Programação Dinâmica (parte 1)

SCC218 - Alg. Avançados e Aplicações

João Batista

Programação Dinâmica

- Problemas desafiadores em programação são muitas vezes aqueles que envolvem otimização.
  - encontrar uma solução que minimize ou maximize alguma função

- Vimos aqui alguns exemplos
  - Qual o nro mínimo de moedas ... lembra-se ?
  - Qual o nro máximo de tarefas compatíveis ... lembra-se ?

- Este tipo de algoritmo requer que provemos que sempre **melhor solução possível** é retornada.

Programação Dinâmica

- Algoritmos Gulosos tomam a melhor decisão local, na esperança de gerar o melhor globalmente. São eficientes, mas **não garantem otimalidade**.

- Busca exaustiva (força bruta) tenta todas as possibilidades e seleciona aquela que produz a melhor resposta. O problema, muitas vezes é o **custo proibitivo**.

Programação Dinâmica

- Programação dinâmica, combina o melhor das duas estratégias.
  - Nos dá uma forma sistemática de buscar todas as possibilidades (garante otimalidade), ao armazenar resultados parciais de forma a evitar recálculo (o que dá eficiência).

- Ao armazenar o resultado de todas as possíveis decisões, e usar tal informação de forma sistemática, o esforço final é minimizado tremendamente.

Programação Dinâmica

- É uma técnica eficiente de implementar algoritmos recursivos armazenando resultados parciais.
- O truque é perceber quando o algoritmo recursivo original calcula a mesma coisa várias vezes.
- Quando este é o caso, utilize algum mecanismo para armazenar a resposta de cada sub-problema em uma tabela. Esta tabela deve ser consultada sempre, de forma a evitar calcular novamente aquilo que já foi calculado.

Programação Dinâmica

- Opa... falamos em subproblemas.... Então isso também tem algo a ver com dividir e conquistar?
- Ambos paradigmas separam o input em partes, encontra subsoluções para elas e gera soluções mais gerais a partir das menores.
- Dividir e conquistar divide em pontos pré-determinados (sempre ao meio)
- PD divide em todos os possíveis pontos ao invés de um ponto pré-especificado. Após tentar todos, determina aquele que é ótimo.

Programação Dinâmica

- Programação Dinâmica (PD) é uma poderosa técnica de programação matemática.

  - PD somente é aplicável a um conjunto de problemas que obedece o princípio de otimalidade de Bellman (subestrutura ótima)

  - Um problema satisfaz o princípio da otimalidade se as sub soluções de uma solução ótima do problema são em si soluções ótimas para seus subproblemas

- "*Principle of Optimality: An optimal policy has the property that whatever the initial state and initial decision are, the remaining decisions must constitute an optimal policy with regard to the state resulting from the first decision* (Bellman, 1957)."

Princípio da Otimalidade

- Considere um processo de decisão multi-estágio
  - Problema com N variáveis representado por N problemas de apenas uma variável em série de forma que a saída de um é a entrada de outro.

$$\begin{aligned} \min_{x_k, u_k} \quad J &= \sum_{k=0}^{N-1} g_k(x_k, u_k) + g_N(x_N) \\ \text{s. to} \quad x_{k+1} &= f(x_k, u_k), \qquad k = 0, 1, \cdots, N-1 \\ x_0 &= x_{init} \end{aligned}$$

- *k é um índice no tempo (discreto)*
- *x_k é o estado no tempo k*
- *u_k é uma decisão de controle qualquer*
- *$g_N(.)$* é o custo do estado terminal.
- $g_k(.)$ é o custo 'instantâneo'

Princípio da Otimalidade

- Assuma que em um passo k, são conhecidas todas as decisões futuras ótimas [u(k+1), u(k+2), ... , u(N-1)]   k = 0..N-1
- Compute a melhor solução para o passo atual e pareando com as futuras decisões, escolha o par que retorna a solução ótima.
- Isso é feito recursivamente, começando de trás para frente...

Princípio da Otimalidade

- Seja $V_k(x_k)$ o custo ótimo até o final, ou seja, do passo k até N, dado o estado corrente x_k.
- O princípio da otimalidade pode ser escrito recursivamente como:

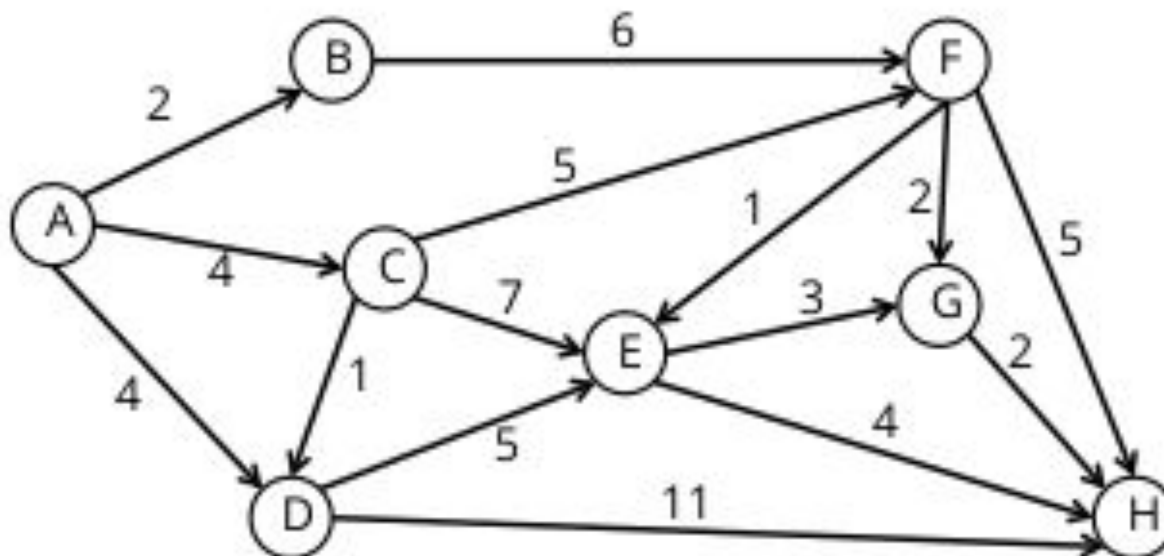
$$V_k(x_k) = \min_{u_k} \{g(x_k, u_k) + V_{k+1}(x_{k+1})\}, \qquad k = 0, 1, \cdots, N-1$$

Sendo a condição de fronteira

$$V_N(x_N) = g_N(x_N)$$

O problema é resolvido de trás para frente e recursivamente !

Princípio da Otimalidade

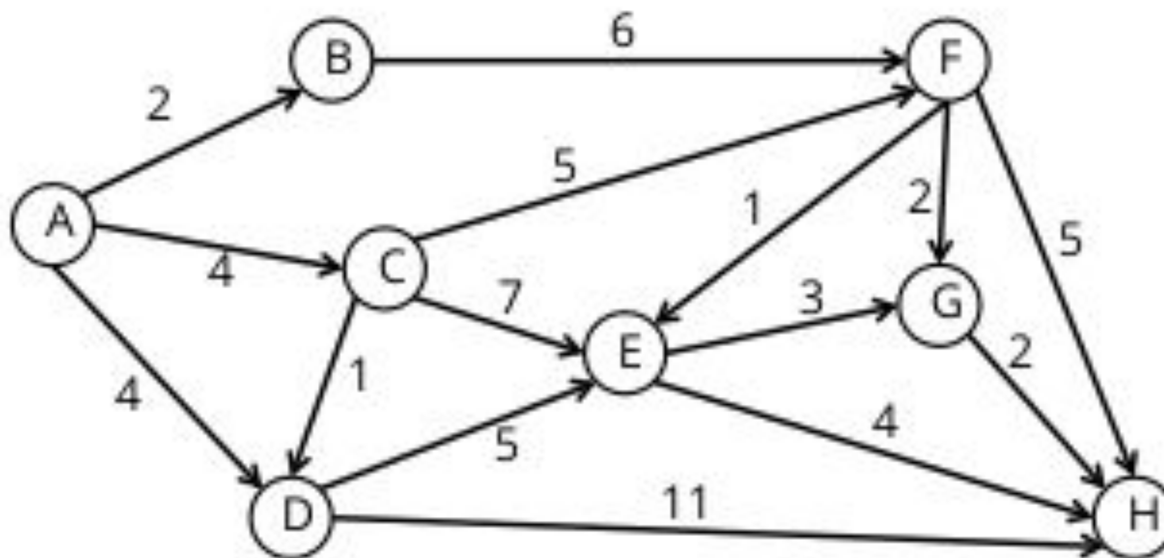


Encontre o Caminho mínimo de A a H

$$V(i) = \min_{j \in N_i^d} \{c(i,j) + V(j)\}$$

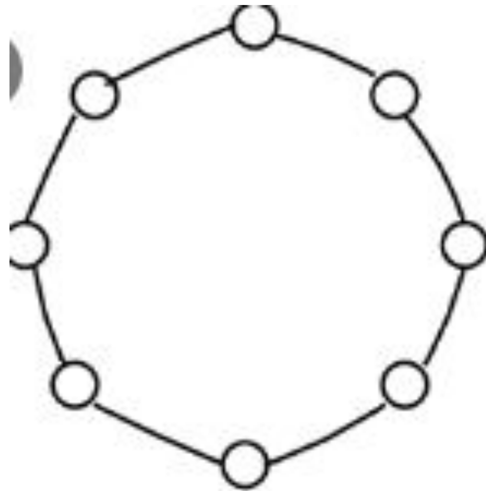
$$V(H) = 0 \qquad N_C^d = \{D, E, F\}.$$

Princípio da Otimalidade



- Comece em V(G) = min {V(G,H) + V(H)} = 2 + 0 = 2
- Recursivamente, aplique a equação de otimalidade, até chegar ao vértice A.

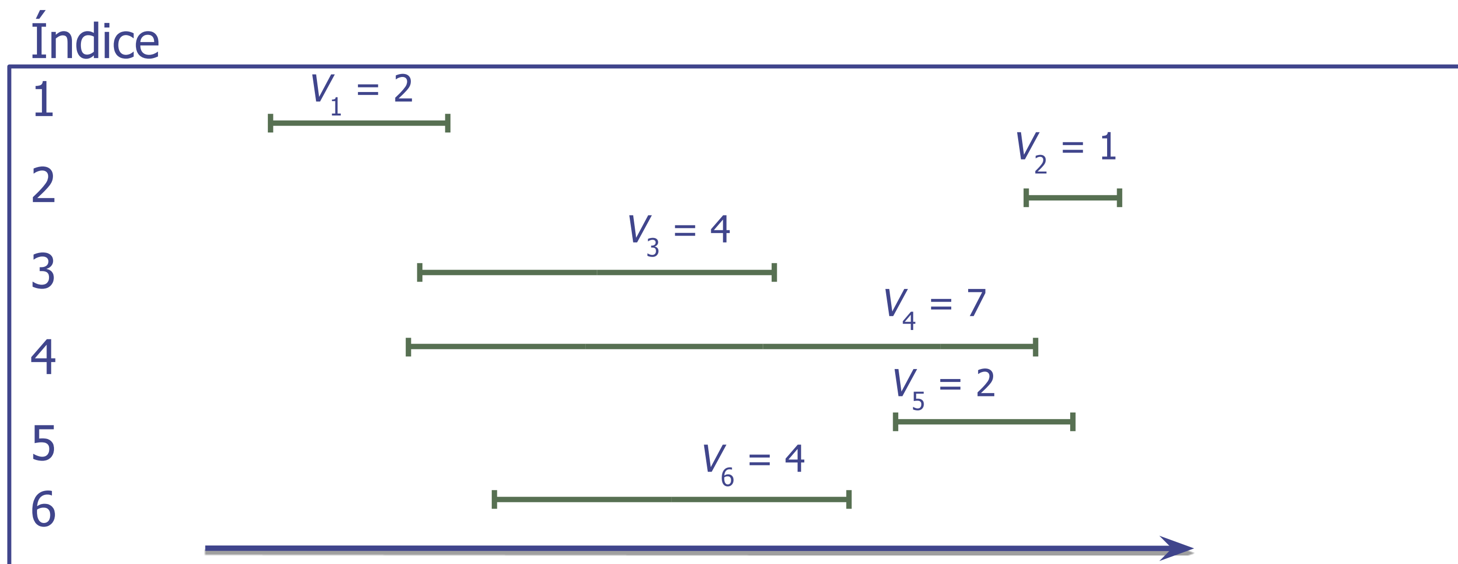
Princípio da Otimalidade



- Seja o grafo acima, em forma de anel, com vértices A,..., H
- Procure o caminho mais longo.:
  - Podemos aplicar o princípio da otimalidade aqui????

Exemplo 1 – WISP

- Problema de escalonamento de intervalos com pesos (*Weighted Interval Scheduling Problem* – WISP):
  - Selecionar um subconjunto de intervalos com a maior soma de pesos possível sem que dois intervalos estejam sobrepostos.

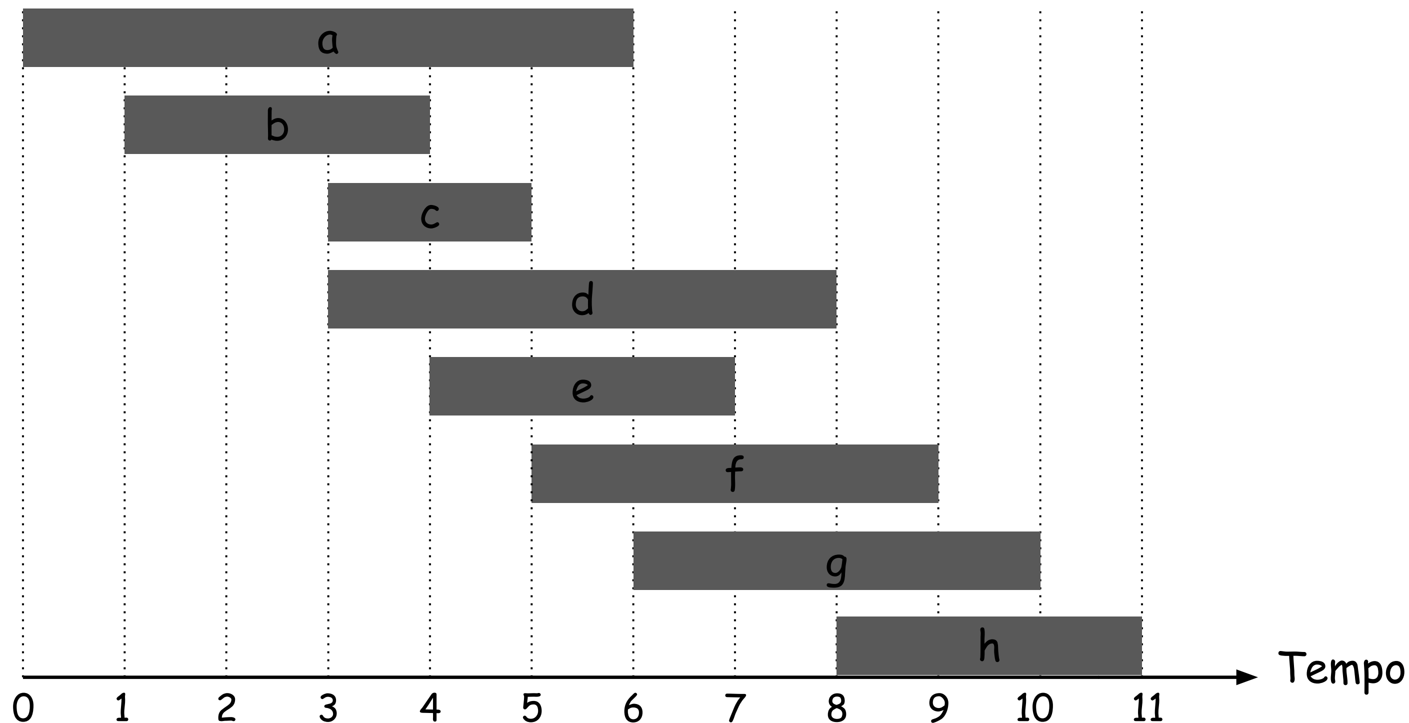


Vamos relembrar um caso que vimos na aula de Algoritmos GULOSOS

- Era o mesmo problema de intervalos, apenas mais específico em que os pesos não existiam
- Mas podemos assumir que os pesos eram todos iguais e de valor = 1
- …….

Agendamento de Intervalos.

- Tarefa j começa em s_j e termina em f_j.
- Duas tarefas são compatíveis se não há sobreposição.
- Objetivo: encontre o subconjunto máximo de tarefas mutuamente compatíveis.



Agendamento de Intervalos: Algoritmos Gulosos

Modelo guloso. Considere tarefas em alguma ordem. Cada tarefa é escolhida obedecendo-se o mesmo critério utilizado nas escolhas prévias.

- [Tempo de início mais cedo] Considere tarefas em ordem ascendente de tempo de início s_j.
- [Tempo de fim mais cedo] Considere tarefas em ordem ascendente em tempo de fim f_j.
- [Menor intervalo] Considere tarefas em ordem ascendente de tamanho de intervalo $f_j - s_j$.
- [Menor número de conflitos] Para cada tarefa, conte o número de tarefas em conflito c_j. Agende em ordem ascendente de conflitos c_j.

Agendamento de Intervalos: Algoritmos Gulosos

Modelo guloso. Considere tarefas em alguma ordem. Cada tarefa é escolhida desde que seja compatível com as outras escolhidas previamente.

falha para:

Tempo de início mais cedo

Menor intervalo

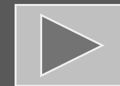
Menor número de conflitos

Agendamento de Intervalos: Algoritmos Gulosos

Algoritmo guloso. Considere tarefas em ordem crescente de tempo de término. Cada tarefa é escolhida desde que seja compatível com as outras escolhidas previamente.

```
Sort jobs by finish times so that f1 ≤ f2 ≤ ... ≤ fn.

     jobs selected
    ↙
A ← φ
for j = 1 to n {
   if (job j compatible with A)
      A ← A ∪ {j}
}
return A
```

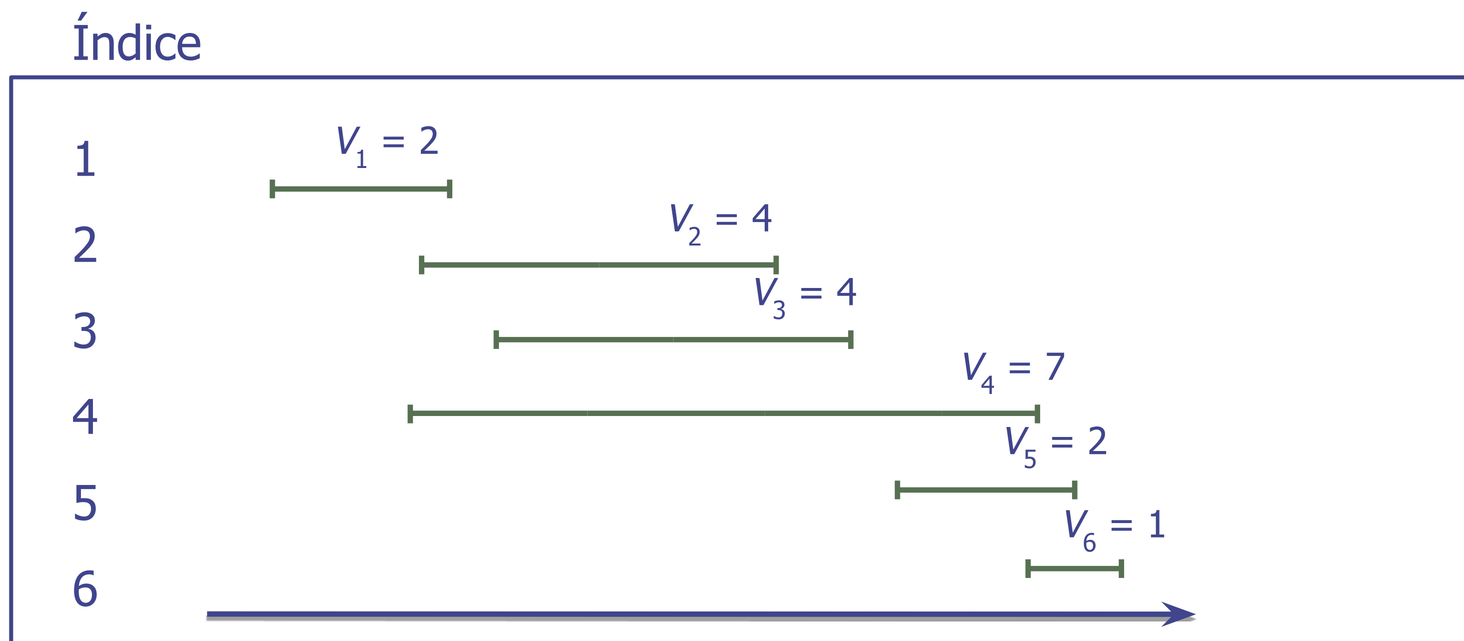


Implementação. O(n log n).

- Guarde a tarefa j* que foi adicionada por último em A.
- Tarefa j é compatível com A se $s_j \geq f_{j^*}$.

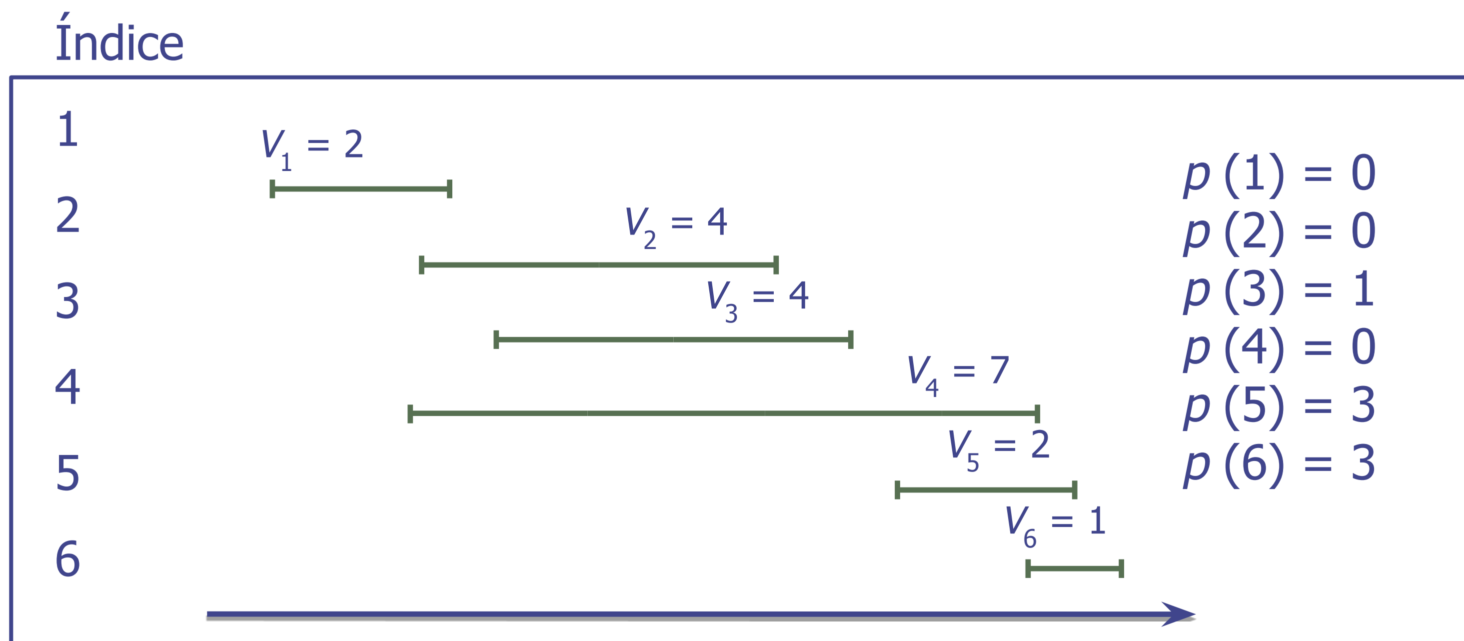
Retomando exemplo – WISP

- Vamos supor que os intervalos estão ordenados pelo tempo de término.



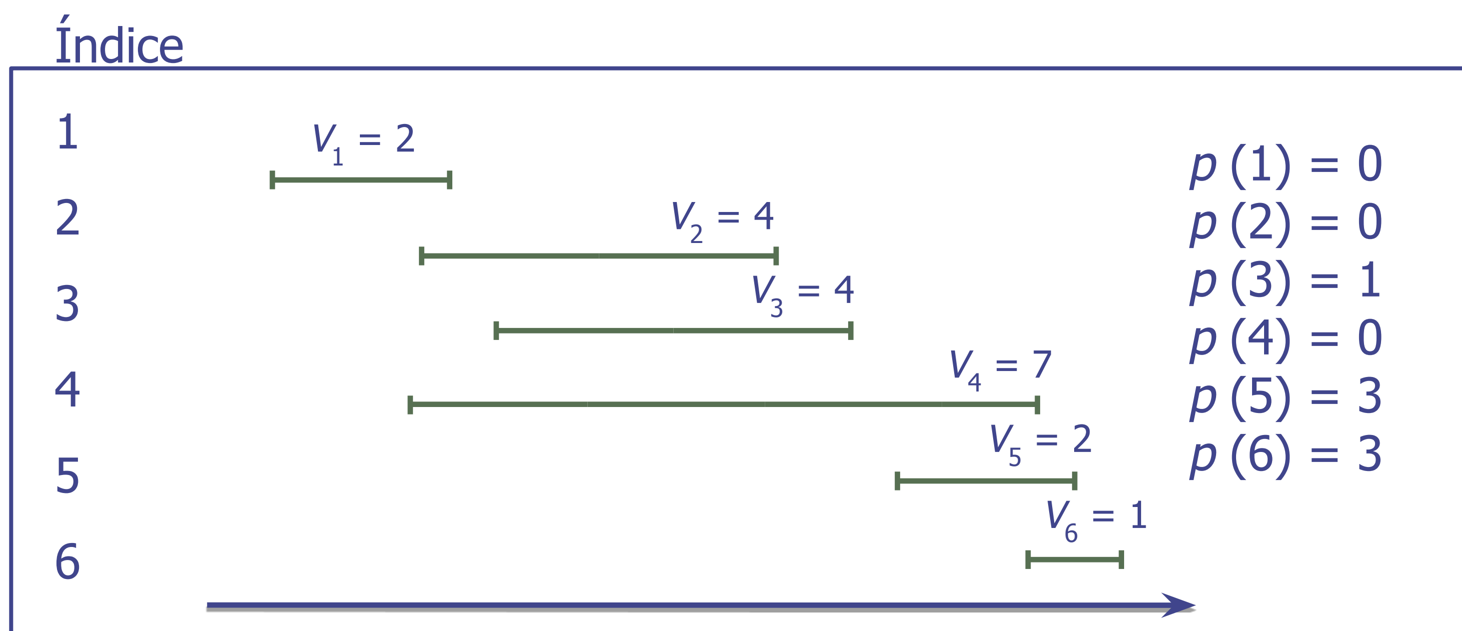
Exemplo 1 – WISP

- Vamos supor que os intervalos estão ordenados pelo tempo de término.
- E que definimos $p(j)$ como o maior índice $i < j$, tal que os intervalos i e j são disjuntos.



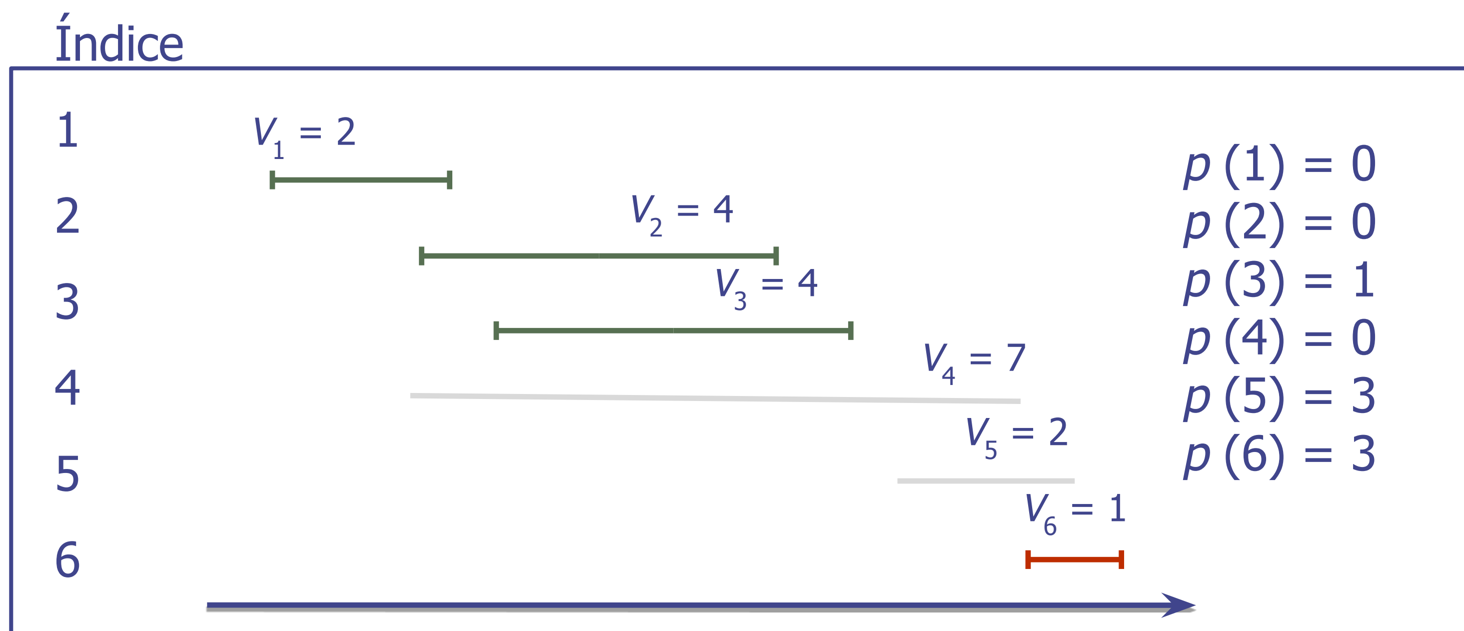
Exemplo 1 – WISP

- Um pouco mais formalmente:
  - Podemos rotular os intervalos 1,...,n
  - Estamos à procura de um sub-conjunto $S \subseteq \{1,...,n\}$ que maximize $\Sigma_{i \in S} v_i$



Exemplo 1 – WISP

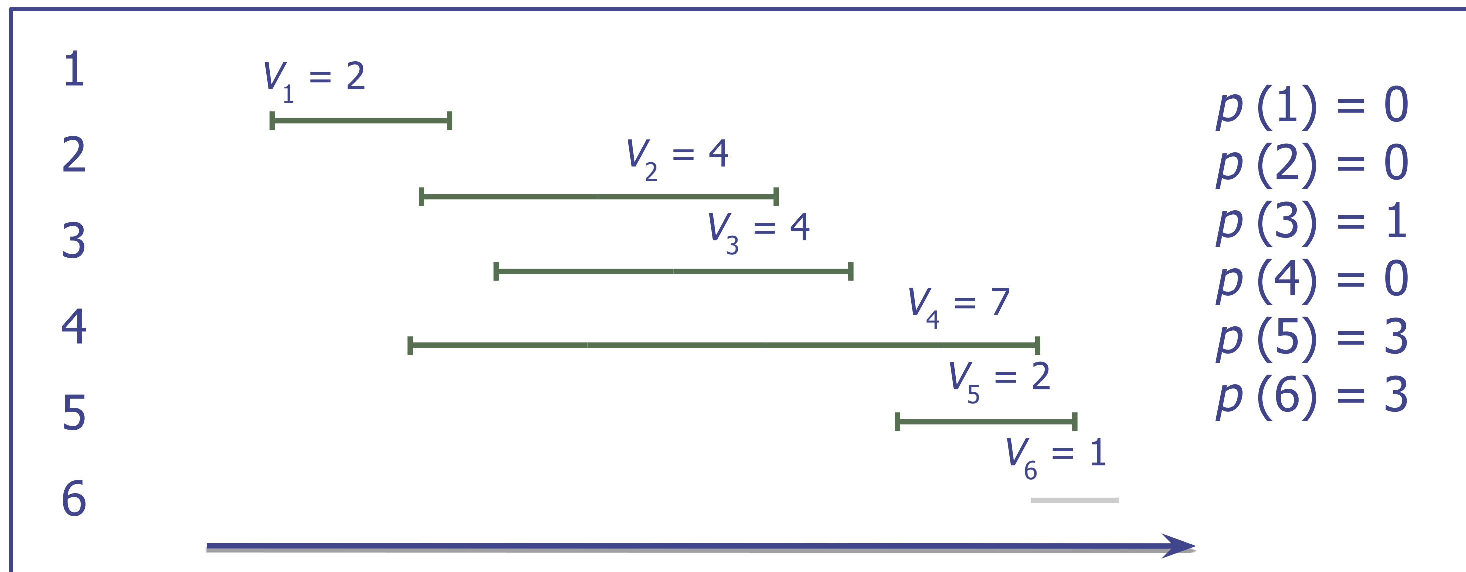
- Uma coisa óbvia que podemos dizer sobre *S*:
  - Ou *n* (último intervalo) pertence a *S*, ou *n* não pertence a *S*
  - Se $n \in S$, então nenhum intervalo entre $p(n)+1$ e $n - 1 \in S$
  - Ainda, *S* possui uma solução ótima para os intervalos $\{1, \ldots, p(n)\}$



Exemplo 1 – WISP

- Uma coisa óbvia que podemos dizer sobre *S*:
  - Ou *n* (último intervalo) pertence a *S*, ou *n* não pertence a *S*.
  - Se $n \notin S$, então existe uma solução ótima com os intervalos do conjunto $\{1, \ldots, n-1\}$

Índice



Exemplo 1 – WISP

- Encontrar uma solução ótima no intervalo $\{1, 2, \ldots, n\}$ envolve encontrar soluções ótimas em um intervalo menor $\{1, 2, \ldots, j\}$.

- Seja OPT(j) a soma ótima dos intervalos para $\{1, 2, \ldots, j\}$. Então
  - Se $j \in S$, $OPT(j) = v_j + OPT(p(j))$
  - Se $j \notin S$, $OPT(j) = OPT(j - 1)$

Exemplo 1 – WISP

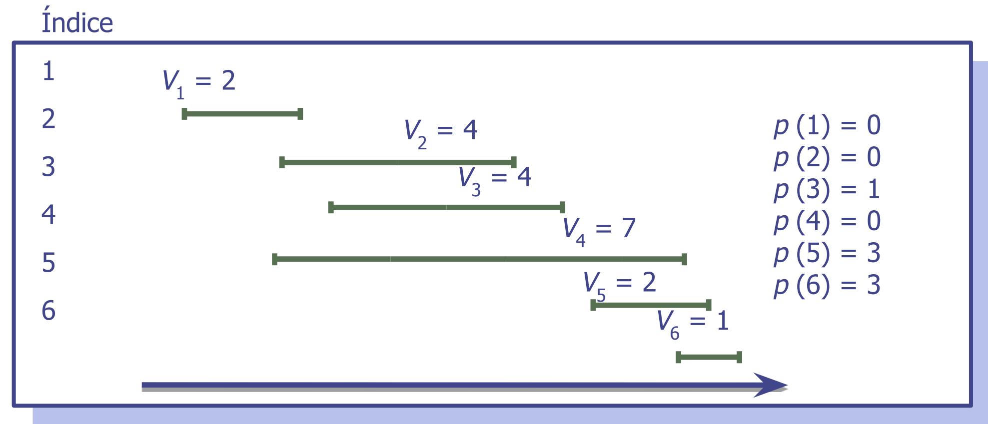
- Encontrar uma solução ótima no intervalo $\{1, 2, \dots, n\}$ envolve encontrar soluções ótimas em um intervalo menor $\{1, 2, \dots, j\}$.
- Seja OPT(j) a soma ótima dos intervalos para $\{1, 2, \dots, j\}$. Então:
  - Se $j \in S$, $OPT(j) = v_j + OPT(p(j))$
  - Se $j \notin S$, $OPT(j) = OPT(j - 1)$
- Ou seja:

$$OPT(j) = \max(v_j + OPT(p(j)), OPT(j - 1))$$

Exemplo 1 – WISP

```
Compute-Opt(j)
    if j = 0 then
        return 0
    else
        return max(v[j] + Compute-Opt(p(j)), Compute-Opt(j-1))
end
```

Exemplo 1 – WISP



OPT(6)
OPT(3) OPT(5)
OPT(1) OPT(2) OPT(3) OPT(4)
OPT(1) OPT(1) OPT(2) OPT(3)
OPT(1) OPT(1) OPT(2)
OPT(1)

Exemplo 1 – WISP

- Existe uma grande ineficiência nessa árvore de execução, por exemplo:
  - OPT(1) é calculado 6 vezes;
  - OPT(2) é calculado 3 vezes, e assim por diante.
- A complexidade do procedimento Compute-Opt é exponencial.
  - Pode-se mostrar que o número de chamadas de Compute-Opt cresce como a série de Fibonacci
  - Que por sua vez cresce exponencialmente. Um horror!

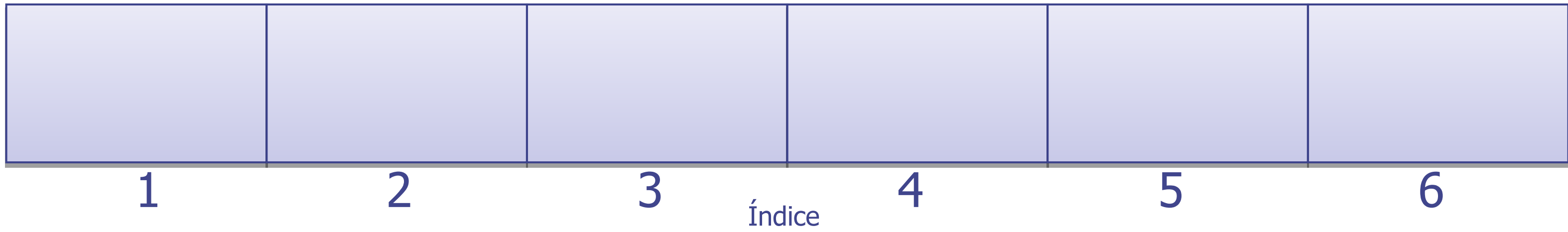
Exemplo 1 – WISP

- Na verdade Compute-Opt somente calcula as soluções para n + 1 sub-problemas. Mas faz isso repetidas vezes...

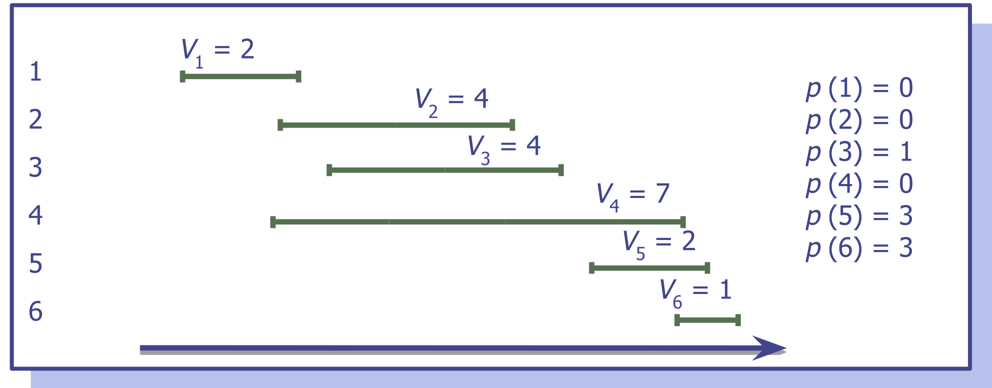
- Uma solução para esse problema é a *memoização*, que consiste em armazenar as soluções para problemas parciais em uma estrutura global.

Exemplo 1 – WISP

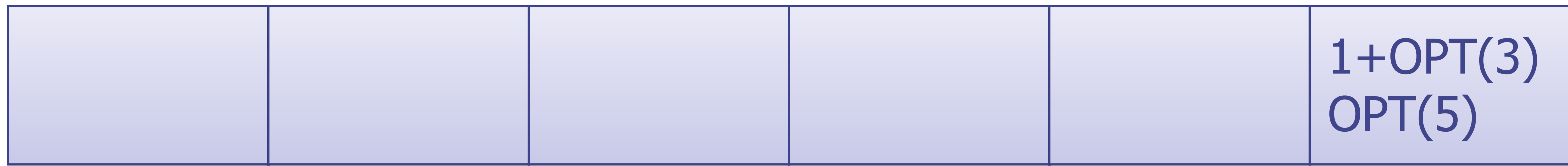
```
M-Compute-Opt(j)
    if j = 0 then
        return 0
    else if M[j] is not empty then
            return M[j]
        else
            M[j] = max(v[j] + Compute-Opt(p(j)), Compute-Opt(j-1))
            return M[j]
end
```



Índice

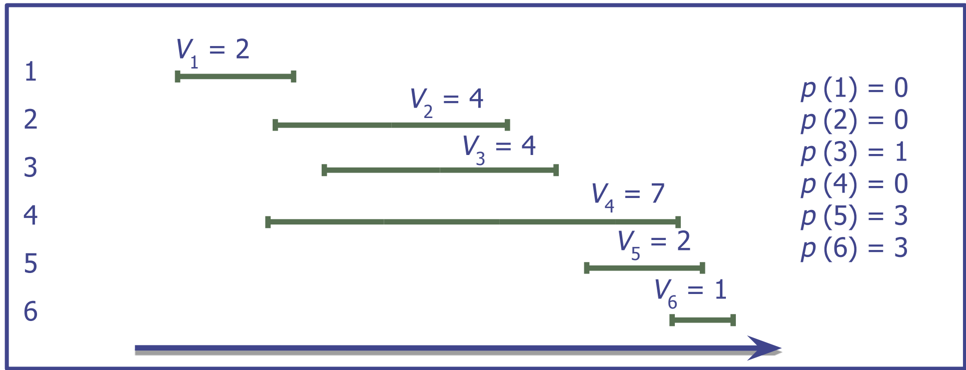


OPT(6)
OPT(3) OPT(5)
OPT(1) OPT(2) OPT(3) OPT(4)
OPT(1) OPT(1) OPT(2) OPT(3)
OPT(1) OPT(1) OPT(2)
OPT(1)



1 2 3 4 5 6

Índice



OPT(6)

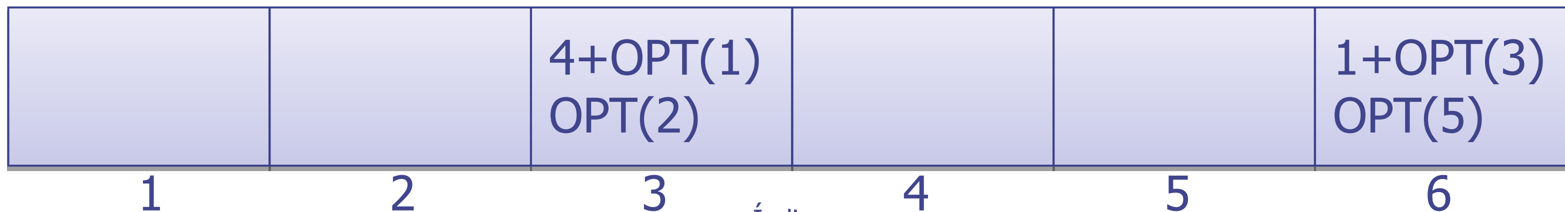
OPT(3) OPT(5)

OPT(1) OPT(2) OPT(3) OPT(4)

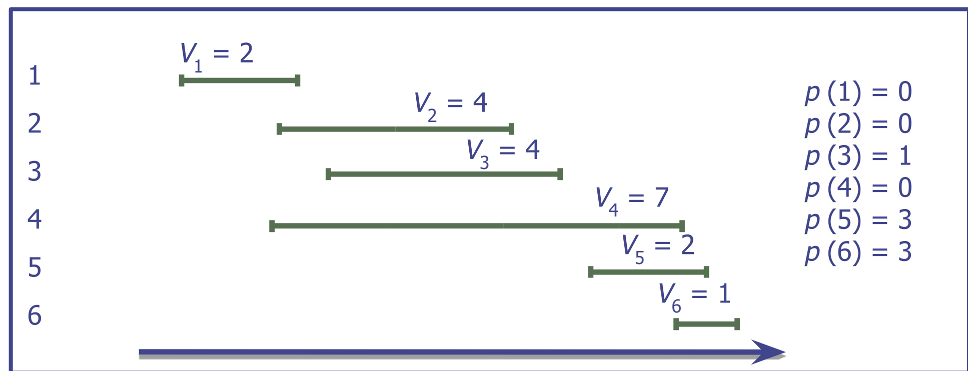
OPT(1) OPT(1) OPT(2) OPT(3)

OPT(1) OPT(1) OPT(2)

OPT(1)



Índice



OPT(6)

OPT(3) OPT(5)

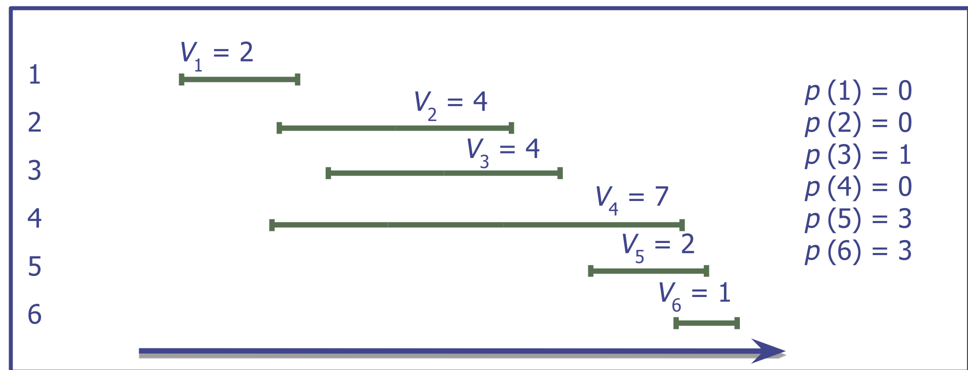
OPT(1) OPT(2) OPT(3) OPT(4)

OPT(1) OPT(1) OPT(2) OPT(3)

OPT(1) OPT(1) OPT(2)

OPT(1)

1	2	3	4	5	6
2+OPT(0) OPT(0)		4+OPT(1) OPT(2)			1+OPT(3) OPT(5)



OPT(6)

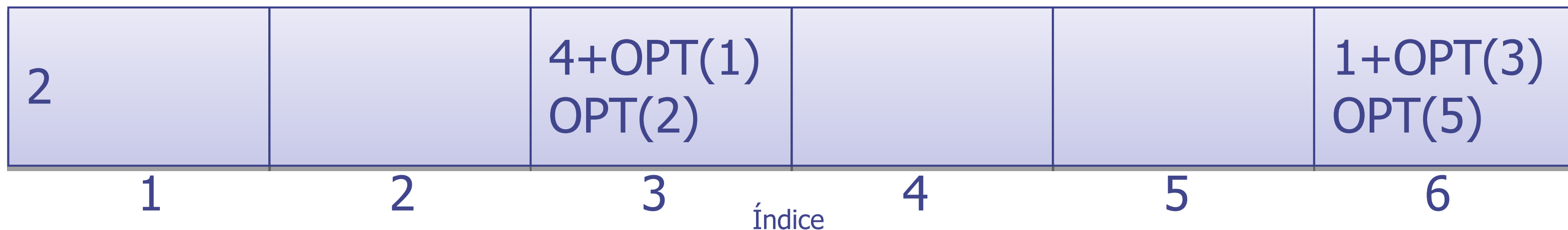
OPT(3) OPT(5)

OPT(1) OPT(2) OPT(3) OPT(4)

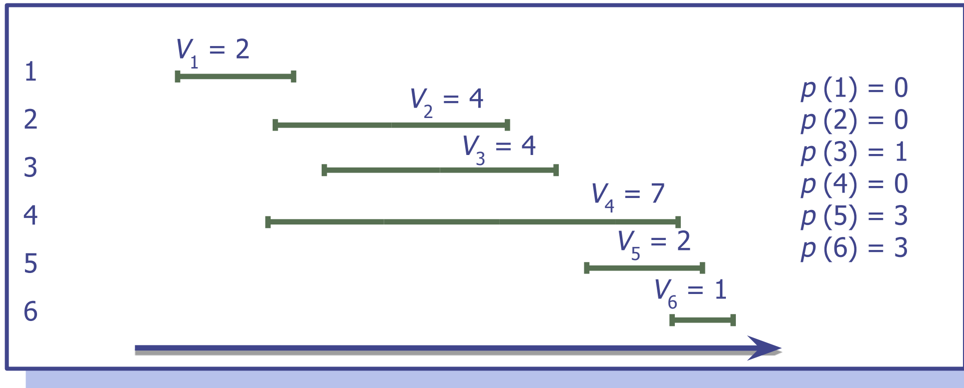
OPT(1) OPT(1) OPT(2) OPT(3)

OPT(1) OPT(1) OPT(2)

OPT(1)



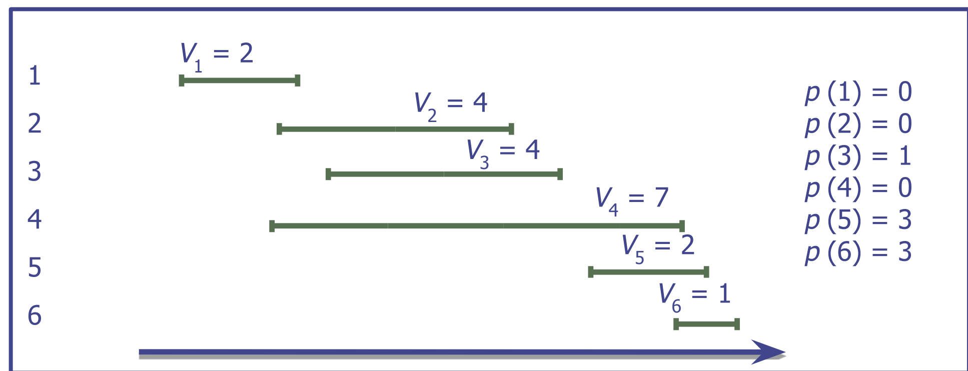
1	2	3	4	5	6
2		4+OPT(1) OPT(2)			1+OPT(3) OPT(5)



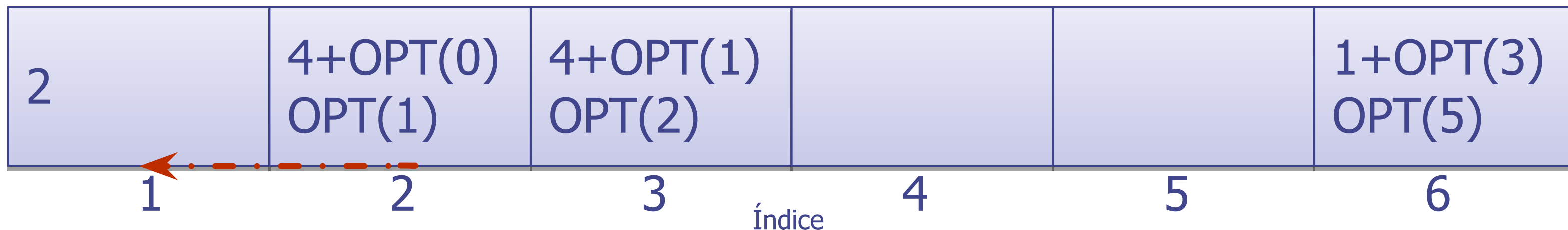
OPT(6)
OPT(3) OPT(5)
OPT(1) OPT(2) OPT(3) OPT(4)
OPT(1) OPT(1) OPT(2) OPT(3)
OPT(1) OPT(1) OPT(2)
OPT(1)

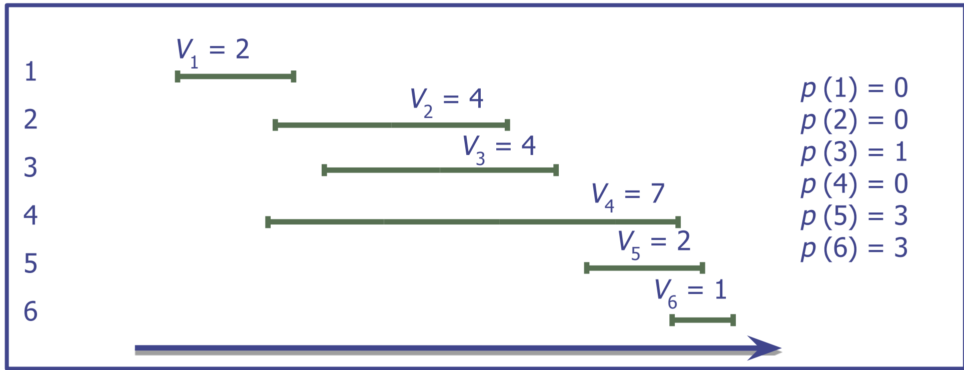
2	4+OPT(0) OPT(1)	4+OPT(1) OPT(2)			1+OPT(3) OPT(5)
1	2	3	4	5	6

Índice



OPT(6)
OPT(3) OPT(5)
OPT(1) OPT(2) OPT(3) OPT(4)
OPT(1) OPT(1) OPT(2) OPT(3)
OPT(1) OPT(1) OPT(2)
OPT(1)





OPT(6)

OPT(3) OPT(5)

OPT(1) OPT(2) OPT(3) OPT(4)

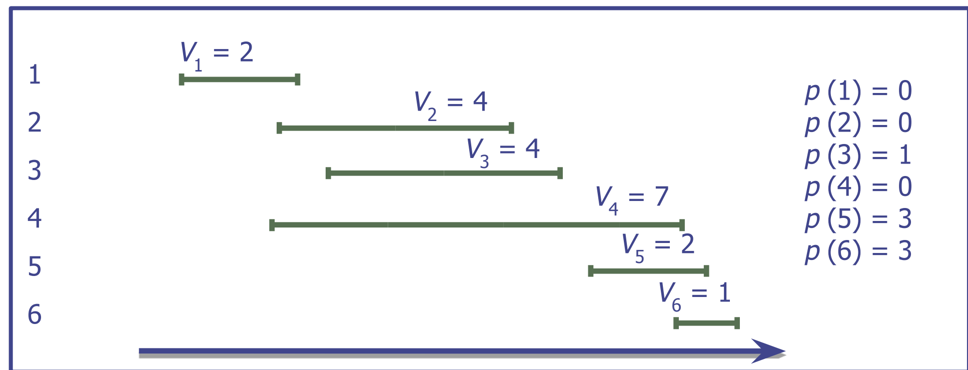
OPT(1) OPT(1) OPT(2) OPT(3)

OPT(1) OPT(1) OPT(2)

OPT(1)

2	4	4+OPT(1) OPT(2)			1+OPT(3) OPT(5)
1	2	3	4	5	6

Índice



OPT(6)

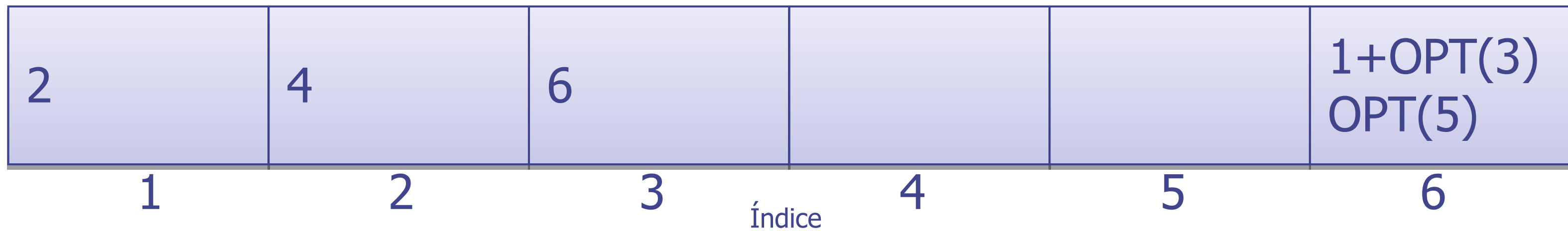
OPT(3) OPT(5)

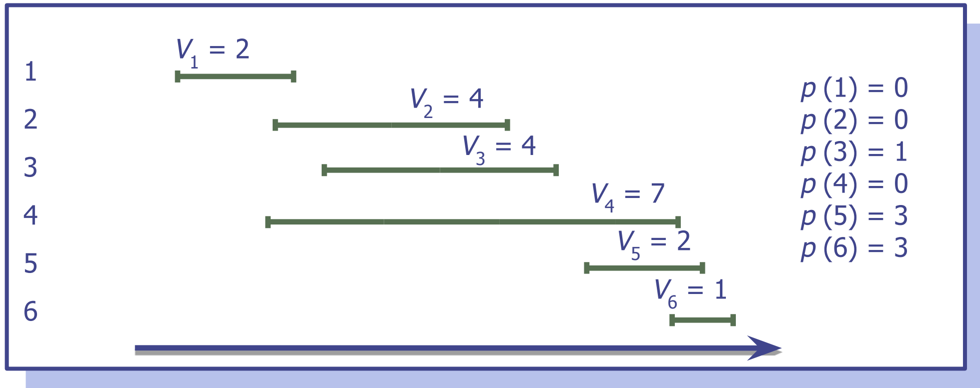
OPT(1) OPT(2) OPT(3) OPT(4)

OPT(1) OPT(1) OPT(2) OPT(3)

OPT(1) OPT(1) OPT(2)

OPT(1)





OPT(6)

OPT(3) OPT(5)

OPT(1) OPT(2) OPT(3) OPT(4)

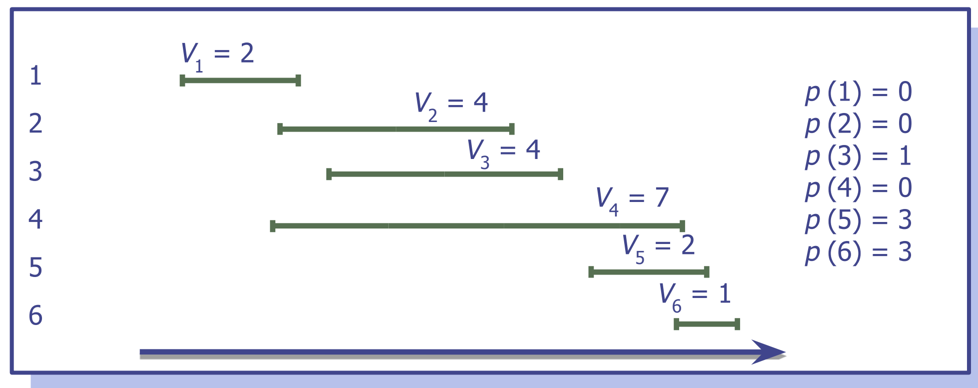
OPT(1) OPT(1) OPT(2) OPT(3)

OPT(1) OPT(1) OPT(2)

OPT(1)

1	2	3	4	5	6
2	4	6		2+OPT(3) OPT(4)	1+OPT(3) OPT(5)

Índice



OPT(6)

OPT(3) OPT(5)

OPT(1) OPT(2) OPT(3) OPT(4)

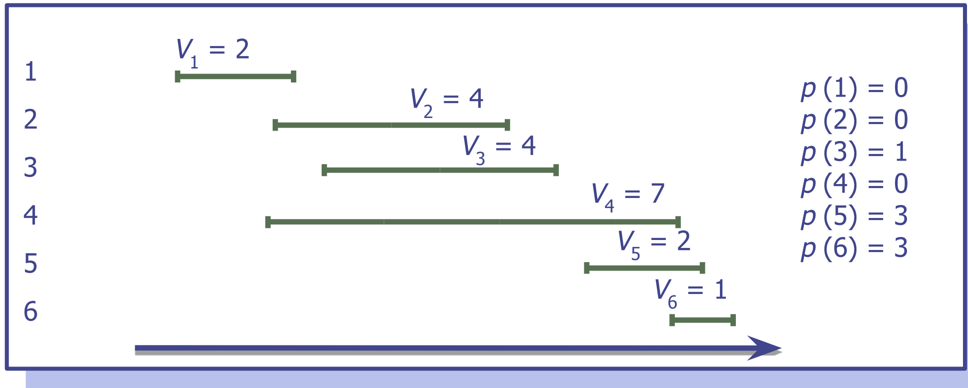
OPT(1) OPT(1) OPT(2) OPT(3)

OPT(1) OPT(1) OPT(2)

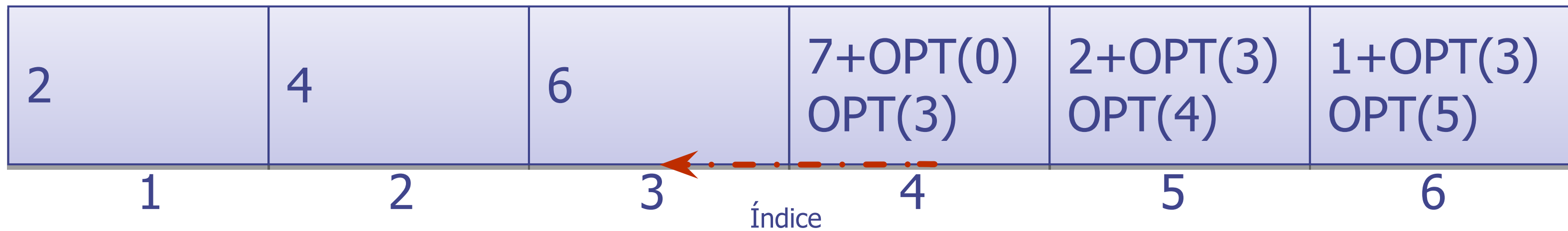
OPT(1)

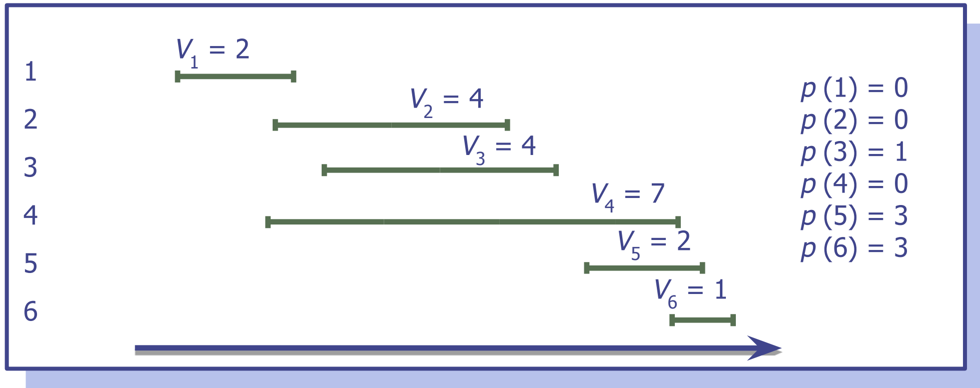
2	4	6	7+OPT(0) OPT(3)	2+OPT(3) OPT(4)	1+OPT(3) OPT(5)
1	2	3	4	5	6

Índice



OPT(6)
OPT(3) OPT(5)
OPT(1) OPT(2) OPT(3) OPT(4)
OPT(1) OPT(1) OPT(2) OPT(3)
OPT(1) OPT(1) OPT(2)
OPT(1)





OPT(6)

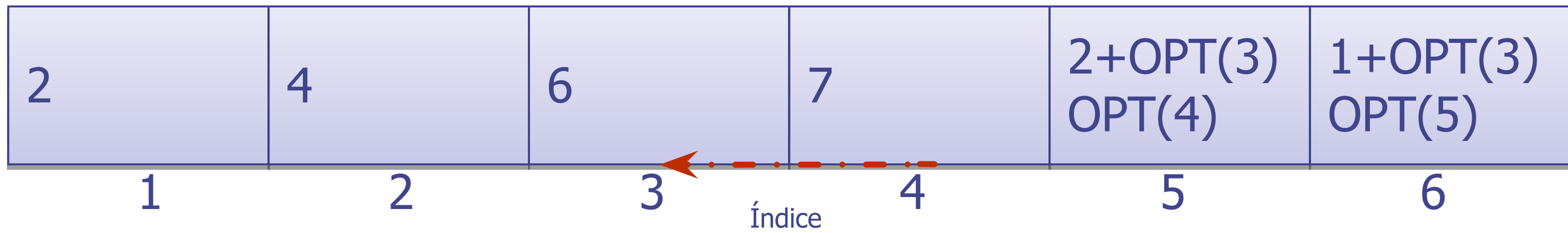
OPT(3) OPT(5)

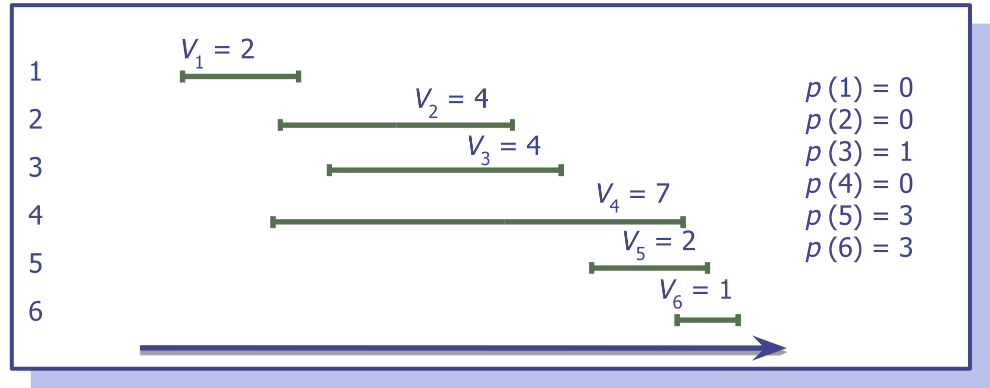
OPT(1) OPT(2) OPT(3) OPT(4)

OPT(1) OPT(1) OPT(2) OPT(3)

OPT(1) OPT(1) OPT(2)

OPT(1)





OPT(6)

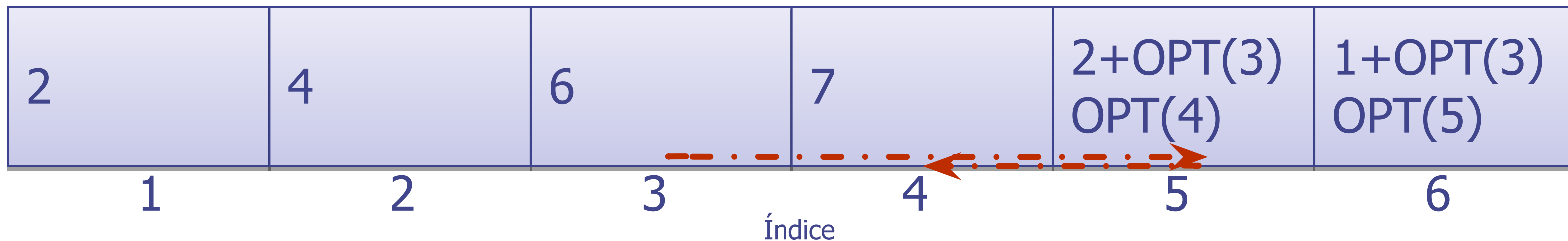
OPT(3) OPT(5)

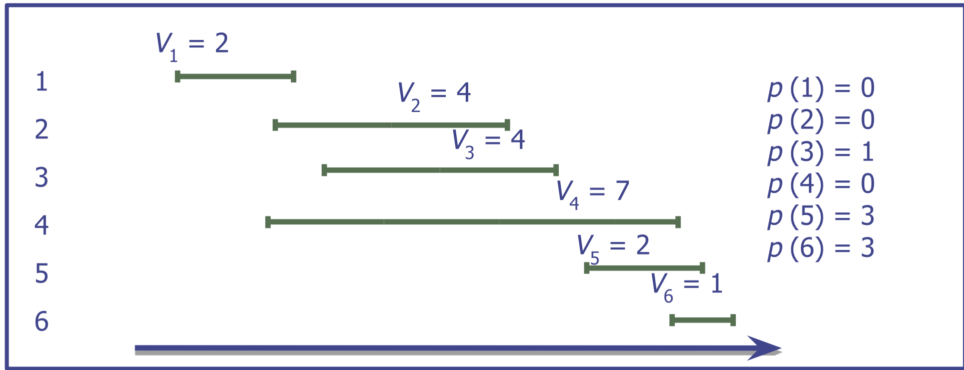
OPT(1) OPT(2) OPT(3) OPT(4)

OPT(1) OPT(1) OPT(2) OPT(3)

OPT(1) OPT(1) OPT(2)

OPT(1)





OPT(6)

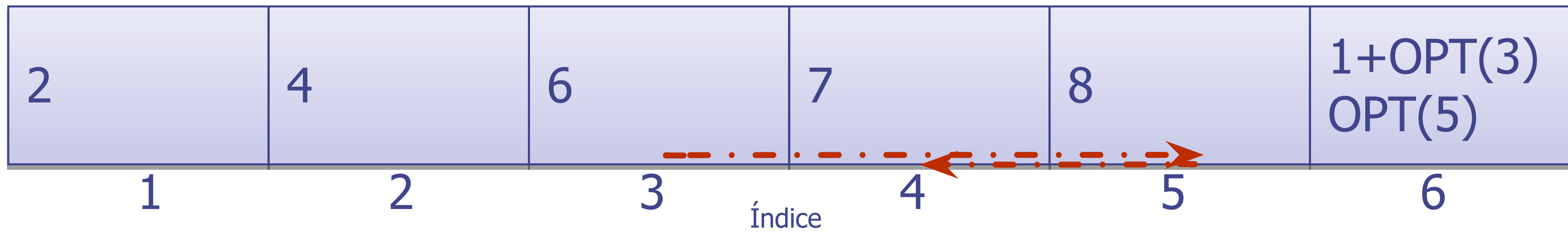
OPT(3) OPT(5)

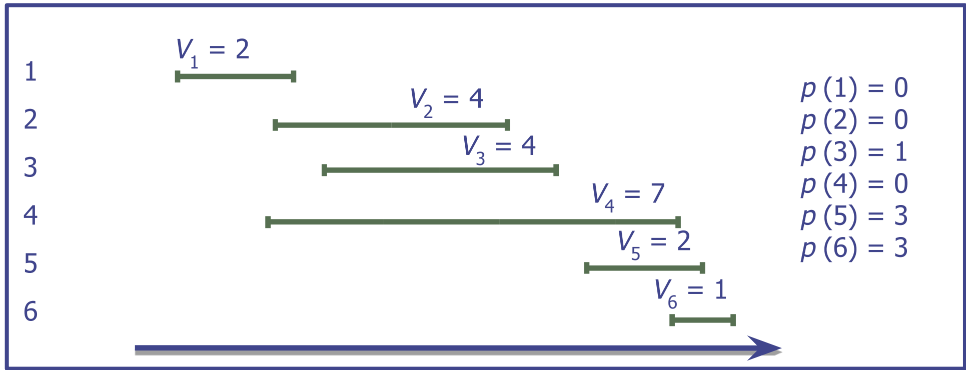
OPT(1) OPT(2) OPT(3) OPT(4)

OPT(1) OPT(1) OPT(2) OPT(3)

OPT(1) OPT(1) OPT(2)

OPT(1)





OPT(6)

OPT(3) OPT(5)

OPT(1) OPT(2) OPT(3) OPT(4)

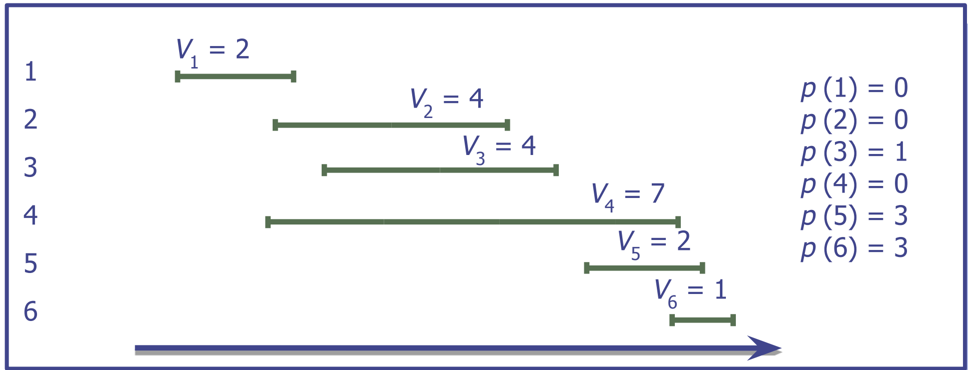
OPT(1) OPT(1) OPT(2) OPT(3)

OPT(1) OPT(1) OPT(2)

OPT(1)

2	4	6	7	8	1+OPT(3) OPT(5)
1	2	3	4	5	6

Índice



OPT(6)

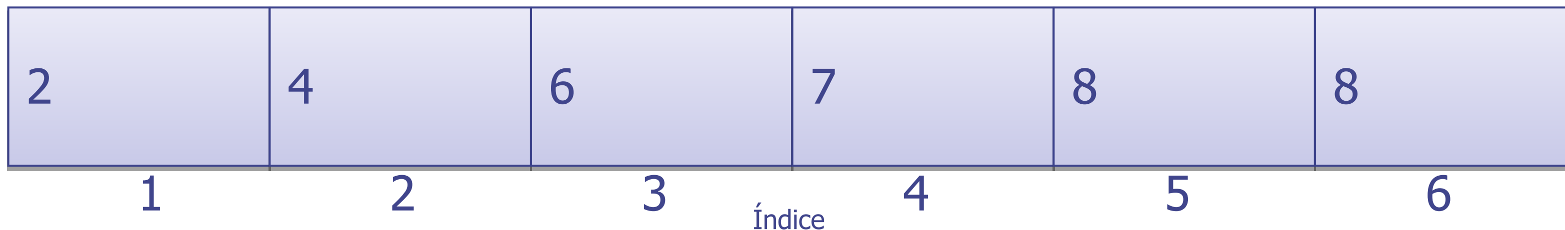
OPT(3) OPT(5)

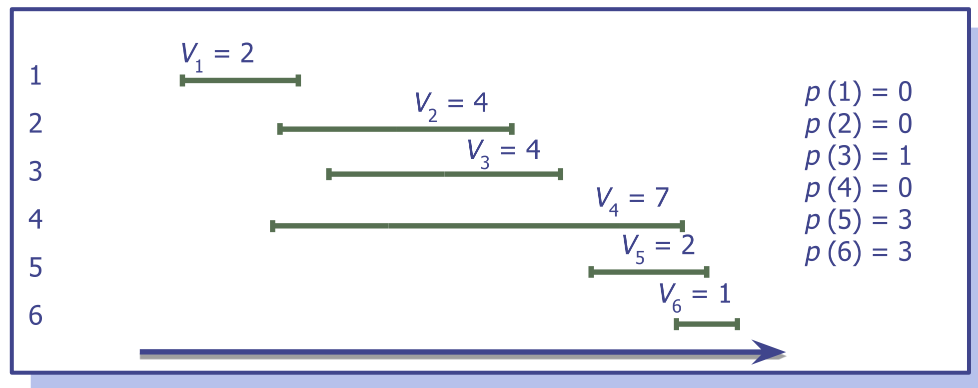
OPT(1) OPT(2) OPT(3) OPT(4)

OPT(1) OPT(1) OPT(2) OPT(3)

OPT(1) OPT(1) OPT(2)

OPT(1)





OPT(6)

OPT(3) OPT(5)

OPT(1) OPT(2) OPT(3) OPT(4)

OPT(1) OPT(1) OPT(2) OPT(3)

OPT(1) OPT(1) OPT(2)

OPT(1)

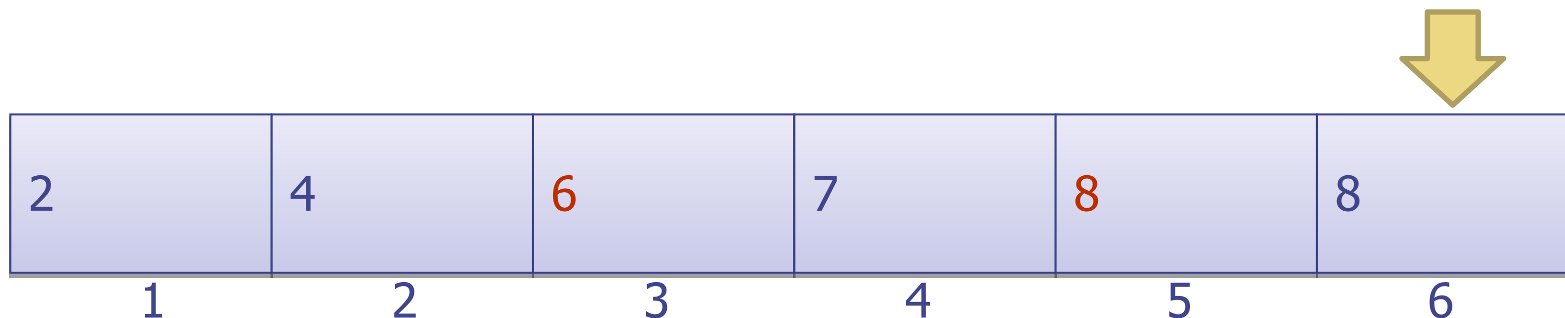
Exemplo 1 – WISP

- A solução com memoização é eficiente e requer apenas $O(n)$ passos desde que os intervalos estejam ordenados.

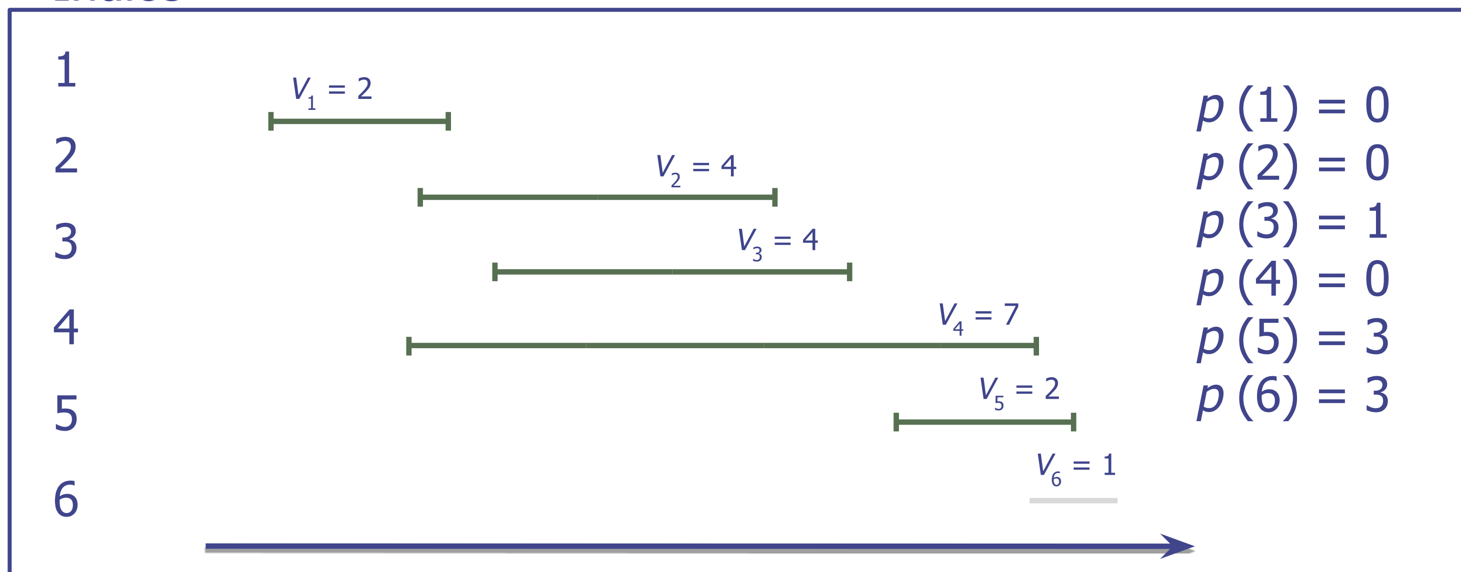
- O vetor M auxilia não somente no cálculo do valor da solução, como também pode ser utilizado para encontrar os intervalos que compõem a solução.

- Um intervalo pertence a solução se e somente se
  - $v_j + \mathrm{OPT}(p(j)) \geq \mathrm{OPT}(j-1)$

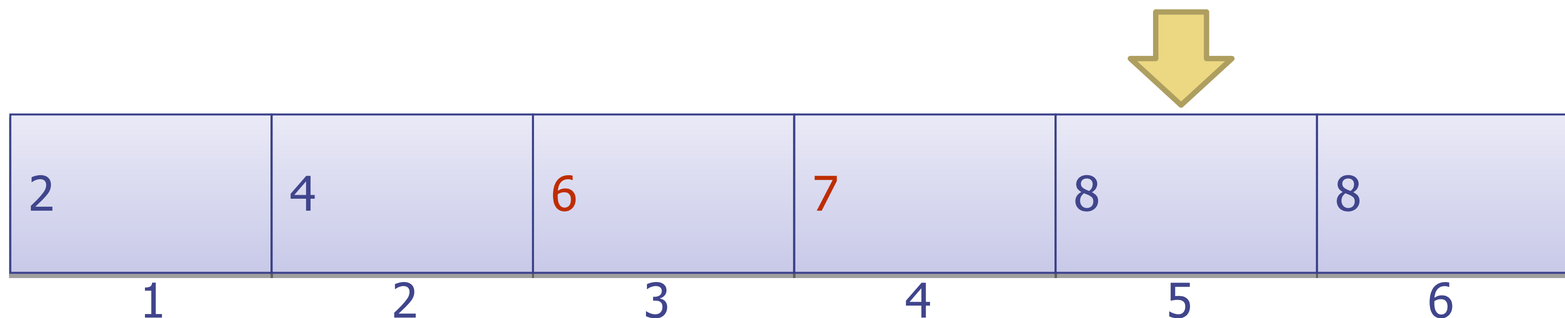
Exemplo 1 – WISP



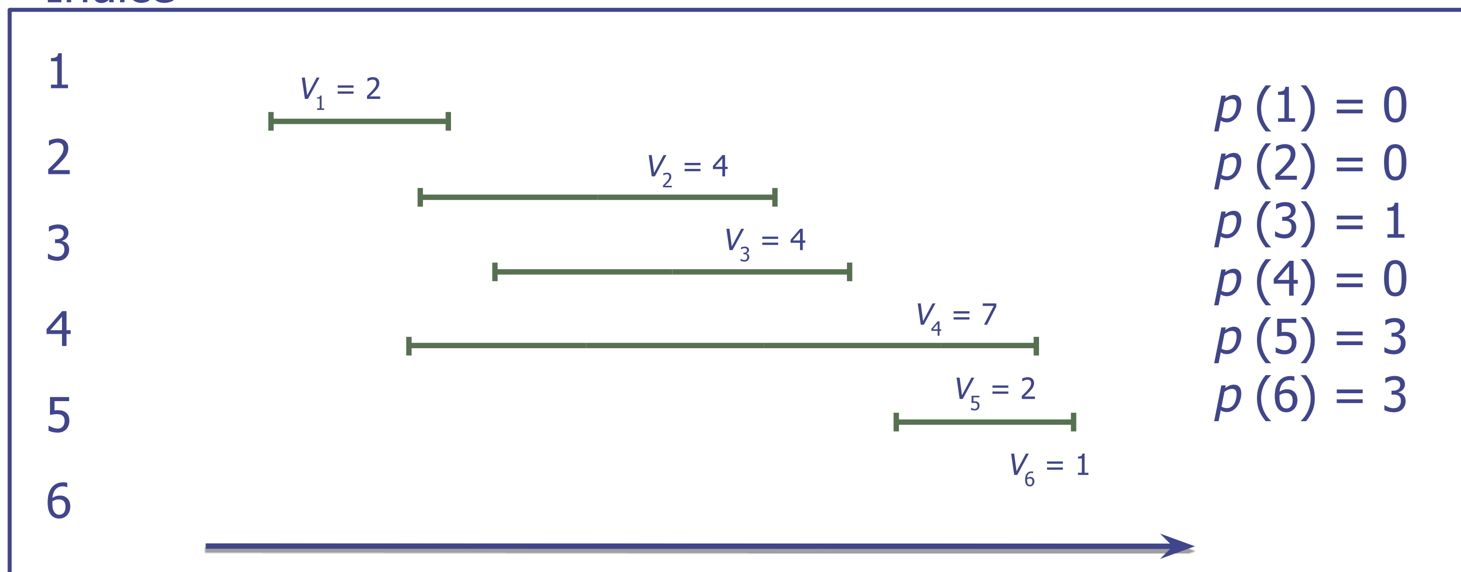
Índice



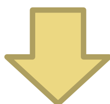
Exemplo 1 – WISP

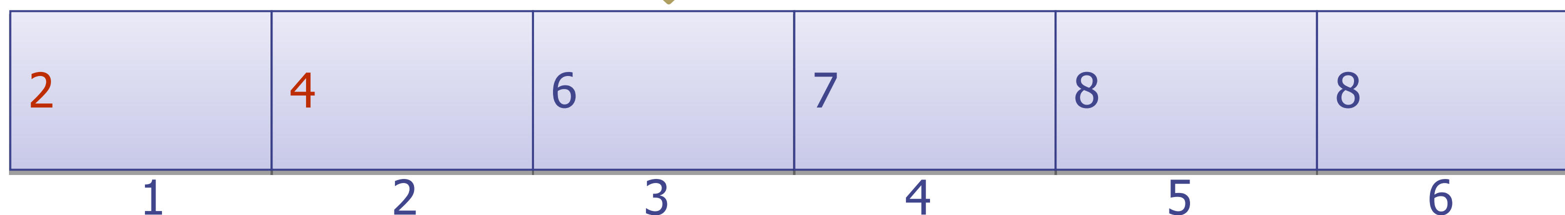


Índice

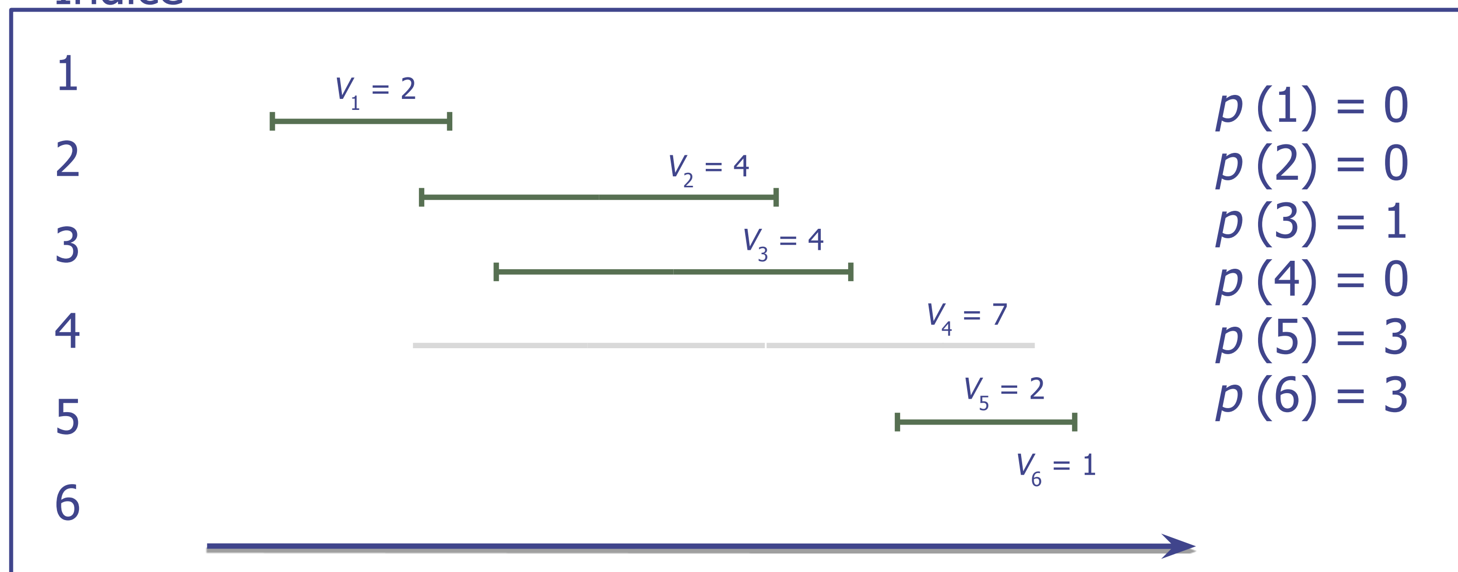


Exemplo 1 – WISP

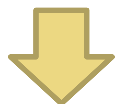


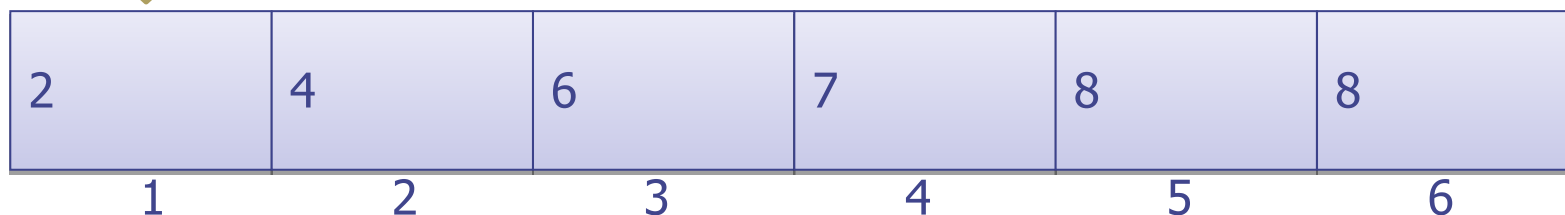


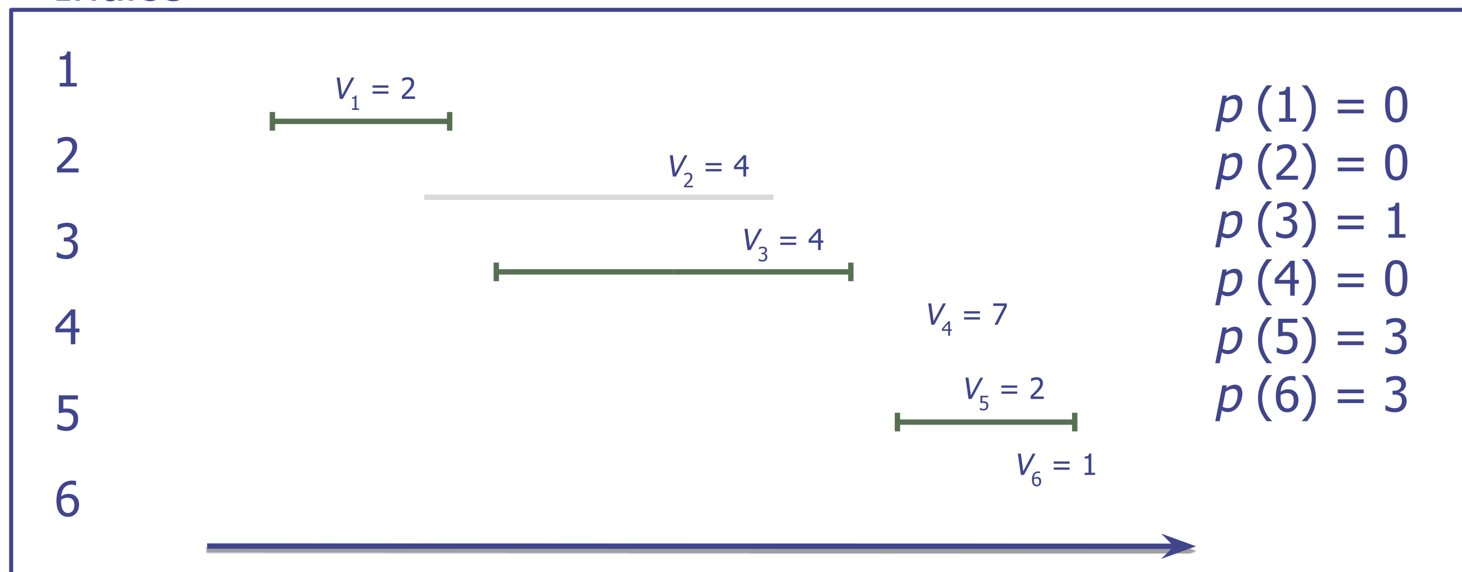
Índice



Exemplo 1 – WISP







Exemplo 1 – WISP

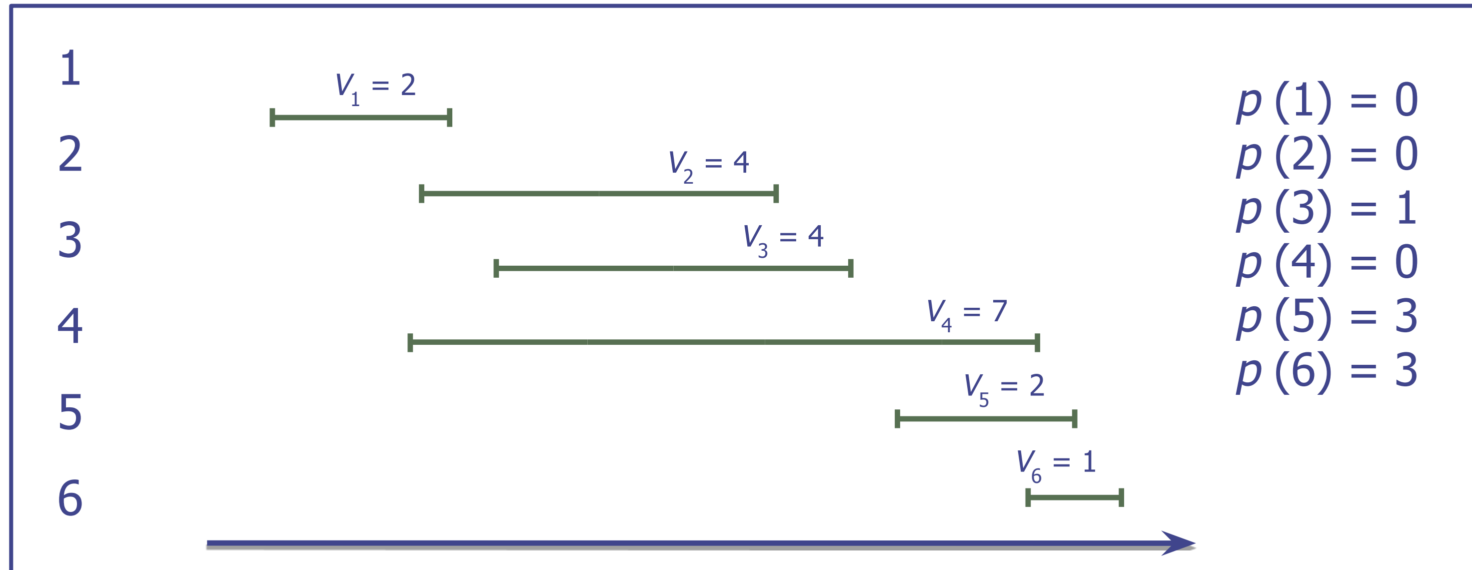
- Não discutimos ainda a complexidade do algoritmo, mas parece óbvio que é?
  - O(n), certo ???
- Mas a versão anterior faz um monte de entradas na recursão desnecessariamente…
- Isso deve ser incluído na complexidade????

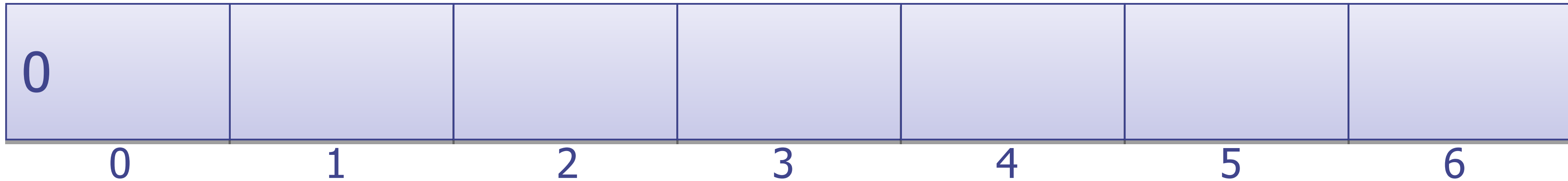
Exemplo 1 – WISP

- Entretanto, a verdadeira Programação Dinâmica é caracterizada por:
  - Memoização
  - Iteração sobre sub-problemas
- Pode-se substituir a recursão por uma iteração, essa é a forma mais simples e eficiente de PD.

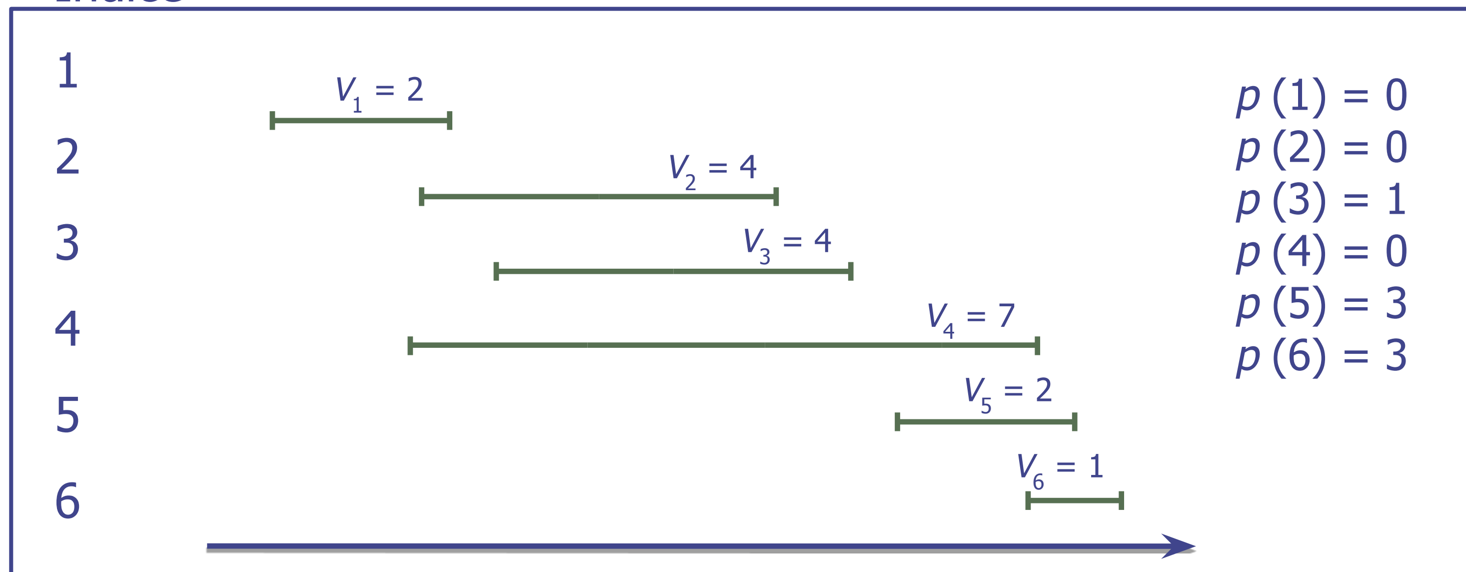
```
Iterative-COmpute-Opt
    M[0] = 0
    for j = 1,2, ... , n
        M[j] = max(v[j]+M[p(j)], M[j-1])
end
```

```
Iterative-COmpute-Opt
    M[0] = 0
    for j = 1,2, ... , n
        M[j] = max(v[j]+M[p(j)], M[j-1])
end
```

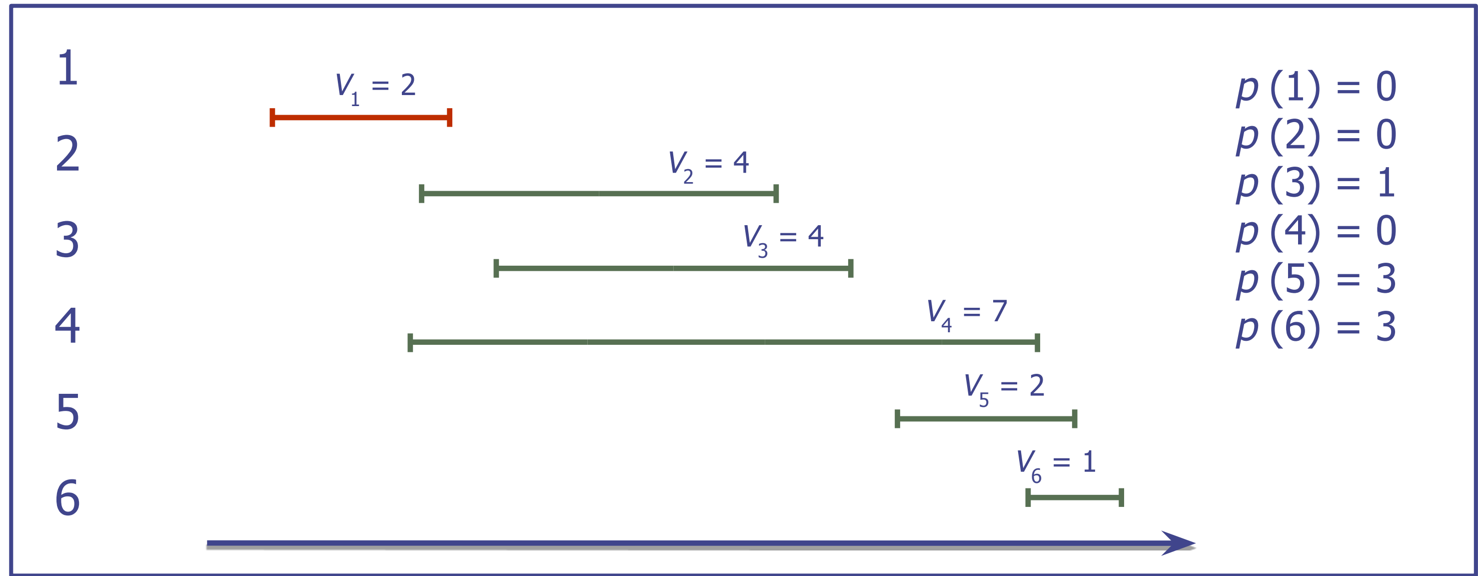





```
Iterative-COmpute-Opt
    M[0] = 0
    for j = 1,2, ... , n
        M[j] = max(v[j]+M[p(j)], M[j-1])
end
```

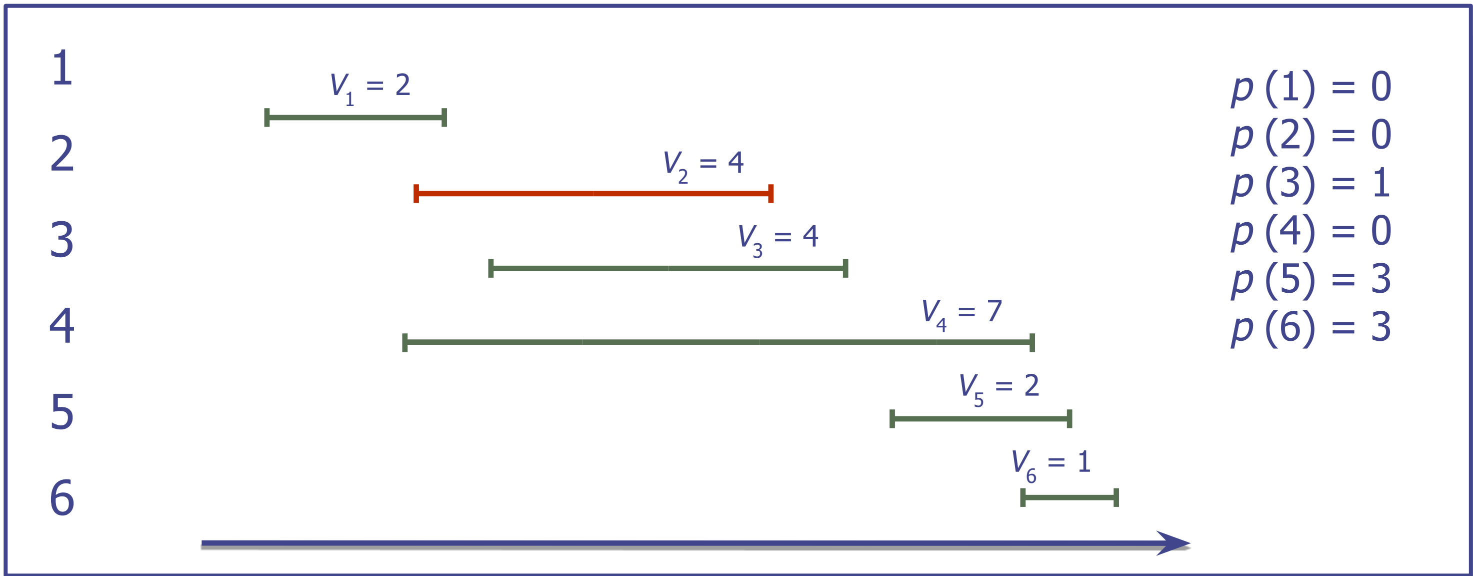


```
Iterative-COmpute-Opt
    M[0] = 0
    for j = 1,2, ... , n
        M[j] = max(v[j]+M[p(j)], M[j-1])
end
```

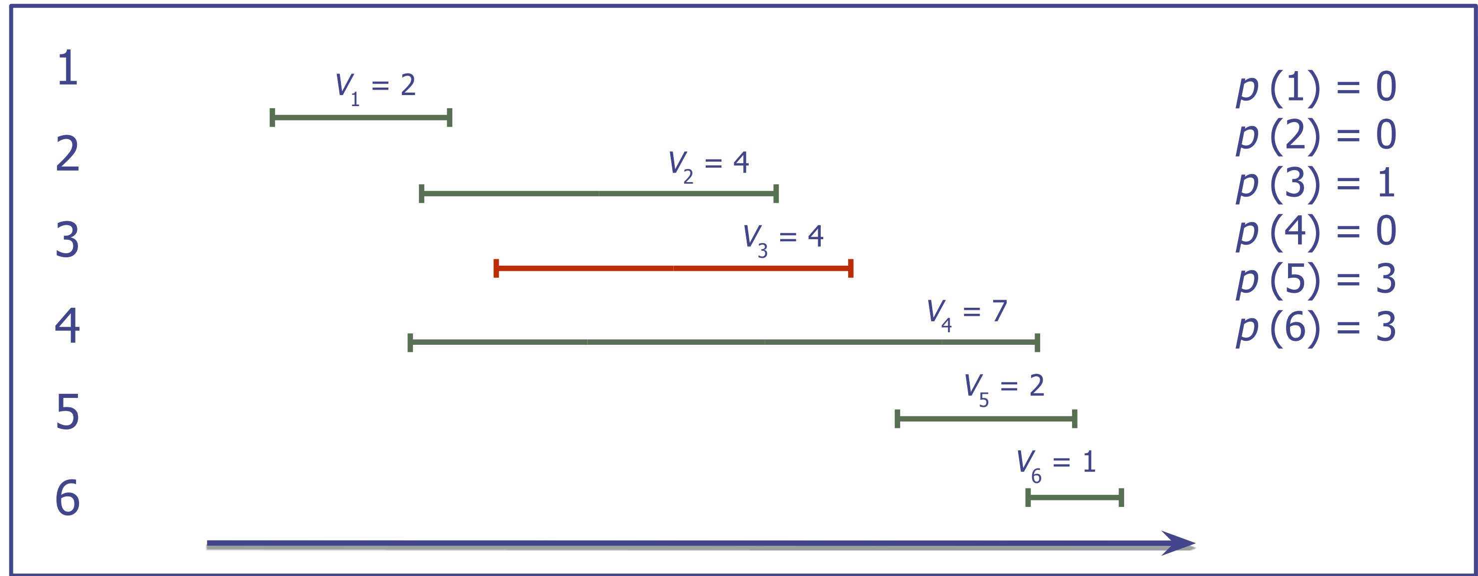


```
Iterative-COmpute-Opt
    M[0] = 0
    for j = 1,2, ... , n
        M[j] = max(v[j]+M[p(j)], M[j-1])
end
```

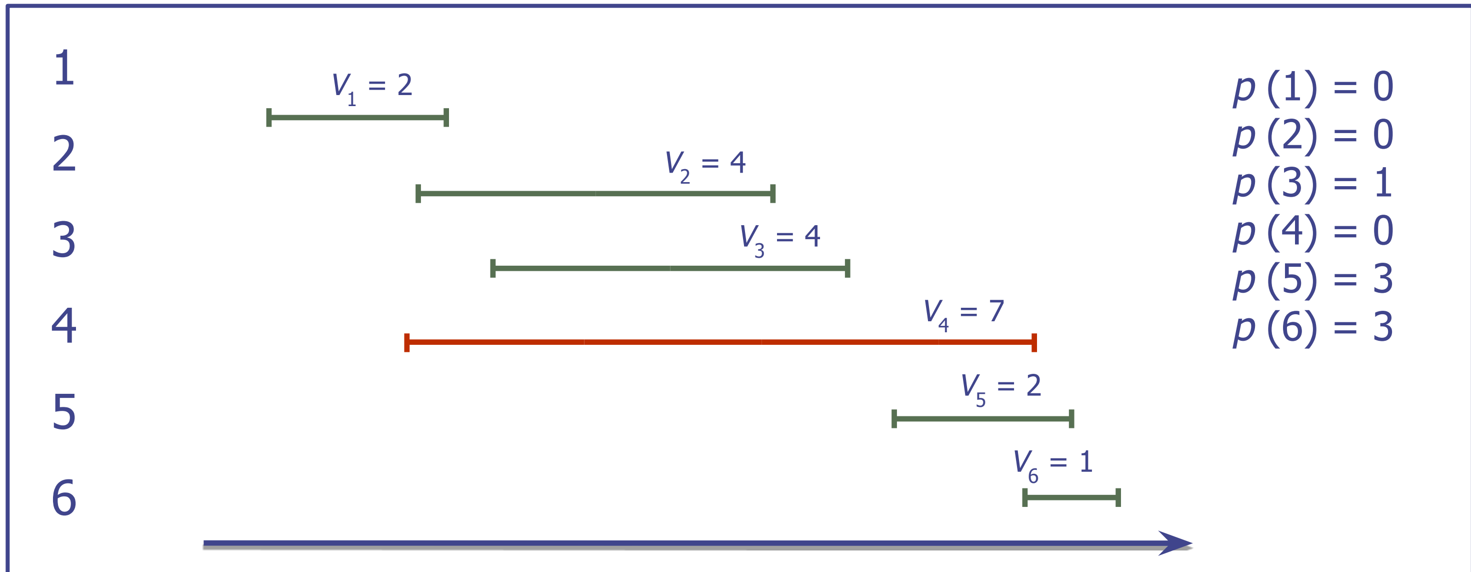
Índice



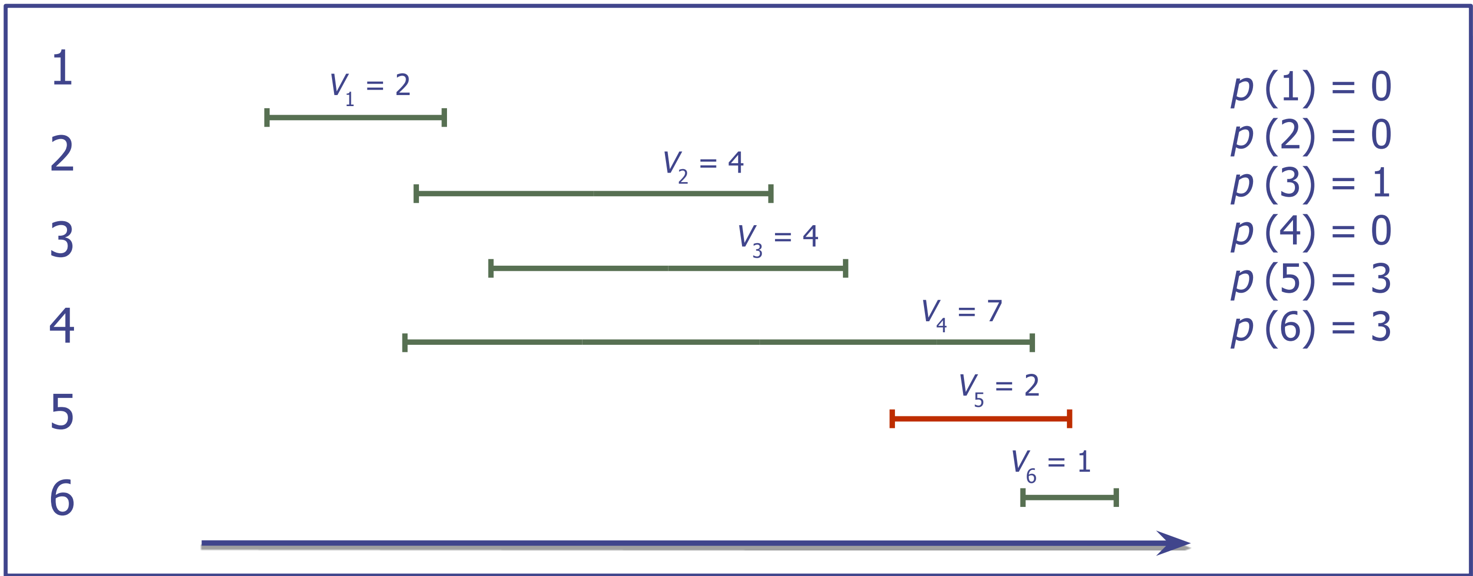
```
Iterative-COmpute-Opt
    M[0] = 0
    for j = 1,2, ... , n
        M[j] = max(v[j]+M[p(j)], M[j-1])
end
```



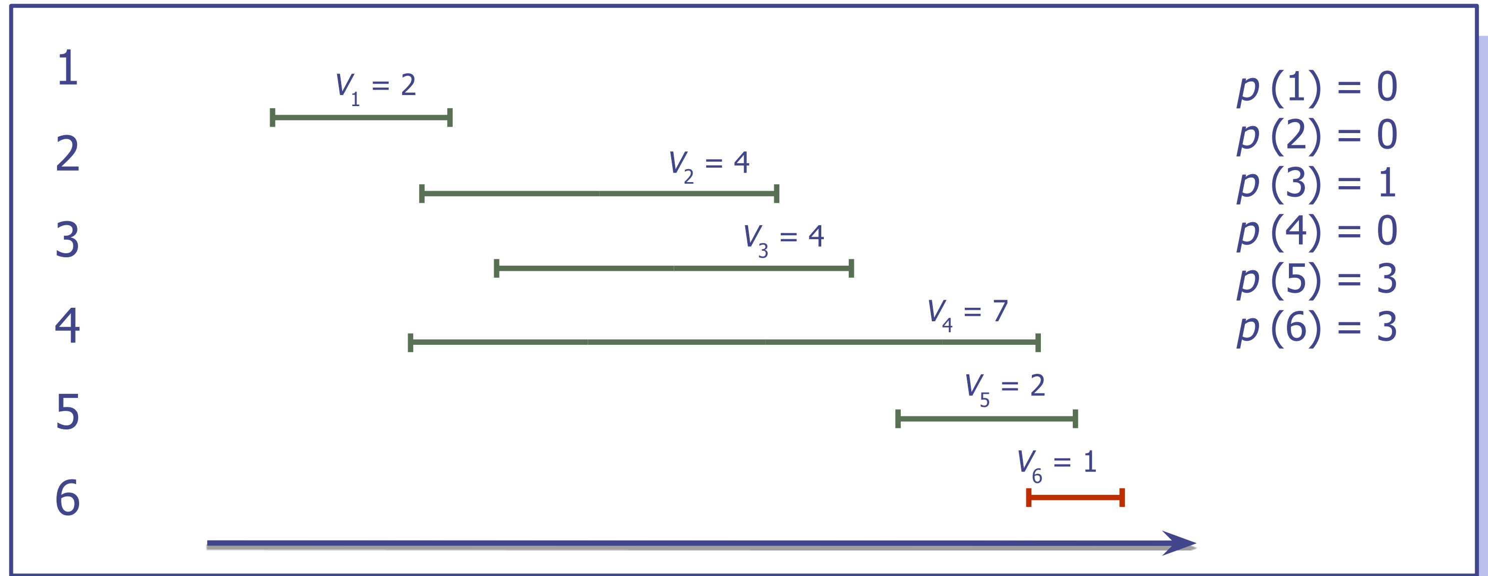
```
Iterative-COmpute-Opt
    M[0] = 0
    for j = 1,2, ... , n
        M[j] = max(v[j]+M[p(j)], M[j-1])
end
```



```
Iterative-COmpute-Opt
    M[0] = 0
    for j = 1,2, ... , n
        M[j] = max(v[j]+M[p(j)], M[j-1])
end
```



```
Iterative-COmpute-Opt
    M[0] = 0
    for j = 1,2, ... , n
        M[j] = max(v[j]+M[p(j)], M[j-1])
end
```



Observações

- Tanto a solução com recursiva memoização quanto a solução iterativa são $O(n)$.
- De uma forma geral, o principal passo para resolução do problema é encontrar a relação de recorrência.
- Dada essa relação, escolher pela solução recursiva com memoização ou a solução iterativa é basicamente uma escolha de abordagem *top-down* (recursiva) ou *bottom-up* (iterativa).
- A solução iterativa é preferida, por ser simples e não requerer o esforço computacional da recursão.

Exemplo 2 - WSSP

- Um segundo problema que pode ser resolvido com PD é chamado de Soma de Subconjuntos com Pesos (Weighted Subset Sum Problem – WSSP).
- É uma variação do problema da mochila. Dados:
  - Um conjunto de itens {1, ..., *n*}, cada um com um peso w_i
  - A mochila tem capacidade W
  - Deseja-se encontrar um subconjunto S, tal que $\Sigma_{i \in S}\, w_i$ é máxima, mas inferior a W.
- Existe alguma solução *greedy* para esse problema?

Exemplo 2 - WSSP

- Utilizando o mesmo raciocínio para o problema anterior:
  - Se $n \notin S$, então OPT(n) = OPT(n - 1)
  - Se $n \in$ S, então …

Exemplo 2 - WSSP

- Utilizando o mesmo raciocínio para o problema anterior:
  - Se $n \notin S$, então OPT(n) = OPT(n - 1)
  - Se $n \in$ S, então ...
- O problema aqui é que aceitar a requisição n significa ter menos espaço disponível.

Exemplo 2 - WSSP

- Portanto:
  - Se $n \notin S$, então OPT(n) = OPT($n - 1$, W)
  - Se $n \in$ S, então OPT(n) = OPT($n - 1$, W - w_n)
  - Uma observação importante é que um determinado item pode ser muito grande, ou seja $W < w_n$, nesse caso o item deve ser ignorado.
  - Tem-se a seguinte recorrência:

Se $W < w_i$ então OPT(i, W) = OPT(i - 1, W), senão
OPT(i, W) = max(OPT($i - 1$, W), w_i + OPT($i - 1$, $W - w_i$))

Exemplo 2 - WSSP

```
int wssp(int n, int w) {
    printf("%d **** %d\n", n, w);
    for (int i=1; i<=n; i++) // para cada item
        for (int peso=0; peso<=w; peso++)    // para cada peso
            if (W[i] > peso)
                memo[i][peso] = memo[i-1][peso];
            else
                memo[i][peso] = max(memo[i-1][peso], W[i] + memo[i-1][peso-W[i]]);

    return memo[n][w];
}
```

```
for (int i=1; i<=n; i++) // para cada item
    for (int peso=0; peso<=w; peso++)    // para cada peso
        if (W[i] > peso)
            memo[i][peso] = memo[i-1][peso];
        else
            memo[i][peso] = max(memo[i-1][peso], W[i] + memo[i-1][peso-W[i]]);
```

- Suponha:
  - $W = 6$ e
  - $w_1 = 2$, $w_2 = 2$ e $w_3 = 3$.

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

w

```
for (int i=1; i<=n; i++) // para cada item
    for (int peso=0; peso<=w; peso++)    // para cada peso
        if (W[i] > peso)
            memo[i][peso] = memo[i-1][peso];
        else
            memo[i][peso] = max(memo[i-1][peso], W[i] + memo[i-1][peso-W[i]]);
```

- Suponha:
  - $W = 6$ e
  - $w_1 = 2$, $w_2 = 2$ e $w_3 = 3$.

i \ w	0	1	2	3	4	5	6
3							
2							
1	0						
0	0	0	0	0	0	0	0

```
for (int i=1; i<=n; i++) // para cada item
    for (int peso=0; peso<=w; peso++)    // para cada peso
        if (W[i] > peso)
            memo[i][peso] = memo[i-1][peso];
        else
            memo[i][peso] = max(memo[i-1][peso], W[i] + memo[i-1][peso-W[i]]);
```

- Suponha:
  - $W = 6$ e
  - $w_1 = 2$, $w_2 = 2$ e $w_3 = 3$.

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

w

```
for (int i=1; i<=n; i++) // para cada item
    for (int peso=0; peso<=w; peso++)     // para cada peso
        if (W[i] > peso)
            memo[i][peso] = memo[i-1][peso];
        else
            memo[i][peso] = max(memo[i-1][peso], W[i] + memo[i-1][peso-W[i]]);
```

- Suponha:
  - $W = 6$ e
  - $w_1 = 2$, $w_2 = 2$ e $w_3 = 3$.

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

w

```
for (int i=1; i<=n; i++) // para cada item
    for (int peso=0; peso<=w; peso++)     // para cada peso
        if (W[i] > peso)
            memo[i][peso] = memo[i-1][peso];
        else
            memo[i][peso] = max(memo[i-1][peso], W[i] + memo[i-1][peso-W[i]]);
```

- Suponha:
  - $W = 6$ e
  - $w_1 = 2$, $w_2 = 2$ e $w_3 = 3$.

i \ w	0	1	2	3	4	5	6
3							
2							
1	0	0	2	2			
0	0	0	0	0	0	0	0

```
for (int i=1; i<=n; i++) // para cada item
    for (int peso=0; peso<=w; peso++)     // para cada peso
        if (W[i] > peso)
            memo[i][peso] = memo[i-1][peso];
        else
            memo[i][peso] = max(memo[i-1][peso], W[i] + memo[i-1][peso-W[i]]);
```

- Suponha:
  - $W = 6$ e
  - $w_1 = 2$, $w_2 = 2$ e $w_3 = 3$.

i \ w	0	1	2	3	4	5	6
3							
2							
1	0	0	2	2	2	2	2
0	0	0	0	0	0	0	0

```
for (int i=1; i<=n; i++) // para cada item
    for (int peso=0; peso<=w; peso++)     // para cada peso
        if (W[i] > peso)
            memo[i][peso] = memo[i-1][peso];
        else
            memo[i][peso] = max(memo[i-1][peso], W[i] + memo[i-1][peso-W[i]]);
```

- Suponha:
  - $W = 6$ e
  - $w_1 = 2$, $w_2 = 2$ e $w_3 = 3$.

i							
3							
2	0	0	2				
1	0	0	2	2	2	2	2
0	0	0	0	0	0	0	0
w	0	1	2	3	4	5	6

```
for (int i=1; i<=n; i++) // para cada item
    for (int peso=0; peso<=w; peso++)     // para cada peso
        if (W[i] > peso)
            memo[i][peso] = memo[i-1][peso];
        else
            memo[i][peso] = max(memo[i-1][peso], W[i] + memo[i-1][peso-W[i]]);
```

- Suponha:
  - $W = 6$ e
  - $w_1 = 2$, $w_2 = 2$ e $w_3 = 3$.

i \ w	0	1	2	3	4	5	6
3							
2	0	0	2	2			
1	0	0	2	2	2	2	2
0	0	0	0	0	0	0	0

```
for (int i=1; i<=n; i++) // para cada item
    for (int peso=0; peso<=w; peso++)    // para cada peso
        if (W[i] > peso)
            memo[i][peso] = memo[i-1][peso];
        else
            memo[i][peso] = max(memo[i-1][peso], W[i] + memo[i-1][peso-W[i]]);
```

- Suponha:
  - $W = 6$ e
  - $w_1 = 2$, $w_2 = 2$ e $w_3 = 3$.

i \ w	0	1	2	3	4	5	6
3							
2	0	0	2	2	4		
1	0	0	2	2	2	2	2
0	0	0	0	0	0	0	0

```
for (int i=1; i<=n; i++) // para cada item
    for (int peso=0; peso<=w; peso++)    // para cada peso
        if (W[i] > peso)
            memo[i][peso] = memo[i-1][peso];
        else
            memo[i][peso] = max(memo[i-1][peso], W[i] + memo[i-1][peso-W[i]]);
```

- Suponha:
  - $W = 6$ e
  - $w_1 = 2$, $w_2 = 2$ e $w_3 = 3$.

i							
3							
2	0	0	2	2	4	4	4
1	0	0	2	2	2	2	2
0	0	0	0	0	0	0	0
w	0	1	2	3	4	5	6

```
for (int i=1; i<=n; i++) // para cada item
    for (int peso=0; peso<=w; peso++)     // para cada peso
        if (W[i] > peso)
            memo[i][peso] = memo[i-1][peso];
        else
            memo[i][peso] = max(memo[i-1][peso], W[i] + memo[i-1][peso-W[i]]);
```

- Suponha:
  - $W = 6$ e
  - $w_1 = 2$, $w_2 = 2$ e $w_3 = 3$.

i \ w	0	1	2	3	4	5	6
3	0	0	2	3			
2	0	0	2	2	4	4	4
1	0	0	2	2	2	2	2
0	0	0	0	0	0	0	0

```
for (int i=1; i<=n; i++) // para cada item
    for (int peso=0; peso<=w; peso++)    // para cada peso
        if (W[i] > peso)
            memo[i][peso] = memo[i-1][peso];
        else
            memo[i][peso] = max(memo[i-1][peso], W[i] + memo[i-1][peso-W[i]]);
```

- Suponha:
  - $W = 6$ e
  - $w_1 = 2$, $w_2 = 2$ e $w_3 = 3$.

i \ *w*	0	1	2	3	4	5	6
3	0	0	2	3	4		
2	0	0	2	2	4	4	4
1	0	0	2	2	2	2	2
0	0	0	0	0	0	0	0

```
for (int i=1; i<=n; i++) // para cada item
    for (int peso=0; peso<=w; peso++)    // para cada peso
        if (W[i] > peso)
            memo[i][peso] = memo[i-1][peso];
        else
            memo[i][peso] = max(memo[i-1][peso], W[i] + memo[i-1][peso-W[i]]);
```

- Suponha:
  - $W = 6$ e
  - $w_1 = 2$, $w_2 = 2$ e $w_3 = 3$.

i \ w	0	1	2	3	4	5	6
3	0	0	2	3	4	5	5
2	0	0	2	2	4	4	4
1	0	0	2	2	2	2	2
0	0	0	0	0	0	0	0

Exemplo 2 - WSSP

- O algoritmo calcula corretamente a resposta para o problema WSSP em tempo $O(nW)$
  - O algoritmo é chamado de pseudo-polinomial

- Dada a tabela M, pode-se encontrar o conjunto S em tempo $O(n)$

Uva 11450

One of our best friends is getting married and we all are nervous because he is the first of us who is doing something similar. In fact, we have never assisted to a wedding, so we have no clothes or accessories, and to solve the problem we are going to a famous department store of our city to buy all we need: a shirt, a belt, some shoes, a tie, etcetera.

Patricia Smith and **José Antonio Sánchez**
request the pleasure of
the company of
Mr and Mrs James and Sarah Student
at their wedding,
at St Mary's Church, Espinardo
on Saturday, May 17th, 2008 at 9 o'clock
and afterwards at
Cantina Hall, CSU.
RSVP
Universitary Campus, Espinardo, WT8 4EG

We are offered different models for each class of garment (for example, three shirts, two belts, four shoes, ...). We have to buy one model of each class of garment, and just one.

As our budget is limited, we cannot spend more money than it, but we want to spend the maximum possible. It's possible that we cannot buy one model of each class of garment due to the short amount of money we have.

Uva 11450

Input

The first line of the input contains an integer, N, indicating the number of test cases. For each test case, some lines appear, the first one contains two integers, M and C, separated by blanks ($1 \leq M \leq 200$, and $1 \leq C \leq 20$), where M is the available amount of money and C is the number of garments you have to buy. Following this line, there are C lines, each one with some integers separated by blanks; in each of these lines the first integer, K ($1 \leq K \leq 20$), indicates the number of different models for each garment and it is followed by K integers indicating the price of each model of that garment.

Output

For each test case, the output should consist of one integer indicating the maximum amount of money necessary to buy one element of each garment without exceeding the initial amount of money. If there is no solution, you must print '`no solution`'.

Uva 11450

Sample Input

```
3
100 4
3 8 6 4
2 5 10
4 1 3 3 7
4 50 14 23 8
20 3
3 4 6 8
2 5 10
4 1 3 5 5
5 3
3 6 4 8
2 10 6
4 7 3 1 7
```

Sample Output

```
75
19
no solution
```

Uva 11450

- Solução gulosa, funciona?
  - Para cada tipo roupa, selecione aquele que ainda cabe no orçamento restante (ou seja, pegue o maior valor e subtraia do orçamento total)..
- M = 12 (ou 20??) e C = 3
  - G = 0 → 6, 4, 8
  - G = 1 → 5, 10
  - G = 2 → 1,5, 3, 5
- ????

Uva 11450

- Solução “Divide and Conquer??
  - Tem sentido??
- Por que não serve
  - O que vc tem a dizer sobre os subproblemas?
  - São independentes??????

Uva 11450

- Busca Exaustiva???
  - Usar backtracking recursivo..
  - Função pair(money, g)
    - O dinheiro que temos no momento e o item de compra
  - Money = W e g = 0
    - Selecione modelo I de g e subtraia o dinheiro
    - Chame recursivamente para g=1
    -
  - Muito lento!!! vai dar Time limit Exceeded !!
  - (veja exemplo WISP por backtracking!)

Uva 11450

- De qq forma, a recorrência para o problema de busca seria
  - Se money < 0 → shop(money, g) = -infinito
  - Se compramos um modelo para o último item (g=C), então
    - shop(money,g) = M – money;
  - Para qq outro modelo em [1..K] de qq outro item g
    - shop(money, g) = max(shop(money – price[g,model], g+1);

Uva 11450

- A solução em PD para este problema pode ser derivada da solução de busca, mas sem este nro excessivo de comparações
  - Usar uma memo table (memoization !)
    - Memo[210][25]... Na verdade teremos 201 x 20 possíveis estados. Inicializar com -1
  - O código é dado por...
    - atenção spoiler! seja corajoso(a) e NÃO OLHE os próximos 2 slides…. tente fazer o seu código antes, baseado no que foi dito até agora..

```
int shop(int money, int g){
    if (money < 0)
        return -100000000; // nao temos dinheiro para este item....

    if (g == C) // ja compramos tudo,,, tchau... retornando o dinheiro g
        return (M-money);

    if (memo[money][g] != -1)
        return memo[money][g];

    int ans = -1;
    for (int model=1; model<=price[g][0]; model++)    // tente todos os
        ans = MAX(ans, shop(money-price[g][model],g+1));
    memo[money][g] = ans;
    return memo[money][g];
}
```

```c
int main(){
    int i,j,TC, score;

    scanf("%d", &TC);

    while(TC--){
        scanf("%d %d", &M, &C);
        for(i=0; i<C; i++){
            scanf("%d", &price[i][0]); // a qtd de modelos eh
            for(j=1; j<=price[i][0]; j++)
                scanf("%d", &price[i][j]);
        }
        memset(memo, -1, sizeof(memo));   // inicia memoizatio
        score = shop(M, 0);       // inicia a Prog Dinamica TOP
        if (score < 0)
            printf("no solution\n");
        else printf("%d\n", score);
    }
    return 0;
}
```

Uva 11450

- A versão apresentada nos slides anteriores é equivalente à versão WISP que combina memoização com recursão
- Não é de fato uma versão DP iterativa (sem recursão)

	0	1	2
0	0	0	0
1	0	0	0
2	0	0	0
3	0	0	0
4	0	0	0
5	0	0	0
6	0	0	0
7	0	0	0
8	0	0	0
9	0	0	0
10	0	0	0
11	0	0	0
12	1	0	0
13	0	0	0
14	1	0	0
15	0	0	0
16	1	0	0
17	0	0	0
18	0	0	0
19	0	0	0
20	0	0	0

	0	1	2
0	0	0	0
1	0	0	0
2	0	1	0
3	0	0	0
4	0	1	0
5	0	0	0
6	0	1	0
7	0	1	0
8	0	0	0
9	0	1	0
10	0	0	0
11	0	1	0
12	1	0	0
13	0	0	0
14	1	0	0
15	0	0	0
16	1	0	0
17	0	0	0
18	0	0	0
19	0	0	0
20	0	0	0

	0	1	2
0	0	0	0
1	0	0	1
2	0	1	1
3	0	0	1
4	0	1	1
5	0	0	1
6	0	1	1
7	0	1	0
8	0	0	1
9	0	1	0
10	0	0	1
11	0	1	0
12	1	0	0
13	0	0	0
14	1	0	0
15	0	0	0
16	1	0	0
17	0	0	0
18	0	0	0
19	0	0	0
20	0	0	0

Figure 3.8: UVa 11450 - Bottom-Up DP Solution

O próximo slide contém o código com a versão PD iterativa… Tente montar o seu código antes de lê-lo !!!!

```
int main() {
  int i, j, k, TC, M, C;
  int price[25][25];                          // price[g (<= 20)][model (<= 20)]
  bool reachable[25][210];    // reachable table[g (<= 20)][money (<= 200)]
  scanf("%d", &TC);
  while (TC--) {
    scanf("%d %d", &M, &C);
    for (i = 0; i < C; i++) {
      scanf("%d", &price[i][0]);                   // we store K in price[i][0]
      for (j = 1; j <= price[i][0]; j++) scanf("%d", &price[i][j]);
    }

    memset(reachable, false, sizeof reachable);           // clear everything
    for (i = 1; i <= price[0][0]; i++)           // initial values (base cases)
      if (M - price[0][i] >= 0)        // to prevent array index out of bound
        reachable[0][M - price[0][i]] = true;  // using first garment g = 0

    for (i = 1; i < C; i++)                       // for each remaining garment
      for (j = 0; j < M; j++) if (reachable[i - 1][j]) // a reachable state
        for (k = 1; k <= price[i][0]; k++) if (j - price[i][k] >= 0)
          reachable[i][j - price[i][k]] = true;   // also a reachable state

    for (j = 0; j <= M && !reachable[C - 1][j]; j++); // the answer in here

    if (j == M + 1) printf("no solution\n");           // last row has on bit
    else            printf("%d\n", M - j);
} } // return 0;
```

Top-Down	**Bottom-Up**
Pro: 1. It is a natural transformation from normal Complete Search recursion 2. Compute sub-problems only when necessary (sometimes this is faster)	Pro: 1. Faster if many sub-problems are revisited as there is no overhead from recursive calls 2. Can save memory space with DP 'on-the-fly' technique (see comment in code above)
Cons: 1. Slower if many sub-problems are revisited due to recursive calls overhead (usually this is not penalized in programming contests) 2. If there are M states, it can use up to $O(M)$ table size which can lead to Memory Limit Exceeded (MLE) for some hard problems	Cons: 1. For some programmers who are inclined with recursion, this may be not intuitive 2. If there are M states, bottom-up DP visits and fills the value of *all* these M states

Table 3.1: DP Decision Table