

Discrete (0/1) Knapsack

Frank

October 28, 2021

1 Discrete Knapsack Problem

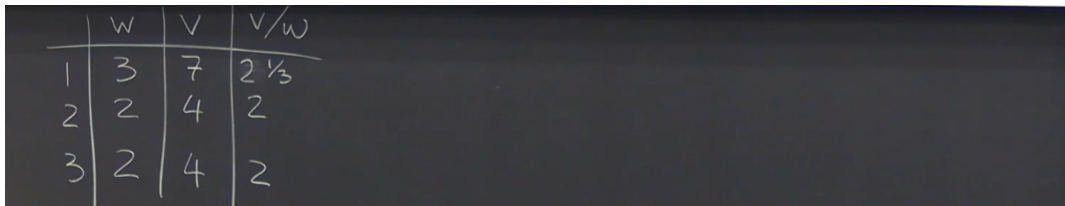
1.1 Problem

- **Input:**
 - A set of items $1, 2, \dots, n$
Item i has value $v(i) > 0$, and weight $w(i) > 0$
 - A knapsack of capacity C
Note: both weight and capacity are integers
- **Output:** A subset S of the items that
 1. $\sum_{i \in S} w(i) \leq C \leftarrow$ "Knapsack"
 2. maximize $\sum_{i \in S} v(i)$

Where a knapsack is a subset of the items that can fit into the knapsack

1.2 Why does greedy approach fail?

- Consider table



	w	v	v/w
1	3	7	$2\frac{1}{3}$
2	2	4	2
3	2	4	2

Let $C = 4$, then we see that once we take the item with the highest value to weight ratio, we are done, but do not have an optimal knapsack.

1.3 DP approach

Suppose S is an optimal knapsack for items $1..n$, we have 2 cases

1. $n \notin S \implies S$ is optimal for items $1..(n-1)$
2. $n \in S \implies S = S' \cup \{n\}$

Where S' is optimal for items $1..(n-1)$ for capacity $C - w(n)$ by cut-and-paste argument. Note that leftover capacity is an additional parameter that we need to keep track of, and defines more subproblems we have to worry about (more possibilities).

1.4 Subproblem

Find the optimal knapsack for 1.. i with capacity c , where $0 \leq c \leq C$ and $0 \leq i \leq n$, C is the maximum capacity.

Then our solution will be the knapsack for 1.. n with capacity C .

- As usual, compute the value of the knapsack, then retrofit it onto our original problem.

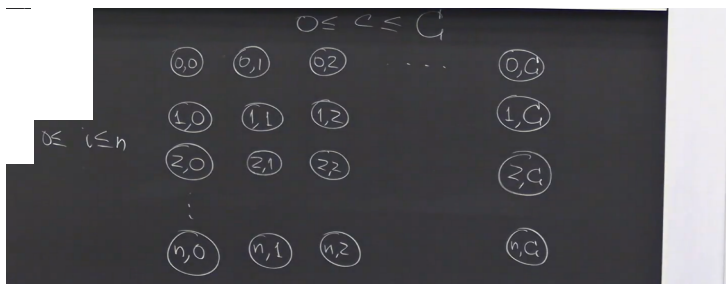
$K(i, c)$ = value of optimal knapsack for items 1.. i with capacity c , where $0 \leq i \leq n$ and $0 \leq c \leq C$ (\star)

Then consider recurrence

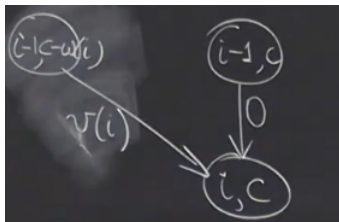
$$K(i, c) = \begin{cases} 0 & \text{if } i = 0 \text{ or } c = 0 \\ K(i-1, c) & \text{if } i > 0, c > 0, c < w(i) \quad (\dagger) \\ \max[K(i-1, c), v(i) + K(i-1, c - w(i))] & \text{if } i > 0, c > 0, c \geq w(i) \end{cases}$$

Where the first argument corresponds to case 1 (did not take i) and the second argument corresponds to case 2 (took i)

Visualize this recurrence



We need to consider the max between $K(i-1, c - w(i))$ (case 2) and $K(i-1, c)$ (case 1) to compute $K(i, c)$ in a directed-acyclic graph. Note that we can associate a weight between the edge of case 1 and case 2 that corresponds to the value of taking or not taking item i . Then we want maximum weight path from $K(0, 0)$ to $K(n, C)$ since we are trying to maximize value.



Observe that we do not actually need the values of all the sub problems, because we are essentially skipping over all nodes $K(i-1, j)$ where $c - w(i) < j < c$

1.5 Algorithm

```
1: procedure KNAPSACK-VALUE( $w, v, C$ )
2:   for ( $c = 0..C$ ) do                                     ▷ base cases
3:      $K(0, c) \leftarrow 0$ 
4:   for ( $i = 1..n$ ) do                                       ▷ already computed  $K(0, 0)$ 
5:      $K(i, 0) \leftarrow 0$ 
6:   for ( $i = 1..n$ ) do
7:     for ( $c = 1..C$ ) do
8:       if ( $c < w(i)$ ) then
9:          $K(i, c) \leftarrow K(i - 1, c)$ 
10:      else
11:         $K(i, c) \leftarrow \max[K(i - 1, c), v(i) + K(i - 1, c - w(i))]$ 
12:   return  $K(n, C)$ 
```

```
1: procedure KNAPSACK( $w, v, C$ )
2:   for ( $c = 0..C$ ) do                                     ▷ base cases
3:      $K(0, c) \leftarrow 0$ 
4:   for ( $i = 1..n$ ) do                                       ▷ already computed  $K(0, 0)$ 
5:      $K(i, 0) \leftarrow 0$ 
6:   for ( $i = 1..n$ ) do
7:     for ( $c = 1..C$ ) do
8:       if ( $c < w(i)$ ) then
9:          $K(i, c) \leftarrow K(i - 1, c)$ 
10:      else
11:         $K(i, c) \leftarrow \max[K(i - 1, c), v(i) + K(i - 1, c - w(i))]$ 
12:    $S \leftarrow \emptyset$ 
13:    $i \leftarrow n$ 
14:    $c \leftarrow C$ 
15:   while ( $i > 0$  and  $c > 0$ ) do
16:     if ( $K(i, c) = K(i - 1, c)$ ) then
17:        $i \leftarrow i - 1$ 
18:     else
19:        $S \leftarrow S \cup \{i\}$ 
20:        $i \leftarrow i - 1$ 
21:        $c \leftarrow c - w(i)$ 
22:   return  $S$ 
```

1.6 Running Time

- Doubly nested loop, so we have $O(n \times C)$, but this is not poly-time algorithm.

A poly-time algorithm means the algorithm is polynomial in input size (length of the list of items and how big are the numbers of value, weight, and capacity). This is because we have C as a part of our time complexity analysis.

Assume $v(i), w(i) \leq C$, or that the value, weight, capacity can be represented by the same amount of bits (dominated by C).

Then input size is $O(n \log C)$, because we need $\log C$ bits to store the number C . But C is exponentially bigger than $\log C$, so our algorithm is not polynomial-time in input-size. Ex. Our algorithm takes input size $O(n \log C)$ but requires $O(n \times C)$ time, which is not of polynomial difference.

- This type of algorithm is **pseudo-polynomial algorithm**, which means it is polynomial in input-value, but not input-size.

Then the limitations of this algorithm is:

If C is small enough, the algorithm is practical

If $C = 2^{64}$, then we would need a table with 2^{64} words, which is impractical. Not to mention computational time is going to be extremely slow. So even if we don't have huge numbers, this algorithm may still end up being impractical

1.7 Observations

- **Subset Sum Problem**

Given integers x_1, x_2, \dots, x_n and X

Want subset of x_1, \dots, x_n with max sum not exceeding X

A special case of Knapsack Problem where value of the items are equal to their weight

Still NP-Complete

- **Knapsack with repetition**

Given value and weights for n different items, but allow for copies of the same item.

Then we return a multi-set, because a set does not allow for repetition

2 DP and Recursion

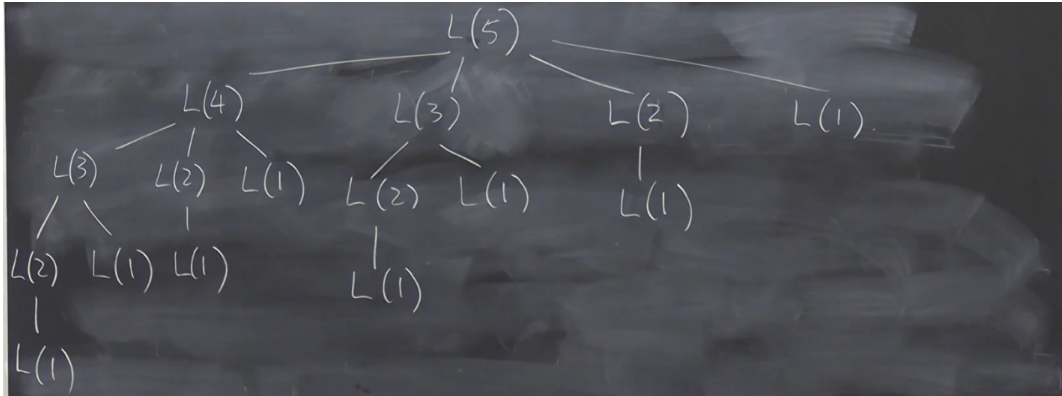
Why are our DP algorithms all iterative, when it seems natural to use recursion?

2.1 Analysis

- **LIS Problem:**

$$L(i) = 1 + \max\{L(j) : 1 \leq j < i \text{ \& } a_j < a_i\}$$

- Suppose we want $L(5)$, we would have the following dependencies:



Then the number of recursive calls will be $2^{5-1} = 16$, notice a lot of redundancies, we keep solving the same problem over and over again.

- (1) Note that the height of our recursive tree is now linear, as opposed to logarithmic in DC, since in DC, the recursive step shrinks the input by a constant factor, where as in DP, we shrink the input by a constant amount.
 - (2) Note that in DP we have a phenomenon called **overlapping subproblems**, which means that we solve problems that occur over and over again in different parts of the algorithm. In DC algorithms, once the problem is divided into parts, the parts do not have any shared subproblems. To increase efficiency when we have this phenomenon, we should only solve a problem once and reuse the solution.
- Solution: **Memoization (Caching)**: When recursive call to $P(i)$ is made, look in Table $T[i]$, if $T[i] = \text{nil}$, then execute the call, and set $T[i]$ to the result, otherwise simply return $T[i]$.

- **ED Problem:**

$$E(i, j) = \min\{ED(i-1, j-1) + \text{diff}(i, j), ED(i-1, j) + 1, ED(i, j-1) + 1\}$$

- A naive way (without memoization) of implementing this algorithm is:

```

1: procedure ED( $m, n$ ) ▷  $O(k^n)$ 
2:   if ( $m = 0$ ) then
3:     return  $n$ 
4:   else if ( $n = 0$ ) then
5:     return  $m$ 
6:   else
7:     return  $\min\{ED(m-1, n-1) + \text{diff}(m, n), ED(m-1, n) + 1, ED(m, n-1) + 1\}$ 

```

- The memoized way of implementing this algorithm is:

```

1: Init  $T[i, j] \leftarrow \text{nil}$  ▷ cache
1: procedure MED( $m, n$ ) ▷  $O(m \times n)$ 
2:   if ( $m = 0$ ) then
3:     return  $n$ 
4:   else if ( $n = 0$ ) then
5:     return  $m$ 
6:   else
7:     if ( $T[m, n] = \text{nil}$ ) then
8:        $T[m, n] \leftarrow \min\{\dots\}$ 
9:     return  $T[m, n]$ 

```

- **Knapsack Problem:**

$$K(i, c) = \max[K(i-1, c), v(i) + K(i-1, c - w(i))]$$

- Suppose we have the following items:

	W	V
1	100	
2	200	
3	300	

Then the only values of c we are concerned about are multiples of 100. All other subproblems with c values outside of our range can be computed, but will not be of any use when computing the final solution to our problem.

The recursive algorithm with memoization will only look at subproblems that it needs, and will not look at the unnecessary subproblems. Note recursive algorithm does this naturally.

3 Piano Fingering Problem

3.1 Introduction

Suppose we have

- F = set of fingers $1, 2, \dots, k$
- N = set of notes a, b, \dots, h

- **Input:**

- $S[1..n]$ sequence of notes
- $d(f, t, f', t')$ is the difficulty of going from finger f hitting note t to finger f' hitting note t'

- **Output:**

- $G[1..n]$ sequence of fingers that minimizes $\sum_{1 \leq i < n} d(G[i], S[i], G[i+1], S[i+1])$, or the optimal fingering of the score

3.2 Subproblem

Compute prefixes $G[1..i]$, $1 \leq i \leq n$, but as usual, we focus on the cost of optimal G

$$c(i) = \min_{f \in F} [c(i-1)] + \text{cost of last transition to } f$$

So compute $c(i-1)$ for all fingers $f \in F$

- Then we have that the subproblems are: $c(i, f)$ = cost of optimal fingering of $S[1..i]$ ending with finger f (\star)

$$c(i, f) = \min_{f' \in F} \{c(i-1, f') + d(f', S[i-1], f, S[i])\} \quad (\dagger)$$

So compute $c(i-1)$ for all fingers $f' \in F$

3.3 Homework

Try and design this algorithm, and perhaps map out the recurrence for (\dagger) to see a shortest path.