

Resumo PAA II

Sophia Carrazza

- P1
1. Visão geral do semestre e alguns conceitos
 2. Revisão de grafos
 3. Custo computacional e ordens de complexidade
 4. Tratabilidade → Problema tratável ou não
 5. Algoritmos Gulosos
 6. Divisão e conquista
 7. Programação dinâmica
 8. Outras estratégias de projeto de algoritmos
- P2

> intervalos diferentes

<https://youtu.be/-RNISTASGKk?si=kc7ap4sqY2Ax5MP>

→ intervalos diferentes e coloração c/ gulosos

<https://youtube.com/playlist?list=PL5TPkym335qxReLGUyn7kJwRRYPogEFVi&si=T6gomJUuVdWS1qCK>

→ playlist de programação dinâmica

https://youtube.com/playlist?list=PL6mfjjCaO1WoxRXI_yyAG_7S8gz9dxqhV&si=wvFXeHhHB8MijXnq

→ playlist de PAA da Rudini

<https://youtu.be/m1qkWdQeiv8?si=o3ietx5TQPFUx7x7>

→ intervalos diferentes c/ guloso

https://www.ime.usp.br/~pf/analise_de_algoritmos/aulas/intervalos.html

→ intervalos diferentes da USP (O + IMPORTANTE DE LTR !!!)

PAA

↳ Divisão e Conquista: divisão do problema original em subproblemas menores do mesmo tipo, resolvidos recursivamente.

Etapas do projeto de algoritmos por divisão e conquista

- Dividir: o problema original é dividido em subproblemas menores do mesmo tipo.
- Conquistar: os subproblemas são resolvidos recursivamente, sendo que os subproblemas pequenos são caso base.
- Combinar: as soluções dos subproblemas são combinadas numa solução do problema original.

Vejas alguns algoritmos que usam o método da divisão e conquista:

- algoritmo ALTURA-DC para o problema do segmento de soma máxima,
- Busca binária: divide a instância em duas menores e resolve uma delas; a fase de combinação é vazia.
- Mergesort: divide a instância em duas menores (essa fase é muito rápida) e resolve as duas instâncias menores; a fase de combinação é a que consome mais tempo.
- Quicksort: a fase da divisão é lenta e produz duas instâncias menores; a fase de combinação é muito rápida.
- Algoritmo de mediana: a fase da divisão, que produz duas instâncias menores, é lenta; a fase da conquista resolve uma dessas instâncias; a fase de combinação é muito rápida.
- Algoritmo de Karatsuba: a fase da divisão é muito rápida e produz três instâncias menores; a fase de combinação consiste em algumas operações aritméticas.

↳ Algoritmos Gulosos: método que tenta resolver o problema fazendo a escolha localmente ótima p/ encontrar uma correspondência global ótima.

- não se arrepende de uma decisão. Eles são definitivos.
- podem fazer cálculos repetitivos
- nem sempre produz a melhor solução

Gulosos ✕ Programações Dinâmica

- mapeia
- local → global
- solução de cima p/ baixo
(dos maiores subproblemas
p/ os menores).

- há superposição de problemas
- problemas são dependentes
- solução de baixo p/ cima (dos menores subproblemas p/ os maiores)

Programação Dinâmica :

resolvemos uma entrada dividindo-a em entradas menores do mesmo tipo e armazenando a solução destas para que recompute-las.

"Dynamic programming is a fancy name for [recursion] with a table. Instead of solving subproblems recursively, solve them sequentially and store their solutions in a table. The trick is to solve them in the right order so that whenever the solution to a subproblem is needed, it is already available in the table."

— Ian Parberry, *Problems on Algorithms*

Dynamic programming é uma espécie de tradução iterativa inteligente da recursão e pode ser definido como "recursão com apoio de uma tabela".

Aísim, cada instância do problema original é resolvida a partir de subinstâncias de original, com auxílio de uma tabela que armazena suas soluções.

* o consumo de tempo do algoritmo é, em geral, proporcional ao tamanho da tabela.

Na execução de um fibonacci, por exemplo:

sem P.D.:

```
python
def fib_recursoivo(n):
    if n <= 1:
        return n
    return fib_recursoivo(n-1) + fib_recursoivo(n-2)

Execução para n = 5:
text
fib(5)
└─ fib(4)
   |  └─ fib(3)
   |  |  └─ fib(2)
   |  |  |  └─ fib(1) = 1
   |  |  |  └─ fib(0) = 0
   |  |  fib(1) = 1
   |  |  |  └─ fib(0) = 0
   |  fib(2)
   |  |  └─ fib(1) = 1
   |  |  └─ fib(0) = 0
   fib(3)
   |  └─ fib(2)
   |  |  └─ fib(1) = 1
   |  |  └─ fib(0) = 0
   fib(1) = 1
```

com P.D.:

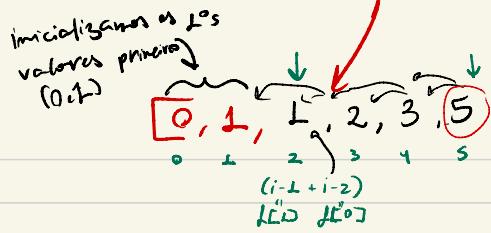
```
python
def fib_pd(n):
    if n <= 1:
        return n
    f = [0] * (n+1)
    f[0], f[1] = 0, 1
    for i in range(2, n+1):
        f[i] = f[i-1] + f[i-2]
    return f[n]

Execução para n = 5:
text
Passo 1: f[0] = 0, f[1] = 1
Passo 2: f[2] = f[1] + f[0] = 1 + 0 = 1
Passo 3: f[3] = f[2] + f[1] = 1 + 1 = 2
Passo 4: f[4] = f[3] + f[2] = 2 + 1 = 3
Passo 5: f[5] = f[4] + f[3] = 3 + 2 = 5
```

↳ redundância alta ($fib(2)$ é calculado 3x
+ $fib(3)$ 2x)

↳ $O(2^n)$ → chamadas recursivas

↳ cada $f[i]$ é calculado só 1 vez
↳ $O(n)$ → tamanho da tabela



Problema dos Intervalos Disjuntos:

(Problema da coleção disjunta máxima de intervalos). $[a[i], b[i]]$ \downarrow $\begin{matrix} \text{início} \\ \text{termino} \end{matrix}$

- evento → acontece por um intervalo de tempo ($[a[i], b[i]]$)
- 2 eventos são compatíveis se seus intervalos forem disjuntos.
- Devemos atender o maior número possível de eventos compatíveis

↳ "Encontrar uma subcoleção disjunta máxima de uma coleção".

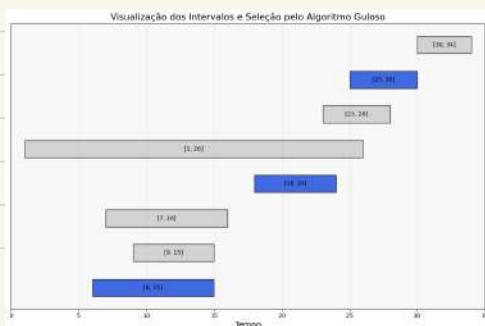
que é $O(n \log(n))$



↳ solução: Algoritmo Gúlico.

- primeiro, começamos com a ordenação dos intervalos em ordem crescente de término. Ou seja $b[p] \leq b[p+1] \leq \dots \leq b[r]$
- a intervalo que termina + cedo possível
- cada iteração do algoritmo escolhe um intervalo de tempo mínimo e remove da coleção todos os intervalos que não são posteriores ao intervalo escolhido (ou seja, que se sobreponham a ele). $a[j] \leq b[i]$
- depois, procuramos o próximo intervalo que comece depois do término do intervalo escolhido e repetimos todo o processo até não haver + intervalos não analisados.

↳ porque isso funciona?: Pois ao escolher o intervalo que termina mais cedo, você deixa mais espaço para caber mais intervalos.



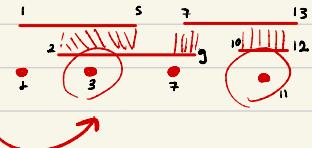
Intervalos ordenados pelo término:
Intervalo 1: [6, 15]
Intervalo 2: [9, 15]
Intervalo 3: [7, 16]
Intervalo 4: [18, 24]
Intervalo 5: [1, 26]
Intervalo 6: [23, 28]
Intervalo 7: [25, 30]
Intervalo 8: [30, 34]

Processo de seleção dos intervalos:
Escolhido intervalo 1: [6, 15]
Ignorado intervalo 2: [9, 15] (sobreposição)
Ignorado intervalo 3: [7, 16] (sobreposição)
Escolhido intervalo 4: [18, 24]
Ignorado intervalo 5: [1, 26] (sobreposição)
Ignorado intervalo 6: [23, 28] (sobreposição)
Escolhido intervalo 7: [25, 30]
Ignorado intervalo 8: [30, 34] (sobreposição)

Conjunto máximo de intervalos disjuntos escolhidos:
[6, 15]
[18, 24]
[25, 30]

Problema de Cobertura de Intervalos (Interval Stabbing Problem)

"Um intervalo I é estável em relação a um ponto P se P está no intervalo I. Projete uma solução para encontrar a menor quantidade de pontos que deixe todos os intervalos estáveis."



Solução: (pensei c/ o gênero dos intervalos disjuntos)

- ▷ ordenarmos os intervalos em ordem crescente de término
- ▷ escolhermos os intervalos de término + cedo e adicionares o ponto mais à direita desse intervalo à solução.
- ▷ excluir todos os intervalos que começam antes ou igual ao ponto escolhido.

Problema de Cobertura Mínima de Intervalos

?

Scheduling Problems:

Given a set of n jobs each job i has:

- a start time s_i
- an end time f_i
- a weight (or value) w_i

"Seja I um conjunto de intervalos e $W : I \rightarrow \mathbb{Z}^+$ uma função que pondera cada intervalo de I . Como projetar uma solução pra maximizar a soma de pesos de um subconjunto de intervalos mutuamente disjuntos em $O(n^2)$?"

Solução: Programação Dinâmica (Weighted Interval Scheduling)

- ordenamos os intervalos pelo tempo de término
- consideramos o último intervalo (o que termina por último)
- p/ cada intervalo, temos duas opções:
 - 1º - escolher esse intervalo, adicionando o peso dele ao total e removendo todos os intervalos que se sobreponem a ele
 - 2º - não escolhi-lo \Rightarrow resolvemos o problema p/ os $n-1$ tempos restantes, ignorando o atual.
- a solução p/ os primeiros i intervalos é o máximo entre: \rightarrow o peso do intervalo i + a solução ótima p/ os intervalos que não conflitam com ele
 \rightarrow a solução ótima p/ os primeiros $i-1$ intervalos.

► Ima chega na relação de recorrência:

$$OPT(i) = \max(W_i + OPT(p(i)), OPT(i-1))$$

maior índice

peso do intervalo i .
 $j < i$ tal que o intervalo j não sobreponha ao intervalo i .

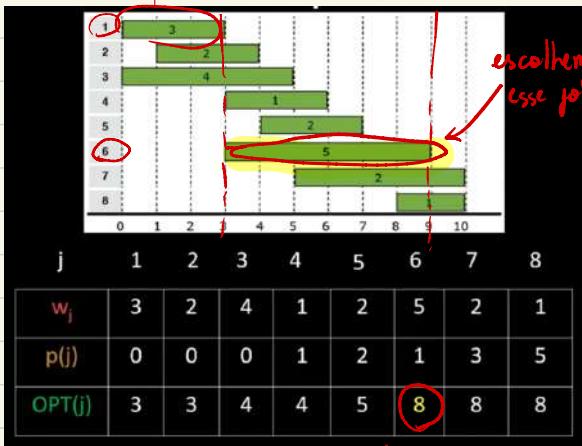
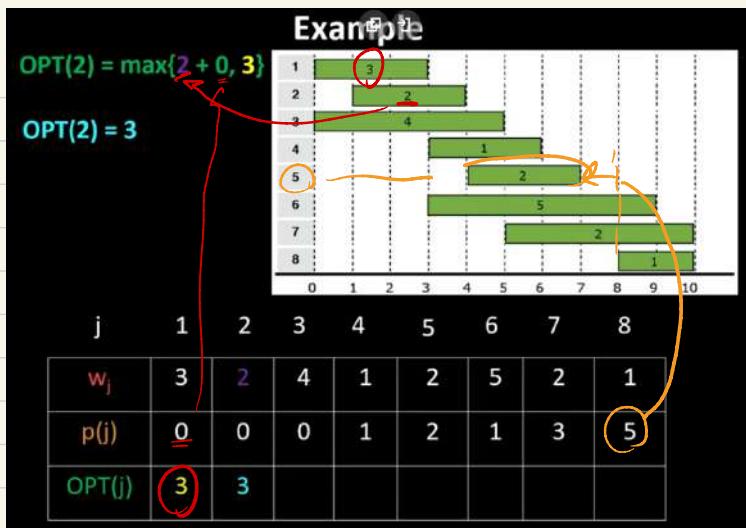
Idea: If we assume we only look at the last job when having sorted the jobs by finish time, then either we pick it or we don't.

Case 1 (We pick it): Then we need to remove all overlapping jobs and compute the solution for the subproblem of the jobs left.

Case 2 (We don't pick it): Then we simply compute optimal solution for the subproblem with $n-1$ items left.

We can do this recursively for $j = 1, 2, \dots, n$ and we define $OPT(j) =$ "Maximum weight that can be achieved by selecting the jobs from first j intervals"

em outras palavras, $p(i)$ é o índice do intervalo mais recente que termina antes de inicio do intervalo i . !!



Finding optimal items

What we then want to return is $\text{OPT}(n)$ which is the maximum weight for all items.

The solution for finding which items we picked is by backtracking, simply if $w_j + \text{OPT}(p(j)) \geq \text{OPT}(j-1)$ then we add item j to optimal solution and look at $\text{OPT}(p(j))$ else we ignore this item and look at subproblem $\text{OPT}(j-1)$.

Time complexity:

- Sorting is $O(n\log n)$
- Finding $p(j)$, $j = 1, 2, \dots, n$ has time complexity $O(n\log n)$ (clever use of binary search)
- To find maximal weight (when p found): $O(n)$
- Backtrack to find optimal items: $O(n)$

Total: $O(n\log n)$

no final de solução temos o intervalo 6 + o 8 (que é o que não sobrepuja o intervalo 6).

Todos os problemas de Intervalos Disjuntos:

- 1 - Interval Scheduling
- 2 - Particionamento
- 3 - Estabilidade
- 4 - Cobertura Total de um Intervalo

1 - Interval Scheduling → Agendamento de Intervalos

- ↳ objetivo = selecionar o maior n. de intervalos não sobrepostos
- ↳ como pode ser cobrado = escalonamento de tarefas p/ evitar overlaps, alocação de portas de aeroporto p/ decolagens e aterrissagens.

↳ algoritmo: Tempo de Fim

$A = \{I^i\}_{i=1}^n$ (intervalo)

ultimo termino = $f_{[0]}$

- ordenamos os intervalos pelo tempo de término em ordem crescente
- selecionaremos iterativamente o intervalo com menor tempo de término que não sobreponha ao último selecionado, ou seja:
p/ cada j :
 $\forall i \rightarrow [f_j] \geq \text{ultimo término} : A \cdot \text{activations}(j) \in \text{ultimo tempo}$
 $= f_{[j]}$

↳ complexidade = $O(n \log n)$ → preço da ordenação (se for heaps ou $O(n^2)$ → se for bubblesort ou quicksort)

Outros soluções p/ o Interval Scheduling: (que não são ótimas)

- ↳ tempo de início → selecionar os intervalos em ordem crescente de tempo de início
 - * um intervalo que começo cedo pode se estender por muito tempo, bloqueando intervalos novos.
- ↳ por duração (tempo) → selecionar os intervalos em ordem crescente de durações
 - * um intervalo curto pode bloquear dois intervalos não conflitantes entre si.
- ↳ por qntd de conflitos → p/ cada intervalo, contar a qntd de outros intervalos q/ os quais ele conflita e selecionar esses intervalos em ordem crescente por esse número.
 - * isso ignora a ordem temporal dos intervalos, podendo selecionar os de menor conflito mas q/ não são os melhores.

2 - Particionamento

- ↳ objetivo = alocar todos os pacientes em um número mínimo de salas, sem sobreposição (do tempo de duração deles pacientes)
- ↳ como pede no enunciado = agendamentos de voo em portões de embarque, aulas em salas e consultas médicas em salas de hospital.

↳ algoritmo:

- ordenamos os intervalos por tempo de inicio
 - alocamos as salas usando uma fila de prioridade baseada no término de cada paciente.
- * logo, a passo a passo + detalhado é:
- 1- ordenamos os intervalos por tempo de inicio
 - 2- inicializamos a fila de prioridade p/ as salas vazia.
 - 3- processamos cada intervalo na ordem de inicio (p/ cada paciente j):
 - 4- verificamos se há alguma sala que já esteja disponível (i.e o término da última paciente dela é menor que o inicio de j).
 - 5- se encontrarmos uma sala compatível, removemos ela da fila de prioridade, descoamo j a essa sala e atualizamos o horário de término dela p/ $f[j]$. e reinserimos ela na fila de prioridade.
 - 6- se não encontrarmos uma sala, criamos uma nova (incrementando n), descoamo a paciente j nessa sala e inserimos ela na fila de prioridade.

* porque o algoritmo é ótimo? → o algoritmo só abre uma nova sala quando totalmente necessário (quando já há outra paciente acontecendo simultaneamente). Logo, isso garante o uso mínimo de salas p/ todos os pacientes (considerando que todos devem ocorrer).



3- Estabilidade

↳ objetivo = atribuir recursos (pontos, cores, etc) a intervalos de forma que os intervalos tenham cores distintas, minimizando o n° de cores.

poderia ser resolvido c/ ordenação por tempo de início e pegar o menor ponto p/ cada intervalo, mas há contra exemplo.

4- Cobertura Total de um Intervalo

→ objetivo = selecionar o menor número de intervalos p/ cobrir completamente um intervalo alvo.

→ passo a passo: dada um intervalo alvo $[s, f]$:

1- ordenarmos os intervalos por tempo de início

2- cobertura = lista vazia e posição atual = s

3- enquanto a posição atual é menor que f:

→ selecionamos o intervalo j com $s[j] \leq$ posição atual e o maior $f[j]$ e adicionamos j à cobertura.

→ se esse j não existe, a solução é impossível

4- posição atual = $f[j]$

Problema dos moedas:

"Seja $M = \{1, 2, 5\}$ um conjunto de moedas. Seja X um valor que deve ser atingido. Projete uma solução ótima para atingir X com essas moedas (com a menor qntd de moedas possível)"

poderíamos resolver pelo método guloso, selecionando sempre as moedas de maior valor, mas isso só funcionaria p/ certos valores, como os neededs das EVA (25, 10, 5 e 1 centavos).

Logo, a solução correta deve problema é usando programação dinâmica, já que a recursão é muito inefficiente.

* caso base \rightarrow se X é o valor de uma das moedas, a resposta é 1 única moeda

→ Reduzimos o trabalho feito ao lembrar de alguns resultados anteriores p/ evitar recalcular o que já fizemos

↳ solução passo a passo:

1º - criarmos um array "dp" de tamanho $X+1$, onde $dp[i]$ armazena o n° mínimo de moedas p/ formar o valor i .

↳ $dp[0] = 0$ (sem moedas p/ formar $X=0$)

↳ $dp[i] = \infty$ p/ $i \geq 1$ (assumimos que não imperiais)

2º - p/ cada valor de i , de 1 a X , atualizamos

$dp[i]$ verificando todos as moedas $c \in M$:

$$dp[i] = \min_{\substack{c \in M \\ c \leq i}} (dp[i-c] + 1)$$

3º - se $dp[X]$ permanece ∞ , o valor X não pode ser formado. Caso contrário, $dp[X]$ é o n° mínimo de moedas.

Exemplo para $X = 8$:

i	Moedas Possíveis ($c \leq i$)	Cálculo de $dp[i]$	$dp[i]$
0	-	$dp = 0$	0
1	1	$dp + 1 = 1$	1
2	1	$dp[1] + 1 = 2$	2
3	1	$dp[2] + 1 = 3$	3
4	1, 4	$\min(dp[3] + 1, dp[1] + 1) = 1$	1
5	1, 4	$\min(dp[4] + 1, dp[1] + 1) = 2$	2
6	1, 4, 6	$\min(dp[5] + 1, dp[2] + 1, dp[1] + 1) = 1$	1
7	1, 4, 6	$\min(dp[6] + 1, dp[3] + 1, dp[1] + 1) = 2$	2
8	1, 4, 6	$\min(dp[7] + 1, dp[4] + 1, dp[2] + 1) = 2$	2

Solução: 2 moedas (4 + 4).

Change to Make											
1	2	3	4	5	6	6	8	9	10	11	
1											
1	2										
1	2	3									
1	2	3	4								
1	2	3	4	1							
...											
1	2	3	4	1	2	3	4	5	1		
1	2	3	4	1	2	3	4	5	1	2	

Figura 4: Mínimo Número de Moedas Necessárias para Fazer o Troco

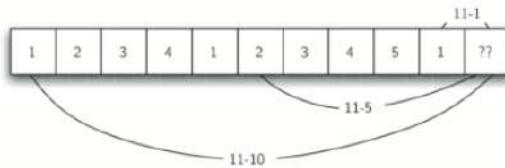


Figura 5: Três Opções a Considerar para o Número Mínimo de Moedas para 11 Centavos

aqui, assumimos que temos um estoque infinito de moedas:

moedas \ X	0	1	2	3	4	5	6	7	8
L	0	1	2	3	4	5	6	7	8
{L, 4}	0	1	2	3	①	2	3	4	②
{L, 4, 6}	0	1	2	3	4	2	1	2	③ 2 (retornarão o de cima)

$$y-y=0 \rightarrow \text{olha na coluna } 0, \text{ linha anterior} \rightarrow X+L$$

$$0+L=L$$

$$7-4=3+1 \rightarrow \min(\text{veta}, \text{veta acima})$$

$$5-y=4+1$$

$$7-4=7+1$$

$$8-y=8+1$$

$$8-y=7+1$$

↳ passo a passo de preenchimento da tabela:

- a coluna 0 é inicializada c/ 0 (não precisamos de nenhuma moeda p/ formar o valor 0).
- p/ cada valor da tabela, escolhemos o valor mínimo entre:
 - 1- manter o valor da linha anterior na mesma coluna (não usar a nova moeda)
 - 2- L + (valor da linha anterior que está na coluna da moeda: valor desejado (de coluna) - valor da nova moeda adicionada)
- o resultado será o valor da coluna do X desejado na última linha.

↳ solução c/ vetor único:

```
// Inicializar vetor
dp[0] = 0
para i de 1 até valor_máximo:
    dp[i] = infinito

// Para cada moeda disponível
para cada moeda m em {1,4,6}:
    // Atualizar todos os valores possíveis
    para i de m até valor_máximo:
        dp[i] = min(dp[i], dp[i-m] + 1)
```

Problema de Seqüência Consecutiva de maior soma:

↳ é o "problema do segmento de soma máxima" \rightarrow dado um vetor de $A[1 \dots n]$ de números inteiros, encontre um segmento do vetor que tenha soma máxima.

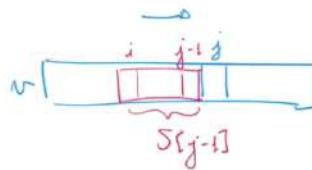
↳ Solução: podemos resolver esse problema de 3 formas:

1º- analisar, matematicamente todos os segmentos de $A[1 \dots n]$ e escolher o de maior soma \rightarrow (força bruta)
o consumo total desse algoritmo é cubico $O(n^3)$

2º- divisão e conquista \rightarrow dividimos a instância dada ao meio, resolvemos cada uma das metades e combinamos as duas soluções.

3º-

- Isto é, vamos pensar em como resolver nosso problema,
 - usando soluções para instâncias menores do mesmo problema.
- De fato, vamos pensar em instâncias de um problema relacionado,
 - em que o limite direito do intervalo é fixo.
- Vamos calcular $S[j]$, i.e., o segmento de soma máxima
 - que termina e contém j
- Seja $S[i:j] = v[i:j]$ um segmento de soma máxima
 - que termina e contém j



se $S[j-1] < 0$ então

$$S[j] = v[j]$$

$$i=j$$

se $S[j-1] \geq 0$ então

$$S[j] = v[j] + S[j-1]$$

Problema da Mochila

→ tenas uma capacidade da mochila e um conjunto de itens c/ peso e lucro (valor).

*****A tabela será construída com:

→ capacidade

	0	1	2	3	4	5	6	7	8	9	10	11
0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	1	1	1	1	1	1	1	1	1	1	1
2	1	2	3	5	5+1	6	6	6	6	6	6	6
3	0	1	1	5	5+1	6	6	6	6	6	6	6
4	0	1	2	5	6	8	3	0	13	14	14	14
5	0	1	4	5	6	8	9	9	13	14	14	14

48-5-3

cabe + o de peso
(5+8)

aqui já podemos
colocar o item de
piso 3 de acordo

a capacidade (luzo S)

agrinaria o L_1 ,
más cerca L_0 e
poco que a finales
anterior (L_3), matem

Tens		Centas & Decatas		Unidades	
	Item	Peso	Luzas		
1		1	1		
3		3	5		
5		5	8		
8		8	10		

→ o que preencheres
será o lince
(o melhor resultado
possível de lince)

*agora p/ verificar quais items foram inseridos:

	0	1	2	3	4	5	6	7	8	9	10	11
0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	L	T	L	T	L	L	L	L	L	L	L
2	0	L	T	L	T	L	L	L	L	L	L	L
3	0	L	T	L	5	5	6	6	6	6	6	6
4	0	L	T	L	5	6	8	9	0	13	4	14
5	0	L	T	L	5	6	8	9	0	13	4	14
6	0	L	L	S	6	8	9	9	L	14	14	14

3) noure mudanca de valor
de l'obra debaxo p/ de
cima?

Logo, a expressão de inserção na tabela é:

$$Z(i, w) = \max \{ Z(i-1, w), U_i + Z(i-1, w - p_i) \}$$

item ↴

↓ capacidade

lucro ↴

lucro

↓ não lucro o item

↓ lucro + item

↓ se não: o item deve para não foi inserido

↓ se sim: one item

↓ se não: levando o item

↓ se sim: que não pode ser levado o item

↓ se não: o item deve para não foi inserido

$$Z(i, \omega) = \begin{cases} 0, & \omega_i = 0 \\ 0, & \omega_i = 0 \\ \max_{\substack{x_i \in \{0,1\} \\ p_i x_i \leq \omega}} \{ u_i x_i + Z(i-1, \omega - p_i x_i) \} \end{cases}$$

variável que indica se pegamos ou não o item $(0, 1)$