: Complexidade e Projeto de Algoritmos **versão: 5/6/24**

Tópico 3 — Algoritmos de ordenação

Professor: Vinícius Dias *Autor: Vinícius Dias*

Sumário

3.1 Introdução sobre algoritmos de ordenação

comentar sobre a importância de ordenação e que geralmente ordenamos chaves de itens

3.2 Bubble sort

A ideia do bubblesort é realizar sucessivas trocas de elementos adjacentes que estejam fora de ordem até que todos estejam garantidamente em sua posição correta ordenada. É importante observar que a quantidade máxima de vezes em que um elemento do arranjo se encontre fora de ordem com seu adjacente é $n-1$, sendo n o tamanho do arranjo. Em outras palavras, ele deveria ser trocado com todos os outros elementos do arranjo que não ele próprio e por isso, o laço mais externo do algoritmo é executado $n-1$ vezes. Na primeira iteração, o menor elemento é levado à primeira posição. Na segunda iteração, o segundo menor é levado à segunda posição, e assim por diante.

```
BUBBLE-SORT(A)
for i ← 1 to A.length − 1
    for j ← A.length downto i + 1
        if A[j] < A[j − 1]
            swap(A, j, j − 1)
```

Complexidade de tempo (pior caso e melhor caso): $\Theta(n^2)$, pois a operação de comparação entre elementos acontece sem restrições dentro dos dois laços aninhados.

Complexidade de espaço (pior caso e melhor caso): $\Theta(1)$, pois usamos apenas uma quantidade constante de memória para variáveis auxiliares além da memória fornecida na entrada.

3.3 Insertion sort

A ideia do insertion sort é começar com um subarranjo ordenado e adicionar elemento a elemento em sua posição correta no nesse subarranho ordenado. Ao final, o subarranjo ordenado vai conter todos os elementos do arranjo e portanto, o problema estará resolvido. A analogia para este método é a ordenação de cartas de baralho na mão: a cada carta retirada do monte é posicionado na ordem, em sua posição correta na sequência em construção.

```
INSERTION-SORT(A)
for i ← 2 to A.length
  elem ← A[i]
  j ← i − 1
  while j ≥ 1 and elem < A[j]
    A[j + 1] ← A[j]
    j ← j − 1
  A[j + 1] ← elem
```

Complexidade de tempo (pior caso): $\Theta(n^2)$, sendo que isso acontece quando a comparação $elem < A[j]$ acontece todas as vezes possíveis a cada iteração do laço exterior, isto é, $i-1$ vezes.

Complexidade de tempo (melhor caso): $\Theta(n)$, sendo que isso acontece quando a comparação $elem < A[j]$ é feita apenas uma vez a cada iteração do laço exterior, isto é, quando o arranjo já se encontrar ordenado.

Complexidade de espaço (pior caso e melhor caso): $\Theta(1)$, pois usamos apenas uma quantidade constante de memória para variáveis auxiliares além da memória fornecida na entrada.

3.4 Merge sort

O merge sort é um algoritmo de divisão e conquista que utiliza intercalação para combinar subarranjos ordenados dois a dois em arranjos maiores também ordenados. O processo de intercalação (merge) pode ser feito de forma eficiente em tempo linear partindo da premissa que os arranjos de entrada são fornecidos ordenados.

```
MERGE-SORT(A)
if A.length = 1 or A.length = 0 return
mid ← ⌊A.length/2⌋
left ← A[1···mid]
right ← A[mid+1···A.length]
MERGE-SORT(left)
MERGE-SORT(right)
MERGE(left, right, A)
```

```
MERGE(left, right, A)
i ← 1; j ← 1; k ← 1
while i ≤ left.length and j ≤ right.length
    if left[i] < right[j]
        A[k] ← left[i]
        i ← i + 1
    else
        A[k] ← right[j]
        j ← j + 1
    k ← k + 1
while i ≤ left.length
    A[k] ← left[i]
    i ← i + 1; k ← k + 1
while j ≤ right.length
    A[k] ← right[j]
    j ← j + 1; k ← k + 1
```

Complexidade de tempo (pior caso e melhor caso): A recorrência desse algoritmo é a seguinte:

$$T(n) = \begin{cases} 2T(n/2) + \Theta(n) & \text{se } n > 1 \\ \Theta(1) & \text{se n<=1} \end{cases}$$

Pois resolvemos dois subproblemas de metade do tamanho original n ($2T(n/2)$), representando o custo de conquistar; gastamos $\Theta(1)$ para dividir com o problema com uma operação aritmética simples; e gastamos $\Theta(n)$ para criar os dois arranjos intermediários das metades e para intercalá-los (merge).

Complexidade de espaço (pior caso e melhor caso): $\Theta(n)$, por conta dos dois arranjos das metades que precisam ser alocados para as chamadas recursivas.

3.5 Quick sort

Também se trata de um algoritmo de divisão e conquista, mas que gera dois subproblemas de tamanho variável. A ideia é escolher um pivot que servirá como elemento central que particiona os elementos do arranjo em $\leq pivot$ e $> pivot$. Cada uma das metades é então ordenada recursivamente.

```
QUICK-SORT(A, l, r)
if l ≥ r return
p ← PARTITION(A, l, r)
QUICK-SORT(A, l, p − 1)
QUICK-SORT(A, p + 1, r)
```

```
PARTITION(A, l, r)
pivot ← A[r]
i ← l − 1
for j ← 1 to r − 1
    if A[j] ≤ pivot
        i ← i + 1
        swap(A, j, i)
swap(A, i + 1, r)
return i + 1
```

Complexidade de tempo (pior caso): $\Theta(n^2)$, representando duas partições desbalanceadas, uma com zero elementos e uma com todo o restante de elementos menos o pivot. Se isso acontecer, a cada chamada o problema diminui de apenas uma unidade (o pivot).

Complexidade de tempo (melhor caso): $\Theta(n \log n)$, representando um particionamento ideal a cada chamada recursiva, isto é, totalmente balanceado.

Complexidade de espaço: $\Theta(1)$, pois o algoritmo é *in-place*.

$$T(n) = T(n/10) + T(9n/10) + n$$

$$a_1 = 1; b_1 = 1/10; a_2 = 1; b_2 = 9/10; f(n) = n$$

$$1(1/10)^p + 1(9/10)^p = 1 \implies p = 1$$

$$T(n) \in \Theta\left(n^1 + n^1 \int_{n_0}^{n} \frac{x}{x^2} dx\right) = \Theta(n \ln n)$$

3.6 Heap sort

O algoritmo heap sort se baseia em uma estrutura de dados chamada heap, usada para garantir o acesso ao maior (ou menor) elemento com custo constante. O heap máximo pode ser visualizado como uma árvore binária de chaves que mantém a seguinte propriedade: todo nó filho da árvore é menor ou igual ao seu pai. Uma característica importante dessas estruturas heap é que se houver alguma violação de sua propriedade em um único nó, é possível reorganizar seus itens em um heap válido em tempo $\Theta(\log n)$ – isso é feito através do procedimento MAX-HEAPIFY.

```
LEFT(i)
  return 2i
```

```
RIGHT(i)
  return 2i + 1
```

```
MAX-HEAPIFY(A, i) // Θ(h)
  l ← LEFT(i); r ← RIGHT(i)
  if l ≤ A.heapsize and A[l] > A[i]
     largest ← l
  else largest ← i
  if r ≤ A.heapsize and A[r] > A[largest]
     largest ← r
  if largest ≠ i
     swap(A, largest, i)
     MAX-HEAPIFY(A, largest)
```

```
BUILD-MAX-HEAP(A) // Θ(n)
  A.heapsize ← A.length
  for i ← ⌊A.length/2⌋ downto 1
     MAX-HEAPIFY(A, i)
```

```
HEAP-SORT(A) // Θ(n log n)
  BUILD-MAX-HEAP(A)
  for i ← A.heapsize downto 2
     swap(A, 1, i)
     A.heapsize ← A.heapsize − 1
     MAX-HEAPIFY(A, 1)
```

Complexidade de tempo (pior caso): $\Theta(n \log n)$. A complexidade do algoritmo MAX-HEAPIFY é da ordem da altura do nó na posição i. O procedimento BUILD-MAX-HEAP realiza chamadas ao MAX-HEAPIFY na metade dos nós (a segunda metade sempre serão folhas da árvore binária e portanto, já são heaps válidos!). Por ser uma árvore binária, temos mais nós com altura pequena do que nós com altura grande e portanto, ao somar os custos de MAX-HEAPIFY para cada nó na metade inferior do arranjo, temos um somatório que descresce muito rápido em seus termos, levando a um custo linear $\Theta(n)$. Juntando tudo, o procedimento heap sort realiza $n-1$ chamadas ao procedimento MAX-HEAPIFY e esse custo domina a chamada inicial linear da construção do heap.

Complexidade de espaço: $\Theta(1)$, pois o algoritmo é *in-place*.

3.7 Ordenação em tempo linear

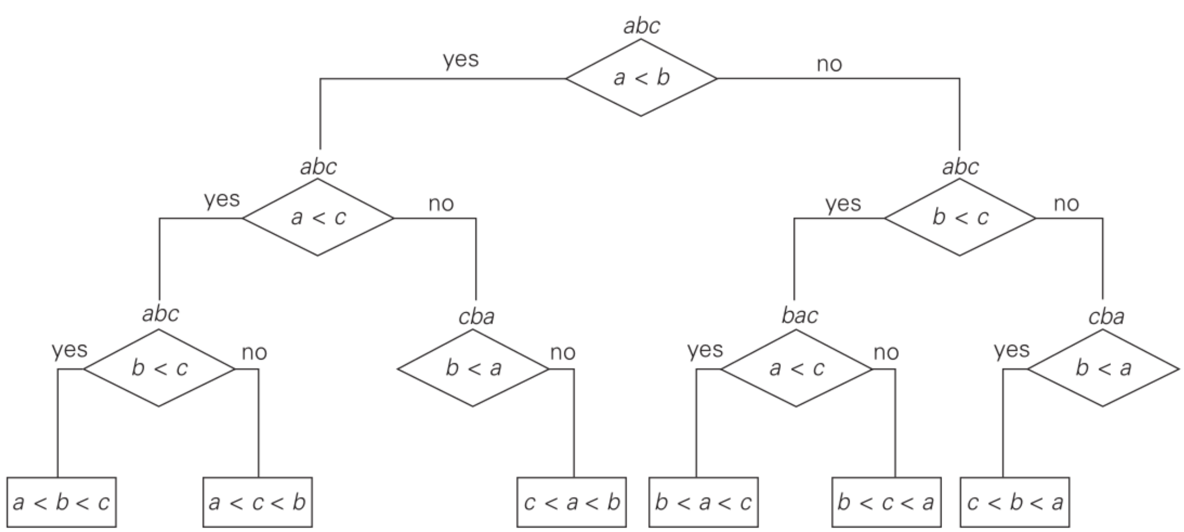
Uma abordagem tradicional em estudo de algoritmos é classificá-los em classes de complexidade assintótica, permitindo a comparação direta entre algoritmos. Uma segunda abordagem complementar é ser capaz de afirmar algo sobre a complexidade de um *problema* frente a um modelo de computação. Em outras palavras, podemos querer responder o seguinte:

> Dados um modelo de computação e um problema, qual é a complexidade mínima que um algoritmo precisa ter para ser correto em solucionar o problema?

Veja que a resposta para essa pergunta seria uma afirmação muito mais forte do que dizer algo sobre a complexidade de algoritmos isoladamente, já que ela envolve considerar todos os infinitos e possíveis algoritmos que solucionam um problema. Chamamos isso de *limites inferiores para problemas*.

3.7.1 Limite inferior para ordenação

Se considerarmos um modelo de computação baseado em operações de comparação, podemos ilustrar a operação de qualquer algoritmo de ordenação através de uma árvore de decisão. Uma árvore de decisão é uma árvore binária onde cada nó representa uma comparação feita em algum momento no algoritmo (operação) e as arestas representam as respostas para essas comparações: verdadeiro ou falso. Dessa forma, um caminho da raíz até uma das folhas dessa árvore representa uma execução do algoritmo, ou o conjunto de testes/decisões que o algoritmo toma dependendo da entrada. Veja um exemplo para um algoritmo que ordena três números:



Podemos generalizar para o problema de ordenação de um arranjo qualquer $[a_1, \cdots, a_n]$ e vamos constatar que as folhas dessa árvore binária de decisão representam possíveis saídas de um algoritmo de ordenação, isto é: $\#folhas \geq n!$, pois cada permutação do arranjo precisa ter uma folha representante, senão o algoritmo seria incorreto!

A altura dessa árvore representa exatamente o número de comparações (operações) realizadas em alguma execução, seja lá qual for o algoritmo. Dessa forma, se estimarmos a altura da árvore estaremos na verdade estimando o custo mínimo que um algoritmo de ordenação precisa ter para resolver o problema corretamente para qualquer instância de entrada. Sabemos também que a altura h de uma árvore binária com k folhas é, pelo menos, $\log_2 k$:

$$
\begin{aligned}
\text{Sabemos que: } & h \geq \log_2(\#folhas), \#folhas \geq n! \\
\text{Aplicando log e combinando: } & h \geq \log_2(\#folhas) \geq \log_2(n!) \\
\text{Desenvolvendo: } & h \geq \log_2(n!) = \log_2(n) + \log_2(n-1) + \cdots + \log_2(1) \\
& \geq \sum_{i=1}^{n} \log_2(i) \geq \sum_{i=n/2}^{n} \log_2(i) \geq \sum_{i=n/2}^{n} \log_2(n/2) \\
& = \sum_{i=n/2}^{n} \log_2(n/2) - \sum_{i=n/2}^{n} \log_2(2) = (n/2+1)\log_2(n/2) - (n/2+1) \in \Omega(n \log_2 n)
\end{aligned}
$$

Conclusão: Não pode existir nenhum algoritmo para o modelo de computação baseado em comparações que tenha um custo assintótico melhor do que $\Omega(n \log n)$.

3.7.2 Counting sort

A primeira etapa é alocar um arranjo de tamanho $k+1$ que será usado para contar o número de ocorrências de cada elemento. Com isso, vamos acumular nas posições de C de maneira que $C[i]$ contenha o número de elementos $\leq i$, dessa forma podemos identificar para cada elemento de A a posição exata que ele deve ocupar no arranjo ordenado B. Com C iremos então, para cada elemento de A em sua ordem inversa, colocá-lo na sua posição correta em B. Toda vez que adicionamos um novo elemento em sua posição correta precisamos decrementar a quantidade de elementos menores ou iguais àquele, para que repetições de um mesmo elemento possam ser posicionadas em suas posições corretas também.

```
COUNTING-SORT(A, B, k)
C ←"novo arranjo com k+1 posições: 0···k"
for i ← 0 to k
    C[i] ← 0
for j ← 1 to A.length
    C[A[j]] ← C[A[j]] + 1
for i ← 1 to k
    C[i] ← C[i] + C[i−1]
for j ← A.length downto 1
    B[C[A[j]]] ← A[j]
    C[A[j]] ← C[A[j]] − 1
```

3.7.3 Radix sort

3.7.4 Bucket sort