The University of Newcastle
School of Information and Physical Sciences

COMP2230/6230 Algorithms

Tutorial Week 7 Solutions

30th August – 3rd September 2021

Tutorial

1. The following algorithm (Algorithm 5.1.4 from the text) is an algorithm for tiling a deficient plane with L shaped tiles (trominoes). Give a recurrence relation for the running time of the algorithm. Give the asymptotic solution for this recurrence relation.

```
Input Parameters: n, a power of 2 (the board size);
the location L of the missing tile
Output Parameters: none

tile(n,L)
{
   if  (n == 2){
       //the  board  is  a  right  tromino  T
       tile  with  T
       return
    }
   divide  the  board  into  four  n/2  by  n/2  subboards
   place  one  tromino  in  the  center
   //this  tile  is  placed  so  that  each  subboard  has  a
   //missing  square
   //let m[i] denote the location of the missing squares
   for  i  =  1  to  4  do
        tile(n/2, m[i])
}
```

Solution. The running time of the algorithm can be expressed with the following recurrence:

$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 2 \\ 4T(n/2) + \Theta(1) & \text{if } n > 2 \end{cases}$$

Note that there is a presumption that $n = 2^i$ for some i, otherwise the algorithm wouldn't work anyway, so we don't have to worry about the recurrence too much. Applying the main recurrence theorem (p. 63 of the text) with a = 4, b = 2, and k = 0, we have that $a > b^k$, so the third bound applies. Therefore

$$T(n) = \Theta(n^{\log_b a}) = \Theta(n^2).$$

2. Binary insertion sort sorts the array a[1, . . . , n] by using a binary search to determine where to place the next element to be sorted. That is, if the array is sorted up to index k, then a binary search is performed to determine where a[k + 1] should be inserted. What is wrong with the following statement? Since binary search is Θ(log n) in the worst case, and there are n − 1 elements to insert, the worst case time of binary insertion sort is Θ(n log n).

Solution: This analysis does not take into account the fact that even though it will only take $\Theta(\log n)$ time to find where to insert the next element, all the elements from that point must be shifted to the right to make room (otherwise the first one would be overwritten). This takes $\Theta(n)$ time, giving the same $\Theta(n^2)$ bound for n inserts.

3. Counting sort

```
Input Parameters: a,m
Output Parameters: a

counting_sort(a,m)
{
    // set c[k] = the number of occurrences of value k
    // in the array a.
    // begin by initializing c to zero.
    for k = 0 to m
        c[k] = 0
     n = a.last
     for i = 1 to n
        c[a[i]] = c[a[i]] + 1
     // modify c so that c[k] = number of elements ≤ k
     for k = 1 to m
        c[k] = c[k] + c[k - 1]
    // sort a with the result in b
     for i = n downto 1 {
         b[c[a[i]]] = a[i]
          c[a[i]] = c[a[i]] - 1
     }
     // copy b back to a
     for i = 1 to n
         a[i] = b[i]
}
```

Trace the counting sort for the following array: 15 3 17 2 9 10 8 10

Solution: In the text, page 702 (exercise 1, Section 6.4)

4. Consider the following version of the counting sort.

```
   counting_sort(a,m)
{
 for k = 0 to m - 1
  c[k] = 0
 for i = 1 to n
  c[a[i]] = c[a[i]] + 1
 for k = 1 to m
  c[k] = c[k] + c[k - 1]
 for i = 1 to n {
  b[c[a[i]]] = a[i]
  c[a[i]] = c[a[i]] - 1
 }
 for i = 1 to n
  a[i] = b[i]
}
```

Is this algorithm correct? If so, is it stable?

Solution:

Line 2: *m-1* should be *m*

Add *n* = *a.last* after the third line

The sort is not stable as *for loop* in Line 8 goes from *1* to *n* instead of from *n* downto *1*.

Homework

5. Write an algorithm that takes an array T [1, . . . , n] of real numbers and a real number v, and returns true if there exists two indices i and j such that T [i] +T [j] = v and false otherwise. What is the running time of your algorithm?

Solution:

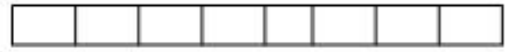
```
findij(real[]  T,  real v) {
           mergeSort(T);
           for  (int  i = 0; i < T.length;  i++){

                  int  j  =  binarySearch(T,v-T[i]);
                  if  (j  !=  -1){ print(i  +  "  "  +  j);
                       return    true;
                   }
            }
             return  false;
}
```

This algorithm has running time Θ(n log n).

6. Write an algorithm that computes the union A ∪ B of two n-element sets of real numbers. The sets are represented as sorted arrays (there are no duplicates within a set). The result should be a sorted array with no duplicates. What is the running time of this algorithm?

   Solution: Simply run the merge algorithm used in mergesort on the two arrays with the modification that if A[i] = A[j], you insert one of them, and increment both i and j , rather than just one of them. As there are no duplicates in A and B , we do not have to worry about dealing with multiple recurrences of the same number, so when we see it once, that's it. This has linear running time ($\Theta(n)$).



7. Show how insertion sort works on the following array:

   14 40 31 28 3 15 17 51

   Solution: See the text, p. 240, or the lecture slides (Lecture 8, Sorting) for examples of Insertion Sort.

8. Show how the partition algorithm from quicksort partitions the array in workshop question 7.

   Solution: See the text, pp. 243-246, or the lecture slides (Lecture 8, Sorting) for examples of quicksort.

9. Trace the counting sort for the following array: 15 3 17 2 9 10 8
   Solution: In the text, page 702 (exercise 1, Section 6.4)

Extra Questions

10. Prove that the running time of the insertion sort in the worst case is $\Theta(n^2)$, by proving both O and Ω.

    Solution idea: Let T(n) be the number of times a barometer statement is executed in the worst case. By analysing the two loops in the algorithm, we obtain $T(n) \leq 1+2+3+\ldots+n=n(n-1)/2$; therefore $T(n)= O(n^2)$. On the other hand, when the input is an array sorted in decreasing order, the barometer statement will execute n(n-1)/2 times, and thus $T(n) = \Omega(n^2)$. Therefore $T(n)= \Theta(n^2)$.

11. Prove that the running time of the quicksort in the worst case is $\Theta(n^2)$, by proving both O and Ω.

    Solution: see Theorem 6.2.5 from the textbook.

12. Show that any sorting algorithm that moves data only by swapping adjacent elements has worst cast time $\Omega(n^2)$.

Solution:

Swapping adjacent elements remove at most one inversion. Since there are n(n-1)/2 inversions in a decreasing sequence, an algorithm that swaps only adjacent elements must remove $\Theta(n^2)$ inversions when the input is a decreasing sequence. Such algorithm requires worst-case time $\Omega(n^2)$.

13. Show that if the size of the array is smaller than some number s, which is dependent on the implementation, system, etc., that insertion sort can run faster than mergesort. Modify mergesort to take advantage of this so that mergesort runs faster.

Solution:

We consider the variable s to be global. insertion_sort2 is insertion sort suitably modified so that it sorts a[i],……..,a[j].

```
mergresort2(a, i, j) {

    if(i-j<= s)

           insertionsort2(a, i, j)

    else{

           m=(i +j )/2;

           mergesort2(a, i, m)

           mergesort2(a, m+1, j)

           merge(a,i, j)

    }

}
```

14. What is the time of quicksort when the values of all the data are equal?

Solution:

$\Theta(n^2)$

15. Show how mergesort merges the array in exercises 5.2.1-5.2.4 in the text.

Solution:

Mergesort divides the array to be sorted into two nearly equal parts. Each part is then sorted. The two sorted halves are then merged into one sorted array.

16. Write a non-recursive version of mergesort.

Solution:

```
Merge_small_extra_space(a, m, n){
          if(m<=n) {
              // copy of a[1]…….a[m] into b
              for i= 1 to m
                  b[i]=a[i]
              // merge a[m+1]...a[m+n] and b into a
              p=r=1;
              q=m+1;
              while(p<= m && q<= m+n) {
                  if(b[p]<= a[q] ) {
                         a[r] = b[p]
                         p= p+1;
                  }
                  else {
                         a[r]= a[q]
                         q=q+1;
                  }
                  r=r+1;

               } // while

              while( p<= m) {
                  a[r]=b[p]
                  p=p+1;
                  r=r+1;
               }//while
               while( q<= m+n) {
                  a[r]=a[q]
                  q=q+1;
                  r=r+1;
               }//while
            }// if end
            else {// m>n
                // copy a[m+1]……..a[m+n] into b
                for i = 1 to n
                    b[i]= a[i+m]
                // merge a[1] …….a[m] and b into a; in reverse
                p=n
                q=m
                r= m+n
                while(p>=1 && q>= 1){
                      if(b[p]>=a[q] {
                           a[r]=b[p]
                           p=p-1;
                       }//if
                       else {
                           a[r]=a[q]; q=q-1;
                       }//else
                       r=r-1
                }//while

                while (p>=1){
                       a[r]=b[p]
                       p=p-1; r=r-1;
                }//while
                while(q>=1){
                       a[r] =a[q]; q=q-1; r=r-1;
                }//while

             }// else
       }//main
```

17. Trace counting sort for the following arrays:

a. 4 4 4 4 4 4
b. 2 3 4 5

Solution:

The algorithm first sets c[k] to the number of occurrences to the value of the k in the array. The nonzero value is c[4]=6. Modifying c so that c[k] is equal to the number of elements in the array lesss than or equal to k does not change the value of c[4].

The last element of the array is 4. Since c[4]=6, 4 is copied to the sixth cell of b

					4

The 5th element of array is also 4. Since c[4]=5, is copied to the fifth cell of b.

				4	4

Similarly decrementing c[4] and so on we get the sorted array

4	4	4	4	4	4

18. Trace the radix sort for each of the following arrays:

a. 34 9134 20134 29134 4 134
b. 4 34 134 9134 20134 29134

Solution:

(b) After sorting on the 1's digit, the array is unchanged. After sorting on the 10's digit , the array is also unchanged. After sorting on the 100's digit , the array is also unchanged. After sorting on the 1000's digit, the array becomes

4 34 134 20134 9134 29134

After sorting on the 10000's digit , the array is also unchanged..

4 34 134 9134 20134 29134

19. What is the worst-case time of radix sort if we replace counting sort by insertion sort? By mergesort?

Solution:

$\Theta(kn^2)$
$\Theta(kn \lg n)$