Optimization Problems

Introduction

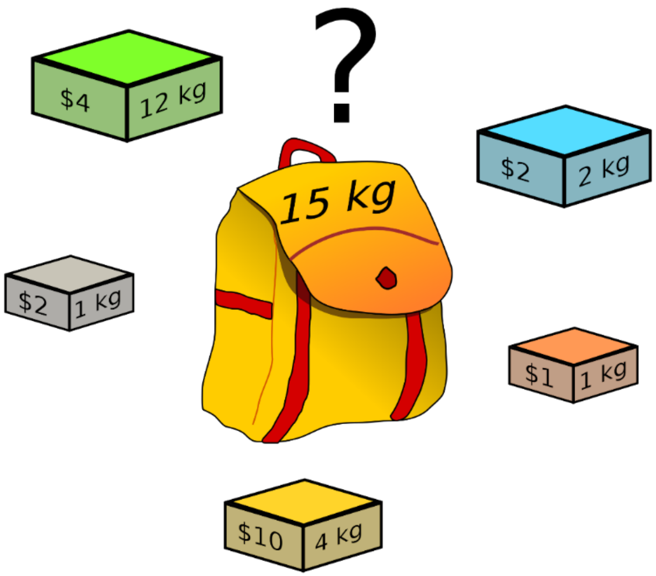
- An ***optimization problem*** is a problem of finding an optimal (biggest, smallest, best in some sense, etc.) solution among those in a set of candidate solutions.
- This usually involves finding a target configuration (ordering, subset, partition, parameter values, etc.) of/for a finite set of input objects.
- An optimization problem consists of two parts:
  1. An ***objective function*** of the input that we want to maximize or minimize
  2. A set of ***constraints*** that limits the search space (i.e., the set of ***feasible solutions***)
- Usually, an exhaustive search is prohibitively expensive.

Examples

- *Traveling salesman:* Find the shortest route that visits each point from a set exactly one.
- *Minimum spanning tree:* Find the cheapest way to connect a set of terminals.
- *Activity selection:* Schedule a maximum number of compatible activities requesting the same resource.
- *Clique:* Given a social network, find the largest subset whose members know every other member in the subset.
- *Knapsack:* Given a set of potential investments, find those that maximize the return for a given budget.
- *Clustering:* Given a set S of points in $\mathbb{R}^d$ and $k \in \mathbb{N}$, partition S into k sets such the minimum distance between points in different sets is maximized.

Example: The Knapsack Problem

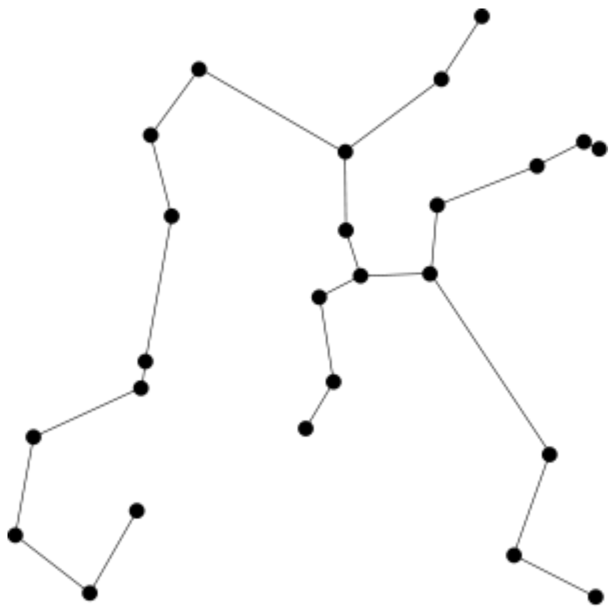
- You are given a container with a limited weight capacity W and a list of items, each with a weight and a value. Choose which items to place in the container so that the weight limit is not exceeded and the total value of the packed items is as large as possible.



- A fund manager is considering 100 potential investments and has estimated the expected return from each one. Choose which ones to buy to maximize the return without exceeding the budget.

Example: Minimum Spanning Tree

- Given a set of points in the plane, connect pairs of points with edges so that the sum of lengths of all edges is minimal and there is a path, using the edges, from every point to every other point.



Example: Activity Selection

Input: set $A = \{a_1, \ldots, a_n\}$ of n activities/events requiring *exclusive* access to a common resource

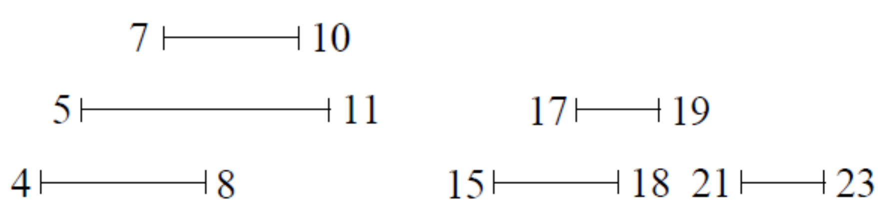
Starts at

$$a_i = [s_i, f_i)$$

Finishes at

Output: the largest set A of nonoverlapping activities

Example: schedule use of a room to maximize the number of events that use it



Example: Clustering

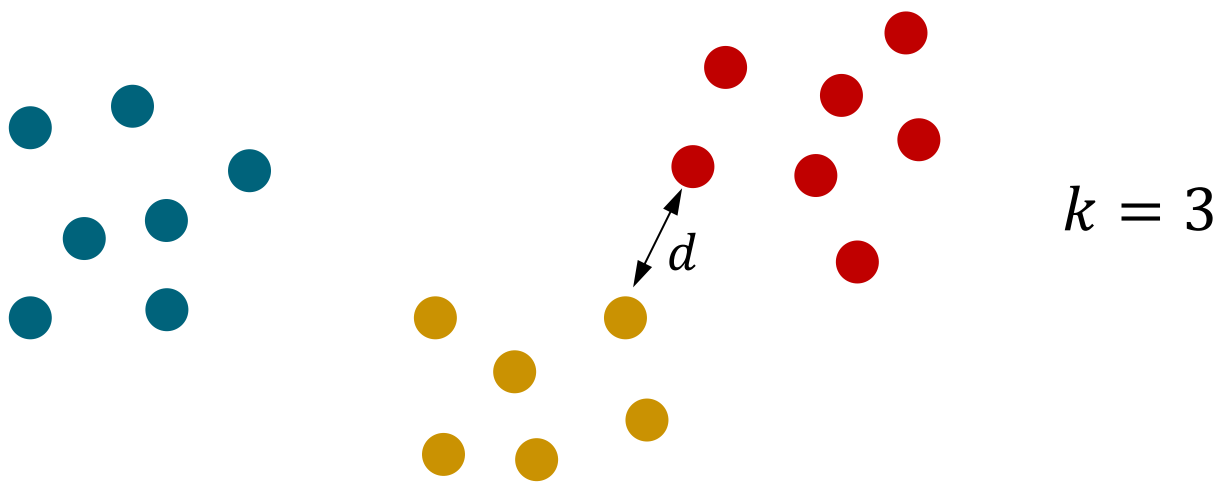
Input: set $X = \{x_1, \ldots, x_n\}$ of d-dimensional feature vectors (points in $\mathbb{R}^d$) and positive integer $k \geq 2$

Output: a partition $X_1, X_2, \ldots, X_k$ of X such that:

1. $\bigcup_{i=1}^{k} X_i = X$

2. $X_i \cap X_j = \emptyset$, for $i \neq j$

3. The smallest distance d between points in different sets is maximized



END

Introduction to Dynamic Programming

Introduction to Dynamic Programming

- Algorithm design technique
  - Many apparently exponential optimization problems have polynomial solutions using DP.
- Has been described as *divide and conquer with memory*
- Title refers not to computer programming, but to the process of gradually (i.e., dynamically) filling a table of partial results in a systematic way

Example 1: Divide and Conquer

1 1 2 3 5 8 13 . . . .

- **Problem:** Given i, find the ith Fibonacci number.

30k 20k

15k 13k

29,999

29,998

```
def fib(n):
    if n<2:
        return n
    else:
        return fib(n-2) + fib(n-1)
```

- Time? $T(n) = T(n-1) + T(n-2) + 1$

$$\geq 2T(n-2) + 1 \geq 2^{n/2}$$

DP Hallmarks

1. *Optimal substructure:* An optimal solution to a problem instance is made up of optimal solutions to subproblem instances.
   - This suggests the possibility of DAC.
2. *Overlapping subproblems:* A recursive solution contains a *small number* of distinct problem instances repeated *many* times.
   - This suggests storing solutions to subproblems in case they are needed later.

Solution 2: Memo(r)ization

- Can speed up the algorithm by storing the results of our recursive calls in a “memo” and reusing them when needed again

```
memo = dict()

def fib(n):
      if n<2:
            return n
      elif memo.get(n) != None: return 
memo[n]
      else:
            memo[n] = fib(n-2) + fib(n-1)
            return memo[n]
```

Time? $T(n) = T(n-1) + 1 \in O(n)$

Solution 3: Dynamic Programming

- Can get a simpler algorithm with the same performance as solution 2 by proceeding bottom-up and recording previous solutions in a list

```
def fib(n):
    if n<2:
        return n
    fibs = [0,1]
    for i in range(2,n+1):
        next = fibs[-1]+fibs[-2]
        fibs.append(next)
    return next
```

END

Time?

$$T(n) \in O(n)$$

NCoins

The Problem

How does one make change for N cents?

- Example: N = 29
  - 25, 1, 1, 1, 1

- But there are other ways of doing it
  - 10, 10, 5, 1, 1, 1, 1
  - 5, 5, 5, 5, 5, 1, 1, 1, 1
  - 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1…………., 1

Optimization Problems

There are multiple solutions to a problem.

- Select the “optimal” solution
- For NCoins: the least number of coins
  - 25, 1, 1, 1, 1
- This is an example of a greedy algorithm

NCoins Generalized

Add a coin to the set: 25, **12**, 10, 5, 1.

- Our "greedy" approach gives (N = 29).
  - 25, 1, 1, 1, 5
    - ~~Four~~ 5 coins
- But this is no longer the optimal solution.
  - 12, 12, 5
    - Three coins

Divide-and-Conquer Solution

- Hand each coin one at a time (N = 29).
- At each step, there are five coin choices:
  1. Give a 25 → leaving 4 cents to give
  2. Give a 12 → leaving 17 cents to give
  3. Give a 10 → leaving 19 cents to give
  4. Give a 5 → leaving 24 cents to give
  5. Give a 1 → leaving 28 cents to give
- Select the choice giving the least coins.

Repeated Work

29 → 4 (25), 17 (12), 19 (10), 24 (5), 28 (1)

24 → X (25), 12 (12), 14 (10), 19 (5), **23 (1)**

28 → 3 (25), 16 (12), 18 (10), **23 (5)**, 27 (1)

- We get to 23 cents change to go if we:
  - Give a 5 and then a 1
  - Give a 1 and then a 5
  - Order doesn't matter

Speed-up Options

- Memoization (memorize with memory)
  - Top-down
  - Cache answers
- Dynamic programming
  - Bottom-up
  - Calculate the answers to the subproblems first

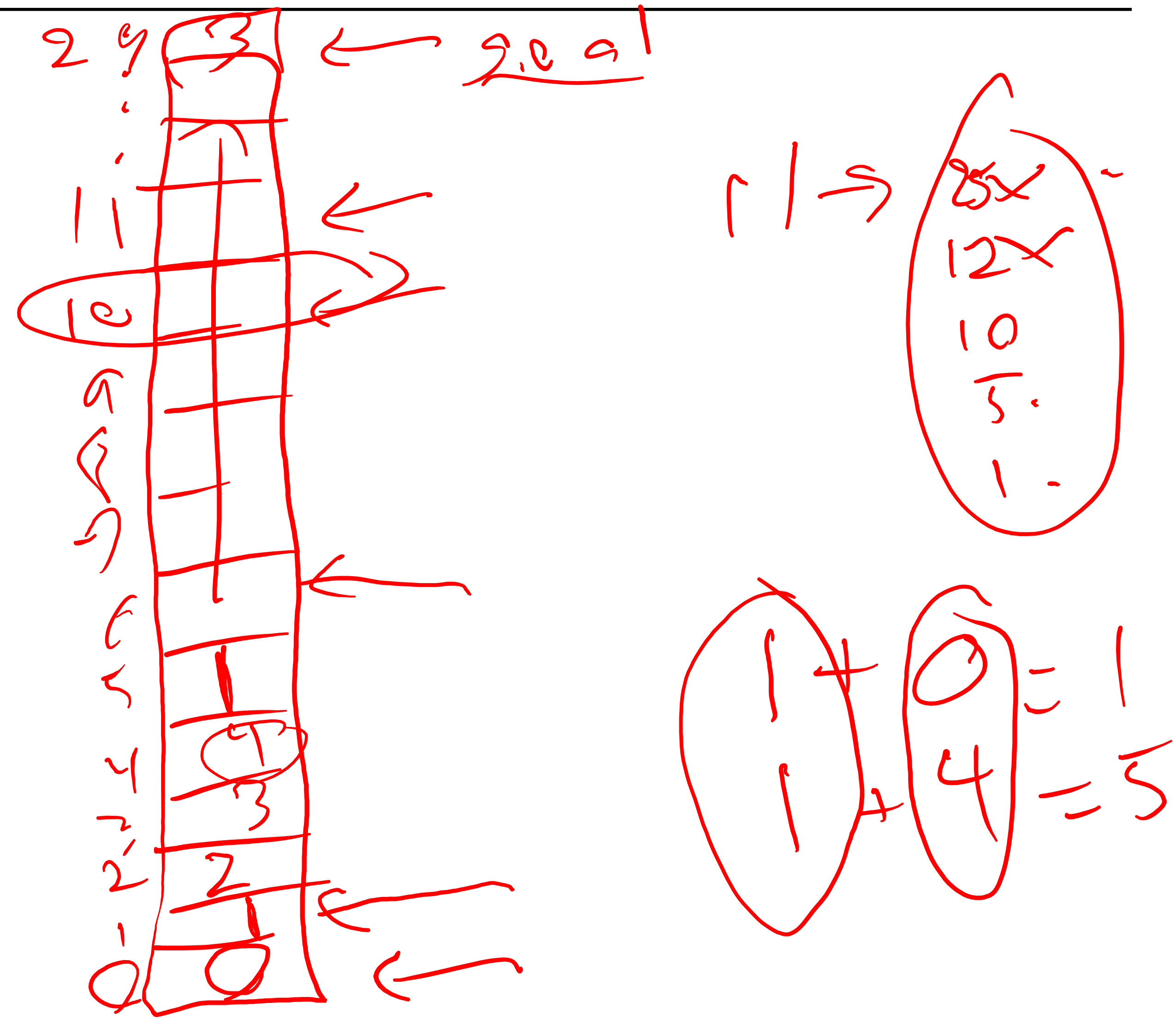
END

NCoins
Dynamic Programming

NCoins Dynamic Programming

- NCoins is a problem with one variable (N).
  - Will require a 1D table
- Fill in the table with the known base cases.
- Identify the goal location in the table.
- Determine the order to fill in the table.
  - Each subgoal must already be filled in

NCoins Table Construction



NCoins: Two Tables

- However, this minCoins table only gives the minimum number of coins needed.
  - The thing being optimized
- We usually also want the actual coins used.
  - So we can make the change
- Build a second “winner” or “traceback” table.

NCoins Winner Table Construction

6	2	6	1
5	1	5	5
4	4	4	1
3	3	3	1
2	2	2	1
1	1	1	1
0	0	0	0

1¢ $1 + 4 = 5$ ✓

5¢ $1 + 0 = 1$ ✓

1¢ : $1 + 1 = 2$

5¢ : $1 + 1 = 2$

1, 5

NCoins Table Traceback

- Can reconstruct the coins used by tracing backwards through the table of winners
  - Sometimes called the traceback table
- Note that the traceback table is built during the construction of the optimal table
  - But never used to determine values in the optimal table

END

Acknowledgements

- Wikipedia.org