CS21003 Algorithms I, Autumn 2012–13

Class test 1

Maximum marks: 20 | Time: 11-Sep-2012 | Duration: 1 hour

Roll no: ________________ **Name:** ________________________________

[*Write your answers in the question paper itself. Be brief and precise. Answer all questions.*]

1. The Fibonacci numbers F_n, $n \geqslant 0$, are defined as $F_0 = 0$, $F_1 = 1$, and $F_n = F_{n-1} + F_{n-2}$ for $n \geqslant 2$. In order to compute F_n, we initialize each entry of an array $F[0 \ldots n]$ to -1. Then, we call a recursive function which, upon input m, first checks whether the array location $F[m]$ stores -1. If so, it recursively computes F_m, and stores this value in $F[m]$. Otherwise, the function immediately returns.

```
int Fib ( int m, int *F )
{
   if (F[m] == -1)
      if (m <= 1) F[m] = m; else F[m] = Fib(m-1,F) + Fib(m-2,F);
   return F[m];
}

/* Inside main() */
for (i=0; i<=n; ++i) F[i] = -1;
printf("F_%d = %d\n", n, Fib(n,F));
```

What is the running time of the call **Fib(n,F)** in **main()**? Justify. **(10)**

Solution Let us look at what happens in the call stack, and the changes in the $F[\,]$ array. **Fib(n)** calls **Fib(n-1)**, **Fib(n-1)** calls **Fib(n-2)**, $\ldots$, **Fib(2)** calls **Fib(1)**. The call **Fib(1)** sets $F[1] = 1$, and returns to the call of **Fib(2)**. **Fib(2)** then makes the second recursive call **Fib(0)** which sets $F_0 = 0$ and returns again to the call of **Fib(2)**. After both the recursive calls return, **Fib(2)** adds $F[0]$ and $F[1]$ (the two return values), saves this sum in $F[2]$, and returns to the call **Fib(3)**. When **Fib(3)** makes the second recursive call **Fib(1)**, the value of $F[1]$ is already computed, so this value is returned without making any more recursive calls. Proceeding in this way, each call **Fib(i)** makes a second recursive call **Fib(i-2)** which sees the array element $F[i-2]$ already computed, so this value is straightaway returned to **Fib(i+1)**.

It follows that the outermost call makes a total of $2n$ further recursive calls of **Fib()**. Out of these, only n calls set the elements in $F[\,]$, and the remaining n calls return these values. Finally, the outermost call sets $F[n]$, and returns to **main()**. Therefore, the running time of **Fib(n,F)** in **main()** is $\Theta(n)$.

2. Ms. Rotunda is making a long train journey. She can stay without food for four hours. The train does not have a pantry car, so Ms. Rotunda can eat only when the train stops at stations. Given the complete timetable for the train, design an efficient algorithm to identify the stations where Ms. Rotunda would eat so that she never feels hungry throughout the journey, and the number of meals is as small as possible. Assume that she takes her first meal just before the train leaves its source station. Assume also that the train halts at least once in any period of four hours (otherwise, there is no solution to Ms. Rotunda's problem). Neglect the halting times of the train at stations. Prove the correctness of your algorithm, and deduce its running time. **(10)**

Solution **The algorithm:** Let $S_0, S_1, S_2, \ldots, S_n$ be the stations where the train stops (in that sequence), where S_0 is the source, and S_n the destination. The times t_i (in hours) for the train to go from Station S_{i-1} to Station S_i are also given for $i = 1, 2, \ldots, n$. Neglecting halting times at stations, the time taken by the train to travel from station S_i to S_j (with $j \geqslant i$) is then $t_{i+1} + t_{i+2} + \cdots + t_j$. The following greedy algorithm solves Ms. Rotunda's minimization problem.

Print "*Take meal at Station* 0".
Set $lastmeal = 0$, $i = 0$, and $fasttime = 0$.
While $(i \leqslant n)$ {
    Set $fasttime = fasttime + t_{i+1}$.
    If $(fasttime > 4)$ {
        Print "*Take meal at Station* i".
        Set $lastmeal = i$ and $fasttime = 0$.
    } else {
        Increment i by 1.
    }
}

Running time: Under the assumption that each $t_i \leqslant 4$, the loop of the above program runs for at most $2n$ times. Each iteration of the loop takes constant time. So the running time of this greedy algorithm is $\Theta(n)$.

Correctness: Let $0, i_1, i_2, \ldots, i_k$ be an optimal solution to Ms. Rotunda's problem, whereas $0, j_1, j_2, \ldots, j_l$ be the solution produced by the greedy algorithm. Clearly, $i_1 \leqslant j_1$, so $0, j_1, i_2, i_3, \ldots, i_k$ continues to remain an optimal solution. We must have $i_2 \leqslant j_2$, so $0, j_1, j_2, i_3, i_4, \ldots, i_k$ is again an optimal solution. Proceeding in this way, we can convert the optimal solution to the greedy solution without increasing the number of meals. Thus, $k \geqslant l$. But since $0, i_1, i_2, \ldots, i_k$ is an optimal solution, we must have $k \leqslant l$. Therefore, $k = l$, that is, the greedy solution too is optimal.