4 Greedy, Dynamic Programming

Greedy

- heuristic algorithm for optimisation problems
- always picks local optimum
- *hope* to find global optimum

Knapsack (Fractional)

w1, p1	w2, p2	w3, p3
w4, p4	w5, p5	w6, p6

maxWeight = 20

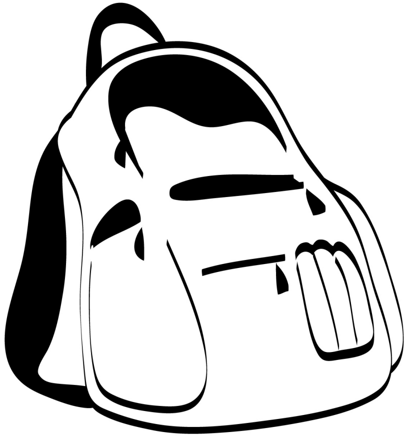
w_i - weights, p_i - profits

Objective:

Maximise profit subject to maxWeight constraint.

Knapsack (Fractional)

w1, p1	w2, p2	w3, p3
w4, p4	w5, p5	w6, p6



maxWeight = 20

w_i - weights, p_i - profits

Objective:

Maximise profit subject to maxWeight constraint.

Can use a **fraction** of an object.

Find the Solution

- “always picks *local optimum*”
    - best solution to a *subproblem*

Find the Solution

- “always picks *local optimum*”
  - best solution to a *subproblem*

- KNAP-FR ⇒ subproblem is:
  - “Maximise profit for ***one*** unit of the backpack”

Find the Solution

- “always picks *local optimum*”
  - best solution to a *subproblem*

- KNAP-FR ⇒ subproblem is:
  - “Maximise profit for ***one*** unit of the backpack”

- The choice ⇒ item with *best value per unit*

Local Optimum ⇒ Global Optimum?

- This **must hold**
  - otherwise greedy solution not appropriate

- Check
  - Dantzig, George B. "Discrete-variable extremum problems."
  - *Combinatorial Optimization: Theory and Algorithms*, Algorithms and Combinatorics

Algorithm

1. Sort the inputs based on value/weight

2. Pick best unused until backpack is full

3. Fill remaining with a fraction

Intermezzo - Sorting in Java

```
class Item {double ratio, …;}

List<Item> items = ….

Collections.sort(items);
```

How does Java know to sort Items?

Intermezzo - Sorting in Java

1. *Natural* order:

```
class Item implements Comparable<Item> {
 double ratio, …;
 public int compareTo(Item otherItem) {..}
}
```

Intermezzo - Sorting in Java

2. *Custom comparator*:

```
ItemComp comparator = new ItemComp();
Collections.sort(items, comparator);

class ItemComp implements Comparator<Item>{
  public int compare(Item i1, Item i2) {}
}
```

Intermezzo - Sorting in Java

2. *Custom comparator* (anonymous class):

```
Collections.sort(items,
  new Comparator<Item>() {
    public int compare(Item i1, Item i2) {}
  }
);
```

Intermezzo - Sorting in Java

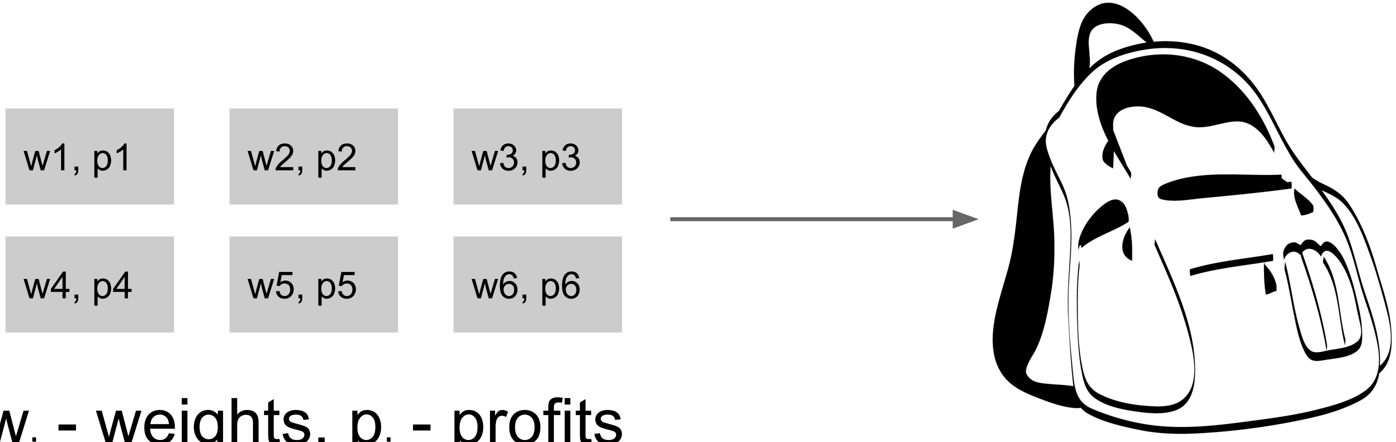
3. *Using lambdas* (JDK 1.8-ea):

```
Collections.sort(items,
  (item1, item2) -> {return …;}
);
```

Greedy Summary

- May not produce optimal results
  - Prove local optimum ⇒ global optimum
- Fast and not too hard to code
- Some more problems [12405](), [10026](), [10037]()

Knapsack (Discrete)



w_i - weights, p_i - profits

maxWeight = 20

Objective:

Maximise profit subject to maxWeight constraint.

Cannot use a **fraction** of an object.

Knapsack (Discrete)

- “**Cannot** use a **fraction** of an object.”
    - This breaks our greedy solution

Knapsack (Discrete)

- "**Cannot** use a **fraction** of an object."
  - This breaks our greedy solution
  - Counter-example

maxWeight = 5

w = 4, p = 10, r = 2.5

w = 3, p = 7, r = 2.33

w = 2, p = 4, r =2

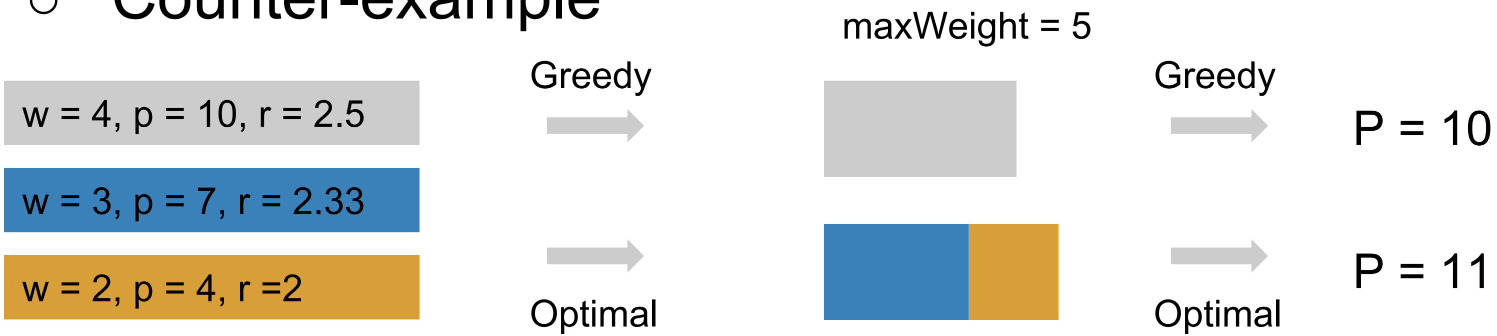
Greedy

Greedy

P = 10

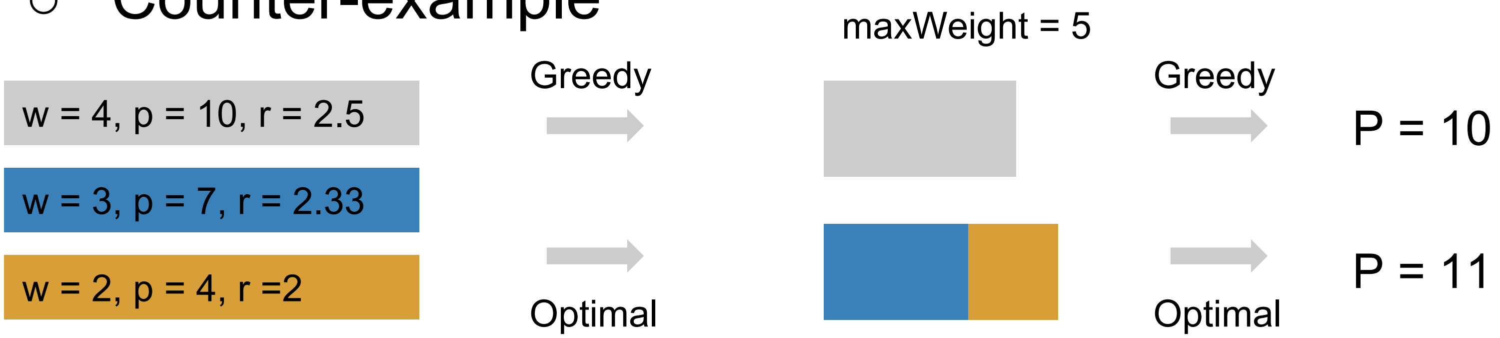
Knapsack (Discrete)

- “**Cannot** use a **fraction** of an object.”
  - This breaks our greedy solution
  - Counter-example



Knapsack (Discrete)

- "**Cannot** use a **fraction** of an object."
  - This breaks our greedy solution
  - Counter-example



- Seems like we have to use complete search

Knapsack (Discrete)

- Recurrence relation
  - best(i, w) = best profit using the first *i* items, up to a maximum weight of w

Knapsack (Discrete)

- Recurrence relation
  - best(i, w) = best profit using the first *i* items, up to a maximum weight of w

⇒ best(0, *), best(*, 0) = 0

Knapsack (Discrete)

- Recurrence relation
  - best(i, maxW) = best profit using the first *i* items, up to a maximum weight of maxW

⇒ best(0, *), best(*, 0) = 0

⇒ best(n - 1, maxWeight) - the maximum profit

Knapsack (Discrete)

best(i, maxW) =

max(best(i - 1, maxW), best(i - 1, maxW - w[i]) + p[i]))

1. Don’t pick item *i*
2. Same weight limit
3. Same best profit

Knapsack (Discrete)

best(i, maxW) =

max(best(i - 1, maxW), best(i - 1, maxW - w[i]) + p[i]))

1. Don’t pick item *i*
2. Same weight limit
3. Same best profit

1. Pick item *i*
2. Update remaining capacity
3. Add its profit

Knapsack (Discrete)

best(i, maxW) =

max(best(i - 1, maxW), best(i - 1, maxW - w[i]) + p[i]))

1. Don't pick item *i*
2. Same weight limit
3. Same best profit

1. Pick item *i*
2. Update remaining capacity
3. Add its profit

Two options: use/don't use item i

Pick the best!

Overlapping Subproblems

- There seem to be many *overlapping* subproblems

Overlapping Subproblems

- There seem to be many *overlapping* subproblems

- **Dynamic Programming**
  - solve each subproblem only once

Overlapping Subproblems

- There seem to be many *overlapping* subproblems

- **Dynamic Programming**
  - solve each subproblem only once
  - store results to subproblems (*memoization*)

Overlapping Subproblems

- There seem to be many *overlapping* subproblems

- **Dynamic Programming**
  - solve each subproblem only once
  - store results to subproblems (*memoization*)
  - reuse results without recomputation

Top Down vs Bottom Up

- Top Down approach
  - start with best(n - 1, maxWeight)
  - recurse to solve subproblems

- Bottom up
  - start with the smaller problems
  - build-up to solution

Summary

When Complete Search is too slow:

- Greedy
  - when local optimum ⇒ global optimum
- Dynamic programming
  - remove repeating/overlapping subproblems

Maximum Sum [UVA] [git]

Given a matrix of integers, find the sub-matrix with the maximum sum.

4 0 -2 -7 0 9 2 -6 2 -4 1 -4 1 -1 8 0 -2	15

Maximum Sum - Complete Search

Complete search in O(n^6):

- Iterate through every starting point O(n^2)
  - Iterate through every possible length O(n^2)
    - Sum up the numbers O(n^2)
- This is too slow for n ~ 100

Maximum Sum - Faster

- Observation: there are quite a few redundant computations (e.g. sum of square in column 1 is recomputed in col 2 and again in col 3)

```
4
 0 -2 -7  0
 9  2 -6  2
-4  1 -4  1
-1  8  0 -2
```

```
4
 0 -2 -7  0
 9  2 -6  2
-4  1 -4  1
-1  8  0 -2
```

```
4
 0 -2 -7  0
 9  2 -6  2
-4  1 -4  1
-1  8  0 -2
```

Maximum Sum - Faster

Could speed up by avoiding recomputation.
Idea: Precompute sums of all submatrices starting from (0, 0)

0 -2 -7 0 9 2 -6 2 -4 1 -4 1 -1 8 0 -2	0 -2 -9 -9 9 9 -4 -2 5 6 -11 -8 4 13 -4 -3

Maximum Sum - Faster

Can reconstruct sum of arbitrary sub-matrix:

0	-2	-9	-9
9	9	-4	-2
5	6	-11	-8
4	13	-4	-3

Sub_Sum(2,2,3,3) =
Sum(2, 2, 3, 3)
Sum(0, 0, 1, 3) +
Sum(0, 0, 3, 1) -
Sum(0, 0, 1, 1)

```
Maximum Sum
  Precompute        // O(n^2)
  for (i, j)        // starting positions
    for (k, l)      // end position
      sub_sum = sum[k][l] +
                sum[i-1][l] +
                sum[k][j-1] -
                sum[i-1][j-1]
```

Total Complexity = O(n^2 + n^4) - Much Faster!