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

Algorithms Theory

Sample Solution Exercise Sheet 3

Due: Monday, 3rd of December, 2018, 14:15 pm

Exercise 1: Packaging Marbles *(4+4 Points)*

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 $A[i]$ equals 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 1 `profit(`n`)` ▷ assume we have a 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 $O(n)$ in the current step (determining the maximum of at most n numbers) not counting the cost of recursive calls. The total cost is therefore $O(n^2)$.

Not required for full points: 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.

Exercise 2: Breaking Eggs[1] *(1+5+5+5 Points)*

Imagine a building with n floors. Additionally, 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.

When eggs break when dropped from a floor, they also break when dropped from higher floors. When 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 minimal 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.

(b) For $n \gg k$, describe a strategy that finds the critical floor with $\mathrm{O}\big(k\sqrt[k]{n}\big)$ attempts, if it exists.

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

(d) Argue the correctness of your algorithm and its running time.

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.

(b) 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 $|S_i| \le s_{i-1}$. 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: (not required for full points). 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 $\mathrm{O}\big(k\sqrt[k]{n}\big)$.

(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`$[n,k] \leftarrow 1 + \min_{i\in[1..n]}\{\max\big($`attempts`$(i,k-1),$ `attempts`$(n-i-1,k)\big)\}$
return `memo`$[n,k]$

(d) 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 (it does not really matter where exactly those floors are located on the building as long as they are consecutive).

[1] This question was part of some of Google's job-interviews.

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 worse of both outcomes (in terms of attempts). Thus we obtain the following recursion:

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

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 $O(kn^2)$.

Exercise 3: Binary Counter *(3+4 Points)*

If we use a counter in standard binary representation, then an arbitrary series of `increment` and `decrement` operations can incur relatively high cost in terms of number of bits that need to be flipped. E.g. decrementing and incrementing the number 2^k in binary representation repeatedly in an alternating fashion incurs $\Theta(k)$ bit flips for each operation.

We introduce a more efficient data structure, where the counter is represented by two binary numbers P and N, and the current state of the counter is $P-N$. The counter is initialized with $P = N = 0$. The `increment` and `decrement` operations are implemented as follows.

Algorithm 3 `increment`(P, N)

```
i ← 0
while P_i = 1 do          ▷ P_i is the i^th bit of P.
    P_i ← 0
    i ← i + 1
if N_i = 1 then N_i ← 0
else P_i ← 1
```

Algorithm 4 `decrement`(P, N)

```
i ← 0
while N_i = 1 do          ▷ N_i is the i^th bit of N.
    N_i ← 0
    i ← i + 1
if P_i = 1 then P_i ← 0
else N_i ← 1
```

(a) Let $P = 100110$ and $N = 001000$. Give the state of P, N (in binary representation) and $P-N$ (in decimal representation) after each of the following operations: `increment`, `increment`, `increment`, `increment`, `decrement`, `decrement`.

(b) Use the accounting method to show that any series of `increment` and `decrement` operations on a counter initialized with $P = N = 0$ has amortized costs of $O(1)$ (number of bits flipped) per operation.

Sample Solution

(a) The results are given in the following table:

	`increment`	`increment`	`increment`	`increment`	`decrement`	`decrement`
P	100111	100000	100001	100010	100010	100000
N	001000	000000	000000	000000	000001	000000
$P-N$	31	32	33	34	33	32

(b) As in the lecture, we pay 1 coin to the bank whenever we flip a bit from zero to one. When we flip the same bit back to zero we take the coin back from the bank account to pay for that flip. Obviously, we always have enough coins on our account to pay for flips from one to zero.

If the operation itself costs one coin, flips from zero to one cost 2 coins (one for the operation itself, one is paid to the bank). Flips from one to zero are essentially "free" (we take a coin from the bank account to pay for it).

As we can see from the given pseudo codes we make at most one flip from zero to one per execution. Thus the amortized cost of either `increment` or `decrement` is 2 coins (= bitflips).

Exercise 4: Dynamic Array *(2+7 Points)*

In the lecture we saw a dynamically growing array that implements the `append` operation in amortized O(1) (reads/writes). For an array of size N that is already full, the `append` operation allocates a new array of size $2N$ ($\beta = 2$) before inserting an element at the first free array entry.

Additionally, we introduce another operation `remove`, which writes `Null` into the last non-empty entry of the dynamic array. However, by appending many elements and subsequently removing most of them, the ratio of unused space can become arbitrarily high. Therefore, when the dynamic array of size N contains $\frac{N}{4}$ or less elements after `remove`, we copy each element into a new array of size $\frac{N}{2}$.

(a) Given an array of size N which has at least $\frac{N}{2}$ elements, show that any series of `remove` operations has an amortized cost of O(1) (reads/writes) per operation.

(b) Use the potential function method to show that any series of `append` and `remove` operations has amortized cost of O(1) (reads/writes) per operation. Assume that the number of elements n in the array is initially 0 and assume that the array never shrinks below its initial size N_0 (we stop allocating smaller arrays in that case).

Remarks: You may assume that N is always a power of two. You may also assume that allocating an empty array is free, only copying elements costs one read and one write for each copied element. If you do part (b) absolutely correctly you automatically receive all points for part (a).

Sample Solution

(a) We pay 2 coins to the bank for each `remove` operation (1 coin = 1 read/write operation). When the number of elements in the array reaches $\frac{N}{4}$ (down from $\frac{N}{2}$) we conducted exactly $\frac{N}{4}$ `remove` operations and have $2 \cdot \frac{N}{4} = \frac{N}{2}$ coins on our bank account.

We use this amount to copy (one read and subsequently one write) each element into the new, smaller array for "free" (we make $\frac{N}{4}$ reads and $\frac{N}{4}$ writes). Each `remove` operation costs 1 coin (1 write) plus 2 coins (paid to bank) which makes an amortized cost of 3 coins = O(1) (read/writes).

Afterwards, the newly allocated array is again half full and the argument can be repeated.

(b) We define a potential function $\Phi(n, N)$ that is small when it contains exactly $\frac{N}{2}$ elements, and large when the number of elements approaches N or $N/4$ respectively, in order to pay for the imminent increase or decrease of the array size:

$$\Phi(n, N) := 2 \cdot |2n - N|.$$

Since we use absolute values, we obviously have $\Phi(n, N) \geq 0$ for any values n, N. For the amortized costs we make some case distinctions. We start with the amortized cost $a_{n,N}$ of `append`:

Case $n = N$:

$$\begin{aligned} a_{N,N} &= 2N + \Phi(N{+}1, 2N) - \Phi(N, N) \\ &= 2N + 2(2(N{+}1) - 2N) - 2(2N - N) = 4 \end{aligned}$$

Case $n < N$, $n \geq N/2$:

$$\begin{aligned} a_{n,N} &= 1 + \Phi(n{+}1, N) - \Phi(n, N) \\ &= 1 + 2(2(n{+}1) - N) - 2(2n - N) = 1 + 4 = 5 \end{aligned}$$

Case $n < N$, $n < N/2$:

$$\begin{aligned} a_{n,N} &= 1 + \Phi(n{+}1, N) - \Phi(n, N) \\ &= 1 + 2(N - 2(n{+}1)) - 2(N - 2n) = 1 - 4 \leq 0 \end{aligned}$$

Now the amortized cost $r_{n,N}$ of `remove`:

Case $n = N/4{+}1$:

$$\begin{aligned} r_{N/4+1,N} &= N/2 + \Phi(N/4, N/2) - \Phi(N/4{+}1, N) \\ &= N/2 + 2(N/2 - 2(N/4)) - 2(N - 2(N/4{+}1)) = N/2 - N + 4 \leq 4 \end{aligned}$$

Case $n > N/4{+}1$, $n \leq N/2$:

$$\begin{aligned} r_{n,N} &= 1 + \Phi(n{-}1, N) - \Phi(n, N) \\ &= 1 + 2(N - 2(n{-}1)) - 2(N - 2n) = 1 + 4 = 5 \end{aligned}$$

Case $n > N/4{+}1$, $n > N/2$:

$$\begin{aligned} r_{n,N} &= 1 + \Phi(n{-}1, N) - \Phi(n, N) \\ &= 1 + 2(2(n{-}1) - N) - 2(2n - N) = 1 - 4 \leq 0 \end{aligned}$$