Technische Universität München
Fakultät für Informatik
Lehrstuhl für Effiziente Algorithmen
Dmytro Chibisov
Sandeep Sadanandan

Winter Semester 2007/08
Solution Sheet 4
November 14, 2007

Fundamental Algorithms

Problem 1 (15 Points)

The iteration operator "*" used in the $\lg^*$ function can be applied to monotonically increasing functions over the reals. For a function f satisfying $f(n) < n$, we define the function $f^{(i)}$ recursively for nonnegative integers i by

$$f^{(i)}(n) = \begin{cases} f(f^{(i-1)}(n)) & \text{if } i > 0, \\ n & \text{if } i = 0. \end{cases}$$

For a given constant $c > 0 \in \mathbf{R}$, we define the iterated function f_c^* by
$f_c^*(n) = \min\{i \geq 0 : f^{(i)}(n) \leq c\}$,
which need not be well-defined in all cases. In other words, the quantity $f_c^*(n)$ is the number of iterated applications of the function f required to reduce its argument down to c or less.

For each of the following functions $f(n)$ and constants $c > 0$, give as tight a bound as possible on $f_c^*(n)$.

	$f(n)$	c	$f_c^*(n)$
a.	$\lg n$	1	
b.	$n-1$	1	
c.	$n/2$	1	
d.	$n/2$	2	
e.	$\sqrt{n}$	2	

Solution

	$f(n)$	c	$f_c^*(n)$	Hint
a.	$\lg n$	1	$\lg^* n$	The number of times lg can be applied to a number is defined as $\lg^*$
b.	$n-1$	1	$n-1$	
c.	$n/2$	1	$\lg n$	The number of times a number can be divided by 2 is lg of the number
d.	$n/2$	2	$\lg n - 1$	From the above, this is evident
e.	$\sqrt{n}$	2	$\lg(\lg n)$	The sequence of numbers will be $n, n^{\frac{1}{2}}, n^{\frac{1}{4}}, \ldots, n^{\frac{1}{2^k}}$. So, finally after k iterations, $n^{\frac{1}{2^k}} = 2$. From which we can see the value of $k = \lg(\lg n)$

Problem 2 (10 Points)

From the Mergesort algorithm explained in the class, Explain that the recurrence relation for the time cost of the algorithm is

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

Prove that the cost is $O(n \lg n)$.

Instead of dividing the array into two parts, Would it be of any advantage if it is divided into three parts? What changes will have to be done in merging? Write down the recurrence relation.

Solution

In the Mergesort algorithm explained in the class, the main operations done are

1. Doing Mergesort on the left half of the array
2. Doing Mergesort on the right half of the array
3. Merging the sorted left and right halves

If we represent the cost of Mergesort for an input of size n by $T(n)$. The cost of doing Mergesort on both halves will be $T(\frac{n}{2})$ each. In merging two arrays of size $\frac{n}{2}$, the cost is limited by n comparisons. Hence the cost of doing Mergesort on an input of size n is

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

Lets try to expand the cost function

$$\begin{aligned}
T(n) &= 2 \cdot T(\frac{n}{2}) + n \\
&= 2 \cdot (2 \cdot T(\frac{n}{4}) + \frac{n}{2}) + n \\
&\vdots \\
&= 2 \cdot (2 \cdot (\cdots 2 \cdot (T(1) + \frac{n}{k}) + \cdots) + \frac{n}{2}) + n \\
&\quad \text{Every denominator of } n \text{ will be cancelled out by} \\
&\quad \text{the number or 2s outside the paranthesis} \\
&= n + n + \cdots + n \text{ ; } \lg n \text{ times}
\end{aligned}$$

In earlier tutorials we have seen that the number of times we can divide a number n by 2 is $\lg n$ times. So the cost is $O(n \lg n)$.

If we divide the array into three different parts, the algorithm will look as follows

Algorithm *MergeSort*(Array A, n)
(∗ The algorithm for sorting an array using mergesort ∗)
1. $FirstOneThird \leftarrow A[1$ **to** $\frac{n}{3}]$
2. $SecondOneThird \leftarrow A[\frac{n}{3} + 1$ **to** $\frac{2n}{3}]$
3. $ThirdOneThird \leftarrow A[\frac{2n}{3} + 1$ **to** $n]$
4.
5. $MergeSort(FirstOneThird)$
6. $MergeSort(SecondOneThird)$
7. $MergeSort(ThirdOneThird)$
8. $A \leftarrow Merge(FirstOneThird, SecondOneThird, ThirdOneThird)$
9. **return**

In the Merge algorithm, now we have three sorted arrays to merge. The naive method is to merge two of them first and then merge the third one to the result. Or, all the three of them can be merged together as follows

Algorithm *Merge*(Array A1, Array A2, Array A3)
(∗ The algorithm for merging 3 sorted arrays. A_i has n_i elements ∗)
1. Initialize Array A.
2. $i \leftarrow 1$
3. $j \leftarrow 1$
4. $k \leftarrow 1$
5. $k \leftarrow 1$
6. **while** $i \leq n_1$ and $j \leq n_2$ and $k \leq n_3$
7. **if** $A_1[i] < A_2[j]$
8. **if** $A_1[i] < A_3[k]$
9. $A[l] \leftarrow A_1[i]$
10. $i \leftarrow i + 1$
11. **else**
12. $A[l] \leftarrow A_3[k]$
13. $k \leftarrow k + 1$
14. **else**
15. **if** $A_2[i] < A_3[k]$
16. $A[l] \leftarrow A_2[j]$
17. $j \leftarrow j + 1$
18. **else**
19. $A[l] \leftarrow A_3[k]$
20. $k \leftarrow k + 1$
21. $l \leftarrow l + 1$
22.

(∗ Now there are only two arrays remaining ∗)
(∗ Merge them together to the end of A as in the normal merging ∗)
23. **return** A

Reccurance Relation: In the Mergesort algorithm explained here, the main operations done are

1. Doing Mergesort on the FirstOneThird of the array
2. Doing Mergesort on the SecondOneThird of the array
3. Doing Mergesort on the ThirdOneThird of the array
4. Merging the sorted arrays

From the merge algorithm, we can see that the number of comparisons is in the order of n. So the relation could be written as follows.

$$T(n) = 3 \cdot T(\frac{n}{3}) + O(n)$$

Problem 3 (10 Points)

The Mergesort is an example for a paradigm called *Divide and Conquer*. The method is to divide the problem into smaller problems, solve them separately and then join the solutions.

Give another example for a *Divide and Conquer* algorithm.

Solution

The *divide-and-conquer* strategy solves a problem by:

1. Breaking it into subproblems that are themselves smaller instances of the same type of problem
2. Recursively solving these subproblems
3. Appropriately combining their answers

Example: Binary search

The ultimate divide-and-conquer algorithm is, of course, binary search: to find a key k in a large file containing keys $A[1, \ldots, n]$ in sorted order, we first compare k with $A[\frac{n}{2}]$, and

depending on the result we recurse either on the first half of the file, $A[1, \ldots, \frac{n}{2}]$, or on the second half, $A[\frac{n}{2} + 1, \ldots, n]$. The recurrence now is $T(n) = T(\frac{n}{2}) + O(1)$.

> Given amble time, only 10% of professional programmers were able to get this small program right. In the history in section 6.2.1 of his SORTING AND SEARCHING, Knuth points out that while the first binary search was published in 1946, the first published binary search without bugs did not appear until 1962 – *Shamelessly copied from Column 4 of Programming Pearls (Jon Bently)*

EXAMPLE: MULTIPLICATION

This is something which we do daily in our lives. Without taking a paper and pen, calculate the value of 82×76?

Can we do it as $82\times76 = 82\times(75+1) = (80\times75)+(2\times75)+(82\times1) = 6000+150+82 = 6232$

What we did was using the formula $(a + b) \cdot (c + d) = ac + ad + bc + bd$

Problem 4 (10 Points)

Using induction, develop an algorithm that finds the second largest number in a set of n (pairwise different) natural numbers.

Solution

Algorithm *FindSecondLargest*(Array A, n)
(∗ The algorithm for finding out the second largest number in the set of given numbers ∗)

```
1.   if n < 2
2.           There is no second largest number.
3.           return
4.
(∗ Now there are atleast 2 numbers ∗)
5.   if A[1] < A[2]
6.           Largest ← A[2]
7.           SecondLargest ← A[1]
8.      else
9.           Largest ← A[1]
10.          SecondLargest ← A[2]
11.
(∗ If there are only 2 numbers, we can give the result now itself ∗)
12.  if n = 2
13.          Result ← SecondLargest
14.          return
```

```
15.
(* More than 2 numbers, We have to compare with all the other numbers *)
16.  for i ← 3 to n
17.          if A[i] > SecondLargest
18.                  if A[i] > Largest
19.                          SecondLargest ← Largest
20.                          Largest ← A[i]
21.                    else
22.                          SecondLargest ← A[i]
23.
(* SecondLargest is the result value *)
24.  Result = SecondLargest
25.  return
```

The complexity of the algorithm is decided by the for loop at line 16. The loop will be executed $n-2$ times and the algorithm is $O(n)$.

Extra

This was already taken in the class. So this could be avoided. We will do this if time permits.

Problem 5 (10 Points)

Analyze the total number of operations (not only key-comparisons) executed when InsertionSort is applied to an input consisting of n keys. Write this resulting complexity in Landau notation and compare it to the number of key-comparisons analyzed in the lecture.

Solution

Algorithm *InsertionSort*(Array A, n)
(* Algorithm to sort n numbers using InsertionSort *)

```
1.  for i ← 2 to n
2.          j = FindPos(A[], i)
3.          tmp = A[i]
4.          Shift all elements of the array from A[j] to A[i − 1] to one position right.
5.          A[j] = tmp
```

The for loop will be executed $n-1$ times. If we assign cost to each of the operations inside the loop, Line 2 has $\lg i$ number of operations inside $FindPos$, Line 3 has a constant cost c_1, Line 4 has the cost of $i-j$ shifts and finally line 5 has again constant cost c_2. Here, for ease of calculation we can use the cost $i-1$ instead of $i-j$ for line 4.

So the total cost can be calculated as: $\sum_{i=2}^{n} \lg i + (n-1) \cdot c_1 + \sum_{i=2}^{n} (i-1) + (n-1) \cdot c_2$ which could be simplified to get a polynomial of the order $O(n^2)$.

Hence, the total operations of InsertionSort is $O(n^2)$ where as the total number of comparisons is $O(n \lg n)$.