University of Freiburg
Dept. of Computer Science
Prof. Dr. F. Kuhn
P. Bamberger, P. Schneider

Algorithms Theory
Sample Solution Exercise Sheet 4

Due: Tuesday, 1st of December 2020, 4 pm

Exercise 1: Number Picking Game *(10 Points)*

Consider an array $A[0..n-1]$ of positive integer numbers. Consider the following game. The goal is to pick numbers from A with a sum as large as possible. In the first turn you can pick *one* number $A[0]$ or $A[n-1]$ from either end of the array $A[0..n-1]$. In the second turn an adversary picks the first or last entry in the *remaining* array (either from $A[1..n-1]$ or $A[0..n-2]$). Generally speaking, each player picks in its turn either the number $A[i]$ or $A[j]$ from the remaining array $A[i..j]$.

(a) Assuming that both players play optimally (and knowing that their respective adversary does as well), give a *recursive function* $w(i,j)$ describing the *winnings* the player who picks next from one end of $A[i..j]$ can still make for the rest of the game. Do not forget the base cases. Explain the correctness of the recursion. *(5 Points)*

(b) We want to compute your *total winnings* provided that you are allowed to pick first and both you and your adversary always play optimally (and both knowing that). Use $w(i,j)$ and the principle of dynamic programming to provide pseudo code that computes your *total winnings* under these circumstances in $O(n^2)$ time. Explain the running time. *(5 Points)*

Sample Solution

(a) We start by observing that, since both players play optimally, the number $w(i,j)$ can be applied symmetrically for both sides. The second observation is that the game is what is called a *zero sum game*. That means that the winnings of one player represents the amount that the other player can *not* gain anymore. So essentially the winnings of one player is the loss of the other player. Formally, if $W_{ij} := \sum_{k=i}^{j} A[k]$ is the maximum amount that can be won from the remaining array $A[i..j]$, then $w(i,j) = \max\{W_{ij} - w(i+1,j), W_{ij} - w(i,j-1)\}$ given that $i < j$ (two choices left). If we have $i = j$, then obviously $w(i,j) = A[i]$ (which represents the base case).

(b) **Algorithm 1** `winnings`(A,i,j,W) ▷ assume $W = W_{ij}$ and global dict `memo` initialized to `Null`

```
if $i = j$ then return $A[i]$                                   ▷ base case
if memo[$i,j$] $\neq$ Null then return memo[$i,j$]
memo[$i,j$] $\leftarrow \max\{W -$ winnings($A, i+1, j, W - A[i]$), $W -$ winnings($A, i, j-1, W - A[j]$)$\}$
return memo[$i,j$]
```

Assume that as a first step we compute the sum $W := W_{0,n-1}$. Clearly this can be done in $O(n)$ time. Then we call `winnings`$(A, 0, n-1, W)$. The runtime of `winnings`$(A, 0, n-1, W)$ *without* the runtime of the recursive sub-calls is just $O(1)$.

It remains to count the total number of recursive sub calls of `winnings`$(A, 0, n-1, W)$ we can have. As soon as the value `winnings`(A, i, j, W) for a pair i, j is computed, it will be put into `memo`$[i,j]$. From there on there will never be a recursive call `winnings`(A, i, j, W) for particular pair (i,j) again. In total we have $O(n^2)$ pairs $(i,j) \in [0, \ldots n-1]^2$ (note that the condition $i \leq j$ does not actually help in an asymptotic sense).

Exercise 2: Breaking Eggs *(10 Points)*

Imagine a building with n floors. We are given a supply of k eggs. For some reason we need to find out at which floor eggs start breaking when dropped from a window on that floor.

Suppose that dropping an egg from a certain floor always produces the same result, regardless of which egg is used and any other conditions. Initially, we do not have any knowledge at which height eggs might break. If an egg does not break, then it does not take any harm and can be fully reused.

If an egg breaks when dropped from a floor, it also breaks when dropped from higher floors. If eggs survive being dropped from a floor, they survive being dropped from lower floors. We call the floor from which dropped eggs break but survive at all floors beneath, the *critical floor.* The goal is to find the critical floor with minimum number of attempts (number of times eggs are being dropped).

(a) Suppose we have only one egg. Give a strategy to always find the critical floor, if it exists. *(1 Point)*

(b) We want some advance knowledge before starting to drop eggs. For inputs n and k, we want to compute the *exact* number of attempts $a(n,k)$ which an optimal strategy requires *in the worst case*, until it finds the critical floor (if it exists). Give a recursive relation for $a(n,k)$. Do not forget the base cases. Explain the correctness of the recursion. *(5 Points)*

(c) Give an algorithm that uses the principle of dynamic programming to compute $a(n,k)$ in $\mathrm{O}(kn^2)$ time. Explain the running time. *(4 Points)*

Sample Solution

(a) We start trying the first floor and as long as the egg survives we retry on the next higher floor. If the egg breaks, the current floor is the critical one. If the egg survives floor n, then there is no critical floor.

Fun fact: With k eggs, the runtime can be improved to $\mathrm{O}\big(k\sqrt[k]{n}\big)$. We repeat the strategy from part (a) for k phases in a nested fashion. Let $s_i := \lfloor(\sqrt[k]{n})^{k-i}\rfloor$ be the stepwidth in phase i and let $S_1 = [1..n]$ be the initial search space. For the first phase $i = 1$ we try every s_1-th floor in S_1 until the egg breaks and if it does not break we also try the n-th floor.

If the egg never breaks we are done since we know that there is no critical floor. If the egg breaks in the j-th attempt, we know that the critical floor must be one of the floors between floor $(j-1)s_1$ (excluding) and floor js_1 (including). We call this interval S_2. The size of S_2 is at most s_1 floors.

In the second phase $i = 2$ We repeat the procedure with stepwidth s_2 in interval S_2, which gives us an interval S_3 of size s_2, and so on. In general we repeat the procedure with stepwidth s_i on a search space S_i of size at most $|S_i| \leq s_{i-1}$. Finally, in phase k we have $i = k$ and the stepwidth is $s_k = 1$. Then we will definitely find the critical floor within search space S_k (c.f. part (a)).

Runtime: In iteration i we have $|S_i| = s_{i-1}$ floors left which we search with stepwidth s_i. This requires at most $\mathrm{O}\big(\frac{s_{i-1}}{s_i}\big) = \mathrm{O}\big(\sqrt[k]{n}\big)$ attempts. Since we have at most k phases the total number of attempts is $\mathrm{O}\big(k\sqrt[k]{n}\big)$.

(b) Assume we have k eggs and an interval of n consecutive floors that are left to check for the critical floor. We enumerate these from 1 to n (note that, in terms of computing the number of attempts, it does not really matter where exactly those floors are located on the building as long as they are consecutive).

If we drop an egg from floor i it might either break or survive. If it breaks, we have one egg less but we know that the critical floor must be somewhere in the interval $[1..i]$. If it survives we know that the critical floor must be somewhere in the interval $[i+1..n]$ (floor i can be ruled out).

Since we have to consider the worst case we must assume that the worst of the both options: "egg breaks", "egg survives" occurs. From all possible floors $i \in [1..n]$ we choose the one which minimizes the worst (i.e., maximum) of both outcomes in terms of number of attempts. Thus we obtain the following recursion:

$$a(n,k) = 1 + \min_{i \in [1..n]} \big\{ \max\big(a(i,k-1), a(n-i,k) \big) \big\}.$$

In terms of base cases, we know that worst case for $k = 1$ egg is n attempts, given that the critical floor is the top most one. In another base case we have narrowed down our interval of floors to $n = 0$, in which case we already found the critical floor. This must be the critical one under the assumption that it exists.

(c) **Algorithm 2** `attempts`(n,k) ▷ assume we have a global dictionary `memo` initialized with `Null`

```
  if $k = 1$ or $n = 0$ then return $n$                                    ▷ base case
  if memo[$n,k$] $\neq$ Null then return memo[$n,k$]
  memo[$n,k$] $\leftarrow 1 + \min_{i \in [1..n]} \big\{ \max\big($attempts($i, k-1$), attempts($n-i, k$)$\big) \big\}$
  return memo[$n,k$]
```

We have at most kn recursions and each recursion takes $O(n)$ time steps to compute the minimum (neglecting the time required for recursive subcalls). The total running time is therefore $O(kn^2)$.