Algorithms & Complexity: Assignments 1

Sander van Veen & Taddeüs Kroes
6167969 & 6054129
sandervv@gmail.com & taddeuskroes@hotmail.com

September 24, 2010

Contents

1 Assignments part I **2**

1 Assignments part I

Note: Even though the assignments are written in Dutch, we have written the answers in English to improve our language skills.

Assignment 1

(a) Given an alphabet $\Sigma = \{0, 1\}$ and array A of symbols (each $a \in \Sigma$). An algorithm to sort the array A with a time complexity of $\mathcal{O}(n)$ is described in the following steps:

1. Create an empty array B.
2. If symbol $a = 0$, put a at the beginning of array B.
3. If symbol $a = 1$, put a at the end of array B.
4. If there are no symbols left, return array B.

(b) Ideal behavior for a sort algorithm is $\mathcal{O}(n)$, but this is not possible in the average case. A sort algorithm requires searching in array B (to determine the position to insert) for each symbol in A. If searching through the array is in order of $\mathcal{O}(log\ n)$, the sort algorithm has a time complexity of $\mathcal{O}(n\ log\ n)$.

The "search algorithm" above is in order of $\mathcal{O}(1)$ (value of symbol s determines its position to insert). Therefore is the sorting able to complete in a time complexity of $\mathcal{O}(n)$.

Assignment 2

(a) If elements with a higher frequency are put at the beginning of the list, the search operation stops earlier, when it's looking for an element with a high frequency. Therefore, less comparisons are required with a descending frequency sorted list.

(b) The total number of "good searches" is independent of the used storing technique, since a "good search" occurs for every element in list s. It is impossible to have a different number of "good searches", because that would indicate that one or more elements of s are not stored in list l.

(c) Given $l = \{A, B\}$ and s is an list of search operations, containing m times A and n times B.

The theorem states that the total number of false comparisons is not larger than $min(m, n)$, when the optimum storage technique is used.

For example, use $s = \{A, B, A, A, B, A, B, A\}$. This will result in three false comparisons (one for each B). If $s = \{A, A, A, A, A, B, B, B\}$ (same m and n, but different order), only one false comparison (for the first B) occurs. Given $l = \{A, B\}$, the optimum storage technique will fail for the element with the lowest frequency.

(d) Given $l = \{A, B\}$ and s is an list of search operations, containing m times A and n times B.

For example, use $s = \{B, A, B, A, B, A, B, A\}$. When MFT is used as storage technique, l changes as follows:

l_{before}	search	l_{after}	false comparisons
{A,B}	$s_i = B$	{B,A}	1
{B,A}	$s_i = A$	{A,B}	2
{A,B}	$s_i = B$	{B,A}	3
{B,A}	$s_i = A$	{A,B}	4
{A,B}	$s_i = B$	{B,A}	5
{B,A}	$s_i = A$	{A,B}	6
{A,B}	$s_i = B$	{B,A}	7
{B,A}	$s_i = A$	{A,B}	8

The optimum storage technique requires at most 4 $(= min(4, 4))$ false comparisons. The example given above is the worst case for MFT, since every search operation results in one false comparison (and the element is found after the first comparison). A total of eight false comparison occured, which is two times the maximum of the optimum storage technique.

(e) MFT uses the properties of pairwise independence to 'predict' how many times an element will be searched. In the answer to questions c and d we see that the time complexity of MFT is at most twice as expensive as when we already know which element is searched the most.

Assignment 3

(a) Bubble sort. The bubble sort algorithm has a worst-case time complexity of $n(n-1) = n^2 - n$. Since $n^2 - n \le n^2$ for $n \ge 0$, bubble sort is in the order of $o(n^2)$.

(b) Bubble sort. In the best case scenario, the array is already sorted. In that case, the algorithm stops when it concludes that no swaps were done after $n - 1$ comparisons, which is in the order of $\Omega(n)$.

(c) Selection sort. Finding the minimum value in the part of an array from k to n costs $n - k$ comparisons, which is $\mathcal{O}(n)$. Since k runs from 0 to n, the total complexity is in order of both $\Omega(n * n) = \Omega(n^2)$ and $\mathcal{O}(n^2)$.

(d) We are looking for an algorithm with a higher growth rate than $c * 2^n$. For example, when determining the shortest path between two points we could try all permuations and see which is the cheapest, which costs $n!$ iterations ($n! > 2^n$ for $n \to \infty$).

(e) Binary search tree. Searching a leaf in a balanced binary tree always requires $\log n$ comparisons, never less and never more. So in this case $c_1 = c_2$.

(f) Sorting an array with n elements. This always requires n iterations, so it is in the order of $\sim n$.

Assignment 4

When $k > 0$, both $log^k(n)$ grows slower than $c * n^{1/k}$, so $log^k(n) \in \mathcal{O}(n^{1/k})$.
When $k = 0$, $n^(1/k)$ is undefined.
When $k < 0$, $c * n^{1/k}$ is descending so at a certain point it will stay under $log^k(n)$.
Conclusion: $log^k(n) \in \mathcal{O}(n^{1/k})$ for $k > 0$.

Assignment 5

Given $f \in \mathcal{O}(g)$, f and g have the same growth rate ($f' = c * g'$). This means that $\frac{f}{g} = c$, which is the same as $\frac{f}{g} \in \mathcal{O}(1)$.