CSCE 411-502 Design and Analysis of Algorithms

Spring 2025

Instructor: Dr. Jianer Chen
Office: PETR 428
Phone: (979) 845-4259
Email: chen@cse.tamu.edu
Office Hours: MW 1:30 pm–3:00 pm

Senior Grader: William Kang
Phone: 979) 575-9987
Email: rkdvlfah1018@tamu.edu
Questions: via phone and email and by appointments

Assignment #1 Solution

1. Write a recursive binary-search algorithm B-Search(A[0..n-1], x) that searches the given number x in the array A[0..n-1] that is sorted in non-decreasing order. Give a detailed analysis on the time complexity of your algorithm, including presenting the recurrence relations, and the procedure that solves the recurrence relations.

Solution. The algorithm consists of a main program and a recursive function B-Search, as follows.

```
B-Search(A,l,r,x)
1.  if (l > r) return (false);
2.  if (l = r) return (A[r] == x);
3.  m = (l+r)/2;
4.  if (x <= A[m]) return B-Search(A,l,m,x);
    else return B-Search(A,m+1,r,x).

main(A[0..n-1], x)
   return B-Search(A,0,n-1,x).
```

Time Complexity Analysis:
Assume that the recursive algorithm B-Search(A,l,r,x) runs in time $T(h)$ on the subarray A[l..r] of $h = r - l + 1$ elements. By steps 1-2, we have $T(h) = O(1)$ when $h \leq 1$. For the case where $h > 1$, step 4 shows that the recurrence relation is $T(h) \leq T(h/2) + O(1)$ (note that only one of the two recursive calls is executed in step 4). This gives the following recurrence relation:

$$T(h) = T(h/2) + O(1) \text{ for } h \geq 2 \qquad \text{and} \qquad T(h) = O(1) \text{ for } h \leq 1.$$

To solve the recurrence relation, we follow the procedure given in the class. First, we replace the recurrence relation by

$$T(h) \leq T(h/2) + c \text{ for } h \geq 2 \qquad \text{and} \qquad T(h) \leq c \text{ for } h \leq 1. \tag{1}$$

By the recurrence relation, we also have $T(h/2) \le T(h/2^2) + c$. Thus, replacing $T(h/2)$ in the first inequality in (1) with $T(h/2^2) + c$, we get

$$T(h) \le T(h/2^2) + 2c. \tag{2}$$

Similarly, by $T(h/2^2) \le T(h/2^3) + c$ and (2), we get

$$T(h) \le T(h/2^3) + 3c.$$

Now it should be clear that for general k, we should have

$$T(h) \le T(h/2^k) + kc. \tag{3}$$

Letting $k = \log h$ in (3) gives

$$T(h) \le T(h/2^{\log h}) + c \log h = T(1) + c \log h \le c + c \log h = O(\log h).$$

Since the main program calls on the array A[0..n-1] of size n, we conclude that the above binary search algorithm runs in time $O(\log n)$ on input arrays of n elements.

2. Consider the following array:

A =

21	15	32	6	7	12	3	29	1	15

Apply the algorithm HeapSort to sort the array. Give the content of the array **after**:

(1) the execution of the algorithm MakeHeap(A[0..9]) ;
(2) the 1st, 4th, 7th, and 9th executions of the while-loop in the algorithm SortHeap(A[0..9]).

(see the lecture notes for details of the algorithms.)

Solution.

(1) After MakeHeap(A[0..9]):

A =

32	29	21	15	15	12	3	6	1	7

(2)

(2.1) After the 1st execution of the while-loop in SortHeap(A[0..9]):

A =

29	15	21	7	15	12	3	6	1	32

heap tail t = 8.

(2.2) After the 4th execution of the while-loop in SortHeap(A[0..9]):

A =

15	7	12	3	6	1	15	21	29	32

heap tail t = 5.

(2.3) After the 7th execution of the while-loop in SortHeap(A[0..9]):

A =

6	3	1	7	12	15	15	21	29	32

heap tail t = 2.

(2.4) After the 9th execution of the while-loop in SortHeap(A[0..9]):

A =

1	3	6	7	12	15	15	21	29	32

heap tail t = 0.

3. Suppose that a heap is given by an array H[0..n-1] and an integer t (i.e., the tail of the heap), where t < n-1. Write an algorithm Insert(H[0..n-1], t, x) that add a new element x to the heap (and make the result a heap again). What is the time complexity of your algorithm?

Solution. Basic idea: since t < n-1, the array element H[t+1] exists and is not used by the heap. Thus, we can increase the heap size by 1 then place the new element x in that position. Note that only the value of the new element x in the new structure can violate the conditions of heap structures. Thus, we simply use FixHeap to restore the heap structure. The algorithm is given as follows:

```
Insert(H[0..n-1],t,x)
1.  t = t + 1;
2.  H[t] = x;
3.  FixHeap(H,t,t).
```

Steps 1-2 take time $O(1)$. Step 3 calls FixHeap on a heap of size $t + 1 \leq n$, which, by the discussion in the class, takes time $O(\log(t + 1)) = O(\log n)$.