Universidade Federal de Uberlândia - UFU
Programa de Pós Graduação em Ciência da Computação
Inteligência Artificial

INTERVALOS PONDERADOS

CIBELE MARA FONSECA

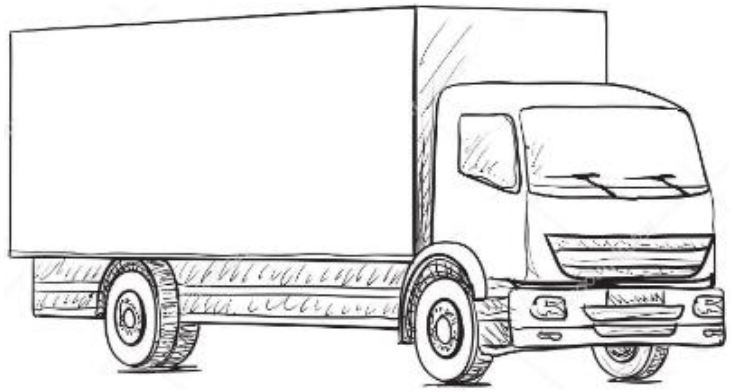
MALENA ALVES RUFINO

DISCIPLINA: ANÁLISE DE ALGORITMOS

PROFESSORA DRA.: MÁRCIA APARECIDA FERNANDES

Contextualizando...

- Suponhamos que temos apenas um caminhão para realizar n viagens;
- O caminhão pode realizar apenas uma viagem por vez, logo só pode começar uma nova viagem ao terminar a anterior;
- Cada viagem tem uma data de início, uma data de término e uma prioridade;
- O objetivo é fazer as viagens cujo a soma de prioridades seja a maior possível durante o período de tempo disponível.



Cidade	Início	Término	Prioridade
Uberlândia	01	16	9
Belo Horizonte	11	23	10
Goiânia	19	31	2

Definição

Dado um conjunto S de n atividades onde para cada uma são definidos tempo de início **s**i, tempo de término **f**i e peso **v**i. Deseja-se encontrar o subconjunto cujo a soma do pesos de tarefas mutuamente compatíveis entre si seja a maior possível.

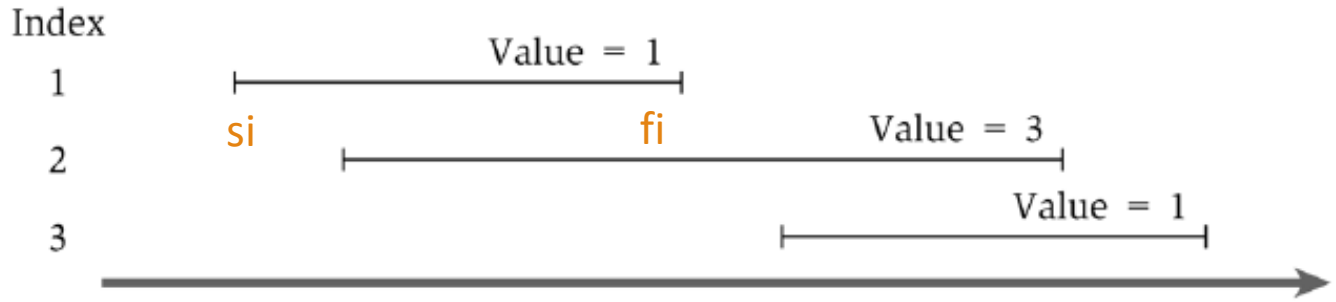


Figure 6.1 A simple instance of weighted interval scheduling.

Atividade compatível: {1,3} com peso 2;
{2} com peso 3, é a solução ótima.

Definição

- Restrição: Intervalos ordenados por tempo de término não decrescente: f_1 $f_2 \leq \ldots \leq f_n$;
- i< j se $f_i < f_j$;
- Intervalo i é compatível a j, se $f_i \leq s_j$;
- p(j)=i para um intervalo j, seja i o maior índice i<j tal que os intervalos i e j são compatíveis;
- p(j)=0 se não existe nenhum intervalo i, tal que i<j e i é compatível a j;
- Para computar p(j) utilizamos uma busca binária, que procura identificar um $f_i \leq s_j$ e atribui p(j)=i.

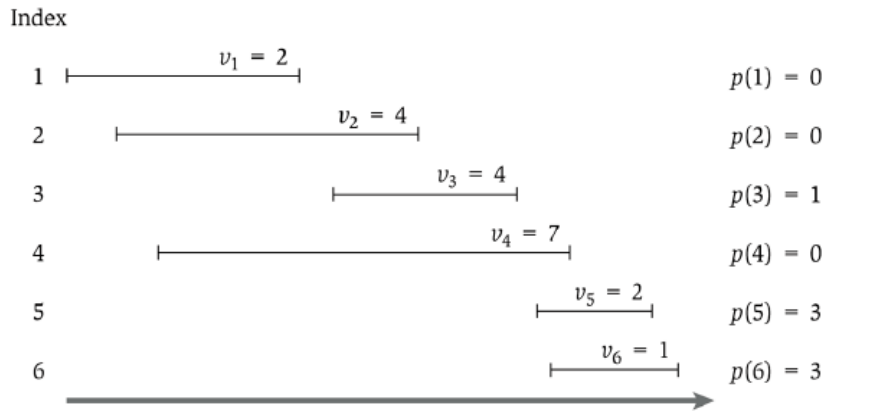


Figure 6.2 An instance of weighted interval scheduling with the functions $p(j)$ defined for each interval j.

Um exemplo de agendamento de intervalos ponderados com a função p(j) definida para cada intervalo j.

Definição

- Considere uma solução ótima O (não temos ideia do que é);
- Intervalo n petence a O ou não;
- Se n ∈ O, então nenhum intervalo entre p(n) e n pertence a O;
- Se n ∈ O, então O deve conter solução ótima de {1,...,p(n)};
- Se n ∉ O, então O é igual a solução ótima do subproblema {1,...,n-1};

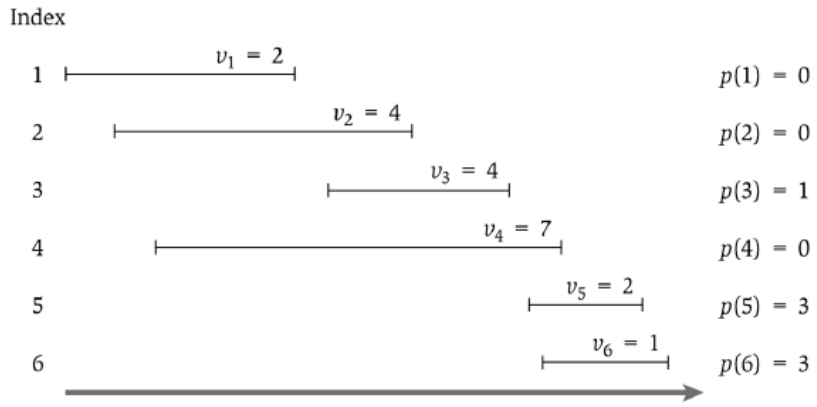


Figure 6.2 An instance of weighted interval scheduling with the functions $p(j)$ defined for each interval j.

Definição

- Definimos OPT(0)=0, considerando que essa é a solução ótima para conjuntos vazios;
- A solução que procuramos é On, com o valor de OPT(n);
- Para a solução ótima Oj em que j=n (Nos leva ao princípio do começo que n pertence ou não a O, ou seja, a escolha binária).

Escolha Binária

OPT (j) = valor da solução ótima para o problema que consiste em intervalos 1, 2, ..., j.

Caso 1: j ∈ OPT(j)

Não pode usar intervalos incompatíveis a j e deve incluir solução ótima para problema consistindo em intervalos compatíveis restantes 1, 2, ..., p (j).

Caso 2: j ∉ OPT(j)

Deve incluir a solução ótima para o problema de intervalos compatíveis 1, 2, ..., j-1.

Expressão Recursiva

$$OPT(j) = \begin{cases} 0 & \text{if } j = 0 \\ \max\{v_j + OPT(p(j)), OPT(j-1)\} & \text{otherwise} \end{cases}$$

Escolha Binária

Como decidir se o intervalo j pertece a Oj?

Ele pertence à solução ótima, se somento se o Caso 1 for pelo menos tão bom quanto o caso 2.

Em outras palavras:

$$vj + OPT(p(j)) \geq OPT(J-1)$$

Expressão Recursiva

$$OPT(j) = \begin{cases} 0 & \text{if } j = 0 \\ \max\{v_j + OPT(p(j)), OPT(j-1)\} & \text{otherwise} \end{cases}$$

Método Recursivo

```
Compute_Opt(j) {
  If j = 0
    Returna 0
  Else
    Retorna max(vj+Compute_Opt(p(j)) , Compute_Opt(j-1))
}
```

Árvore de Recursão

A árvore dos subproblemas cresce muito rapidamente!

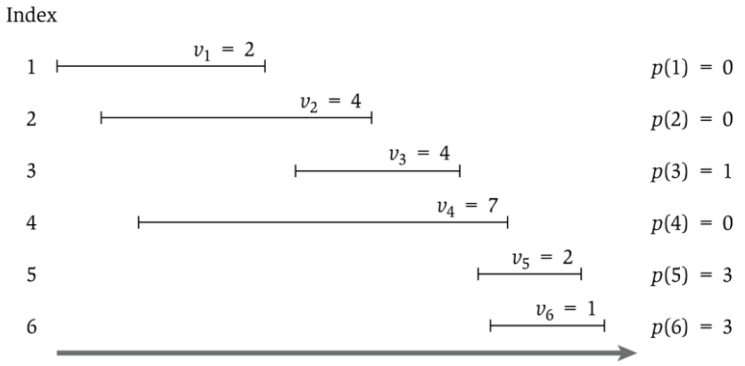


Figure 6.2 An instance of weighted interval scheduling with the functions $p(j)$ defined for each interval j.

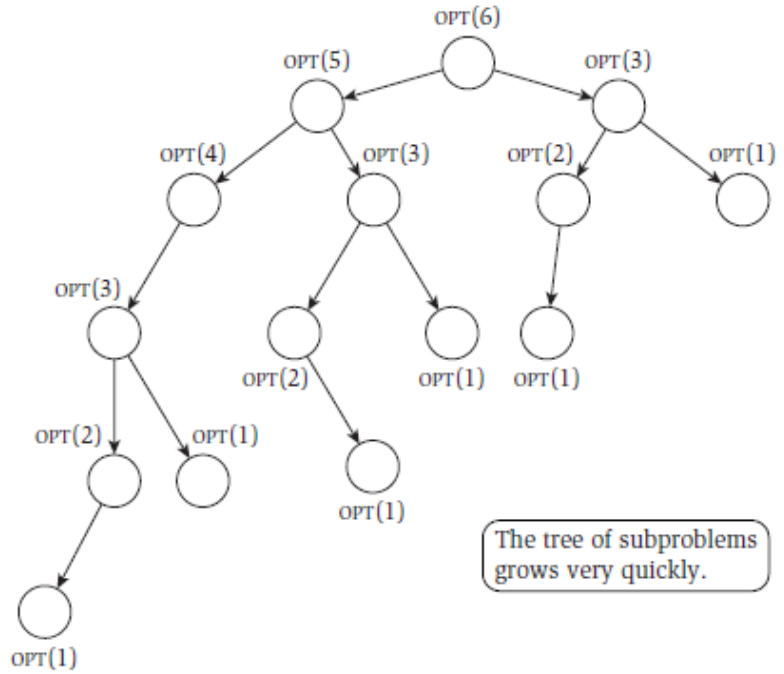


Figure 6.3 The tree of subproblems called by `Compute-Opt` on the problem instance of Figure 6.2.

Método Recursivo – Prova

- Compute_Opt(j) calcula a solução corretamente cada OPT(j) para j = 1,2,...,n
- Pela definição OPT(0)=0. Agora tome algum j>0 e supondo por indução que Compute_Opt(i) também computa OPT(i) corretamente para todo i<j. Pela hipótese de indução sabemos que Compute_Opt(p(j))=OPT(p(j)) e Compute_Opt(j-1)=OPT(j-1).
- Porém se implementarmos o algoritmo dessa forma, ele pode levar tempo exponencial no pior caso.

Memorização da Recursão

- Existem chamadas recursivas redundantes no algoritmo anterior.
- O algoritmo recursivo deve resolver n+1 problemas diferentes: Compute_Opt(0), Compute_Opt(1),...,Compute_Opt(n).
- O algoritmo anterior faz várias vezes uma mesma chamada. Por exemplo, chama várias vezes Compute_Opt(2) e resolve recursivamente.
- Para eliminar a redundância podemos armazenar o valor de Compute_Opt na primeira vez que o executamos, e usarmos esse valor no lugar de futuras chamadas recursivas.
- Para isso faremos o uso de um array M[0..n] que começará com valor vazio mas manterá o valor de Compute_Opt(j) assim que computado pela primeira vez.

Método Recursivo com Memorização

Input: n, s1,…,sn , f1,…,fn , v1,…,vn

Ordenar por f f1 ≤ f2 ≤ … ≤ fn.

Compute p(1), p(2), …, p(n)

```
for j = 1 to n
    M[j] = Ø
M[0] = 0

M-Compute-Opt {
  if M[j] == Ø
    M[j] = max(vj + M-Compute-Opt(p(j)), M-Compute-Opt(j-1))
  return M[j]
}
```

O(n)

Método Recursivo com Memorização – Prova

- M-Compute-Opt (j): cada chamada leva o tempo O (1) e também:

i. retorna um valor existente M [j] ;

ii. preenche uma nova entrada M [j] e faz duas chamadas recursivas.

- O caso (ii) ocorre no máximo n vezes → no máximo 2n chamadas recursivas em geral.
- Medida de progresso: avança conforme encontramos uma solução para subestrutura otima;
- Inicialmente esta medida é 0, cada vez que o metodo chama a recorrência emitindo as duas chamadas recursivas o M_Compute_Opt preenche uma nova posição de M aumentando o progresso em 1;
- Como M possui n+1 posições pode-se haver no máximo O(n) chamadas, portanto o tempo de execução total do M-Compute-Opt é O(n).

Conjunto ótimo de intervalos

- Ao calcular a solução ótima, também desejamos obter o conjunto ótimo de intervalos.
- Sabemos que j pertence a uma solução ótima para o conjunto de intervalos {1,. . . , j} se e somente se

$$v_j + OPT(p(j)) \geq OPT(j - 1).$$

- Usando esta observação, obtemos o seguinte procedimento, que “traça de volta” através do array M para encontrar o conjunto de intervalos em uma solução ótima.

Find–Solution

```
Run M-Compute-Opt(n)
Run Find-Solution(n)
Find-Solution(j) {
    if (j = 0)
        output Ø
    else if (vj + M[p(j)] ≥ M[j-1])
        print j
        Find-Solution(p(j))
    Else
        Find-Solution(j-1)
}
```

O(n)

Exemplo

M[1] = max (2+M[0], M[0]) = max (2+0, 0) = 2
M[2] = max (4+M[0], M[1]) = max (4+0, 2) = 4
M[3] = max (4+M[1], M[2]) = max (4+2, 4) = 6
M[4] = max (7+M[0], M[3]) = max (7+0, 6) = 7
M[5] = max (2+M[3], M[4]) = max (2+6, 7) = 8
M[6] = max (1+M[3], M[5]) = max (1+6, 8) = 8

$M =$

0	1	2	3	4	5	6
0	2	4	6	7	8	8

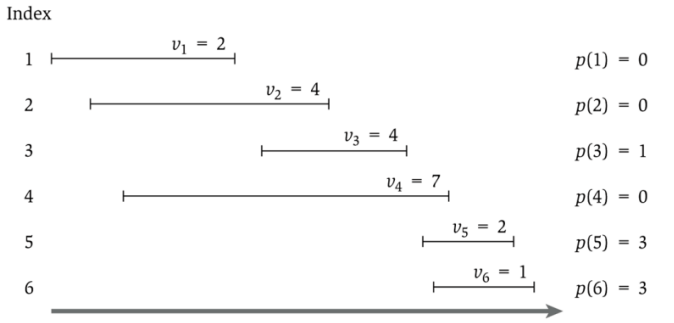


Figure 6.2 An instance of weighted interval scheduling with the functions $p(j)$ defined for each interval j.

M[6]=M[5]: 6 não pertence ao OPT
M[5]=v5+M[3]: OPT contém 5 e uma solução ótima para o problema de {1,2,3}
M[3]=v3+M[1]: OPT contém 5,3 e uma solução ótima para o problema de {1}
M[1]=v1+M[0]: OPT contém 5,3 e 1 (e uma solução ótima para o problema vazio)

Solução: {5,3,1}
Valores: 2+4+2=8

Find-Solution – Prova

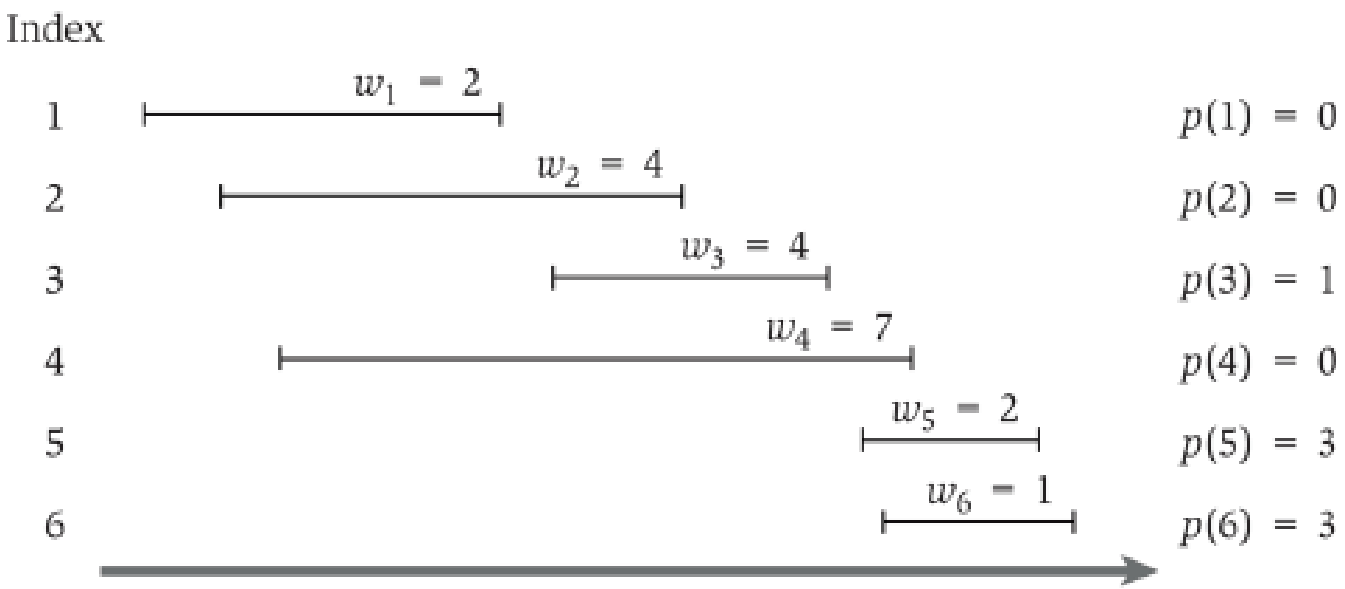
Como o Find-Solution chama-se recursivamente somente em valores estritamente menores, faz um total de O (n) chamadas recursivas; e uma vez que gasta tempo constante por chamada, temos: Dado o array M dos valores ótimos dos sub-problemas, o Find-Solution retorna uma solução ótima no tempo O (n).

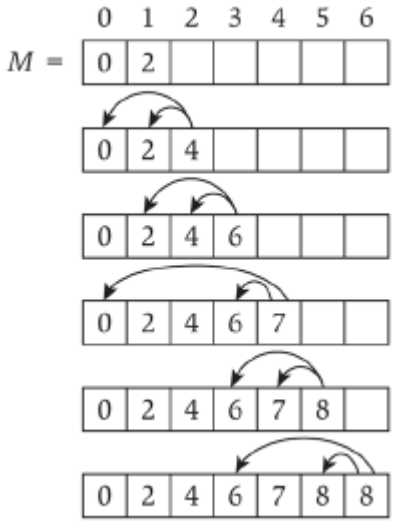
Iterações sobre os subproblemas

- Também pode-se solucionar o problema iterando sobre os subproblemas, em vez de computar soluções de forma recursiva.
- A chave para o algoritmo é o array M[n], mostrado anteriormente.
- M[j] contém solução ótima para um problema {1,...,j}.
- M[n] contém a solução ótima do problema como um todo.
- Logo é possível calcular a solução do problema usando um algoritmo iterativo, em vez de recursão com memorização.
- Abordagem bottom-up.
- Começa-se com M[0]=0 e incrementa o j, cada vez que temos que determinar o valor de M[j] a resposta é dada por:

max (vj + OPT(p(j)), OPT(j-1))

Método Iterativo





M[1] = max (2+M[0], M[0]) = max (2+0, 0) = 2
M[2] = max (4+M[0], M[1]) = max (4+0, 2) = 4
M[3] = max (4+M[1], M[2]) = max (4+2, 4) = 6
M[4] = max (7+M[0], M[3]) = max (7+0, 6) = 7
M[5] = max (2+M[3], M[4]) = max (2+6, 7) = 8
M[6] = max (1+M[3], M[5]) = max (1+6, 8) = 8

Método Iterativo

Input: n, s1,…,sn , f1,…,fn , v1,…,vn

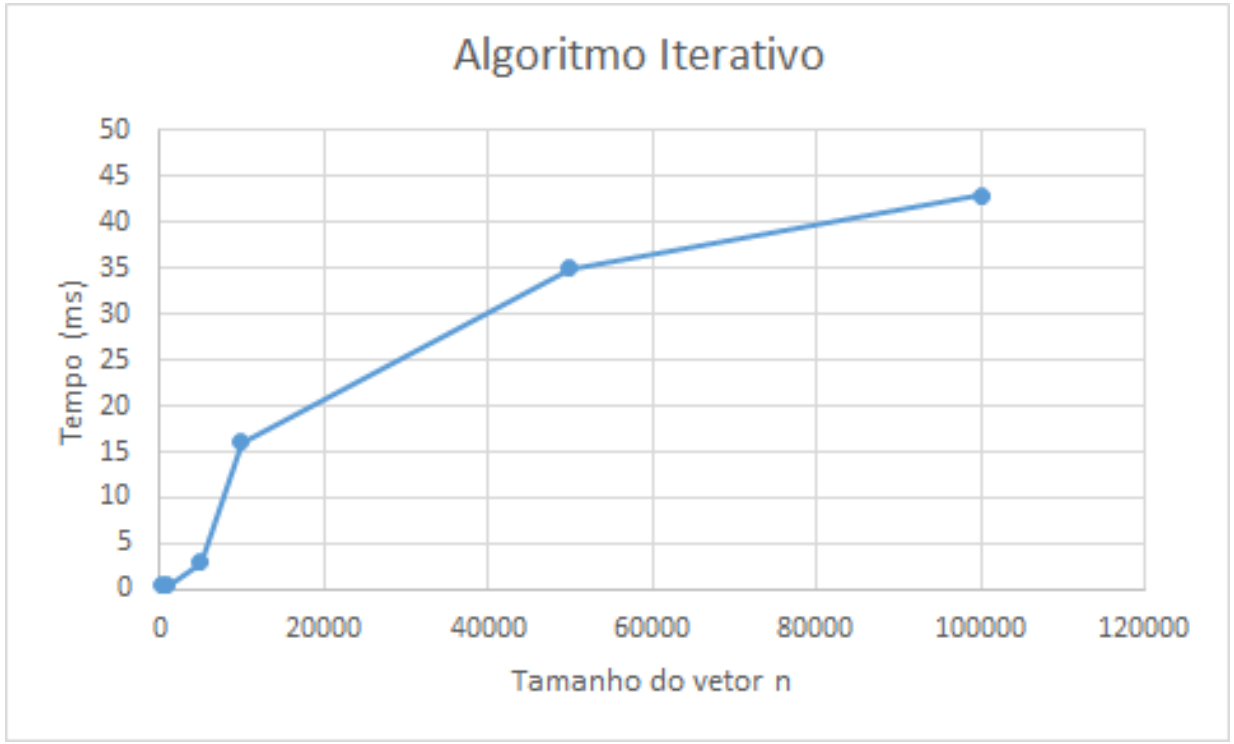
Ordenar por f f1 ≤ f2 ≤ … ≤ fn.

Compute p(1), p(2), …, p(n)

```
Iterative-Compute-Opt {
    M[0] = 0
   for j = 1 to n
     M[j] = max(vj + M[p(j)], M[j-1])
}
```

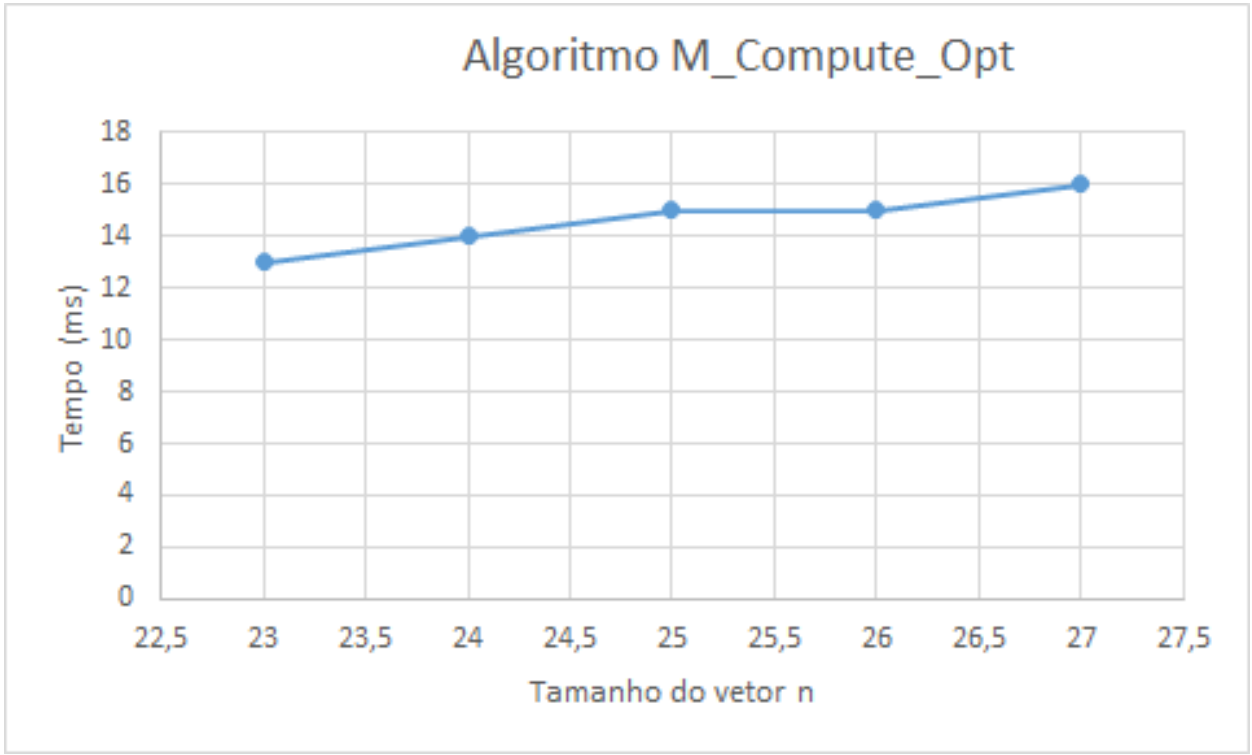
O(n)

Curva de tempo



N Entradas	Tempo (ms)
100	0,5
1.000	0,5
5.000	3
10.000	16
50.000	35
100.000	43

Curva de tempo



N entradas	Tempo (ms)
22	0
23	13
24	14
25	15
26	15
27	16
28	15
29	Para!

CERTINHO