Albert-Ludwigs-Universität, Inst. für Informatik
Prof. Dr. Fabian Kuhn
H. Ghodselahi, O. Saukh, M. Ahmadi November 2, 2015

Algorithm Theory, Winter Term 2015/16
Problem Set 1 - Sample Solution

Exercise 1: Complexity (2+2 points)

Characterize the relationship between $f(n)$ and $g(n)$ in the following two examples by using the O, Θ, or Ω notation. Hence, state if $f(n) = \Theta(g(n))$ and otherwise if $f(n) = O(g(n))$ or $f(n) = \Omega(g(n))$. Explain your answers (formally)!

a) $f(n) = n^\epsilon$ (for any positive constant $\epsilon < \frac{1}{2}$) $g(n) = \log n$

b) $f(n) = \log n!$ $g(n) = n \log n$

Solution:

a) $f(n) \in \Omega(g(n))$.

Let us assume that the base of logarithm in $g(n)$ is some positive constant c. Moreover, consider some positive constant $c' > 1$. Then,

$$\lim_{n\to\infty} \frac{\log_c n}{n^\epsilon} = \lim_{n\to\infty} \frac{\frac{\log_{c'} n}{\log_{c'} c}}{(c'^\epsilon)^{\log_{c'} n}} = \lim_{n\to\infty} \frac{\log_{c'} n}{\log_{c'} c \cdot (c'^\epsilon)^{\log_{c'} n}} \overset{(1)}{=} 0.$$

(1) c'^ϵ is a constant greater than 1.

From the above inequality, it follows that $f(n) \notin O((g(n))$. Therefore, we have $f(n) \notin \Theta(g(n))$.

Note: Different logarithm base just leads to a constant difference in growth speed.

b) $f(n) \in \Theta(g(n))$.

1) $f(n) \in O(g(n))$

$f(n) = \log n! = \log(n \cdot (n-1) \cdot (n-2) \dots 1) = \log n + \log(n-1) + \dots + \log 1$

$g(n) = n \log n = \underbrace{\log n + \log n + \dots + \log n}_{n \text{ times}}$

$\Rightarrow f(n) \leq g(n)$ for any $n > 1$. Therefore, based on the formal definition of the big O notation, by taking $c = 1$ and $n_0 = 1$ we have $f(n) \in O(g(n))$.

2) $f(n) \in \Omega(g(n))$

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \lim_{n\to\infty} \frac{\sum_{i=1}^{n} \log(i)}{n \log n} \geq \lim_{n\to\infty} \frac{\sum_{i=n/2}^{n} \log(i)}{n \log n} \geq \lim_{n\to\infty} \frac{\frac{n}{2} \cdot \log(n/2)}{n \log n} = \frac{1}{2} > 0$$

From (1) and (2), we can conclude that $f(n) \in \Theta(n)$.

Exercise 2: Recurrence Relations (2+2 points)

a) Guess the solution of the following recurrence relation by repeated substitution.

$$T(n) \leq 2 \cdot T(\frac{n}{2}) + c \cdot n \log_2 n, \quad T(1) \leq c$$

where $c > 0$ is a constant.

b) Use induction to show that your guess is correct.

Solution:

a) First, by using substitution, we achieve a guess for the answer of the inequality.

$$\begin{aligned}
T(n) &\leq 2T(n/2) + cn \log n \\
&\leq 2\big(2T(n/4) + cn/2(\log n - 1)\big) + cn \log n \\
&\leq 4T(n/4) + 2cn \log n - cn \\
&\leq 4\big(2T(n/8)cn/4(\log n - 2)\big) + 2cn \log n - cn \\
&\leq 8T(n/8) + 3cn \log n - 3cn \\
&\vdots \\
&\leq 2^i T(n/2^i) + icn \log n - \sum_{j=1}^{i-1} j \cdot cn
\end{aligned}$$

By considering $i = \log n$,

$$T(n) \leq cn + \frac{1}{2}cn \log^2 n + \frac{1}{2}cn \log n$$

b) Here we use induction to prove our guess achieved by induction.

Induction base: $T(1) \leq c(1) + \frac{1}{2}c(1)\log^2 1 + \frac{1}{2}c(1)\log 1 = c \leq c$ ✓

Induction hypothesis: $\forall n' < n : T(n') \leq cn' + \frac{1}{2}cn' \log^2 n' + \frac{1}{2}cn' \log n'$

Induction step:

$$\begin{aligned}
T(n) &\leq 2T(n/2) + cn \log n \\
&\leq 2\left(c\frac{n}{2} + \frac{1}{2}c\frac{n}{2}\log^2\frac{n}{2} + \frac{1}{2}c\frac{n}{2}\log(\frac{n}{2})\right) + cn \log n \\
&\leq cn + \frac{1}{2}cn \log^2 n + \frac{1}{2}cn \log n \ \checkmark
\end{aligned}$$

From the induction we have $T(n) \leq cn + \frac{1}{2}cn \log^2 n + \frac{1}{2}cn \log n$, for all $n \geq 1$.

Exercise 3: Maximum Sum Subsequence (4 points)

Given an integer array $A = \{x_0, x_1, \ldots, x_{n-1}\}$ where $x_i \in \mathbb{Z}$ for all $i \in \{0, 1, \ldots, n-1\}$ (note that the values x_i can be negative).

Devise an efficient **divide-and-conquer** algorithm to find a contiguous subsequence of elements in A with maximum possible sum. That is, you need to find indices $0 \leq i_1 \leq i_2 \leq n-1$ such that the sum $\sum_{i=i_1}^{i_2} x_i$ is maximized. Write down the recurrence relation which describes the running time of your algorithm and use the Master theorem to derive the running time of your algorithm.

Hint: *There is a divide-and-conquer solution which runs in time $O(n)$, an $O(n \log n)$ solution gives partial points. In order to find an $O(n)$ algorithm, it might help to first design an $O(n \log n)$ algorithm.*

Solution:

For simplicity let us assume that n is power of 2.

- **Divide step:** In this step we simply divide the given array X into two equal-sized arrays X_l and X_r. This step can be done in constant time.
- **Conquer step:** In this step we solve the problem recursively for each of the two equal-sized subarrays.
  In solving the problem for each subarray, we find the following three subsequences:
  - S - The contiguous subsequence with maximum sum (*green* in Figure 1).
  - L - The contiguous subsequence with maximum sum which starts from the first element of the subarray (*orange* in Figure 1).
  - R - The contiguous subsequence with maximum sum which ends on the last element in the subarray (*red* in Figure 1).

  Moreover, for subarrays of size one consisting of one element x, we apparently have $S = L = R = \{x\}$.
- **Combine step:** In this step we intend to use the solutions for the two subarrays X_l and X_r and obtain the solution for the array consisting these two subarrays.
  Let us assume that the solution for the left subarray is $\{S_l, L_l, R_l\}$ and for the right subarray is $\{S_r, L_r, R_r\}$. Now we would like to obtain $\{S, L, R\}$ for the array consisting these two subarrays.

  We can simply find L and R. L is either L_l or the concatenation of X_l and L_r (the one with maximum sum). Similarly, R is either R_r or the concatenation of R_l and X_r.

  The only thing which is left is to find S. S is one of the following three possibilities that has the maximum sum.

  a) S_l
  b) S_r
  c) The concatenation of R_l and L_r

  The combine step only needs a constant number of comparisons. Therefore, both the divide and combine steps can be done in constant time.

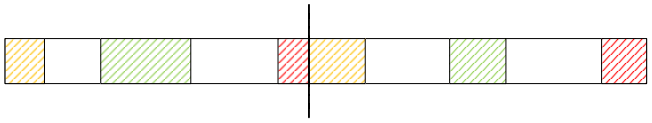

Figure 1: subarrays that need to be obtained in every recursion step.

The following pseudo code explains the algorithm in more details:

```
// Elements in structure type to return.
struct Solution {
int L_l, L_r; // start and end indices of L.
int R_l, R_r; // start and end indices of R.
int S_r, S_l; // start and end indices of S.
int A_l, A_r; // start and end indices of A.
int S_sum, L_sum, R_sum, A_sum; // sums of elements
}
```

Algorithm 1: MaxSumSubsequence(array A, length of array n)

```
Solution sol = getSolution(A)
return (start, end, sum)
```

Algorithm 2: DCMaxSum(A, A_l, A_r)

Solution loc;
if $A_l - A_r == 1$ **then**
  $loc.L_l = A_l; loc.L_r = A_r; loc.L_s = A[A_l];$
  $loc.R_l = A_l; loc.R_r = A_r; loc.R_S = A[A_l];$
  $loc.S_l = A_l; loc.S_r = A_r; loc.S_s = A[A_l];$
  $loc.A_l = A_l; loc.A_r = A_r; loc.A_s = A[A_l];$
  return loc;
else
  Solution $left$, $right$;
  $left = DCMaxSum(A, A_l, (A_r - A_l)/2);$
  $right = DCMaxSum(A, (A_r - A_l)/2, A_r);$
  $loc.L_l = left.L_l;$
  $loc.R_r = right.R_r;$
  $loc.A_l = left.A_l; loc.A_r = right.A_r; loc.A_{sum} = left.A_{sum} + right.A_{sum};$
  if $left.L_{sum} > left.A_{sum} + right.L_{sum}$ **then**
    $loc.L_r = left.L_r; loc.L_{sum} = left.L_{sum}$
  else
    $loc.L_r = right.L_r; loc.L_{sum} = left.A_{sum} + right.L_{sum};$
  if $right.R_{sum} > right.A_{sum} + left.R_{sum}$ **then**
    $loc.R_l = right.R_l; loc.R_{sum} = right.R_{sum};$
  else
    $loc.R_l = left.R_l; loc.R_{sum} = right.A_{sum} + left.R_{sum};$
return *sol*

Analysis: We can conclude that in each step, the given subarray is divided into two equal-sized subarrays in constant time and the two subarrays can be solved recursively. In addition, the combine step also takes constant time to be done. If we assume that divide and conquer are done in constant time c, we have the following recurrence relation as running time of the algorithm:

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

It is also clear from above that solving the problem instance of size 1 needs a constant amount of time, so it is concluded that $T(1) \in O(1)$. Using the master theorem it can be concluded that the running time is $O(n)$.