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

Algorithms and Data Structures
Summer Term 2019
Sample Solution Exercise Sheet 10

Exercise 1: Edit Distance

Let $A = a_1 \dots a_n, B = b_1 \dots b_m$ be two words. For $k \leq n, \ell \leq m$ let $A_k = a_1 \dots a_k, B_\ell = b_1 \dots b_\ell$ be the prefixes of A und B. Let $ED_{k,\ell} := ED(A_k, B_\ell)$ be the edit distance of A_k, B_ℓ. Use the dynamic programming algorithm from the lecture to compute $ED_{n,m}$ for the inputs $A = ananas$ und $B = bananen$ by filling a table with values $ED_{k,\ell}$.

Sample Solution

We fill the following table according to the following recursion given in the lecture:

$$ED_{k,\ell} = \min\left(ED_{k,\ell-1} + 1, ED_{k-1,\ell} + 1, ED_{k-1,\ell-1} + \mathbb{1}_{a_k \neq b_\ell}\right)$$

$ED_{k,\ell}$	ε	b	a	n	a	n	e	n
ε	0	1	2	3	4	5	6	7
a	1	1	1	2	3	4	5	6
n	2	2	2	1	2	3	4	5
a	3	3	2	2	1	2	3	4
n	4	4	3	2	2	1	2	3
a	5	5	4	3	2	2	2	3
s	6	6	5	4	3	3	3	3

Exercise 2: Binomial Coefficient

Consider the following recursive definition of the binomial coefficient

$$\binom{n}{k} = \binom{n-1}{k} + \binom{n-1}{k-1},$$

with base cases $\binom{n}{0} = \binom{n}{n} = 1$. Give an algorithm that uses the principle of dynamic programming to compute $\binom{n}{k}$ in $\mathcal{O}(n \cdot k)$ time steps. Argue the running time of your algorithm

Sample Solution

In the worst case, the routine $\textsc{Binom}(n,k)$ computes all partial results $\textsc{Binom}(m,l)$ for $m \leq n$ und $l \leq k$. However, each partial result is computed at most *once* before it is globally available in `memo`. There are at most $\mathcal{O}(n \cdot k)$ partial results, hence we call $\textsc{Binom}(\cdot,\cdot)$ at most $\mathcal{O}(n \cdot k)$ times when computing $\textsc{Binom}(n,k)$. Each call of $\textsc{Binom}(\cdot,\cdot)$ takes $\mathcal{O}(1)$ if we neglect the time required for subcalls. Therefore the total time required is $\mathcal{O}(n \cdot k)$.

Algorithm 1 $\textsc{Binom}(n, k)$ ▷ *global dictionary* `memo` *initialized with* `Null`

```
if k = 0 or k = n then return 1                          ▷ base cases
if memo[n, k] = Null then                                ▷ result not yet computed
    memo[n, k] ← Binom(n−1, k) + Binom(n−1, k−1)         ▷ compute partial results
return memo[n, k]
```

Exercise 3: Packaging marbles

We are given n marbles and have access to an (arbitrary) supply of packages. We are also given an array $A[1..n]$, where entry $A[i] \geq i$ is the value of a package containing exactly i marbles. Our profit is the total value of all packages containing at least one marble, minus the cost of packaging, which is i for a package containing i marbles. We want to maximize our profit.

(a) Give an efficient algorithm that uses the principle of dynamic programming to package marbles for a maximum profit.

(b) Argue why your algorithm is correct. Give a tight (asymptotic) upper bound for the running time of your algorithm and prove that it is an upper bound for your solution.

Sample Solution

(a) We propose the following algorithm:

Algorithm 2 `profit(n)` ▷ *global dictionary* `memo` *initialized with* `Null`

```
if n = 0 then return 0                                   ▷ base case
if memo[n] ≠ Null then return memo[n]                    ▷ profit was computed before
memo[n] ← max_{i∈[1..n]} (A[i] + profit(n−i))           ▷ memoization
return memo[n]
```

(b) The first observation is that the packaging cost always sums up to n, so it does not play any role for our profit and can therefore be neglected (we could also just subtract i from array entry $A[i]$).

Let $p(n)$ be the maximum profit we can achieve when we have n marbles left for packaging. Obviously, we have n choices how many marbles (say i) to put into the next package. We choose the number i which optimizes the profit for the current package plus the maximum profit we can achieve with the remaining marbles. We obtain the following recursion:

$$p(n) = \max_{i\in[1..n]} \big(A[i] + p(n-i)\big), \quad p(0) = 0$$

Due to the memoization we compute each value $p(i)$ for $i \in [1..n]$ at most once. Each computation of $p(i)$ costs at most $\mathcal{O}(n)$ in the current step (determining the maximum of at most n numbers) not counting the cost of recursive calls. The total cost is therefore $\mathcal{O}(n^2)$. The upper bound for this algorithm is tight because we compute each value $p(i)$ for $i \in [1..n]$ with a cost of $\Omega(i)$ in the current recursion.