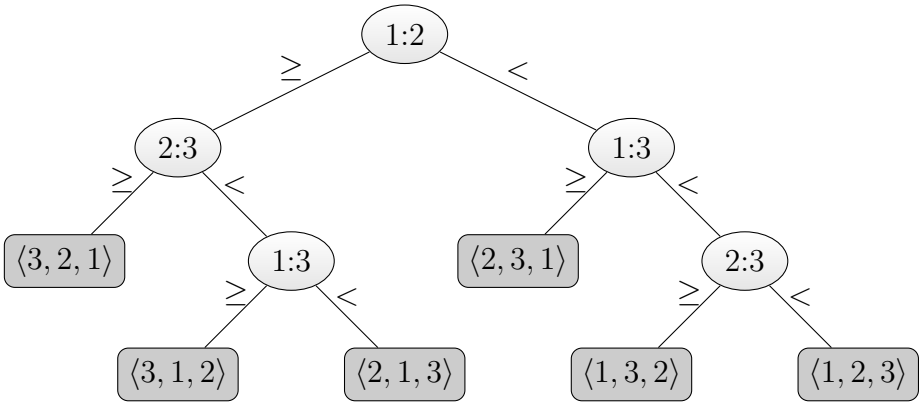
MAC 5711 - Análise de Algoritmos

Rodrigo Augusto Dias Faria
Departamento de Ciência da Computação - IME/USP

19 de outubro de 2015

1. Desenhe a árvore de decisão para o SELECTIONSORT aplicado a $A[1..3]$ com todos os elementos distintos.



2. **(CLRS 8.1-1)** Qual a menor profundidade (= menor nível) que uma folha pode ter em uma árvore de decisão que descreve um algoritmo de ordenação baseado em comparações?

Resposta: A menor profundidade da árvore de decisão é, coincidentemente, a cota inferior das alturas de todas as árvores de decisão nas quais aparecem uma folha (uma das $n!$ permutações da entrada).

Também podemos dizer que é o melhor caso em tempo de execução de qualquer algoritmo de ordenação baseado em comparações.

Logo, é o caso em que apenas $n-1$ comparações são realizadas para ordenar o vetor que ocorre, por exemplo, quando o vetor já está ordenado.

3. Mostre que $lg(n!) \geq \frac{n}{4} lgn$ para $n \geq 4$ sem usar a fórmula de Stirling.

Resposta: $n!$ pode ser escrito como um produtório:

$$\prod_{k=1}^{n} k$$

Também podemos escrever o produtório como um somatório, pela seguinte identidade (CLRS pp 1061 - 2ed):

$$lg(\prod_{k=1}^{n} k) = \sum_{k=1}^{n} lg(k)$$

Logo:

$$\begin{aligned} lg(n!) = \sum_{k=1}^{n} lg(k) &= \sum_{k=1}^{\lfloor \frac{n}{4} \rfloor} lg(k) + \sum_{k=\lfloor \frac{n}{4} \rfloor + 1}^{2\lfloor \frac{n}{4} \rfloor} lg(k) + \sum_{k=2\lfloor \frac{n}{4} \rfloor + 1}^{3\lfloor \frac{n}{4} \rfloor} lg(k) + \sum_{k=3\lfloor \frac{n}{4} \rfloor + 1}^{n} lg(k) \\ &\geq \sum_{k=\lfloor \frac{n}{4} \rfloor + 1}^{n} lg(k) \qquad \text{(descartando 1/4 da som} \\ &\geq \sum_{k=\frac{n}{4}}^{n} lg(\frac{n}{4}) \\ &\geq \frac{n}{4} lg \frac{n}{4} = \frac{n}{4}(lgn - lg2^2) = \frac{n}{4}(lgn - 2) = \frac{n}{4}lgn - \frac{1}{2}n \end{aligned}$$

$$\therefore lg(n!) \geq \frac{n}{4}lgn$$

4. (**CLRS 8.1-3**) Mostre que não há algoritmo de ordenação baseado em comparações cujo consumo de tempo é linear para pelo menos metade de $n!$ permutações de 1 a n. O que acontece se trocarmos "metade"por uma fração de $1/n$? O que acontece se trocarmos "metade"por uma fração de $1/2^n$?

Resposta: Assim como visto em aula uma árvore de decisão pode ser utilizada para representar o número de comparações executadas por um algoritmo. A árvore de decisão é uma árvore binária onde cada nó não folha é uma comparação e cada folha é uma permutação de entrada, o caminho da raiz da árvore até uma de suas folhas mostra a quantidade de comparações executadas para aquela permutação, portanto a distância da raiz para a folha mais distânte (altura da árvore) reflete o tempo gasto pelo pior caso do algoritmo.

Vamos verificar qual a altura da árvore de decisão para saber qual o tempo gasto no pior caso para pelo menos metade das possíveis permutações $n!/2$. Seja l o número de folhas do algoritmo e h a altura da árvore, como a árvore de decisão é uma árvore binária sabemos que ela terá no máximo 2^h folhas. Portanto temos a seguinte relação:

$$\frac{n!}{2} \leq l \leq 2^h$$

Calculando o logaritmo dos dois lados e sabendo que logaritmo é uma função monotonica-

mente crescente temos que:

$$\begin{aligned} \lg 2^h &\geq \lg \frac{n!}{2} \\ h &\geq \lg n! - \lg 2 \\ h &\geq \lg n! - 1 \end{aligned}$$

Como visto em aula $\lg n! \geq \frac{1}{2} n \lg n$, dessa inequação chegamos que $h \geq \lg n! \geq \frac{1}{2} n \lg n$. Disso podemos concluir que $h \geq \frac{1}{2} n \lg n - 1$, ou seja, mesmo utilizando apenas a metade das permutações assintoticamente o algoritmo ótimo gasta $\Omega(n \lg n)$ no pior caso.

Trocando a "metade"por uma fração $1/n$, podemos utilizar o mesmo raciocínio, definindo l como a quantidade de folhas da árvore de decisão e h a altura da árvore, teremos a seguinte inequação:

$$\frac{n!}{n} \leq l \leq 2^h$$

Calculando o logaritmo dos dois lados e sabendo que logaritmo é uma função monotonicamente crescente temos que:

$$\begin{aligned} \lg 2^h &\geq \lg \frac{n!}{n} \\ h &\geq \lg n! - \lg n \\ h &\geq n \lg n - \lg n \end{aligned}$$

Note $n \lg n - \lg n = \Theta(n \lg n)$, portanto podemos concluir que para uma fração de $1/n$ das permutações também mantem-se gasto de tempo de $\Omega(n \lg n)$ no pior caso.

Trocando a "metade"pela fração de $1/2^n$, podemos utilizar o mesmo raciocínio para calcular o pior caso de qualquer algoritmo de ordenação baseada em comparações. Seja l o número de folhas e h a altura da árvore teremos a seguinte relação:

$$\frac{n!}{2^n} \leq l \leq 2^h$$

Calculando o logaritmo dos dois lados e sabendo que logaritmo é uma função monotonicamente crescente temos que:

$$\begin{aligned} \lg 2^h &\geq \lg(\frac{n!}{2^n}) \\ h &\geq \lg(n!) - \lg 2^n \\ h &\geq n \lg n - n \end{aligned}$$

Note que $n \lg n - n = \Theta(n \lg n)$, portanto da relação acima temos que mesmo para a fração de $1/2^n$ de permutações os algoritmos de ordenação que utilização comparações consomem tempo $\Omega(n \lg n)$ no pior caso.

Solução alternativa O teorema 8.1 diz que qualquer algoritmo de ordenação por comparação exige $\Omega(nlgn)$ no pior caso.

Estabelecendo uma cota inferior, temos:

$$lg(n!) = lg\Big(\prod_{k=1}^{n} k\Big) = \sum_{k=1}^{n} lg(k) = \sum_{k=1}^{\lfloor \frac{n}{2} \rfloor} lg(k) + \sum_{k=\lfloor \frac{n}{2} \rfloor + 1}^{n} lg(k)$$

$$\geq \sum_{k=\lfloor \frac{n}{2} \rfloor + 1}^{n} lg(k) \qquad \text{(descartando 1/2 da s}$$

$$\geq \sum_{k=\frac{n}{2}}^{n} lg\left(\frac{n}{2}\right)$$

$$\geq \frac{n}{2} lg\left(\frac{n}{2}\right) = \frac{n}{2}(lgn - 1) = \frac{1}{2}nlgn - \frac{1}{2}n = \Omega(nlgn)$$

Dizer que o tempo de execução do algoritmo é linear para pelo menos metade das $n!$ permutações, deve valer para $lg\left(\frac{n!}{2}\right)$, o que é absurdo, vejamos:

$$lg\left(\frac{n!}{2}\right) = lgn! - lg2 = lgn! - 1 \geq \frac{1}{2}nlgn - \frac{1}{2}n - 1$$

O mesmo ocorre com $\left(\frac{n!}{n}\right)$ e $\left(\frac{n!}{2^n}\right)$:

$$lg\left(\frac{n!}{n}\right) = lgn! - lgn \geq \frac{1}{2}nlgn - \frac{1}{2}n - lgn$$

$$lg\left(\frac{n!}{2^n}\right) = lgn! - lg2^n = lgn! - n \geq \frac{1}{2}nlgn - \frac{1}{2}n - n = \frac{1}{2}nlgn - \frac{3}{2}n$$

4.1 (CLRS 8.1-4) Suppose that you are given a sequence of n elements to sort. The input sequence consists of n/k subsequences, each containing k elements. The elements in a given subsequence are all smaller than the elements in the succeeding subsequence and larger than the elements in the preceding subsequence. Thus, all that is needed to sort the whole sequence of length n is to sort the k elements in each of the n/k subsequences. Show an $\Omega(nlgk)$ lower bound on the number of comparisons needed to solve this variant of the sorting problem. (*Hint:* It is not rigorous to simply combine the lower bounds for the individual subsequences.)

Resposta: A entrada consiste de n/k subsequências, cada uma contendo k elementos, sendo que os elementos da subsequência, digamos s_i são menores que os elementos da subsequência s_{i+1} e maiores que os elementos da subsequência anterior s_{i-1}.

O que precisamos fazer para ordenar os n elementos é ordenar os elementos de cada subsequência s.

Se pensarmos apenas nos k elementos de uma subsequência, podemos estabelecer uma cota inferior para ordenação:

$$lg(k!) = lg(\prod_{i=1}^{k} i) = \sum_{i=1}^{k} lg(i) = \sum_{i=1}^{\lfloor \frac{k}{2} \rfloor} lg(i) + \sum_{i=\lfloor \frac{k}{2} \rfloor + 1}^{k} lg(i)$$

$$\geq \sum_{i=\lfloor \frac{k}{2} \rfloor + 1}^{k} lg(i) \qquad \text{(descartando 1/2 da so}$$

$$\geq \sum_{i=\frac{k}{2}}^{k} lg\left(\frac{k}{2}\right)$$

$$\geq \frac{k}{2} lg\left(\frac{k}{2}\right) = \frac{k}{2}(lgk - 1) = \frac{1}{2}klgk - \frac{1}{2}k = \Omega(klgk)$$

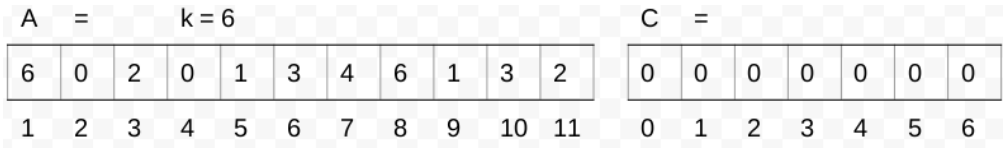
Como nós temos n/k subsequências, basta combinarmos a cota inferior de cada subsequência com k elementos:

$$\frac{n}{k}\left(\frac{1}{2}klgk - \frac{1}{2}k\right) = \frac{n}{k}\left(\frac{1}{2}klgk\right) - \frac{1}{2}n = \frac{1}{2}nlgk - \frac{1}{2}n = \Omega(nlgk)$$

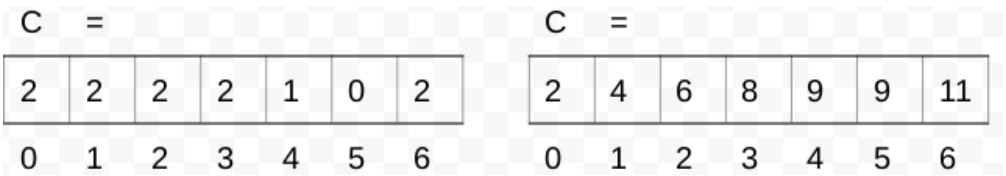
5. (**CLRS 8.2-1**) Simule a execução do CountingSort usando como entrada o vetor:

$$A[1 \cdots 11] = \langle 6, 0, 2, 0, 1, 3, 4, 6, 1, 3, 2 \rangle$$

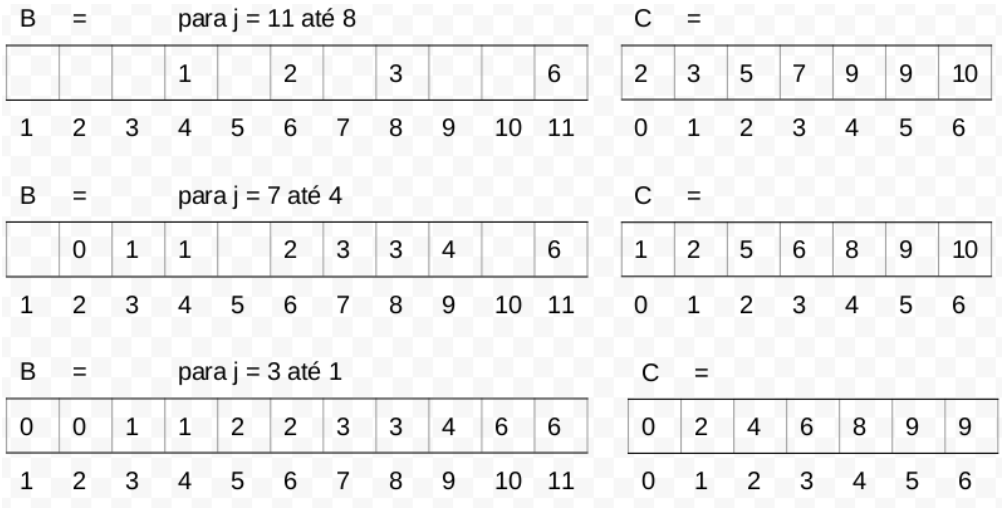
Resposta: A configuração inicial do algoritmo após o laço inicial será:



O vetor A nunca é alterado, o vetor C é representado abaixo após o final do segundo laço (esquerda) e após o final do terceiro laço (direita):



O último laço começa pelo com o índice $j = n$ e vai até 1, o vetor B começa vázio e vai sendo preenchido até o final do algoritmo. Abaixo segue os estados dos vetores C e B no fim das iterações $j = 8, j = 4$ e $j = 1$:
Então o algoritmo se encerra mantendo o vetor B ordenado.



6. (**CLRS 8.2-2**) Mostre que o COUNTING-SORT é estável.

Resposta: Vimos em sala que um algoritmo de ordenação é estável se sempre que, inicialmente, $A[i] = A[j]$ para $i < j$, a cópia $A[i]$ termina em uma posição menor do vetor que a cópia $A[j]$.

O COUNTING-SORT conta quantas vezes um inteiro de $1..k$ aparece em $A[1..n]$ e armazena essa informação no vetor C. A próxima iteração faz com que o vetor C tenha a contagem acumulada dos elementos em A. Logo, após essa iteração nós sabemos que $C[i] < C[i+1]$.

A última iteração ordena efetivamente A, copiando seus elementos de $n..1$ para um vetor auxiliar B. Como a contagem foi feita na ordem $1..n$ e os valores de C são usados como índices em B e eles são decrementados a cada vez que uma cópia é feita, a ordem relativa em que os elementos aparecem em A é preservada, mesmo no caso onde há repetições, garantindo, assim, a estabilidade do algoritmo.

7. **(CLRS 8.2-3)** Suponha que o **para** da linha 7 do COUNTINGSORT é substituído por

```
1  for i = 1 to n
```

Mostre que o COUNTINGSORT ainda funciona. O algoritmo resultante continua estável?

Resposta: O COUNTINGSORT continua ordenando o vetor de entrada, o laço substituído ficará:

```
1  for j = 1 to n
2        B[C[A[j]]] = A[i]
3        C[A[i]] = C[A[i]] − 1
```

Sabemos que o algoritmo não foi modificado antes da linha 7 (do algoritmo original), então o vetor C continua informando, qual a última posição que os elementos de A devem

ser inseridos, ou seja, $C[i]$ indica onde $A[i]$ deve ser inserido no vetor ordenado B, após inseri-lo ele $C[i]$ é decrementando e indicando que o proximo elemento de A que seja igual a $A[i]$ deve ser colocado em uma posição anterior assim como o algoritmo dado em aula faz, logo o algoritmo continua ordenado o vetor A. Mas com essa alteração o algoritmo perde informação satelite, ou seja, perde estabilidade, por que ao encontrar um elemento $A[i]$ o mesmo será inserido em $C[A[i]]$ que indica a última posição daquele elemento, ao encontrar $A[j] = A[i]$ para $j > i$, o algoritmo ira inseri-lo em $C[A[i]] - 1$, perdendo assim a estabilidade já que $A[i]$ estará em uma posição maior que $A[i]$ estará em uma posição maior que $A[j]$ para $A[i] = A[j]$ e $i < j$.

8. (**CLRS 8.2-4**) Descreva um algoritmo que, dados n inteiros no intervalo de 1 a k, preprocesse sua entrada e então responda em $O(1)$ qualquer consulta sobre quantos dos n inteiros dados caem em um intervalo $[a \cdots b]$. O preprocessamento efetuado pelo seu algoritmo deve consomir tempo $O(n + k)$.

Resposta: O algoritmo desenvolvido é baseado no algoritmo de *Counting Sort* descrito no livro do Cormen. O algoritmo possuí duas rotinas, uma de preprocessamento que devolve um vetor de contagem C e um de consulta que retorna a quantidade de elementos conforme o enunciado. O algoritmo de preprocessamento abaixo recebe um vetor A de n inteiros qualquer tal que seus elementos estão dentro do intervalo $[1 \cdots k]$ e realiza o preprocessamento devolvendo o vetor "C"do *counting sort* antes de ordena-lo no vetor B. Esse vetor possuí o índice do último valor no vetor ordenado de cada índice, por exemplo, $C[x] = y$, indica que o último elemento x está no índice y do vetor ordenado.

PrePrecessamento(A, k, n)

```
1  for i = 1 to k
2        C[i] = 0
3  for i = 1 to n
4        C[A[i]] = C[A[i]] + 1
5  for i = 2 to k
6        C[i] = C[i] + C[i − 1]
7  return C
```

O procedimento ContaNoIntervalo recebe o vetor gerado pelo PreProcessamento e os extremos do intervalo $[a \cdots b]$ e devolve a quantidade de elementos do vetor original A que estão dentro desse vetor.

ContaNoIntervalo(C, a, b)

```
1  if a ≤ 1
2        a' = 1
3  else a' = C[a − 1] + 1
4  return C[b] − a' + 1
```

Corretude

O procedimento de preprocessamento utiliza o *counting sort* disso sabemos que o vetor $C[i]$

contém a quantidade de elementos menores ou iguais a i do vetor original, isso implica que o valor de $C[i]$ é a posição do último elemento i (não o elemento da posição i) do vetor original no vetor ordenado, isso implica também que o valor de $C[i-1]+1$ é a posição do primeiro elemento i do vetor original A no vetor ordenado. Disso temos que $C[b] - C[a-1] + 1$ é quantidade de elementos com os valores entre $[a \cdots b]$ do vetor original. Note que o if do procedimento ContaNoIntervalo, apenas garante que $a - 1 \geq 1$ para que não seja acessado uma posição fora do vetor C, caso isso ocorra consideramos que a posição inicial é 1, note que isso é válido para quando existem valores 1 em A e quando não existem.

Desempenho

Veja a tabela de gastos em cada linha do procedimento PreProcessamento:

Linha	Tempo
1	$\Theta(k)$
2	$\Theta(k)$
3	$\Theta(n)$
4	$\Theta(n)$
5	$\Theta(k)$
6	$\Theta(k)$
7	$\Theta(1)$
Total	$\Theta(n+k)$

O procedimento gasta o tempo total $\Theta(n+k)$. O tempo gasto no procedimento ContaNoIntervalo está expresso na tabela abaixo:

Linha	Tempo
1	$\Theta(1)$
2	$\Theta(1)$
3	$\Theta(1)$
4	$\Theta(1)$
Total	$\Theta(1)$

Portanto o algoritmo gasta tempo $\Theta(n+k)$ para o preprocessamento e tempo $\Theta(1)$ para a chamada a contagem dos elementos conforme enunciando do exercício.

8.1 CLRS 8.3-1 Using figure 8.3 as a model, illustrate the operation of RADIX-SORT on the following list of English words: COW, DOG, SEA, RUG, ROW, MOB, BOX, TAB, BAR, EAR, TAR, DIG, BIG, TEA, NOW, FOX.

Basta ordenarmos do dígitos menos significativo ao mais significativo, ou seja, da esquerda para a direita.

COW	SEA	TAB	BAR
DOG	TEA	BAR	BIG
SEA	MOB	EAR	BOX
RUG	TAB	TAR	COW

ROW	DOG	SEA	DIG
MOB	RUG	TEA	DOG
BOX	DIG	DIG	EAR
TAB	BIG	BIG	FOX
BAR	BAR	MOB	MOB
EAR	EAR	DOG	NOW
TAR	TAR	COW	ROW
DIG	COW	ROW	RUG
BIG	ROW	NOW	SEA
TEA	NOW	BOX	TAB
NOW	BOX	FOX	TAR
FOX	FOX	RUG	TEA

8.2 (CLRS 8.3-2) Which of the following sorting algorithms are stable: INSERTION-SORT, MERGE-SORT, HEAP-SORT, QUICKSORT? Give a simple scheme that makes any sorting algorithm stable. How much additional time and space does your scheme entail?

Os algoritmos estáveis são INSERTION-SORT e MERGE-SORT. Podemos manter um outro vetor de tamanho n com a ordem relativa dos elementos originalmente, o que demandaria $\Theta(n)$ espaço adicional e não mudaria o comportamento assintótico do algoritmo.

9. CLRS 8.3-4 Mostre como ordenar n inteiros no intervalo $0..n^2 - 1$ em tempo $O(n)$.

Podemos usar o próprio algoritmo RADIX-SORT. Neste caso, temos n inteiros de 2 dígitos, sendo $k = n$ possíveis valores. Logo:

$$\Theta(d(n + k)) = \Theta(2(n + n)) = \Theta(2n) + \Theta(2n) = \Theta(4n) = \Theta(n)$$

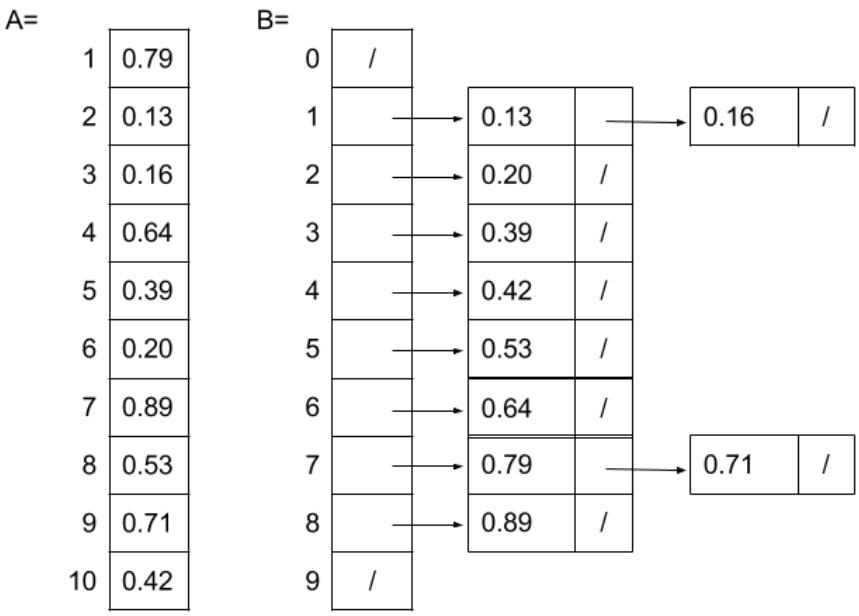
10. Simule a execução do BUCKETSORT com o vetor.

$$A[1..10] = \langle 0.79, 0.13, 0.16, 0.64, 0.39, 0.20, 0.89, 0.53, 0.71, 0.42 \rangle$$

Resposta:
Então cada lista de B é ordenada e concatenada em ordem para gerar o vetor ordenado.

11. **(CLRS 8.4-2**: Qual é o consumo de tempo do pior caso para o BUCKETSORT? Que simples ajuste do algoritmo melhora o seu pior caso para $O(n \lg n)$ e mantem o seu consumo esperado de tempo linear.

Resposta: A resposta do exercício é baseada no algoritmo do livro do CLRS, conforme descrito abaixo:



BucketSort(A)

```
1  Let B[0..n − 1] be a new array
2  n = A.length
3  for i = 0 to n − 1
4        make B[i] an empty list
5  for i = 1 to n
6        insert A[i] into list B[⌊nA[i]⌋]
7  for i = 0 to n − 1
8        sort list B[i] with insertion sort
9  concatenate the lists B[0], B[1], · · · , B[n − 1] together in order
```

O pior caso do BucketSort acontence quando a entrada não segue uma distribuição uniforme e todos os itens caem em um único *Bucket*. Como o algoritmo apresentado no CLRS utiliza *insertion sort* para ordenar cada *Bucket* ele gastará tempo $O(n^2)$ no pior caso. Podemos trocar o *insertion sort* pelo Mergesort ou Heapsort que gastam $O(n \lg n)$ no pior caso. Na prática o *insertion sort* é a melhor opção por que embora gaste tempo $O(n^2)$ *insertion sort* é mais rápido que o Mergesort ou o Heapsort para vetores pequenos, note que para que o BucketSort seja uma boa opção cada *Bucket* tem que ter um número pequeno de elementos.

1. mac338-2011, p2 Suponha que temos que ordenar um vetor com n registros e que a chave de cada registro (o campo que deve ser usado para a ordenação) tem um valor 0 ou 1. Um algoritmo para ordenar esse vetor pode ter um subconjunto das seguintes características:

(i) o algoritmo consome tempo $O(n)$;

(ii) o algoritmo estável;

(iii) o algoritmo ordena no lugar (ou seja, não usa vetor auxiliar para fazer a ordenação; apenas eventualmente um número constante de variáveis simples).

(a) Dê um algoritmo que tenha as características (i) e (ii).
(b) Dê um algoritmo que tenha as características (i) e (iii).
(c) Dê um algoritmo que tenha as características (ii) e (iii).

2. mac338-2011, p2 (Esta questão é uma modificação de um dos exercícios de uma das listas.) Seja $x_1, x_2, \ldots, x_n$ uma sequência de números, onde n é par. Um pareamento de $x_1, x_2, \ldots, x_n$ é uma partição do (multi)conjunto $\{x_1, x_2, \ldots, x_n\}$ em pares. Se P é um pareamento de $x_1, x_2, \ldots, x_n$, então a altura de P é o valor $max\{x_i + x_j : \{x_i, x_j\}$ é um par de $P\}$. Um pareamento de $x_1, x_2, \ldots, x_n$ é ótimo se tem altura mínima.

(a) Escreva (em pseudo-código, como nas aulas) um algoritmo que, dado n e um vetor $x[1..n]$ com n números, encontre e devolva um pareamento ótimo de $x[1], x[2], \ldots, x[n]$. Seu algoritmo deve consumir tempo $O(nlgn)$.
(b) Analise o consumo de tempo do seu algoritmo, concluindo que de fato é $O(nlgn)$.
(c) Prove que ele de fato produz um pareamento ótimo, ou seja, que nenhum emparelhamento pode ter uma altura menor que a do emparelhamento produzido pelo seu algoritmo.

Resposta: Abaixo o algoritmo que retorna o pareamento ótimo. (resposta do gabarito):

```
PAREAMENTO(x, n)
1  ORDENA(x, n) // em ordem crescente
2  for i = 1 to n/2
3        P[i] = {x[i], x[n − i + 1]}
4  return P
```

É fácil ver que o algoritmo consome tempo $O(nlgn)$. Resta justificar porque ele produz um pareamento ótimo.

Tome um pareamento ótimo P* que inclua pares $P[1], \ldots, P[i]$ para i o maior possível. Se $i = n/2$, não há nada a provar: $P = P*$ e, portanto, P é um pareamento ótimo.

Se $i < n/2$, então considere o pareamento P', derivado de $P*$ da seguinte maneira. $EmP*$ os elementos $x[i+1]$ e $x[n-(i+1)+1]$ não formam um par. Então sejam j e k tais que, em $P*$, temos os pares $\{x[i+1], x[j]\}$ e $\{x[n-(i+1)+1], x[k]\}$. Seja P' o pareamento que coincide com $P*$ exceto pela troca entre $x[j]$ e $x[n-(i+1)+1]$ nos pares acima.

Qual é a altura do pareamento P'? Observe que $i+1 < j < n-(i+1)+1$, logo $x[j] \leq x[n-(i+1)+1]$. Ou seja, a soma do segundo par (para onde foi $x[j]$) ou ficou igual ou diminuiu. Em fórmulas, $x[j] + x[k] \leq x[n-(i+1)+1] + x[k]$.

Por outro lado, como $x[i+1] \leq x[k]$, temos também que $x[i+1] + x[n-(i+1)+1] \leq x[k] + x[n-(i+1)+1]$. Ou seja, com certeza a altura de P' é menor ou igual à altura de $P*$. Mas como $P*$ é um pareamento ótimo, essas alturas são iguais e P' é, também, um pareamento ótimo.

Porém P' tem um par a mais em comum com P (coincidiria com P até $i+1$ pelo menos), uma contradição à escolha de $P*$. Ou seja, esse caso não ocorre. Ocorre apenas o caso em que $i = n/2$, e portanto, P é um pareamento ótimo.

Lista 6

1. (**CLRS 15.2-1**) Determine a parentização ótima de um produto de cadeia de matrizes cuja sequência de dimensões é $\langle 5, 10, 3, 12, 5, 50, 6 \rangle$.

Resposta: Como a sequência tem 7 elementos, o valor de $n = 6$. A figura 1 mostra cada iteração da entrada para geração da matriz m que contém o número mínimo de multiplicações escalares necessário para calcular o produto $A_1..A_n$.

l	i	n-l+1	j	k	q
2	1	5	2	1	150
	2	5	3	2	360
	3	5	4	3	180
	4	5	5	4	3000
	5	5	6	5	1500
3	1	4	3	1	960
	1	4	3	2	330
	2	4	4	2	330
	2	4	4	3	960
	3	4	5	3	4800
	3	4	5	4	930
	4	4	6	4	1860
	4	4	6	5	6600
4	1	3	4	1	580
	1	3	4	2	405
	1	3	4	3	630
	2	3	5	2	2430
	2	3	5	3	9360
	2	3	5	4	2830
	3	3	6	3	2076
	3	3	6	4	1770
	3	3	6	5	1830
5	1	2	5	1	4930
	1	2	5	2	1830
	1	2	5	3	6330
	1	2	5	4	1655
	2	2	6	2	1950
	2	2	6	3	2940
	2	2	6	4	2130
	2	2	6	5	5430
6	1	1	6	1	2250
	1	1	6	2	2010
	1	1	6	3	2550
	1	1	6	4	2055
	1	1	6	4	3155

Figura 1: Cálculo do número mínimo de multiplicações escalares

Os valores destacados em azul são aqueles menores para uma dada iteração i, j e, consequentemente, são armazenados na matriz $m[i, j]$, como pode ser visto na figura 2.

	1	2	3	4	5	6
1	0	150	330	405	1655	2010
2		0	360	330	2430	1950
3			0	180	930	1770
4				0	3000	1860
5					0	1500
6						0

Figura 2: Matriz m preenchida após todas as iterações

A figura 3 mostra uma segunda matriz s utilizada para armazenar o valor de k, de forma que possamos rastrear a sequência ótima para multiplicação posteriormente.

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

Figura 3: Cálculo do número mínimo de multiplicações escalares

Sendo assim, a sequência ótima é $((A_1A_2)((A_3A_4)(A_5A_6)))$.

2. (**15.2-2**) Adapte o algoritmo dado em aula para que calcule uma matriz s que pode ser usada para determinar uma ordem ótima de multiplicação das matrizes. Dê um algoritmo recursivo que recebe as matrizes $A_1, \ldots, A_n$ em uma matriz tridimensional A, recebe a matriz s e índices i e j, com $i \leq j$, e faz o produto das matrizes $A_i \ldots A_j$ usando a ordem ótima dada em s.

Resposta: Uma modificação do algoritmo dado no CLRS p. 375 já realiza o procedimento com a matriz s, sendo o parâmetro p a sequência com as dimensões das matrizes e n o tamanho de $p - 1$.

```
MATRIX-CHAIN-ORDER(p, n)
 1  m[1..n, 1..n] e s[1..n, 1..n] são novas tabelas
 2  for i = 1 to n
 3       m[i, i] = 0
 4  for l = 2 to n // l é o tamanho da cadeia
 5       for i = 1 to n − l + 1
 6            j = i + l − 1
 7            m[i, j] = ∞
 8            for k = i to j − 1
 9                 q = m[i, k] + m[k + 1, j] + p_{i−1} p_k p_j
10                 if q < m[i, j]
11                      m[i, j] = q
12                      s[i, j] = k
13  return m, s
```

Agora, basta calcular o resultado das multiplicações das matrizes na sequência ótima dada por s. Quando $i = j$, estamos na base da recursão, ou seja, no ponto em que uma das matrizes de A deve ser retornada para realização da multiplicação. Caso contrário, efetuamos o produto dos pares de matrizes e do resultado do produto dos pares de matrizes que, como é realizado *bottom-up*, termina com uma única matriz quando todas as chamadas recursivas forem concluídas.

```
MATRIX-CHAIN-MULTIPLY(A, s, i, j)
1  if i == j
2       return A[i]
3  else a = MATRIX-CHAIN-MULTIPLY(A, s, i, s[i, j])
4       b = MATRIX-CHAIN-MULTIPLY(A, s, s[i, j] + 1, j)
5       return a * b
```

4. (**15.4-5**) Escreva um algoritmo que, dado n e um vetor $v[1..n]$ de inteiros, determina uma subsequência crescente mais longa de v. Seu algoritmo deve consumir tempo $O(n^2)$. Se puder, dê um algoritmo para este problema que consome tempo $O(nlgn)$.

Resposta: Para este exercício, basta utilizarmos os algoritmos LCS-LENGTH e PRINT-LCS disponíveis no CLRS 15.4, sendo que o parâmetro u é uma cópia do vetor v ordenado por algum algoritmo de ordenação que, neste caso, usamos o QUICKSORT.

```
BUILD-LONGEST-INCREASING-SUBSEQUENCE(v, n)
1  u[1..n] vetor auxiliar utilizado para copiar e ordenar v
2  for i = 1 to n
3       u[i] = v[i]
4  QUICKSORT(u, 1, u.length)
5  c, b = LCS-LENGTH(v, u)
6  PRINT-LCS(b, v, v.length, u.length)
```

Como nós queremos encontrar a maior subsequência crescente de v, e u é a cópia ordenada de v em ordem crescente, o LCS-Length deverá, então, encontrá-la dentre todas as subsequências crescentes dadas por u no vetor original v.

A tabela 1 mostra o tempo de execução do algoritmo Build-Longest-Increasing-Subsequence.

Linha	Tempo
1	$\Theta(1)$
2	$\Theta(n)$
3	$\Theta(n)$
4	$\Theta(nlogn)$
5	$O(n * n)$
6	$O(n + n)$
Total	$O(n^2 + 2n) + \Theta(nlogn) + \Theta(2n) = O(n^2)$

Tabela 1: Tempo de execução

Originalmente, o LCS-Length tem tempo de execução em função de m e n ($O(mn)$), porém, como sabemos que v e u têm o mesmo tamanho (n), teremos um tempo de execução $O(n^2)$. A mesma analogia vale para o algoritmo Print-LCS que é executado na linha 6.

5. Quantas árvores de busca binária existem que guardam 6 diferentes chaves?

Resposta: Vimos em sala que podemos representar 3 nós em 5 árvores de busca binária distintas.

Vamos definir por $t(n)$ o número de diferentes árvores de busca binária para n nós. Sendo assim, o número de diferentes árvores de busca binária que podemos obter deve ter uma raiz, uma subárvore à esquerda com i nós e outra à direita com $n - 1 - i$ nós para cada i, o que nos dá:

$$t(n) = t(0)t(n-1) + t(1)t(n-2) + ... + t(n-1)t(0)$$

Logo, podemos estabelecer $t(n)$ como uma recorrência:

$$t(n) = \begin{cases} 1, & n = 0 \text{ ou } n = 1 \\ \sum_{i=1}^{n} t(i-1)t(n-i), & n > 1 \end{cases}$$

A base da recorrência nos diz que há apenas uma árvore de busca binária com nenhum ou um nó.

Sendo assim, para 6 nós, temos 132 árvores de busca binária:

$$
\begin{aligned}
&t(0) = 1\\
&t(1) = t(0)t(0) = 1\\
&t(2) = t(0)t(1) + t(1)t(0) = 2\\
&t(3) = t(0)t(2) + t(1)t(1) + t(2)t(0) = 5\\
&t(4) = t(0)t(3) + t(1)t(2) + t(2)t(1) + t(3)t(0) = 14\\
&t(5) = t(0)t(4) + t(1)t(3) + t(2)t(2) + t(3)t(1) + t(4)t(0) = 42\\
&t(6) = t(0)t(5) + t(1)t(4) + t(2)t(3) + t(3)t(2) + t(4)t(1) + t(5)t(0) = 132
\end{aligned}
$$

O número de árvores de busca binária também pode ser calculado como o n-*ésimo* número de catalão:

$$C_n = \frac{1}{n+1}\binom{2n}{n} = \frac{(2n)!}{n!(n+1)!} = \frac{(2\times 6)!}{6!(6+1)!} = \frac{12\times 11\times 10\times 9\times 8\times 7!}{6!7!} = 132$$

8. Pode-se generalizar o problema da árvore de busca binária ótima para que inclua na sua descrição uma estimativa do número de buscas mal-sucedidas. Mais concretamente, nessa variante do problema, seriam dados não apenas os números de acessos a_i de cada chave i, mas também números b_i, para $i = 0, ..., n$, representando uma estimativa do número de buscas mal-sucedidas a elementos entre i e $i + 1$ (considere 0 como $-\infty$ e $n - 1$ como $+\infty$ para que b_0 e b_n sejam o que se espera). Encontre a recorrência para o custo de uma árvore de busca binária ótima para esta variante do problema. Escreva o algoritmo de programação dinâmica gerado a partir dessa recorrência. Quanto tempo consome o seu algoritmo?

Resposta: Para generalizarmos o problema da ABB ótima, tal que possamos incluir estimativas de acesso mal sucedidas, vamos pensar em uma ABB que armazena as chaves $k = \langle k_1, k_2, ..., k_n\rangle$ ordenadas e, para cada chave, tenhamos a estimativa do número de acessos a cada elemento de k dado por a_i, para todo $i = 1, 2, ..., n$. Essa representação é a que nós já vimos em sala de aula e pode ser vista na figura 4.

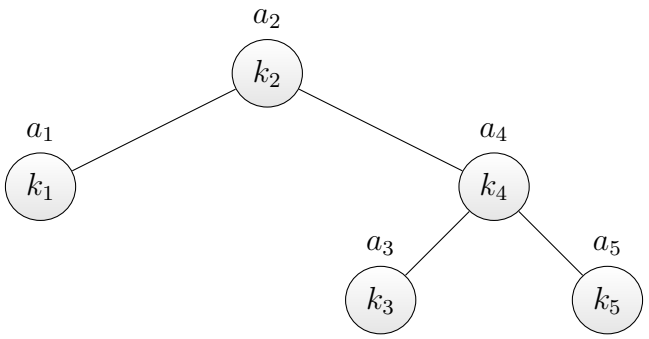


Figura 4: ABB com número de acessos a cada chave

Como temos buscas que são mal sucedidas, ou seja, valores que não estão em k, nós devemos estender a ABB de forma a incluir $n+1$ chaves que representam todas essas buscas que, neste caso, denominamos por $q_0, q_1, ..., q_n$. A figura 5 mostra essa nova representação. Note que, cada nó terminal k_i na estrutura anterior agora recebe novos nós (folhas) que representam as buscas mal sucedidas entre k_i e k_{i+1}. Vale ressaltar que, q_0 representa todas os valores menores que k_1 e q_n representa todos os valores maiores que k_n.

Da mesma forma que nas buscas bem sucedidas, temos a estimativa do número de acessos associado a cada chave q_i não encontrada em k que é dado por $b_0, b_1, ..., b_n$. Por exemplo, se tivéssemos $k = \langle 3, 5, 7, 11, 12 \rangle$, se em uma dada ABB $k_3 = 7$ e $k_4 = 11$, o nó terminal q_3 representaria os valores não encontrados $\{8, 9, 10\}$.

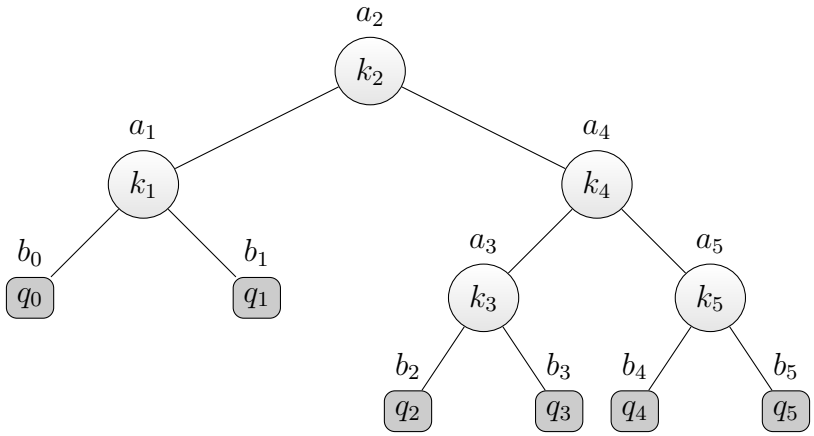


Figura 5: ABB estendida adicionando número de buscas mal-sucedidas

Visto isso, podemos definir a **subestrutura ótima** do problema: se uma ABB é ótima, então as subárvores esquerda e direita também são ótimas. Vale ressaltar que, quando escolhemos uma das chaves, digamos k_r $(i \leq r \leq j)$, como a raiz de uma subárvore $k_i, ..., k_j$, a subárvore à esquerda de k_r deve conter as chaves $k_i, ..., k_{r-1}$, bem como as representantes de buscas mal sucedidas $q_{i-1}, ..., q_{r-1}$ e a subárvore à direita de k_r deve conter as chaves $k_{r+1}, ..., k_j$ e $q_r, ..., q_j$.

Agora, devemos definir uma solução recursiva para encontrar uma ABB ótima para as chaves $k_1, k_2, ..., k_n$ cujo número esperado de comparações para uma busca bem sucedida $a_1, a_2, ..., a_n$ e o número esperado de comparações para uma busca mal sucedida $b_0, b_1, b_2, ..., b_n$ seja mínimo.

Seja $e[i, j]$ o custo esperado para uma busca em uma ABB ótima com as chaves $i, i+1, ..., j$. Vale a seguinte recorrência para $e[i, j]$:

$$e[i,j] = \begin{cases} b_{i-1}, & \text{se } i > j \\ \min\limits_{i \leq r \leq j} e[i, r-1] + e[r+1, j] + w[i,j], & \text{se } i \leq j \end{cases}$$

Onde:

$$w[i,j] = \begin{cases} b_{i-1}, & \text{se } i > j \\ w[i, j-1] + a_j + b_j, & \text{se } i \leq j \end{cases}$$

Ou seja, $w[i,j]$ é a soma das estimativas do número de buscas bem e mal sucedidas para um dado i, j:

$$w[i,j] = \sum_{l=i}^{j} a_l + \sum_{l=i-1}^{j} b_l$$

Agora, basta efetuarmos as alterações necessárias no algoritmo dado em sala de aula para calcularmos as matrizes w e e. Também vamos armazenar o valor de r em uma matriz $root$, ou seja, qual a raiz foi escolhida para uma determinada subárvore $k_i, ..., k_j$. O algoritmo Extended-Optimal-BST mostra o resultado.

Extended-Optimal-BST(a, b, n)

```
 1  for i = 1 to n + 1
 2      e[i, i − 1] = b_{i−1}
 3      w[i, i − 1] = b_{i−1}
 4  for l = 1 to n
 5      for i = 1 to n − l + 1
 6          j = i + l − 1
 7          w[i, j] = w[i, j − 1] + a_j + b_j
 8          e[i, j] = ∞
 9          for r = i to j
10              aux = e[i, r − 1] + e[r + 1, j] + w[i, j]
11              if aux < e[i, j]
12                  e[i, j] = aux
13                  root[i, j] = r
14  return e, root
```

O custo mínimo da ABB ótima é retornado na posição $e[1, n]$. A tabela 2 mostra o tempo de execução do algoritmo Extended-Optimal-BST. As linhas 4-13 têm 3 *loops* encadeados, que tomam no máximo n iterações cada um, o que nos dá um consumo total $\Theta(n^3)$.

Linha	Tempo
1-3	$\Theta(n)$
4-13	$\Theta(n^3)$
14	$\Theta(1)$
Total	$\Theta(n) + \Theta(n^3) = \Theta(n^3)$

Tabela 2: Tempo de execução

10. Escreva uma versão recursiva com memoização do algoritmo descrito em aula para a determinação do custo de uma árvore de busca binária ótima. Quanto tempo consome essa versão do algoritmo para calcular $c(1, n)$?

Basta alterarmos os loops que calculam a tabela $c[i, j]$ por uma versão recursiva memoizada, onde o valor de $c[i, j]$ é retornado, caso já tenha sido calculado. O algoritmo Memoized-Optimal-BST mostra essa mudança.

```
MEMOIZED-OPTIMAL-BST(e, n)
1  // Seja s[0..n] um novo vetor e c[0..n][0..n] uma nova matriz
2  s[0] = 0
3  for i = 1 to n
4       for j = 0 to n
5            c[i, j] = ∞
6       s[i] = s[i − 1] + e[i]
7  return LOOKUP-OPTIMAL-BST(e, s, 1, n)

LOOKUP-OPTIMAL-BST(e, s, i, j)
1  if c[i, j] < ∞
2       return c[i, j]
3  if i > j
4       c[i, j] = 0
5  else for k = i + 1 to j
6            aux = LOOKUP-OPTIMAL-BST(e, s, i, k − 1)+
                   s[j] − s[i − 1]+
                   LOOKUP-OPTIMAL-BST(e, s, k + 1, j)
7            if aux < c[i, j]
8                 c[i, j] = aux
9  return c[1, n]
```

Falta fazer a análise de tempo

12. Escreva uma função que recebe como parâmetros um inteiro $n > 0$ e um vetor que armazena uma sequência de n inteiros e devolve o comprimento de uma subsequência crescente da sequência dada de soma máxima. Sua função deve consumir tempo $O(n^2)$. Modifique a sua função (sem piorar a sua complexidade assintótica) para que ela devolva também uma subsequência crescente da sequência dada de soma máxima.

Resposta: Assim como no exercício 4, podemos usar uma variação do LCS-LENGTH para resolver este problema.

Vamos, então, definir a **subestrutura ótima** do problema. Seja $X[1..n]$ um vetor com n inteiros e $Y[1..n]$ uma cópia do vetor X ordenado, temos que $Z[1..k]$ é uma subsequência crescente de X de soma máxima, se:

- $X[n] = Y[n]$, então $Z[k] = X[n] = Y[n]$ e $Z[1..k-1]$ é subsequência crescente de soma máxima de $X[1..n-1]$ e $Y[1..n-1]$
- $X[n] \neq Y[n]$, então $Z[k] \neq X[n]$ implica que $Z[1..k]$ é subsequência crescente de soma máxima de $X[1..n-1]$ e $Y[1..n]$
- $X[n] \neq Y[n]$, então $Z[k] \neq Y[n]$ implica que $Z[1..k]$ é subsequência crescente de soma máxima de $X[1..n]$ e $Y[1..n-1]$

Logo, percebemos que a subestrutura ótima do problema original não muda. Vamos definir por $c[i, j]$ a soma dos elementos de uma subsequência crescente de soma máxima de $X[1..i]$ e $Y[1..j]$. Vejamos, então, a **recorrência** para calcular o comprimento da subsequência crescente de soma máxima:

$$c[i,j] = \begin{cases} 0, & \text{se } i = 0 \text{ ou } j = 0 \\ c[i-1, j-1] + X[i], & \text{se } i, j > 0 \text{ e } X[i] = Y[j] \\ \max(c[i, j-1], c[i-1, j]), & \text{se } i, j > 0 \text{ e } X[i] \neq Y[j] \end{cases}$$

A diferença para a recorrência original se dá no caso 2, onde o tamanho da LCS era contabilizada. Neste caso, somamos cada elemento de X[i] em uma dada subsequência $X[i..j]$.

O algoritmo BUILD-MAX-SUM-LCS efetua, então, o cálculo desejado a partir de um vetor $X[1..n]$. Inicialmente, uma cópia de X é criada e ordenada com um algoritmo de ordenação que, neste caso, utilizamos o QUICKSORT. Posteriormente, chamamos a função LCS-MAX-SUM-LENGTH que calcula e retorna a matriz c com a soma e a matriz b para que possamos rastrear a subsequência crescente, respectivamente.

Assim que a matriz b é calculada, ela é submetida à rotina GET-LCS-MAX-SUM que retorna, então, o vetor Z com a subsequência crescente de soma máxima, bem como o tamanho da subsequência.

```
BUILD-MAX-SUM-LCS(X, n)
1  Y[1..n] vetor auxiliar utilizado para copiar e ordenar X
2  for i = 1 to n
3       Y[i] = X[i]
4  QUICKSORT(Y, 1, Y.length)
5  c, b = LCS-MAX-SUM-LENGTH(X, Y)
6  Z, l = GET-LCS-MAX-SUM(X, X.length, b)
7  return Z, l
```

```
LCS-MAX-SUM-LENGTH(X, Y)
 1  n = X.length
 2  for i = 0 to n
 3       c[i, 0] = 0
 4  for j = 1 to n
 5       c[0, j] = 0
 6  for i = 1 to n
 7       for j = 1 to n
 8            if X[i] == Y[j]
 9                 c[i, j] = c[i − 1, j − 1] + X[i]
10                 b[i, j] = " ↖ "
11            elseif c[i − 1, j] ≥ c[i, j − 1]
12                 c[i, j] = c[i − 1, j]
13                 b[i, j] = " ↑ "
14            else c[i, j] = c[i, j − 1]
15                 b[i, j] = " ← "
16  return c, b
```

A última linha do algoritmo GET-LCS-MAX-SUM nos devolve a partição de Z que foi preenchida e o tamanho do vetor resultante, ou seja, $n - k$.

```
GET-LCS-MAX-SUM(X, n, b)
 1  k = n
 2  i = n
 3  j = n
 4  while i > 0 and j > 0
 5       if b[i, j] == " ↖ "
 6            Z[k] = X[i]
 7            k = k − 1
 8            i = i − 1
 9            j = j − 1
10       elseif b[i, j] == " ↑ "
11            i = i − 1
12       else j = j − 1
13  return Z[k..n], n-k
```

Sabemos que o tempo de execução do QUICKSORT, LCS-LENGTH e GET-LCS é $\Theta(nlogn)$, $\Theta(mn)$ e $O(m+n)$, respectivamente. As alterações que fizemos em LCS-MAX-SUM-LENGTH e GET-LCS-MAX-SUM das versões originais dos respectivos algoritmos não alteram o comportamento assintótico.

Vale lembrar que, o comportamento assintótico do LCS-LENGTH e GET-LCS são em função de m e n originalmente, como ambos vetores X e Y são de tamanho n, o tempo de execução passa a ser em função de n, exclusivamente.

Portanto, o consumo de tempo total do BUILD-MAX-SUM-LCS é dado na tabela 3:

Linha	Tempo
1	$\Theta(0)$
2-3	$\Theta(n)$
4	$\Theta(nlogn)$
5	$\Theta(n^2)$
6	$O(n+n)$
7	$\Theta(1)$
Total	$\Theta(n) + \Theta(nlogn) + \Theta(n^2) + O(2n) = \Theta(n^2)$

Tabela 3: Tempo de execução

14. Problema 14-4 do CLRS (Planejando uma festa da empresa) O professor Stewart presta consultoria ao presidente de uma corporação que está planejando uma festa da empresa. A empresa tem uma estrutura hierárquica; isto é, a relação de supervisores forma uma árvore com raiz no presidente. O pessoal do escritório classificou cada funcionário com uma avaliação de sociabilidade, que é um número real. Para tornar a festa divertida para todos os partocipantes, o presidente não deseja que um funcionário e seu supervisor imediato participem.

O professor Stewart recebe uma árvore que descreve a estrutura hierárquica da corporação, usando a representação de filho da esquerda, irmão da direita, usanda para armazenamento de árvores enraizadas (olhe na seção 10.4 se precisar). Cada nó da árvore contém além dos ponteiros, o nome de um funcionário e a ordem de sociabilidade desse funcionário. Escreva um algoritmo para compor a lista de convidados que maximize a soma das avaliações de sociabilidade dos convidados. Analise o tempo de execução do seu algoritmo.

Resposta: Vamos resolver esse problema com a técnica de programação dinâmica.

Sub-estrutura ótima: Seja T a árvore hierarquica da empresa, sabemos que dado a solução ótima S que maximiza a soma de sociabilidade da empresa, se a raiz de T pertece a essa solução S a sociabilidade dessa solução é a soma da sociabilidade do presidente (raiz da árvore) e das soluções ótimas das subárvores enraizadas em seus subordinados diretamente, note que seus subordinados não podem ser convidados. Se o presidente não pertence a solução ótima desse problema então a sociabilidade da solução ótima é a soma das sociabilidades das soluções ótimas das subárvores enraizadas em seus subordinados direitos. **Prova:** Vamos assumir por contradição que na solução ótima, dado as subárvores enraizadas nos subordinados diretos do presidente exista uma delas que não seja ótima, então com certeza podemos trocar essa subárvore que não é ótima pela subárvore ótima e conseguir uma solução com sociabilidade maior que a solução ótima assumida que é uma contradição! Já que a solução ótima maximiza a sociabilidade.

Recorrência: Vamos definir identificar cada nó de nossa árvore como o nome do funcionário que o nó representa. Então definimos $c[x]$ como:

> $c[x]$:= é a soma maxíma da sociabilidade dos convidados da festa da árvore enraizada em x tal que dado que um nó foi convidado seu pai não foi.

definimos $p(x)$ como o grau de sociabilidade do nó x, $netos(x)$ como a operação que devolve os netos do nó x na árvore, e $filhos(x)$ a operação que devolve os nós filhos de x na árvore. Também definimos que $netos(x) = \emptyset$ e $filhos(x) = \emptyset$ quando x não tem nós netos ou filhos respectivamente.

Agora definimos a recorrência da seguinte forma:

$$c[x] = \begin{cases} 0 & \text{,se } x = \emptyset \\ p(x) & \text{,se } x \text{ é uma folha} \\ \max\left(p(x) + \sum_{y \in netos(x)} c[y], \sum_{y \in filhos(x)} c[y]\right) & \text{,caso contrário} \end{cases}$$

Algoritmo de programação dinâmica: Abaixo segue o algoritmo de programação dinâmica.

15. PC 111105 (Cortes de tora) Você deve cortar uma tora de madeira em vários pedaços. A empresa mais em conta para fazer isso é a *Analog Cutting Machinery* (ACM), que cobra de acordo com o comprimento da tora a ser cortada. A máquina de corte deles permite que apenas um corte seja feito por vez. Se queremos fazer vários cortes, é fácil ver que ordens diferentes destes cortes levam a preços diferentes. Por exemplo, considere uma tora

com 10 metros de comprimento, que tem que ser cortada a 2, 4 e 7 metros de uma de suas extremidades. Há várias possibilidades. Podemos primeiramente fazer o corte dos 2 metros, depois dos 4 e depois dos 7. Tal ordem custa 10+8+6 = 24, porque a primeira tora tinha comprimento 10, o que restou tinha 8 metros de comprimento e o último pedaço tinha comprimento 6. Se cortássemos na ordem 4, depois 2, depois 7, pagaríamos 10 + 4 + 6 = 20, que é mais barato.
Seu chefe encomendou um programa que, dado o comprimento l da tora e k pontos $p_1, ..., p_k$ de corte da tora, encontre o custo mínimo para executar esses cortes na ACM.

Resposta: Assumindo que os pontos de corte estão ordenados, ou seja, $p_1 \leq p_2 \leq p_3 \leq ... \leq p_n$, podemos olhar este problema de forma semelhante ao da ABB ótima.

Antes de mais nada, devemos inserir o valor 0 na posição 0 de p e o tamanho l da tora na posição p_{k+1}, para que possamos calcular o tamanho restante da tora após um corte.

Podemos. então, caracterizar a **suesbtrutura ótima** da seguinte maneira: se o custo de corte da tora t de t$[i..j]$ é mínimo, então, o custo de corte da tora de $t[i..m]$ e $t[m..j]$ também é mínimo.

Vamos definir por $c[i, j]$ o custo mínimo de corte da tora de $t[i..j]$. Vejamos agora a **recorrência** para calcular o custo mínimo de corte da tora:

$$c[i,j] = \begin{cases} 0, & \text{se } i+1 = j \\ \min\limits_{i<m<j} c[i,m] + c[m,j] + p_j - p_i, & \text{se } i+1 < j \end{cases}$$

A figura 6 mostra a tabela $c[i, j]$ preenchida após todas as iterações para a entrada $p = \langle 4, 5, 7, 8\rangle$ e $l = 10$. A matriz será preenchida pelas diagonais, de cima para baixo. Vale lembrar que inserimos os valores 0 e l em p antes de executar a recorrência, resultando em $p' = \langle 0, 4, 5, 7, 8, 10\rangle$.

c[i, j]

	1	2	3	4	5
0	0	5	12	15	22
1		0	3	7	12
2			0	3	8
3				0	3
4					0
5					

Figura 6: Matriz c preenchida após todas as iterações

O algorimto Analog-Cutting-Machinery calcula o custo mínimo de corte da tora, com base na recorrência dada por $c[i, j]$.

```
ANALOG-CUTTING-MACHINERY(p, l)
 1  INSERT(p, 0, 0)
 2  INSERT(p, p.length, l)
 3  n = p.length
 4  for i = 0 to n − 1
 5        c[i, i + 1] = 0
 6  for k = 1 to n
 7        for i = 0 to n − k − 1
 8              j = i + k + 1
 9              c[i, j] = ∞
10              for m = i + 1 to j
11                    aux = c[i, m] + c[m, j]
12                    if aux < c[i, j]
13                          c[i, j] = aux
14              c[i, j] = c[i, j] + p[j] − p[i]
15  return c[0, n]
```

O custo mínimo do corte da tora é retornado na posição $c[0, n]$. A tabela 4 mostra o tempo de execução do algoritmo ANALOG-CUTTING-MACHINERY. Como as inserções em p nas linhas 1-2 podem ser feitas em tempo $\Theta(n)$, o consumo total é $\Theta(n^3)$. já que as linhas 6-14 têm 3 *loops* encadeados que tomam, no máximo, n iterações cada um.

Linha	Tempo
1-2	$\Theta(n)$
3	$\Theta(1)$
4-5	$\Theta(n)$
6-14	$\Theta(n^3)$
15	$\Theta(1)$
Total	$\Theta(3n) + \Theta(n^3) = \Theta(n^3)$

Tabela 4: Tempo de execução

18. Problema 15-7 do CLRS (Programando para maximizar o lucro) Suponha que você tem uma máquina e um conjunto de n trabalhos, identificados pelos números $1, 2, \ldots, n$, para processar nessa máquina. Cada trabalho j tem um tempo de processamento t_j , um lucro p_j e um prazo final d_j . A máquina só pode processar um trabalho de cada vez, e o trabalho j deve ser executado ininterruptamente por t_j unidades de tempo consecutivas. Se o trabalho j for concluído em seu prazo d_j , você recebe um lucro p_j, mas, se ele for completado depois do seu prazo final, você não recebe nenhum lucro. Escreva um algoritmo para encontrar a ordem de execução dos trabalho que maximiza a soma dos lucros, supondo que todos os tempos de processamento são inteiros entre 1 e n. Qual é o tempo de execução do seu algoritmo?

Resposta: Temos $a_1, a_2, a_3, \ldots, a_n$ trabalhos e cada trabalho a_j tem um tempo de processamento t_j, lucro p_j e prazo d_j.

Devemos ordenar os a_n trabalhos de acordo com o prazo de entrega e, desta forma, podemos pensar na solução como uma variação do problema da mochila onde a **subestrutura ótima** é: se o lucro dos trabalhos $a[1..i]$ é máximo, então, o lucro de $a[1..i-1]$ também é máximo, desde que o prazo de entrega d_i e d_{i-1}, respectivamente, seja respeitado.

Assim como no problema da mochila nós temos uma capacidade máxima W, aqui, temos um prazo máximo $D = d_n$ para executar o trabalho a_n, já que ordenamos os trabalhos pelo prazo de entrega.

Vamos definir por $w[i, j]$ o lucro máximo obtido processando os trabalhos $a_1, a_2, a_3, \ldots a_i$ no prazo de entrega j. A recorrência pode, então, ser definida da seguinte maneira:

$$w[i,j] = \begin{cases} 0 & \text{, se } i = 0 \text{ ou } j = 0 \\ w[i-1,j] & \text{, se } t_i > j \\ \max\{w[i-1,j], w[i-1,j-t_i] + p_i\} & \text{, se } t_i \leq j \end{cases}$$

Note que o caso base é quando não executamos nenhum trabalho ou o prazo para processamento é nulo. O segundo caso se dá quando o tempo de processamento t_i do trabalho a_i é maior que o limite de prazo j. Percebemos, também, que a matriz $w[i, j]$ tem dimensões nxD e será preenchida da esquerda para direita, de cima para baixo.

O algoritmo está implementado em *maximum_profit.py* e tem ordem de complexidade $\Theta(nD)$ já que a ordenação pode ser feita em tempo $\Theta(nlgn)$. Assim como o problema da mochila, D representa o prazo máximo que temos para executar os trabalhos, o que corresponde a d_n assim que o vetor d é ordenado, o que faz com que o algoritmo seja pseudo-polinomial já que D não é, garantidamente, polinomial no tamanho da entrada.

Se considerarmos, $D = 1.000.000.000.000$, precisamos de 40 bits para representar este número, então, o tamanho da entrada é 40, mas o tempo de execução, neste caso, usa este fator de D, o que nos dá $O(2^{40})$.

Logo, o tempo de execução seria, de forma mais precisa, $O(n2^n)$, se a quantidade de bits para representar $D = n$, o que tornaria o algoritmo exponencial.

21. (Bandeiras) No dia da Bandeira na Rússia o proprietário de uma loja decidiu decorar a vitrine de sua loja com faixas de tecido das cores branca, azul e vermelha. Ele deseja satisfazer as seguintes condições: faixas da mesma cor não podem ser colocadas uma ao lado da outra. Uma faixa azul sempre está entre uma branca e uma vermelha, ou uma vermelha e uma branca. Escreva um programa que, dado o número n de faixas a serem colocadas na vitrine, calcule o número de maneiras de satisfazer as condições do proprietário.

Exemplo: Para $n = 3$, o resultado são as seguintes combinações: BVB, VBV, BAV, VAB.

Resposta: Podemos pensar no processo de formação da combinação das cores da seguinte maneira: a primeira posição, obrigatoriamente, deve conter um B ou um V. Vamos fazer uma simulação iniciando com B que, ao final, basta multiplicarmos por 2 para termos o número total de combinações. No segundo nível teremos então duas possibilidades: V e A.

Percebemos então que, a cada vez que temos um A, temos apenas uma possibilidade no nível seguinte. Quando temos um B ou um V, temos duas novas possibilidades. Devemos

ter o cuidado ao finalizar as combinações para que não haja nós adjacentes com o mesmo valor e, também, nenhuma terminação em A.

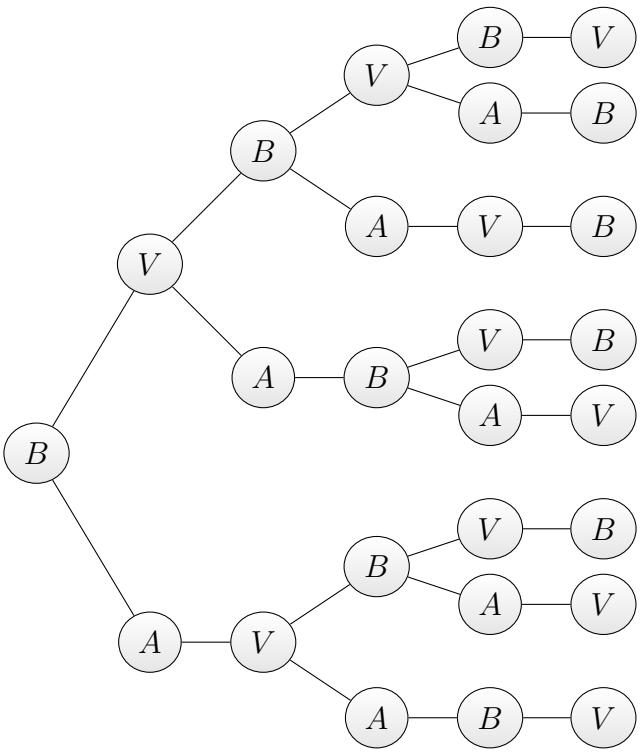


Figura 7: Árvore de combinações possíveis para $n = 6$

Logo, percebemos que o número de combinações de $1..n-1$ cresce conforme a série de Fibonacci e, portanto, podemos calcular o número de maneiras de satisfazer as condições do proprietário com a própria recorrência que calcula a série de Fibonacci:

$$t(n) = \begin{cases} 0, & \text{se } n = 0 \\ 1, & \text{se } n = 1 \\ t(n-1) + t(n-2) + 1, & \text{se } n \geq 2 \end{cases}$$

Como temos duas possibilidades (iniciando a série com B ou V), basta multiplicarmos o resultado da série de Fibonacci(n) por 2 e teremos, então, o número de diferentes maneiras que podemos organizar as faixas de tecido na vitrine. O algoritmo ARRANGE-FLAGS mostra a solução que, na verdade, é uma pequena alteração do algoritmo original de Fibonacci para programação dinâmica.

ARRANGE-FLAGS(n)
1 $f[0] = 0$
2 $f[1] = 1$
3 **for** $i = 2$ **to** n
4 $\quad f[i] = f[i-1] + f[i-2]$
5 **return** 2 * f[n]

Sendo assim, no exemplo supra citado, para $n = 6$, temos que o número de combinações possíveis é 16.

Obviamente o consumo de espaço e tempo do algoritmo é $\Theta(n)$, assim como é o original.

3. mac338-2008, p2 Dado n e uma cadeia de n caracteres $s[1..n]$ que você acredita ser um texto corrompido, em que toda a pontuação foi removida (de modo que pareça com alguma coisa assim... "eraumavezumgatoxadrez..."). Você deseja reconstruir o documento, usando um dicionário, que está disponível na forma de uma função booleana $dict(.)$ para cada cadeia de caracteres w,

$$dict(.) = \begin{cases} true & \text{se } w \text{ é uma palavra válida} \\ false & \text{caso contrário} \end{cases}$$

Escreva um algoritmo de programação dinâmica que, dado n e uma cadeia de caracteres $s[1..n]$, determina se s pode ser reconstituída como uma sequência de palavras válidas. O seu algoritmo deve consumir tempo $O(n^2)$. Justifique porque ele funciona (por exemplo explicando a validade da recorrência de onde ele foi derivado) e porque o seu consumo de tempo é $O(n^2)$.

Neste caso, como a tabela com as possíveis palavras válidas já foi dada, basta consultarmos a tabela com um algoritmo memoizado.

A subestrutura ótima é, se de $s[i..k]$ temos uma palavra válida e de $s[k+1..j]$ também, então, temos que $s[i..j]$ pode ser reconstituída como uma sequência de palavras válidas.

A recorrência fica, então, da seguinte maneira:

$$dp[i,j] = \begin{cases} 0 & \text{, se } i = j-1 \text{ e } dict(s[i,j]) = false \\ 1 & \text{, se } dict(s[i,j]) \\ \max\limits_{i \le k \le j} \{dp[i,k], dp[k+1,j]\} & \text{, c.c.} \end{cases}$$

O algoritmo RECONSTITUI mostra a implementação da recorrência, sendo que as subsequências são consultadas por $dict(.)$ de forma memoizada.

```
RECONSTITUI(s, i, j)
1  if DICT(s[i : j])
2        return true
3  else for k = i to j
4           if RECONSTITUI(s, i, k) and RECONSTITUI(s, k + 1, j)
5                 return true
6  return false
```

Como nós temos n subproblemas e n possibilidades no máximo, o consumo de tempo será $O(n^2)$, já que a cada chamada recursiva nós gastamos tempo constante.

3. mac338-2011, p2 Você conhece a tartaruga Yertle?

O trono de Yertle é composto de uma pilha de tartarugas, conforme a figura .

Cada tartaruga, ao se alistar para fazer parte do trono, fornece o seu peso e a sua força. A sua missão é determinar uma pilha de tartarugas, dentre as alistadas, o maior possível, em que nenhuma tartaruga quebre por entrar na pilha. O peso de cada tartaruga é dado em gramas, e a sua força é em gramas também.

A força de uma tartaruga indica quanto peso, incluindo o seu, ela aguenta. Ou seja, uma tartaruga de peso 300 gramas que tem força de 1000 gramas pode carregar 700 gramas de tartarugas sobre ela. Numa pilha válida de tartarugas, cada tartaruga tem força para sustentar as tartarugas que estão acima dela na pilha. Considere as três possíveis maneiras de ordenarmos as tartarugas:

(i) em ordem decrescente de peso;
(ii) em ordem decrescente de força;
(iii) em ordem decrescente de força menos peso.

(a) Para cada uma destas três ordens, prove ou dê um contra-exemplo para a seguinte afirmação: qualquer pilha válida de tartarugas pode ser reorganizada para que as tartarugas apareçam nesta ordem, de baixo para cima, e a pilha continue válida. Dica: a afirmação é verdadeira para apenas uma das ordens.
(b) Seja n o número de tartarugas, $p[1..n]$ o peso das n tartarugas e $f[1..n]$ a força das n tartarugas.

Suponha que as tartarugas estão ordenadas na ordem correta (a que funciona das três

acima) e que todas tem peso maior que zero. Seja $c[i, w]$ o número de tartarugas na maior pilha válida de tartarugas composta de tartarugas de 1 a i com peso igual a w. (Se não houver pilha válida, deixe $c[i, w] = -1$.)

Escreva uma recorrência válida para $c[i, x]$ (acho que é w aqui). Não esqueça de definir o caso base da recorrência! Explique porque sua recorrência está correta, ou seja, prove (indutivamente) que $c[i, w]$ é o número de tartarugas na maior pilha válida de tartarugas composta de tartarugas de 1 a i com peso igual a w.

(c) A partir da sua recorrência, escreva (em pseudo-código, como nas aulas) um algoritmo de programação dinâmica que, dados n, p e f, representando os dados de n tartarugas, determine e devolva o número de tartarugas em uma pilha válida máxima formada com estas tartarugas. O seu algoritmo deve claramente corresponder à sua recorrência.

(d) Analise o consumo de tempo do seu algoritmo em função de n.

CLRS (Outros)

A.1-7 Avalie o produtório $\prod_{k=1}^{n} 2(4^k)$.

$$\prod_{k=1}^{n} 2(4^k) = \prod_{k=1}^{n} 2((2^2)^k) = \prod_{k=1}^{n} 2(2^{2k}) = \prod_{k=1}^{n} 2^{2k+1}$$

Se avaliarmos o produtório para $n = 3$, por exemplo:

$$\prod_{k=1}^{n} 2^{2k+1} = 2^{2+1} \times 2^{4+1} \times 2^{6+1}$$

Percebemos que o expoente de 2 cresce em uma série aritmética:

$$\sum_{k=1}^{n} 2k + 1 = \sum_{k=1}^{n} 2k + \sum_{k=1}^{n} 1 = 2\sum_{k=1}^{n} k + n = 2(\frac{n(n+1)}{2}) + n = n(n+2)$$

Portanto:

$$\prod_{k=1}^{n} 2(4^k) = 2^{n(n+2)}$$