**. At the top of each sheet you hand in, please write your name, and ID.

1. Implement merge sort. Your function should take an array of integers and the indices of the first and last elements in the list to sort.

```
// Merges two subarrays of arr[].
// First subarray is arr[l..m]
// Second subarray is arr[m+1..r]
void merge(int arr[], int l, int m, int r)
{
    int i, j, k;
    int n1 = m - l + 1;
    int n2 =  r - m;

    /* create temp arrays */
    int L[n1], R[n2];

    /* Copy data to temp arrays L[] and R[] */
    for (i = 0; i < n1; i++)
        L[i] = arr[l + i];
    for (j = 0; j < n2; j++)
        R[j] = arr[m + 1+ j];

    /* Merge the temp arrays back into arr[l..r]*/
    i = 0; // Initial index of first subarray
    j = 0; // Initial index of second subarray
    k = l; // Initial index of merged subarray
    while (i < n1 && j < n2)
    {
        if (L[i] <= R[j])
        {
            arr[k++] = L[i++];
        }
        else
```

```
        {
            arr[k++] = R[j++];
        }
    }

    /* Copy the remaining elements of L[], if there
       are any */
    while (i < n1)
    {
        arr[k++] = L[i++];
    }

    /* Copy the remaining elements of R[], if there
       are any */
    while (j < n2)
    {
        arr[k++] = R[j++];
    }
}

/* l is for left index and r is right index of the
   sub-array of arr to be sorted */
void mergeSort(int arr[], int l, int r)
{
    // only run if more than 1 element
    if (l < r)
    {
        // Same as (l+r)/2, but avoids overflow for
        // large l and r
        int m = l+(r-l)/2;

        // Sort first and second halves
        mergeSort(arr, l, m);
        mergeSort(arr, m+1, r);

        merge(arr, l, m, r);
    }
}
```

2. Implement quicksort. Your function should take an array of integers and the indices of the first and last elements in the list to sort.

```
void swap(int* a, int* b)
{
    int t = *a;
    *a = *b;
    *b = t;
}

/* This function takes last element as pivot, places
   the pivot element at its correct position in sorted
    array, and places all smaller (smaller than pivot)
   to left of pivot and all greater elements to right
   of pivot */
int partition (int arr[], int low, int high)
{
    int pivot = arr[high];    // pivot using Lomuto partition scheme
    int i = (low - 1);  // Index of smaller element. Initialize index outside boundaries.

    for (int j = low; j <= high- 1; j++)
    {
        // If current element is smaller than or
        // equal to pivot
        if (arr[j] <= pivot)
        {
            i++;    // increment index of smaller element
            swap(&arr[i], &arr[j]);
        }
    }
    swap(&arr[i + 1], &arr[high]);
    return (i + 1);
}

/* The main function that implements QuickSort
 arr[] --> Array to be sorted,
  low  --> Starting index,
  high  --> Ending index */
void quickSort(int arr[], int low, int high)
{
    if (low < high)
    {
        /* piv is partitioning index, arr[p] is now
           at right place */
```

```
            int piv = partition(arr, low, high);

            // Separately sort elements before
            // partition and after partition
            quickSort(arr, low, piv - 1);
            quickSort(arr, piv + 1, high);
        }
    }
```

3. Define a recurrence relation for the best case for quicksort. Assume that a partition for an array of size n takes n comparisons.
   $T(n) = 2T(n/2) + n$

4. Solve your recurrence relation.

$$T(1) = 1; T(n) = 2T(\frac{n}{2}) + n$$

$$\begin{aligned} T(n) &= 2T(\frac{n}{2}) + n \\ &= 2(2T(\frac{n}{4}) + \frac{n}{2}) + n \\ &= 4T(\frac{n}{4}) + 2n \\ &= 8T(\frac{n}{8}) + 3n \\ &= \dots \end{aligned} \tag{1}$$

   This pattern can be written as follows:

$$T(n) = 2^k T(\frac{n}{2^k}) + kn$$

   Becoming trivial when $\frac{n}{2^k} = 1$ or $k = \log_2 n$ Putting it all together:

$$T(n) = nT(1) + n\log_2 n = n\log_2 n + n$$

5. Define a recurrence relation for the worst case for quicksort. Assume that a partition for an array of size n takes n comparisons.
   $T(n) = T(1) + T(n-1) + n = T(n-1) + n + 1$

6. Solve your recurrence relation.

$$T(1) = 1; T(n) = T(n-1) + n + 1$$

$$\begin{aligned} T(n) &= T(n-1) + n + 1 \\ &= T(n-2) + n + (n+1) \\ &= T(n-3) + (n-1) + n + (n+1) \\ &= T(n-4) + (n-2) + (n-1) + n + (n+1) \\ &= \dots \end{aligned} \tag{2}$$

The solution to the problem can be found on this piazza post:
https://piazza.com/class/jr9cnq8ekfq3k4?cid=246

7. What are two ways to minimize the chance that worst case occurs?

```
The worst case for this particular partition scheme is an
array which is already ordered, causing maximal inbalance
post-partition.
One way, (discussed in class) is to shuffle the array prior
to sorting it, this minimizes the chance the array will end
up sorted.
Another which is more commonly practiced, is to use a
different partition scheme (random point vs always the first/last)
```