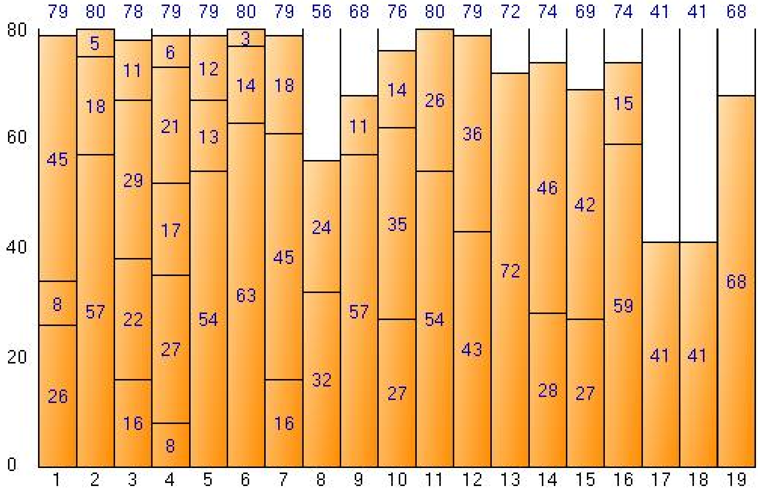
Mathematical Optimization: a Computational Primer

João Pedro Pedroso

August 2018

Outline

Bin packing and cutting stock problems



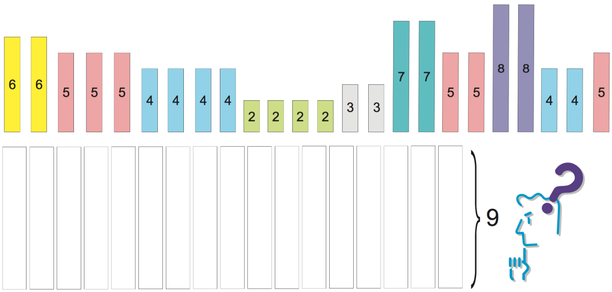
Bin packing

Case study:

> *You are the person in charge of packing in a large company. Your job is to skillfully pack items of various weights in a box with a predetermined capacity; your aim is to use as few boxes as possible. Each of the items has a known weight, and the upper limit of the contents that can be packed in a box is 9 kg. The weight list of items to pack is given below. In addition, the items you are dealing with your company are heavy; there is no concern with the volume they occupy. So, how should these items be packed?*

Weights of items to be packed in bins of size 9.

Weights of items to be packed
6, 6, 5, 5, 5, 4, 4, 4, 4, 2, 2, 2, 2, 3, 3, 7, 7, 5, 5, 8, 8, 4, 4, 5



Weights of items to be packed in bins of size 9.

Weights of items to be packed
6, 6, 5, 5, 5, 4, 4, 4, 4, 2, 2, 2, 2, 3, 3, 7, 7, 5, 5, 8, 8, 4, 4, 5

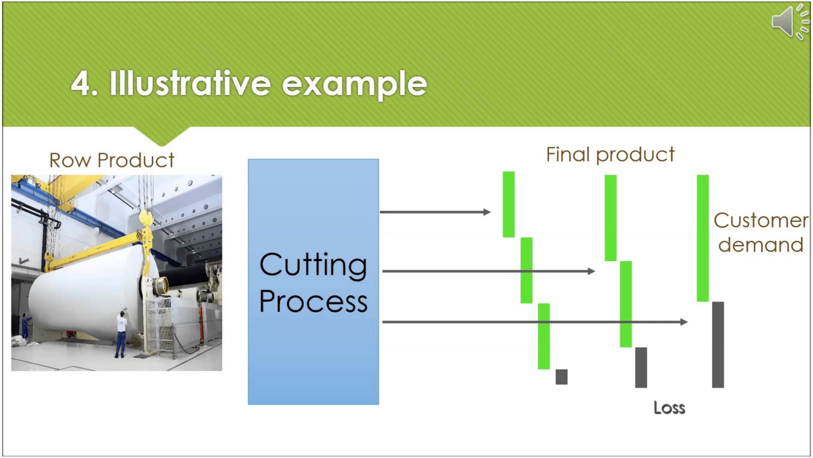
Bin packing problem

- There are n items to be packed and an infinite number of available bins of size B
- Sizes $0 \leq s_i \leq B$ of individual items are known
- Problem: determine how to pack these n items in bins of size B so that the number of required bins is minimum.

Challenge

Try to solve the previous instance

Cutting stock problem



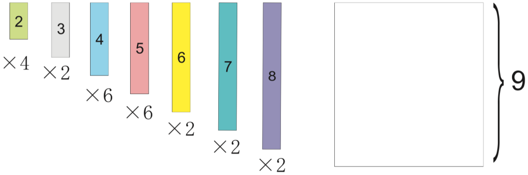
Cutting stock problem

Case study:

You are the person in charge of cutting in a large company producing rolls of paper. Your job is to skillfully cut the large rolls produced in a standard size into smaller rolls, with sizes demanded by the customers. It is not always possible to fully use every roll; sometimes, it is necessary to create leftovers, called trim loss. In this case, your aim is to use as few rolls as possible; in other words, to minimize the trim loss created. The width of the large rolls is 9 meters, and there are customers' orders for seven different sizes, as detailed in the table below. So, how should the large rolls be cut?

Length	Number of rolls
2 m	4
3 m	2
4 m	6
5 m	6

Cutting stock instance



Item lengths and roll size for an instance of the cutting stock problem.

Length	Number of rolls
2 m	4
3 m	2
4 m	6
5 m	6
6 m	2
7 m	2
8 m	2

Cutting stock problem

- there are orders for $i = 1, \ldots, m$ different widths
- quantity q_i ordered for width $0 \leq w_i \leq B$
- items to be cut from standard rolls with width B
- problem is to find a way to fulfill the orders while using the minimum number of rolls

Analysis

- Bin packing and cutting stock problems
  - may appear to be different
  - in fact it is the same problem
- Examples above refer to the same situation:
  - solution using a formulation for one of the problems is also a solution for the other case
  - deciding which to solve depends on the situation

The Bin Packing Problem

- NP-hard combinatorial optimization problem
- Notation:
  - n items
  - each with a given size s_i
  - identical bins with capacity B
- Aim: minimize total number of bins used

The Bin Packing Problem: straightforward formulation

- Assuming upper bound U of the number of bins is given
- Variables:
  - $X_{ij} = 1$ if item i is packed in bin j, 0 otherwise
  - $Y_j = 1$ if bin j is used, 0 otherwise

$$
\begin{aligned}
\text{minimize} \quad & \sum_{j=1}^{U} Y_j & \\
\text{subject to:} \quad & \sum_{j=1}^{U} X_{ij} = 1 & \text{for } i = 1, \cdots, n \\
& \sum_{i=1}^{n} s_i X_{ij} \le B Y_j & \text{for } j = 1, \cdots, U \\
& X_{ij} \le Y_j & \text{for } i = 1, \cdots, n; j = 1, \cdots, U \\
& X_{ij} \in \{0, 1\} & \text{for } i = 1, \cdots, n; j = 1, \cdots, U \\
& Y_j \in \{0, 1\} & \text{for } j = 1, \cdots, U
\end{aligned}
$$

Remarks

- Objective function: minimize number of bins used
- First constraints: force placement of each item in one bin
- Second constraints:
  - upper limit on the bins contents
  - items cannot be packed in a bin that is not in use
- The third constraints: enhanced formulation
  - if bin is not used $\rightarrow Y_j = 0$
  - then, items cannot be placed there $\rightarrow X_{ij} = 0$

AMPL model

```
param n;   # number of items
param U;   # maximum number of bins
param s {1..n};
param B;

var X {1..n, 1..U} binary;
var Y {1..U} binary;

minimize bins: sum {j in 1..U} Y[j];

subject to
Take {i in 1..n}: sum {j in 1..U} X[i,j] = 1;
Cap {j in 1..U}: sum {i in 1..n} s[i] * X[i,j] <= B * Y[j];
Activate {i in 1..n, j in 1..U}: X[i,j] <= Y[j];
```

Programming: generate example's data

```python
def BinPackingExample():
    B = 9
    w = [2,3,4,5,6,7,8]
    q = [4,2,6,6,2,2,2]
    s=[]
    for j in range(len(w)):
        for i in range(q[j]):
            s.append(w[j])
    return s,B
```

- data is prepared as for a cutting stock problem
  - width of rolls B, number of orders q and width orders w
- converted to the bin packing data
  - list s of sizes of items, bin size B

Programming: finding an upper bound of the number of bins

- Heuristics: procedures for obtaining a solution based on rules that do not guarantee reaching the optimum
- First-fit decreasing (FFD): well-known heuristics for bin packing
  - arrange items in non-increasing order of their size, then:
    - for each item try inserting it in the first open bin where it fits
    - if no such bin exists, then open a new bin and insert the item there

Programming: finding an upper bound of the number of bins

Calculate the upper limit U of the number of bins

```python
def FFD(s, B):
    remain = [B]
    sol = [[]]
    for item in sorted(s, reverse=True):
        for j,free in enumerate(remain):
            if free >= item:
                remain[j] -= item
                sol[j].append(item)
                break
        else:
            sol.append([item])
            remain.append(B-item)
    return sol
```

Programming: main

Load data into AMPL object

```
# read data, get ffd solution
s,B = BinPackingExample()
ffd = FFD(s,B)
n = len(s)
U = len(ffd)
print("FFD solution: {} bins, {}".format(U, ffd))

# read model, setup AMPL data
from amplpy import AMPL, Environment, DataFrame
ampl = AMPL()
ampl.setOption('solver', 'gurobi')
ampl.read("bpp.mod")
ampl.param['n'] = n
ampl.param['U'] = U
ampl.param['s'] = s
ampl.param['B'] = B
```

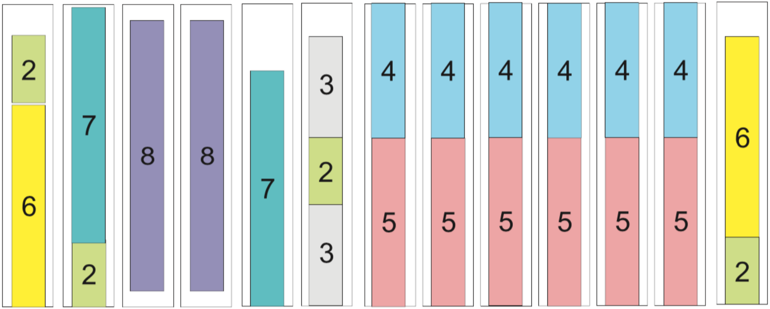
Programming: finding an upper bound of the number of bins

Solve model and report solution

```
# solve and report solution
ampl.solve()
bins = ampl.obj['bins']
print("Objective is:", bins.value())

X = ampl.var['X']
Y = ampl.var['Y']
for j in range(1,U+1):
    v = Y[j].value()
    if v > 0:
        print("bin {}".format(j), end="\t")
        for i in range(1,n+1):
            v = X[i,j].value()
            if v > 0:
                print(s[i-1], end=" ")
        print()
```

Solution obtained for the bin packing example.



Column generation method for the cutting stock problem

- Column generation method for the cutting stock problem: proposed by Gilmore and Gomory in 1961
- Preliminaries:
  - linear optimization problem: can be represented by means of a matrix
  - left-hand side of the constraints' coefficients
    - row of the matrix $\leftrightarrow$ constraint
    - column of the matrix $\leftrightarrow$ variable
- Hence:
  - constraints $\rightarrow$ also called rows
  - variables $\rightarrow$ also called columns

Column generation method for the cutting stock problem

- Column generation method:
  - only a (usually small) subset of the variables is used initially
  - method sequentially adds columns (i.e., variables)
    - dual variables $\rightarrow$ used for finding the approriate variable to add.

Column generation: previous example

- Many ways of cutting the base roll into requested widths
- Valid cutting pattern: set of widths not exceeding the roll's length ($B = 9$ meters)
  - First, generate simple patterns:
    - only of one ordered width
    - repeated as many times as it fits in roll length
    - order j of width $w_j \rightarrow$ can be cut B/w_j rounded down
- Pattern: represented as a vector/list with the number of times each width is cut
  - *e.g.*, width $w_1 = 2$ of order 1 was 2 meters
  - will be cut $\lfloor B/w_1 \rfloor = \lfloor 9/2 \rfloor = 4$ times in case of cutting only the width of order 1
  - cutting pattern: $(4, 0, 0, 0, 0, 0, 0)$
- Repeat for the other orders

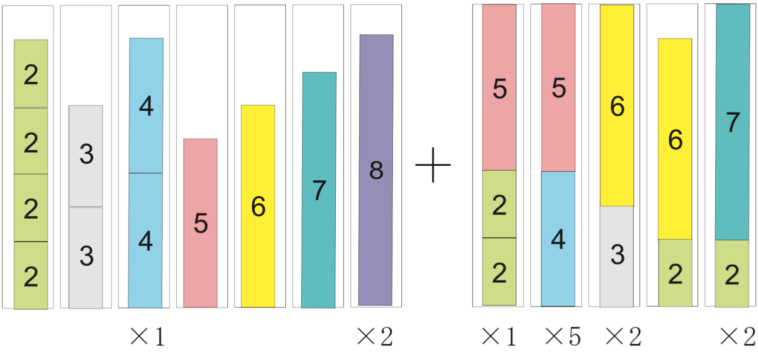
Programming

```
t = []
m = len(w)
for i in range(m):
    pat = [0]*m
    pat[i] = int(B/w[i])
    t.append(pat)
```

Initial set of cutting patterns:

```
[4,0,0,0,0,0,0]
[0,3,0,0,0,0,0]
[0,0,2,0,0,0,0]
[0,0,0,1,0,0,0]
[0,0,0,0,1,0,0]
[0,0,0,0,0,1,0]
[0,0,0,0,0,0,1]
```

Solution obtained for the cutting stock example



Modeling

- Variables:
  - x_i (integer) $\rightarrow$ number of times to use cutting pattern i
- Considering only the initial cutting patterns:
  - finding the minimum number of rolls to meet all the orders:

$$
\begin{array}{lccccccccl}
\text{minimize} & x_1 + & x_2 + & x_3 + & x_4 + & x_5 + & x_6 + & x_7 & & \\
 & 4x_1 & & & & & & & \geq 4 & \\
 & & 3x_2 & & & & & & \geq 2 & \\
 & & & 2x_3 & & & & & \geq 6 & \\
 & & & & x_4 & & & & \geq 6 & \\
 & & & & & x_5 & & & \geq 2 & \\
 & & & & & & x_6 & & \geq 2 & \\
 & & & & & & & x_7 & \geq 2 & \\
 & x_1, & x_2, & x_3, & x_4, & x_5, & x_6, & x_7 & \geq & 0, \text{integer}
\end{array}
$$

- Solving the linear relaxation:
    - optimum $\rightarrow 16\frac{2}{3}$
    - optimal solution $\rightarrow x = (1, 2/3, 3, 6, 2, 2, 2)$
    - for each constraint, dual variable: $\lambda = (1/4, 1/3, 1/2, 1, 1, 1, 1)$
- Dual variables: can be interpreted as the value of each order in terms of the base roll
    - *e.g.*, $\lambda_1 = 1/4 \rightarrow$ *"order 1 is worthy 1/4 of a roll"* (recall *duality* from previous class)
- Notice:
    - first cutting pattern $\rightarrow$ a lot of waste
    - to obtain a more efficient cutting strategy $\rightarrow$ base roll must be cut with better patterns
    - how to find a better pattern?

Improving cutting patterns

- Variables:
  - y_j (integer) $\rightarrow$ how many pieces of order j should be cut
  - aim: finding the cutting pattern with the largest value:

$$\text{minimize } \frac{1}{4}y_1 + \frac{1}{3}y_2 + \frac{1}{2}y_3 + y_4 + y_5 + y_6 + y_7$$
$$2y_1 + 3y_2 + 4y_3 + 5y_4 + 6y_5 + 7y_6 + 8y_7 \leq 9$$
$$y_1, y_2, y_3, y_4, y_5, y_6, y_7 \geq 0, \text{integer}$$

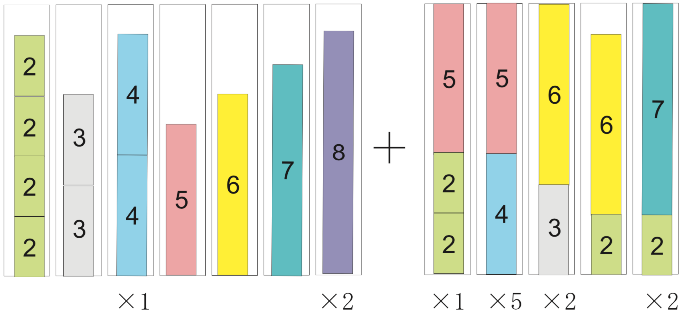
- Integer knapsack problem
  - knapsack variant where variables are non-negative integers
  - known to be NP-hard, but in practice can be solved easily
  - above instance: optimum is 1.5, solution $y = (2, 0, 0, 1, 0, 0, 0)$
    - $\rightarrow$ *a pattern with the value of 1.5 units of the base roll can be obtained by cutting a roll in two pieces of order 1 and one piece of order 4*
- Reduced cost of this new column is $1 - (2\lambda_1 + \lambda_4) = -0.5$
  - adding a column with this cutting pattern it is possible to obtain a benefit of 0.5 base rolls
    (recall *reduced costs* from previous class)

Updating the master problem

- Now, add new column and solve the linear relaxation problem again
  - variable $x_8 \rightarrow$ number of times to use the new cutting pattern

$$
\begin{array}{lcccccccccc}
\text{minimize} & x_1 & +x_2 & +x_3 & +x_4 & +x_5 & +x_6 & +x_7 & +x_8 & & \\
 & 4x_1 & & & & & & & +2x_8 & \geq & 4 \\
 & & 3x_2 & & & & & & & \geq & 2 \\
 & & & 2x_3 & & & & & & \geq & 6 \\
 & & & & x_4 & & & & +x_8 & \geq & 6 \\
 & & & & & x_5 & & & & \geq & 2 \\
 & & & & & & x_6 & & & \geq & 2 \\
 & & & & & & & x_7 & & \geq & 2 \\
 & x_1, & x_2, & x_3, & x_4, & x_5, & x_6, & x_7, & x_8 & \geq & 0
\end{array}
$$

- After adding five new patterns:
  - reduced cost of the new column found by solving the knapsack problem is not negative
  - column generation procedure stops
- We want an integer solution
  - add integrality constraints to the last linear problem
  - solving it: 13 rolls
- In general: no guarantee that all relevant patterns were added
  - solution may not be optimal for the original problem
  - in this particular example it is: minimum number of bins required is $\lceil \sum_{i=1}^{m} q_i w_1 / B \rceil = \lceil 12\frac{2}{9} \rceil = 13$



Modeling tip

Use the column generation method when the number of variables is extremely large.

- For many practical problems, a solution approach is to generate possible patterns and let an optimization model select the relevant ones
- The number of possible patterns may be enormous
  - enumerating all the possibilities impractical
  - solve an appropriate subproblem for generating relevant patterns
    - (knapsack problem, in the case of cutting stock)
- Complicated part: exchange of information between these two problems

Cutting stock problem: formulation

- Notation:
  - vector $(t_1^k, t_2^k, \ldots, t_m^k) \rightarrow k$ th cutting pattern of base roll width B into some of the m widths
    - $t_i^k \rightarrow$ number of times width of order i is cut out in the k\ -th cutting pattern.
    - to be feasible, pattern $(t_1^k, t_2^k, \ldots, t_m^k)$ must satisfy:

$$\sum_{k=1}^{m} t_i^k \leq B$$

  - $K \rightarrow$ current number of cutting patterns
- Cutting stock problem:
  - use currently available cutting patterns
  - cut a total number of ordered width j at least q_j times
  - aim: minimize total number of base rolls used
- Variables:
  - $x_k \rightarrow$ number of times pattern k is cut from the base roll

Cutting stock method: master problem

$$
\begin{aligned}
\text{minimize} \quad & \sum_{k=1}^{K} x_k & \\
\text{subject to:} \quad & \sum_{k=1}^{K} t_i^k x_k \geq q_i & \text{for } i = 1, \ldots, m \\
& x_k \geq 0, \text{integer} & \text{for } k = 1, \ldots, K
\end{aligned}
$$

Relax integrality constraints; this is called the master problem.

Cutting stock method: subproblem

- Notation:
  - $\lambda \to$ optimal dual variable vector of master problem
    - value assigned to each width i
- Aim: find a feasible pattern $(y_1, y_2, \ldots, y_m)$ maximizing the value of selected widths
  - $\to$ integer knapsack problem

$$
\begin{array}{lll}
\text{maximize} & \displaystyle\sum_{i=1}^{m} \lambda_i y_i & \\
\text{subject to:} & \displaystyle\sum_{i=1}^{m} w_i y_i \leq B & \\
 & y_i \geq 0, \text{integer} & \text{for } i = 1, \ldots, m
\end{array}
$$

$\to$ solution will be used as additional pattern in master problem

Column generation algorithm

- Start with simple patterns as initial columns
  - *e.g.*, patterns of $\lfloor B/w_i \rfloor$ rolls of width w_i
- Repeat:
  1. Solve the restricted master problem
     - let λ_i be the optimal dual variable
  2. Identify a new column by solving the knapsack subproblem
     - if optimal value is non negative, break
  3. Add the new column to master problem
- Add integrality constraints and resolve master problem

AMPL model: master problem

```
param K;
param m;
param q {1..m};
param t {1..K, 1..m};

var x {1..K} >= 0, integer;

minimize rolls: sum {k in 1..K} x[k];

subject to
Demand {i in 1..m}: sum {k in 1..K} t[k,i] * x[k] >= q[i];
```

AMPL model: subproblem

```
param m;
param B;
param w {1..m};
param lambda {1..m};

var y {1..m} >= 0, integer;

maximize z: sum {i in 1..m} lambda[i] * y[i];

subject to
Feasible: sum {i in 1..m} w[i] * y[i] <= B;
```

Programming: putting everything together

```
from amplpy import AMPL
B = 9  # roll width (bin size)
w = [2, 3, 4, 5, 6, 7, 8]  # width (size) of orders (items)
q = [4, 2, 6, 6, 2, 2, 2]  # quantitiy of orders

# generate initial patterns with one size for each item width
t = []  # patterns
m = len(w)
for (i, width) in enumerate(w):
    pat = [0] * m  # vector of number of orders to be packed into one roll (bin
    pat[i] = int(B / width)
    t.append(pat)
K = len(pat)
```

Programming: initialize ampl objects

```
# initialize master problem
master = AMPL()
master.option['solver'] = 'gurobi'
master.option['relax_integrality'] = 1
master.read("csp_master.mod")
master.param['K'] = K
master.param['m'] = m
master.param['q'] = q
t_ = master.param['t']
for k in range(1, K + 1):
    for i in range(1, m + 1):
        t_[k,i] = t[k-1][i-1]

# initialize subproblem
kp = AMPL()
kp.option['solver'] = 'gurobi'
kp.read("csp_knapsack.mod")
kp.param['m'] = m
kp.param['B'] = B
kp.param['w'] = w
```

```
while True:
    master.solve()
    rolls = master.obj['rolls']
    demand_ = master.con['Demand']
    lambda_ = {}
    for i in range(1, m + 1):
        lambda_[i] = demand_[i].dual()
    # setup knapsack subproblem
    kp.param['lambda'] = lambda_
    kp.solve()
    z = kp.obj['z']
    if z.value() <= 1:
        break
    # update new pattern
    pat = [0] * m  # vector of number of orders to be packed into one roll (bin
    y = kp.var['y']
    for i in range(1, m + 1):
        v = int(round(y[i].value()))
        print("y[{}] = {} : {}".format(i,v,y[i].value()))
        pat[i-1] = int(v)
    print("added pattern", pat)
    t.append(pat)
    K += 1
    master.param['K'] = K
    for i in range(1, m + 1):
        t_[K, i] = pat[i-1]
[end the cycle] ...
```

```
...
master.option['relax_integrality'] = 0
master.solve()
rolls = master.obj['rolls']
print("master objective:", rolls.value())

x = master.var['x']
rolls = []
for k in range(1,K+1):
    n = int(x[k].value() + .5)  # number of times pattern is used
    for j in range(n):
        rolls.append(sorted([w[i] for i in range(m) if t[k-1][i] > 0 for j in r
rolls.sort()
print(rolls)
```

After finishing the column generation cycle, we solve the (integer) model with all patterns added.

Today's class

- Two views of the same problem:
  - bin packing
  - cutting stock
- Two formulations:
  - straightforward formulation for bin packing
  - column generation for cutting stock