MC202 - Estruturas de Dados

Guilherme P. Telles

IC

18 de Setembro de 2019

Avisos

- Estes slides contêm erros.
- Estes slides são incompletos.
- Estes slides são escritos usando português anterior à reforma ortográfica de 2009.

Parte I

Estruturas de dados

Estruturas de dados

- Uma estrutura de dados é uma forma de organizar registros $R_1, R_2, \ldots, R_n$ na memória para tornar a manipulação desses dados eficiente.
- Cada registro tem um identificador único (chave) e outros dados de interesse da aplicação (dados satélites).
- A chave tipicamente é um número inteiro. Se não for, isso não muda as estruturas de dados mas pode mudar o desempenho delas.

- Exemplos são
  - registros de pessoas com chave CPF,
  - registros de veículos com chave Renavan,
  - registros de estudantes da U com chave RA,
  - registros de produtos em uma loja com chave código-de-barras,
  - etc.

- Quando pensamos a respeito de estruturas de dados temos que levar em conta
  - A composição dos dados e a forma como mudam.
  - O tipo e freqüência das operações realizadas sobre os dados.
  - A eficiência em tempo e memória.

Visões

- Vamos alternar entre duas visões: a de quem usa (especificação) e a de quem constrói (implementação).
- Para a especificação vamos usar o conceito de *tipo abstrato de dados*:
  - Um TAD especifica um tipo de dados em termos dos valores que ele pode assumir e da semântica das operações que podem modificá-lo, independentemente de qualquer implementação.
  - Nossa especificação de TADs vai ser informal ou pouco rigorosa.
- Para a implementação precisamos levar em conta as particularidades da linguagem de programação e do hardware onde a estrutura de dados vai ser processada.

Roteiro

- Análise de algoritmos
- Formas de organizar memória
  - Arrays
  - Listas encadeadas
- Estruturas de dados básicas
- Recursão
- Outra forma de organizar memória: árvores
- O problema da busca e EDs relacionadas
- Filas de prioridades
- Ordenação
- Grafos

Análise assintótica de algoritmos

Analisar um algoritmo

- Podemos analisar um algoritmo para
  - determinar se ele é correto (sempre pára com o resultado esperado, MC458).
  - determinar os recursos que ele usa (tempo, memória, comunicação etc, um pouco aqui, muito mais em MC458).

Análise assintótica

- É uma avaliação teórica que fornece uma aproximação do desempenho de um algoritmo, sem implementar e executar experimentos.
- Normalmente é mais fácil que a análise experimental.
- A análise assintótica é feita contando-se o número de operações que o algoritmo executa e expressando o resultado como uma função do tamanho da entrada e supondo que o tamanho da entrada tende ao infinito.

- O algoritmo abaixo soma os elementos de um vetor.
- O tamanho da entrada é n, o número de elementos do vetor.

```
SUM(A[1 .. n])
sum = 0
for i = 1 to n
        sum = sum + A[i]
return sum
```

- Contamos todas as instruções sem diferenciá-las:

$\textsc{Sum}(A[1\,..\,n])$	operações
$sum = 0$	
for $i = 1$ **to** n	
$\quad sum = sum + A[i]$	
return sum	

- Contamos todas as instruções sem diferenciá-las:

$\textsc{Sum}(A[1\,..\,n])$	operações
$sum = 0$	1
for $i = 1$ **to** n	
$\quad sum = sum + A[i]$	
return sum	

- Contamos todas as instruções sem diferenciá-las:

$\textsc{Sum}(A[1\,..\,n])$	operações
$sum = 0$	1
for $i = 1$ **to** n	$n + 1$
$\quad sum = sum + A[i]$	
return sum	

- Contamos todas as instruções sem diferenciá-las:

$\textsc{Sum}(A[1\,..\,n])$	operações
$sum = 0$	1
for $i = 1$ **to** n	$n + 1$
$sum = sum + A[i]$	n
return sum	

- Contamos todas as instruções sem diferenciá-las:

$\textsc{Sum}(A[1\,..\,n])$	operações
$sum = 0$	1
for $i = 1$ **to** n	$n + 1$
$\quad sum = sum + A[i]$	n
return sum	1

- Contamos todas as instruções sem diferenciá-las:

$\textsc{Sum}(A[1\,..\,n])$	operações
$sum = 0$	1
for $i = 1$ **to** n	$n+1$
$\quad sum = sum + A[i]$	n
return sum	1

- Somando temos: $1 + (n+1) + n + 1 = 2n + 3$.

- Contamos todas as instruções sem diferenciá-las:

$\textsc{Sum}(A[1\,..\,n])$	operações
$sum = 0$	1
for $i = 1$ **to** n	$n + 1$
$\quad sum = sum + A[i]$	n
return sum	1

- Somando temos: $1 + (n + 1) + n + 1 = 2n + 3$.
- Descartamos os coeficientes e termos de menor grau.

- Contamos todas as instruções sem diferenciá-las:

$\textsc{Sum}(A[1\,..\,n])$	operações
$sum = 0$	1
for $i = 1$ **to** n	$n + 1$
$\quad sum = sum + A[i]$	n
return sum	1

- Somando temos: $1 + (n+1) + n + 1 = 2n + 3$.
- Descartamos os coeficientes e termos de menor grau.
- Dizemos que o tempo de execução de $\textsc{Sum}$ é proporcional a n.

- Contamos todas as instruções sem diferenciá-las:

$\text{SUM}(A[1\,..\,n])$	operações
$sum = 0$	1
for $i = 1$ **to** n	$n + 1$
$\quad sum = sum + A[i]$	n
return sum	1

- Somando temos: $1 + (n + 1) + n + 1 = 2n + 3$.
- Descartamos os coeficientes e termos de menor grau.
- Dizemos que o tempo de execução de SUM é proporcional a n.
- Usando a notação $O(n)$ para representar esse fato, dizemos que o *tempo de execução* de SUM é $O(n)$.

Notação O

- Uma função $f(n)$ pertence ao conjunto de funções $O(g(n))$ se

$$0 \leq f(n) \leq cg(n)$$

para constantes positivas c e n_0 e para todo $n \geq n_0$.

Notação O

- Uma função $f(n)$ pertence ao conjunto de funções $O(g(n))$ se

  $$0 \leq f(n) \leq cg(n)$$

  para constantes positivas c e n_0 e para todo $n \geq n_0$.

- Intuitivamente, se $f(n) \in O(g(n))$ então
  - ▷ $f \leq g$
  - ▷ f é limitada superiormente por g
  - ▷ $O(g(n))$ contém as funções limitadas superiormente por $g(n)$

Notação O

- Uma função $f(n)$ pertence ao conjunto de funções $O(g(n))$ se

$$0 \leq f(n) \leq cg(n)$$

  para constantes positivas c e n_0 e para todo $n \geq n_0$.

- Intuitivamente, se $f(n) \in O(g(n))$ então
  - $\triangleright$ $f \leq g$
  - $\triangleright$ f é limitada superiormente por g
  - $\triangleright$ $O(g(n))$ contém as funções limitadas superiormente por $g(n)$

- Como alternativas a pertence, também dizemos

  $f(n)$ é $O(g(n))$

  $f(n) = O(g(n))$

- $3n^3 + 10n^2 + 100 = O(n^3)$

- $3n^3 + 10n^2 + 100 = O(n^3)$
- $5n^2 + 2n = O(n^3)$

- $3n^3 + 10n^2 + 100 = O(n^3)$
- $5n^2 + 2n = O(n^3)$
- $3n\sqrt{n} + 5n = O(n^3)$

- $3n^3 + 10n^2 + 100 = O(n^3)$
- $5n^2 + 2n = O(n^3)$
- $3n\sqrt{n} + 5n = O(n^3)$
- $2n \log n + 1 = O(n^3)$

- $3n^3 + 10n^2 + 100 = O(n^3)$
- $5n^2 + 2n = O(n^3)$
- $3n\sqrt{n} + 5n = O(n^3)$
- $2n \log n + 1 = O(n^3)$
- $30n + 2000 = O(n^3)$

- $3n^3 + 10n^2 + 100 = O(n^3)$
- $5n^2 + 2n = O(n^3)$
- $3n\sqrt{n} + 5n = O(n^3)$
- $2n \log n + 1 = O(n^3)$
- $30n + 2000 = O(n^3)$
- $51 = O(n^3)$

- $3n^3 + 10n^2 + 100 = O(n^3)$
- $5n^2 + 2n = O(n^3)$
- $3n\sqrt{n} + 5n = O(n^3)$
- $2n \log n + 1 = O(n^3)$
- $30n + 2000 = O(n^3)$
- $51 = O(n^3)$
- $n^3 \log n \neq O(n^3)$

Tamanho da entrada

- Formalmente o tamanho da entrada é o número de bits necessários para representá-la.

Tamanho da entrada

- Formalmente o tamanho da entrada é o número de bits necessários para representá-la.
- Quase sempre simplificamos a descrição do tamanho da entrada de acordo com o problema que o algoritmo resolve.

Tamanho da entrada

- Formalmente o tamanho da entrada é o número de bits necessários para representá-la.
- Quase sempre simplificamos a descrição do tamanho da entrada de acordo com o problema que o algoritmo resolve.
- Se no exemplo da soma os números têm 32 bits, o tamanho da entrada é $32n$.

Tamanho da entrada

- Formalmente o tamanho da entrada é o número de bits necessários para representá-la.
- Quase sempre simplificamos a descrição do tamanho da entrada de acordo com o problema que o algoritmo resolve.
- Se no exemplo da soma os números têm 32 bits, o tamanho da entrada é $32n$.
- A constante 32 vai ser diluída nas outras simplificações:

$$2n + 3 \propto 2(32n) + 3 \propto 64n + 3 \propto n$$

Tamanho da entrada

- Formalmente o tamanho da entrada é o número de bits necessários para representá-la.
- Quase sempre simplificamos a descrição do tamanho da entrada de acordo com o problema que o algoritmo resolve.
- Se no exemplo da soma os números têm 32 bits, o tamanho da entrada é $32n$.
- A constante 32 vai ser diluída nas outras simplificações:

$$2n + 3 \propto 2(32n) + 3 \propto 64n + 3 \propto n$$

- Então dizemos que o tamanho da entrada para o problema da soma é n e não $32n$.

Não são simplificações demais?

- Quando n tende ao infinito, os termos de menor grau são dominados pelo de maior grau.

Não são simplificações demais?

- Quando n tende ao infinito, os termos de menor grau são dominados pelo de maior grau.
- As constantes são muito influenciadas pela implementação e pelo computador que vai executar o programa.

Não são simplificações demais?

- Quando n tende ao infinito, os termos de menor grau são dominados pelo de maior grau.
- As constantes são muito influenciadas pela implementação e pelo computador que vai executar o programa.
- Então todas essas simplificações são razoáveis para fornecer uma aproximação do desempenho do algoritmo.

Não são simplificações demais?

- Quando n tende ao infinito, os termos de menor grau são dominados pelo de maior grau.
- As constantes são muito influenciadas pela implementação e pelo computador que vai executar o programa.
- Então todas essas simplificações são razoáveis para fornecer uma aproximação do desempenho do algoritmo.
- E funciona bem na prática.

- O algoritmo abaixo recebe um vetor em que $A[1\,..\,n-1])$ está ordenado e posiciona $A[n]$ de forma que $A[1\,..\,n]$ fique ordenado.

INSERT$(A[1\,..\,n])$

```
key = A[n]
i = n − 1
while i > 0 and A[i] > key
        A[i + 1] = A[i]
        i = i − 1
A[i + 1] = key
```

- Freqüentemente o tempo de execução de um algoritmo depende não só do tamanho da entrada, mas também do valor dela.
- Esse é o caso do INSERT.
- Nesses casos temos dois tipos de análises: de pios caso e de caso médio.

Análise de pior caso

- Na análise de pior caso, o tempo de execução de um algoritmo é o número máximo de instruções que ele pode executar dentre todas as instâncias válidas (ainda em função do tamanho da entrada).
- É boa por fornecer um limite superior para o tempo de execução do algoritmo.
- Pode ser muito pessimista.

- Para INSERT o pior caso é quando $A[n] < A[1 \,..\, n-1]$.

```
INSERT(A[1..n])
key = A[n]
i = n - 1
while i > 0 and A[i] > key
        A[i + 1] = A[i]
        i = i - 1
A[i + 1] = key
```

- Para INSERT o pior caso é quando $A[n] < A[1 \mathinner{\ldotp\ldotp} n-1]$.

INSERT$(A[1 \mathinner{\ldotp\ldotp} n])$ — operações

$key = A[n]$
$i = n - 1$
while $i > 0$ and $A[i] > key$
    $A[i+1] = A[i]$
    $i = i - 1$
$A[i+1] = key$

- Para INSERT o pior caso é quando $A[n] < A[1\,..\,n-1]$.

$\textsc{Insert}(A[1\,..\,n])$	operações
$key = A[n]$	1
$i = n - 1$	
while $i > 0$ and $A[i] > key$	
$\quad A[i+1] = A[i]$	
$\quad i = i - 1$	
$A[i+1] = key$	

- Para INSERT o pior caso é quando $A[n] < A[1\,..\,n-1]$.

INSERT($A[1\,..\,n]$)	operações
$key = A[n]$	1
$i = n - 1$	1
while $i > 0$ and $A[i] > key$	
$A[i+1] = A[i]$	
$i = i - 1$	
$A[i+1] = key$	

- Para INSERT o pior caso é quando $A[n] < A[1\,..\,n-1]$.

INSERT$(A[1\,..\,n])$	operações
$key = A[n]$	1
$i = n - 1$	1
while $i > 0$ and $A[i] > key$	n
$\quad A[i+1] = A[i]$	
$\quad i = i - 1$	
$A[i+1] = key$	

- Para INSERT o pior caso é quando $A[n] < A[1\,..\,n-1]$.

INSERT$(A[1\,..\,n])$	operações
$key = A[n]$	1
$i = n - 1$	1
while $i > 0$ and $A[i] > key$	n
$A[i+1] = A[i]$	$n - 1$
$i = i - 1$	
$A[i+1] = key$	

- Para INSERT o pior caso é quando $A[n] < A[1\,..\,n-1]$.

$\textsc{Insert}(A[1\,..\,n])$	operações
$key = A[n]$	1
$i = n - 1$	1
while $i > 0$ and $A[i] > key$	n
$\quad A[i+1] = A[i]$	$n - 1$
$\quad i = i - 1$	$n - 1$
$A[i+1] = key$	

- Para INSERT o pior caso é quando $A[n] < A[1\,..\,n-1]$.

INSERT($A[1\,..\,n]$)	operações
$key = A[n]$	1
$i = n - 1$	1
while $i > 0$ and $A[i] > key$	n
$A[i+1] = A[i]$	$n - 1$
$i = i - 1$	$n - 1$
$A[i+1] = key$	1

- Para INSERT o pior caso é quando $A[n] < A[1\,..\,n-1]$.

INSERT$(A[1\,..\,n])$	operações
$key = A[n]$	1
$i = n - 1$	1
while $i > 0$ and $A[i] > key$	n
$A[i+1] = A[i]$	$n - 1$
$i = i - 1$	$n - 1$
$A[i+1] = key$	1

- Somando temos $3n + 1 = O(n)$.

- Para INSERT o pior caso é quando $A[n] < A[1\,..\,n-1]$.

INSERT($A[1\,..\,n]$)	operações
$key = A[n]$	1
$i = n - 1$	1
while $i > 0$ and $A[i] > key$	n
$A[i+1] = A[i]$	$n-1$
$i = i - 1$	$n-1$
$A[i+1] = key$	1

- Somando temos $3n + 1 = O(n)$.
- INSERT é $O(n)$ no pior caso.

Análise de caso médio

- Na análise de caso médio computamos a média do tempo de execução para todas as instâncias de um certo tamanho considerando a distribuição de probabilidades para as instâncias desse tamanho.
- Boa por fornecer uma idéia do comportamento esperado de um algoritmo.
- Normalmente é mais difícil de fazer.

- Supondo que $A[n]$ pode ocupar qualquer posição entre 1 e n com a mesma probabilidade, então o número médio de execuções do while de INSERT é

$$\frac{1+2+\ldots+n}{n} = \frac{n(n+1)}{2n} = \frac{n+1}{2} = O(n)$$

- INSERT é $O(n)$ no caso médio.

Análise de melhor caso

- É inútil.

Partition

- Partition recebe um vetor em que $A[1\,..\,n]$ e reposiciona os elementos em torno de $A[1]$: os menores ou iguais a $A[1]$ à esquerda e os maiores ou iguais a $A[1]$ à direita.

```
PARTITION(A[1..n])
p = A[1]
i = 2
j = n
while true
    while A[i] < p and i ≤ n
        i++
    while A[j] > p
        j--
    if j ≤ i
        break
    else
        SWAP(A[i], A[j])
        i++
        j--
SWAP(A[j], A[1])
return j
```

- O número total de vezes que o while-5 e o while-7 são executados é sempre o mesmo (mais ou menos 1).
- Um pior caso é quando quando $A[2\,..\,n]$ tem tamanho par, está em ordem decrescente e $A[1]$ é mediano em $A[2\,..\,n]$. Nesse caso o número de trocas é máximo.

PARTITION($A[1\,..\,n]$)

```
p = A[1]
i = 2
j = n
while true
    while A[i] < p and i ≤ n
        i++
    while A[j] > p
        j--
    if j ≤ i
        break
    else
        SWAP(A[i], A[j])
        i++
        j--
SWAP(A[j], A[1])
return j
```

PARTITION($A[1\,..\,n]$) | operações

```
p = A[1]
i = 2
j = n
while true
    while A[i] < p and i ≤ n
        i++
    while A[j] > p
        j--
    if j ≤ i
        break
    else
        SWAP(A[i], A[j])
        i++
        j--
SWAP(A[j], A[1])
return j
```

```
PARTITION(A[1..n])                                operações
    p = A[1]                                          1
    i = 2
    j = n
    while true
        while A[i] < p and i ≤ n
            i++
        while A[j] > p
            j--
        if j ≤ i
            break
        else
            SWAP(A[i], A[j])
            i++
            j--
    SWAP(A[j], A[1])
    return j
```

PARTITION($A[1\,..\,n]$)	operações
$p = A[1]$	1
$i = 2$	1
$j = n$	
while true	
while $A[i] < p$ **and** $i \leq n$	
$i{+}{+}$	
while $A[j] > p$	
$j{-}{-}$	
if $j \leq i$	
break	
else	
SWAP($A[i], A[j]$)	
$i{+}{+}$	
$j{-}{-}$	
SWAP($A[j], A[1]$)	
return j	

$\textsc{Partition}(A[1\,..\,n])$	operações
$p = A[1]$	1
$i = 2$	1
$j = n$	1
while true	
while $A[i] < p$ **and** $i \leq n$	
$i++$	
while $A[j] > p$	
$j--$	
if $j \leq i$	
break	
else	
$\textsc{swap}(A[i], A[j])$	
$i++$	
$j--$	
$\textsc{swap}(A[j], A[1])$	
return j	

PARTITION$(A[1\,..\,n])$	operações
$p = A[1]$	1
$i = 2$	1
$j = n$	1
while true	$(n-1)/2 + 1$
while $A[i] < p$ **and** $i \leq n$	
$i{+}{+}$	
while $A[j] > p$	
$j{-}{-}$	
if $j \leq i$	
break	
else	
SWAP$(A[i], A[j])$	
$i{+}{+}$	
$j{-}{-}$	
SWAP$(A[j], A[1])$	
return j	

$\text{PARTITION}(A[1\,..\,n])$	operações
$p = A[1]$	1
$i = 2$	1
$j = n$	1
while true	$(n-1)/2+1$
while $A[i] < p$ **and** $i \leq n$	$(n-1)/2+1$
$i++$	
while $A[j] > p$	
$j--$	
if $j \leq i$	
break	
else	
$\text{SWAP}(A[i], A[j])$	
$i++$	
$j--$	
$\text{SWAP}(A[j], A[1])$	
return j	

```
Partition(A[1..n])                              operações
p = A[1]                                        1
i = 2                                           1
j = n                                           1
while true                                      (n − 1)/2 + 1
    while A[i] < p and i ≤ n                    (n − 1)/2 + 1
        i++                                     (n − 1)/2
    while A[j] > p
        j−−
    if j ≤ i
        break
    else
        swap(A[i], A[j])
        i++
        j−−
swap(A[j], A[1])
return j
```

PARTITION$(A[1\,..\,n])$	operações
$p = A[1]$	1
$i = 2$	1
$j = n$	1
while true	$(n-1)/2+1$
while $A[i] < p$ **and** $i \leq n$	$(n-1)/2+1$
$i++$	$(n-1)/2$
while $A[j] > p$	$(n-1)/2+1$
$j--$	
if $j \leq i$	
break	
else	
SWAP$(A[i], A[j])$	
$i++$	
$j--$	
SWAP$(A[j], A[1])$	
return j	

```
PARTITION(A[1..n])                              operações
p = A[1]                                        1
i = 2                                           1
j = n                                           1
while true                                      (n − 1)/2 + 1
    while A[i] < p and i ≤ n                    (n − 1)/2 + 1
        i++                                     (n − 1)/2
    while A[j] > p                              (n − 1)/2 + 1
        j−−                                     (n − 1)/2
    if j ≤ i
        break
    else
        SWAP(A[i], A[j])
        i++
        j−−
SWAP(A[j], A[1])
return j
```

PARTITION($A[1\,..\,n]$)	operações
$p = A[1]$	1
$i = 2$	1
$j = n$	1
while true	$(n-1)/2+1$
while $A[i] < p$ **and** $i \leq n$	$(n-1)/2+1$
$i++$	$(n-1)/2$
while $A[j] > p$	$(n-1)/2+1$
$j--$	$(n-1)/2$
if $j \leq i$	$(n-1)/2+1$
break	
else	
SWAP($A[i], A[j]$)	
$i++$	
$j--$	
SWAP($A[j], A[1]$)	
return j	

$\textsc{Partition}(A[1\,..\,n])$	operações
$p = A[1]$	1
$i = 2$	1
$j = n$	1
while true	$(n-1)/2+1$
while $A[i] < p$ **and** $i \leq n$	$(n-1)/2+1$
$i{+}{+}$	$(n-1)/2$
while $A[j] > p$	$(n-1)/2+1$
$j{-}{-}$	$(n-1)/2$
if $j \leq i$	$(n-1)/2+1$
break	0
else	
$\textsc{swap}(A[i], A[j])$	
$i{+}{+}$	
$j{-}{-}$	
$\textsc{swap}(A[j], A[1])$	
return j	

$\text{PARTITION}(A[1\,..\,n])$	operações
$p = A[1]$	1
$i = 2$	1
$j = n$	1
while true	$(n-1)/2+1$
while $A[i] < p$ **and** $i \leq n$	$(n-1)/2+1$
$i++$	$(n-1)/2$
while $A[j] > p$	$(n-1)/2+1$
$j--$	$(n-1)/2$
if $j \leq i$	$(n-1)/2+1$
break	0
else	0
$\text{SWAP}(A[i], A[j])$	
$i++$	
$j--$	
$\text{SWAP}(A[j], A[1])$	
return j	

PARTITION$(A[1\,..\,n])$	operações
$p = A[1]$	1
$i = 2$	1
$j = n$	1
while true	$(n-1)/2+1$
while $A[i] < p$ **and** $i \leq n$	$(n-1)/2+1$
$i{+}{+}$	$(n-1)/2$
while $A[j] > p$	$(n-1)/2+1$
$j{-}{-}$	$(n-1)/2$
if $j \leq i$	$(n-1)/2+1$
break	0
else	0
SWAP$(A[i], A[j])$	$(n-1)/2$
$i{+}{+}$	
$j{-}{-}$	
SWAP$(A[j], A[1])$	
return j	

$\textsc{Partition}(A[1\,..\,n])$	operações
$p = A[1]$	1
$i = 2$	1
$j = n$	1
while true	$(n-1)/2+1$
while $A[i] < p$ **and** $i \leq n$	$(n-1)/2+1$
$i++$	$(n-1)/2$
while $A[j] > p$	$(n-1)/2+1$
$j--$	$(n-1)/2$
if $j \leq i$	$(n-1)/2+1$
break	0
else	0
$\textsc{swap}(A[i], A[j])$	$(n-1)/2$
$i++$	$(n-1)/2$
$j--$	
$\textsc{swap}(A[j], A[1])$	
return j	

PARTITION($A[1\,..\,n]$)	operações
$p = A[1]$	1
$i = 2$	1
$j = n$	1
while true	$(n-1)/2+1$
while $A[i] < p$ **and** $i \leq n$	$(n-1)/2+1$
$i++$	$(n-1)/2$
while $A[j] > p$	$(n-1)/2+1$
$j--$	$(n-1)/2$
if $j \leq i$	$(n-1)/2+1$
break	0
else	0
SWAP($A[i], A[j]$)	$(n-1)/2$
$i++$	$(n-1)/2$
$j--$	$(n-1)/2$
SWAP($A[j], A[1]$)	
return j	

$\textsc{Partition}(A[1\,..\,n])$	operações
$p = A[1]$	1
$i = 2$	1
$j = n$	1
while true	$(n-1)/2+1$
while $A[i] < p$ **and** $i \leq n$	$(n-1)/2+1$
$i++$	$(n-1)/2$
while $A[j] > p$	$(n-1)/2+1$
$j--$	$(n-1)/2$
if $j \leq i$	$(n-1)/2+1$
break	0
else	0
$\textsc{swap}(A[i], A[j])$	$(n-1)/2$
$i++$	$(n-1)/2$
$j--$	$(n-1)/2$
$\textsc{swap}(A[j], A[1])$	1
return j	

```
PARTITION(A[1..n])                          operações
p = A[1]                                    1
i = 2                                       1
j = n                                       1
while true                                  (n − 1)/2 + 1
    while A[i] < p and i ≤ n                (n − 1)/2 + 1
        i++                                 (n − 1)/2
    while A[j] > p                          (n − 1)/2 + 1
        j−−                                 (n − 1)/2
    if j ≤ i                                (n − 1)/2 + 1
        break                               0
    else                                    0
        SWAP(A[i], A[j])                    (n − 1)/2
        i++                                 (n − 1)/2
        j−−                                 (n − 1)/2
SWAP(A[j], A[1])                            1
return j                                    1
```

- Somando temos menos de $4n$ operações.
- PARTITION é $O(n)$ no pior caso.

Notação Ω

- Uma função $f(n)$ pertence a $\Omega(g(n))$ se

  $$0 \leq cg(n) \leq f(n)$$

  para constantes positivas c e n_0 e para todo $n \geq n_0$.

Notação Ω

- Uma função $f(n)$ pertence a $\Omega(g(n))$ se

$$0 \leq cg(n) \leq f(n)$$

  para constantes positivas c e n_0 e para todo $n \geq n_0$.

- Intuitivamente, $f(n) \in \Omega(g(n))$ então
  - $\triangleright$ $f \geq g$
  - $\triangleright$ f é um limitada inferiormente por g

Notação Θ

- Uma função $f(n)$ pertence a $\Theta(g(n))$ se

$$0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)$$

para constantes positivas c_1, c_2 e n_0 e para todo $n \geq n_0$.

Notação Θ

- Uma função $f(n)$ pertence a $\Theta(g(n))$ se

$$0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)$$

  para constantes positivas c_1, c_2 e n_0 e para todo $n \geq n_0$.

- Intuitivamente, $f(n) \in \Theta(g(n))$ então
  - $\triangleright$ $f = g$
  - $\triangleright$ f é limitada inferior e superiormente por g

Notação o

- Uma função $f(n)$ pertence a $o(g(n))$ se

$$0 \leq f(n) < cg(n)$$

para toda constante positiva c, para uma constante positiva n_0 e para todo $n \geq n_0$.

Notação o

- Uma função $f(n)$ pertence a $o(g(n))$ se

$$0 \leq f(n) < cg(n)$$

para toda constante positiva c, para uma constante positiva n_0 e para todo $n \geq n_0$.

- Intuitivamente, $f(n) \in o(g(n))$ então

$\triangleright$ $f < g$

$\triangleright$ f é limitada superiormente com folga por g

Notação ω

- Uma função $f(n)$ pertence a $\omega(g(n))$ se

$$0 \leq cg(n) < f(n)$$

para toda constante positiva c, para uma constante positiva n_0 e para todo $n \geq n_0$.

Notação ω

- Uma função $f(n)$ pertence a $\omega(g(n))$ se

$$0 \leq cg(n) < f(n)$$

  para toda constante positiva c, para uma constante positiva n_0 e para todo $n \geq n_0$.

- Intuitivamente, $f(n) \in \omega(g(n))$ então
  - $\triangleright$ $f > g$
  - $\triangleright$ f é um limitada inferiormente com folga por g

- $5n^2 + 2n = O(n^3)$

- $5n^2 + 2n = O(n^3)$
- $5n^2 + 2n = \Omega(n \log n)$

- $5n^2 + 2n = O(n^3)$
- $5n^2 + 2n = \Omega(n \log n)$
- $5n^2 + 2n = \Theta(n^2)$

- $5n^2 + 2n = O(n^3)$
- $5n^2 + 2n = \Omega(n \log n)$
- $5n^2 + 2n = \Theta(n^2)$
- $5n^2 + 2n = o(n^3)$

- $5n^2 + 2n = O(n^3)$
- $5n^2 + 2n = \Omega(n \log n)$
- $5n^2 + 2n = \Theta(n^2)$
- $5n^2 + 2n = o(n^3)$
- $5n^2 + 2n = \omega(n)$

- $5n^2 + 2n = O(n^3)$
- $5n^2 + 2n = \Omega(n \log n)$
- $5n^2 + 2n = \Theta(n^2)$
- $5n^2 + 2n = o(n^3)$
- $5n^2 + 2n = \omega(n)$
- $5n^2 + 2n \neq o(n^2)$

- $5n^2 + 2n = O(n^3)$
- $5n^2 + 2n = \Omega(n \log n)$
- $5n^2 + 2n = \Theta(n^2)$
- $5n^2 + 2n = o(n^3)$
- $5n^2 + 2n = \omega(n)$
- $5n^2 + 2n \neq o(n^2)$
- $5n^2 + 2n \neq \omega(n^2)$

Selection-sort

```
SELECTION-SORT(A[1..n])
for i = n downto 2
    max = 1
    for j = 2 to i
        if A[j] > A[max]
            max = j
    exchange A[i] and A[max]
```

Selection-sort

```
SELECTION-SORT(A[1..n])                operações
for i = n downto 2
    max = 1
    for j = 2 to i
        if A[j] > A[max]
            max = j
    exchange A[i] and A[max]
```

Selection-sort

Selection-Sort($A[1\,..\,n]$)	operações
for $i = n$ **downto** 2	n
$max = 1$	
for $j = 2$ **to** i	
if $A[j] > A[max]$	
$max = j$	
exchange $A[i]$ and $A[max]$	

Selection-sort

SELECTION-SORT($A[1..n]$)	operações
for $i = n$ **downto** 2	n
$max = 1$	$n - 1$
for $j = 2$ **to** i	
if $A[j] > A[max]$	
$max = j$	
exchange $A[i]$ and $A[max]$	

Selection-sort

SELECTION-SORT($A[1\,..\,n]$)	operações
for $i = n$ **downto** 2	n
$max = 1$	$n - 1$
for $j = 2$ **to** i	$\sum_{j=1}^{n-1} 1$
if $A[j] > A[max]$	
$max = j$	
exchange $A[i]$ and $A[max]$	

Selection-sort

$\text{Selection-Sort}(A[1\,..\,n])$	operações
for $i = n$ **downto** 2	n
$max = 1$	$n - 1$
for $j = 2$ **to** i	$\sum_{j=1}^{n-1} 1$
if $A[j] > A[max]$	$\sum_{j=0}^{n-2} 1$
$max = j$	
exchange $A[i]$ and $A[max]$	

Selection-sort

SELECTION-SORT($A[1\,..\,n]$)	operações
for $i = n$ **downto** 2	n
$max = 1$	$n-1$
for $j = 2$ **to** i	$\sum_{j=1}^{n-1} 1$
if $A[j] > A[max]$	$\sum_{j=0}^{n-2} 1$
$max = j$	$\sum_{j=0}^{n-2} 1$
exchange $A[i]$ and $A[max]$	

Selection-sort

$\text{Selection-Sort}(A[1\,..\,n])$	operações
for $i = n$ **downto** 2	n
$\quad max = 1$	$n - 1$
$\quad$ **for** $j = 2$ **to** i	$\sum_{j=1}^{n-1} 1$
$\quad\quad$ **if** $A[j] > A[max]$	$\sum_{j=0}^{n-2} 1$
$\quad\quad\quad max = j$	$\sum_{j=0}^{n-2} 1$
$\quad$ exchange $A[i]$ and $A[max]$	$n - 1$

Selection-sort

$\textsc{Selection-Sort}(A[1\,..\,n])$	operações
for $i = n$ **downto** 2	n
$max = 1$	$n - 1$
for $j = 2$ **to** i	$\sum_{j=1}^{n-1} 1$
if $A[j] > A[max]$	$\sum_{j=0}^{n-2} 1$
$max = j$	$\sum_{j=0}^{n-2} 1$
exchange $A[i]$ and $A[max]$	$n - 1$

- O tempo de execução assintótico de $\textsc{Selection-Sort}$ não depende dos valores em A. Não faz diferença (assintoticamente) se o if é tomado ou não.
- $\textsc{Selection-Sort}$ é $\Theta(n^2)$.

```
MAXIMUM(A[1..m])
1  max = 1
2  for i = 2 to m
3      if A[i] > A[max]
4          max = i
5  return max
```

```
SELECTION-SORT(A[1..n])
1  for i = n downto 2
2      max = MAXIMUM(A[1..i])
3      exchange A[i] and A[max]
```

- Chamadas de função custam o número de operações da função.
- Essa versão de SELECTION-SORT também é $\Theta(n^2)$. A linha 3 custa $\Theta(n)$.

Memória

- Para a memória podemos usar a mesma técnica, contando o número de posições de memória usadas pelo algoritmo.
- Não contamos a memória usada pela entrada e pela saída.

Por que isso importa

ops \ n	10	20	50	100	500	1000
$10000n$	0.0001	0.0002	0.0005	0.001	0.005	0.01
$1000n \log n$	0.00003	0.00009	0.0003	0.0007	0.004	0.01
$100n^2$	0.00001	0.00004	0.0003	0.001	0.03	0.1
$10n^3$	0.00001	0.00008	0.001	0.01	1.3	10
$2n^4$	0.00002	0.0003	0.01	0.2	125	0.5 horas
$n^{\log n}$	0.000002	0.0004	4	5.4 horas	10^5 séc.	
2^n	0.000001	0.001	13 dias	10^{11} séc.		
3^n	0.00006	3	10^5 séc.			
$n!$	0.004	77 anos				

Supondo 10^9 operações de algoritmo por segundo.

Dois “modelos” de memória

- acesso aleatório: a memória consiste de posições consecutivas em que cada posição armazena um registro e seus campos. Cada registro pode ser modificado ou recuperado acessando diretamente a posição que ele ocupa na memória, em tempo constante.
  - Array
- encadeado: a memória consiste de nós. Cada nó contém um registro. Campos do registro no nó podem ser apontadores. Para modificar ou recuperar um registro, o endereço dele deve ser conhecido.
  - Listas encadeadas
  - Árvores
- Podem ser combinados em estruturas de dados.

Arrays

Array

- Um array é formado por elementos consecutivos.
- Cada elemento do array pode ser lido ou escrito fazendo apenas um acesso à memória ocupada pelo array.
- O número de elementos de um array é fixo e definido quando ele é criado.
- Um array não aumenta ou diminui de tamanho.

- Nomenclatura:
  - array, arranjo
  - array unidimensional: vetor
  - array bidimensional: matriz
- Arrays são usados para implementar muitas estruturas de dados, como seqüências, filas, pilhas, grafos etc.
- Algumas linguagens de programação não têm arrays de verdade.

Vetores dinâmicos

Vetor dinâmico

- Um vetor dinâmico é um vetor associado com um método para aumentá-lo e reduzi-lo de acordo com a ocupação.
- Bons em situações em que o vetor seria bom mas o número de registros que tem que ser armazenados não é conhecido antecipadamente.

Redimensionamento

- A política de redimensionamento tem impacto no desempenho.
- Uma forma bem estabelecida é:
  - Quando não há mais posições vazias e acontece uma inserção então o tamanho dobra.
  - Quando $3/4$ das posições estão vazias e acontece uma remoção então o tamanho é reduzido à metade.
- O tamanho mínimo pode ser 1 (todos os tamanhos serão potências de 2) ou outra constante.

- Os dados podem ter que ser movidos quando o vetor é redimensionado, o que tem custo:
  - de tempo para mover os dados.
  - de memória, já que ao redimensionar deve haver espaço para as duas cópias.

Inserção

- Antes de inserir um elemento testamos o número de posições ocupadas.
- Se o vetor estiver totalmente ocupado, criamos um vetor maior, copiamos os dados e liberamos o vetor antigo.

Remoção

- Depois de remover um elemento testamos o número de posições ocupadas.
- Se o vetor ficou pouco ocupado, criamos um vetor menor, copiamos os dados e liberamos o vetor antigo.

Desempenho

- Esse processo de ficar movendo os dados sempre que o vetor é redimensionado vale a pena?
- Sim: para cada dado inserido são realizadas menos de 4 operações de escrita no vetor.

- Para ver que isso é verdade suponha que o vetor tem tamanho $2m$ e acabou de ser redimensionado. Suponha que cada nova inserção faz uma escrita no vetor e “deposita” 2 operações de escrita em uma poupança. Quando o vetor estiver cheio teremos $2 \times 2m$ operações de escrita na poupança e usamos essa poupança para copiar os dados no novo vetor.

- Para a remoção, suponha que o vetor tem tamanho m e acabou de ser redimensionado para $m/2$. Suponha que cada uma das m inserções depositou $1/2$ operação de escrita na poupança. A parte da poupança que corresponde à metade do vetor que deixou de existir acumulou $m/4$ operações de escrita e usamos essa poupança para copiar os dados no novo vetor.

- Então a sobrecarga para verificar o tamanho do vetor e redimensionar causa apenas um aumento constante (pequeno) no número de operações por inserção no vetor, o que é bastante eficiente.
- Essas contas valem para uma política de crescimento como PG de razão 2. Para outras políticas, as contas e a conclusão sobre o desempenho podem ser diferentes.

Tabelas dinâmicas

- Essas idéias são generalizáveis para qualquer estrutura tabular.

Vetores de bits

Vetor de bits

- Há dados que têm apenas dois estados: aprovado/reprovado, marcado/não-marcado, vendido/não vendido etc.
- Para armazenar dados desse tipo precisamos no máximo de 1 bit para cada item.
- Não é comum encontrar um tipo primitivo de apenas um bit nas linguagens de programação.
- Em C, o menor inteiro tem 8 bytes. (unsigned char ou uint8_t). Em um vetor de um desses tipos, vamos usar de fato apenas $\frac{1}{8}$ do espaço ocupado.

- Podemos implementar um vetor de bits usando operadores bit-a-bit e máscaras.
- Uma máscara é uma palavra com padrão de bits bem definido.
- Para construir máscaras para selecionar um único bit usamos deslocamento e negação bit-a-bit.
- As operações básicas são set, reset e test para cada bit do vetor.

```
  01011000      01011000
| 00000010    | 00001000
----------    ----------
  01011010      01011000
```

```
   01011000      01011000
 | 00000010    | 00001000
 ----------    ----------
   01011010      01011000

   01011010      01011000
 & 11111101    & 11111101
 ----------    ----------
   01011000      01011000
```

```
   01011000        01011000
 | 00000010      | 00001000
 ----------      ----------
   01011010        01011000

   01011010        01011000
 & 11111101      & 11111101
 ----------      ----------
   01011000        01011000

   01011010        01011000
 & 00000010      & 00000010
 ----------      ----------
   00000010        00000000
```

Em C

- `bitarray-char.h`, `bitarray-char.c`

- A forma considerada “clássica” é por macros para manipular um vetor de `unsigned char`.

```
#define bit_nslots(n) (((n)>>3)+1)
#define bit_set(A,i) ((A)[(i)>>3] |= 0x80 >> ((i) & 7))
#define bit_reset(A,i) ((A)[(i)>>3] &= ~(0x80 >> ((i) & 7)))
#define bit_test(A,i) (((A)[(i)>>3] & (0x80 >> ((i) & 7))) ? 1 : 0)
```

```
unsigned char *B = calloc(bit_nslots(n),sizeof(unsigned char));
```

Listas encadeadas

Lista encadeada

- É formada por nós encadeados em seqüência.
- Cada nó guarda um registro e um dos campos é o endereço do próximo nó.
- O último nó da lista aponta para o endereço nulo.
- O primeiro nó é chamado de cabeça e o último é chamado de cauda (ou rabo).
- Também é chamada de *lista encadeada linear*, *lista ligada* ou simplesmente de *lista*.

Lista encadeada

- Listas são usadas tipicamente quando queremos guardar uma coleção de dados mas não sabemos antecipadamente que tamanho a coleção pode alcançar.
- Permitem inserções e remoções de nós em qualquer posição.
- Não permitem acesso direto a um nó.
- Listas podem ser usadas para implementar muitas estruturas de dados, como filas, pilhas, grafos etc.

Exemplos de nó

```
struct player {

  char* nick;
  char* name;
  int age;
  char gender;
  int id;
  struct player* next;

};

typedef struct player player;
```

Exemplos de nó

```
struct player {

  char* nick;
  char* name;
  int age;
  char gender;
  int id;
  struct player* next;

};

typedef struct player player;
```

```
struct node {

  int data;
  struct node* next;

};

typedef struct node node;
```

Em C

- Encontramos duas formas:
  - usando apenas um apontador para representar a lista,
  - usando um registro para representar a lista.

Representação apenas com um apontador

- A forma mais simples de representar uma lista em C é como um apontador para a cabeça da lista.
- Uma lista vazia é um apontador nulo.
- O apontador para a cabeça da lista é passado (por referência) para funções que manipulam a lista.

Representação com registro

- Nessa forma definimos de um registro (e um tipo) para a lista.
- Esse registro guarda pleo menos um apontador para a cabeça da lista. Ele também pode guardar outras informações a respeito da lista, como um apontador para o rabo, o tamanho da lista etc.
- A lista vazia é uma lista com a cabeça nula.

Nó sentinela

- Um nó sentinela (ou dummy) é um nó adicionado à lista para marcar alguma posição e sinalizar alguma condição na lista.

Sentinela no início

- Nesse caso qualquer lista contém pelo menos um nó.
- Permite escrever programas mais homogêneos, que não precisam tratar o caso de lista vazia.

Estrutura recursiva

- Uma lista encadeada é uma estrutura recursiva: cada nó tem uma referência para uma estrutura menor e do mesmo tipo.
- Logo, funções recursivas podem ser usadas em listas.

Lista encadeada circular

- Na lista circular o rabo aponta para a cabeça.
- O apontador para o início da lista aponta para o rabo.

Lista duplamente encadeada

- Na lista duplamente encadeada cada nó aponta para seu sucessor e para seu predecessor.
- Normalmente guardamos um apontador para a cabeça um apontador para o rabo da lista.
- Usa mais memória por nó.
- Há mais flexibilidade para navegar na lista e para modificá-la.

Combinações e modificações

- Combinações e modificações dos tipos de listas anteriores são possíveis.
- Por exemplo, lista duplamente encadeada com sentinela, lista duplamente encadeada circular etc.

Lista exógena

- As listas que consideramos até aqui são endógenas: cada nó armazena os dados e o(s) apontador(es) que encadeiam.
- Na lista exógena, cada nó armazena o(s) apontador(es) que encadeiam e um apontador para os dados. Isto é, os dados estão fora da lista.
- Boa quando há muitas repetições dos elementos da lista.