# Knapsack Problem

A set of items, each has different size and different value.

We only have one knapsack.

Goal: to pick a subset which can fit into the knapsack and **maximize the value** of this subset.

# Knapsack Problem

**(The Knapsack Problem)** *Given a set* $S = \{a_1, \ldots, a_n\}$ of objects, with specified sizes and profits, size($a_i$) and profit($a_i$), and a knapsack capacity B, find a subset of objects whose total size is bounded by B and total profit is maximized.

Assume size($a_i$), profit($a_i$), and B are all integers.

We'll design an approximation scheme for the knapsack problem.

# Knapsack problem

- Fractional knapsack problem
  - P
- 0/1 knapsack problem
  - NP-Complete
  - Approximation
    - PTAS

# Fractional knapsack problem

- n objects, each with a weight $w_i > 0$

  a profit $p_i > 0$

  capacity of knapsack: M

Maximize $\sum_{1 \le i \le n} p_i x_i$

Subject to $\sum_{1 \le i \le n} w_i x_i \le M$

$0 \le x_i \le 1, 1 \le i \le n$

# The knapsack algorithm

- The greedy algorithm:
  Step 1: Sort $p_i/w_i$ into non-increasing order.
  Step 2: Put the objects into the knapsack according to the sorted sequence as possible as we can.
- e. g.
  $n = 3$, $M = 20$, $(p_1, p_2, p_3) = (25, 24, 15)$
  $(w_1, w_2, w_3) = (18, 15, 10)$
  Sol: $p_1/w_1 = 25/18 = 1.32$
  $p_2/w_2 = 24/15 = 1.6$
  $p_3/w_3 = 15/10 = 1.5$
  Optimal solution: $x_1 = 0$, $x_2 = 1$, $x_3 = 1/2$

# 0/1 knapsack problem

- **Def:** n objects, each with a weight $w_i > 0$

  a profit $p_i > 0$

  capacity of knapsack : M

  Maximize $\sum_{1 \le i \le n} p_i x_i$

  Subject to $\sum_{1 \le i \le n} w_i x_i \le M$

  $x_i = 0$ or $1$, $1 \le i \le n$
- Decision version :

  Given K, $\exists \sum p_i x_i \ge K$ ?

- Knapsack problem : $0 \le x_i \le 1$, $1 \le i \le n$.

# Polynomial Time Approximation Scheme (PTAS)

We have seen the definition of a constant factor approximation algorithm.

The following is something even better.

An algorithm ***A*** is an **approximation scheme** if for every $\epsilon > 0$,
***A*** runs in polynomial time (which may depend on $\epsilon$) and return a solution:

- $SOL \leq (1+\epsilon)OPT$ for a minimization problem
- $SOL \geq (1-\epsilon)OPT$ for a maximization problem

For example, ***A*** may run in time $n^{100/\epsilon}$.

There is a time-accuracy tradeoff.

# Greedy Methods

**General greedy method**:
Sort the objects by some rule,
and then put the objects into the knapsack according to this order.

Sort by object size in non-decreasing order:

Sort by profit in non-increasing order:

Sort by profit/object size in non-increasing order:

Greedy won't work.

# Dynamic Programming for Knapsack

Suppose we have considered object 1 to object i.
We want to remember what profits are achievable.
For each achievable profit, we want to minimize the size.

Let S(i,p) denote a subset of {a1,…,ai} whose total profit is **exactly** p and total size is **minimized**.

Let A(i,p) denote the size of the set S(i,p)

(A(i,p) = $\infty$ if no such set exists).

For example, A(1,p) = size(a1) if p=profit(a1),
Otherwise A(1,p) = $\infty$ (if p $\neq$ profit(a1)).

# Recurrence Formula

Remember: $A(i,p)$ denote the minimize size to achieve profit p using objects from 1 to i.

How to compute $A(i+1,p)$ if we know $A(i,q)$ for all q?

Idea: we either choose object i+1 or not.

If we do not choose object i+1: then $A(i+1,p) = A(i,p)$.

If we choose object i+1:

then $A(i+1,p) = size(a_{i+1}) + A(i,p\text{-}profit(a_{i+1}))$ if $p > profit(a_{i+1})$.

$A(i+1,p)$ is the minimum of these two values.

# Running Time

The input has $2n$ numbers, say each is at most $P$.

So the input has total length $2n\log(P)$.

For the dynamic programming algorithm,

there are $n$ rows and at most $nP$ columns.

Each entry can be computed in constant time (look up two entries).

So the total time complexity is $O(n^2P)$.

The running time is not polynomial if $P$ is very large (compared to $n$).

# Approximation Algorithm

We know that the knapsack problem is NP-complete.

Can we use the dynamic programming technique to design approximation algorithm?

# Scaling Down

**Idea**: to scale down the numbers and
compute the optimal solution in this modified instance

- Suppose $P \geq 1000n$.
- Then $OPT \geq 1000n$.
- Now scale down each element by 100 times (profit*:=profit/100).
- Compute the optimal solution using this new profit.
- Can't distinguish between element of size, say 2199 and 2100.
- Each element contributes at most an error of 100.
- So total error is at most 100n.
- However, the running time is 100 times faster.

# Approximation Scheme

Goal: to find a solution which is at least $(1-\epsilon)$OPT for any $\epsilon > 0$.

**Approximation Scheme for Knapsack**

1. Given $\epsilon > 0$, let $K = \epsilon P/n$, where P is the largest profit of an object.
2. For each object ai, define profit*(ai) = $\lfloor \text{profit}(ai)/K \rfloor$.
3. With these as profits of objects, using the dynamic programming algorithm, find the most profitable set, say S' .
4. Output S' as the approximate solution.

# Quality of Solution

**Theorem.** Let S denote the set returned by the algorithm. Then,
$\text{profit}(S) \geq (1-\epsilon)OPT$.

**Proof.**

- Let O denote the optimal set.
- For each object a, because of rounding down,
  $K \cdot \text{profit}^*(a)$ can be smaller than profit(a), but by not more than K.
- Since there are at most n objects in O,

$$\text{profit}(O) - K \cdot \text{profit}^*(O) \leq nK.$$

- Since the algorithm return an optimal solution under the new profits,

$$\text{profit}(S) \geq K \cdot \text{profit}^*(S) \geq K \cdot \text{profit}^*(O) \geq \text{profit}(O) - nK$$
$$= OPT - \epsilon P \geq (1-\epsilon)OPT$$

because $OPT \geq P$.

# Running Time

For the dynamic programming algorithm,

there are n rows and at most $n\lfloor P/K \rfloor$ columns.

Each entry can be computed in constant time (look up two entries).

So the total time complexity is $O(n^2\lfloor P/K \rfloor) = O(n^3/\epsilon)$.

Therefore, we have an approximation scheme for Knapsack.

# Approximation Scheme

**Quick Summary**

1. Modify the instance by rounding the numbers.
2. Use dynamic programming to compute an optimal solution S in the modified instance.
3. Output S as the approximate solution.