Eidgenössische
Technische Hochschule
Zürich

Ecole polytechnique fédérale de Zurich
Politecnico federale di Zurigo
Federal Institute of Technology at Zurich

Department Informatik
Markus Püschel David Steurer Peter Widmayer
Chih-Hung Liu
Stefano Leucci

# Datenstrukturen & Algorithmen Blatt P9 HS 17

**Solution for Exercise P9.1** *Dyno.*

The problem can be solved by using a dynamic programming algorithm. For $i = 0, \dots, L-1$, we let $\mathrm{OPT}[i]$ be true iff Dyno can reach segment $i$. Moreover, let $E_i$ be true iff segment $i$ is empty. By the problem definition we know that $\mathrm{OPT}[0] = \text{true}$. For $i > 0$, $\mathrm{OPT}[i] = \text{true}$ if (i) segment $i$ is empty and (ii) at least one of the following two conditions holds:

- Dyno can reach segment $i-1$ (as Dyno can walk from segment $i-1$ to segment $i$); or
- Dyno can reach segment $i-D$, if it exists (as Dyno can jump from segment $i-D$ to segment $i$).

Otherwise $\mathrm{OPT}[i] = \text{false}$. In formulas: $\mathrm{OPT}[i] = E_i \wedge \big(\mathrm{OPT}[i-1] \vee \mathrm{OPT}[i-D]\big)$, where we assumed that $\mathrm{OPT}[j] = \text{false}$ for $j < 0$.

All the values $\mathrm{OPT}[i]$ can be computed in time $O(L)$ by considering the $L$ segments in increasing order of index while keeping track of the position of the next cactus (if any). That is, if `cacti` is the array containing the positions of the $C$ cacti (in sorted order), the algorithm maintains the following invariant: immediately before (resp. after) segment $i$ is considered, the algorithm stores the smallest index $k$ such that $\texttt{cacti}[k] \geq i$ (resp. $\texttt{cacti}[k] > i$), if any. Notice that updating $k$ only requires constant time per segment and that, once $k$ is known, it is possible to check whether $E_i = \text{true}$ in constant time.

The solution of the problem is now $\arg\max_{i=0,\dots,L-1} \mathrm{OPT}[i]$, which can be found in $O(L)$ time.

**Solution for Exercise P9.2** *Light Coffee.*

We define EVEN$[i][j]$ (resp. ODD$[i][j]$) to be the maximum number of Grahams that Alice can remove from her wallet if she must pay exactly $j$ Flops, can use only the first $i$ coins, and needs to pay using an even (resp. odd) number of coins. If there is no way to satisfy the above constraints, then we let EVEN$[i][j]$ (resp. ODD$[i][j]$) be equal to a sufficiently small value that we denote by $-\infty$.

Clearly, EVEN$[0][0] = 0$, EVEN$[0][j] = -\infty$ for $j \neq 0$, and EVEN$[i][j] = -\infty$ for any $i$ and $j < 0$. Similarly, ODD$[i][j] = -\infty$ whenever $i = 0$ (regardless of $j$) or $j < 0$ (regardless of $i$).

Consider EVEN$[i][j]$ for $i > 0$ and $j \geq 0$ and notice that, if Alice does not use the $i$-th coin, EVEN$[i][j] =$ EVEN$[i-1][j]$. On the contrary, if Alice does use the $i$-th coin, then she needs to pay $j - v_i$ Flops using an odd number of coins selected from the first $i-1$ coins. Hence:

$$\text{EVEN}[i][j] = \max\{\text{EVEN}[i-1][j], w_i + \text{ODD}[i-1][j-v_i]\},$$

and, with a similar reasoning:

$$\text{ODD}[i][j] = \max\{\text{ODD}[i-1][j], w_i + \text{EVEN}[i-1][j-v_i]\}.$$

Therefore the problem can be solved by using the previous relations in a dynamic programming algorithm that computes all EVEN$[i][j]$ and ODD$[i][j]$ in increasing order of $i = 0, \ldots, n$. Notice that it suffices to store EVEN$[i][j]$ and ODD$[i][j]$ for $0 \leq i \leq n$ and $0 \leq j \leq C$. Since each value can be computed in constant time, the algorithm requires $O(n \cdot C)$ time.

The solution to the problem is exactly the value of OPT$[n][C]$.