Warm-up

Problem 1. Sort the following functions in increasing order of asymptotic growth

$$n, n^3, n\log n, n^n, \frac{3^n}{n^2}, n!, \sqrt{n}, 2^n$$

Solution 1.

$$\sqrt{n}, n, n\log n, n^3, 2^n, \frac{3^n}{n^2}, n!, n^n$$

Problem 2. Sort the following functions in increasing order of asymptotic growth

$$\log\log n, \log n!, 2^{\log\log n}, n^{\frac{1}{\log n}}$$

Solution 2.

$$n^{\frac{1}{\log n}}, \log\log n, 2^{\log\log n}, \log n!$$

Problem 3. Consider the following pseudo-code fragment.

```
def stars(A):
  for i in [1:n]:
    print '*' i many times
```

a) Using the O-notation, upperbound the running time of STARS.

b) Using the Ω-notation, lowerbound the running time of STARS to show that your upperbound is in fact asymptotically tight.

Solution 3.

a) The first iteration prints 1 star, second prints two, third prints three and so on. The total number of stars is $1 + 2 + \cdots + n$, namely,

$$\sum_{j=1}^{n} j \le \sum_{j=1}^{n} n = n^2 = O(n^2).$$

b) Assume for simplicity that n is even. We lowerbound the number of stars printed during iterations $\frac{n}{2}$ through n:

$$\sum_{j=1}^{n} j \geq \sum_{j=n/2}^{n} \frac{n}{2} = \frac{n^2}{4} = \Omega(n^2).$$

Problem 4. Recall the problem we covered in lecture: Given an array A with n entries, find $0 \leq i < j < n$ maximizing $A[i] + \cdots + A[j]$.

Prove that the following algorithm is incorrect: Compute the array S as described in the lectures. Find i minimizing $S[i]$, find j maximizing $S[j+1]$, return (i, j).

Come up with the smallest example possible where the proposed algorithm fails.

Solution 4. Let $A = [1, -2]$, so $S = [0, 1, -1]$. The algorithm return $i = 2$ and $j = 0$. Which does not even obey $i < j$.

Problem solving

Problem 5. Given an array A consisting of n integers, we want to compute the upper triangle matrix C where

$$C[i][j] = \frac{A[i] + A[i+1] + \cdots + A[j]}{j - i + 1}$$

for $0 \leq i \leq j < n$. Consider the following algorithm for computing C:

```
def summing_up(A)
  C = new matrix of len(A) by len(A)
  for i in [0:n-1]
    for j in [i:n-1]
      compute average of entries A[i:j]
      store result in C[i, j]
  return C
```

a) Using the O-notation, upperbound the running time of SUMMING-UP.

b) Using the Ω-notation, lowerbound the running time of SUMMING-UP.

Solution 5.

a) The number of iterations is $n + n - 1 + \cdots + 1 = \binom{n}{2}$, which is bounded by n^2. In the iteration corresponding to indices (i, j) we need to scan $j - i + 1$ entries from A, so it takes $O(j - i + 1) = O(n)$. Thus, the overall time is $O(n^3)$.

b) In an implementation of this algorithm, Line 5 would be computed with a for loop; when $i < \frac{1}{4}n$ and $j > \frac{3}{4}n$, this loop would iterate least $n/2$ times, which takes $\Omega(n)$ time. There are $n^2/16$ pairs (i, j) of this kind, which is $\Omega(n^2)$. Thus, the overall time is $\Omega(n^3)$.

Problem 6. Come up with a more efficient algorithm for computing the above matrix $C[i][j] = \frac{A[i]+A[i+1]+\cdots+A[j]}{j-i+1}$ for $0 \le i \le j < n$. Your algorithm should run in $O(n^2)$ time.

Solution 6. The idea is very simple, suppose we had at our disposal an array B whose ith entry is the entries i through $n - 1$ of A; in other words,

$$B[i] = \sum_{k=0}^{i-1} A[k].$$

Then $C[i][j]$ is simply $\frac{B[j+1]-B[i]}{j-i+1}$. If we can compute B in $O(n^2)$ time we are done. In fact, we can compute B is just $O(n)$ time.

```
1 def summing-up-fast(A)
2   B[0] = 0
3   for i in [1:n-1]:
4     B[i] = B[i-1] + A[i-1]
5   for i in [0:n-1]:
6       for j in [i:n-1]
7       C[i][j] = (B[j+1] - B[i]) / (j-i+1)
8   return C
```

The correctness of the algorithm is clear since

$$C[i][j] = \frac{B[j+1] - B[i]}{j - i + 1} = \frac{\sum_{k=0}^{j} A[k] - \sum_{k=0}^{i-1} A[k]}{j - i + 1} = \frac{\sum_{k=i}^{j} A[k]}{j - i + 1}.$$

as desired.

For the time complexity, we note that the for loop in Line 3 runs in $O(n)$ time and the nested for loops starting in Line 5 runs in $O(n^2)$ time, yielding the desired overall complexity.

Problem 7. Give a formal proof of the transitivity of the O-notation. That is, for function f, g, and h show that if $f(n) = O(g(n))$ and $g(n) = O(h(n))$ then $f(n) = O(h(n))$.

Solution 7. Since $f = O(g)$, it follows that there exists $n_0 > 0$ and $c > 0$ such that $f(n) \leq cg(n)$ for all $n > n_0$. Since $g = O(h)$, it follows that there exists $n_0' > 0$ and $c' > 0$ such that $g(n) \leq ch(n)$ for all $n > n_0$.

It follows that for all $n > \max(n_0, n_0')$ we have

$$f(n) \leq c\,g(n) \leq c\,c'\,h(n).$$

Thus, if we define $n_0'' = \max(n_0, n_0')$ and $c'' = c\,c'$, the above inequality means $f = O(h)$.

Problem 8. Given an array with n integer values, we would like to know if there are any duplicates in the array. Design an algorithm for this task and analyze its time complexity.

Solution 8. The straightforward solution is to do a double for loop over the entries of the array returning "found duplicates" right away when we find a pair of identical elements, and "found no duplicates" at the end if we exit the for loops. The complexity of this solution is $O(n^2)$.

A better solution is to sort the elements and then do a linear time scan testing adjacent positions. As we shall see later in class one can sort an array of length n in $O(n \log n)$ time.