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

Algorithm Theory
Sample Solution Exercise Sheet 3

Exercise 1: Knapsack with Integer Values *(11 Points)*

Given n items $1, \dots, n$ with weights $w_i \in \mathbb{R}$ and values $v_i \in \mathbb{N}$ and a bag capacity W, we want to find a subset $S \subseteq \{1, \dots, n\}$ that maximizes $\sum_{i \in S} v_i$ under the restriction $\sum_{i \in S} w_i \leq W$.

Give an efficient[1] algorithm for this problem that uses the principle of dynamic programming.

Hint: Define a function that computes for a $k \in \{1, \dots, n\}$ and an integer v the minimum weight of a collection of items from $\{1, \dots, k\}$ that has value v.

Sample Solution

Let $\texttt{minweight}(k, v) = \min\{\texttt{minweight}(k-1, v), w_k + \texttt{minweight}(k-1, v - v_k)\}$

Base cases:

- $\texttt{minweight}(k, 0) = 0$
- $\texttt{minweight}(0, v) = \infty$ for $v > 0$
- $\texttt{minweight}(k, v) = \infty$ for $v < 0$

Remark: With this definition, $\texttt{minweight}(k, v)$ equals the minimum weight of a collection of items that has value $= v$ (and is set to ∞ if there is no collection summing up to v). If one changes the third base case to $\texttt{minweight}(k, v) = 0$ for $v < 0$, then $\texttt{minweight}(k, v)$ equals the minimum weight of a collection of items that has value $\geq v$. With both definitions we can proceed as follows.

Let $v_{sum} := \sum_{i=1}^{n} v_i$. Set up a table T of size $(n \times v_{sum})$ with $T[i, j] = \texttt{minweight}(i, j)$. Computing a single entry of the table takes $O(1)$ using the recursive formula, so the overall time to compute T is $O(n \cdot v_{sum})$. Let $j' = \max\{j \mid T[n, j] \leq W\}$. If all entries in row n are $> W$, we set $j' = 0$.
It follows that j' equals the value of an optimal solution. To obtain the solution (i.e., the collection of items summing up to j'), one can trace back from entry (n, j') through the table according to the recursive formula where in each step we jump up one row. If we also jump left when going from row $i + 1$ to row i, we add item i.

Exercise 2: Dynamic Programming *(10 Points)*

Conisder the following functions $f_i : \mathbb{N} \to \mathbb{N}$

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

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

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

[1] under the assumption that the maximum value is polynomial in n

"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 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 $f_{i_1}(f_{i_2}(\ldots(f_{i_k}(n))\ldots) = 1$.

Devise an algorithm in pseudocode to solve the problem and analyze the runtime.

Sample Solution

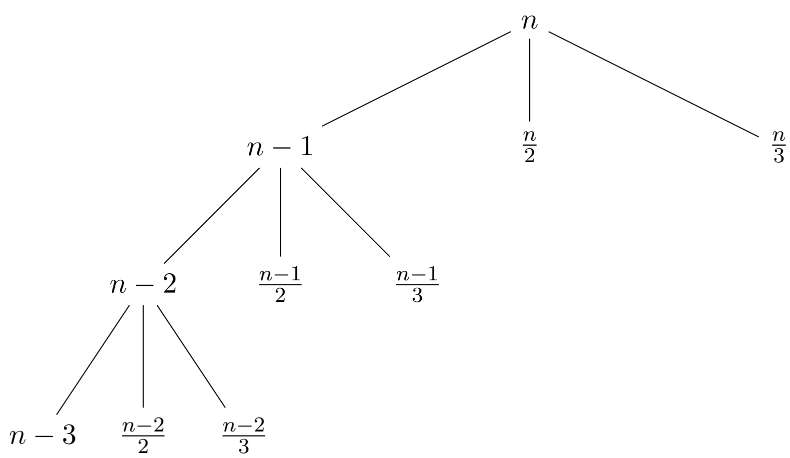
$memo = \{\}$

Algorithm 1 `steps_to_one`(n)

```
 1: if n in memo then
 2:     return memo[n]
 3: if n == 1 then
 4:     s = 0
 5: else
 6:     x = steps_to_one(n − 1)
 7:     if n | 2 then
 8:         y = steps_to_one(n/2)
 9:     else
10:         y = ∞
11:     if n | 3 then
12:         z = steps_to_one(n/3)
13:     else
14:         z = ∞
15:     s = 1 + min{x, y, z}
16: memo[n] = s
17: return s
```

Runtime analysis: By repeatedly calling `steps_to_one`($n-1$) in line 6 we go down the recursion tree until reaching 1. When going the tree up, in each step there are at most three recursive calls of `steps_to_one` and each of them is either a base case or takes a value from *memo*. Therefore, going one level up in the tree takes $O(1)$ and so the overall runtime is $O(n)$.

The recursion tree looks as follows (the branches with fractional values only exist if the fraction is an integer)



Exercise 3: Amortized Analysis *(9 Points)*

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	...

Tabelle 1: Operations and their actual costs

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* $\ell(i) := \lfloor \log_2 i \rfloor$.

Sample Solution

Define $\phi(0) = 0$ and $\phi(i) = 2(i - 2^{\ell(i)})$ for $i \geq 1$. If c_i is the actual cost of operation i, we define the amortized cost of operation i as $a_i = c_i + \phi(i) - \phi(i-1)$. As $\phi(0) = 0$ and $\phi(i) \geq 0$ for $i \geq 0$, we have $\sum_{i=1}^{n} a_i \geq \sum_{i=1}^{n} c_i$ for all $n > 0$ (i.e., the definition of the amortized costs is 'feasible').
For the first operation we get $a_1 = 1 + \phi(1) - \phi(0) = 1$. If $i > 1$ is not a power of 2, we have $\ell(i) = \ell(i-1)$ and hence $a_i = 3$. If $i = 2^k$ for a $k > 0$, we have

$$a_i = 2^k + \phi(i) - \phi(i-1) = 2^k + 0 - 2(2^k - 1 - 2^{k-1}) = 2^k - 2^{k+1} + 2 + 2^k = 2$$