Sheet 4 / Ex 3 has become easier, check Forum for details

Algorithm Theory, Tutorial 3

Johannes Kalmbach

University of Freiburg

johannes.kalmbach@gmail.com

December 2019

General Hints

- Contact tutor (`johannes.kalmbach@gmail.com`) for questions concerning corrections etc.
- Contact forum (`daphne.informatik.uni-freiburg.de`) for everything else
- Suggestion: Submit in groups of two (better for understandable algorithms)
- Submit readable solutions (LaTeX as pdf, CLEAN handwriting (+ good scan if necessary))
- Spend enough time on exercise sheets and writeup (you and I have to understand your submission).

Algorithm Writeups

- Pseudocode, limit to important aspects
- Reader must be able to understand and implement it.
- E.g "Split Array A in two evenly-sized halves L and R"

Dynamic Programming in a nutshell

- Problem can be formulated recursively
- The number of subproblems / subresults that are actually needed/called is bounded (e.g. polynomial in input size)
- Use Memoization to only calculate each subresult once.
- Directly gives us an efficient algorithm
- **Do not forget the base cases for recursion**
- Alternatively: Bottom-up formulation (might be more efficient in practice (not asymptotically), but sometimes harder to reason about).

Knapsack Problem

- We are given n items that each have a weight $w_i > 0$ and a value $v_i > 0$.
- Additional we are given a maximum capacity W of our knapsack.
- What is the maximum value we can pack into our knapsack without exceeding the capacity?
- General case : $w_i, v_i \in \mathbb{R}$: **NP-COMPLETE**
- But we can do something in case we put additional constraints on w_i or v_i

Knapsack with integer **WEIGHTS**

- Example from Lecture, NOT from the exercise (many confused about this)
- $w_i \in \mathbb{N} \Rightarrow$ all valid combinations of items have a weight in $\{0, \dots W\}$.
- Define helper function $OPT(i, w)$ to be the maximum value that we can achieve with a weight $\leq w$ and only the items $\{1, \dots i\}$.
- $OPT(i, 0) = 0$, $OPT(0, k) = 0$ (no weight or no items allowed $\Rightarrow$ no value can be achieved.)
- $OPT(i, w) = 0$ if $w < 0$
- $OPT(i, w) = \max(OPT(i-1, w), OPT(i-1, w-w_i) + v_i)$

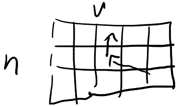
  don't choose item i

  choose i
-
-
-
- We only need $n \cdot W$ different subresults to calculate $OPT(n, W)$, so using memoization gives us $\mathcal{O}(nW)$ algorithm.

Knapsack with integer **VALUES**

- Weights might be arbitrary real numbers now, algorithm from previous slide does not work efficiently.
- However we can do something similar with the values (Read exercises and hints carefully!)
- $v_i \in \mathbb{N} \Rightarrow$ all valid combinations of items have a value in $\{0, \ldots \underline{\sum_i v_i =: V\}}$. define this

Hint
- Define helper function $OPT(i, v)$ to be the minimum weight that we can achieve with a value $\geq v$ and only the items $\{1, \ldots i\}$.
- $OPT(i, 0) = 0$, $OPT(0, v) = \infty$ if $v > 0$ ✓ no items allowed => no value
- $OPT(i, v) = 0$ if $v < 0$
- $OPT(i, v) = \min\left(\underbrace{OPT(i-1, v)}_{\text{don't choose } i}, \underbrace{OPT(i-1, v - v_i) + w_i}_{\text{choose } i}\right)$
-
- We only need $n \cdot V$ different subresults to calculate $OPT(n, V)$, so using memoization gives us $\mathcal{O}(n\cancel{W})$ algorithm.
  $O(nV)$



- Define helper function $OPT(i, v)$ to be the minimum weight that we can achieve with a value $\geq v$ and only the items $\{1, \ldots i\}$.
- Calculate $OPT(n, v)$ for all $v \in \{1, \ldots V\}$. The biggest v for which $OPT(n, v) \leq W$ is our maximum value.
- How to construct the actual **set** that achieves this value? (Was asked in the exercise)
-
- $OPT(i, v) = \min\left(OPT(i-1, v), OPT(i-1, v-v_i) + w_i\right)$
- For each $OPT(i, v)$ we decide whether to include i or not, and which previous value we used. If we store these pointers / edges during computation, we can also retrieve the actual set.

Dynamic Programming

Con~~is~~sider the following functions $f_i : \mathbb{N} \to \mathbb{N}$

$$f_1(n) = n - 1$$

$$f_2(n) = \begin{cases} \frac{n}{2} & \text{if 2 divides } n \\ n & \text{else} \end{cases}$$

$$f_3(n) = \begin{cases} \frac{n}{3} & \text{if 3 divides } n \\ n & \text{else} \end{cases}$$

"m divides n" means there is a $k \in \mathbb{N}$ with $k \cdot m = n$.

For a given $n \geq 1$, we want to find ~~die~~ the minimal number of applications of the functions f_1, f_2, f_3 needed to reach 1. Formally: Find the minimal k for which there are $i_1, \ldots, i_k \in \{1, 2, 3\}$ with $f_{i_1}(f_{i_2}(\ldots(f_{i_k}(n))\ldots)) = 1$.

Consider the following algorithms:

Greedy	Better
10	10
5	9
4	3
2	1
1	

```
def min_f(n: positive integer):
 if n == 1 : return 0
 if 3 divides n:
   return min_f(n / 3) + 1
 if 2 divides n:
   return min_f(n / 2) + 1
 return min_f(n-1) + 1
```

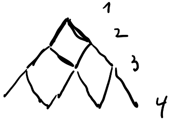
- What type of algorithm is this? Greedy
- What runtime ~~has it?~~ does it have?
- Does it solve the problem?

Dynamic Programming in O(n)

```
def helper(n: positive integer, memo: dictionary):
if n == 1 : return 0
if n in memo:
  return memo[n]
                 memo
x:= helper(n-1)
if 3 divides n:                memo
  x:= min(x, helper(n/3))
if 2 divides n:                  memo
    x:= min(x, helper(n/3))
                        2
memo[n] = x + 1
return x + 1
```

```
def min_t(n):
    return helper(n, empty dictionary)
```

- One student found an even faster solution, based on the following idea:
- The optimal solution can never perform (-1) (-1) (/2), because (/2) (-1) would be faster.
- Similarly, (-1) (-1) (-1) (/3) is not optimal. $\Rightarrow$ (/3) (-1)
- Using this and a nice numerical property $\lfloor \frac{\lfloor \frac{n}{3} \rfloor}{2} \rfloor = \lfloor \frac{\lfloor \frac{n}{2} \rfloor}{3} \rfloor$ we can use this to implement an
- $\mathcal{O}((\log n)^2)$ algorithm (also based on DP)



log n levels

$(1 + 2 + 3 + 4 + 5 + \dots \log n)$

$\in O((\log n)^2)$

Amortized Analysis, Potential function

- Please consider this topic in your exam preparations, they

Exercise 3

Suppose a sequence of n operations are performed on an (unknown) data structure in which the i-th operation costs i if i is an exact power of 2, and 1 otherwise.

Operation	1	2	3	4	5	6	7	8	9	...	15	16	17	...
Actual Cost	1	2	1	4	1	1	1	8	1	...	1	16	1	...

Use the **potential function** method to show that each operation has constant amortized cost.

Hint: *The number of consecutive operations that are not an exact power of* 2 *and are performed immediately before operation* $(i+1)$ *is* $i - 2^{\ell(i)}$ *where* $\ell(i) := \lfloor \log_2 i \rfloor$.

Pot +K

take everything

Operation	1	2	3	4	5	6	7	8	9	...	15	16	17	...
Actual Cost	1	2	1	4	1	1	1	$\boxed{8}$	1	...	1	16	1	...

+2 +2 +2

Pay for

distance to

$$\phi_i := \underbrace{(i - 2^{\lfloor \log_2 i \rfloor})}_{\text{distance to}} \cdot 2$$

0 at powers of 2

previous power of two

$\phi_i \geq 0$

$\phi_i(0) = \text{const}$

Operation	1	2	3	4	5	6	7	8	9	...	15	16	17	...
Actual Cost	1	2	1	4	1	1	1	8	1	...	1	16	1	...

Amortized cost

$$a_i = c_i + (\Theta_i - \Theta_{i-1})$$

Case 1: i not a power of 2

$$\Rightarrow 2^{\lfloor \log i \rfloor} = 2^{\lfloor \log(i-1) \rfloor} =: l$$

$$\Rightarrow a_i = 1 + 2(i-l) - (i-1-l)\cdot 2$$

$$= 3 \checkmark$$

Operation	1	2	3	4	5	6	7	8	9	...	15	16	17	...
Actual Cost	1	2	1	4	1	1	1	8	1	...	1	16	1	...

case 2, i is power of 2

$$a_i = c_i + \Phi_i - \Phi_{i-1} = i + 2\left(\cancel{i} - \cancel{\log}\, \overset{= i}{2^{\lfloor \log i \rfloor}}\right) - 2\left(\cancel{i} - 1 - \overset{\frac{i}{2}}{2^{\lfloor \log (i-1) \rfloor}}\right)$$

$$= 2$$

Operation	1	2	3	4	5	6	7	8	9	...	15	16	17	...
Actual Cost	1	2	1	4	1	1	1	8	1	...	1	16	1	...