# MAC0344 - ANÁLISE DE ALGORITMOS

## ESTUDOS P1

### TÓPICOS DAS AULAS

- aula 1
  - [ ] ordenação
  - [ ] notação O

- aula 2
  - [ ] notação Omega
  - [ ] notação Theta

- aula 3
  - [ ] divisão e conquista
  - [ ] mergesort
  - [ ] intercalação
  - [ ] resolução de recorrências

- aula 4
  - [ ] particione
  - [ ] quicksort
  - [ ] árvore de recorrência

- aula 5
  - [ ] esperança
  - [ ] consumo de tempo esperado

- aula 6
  - [ ] quicksort aleatorizado
  - [ ] select aleatorizado
  - [ ] ordenação por inserção (insertion sort)
  - [ ] árvore de decisão
  - [ ] limite inferior

- aula 7
  - [ ] counting sort
  - [ ] radix sort
  - [ ] bucket sort

- aula 8
  - [ ] multiplicação de inteiros gigantescos
  - [ ] Karatsuba

## COMPLEXIDADES

- MergeSort: $\Theta(n \lg n)$ estável
  - usa o intercala

MERGE SORT $(A, p, q)$ (mescla)
MERGE SORT $(A, p, q)$
INTERCALA $(A, p, q, r)$

- Intercala: $\Theta(n)$
  - intercala dois vetores que estão em ordem crescente
- Quicksort: $O(n)$
  - usa o particione

PARTICIONE $(A, p, q, r)$
QUICKSORT $(A, p, r)$
QUICKSORT $(A, p, r)$

- Particione: $\Theta(n)$
  - divide o vetor em maiores e menores que o pivô
- Selection Sort: $O(n^2)$
- Insertion Sort: $O(n^2)$ estável
- Quicksort aleatorizado: $\Theta(n \log n)$
- Select aleatorizado: $\Theta(n)$

pivô aleatório usando o particione-aleatório

- Select ordenado: $\Theta(n \lg n)$
- Counting Sort: $\Theta(K+n)$ estável
  - se $K = O(n)$, o consumo de tempo é $\Theta(n)$
- RadixSort: $\Theta(d(K+n))$ estável
  - se $d$ é limitado por uma constante (ou seja, se $d = O(1)$) e $K = O(n)$, então o consumo de tempo é $\Theta(n)$
- BucketSort: $O(n)$
  - tempo esperado se os números em $A[1 \dots n]$ são uniformemente distribuídos no intervalo $[0, 1)$ é $O(n)$.

# COMPLEXIDADES

## $O(n)$

- Intercala: $\Theta(n)$
- Particione: $\Theta(n)$
- Select aleatorizado: $\Theta(n)$
- BucketSort: $O(n)$

## $O(n \lg n)$

- MergeSort: $\Theta(n \lg n)$
- Quicksort aleatorizado: $\Theta(n \log n)$

## $O(n^2)$

- Quicksort: $O(n^2)$
- Selection Sort: $O(n^2)$
- Insertion Sort: $O(n^2)$

## outros

- Counting Sort: $\Theta(K+n)$
- RadixSort: $\Theta(d(K+n))$

aula 1

# ORDENAÇÃO

- Ordenação por inserção

Algoritmo rearranja $A[1\,..\,n]$ em ordem crescente

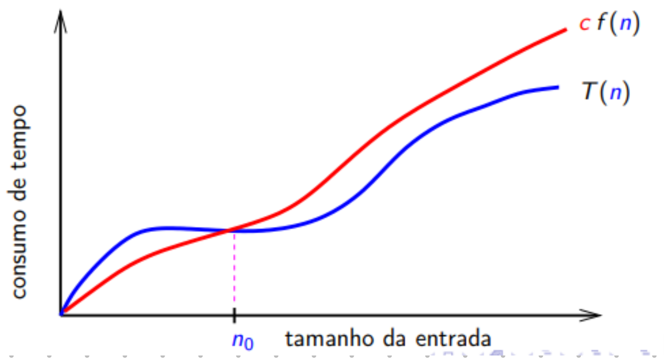
```
ORDENA-POR-INSERÇÃO (A, n)
j ← 2
enquanto j ≤ n faça
    chave ← A[j]

    i ← j − 1
    enquanto i ≥ 1 e A[i] > chave faça
        A[i + 1] ← A[i]      ▷ desloca
        i ← i − 1

    A[i + 1] ← chave      ▷ insere
    j ← j + 1
```

# NOTAÇÃO O

- $O(f(n)) \approx$ funções que não crescem mais rápido que $f(n)$

- Sejam $T(n)$ e $f(n)$ funções dos inteiros nos reais
- dizemos que $T(n)$ é $O(f(n))$ se existem constantes positivas $c$ e $n_0$ tais que

$$T(n) \leq c\, f(n) \quad \text{para todo } n \geq n_0$$




**Exemplo 1**

$10n^2$ é $\mathrm{O}(n^3)$.

Prova: Para $n \geq 0$, temos que $0 \leq 10n^2 \leq 10\,n^3$.

**Exemplo 2**

$\lg n$ é $\mathrm{O}(n)$.

Prova: Para $n \geq 1$, tem-se que $\lg n \leq 1\,n$.

**Exemplo 3**

$20n^3 + 10n\log n + 5$ é $\mathrm{O}(n^3)$.

Prova: Para $n \geq 1$, tem-se que

$$20n^3 + 10n\lg n + 5 \leq 20n^3 + 10n^3 + 5n^3 = 35\,n^3.$$

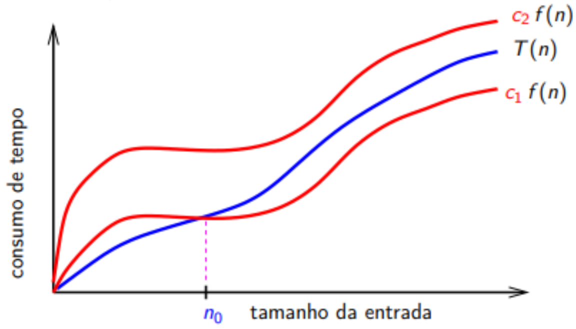
| classe | nome |
|---|---|
| $\mathrm{O}(1)$ | constante |
| $\mathrm{O}(\lg n)$ | logarítmica |
| $\mathrm{O}(n)$ | linear |
| $\mathrm{O}(n\lg n)$ | $n\log n$ |
| $\mathrm{O}(n^2)$ | quadrática |
| $\mathrm{O}(n^3)$ | cúbica |
| $\mathrm{O}(n^k)$ com $k \geq 1$ | polinomial |
| $\mathrm{O}(2^n)$ | exponencial |
| $\mathrm{O}(a^n)$ com $a > 1$ | exponencial |

# NOTAÇÃO THETA Θ

Dizemos que $T(n)$ é $\Theta(f(n))$ se
existem constantes positivas $c_1, c_2$ e $n_0$ tais que

$$c_1\, f(n) \;\leq\; T(n) \;\leq\; c_2\, f(n)$$

para todo $n \geq n_0$.




# COMPARAÇÕES

| comparação | comparação assintótica |
|---|---|
| $T(n) \leq f(n)$ | $T(n)$ é $O(f(n))$ |
| $T(n) \geq f(n)$ | $T(n)$ é $\Omega(f(n))$ |
| $T(n) = f(n)$ | $T(n)$ é $\Theta(f(n))$ |

aula 3

# DIVISÃO E CONQUISTA

- divisão: dividir a instância do problema em instâncias menores do problema.
- conquista: resolver o problema nas instâncias menores recursivamente (ou diretamente, se elas forem pequenas o suficiente.
- combinação: combinar as soluções das instâncias menores para gerar uma solução da instância original.

## MERGESORT

Rearranja $A[p \cdots r]$, com $p \leq r$, em ordem crescente. Divisão e conquista.

```
MERGESORT (A, p, r)
se p < r
    então q ← ⌊(p + r)/2⌋
          MERGESORT (A, p, q)
          MERGESORT (A, q + 1, r)
          INTERCALA (A, p, q, r)
```

## INTERCALAÇÃO

Dados $A[p \cdots q]$ e $A[q+1 \cdots r]$ crescentes, rearranjar $A[p \cdots r]$ de modo que ele fique em ordem crescente.

- intercala dois vetores que estão em ordem crescente

instável:

```
INTERCALA (A, p, q, r)
▷ B[p . . r] é um vetor auxiliar
para i ← p até q faça
    B[i] ← A[i]
para j ← q + 1 até r faça
    B[r + q + 1 − j] ← A[j]      inverte o vetor azul
i ← p
j ← r
para k ← p até r faça
    se B[i] ≤ B[j]
        então A[k] ← B[i]
              i ← i + 1
        senão A[k] ← B[j]
              j ← j − 1
```

estável:

```
INTERCALA (A, p, q, r)
▷ B[p . . r] é um vetor auxiliar
para i ← p até q faça
    B[i] ← A[i]
para j ← q + 1 até r faça
    B[r + q + 1 − j] ← A[j]
i ← p
j ← r
para k ← p até r faça
    se B[i] ≤ B[j] e i ≤ q          ▷ alteração
        então A[k] ← B[i]
              i ← i + 1
        senão A[k] ← B[j]
              j ← j − 1
```

aula 3

# RESOLUÇÃO DE RECORRÊNCIAS

- simplificar série

quantidade de elementos da PG

$$S_n = \frac{a_1(q^n - 1)}{q - 1}$$

soma dos **n** termos da P.G.

primeiro termo da sequência

razão

- verificação:
  - pegar o T(n) original e depois colocar o T(n) encontrado no original
  - tem que resultar no T(n) encontrado

## PARTICIONE

Rearranja $A[p \dots r]$ de modo que $p \leq q \leq r$ e $A[p \dots q-1] \leq A[q] < A[q+1 \dots r]$

- particiona o vetor de modo que o último elemento seja o pivô e o vetor é reordenado de modo que esquerda - menores que o pivô e direita - maiores que o pivô

```
PARTICIONE (A, p, r)
x ← A[r]        ▷ x é o "pivô"
i ← p−1
para j ← p até r − 1 faça
    se A[j] ≤ x
        então i ← i + 1
              A[i] ↔ A[j]
A[i+1] ↔ A[r]
devolva i + 1
```

- consome tempo $\Theta(n)$
- o Quicksort usa o particiona

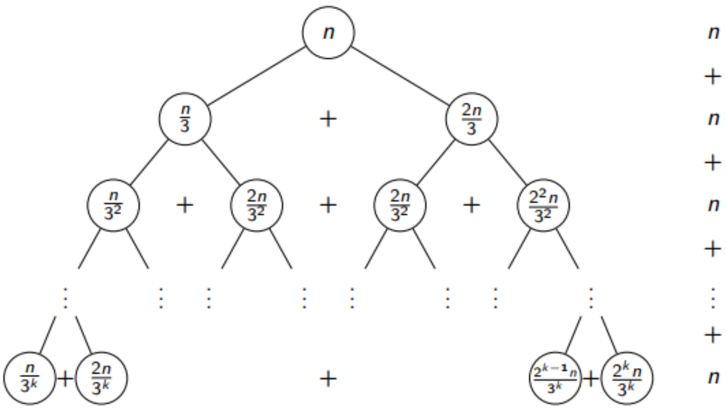
## QUICKSORT

- Rearranja $A[p \dots r]$ em ordem crescente

```
QUICKSORT (A, p, r)
se p < r
    então q ← PARTICIONE (A, p, r)
          QUICKSORT (A, p, q − 1)
          QUICKSORT (A, q + 1, r)
```

- consumo de tempo no pior caso é $\Theta(n)$
- consumo de tempo no melhor caso é $\Theta(n \lg n)$
- apesar de no pior caso ser $\Theta(n)$, o quicksort é comparável e até melhor que algoritmos que no pior caso tem tempo $O(n \lg n)$

# ÁRVORE DE RECORRÊNCIA




Os níveis da esquerda chegarão antes na base, ou seja, a árvore será inclinada para a direita.

# ESPERANÇA

- suponha que $A[1 \dots n]$ é permutação aleatória uniforme de $1 \dots n$
  - cada permutação tem probabilidade $\frac{1}{n!}$
- $X$ = nº total de execuções

$$X_i = \begin{cases} 1 & \text{se é executado} \\ 0 & \text{caso contrário} \end{cases}$$

$$X = X_1 + \dots + X_n$$

- $E[X_i]$ = probabilidade de que $A[i]$ em $A[1 \dots i]$

$$= \frac{1}{i}$$

$$E[X] = E[X_1] + \dots + E[X_n]$$

- $\frac{1}{1} + \dots + \frac{1}{n} < 1 + \ln n = O(\lg n)$

aula 6

Algoritmos que não usam um pivô "padrão" e sim um aleatório, isso aumenta as chances de ter um pivô bom e o tempo consumido diminui

## QUICKSORT ALEATORIZADO

```
PARTICIONE-ALEA(A, p, r)
   i ← RANDOM (p, r)
   A[i] ↔ A[r]
   devolva PARTICIONE (A, p, r)
```

```
QUICKSORT-ALE (A, p, r)
   se p < r
      então q ← PARTICIONE-ALEA (A, p, r)
            QUICKSORT-ALE (A, p, q − 1)
            QUICKSORT-ALE (A, q + 1, r)
```

consumo de tempo: $\Theta(n \log n)$

## SELECT ALEATORIZADOS

```
PARTICIONE-ALEA(A, p, r)
   k ← RANDOM (p, r)
   A[k] ↔ A[r]
   devolva PARTICIONE (A, p, r)
```

```
SELECT-ALEA (A, p, r, k)
   se p = r então devolva A[p]
   q ← PARTICIONE-ALEA (A, p, r)
   se k = q − p + 1
      então devolva A[q]
   se k < q − p + 1
      então devolva SELECT-ALEA (A, p, q − 1, k)
      senão devolva SELECT-ALEA (A, q+1, r, k−(q−p+1))
```

consumo de tempo: $\Theta(n)$

- Consumo de tempo esperado do select aleatorizado para um $k$ arbitrário

Suponha $A[p \dots r]$ permutação de $1 \dots n$

$X_{ab}$ = número de comparações

• • •

# LIMITE INFERIOR

Considere uma árvore de decisão para $A[1 \ldots n]$

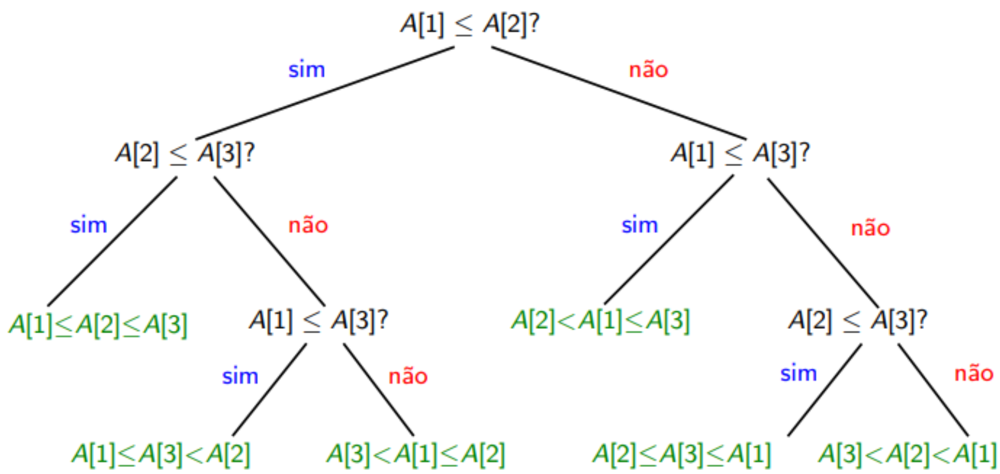
- número de comparações no pior caso: altura $h$ da árvore
- todas as $n!$ permutações de $1, \ldots, n$ devem ser folhas
- o número de folhas de uma árvore de altura $h$ é $2^h$
- assim, devemos ter $2^h \geq n!$
- donde $h \geq \lg(n!)$
- portanto $h \geq \lg(n!) \geq \frac{1}{2} n \lg n$

Portanto, todo algoritmo de ordenação baseado em comparações faz $\Omega(n \lg n)$ comparações no pior caso.

## SELECTION SORT

- a ordenação por seleção melhora o bubble sort
- realiza apenas uma troca a cada passagem pela lista
- procura pelo valor mais alto enquanto faz uma passagem
- depois de completá-la, coloca-o na posição certa
- complexidade $O(n^2)$

ORDENA-POR-INSERÇÃO ($A[1..3]$):




## ÁRVORE DE DECISÕES

- Qualquer algoritmo baseado em comparações é uma árvore de decisões

- não existem algoritmos baseados em comparação que têm complexidade melhor que $O(n \lg n)$

- Considere uma árvore de decisão para $A[1 \dots n]$

  A menor profundidade significa o número de comparações no pior caso, que é a altura $h$ da árvore de decisão, ou seja, a menor profundidade é $n \lg n$.

aula 7

# COUNTING SORT

- Recebe inteiros n e K, e um vetor A[1 ... n] onde cada elemento é um inteiro entre 1 e K.
- Devolve um vetor B[1 ... n] com os elementos de A[1 ... n] em ordem crescente

```
CountingSort(A, n)
para i ← 1 até k faça
    C[i] ← 0
para j ← 1 até n faça
    C[A[j]] ← C[A[j]] + 1
para i ← 2 até k faça
    C[i] ← C[i] + C[i − 1]
para j ← n decrescendo até 1 faça
    B[C[A[j]]] ← A[j]
    C[A[j]] ← C[A[j]] − 1
devolva B
```

- inicia vetor C com zeros
- coloca a quantidade de inteiros de A em C de acordo com os índices de C
- faz a soma acumulada em C
- percorre o vetor A ao contrário para ser estável
- para cada inteiro em A, verifica em qual posição deve ficar em C e coloca em B, e decresce o valor de C.

Complexidade : $\Theta(K + n)$

## RADIXSORT

- inteiros não-negativos com d digitos
- cartões perfurados
- registros cuja chave tem vários campos
- ordena $A[1 \dots n]$ pelo i-ésimo digito dos números em A por meio de um algoritmo estável

consumo de tempo $\Theta(d(K+n))$

## BUCKET SORT

Recebe um inteiro n e um vetor $A[1 \dots n]$ onde cada elemento é um número no intervalo $[0, 1)$

- consumo de tempo esperado é $O(n)$
- é ordenado com o insertion sort, pois o esperado é que sejam listas pequenas
- caso queira que o algoritmo seja melhor pode-se usar o mergesort

| | | | | | | | | | | n |
|---|---|---|---|---|---|---|---|---|---|---|
| $A$ | .47 | .93 | .82 | .12 | .42 | .03 | .62 | .38 | .77 | .91 |

BucketSort$(A, n)$
**para** $i \leftarrow 0$ **até** $n-1$ **faça**
    $B[i] \leftarrow$ NIL
**para** $i \leftarrow 1$ **até** $n$ **faça**
    Insira$(B[\lfloor n\, A[i] \rfloor], A[i])$
**para** $i \leftarrow 0$ **até** $n-1$ **faça**
    OrdeneLista$(B[i])$
$C \leftarrow$ Concatene$(B, n)$
**devolva** $C$

Insira$(p, x)$: insere $x$ na lista apontada por $p$

OrdeneLista$(p)$: ordena a lista apontada por $p$

Concatene$(B, n)$: devolve a lista obtida da concatenação das listas apontadas por $B[0], \dots, B[n-1]$.

# MULTIPLICAÇÃO DE NÚMEROS GIGANTESCOS

## Algoritmo de Multi-DC

Algoritmo recebe inteiros $X[1\,..\,n]$ e $Y[1\,..\,n]$ e devolve $X \cdot Y$.

MULT $(X, Y, n)$
- **se** $n = 1$ **devolva** $X \cdot Y$
- $q \leftarrow \lceil n/2 \rceil$
- $A \leftarrow X[q+1\,..\,n]$ $\quad B \leftarrow X[1\,..\,q]$
- $C \leftarrow Y[q+1\,..\,n]$ $\quad D \leftarrow Y[1\,..\,q]$
- $E \leftarrow$ MULT$(A, C, \lfloor n/2 \rfloor)$
- $F \leftarrow$ MULT$(B, D, \lceil n/2 \rceil)$
- $G \leftarrow$ MULT$(A, D, \lceil n/2 \rceil)$
- $H \leftarrow$ MULT$(B, C, \lceil n/2 \rceil)$
- $R \leftarrow E \times 10^n + (G + H) \times 10^{\lceil n/2 \rceil} + F$
- **devolva** $R$

consumo de tempo: $\Theta(n^2)$

# KARATSUBA

## Algoritmo Multi

Algoritmo recebe inteiros $X[1\,..\,n]$ e $Y[1\,..\,n]$
e devolve $X \cdot Y$ (Karatsuba e Ofman).

KARATSUBA $(X, Y, n)$
- **se** $n \leq 3$ **devolva** $X \cdot Y$
- $q \leftarrow \lceil n/2 \rceil$
- $A \leftarrow X[q+1\,..\,n]$ $\quad B \leftarrow X[1\,..\,q]$
- $C \leftarrow Y[q+1\,..\,n]$ $\quad D \leftarrow Y[1\,..\,q]$
- $E \leftarrow$ KARATSUBA$(A, C, \lfloor n/2 \rfloor)$
- $F \leftarrow$ KARATSUBA$(B, D, \lceil n/2 \rceil)$
- $G \leftarrow$ KARATSUBA$(A + B, C + D, \lceil n/2 \rceil + 1)$
- $H \leftarrow G - F - E$
- $R \leftarrow E \times 10^n + H \times 10^{\lceil n/2 \rceil} + F$
- **devolva** $R$

$T(n)$ = consumo de tempo do algoritmo
para multiplicar dois inteiros com $n$ algarismos.

consumo de tempo = $\Theta(n^{\lg 3})$