Algorithm Theory, Tutorial 3

Johannes Kalmbach

University of Freiburg

johannes.kalmbach@gmail.com

November 2018

Algorithm Writeups

- Slides now available at `https://github.com/joka921/TutorialAlgoTheo`
- Pseudocode, limit to important aspects
- If variables are used, give them good names
- Especially if your algorithm has more than (literally) 3 lines.

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 $A[i]$ 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.

Packaging costs

- Not clear in exercise: $A[i] > i$?
- Do we have to package all marbles, even if it is bad for us?
- Only small changes needed for each case
- If we have to package all marbles or if this is good for us, the packaging cost is always n and can thus be ignored
- Any option was fine

- When we want to pack n marbles optimally we can either pack:
- All the marbles into one package
- One marble into a package and pack $n-1$ marbles optimally
- Two marbles into a package and pack the other $n-s$ marbles optimally
- 2 marbles optimally and the other $n-2$ marbles optimally
- ...
- This reasoning directly gives 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[`i`]` $\neq$ `Null` **then return** `memo[`i`]` ▷ profit was computed before

`memo[`n`]` $\leftarrow \max_{i \in [1..n]} \big(A[i] + \texttt{profit}(n-i)\big)$ ▷ Memoization

return `memo[`n`]`

Dropping eggs

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).

2a

- Suppose we have only one egg. Give a strategy to always find the critical floor, if it exists.
- Solution: trivial (everybody got that)

2b

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

- Idea: Divide into smaller partitions and check the border
- Crucial to set number/size of partitions right to get runtime correct and not run out of eggs.
- Hint: First derive case for $k = 2$

General case

1. Consider all floors
2. Divide the considered floors into $\sqrt[k]{n}$ partitions.
3. Throw an egg at the first floor of each partition
4. When the egg breaks we know that we from now on have to consider only one of the partitions, with this knowledge go back to step 2

Why does this terminate? Look at partition size after i iterations:

- 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 $O(kn^2)$ time.
- Argue the correctness of your algorithm and its running time.

- At each step the only thing we can do is throw eggs from a given floor
- But we can choose the best floor -¿ minimize
- Two things might happen
  - The egg breaks $\Rightarrow$ have one egg less, but only have to search below.
  - The egg survives, we only have to search above with same number of eggs.
- : observation: subproblem gets strictly smaller (number of eggs or floors or both get less)
- Base cases: If we have only one egg, we have to search linearly
- If we have only 0 floors, we have to do nothing
- This directly gives us a dynamic programming algorithm:

Algorithm 2 $\texttt{attempts}(n, k)$ ▷ assume we have a global dictionary `memo` initialized with `Null`

if $k = 1$ or $n = 0$ **then return** n ▷ base case

if $\texttt{memo}[n, k] \neq \texttt{Null}$ **then return** $\texttt{memo}[n, k]$

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

return $\texttt{memo}[n, k]$

Runtime:

Counter

We have given an efficient binary counter implementation:

Algorithm 3 `increment`(P, N)

$i \leftarrow 0$
while $P_i = 1$ **do** ▷ P_i is the i^{th} bit of P.
    $P_i \leftarrow 0$
    $i \leftarrow i + 1$
if $N_i = 1$ **then** $N_i \leftarrow 0$
else $P_i \leftarrow 1$

Algorithm 4 `decrement`(P, N)

$i \leftarrow 0$
while $N_i = 1$ **do** ▷ N_i is the i^{th} bit of N.
    $N_i \leftarrow 0$
    $i \leftarrow i + 1$
if $P_i = 1$ **then** $P_i \leftarrow 0$
else $N_i \leftarrow 1$

3a

Execute the following Operations: 4 Increments, then 2 Decrements

Operation	State of P	State of I
Initial	100110	001000
Increment		
Increment		
Increment		
Increment		
Decrement		
Decrement		

Use accounting to show amortized cost

- Pay one to the bank when we set a bit from 0 to 1.
- This happens at most once per operation.
- Pay for the flips from 1 to 0 using the bank account.
- For each flip from 1 to 0 there was a previous 0 to 1 operation for the same bit, so we indeed have this money on the bank.
- Thus each operation is constant (pay for 1 flip from 0 to $1 + 1$ to the bank) and our bank account is never negative.
- (Bank account is number of ones in counter)

Dynamic Array

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 $\frac{N}{2}$.

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).
Many of you assumed slightly different costs, but this was fine, since they do not change the asymptotic behavior. Still read the exercises Carefully.

4a

- 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).
- 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).

4b

- Use the potential function method to show that any series of `append` and `remove` operations has amortized cost of $O(1)$ (reads/writes). 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).
- 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.

Append or remove, no resize

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

- The cost for those operations is constant
- The change in the potential function is also constant (+- 4)
- Thus those operations are amortized constant.

Append with resize

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

Remove with resize

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