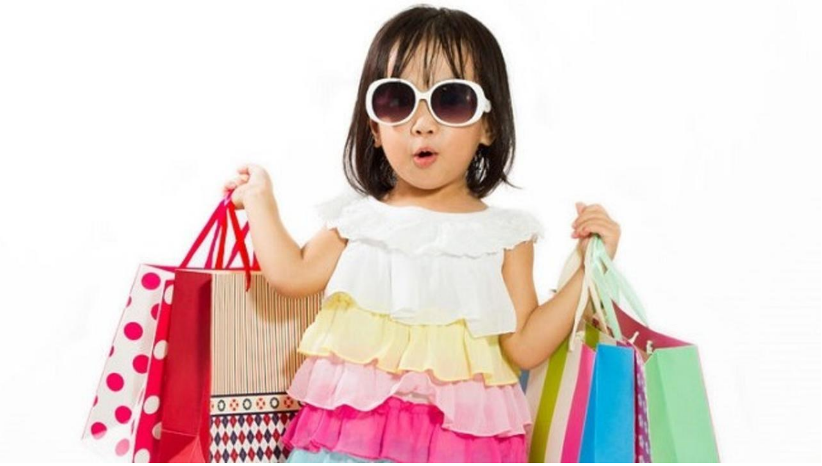
Dynamic programming (knapsack problem)

Reading: Kleinberg & Tardos
Ch. 6
CLRS Ch. 15

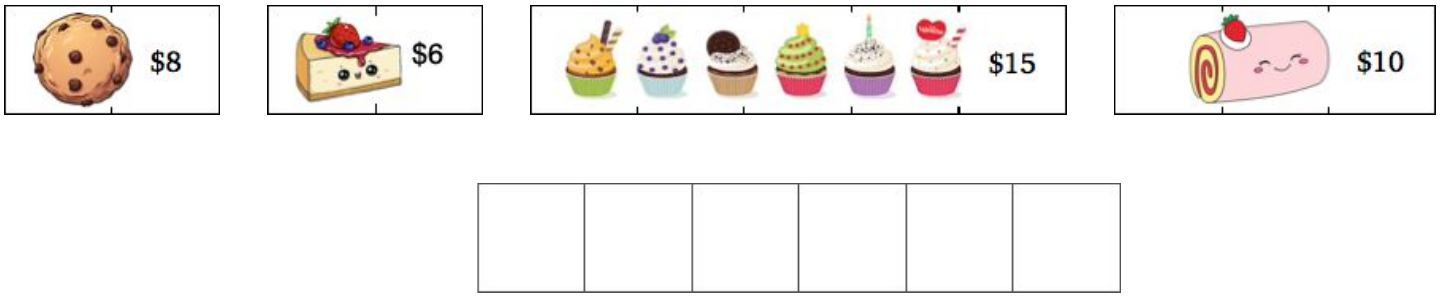
Shopping spree

You just won a contest for a shopping spree at your favorite store! You've been given a knapsack and 15 minutes to run through the store and fill it with whatever you want. It's your favorite store, so you know the prices and sizes of every item (of course).



What is your strategy to maximize the value of your contest loot?

Gaining intuition



https://mhc-algorithms.github.io/KnapsackProblem.html

0-1 Knapsack Problem

- Given capacity W, arrays V and S for values and sizes

  (ignore index 0)

- Maximize $\sum_{i=1}^{n} V[i]x_i$ subject to $\sum_{i=1}^{n} S[i]x_i \leq W$

0-1 Knapsack Problem

- Given capacity W, arrays V and S for values and sizes (ignore index 0)
- Maximize $\sum_{i=1}^{n} V[i]x_i$ subject to $\sum_{i=1}^{n} S[i]x_i \leq W$
- 0-1: x_i is 0 or 1 (in the bag or not in the bag)

0-1 Knapsack Problem

- Given capacity W, arrays V and S for values and sizes

  (ignore index 0)

- Maximize $\sum_{i=1}^{n} V[i]x_i$ subject to $\sum_{i=1}^{n} S[i]x_i \leq W$

- 0-1: x_i is 0 or 1 (in the bag or not in the bag)
- Fractional knapsack can be solved by greedy
  - Determine value density V[*i*]/S[*i*] for each item, fill greedily

https://www.maxi-muth.de/files/knapsack/

0-1 Knapsack

- Notice **optimal substructure**
  - Answers to subproblems give answers without needing refinement
- Notice **overlapping subproblems**
  - Redoing work is expensive!
- How can we solve this?
  - "memoization"

Knapsack: recursive

Recursively consider first *i* items; find max value knapsack can hold.

Assume V value array and S size a… are

```
knapsack( W ):
    return knapsack( n, W )
knapsack( i, W ):
    if ( i == 0 ):   return 0        // base ca
    else:                            // recursi
        if S[i] > W:                 // too big to fit in (empty) knapsack
            return knapsack( i-1, W )
        else:
            return max(
                V[i] + knapsack( i-1, W - S[i] ), // with item i
                knapsack( i-1, W ) ) // without item i
```

- Arbitrarily order the items.
- Why does particular order not matter?
- Either item is in optimal solution or not, but *when* it is added does not impact optimal solution.

0-1 Knapsack

- Notice **optimal substructure**
  - Answers to subproblems give answers without needing refinement
- Notice **overlapping subproblems**
  - Redoing work is expensive!
- How can we solve this?
  - “memoization”

0-1 Knapsack

- Notice **optimal substructure**
  - Answers to subproblems give answers without needing refinement
- Notice **overlapping subproblems**
  - Redoing work is expensive!
- How can we solve this?
  - “memoization”

Knapsack: recursive with memoization

Assume V value array and S size array are global variables

```
M = new int[n+1][W+1]          // M[i][w]: max value of capacity w for first i items
assign M[i][w] = -1 for all i,w  // M[i][w]: -1 not yet computed
knapsack( W ): return knapsack( n, W )

knapsack( i, W ):
    if M[i][W] != -1: return M[i][W]// M[i][W] already computed
    if ( i == 0 ):    M[i][W] = 0    // base case: no items
    else:                            // recursive case:
        if S[i] > W:                 // too big to fit in (empty) knapsack
            M[i][W] = knapsack( i-1, W )
        else:
            M[i][W] = max(
                V[i] + knapsack( i-1, W - S[i] ), // with item i
                knapsack( i-1, W ) ) // without item i
    return M[i][W]
```

Knapsack: bottom up with memoization

Assume V value array and S size array are global variables

```
knapsack( W ):
    M = new int[n+1][W+1]            // M[i][w]: max value capacity w & first i items
    for w = 0 to W: M[0][w] = 0      // base case: no items
    for i from 1 to n:               // for each item to additionally consider
        for w from 0 to W:               // for each capacity
            if S[i] > w:                 // too big to fit in (empty) knapsack
                M[i][w] = M[i-1][w]     // max value is same w, one less item
            else:
                M[i][w] = max(
                    V[i] + M[i-1][w - S[i]], // with item i
                    M[i-1][w] ) // without item i
    return M[n][W]
```

Knapsack: recovering solution

- For item *i* and capacity w
  - Item *i* is in optimal solution if M[*i*][w] > M[*i*-1][w]
- Start at M[n][W] and trace back
  - If item in solution, iterate on M[*i*-1][w-S[*i*]]
  - Otherwise, iterate on M[*i*-1][w]

Val	Wt	Item	Max Weight 0	1	2	3	4	5	6	7
0	0	0	0	0	0	0	0	0	0	0
1	1	1	0	1	1	1	1	1	1	1
4	3	2	0	1	1	4	5	5	5	5
5	4	3	0	1	1	4	5	6	6	9
7	5	4	0	1	1	4	5	7	8	9

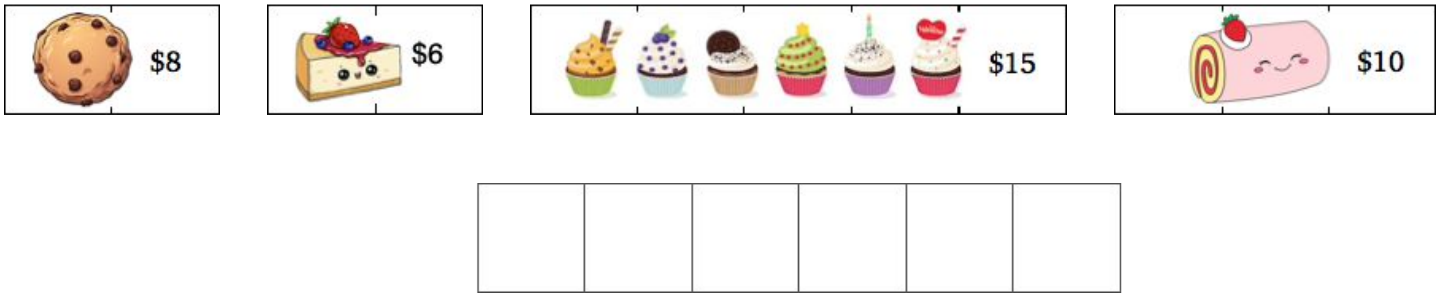
Knapsack: recovering solution

- For item *i* and capacity w
  - Item *i* is in optimal solution if M[*i*][w] > M[*i*-1][w]
- Start at M[n][W] and trace back
  - If item in solution, iterate on M[*i*-1][w-S[*i*]]
  - Otherwise, iterate on M[*i*-1][w]
- Pseudocode:

```
initialize i = n, w = W
includedItems = {}
while ( i > 0 ):          // for each item
    if M[i][w] > M[i-1][w]:  // included?
        includedItems.add( i )
        w = w - S[i] // update remaining capacity
    i--                       // “next” (previous) item
```

Val	Wt	Item	Max Weight 0	1	2	3	4	5	6	7
0	0	0	0	0	0	0	0	0	0	0
1	1	1	0	1	1	1	1	1	1	1
4	3	2	0	1	1	4	5	5	5	5
5	4	3	0	1	1	4	5	6	6	9
7	5	4	0	1	1	4	5	7	8	9

Gaining intuition



https://mhc-algorithms.github.io//KnapsackProblem.html

Running time

- $T(n,W) = \Theta(nW)$
  - Polynomial in input size?
    - n items $\rightarrow \Theta(n)$ bits
    - numerical value W $\rightarrow$??

Running time

- $T(n,W) = \Theta(nW)$
  - Polynomial in input size?
    - n items $\rightarrow \Theta(n)$ bits
    - numerical value W $\rightarrow \log_2 W$ bits
- $T(N) = \Theta(n2^{|W|})$
  - N: size of input (in bits)
- *Pseudo-polynomial*
  - Polynomial in input (non-numerical and numerical **values**)
    - NOT polynomial in representation of input
      - Numerical values compressed logarithmically