ALGORITHMS & DATA STRUCTURES

15th November 2021

TODAY’S PLAN

- Dynamic Programming

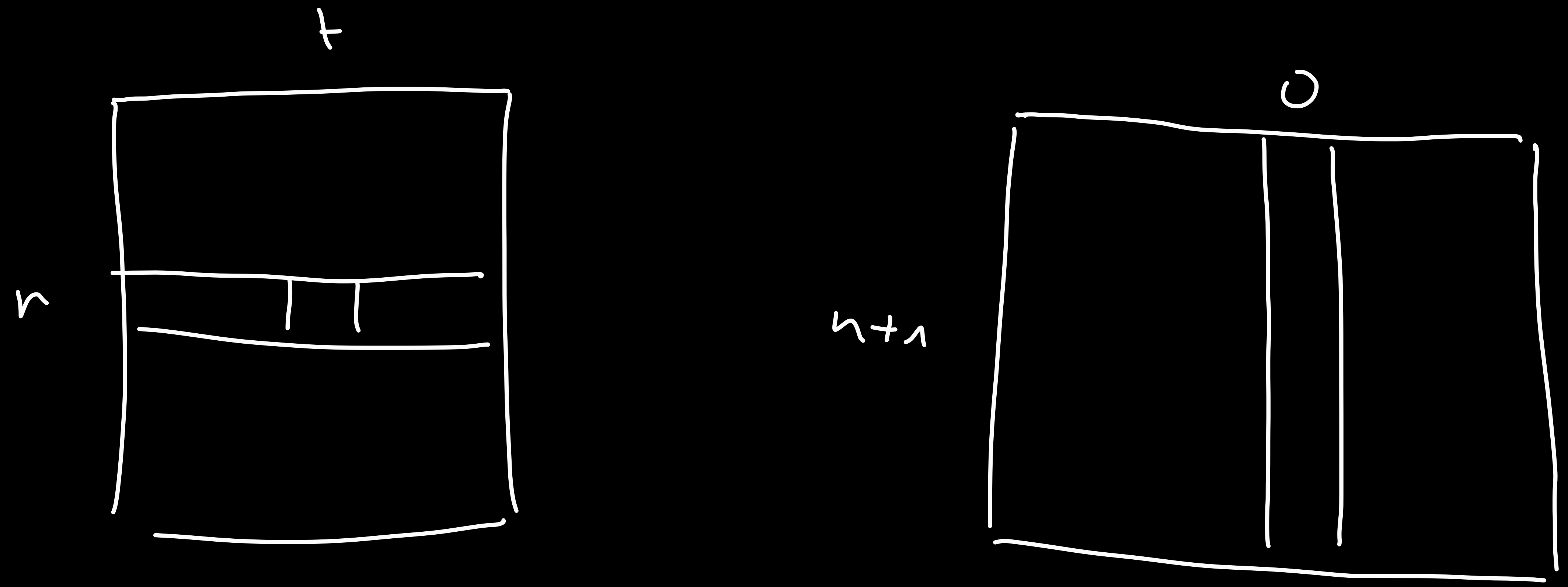
EXERCISE 7.1

Exercise 7.1 *Subset sum for general integers* **(1 point)**.

Let $a_1, \ldots, a_n, t$ be $n+1$ integers in $\mathbb{Z}$. We would like to check whether there is a subset $I \subseteq \{1, \ldots, n\}$ such that $\sum_{i \in I} a_i = t$. Here, we adopt the convention that if I is empty, then $\sum_{i \in I} a_i = 0$.

We have seen in class that if $a_1, \ldots, a_n, t$ are positive, then we can solve this problem in $O(nt)$ time using dynamic programming. In this exercise, we would like to handle the case where some of the integers $a_1, \ldots, a_n, t$ could be negative or zero.

Provide a *dynamic programming* algorithm that solves the subset sum problem for general integers. The algorithm should have $O\left(n \cdot \sum_{i=1}^{n} |a_i|\right)$ runtime.


t
n
0
n+1

Solution: Let $N := \sum_{a_i<0} |a_i|$ and $P := \sum_{a_i>0} a_i$. We can compute N and P in $O(n)$ time. Note that

$$N + P = \sum_{i=1}^{n} |a_i|.$$

It is easy to see that for every $I \subseteq \{1, \ldots, n\}$, we have $-N \leq \sum_{i \in I} a_i \leq P$. Therefore, if $t < -N$ or $t > P$, we can immediately say that the answer is no. In order to handle the case $-N \leq t \leq P$, we need dynamic programming.

Definition of the DP table: For $0 \leq i \leq n$ and $0 \leq j \leq N + P$, the entry $DP[i][j]$ is a boolean value indicating whether there is a subset $I \subseteq \{1, \ldots, i\}$ such that $\sum_{k \in I} a_k = j - N$. Here, we adopt the convention that for $i = 0$, we have $\{1, \ldots, i\} = \varnothing$.

Computation of an entry: Initialize

- $DP[0][N] = \textsc{true}$. This is because $\sum_{k \in \varnothing} a_k = 0 = N - N$.
- $DP[0][j] = \textsc{false}$, for every $j \neq N$.

Now for $i \geq 1$ and $0 \leq j \leq N + P$, we can compute $DP[i][j]$ using the formula

$$DP[i][j] = DP[i-1][j] \ \text{OR} \ (j \geq a_i \ \text{AND} \ DP[i-1][j - a_i]). \tag{1}$$

The proof of correctness of this formula is very similar to the one we saw in class for the subset sum problem: We just need to examine the cases where the subset $I \subseteq \{1, \ldots, i\}$ contains i or not.

Calculation order: We can calculate the entries of DP in order of increasing i. For fixed i, we can compute the entries $(DP[i][j])_{0 \leq j \leq N+P}$ in any order of j.

Extracting the solution: All we have to do is read the value at $DP[n][t + N]$.

Running time: The entry $DP[i][j]$ can be computed in $O(1)$ time. Therefore, the total runtime is

$$\sum_{i=0}^{n} \sum_{j=0}^{N+P} O(1) = O\big((n+1) \cdot (N+P+1)\big) = O(n \cdot (N+P)) = O\left(n \cdot \sum_{i=1}^{n} |a_i|\right).$$

$\Sigma |a_i| + 1$

$n+1$

EXERCISE 7.3

Exercise 7.3 *Road trip* **(1 point)**.

You are planning a road trip for your summer holidays. You want to start from city C_0, and follow the only road that goes to city C_n from there. On this road from C_0 to C_n, there are $n-1$ other cities $C_1, \ldots, C_{n-1}$ that you would be interested in visiting (all cities $C_1, \ldots, C_{n-1}$ are right on the road from C_0 to C_n). For each $0 \leq i \leq n$, the city C_i is at kilometer k_i of the road for some given $0 = k_0 < k_1 < \ldots < k_{n-1} < k_n$.

You want to decide in which cities among $C_1, \ldots, C_{n-1}$ you will make an additional stop (you will stop in C_0 and C_n anyway). However, you do not want to drive more than d kilometers without making a stop in some city, for some given value $d > 0$ (we assume that $k_i < k_{i-1} + d$ for all $i \in [n]$ so that this is satisfiable), and you also don't want to travel backwards (so from some city C_i you can only go forward to cities C_j with $j > i$).

Dimensions of the DP table: The DP table is linear, and its size is $n + 1$.

Definition of the DP table: $DP[i]$ is the number of possible routes from C_0 to C_i (which stop at C_i).

Computation of an entry: Initialize $DP[0] = 1$.

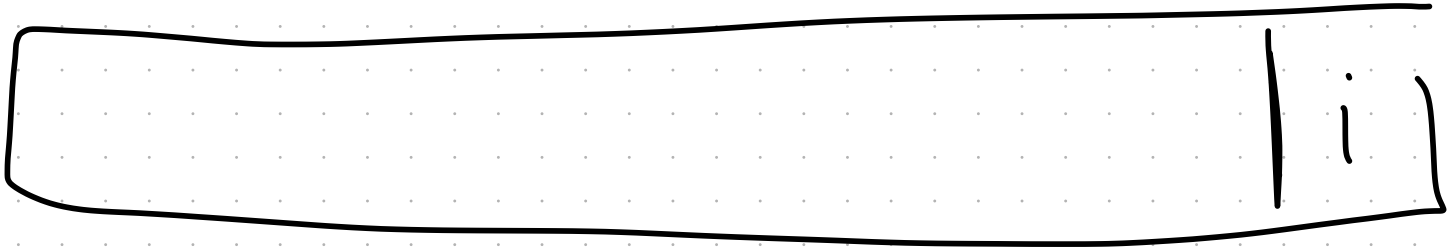
For every $i > 0$, we can compute $DP[i]$ using the formula

$$DP[i] = \sum_{\substack{0 \leq j < i \\ k_i \leq k_j + d}} DP[j]. \tag{2}$$

$O(i)$

$\Rightarrow$ ingesamt $O(n^2)$

$$\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$$



Für Eintrag $i-1$ wir hatten den kleinsten j, s.d. $h_i \leq h_j + d$. Für i, den kleinsten j ist grösser, als der für $i-1$. $\Rightarrow O(n)$

b) If you know that $k_i > k_{i-1} + d/10$ for every $i \in [n]$, how can you turn the above algorithm into a linear time algorithm (i.e., an algorithm that has $O(n)$ runtime) ?

$$k_i > k_{i-1} + \frac{d}{10}$$

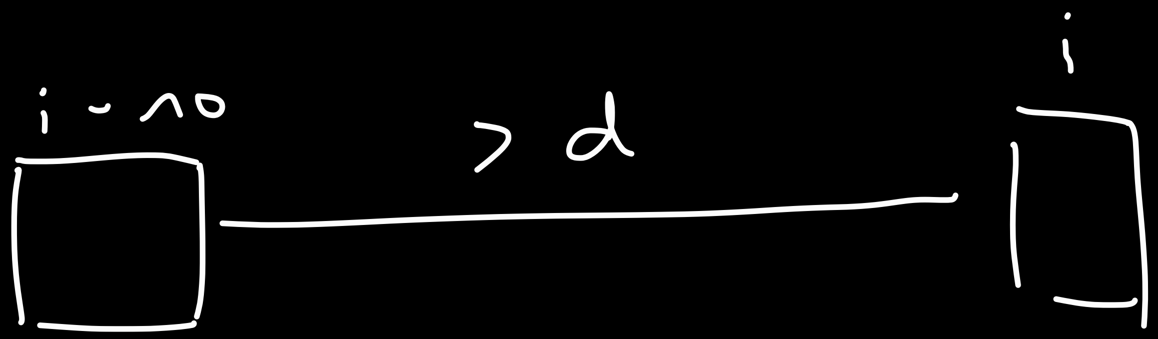
$$> k_{i-2} + \frac{d}{10} \cdot 2$$

$$> \dots$$

$$> k_{i-10} + d$$

Pro Eintrag brauchen wir $\leq 10 \in O(1)$ Schritte

$\Rightarrow O(n)$ total



EXERCISE 7.4

- N animals, each with a value v_i and weight w_i
- We want to maximize the values of the animals we pick without exceeding a combined sum of W
- If we take animal i (for $i >= 4$), we can not take a_{i-1}, a_{i-2}, a_{i-3}

Dimensions of the DP table: The DP table is two-dimensional, and its size is $(n+1) \times (W+1)$.

$\max_{1 \leq j \leq W+1}$

$DP[n+1][j]$

Definition of the DP table: For $0 \leq i \leq n$ and $0 \leq j \leq W$, the entry $DP[i][j]$ represents the maximum value of a collection of animals among $\{A_1, \ldots, A_i\}$, which has a total weight of at most j, and which does not contain any two animals where one of them is a predator of the other. Here, we adopt the convention that for $i = 0$, we have $\{A_1, \ldots, A_i\} = \varnothing$.

Computation of an entry: Initialize $DP[0][j] = 0$ for every $0 \leq j \leq W$.

For $1 \leq i \leq 3$ and $0 \leq j \leq W$, we can compute $DP[i][j]$ exactly like the knapsack problem using the formula

$$DP[i][j] = \max\Big\{DP[i-1][j] \ , \ \mathbf{1}_{\{j \geq w_i\}} \cdot \big(v_i + DP[i-1][j-w_i]\big)\Big\}. \tag{3}$$

Now for $4 \leq i \leq n$ and $0 \leq j \leq W$, we can compute $DP[i][j]$ using a modified formula that takes into account the predator constraint:

$$DP[i][j] = \max\Big\{DP[i-1][j] \ , \ \mathbf{1}_{\{j \geq w_i\}} \cdot \big(v_i + DP[i-4][j-w_i]\big)\Big\}. \tag{4}$$

$$I(j \geq w_i) = \begin{cases} 1, & \text{falls } j \geq w_i \\ 0 & \text{sonst} \end{cases}$$

$$W = f(n)$$

$$\sum_{i=1}^{n} \sum_{j=0}^{W} O(1) = O\big((n+1) \cdot (W+1)\big) = O(nW).$$

$$W = 2^n \qquad \Rightarrow O(nW) = O(n \cdot 2^n)$$

845. Longest Mountain in Array

Medium 1507 50 Add to List Share

You may recall that an array `arr` is a **mountain array** if and only if:

- `arr.length >= 3`
- There exists some index `i` (**0-indexed**) with `0 < i < arr.length - 1` such that:
  - `arr[0] < arr[1] < ... < arr[i - 1] < arr[i]`
  - `arr[i] > arr[i + 1] > ... > arr[arr.length - 1]`

Given an integer array `arr`, return *the length of the longest subarray, which is a mountain*. Return `0` if there is no mountain subarray.

Example 1:

```
Input: arr = [2,1,4,7,3,2,5]
Output: 5
Explanation: The largest mountain is [1,4,7,3,2] which has length 5.
```

Example 2:

```
Input: arr = [2,2,2]
Output: 0
Explanation: There is no mountain.
```

Constraints:

- $1 <= arr.length <= 10^4$
- $0 <= arr[i] <= 10^4$

Problems | Pick One | Prev | 142/344 | Next

```java
class Solution {
    public int comp(int[] arr, int idx){
        if(idx == arr.length - 1) return -1;
        if(arr[idx] >= arr[idx + 1]) return -1;
        int l = 1;
        int i = idx;
        while(i + 1 < arr.length && arr[i] < arr[i + 1]){ i++; l++;}
        if(i >= arr.length - 1 || arr[i] <= arr[i + 1]) return -1;
        while(i + 1 < arr.length && arr[i] > arr[i + 1]){ i++; l++;}
        return l;
    }
    public int longestMountain(int[] arr) {
        if(arr.length == 1) return 0;
        int res = 0;
        int idx = 0;
        while(idx < arr.length){
            int r = comp(arr, idx);
            if(r == -1) idx++;
            else{
                res = Math.max(res, r);
                idx += r - 1;
            }
        }
        return res;
    }
}
```

Testcase | Run Code Result | Debugger

Accepted Runtime: 0 ms

Your input: [875,884,239,731,723,685]

Output: 4 Diff

Expected: 4

Console | Use Example Testcases | Run Code | Submit

LeetCode Explore Problems Interview New Contest Discuss Store LeetCoding Challenge + GIVEAWAY! Premium

Description Solution Discuss (999+) Submissions

322. Coin Change

Medium 8921 225 Add to List Share

You are given an integer array `coins` representing coins of different denominations and an integer `amount` representing a total amount of money.

Return *the fewest number of coins that you need to make up that amount*. If that amount of money cannot be made up by any combination of the coins, return `-1`.

You may assume that you have an infinite number of each kind of coin.

Example 1:

```
Input: coins = [1,2,5], amount = 11
Output: 3
Explanation: 11 = 5 + 5 + 1
```

Example 2:

```
Input: coins = [2], amount = 3
Output: -1
```

Example 3:

```
Input: coins = [1], amount = 0
Output: 0
```

Example 4:

```
Input: coins = [1], amount = 1
```

Problems Pick One < Prev 51/344 Next >

Java Autocomplete

```java
class Solution {
    public int algo(int[] M, int[] coins, int amount){
        if(amount == 0) return 0;
        if(amount < coins[0]) return -1;
        if(M[amount] != -7) return M[amount];
        int best = Integer.MAX_VALUE;
        for(int i = 0; i < coins.length; i++){
            int r = algo(M, coins, amount - coins[i]);
            if(coins[i] <= amount && r != -1) best = Math.min(best, r);
        }
        if(best == Integer.MAX_VALUE){
            M[amount] = -1;
            return M[amount];
        }
        M[amount] = 1 + best;
        return M[amount];
    }
    public int coinChange(int[] coins, int amount) {
        Arrays.sort(coins);
        int[] M = new int[amount + 1];
        for(int i = 0; i < M.length; i++) M[i] = -7;
        return algo(M, coins, amount);

    }
}
```

Testcase Run Code Result Debugger

Accepted Runtime: 0 ms

Your input	[1] 7
Output	7
Expected	7

Diff

Console Use Example Testcases Run Code Submit