Answers
Exercises week 2

Complexity 2011-2012

A. Silva, H. Barendregt, B. Westerbaan & B. Westerbaan

Exercise 1. Assume you have functions f and g such that $f(n)$ is in $O(g(n))$. For each of the following statements, decide whether you think it is true or false and give a proof or a counter-example.

1. $\log_2 f(n)$ is $O(\log_2 f(n))$
2. $2^{f(n)}$ is $O(2^{g(n)})$
3. $f(n)^2$ is $O(g(n)^2)$

Answers

1. By assumption there exist $N \in \mathbb{N}$ and $c \in \mathbb{R}_{>0}$ such that for all $n \in \mathbb{N}$ with $n \geq N$ we have
$$0 \leq f(n) \leq cg(n).$$
But then, since $\log_2$ is order-preserving:
$$\begin{aligned}\log_2 f(n) &\leq \log_2 cg(n) \\ &= \log_2 c + \log_2 g(n).\end{aligned}$$
That looks almost OK. We want to find a $d \in \mathbb{R}_{>0}$ such that $\log_2 f(n) \leq d \log_2 g(n)$. Using the previous, it is sufficient to show:
$$\log_2 c + \log_2 g(n) \leq d \log_2 g(n).$$
And this is OK, if:
$$\frac{\log_2 c}{\log_2 g(n)} + 1 \leq d.$$
However, $\log_2 g(n)$ might get closer and closer to 0 while n gets bigger. This leads us to the following counterexample:
$$2(1 + \frac{1}{n}) \in O(1 + \frac{1}{n}),$$
but
$$\log_2 2 + \log_2(1 + \frac{1}{n}) \notin O(\log_2(1 + \frac{1}{n})).$$

2. We have $2n \in O(n)$, however we saw last week that $2^{2n} \notin O(2^n)$.

3. By assumption there exist $N \in \mathbb{N}$ and $c \in \mathbb{R}_{>0}$ such that for all $n \in \mathbb{N}$ with $n \geq N$ we have

$$0 \leq f(n) \leq cg(n).$$

But then, since squaring is order-preserving (on positive values), also:

$$\begin{aligned} 0 = 0^2 \leq f(n)^2 &\leq (cg(n))^2 \\ &= c^2 g(n)^2. \end{aligned}$$

Thus $f^2 \in O(g^2)$.

Exercise 2. Given a sequence v of n integers and an integer x, we want to obtain the index of the first occurrence of x in v, and -1 in case x does not occur in v. The function `search` below solves this problem.

```
int search(int v[], int n, int x) {
       int i = 0, found = 0;
       while (i < n && !found) {
        if (v[i] == x) found = 1;
        i++;
       }
       if (!found) return -1;
       else return i-1;
 }
```

(i) What will be the number of operations performed in the worst and best cases.

(ii) If the sequence v is ordered, then it is possible to improve the algorithm. Explain how and determine the complexity of the new algorithm.

Answers Let us start with labeling the function above:

```
int search(int v[], int n, int x) {
       int i = 0, found = 0;               c1     1
       while (i < n && !found) {           c2     V
        if (v[i] == x)                     c3     V-1
            found = 1;                     c4     W
        i++;                               c5     V-1
       }
       if (!found)                         c6     1
             return -1;                    c7     1
       else return i-1;                    c8
 }
```

(i) The worst case is when the element x does not occur in v, which means the while loop will run $V = n + 1$ times (The $+1$ here is because the test of the loop needs to be

executed one last time to know that its done). The test in the if will never be true and hence $W = 0$. The total number of operations will then be

$$1 + V + V - 1 + W + V - 1 + 1 + 1 = 1 + 3V = 1 + 3(n + 1) = 4 + 3n \text{ operations}$$

The best case scenario occurs when the element x is found in the first position, which means the while loop will run 1 time only (the second time there will only be a test, $V = 2$). We also have $W = 1$. Hence, we have:

$$1 + 2 + 1 + 1 + 1 + 1 + 1 = 8 \text{ operations}$$

(ii) If the array is sorted we can do a *dictionary*-like search: start in the middle, if x is smaller than the element in the middle search right, otherwise search left (using the same dictionary-like process). The complexity is then $O(\lg n)$.

Exercise 3. (*) Consider the `Bubble sort` algorithm, where N is the size of the input vector:

```
void bubble_sort(int A[], int N) {
  for (i=0 ; i<N ; i++)
    for(j=N-1 ; j>i ; j--)
      if (A[j] < A[j-1])
         swap(A,j,j-1);
}


void swap(int A[], int i, int j) {
  int aux=A[i];
  A[i]=A[j];
  A[j]=aux;
}
```

(i) Study the behavior of the algorithm above in the best and worst case scenarios using the notation Θ.

(ii) How would you characterize the global behavior of this algorithm using the notation O and Ω? What is the relation with the answer you gave in (i)?

(iii) Imagine the algorithm checks, in the outermost cycle, whether `A` is sorted. Repeat (i) under this assumption.

(iv) Analyze the following alternative version of the algorithm

```
void bubble_sort(int A[], int N) {
      for (i=0 ; i<N ; i++)
         for(j=N-1 ; j>i ; j--)
            if (A[j] < A[i])
             swap(A,j,i);
}
```

Answers

(i) We again start by labeling the function above:

```
void bubble_sort(int A[], int N) {
  for (i=0 ; i<N ; i++)                c1   N+1
    for(j=N-1 ; j>i ; j--)             c2   Si
      if (A[j] < A[j-1]                c3   Sj
         swap(A,j,j-1);                c4   Sj
}
```

The sums S_i and S_j are

$$S_i = \sum_{i=0}^{N-1}(N-i+1) = \frac{N}{2}((N+1)+2) = \frac{N^2+3N}{2}$$
$$S_j = \sum_{i=0}^{N-1}(N-1)-i-1 = \frac{N}{2}((N-2)+(-1)) = \frac{N^2-3N}{2}$$

The worst case occurs when the list is ordered backwards, which has as consequence that the test in the if always succeeds and the swap always occurs. Then we have

$$T(n) = Nc_1 + S_i c_2 + S_j(c_3 + c_4).$$

Because S_j and S_i have a N^2 factor, we can conclude that bubble sort executes in the worst case in $\Theta(N^2)$.

The best case is when the vector is sorted and the swap never executes. We then have

$$T(n) = Nc_1 + S_i c_2 + S_j c_3.$$

but unfortunately the complexity remains quadratic, for the same reason as before. $T(n)$ executes in the best case also in $\Theta(n^2)$.

(ii) The algorithm is $\Omega(n^2)$ and $O(n^2)$ which is according to the answers in (i) since $\Theta = O \cap \Omega$.

(iii) The worst case remains the same. The best case changes: we need to check whether the array is sorted (that takes linear time) and then return it. Hence, in the best case we improve from quadratic to linear.

(iv) Looking closely you can see that the number of operations remains the same as in (i) so the answer is the same.

Exercise 4. (†)

Given three arrays of n (floating point) real numbers (the numbers can be positive or negative), write an algorithm to determine if there are three numbers, one from each array whose sum is 0. Design the most efficient algorithm you can think of to solve this problem. (It can be done in $\Theta(n^2)$ time.)

Hint: Given two sorted lists determine an algorithm, in $\Theta(N)$, to check if they have an element in common.