Algoritmo Quicksort

Estrutura de Dados — QXD0010

UNIVERSIDADE FEDERAL DO CEARÁ
CAMPUS QUIXADÁ

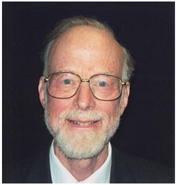
Prof. Roberto Cabral
rbcabral@ufc.br

Universidade Federal do Ceará

2º semestre/2022

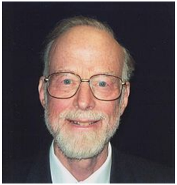
Introdução

- Algoritmo proposto por C. A. R. Hoare em 1960.



Introdução

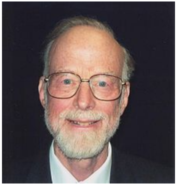
- Algoritmo proposto por C. A. R. Hoare em 1960.



- É o algoritmo de ordenação *in loco* mais rápido que se conhece para uma ampla variedade de situações.

Introdução

- Algoritmo proposto por C. A. R. Hoare em 1960.

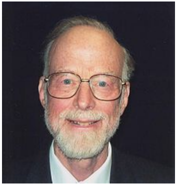


- É o algoritmo de ordenação *in loco* mais rápido que se conhece para uma ampla variedade de situações.
- Apesar disso, possui complexidade $O(n^2)$ no pior caso.

Introdução

UNIVERSIDADE FEDERAL DO CEARÁ
CAMPUS QUIXADÁ

- Algoritmo proposto por C. A. R. Hoare em 1960.



- É o algoritmo de ordenação *in loco* mais rápido que se conhece para uma ampla variedade de situações.

- Apesar disso, possui complexidade $O(n^2)$ no pior caso.

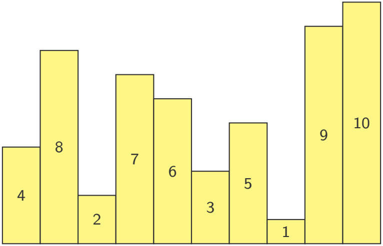
- Provavelmente é o mais utilizado (ou pelo menos deveria ser).

Algoritmo

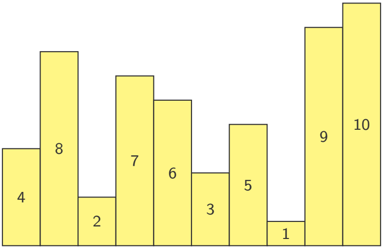
- Como o mergesort, é um algoritmo de divisão e conquista.
- Basicamente divide o problema de ordenar um conjunto com n itens em dois problemas menores.
- Os problemas menores são ordenados independentemente.
- As partições são combinadas para produzir a solução final.

Quicksort - Ideia

UNIVERSIDADE FEDERAL DO CEARÁ
CAMPUS QUIXADÁ

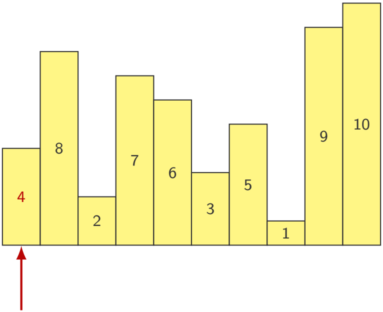


Quicksort - Ideia



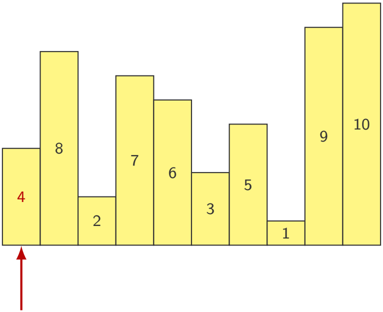
- Escolhemos um pivô (ex: 4)

Quicksort - Ideia



- Escolhemos um pivô (ex: 4)

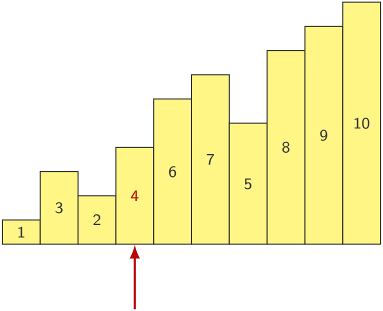
Quicksort - Ideia



- Escolhemos um pivô (ex: 4)
- Colocamos os elementos menores que o pivô à esquerda dele
- e os elementos maiores que o pivô à direita

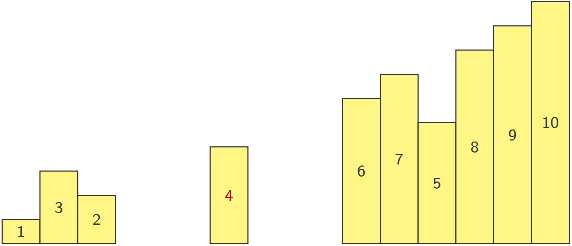
Quicksort - Ideia

Universidade Federal do Ceará
Campus Quixadá



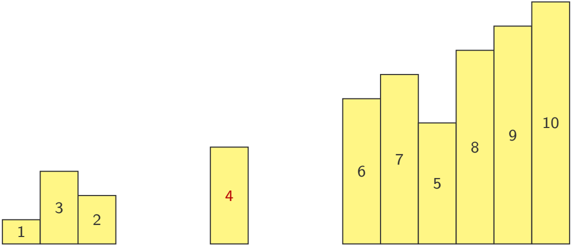
- Escolhemos um pivô (ex: 4)
- Colocamos os elementos menores que o pivô à esquerda dele
- e os elementos maiores que o pivô à direita

Quicksort - Ideia



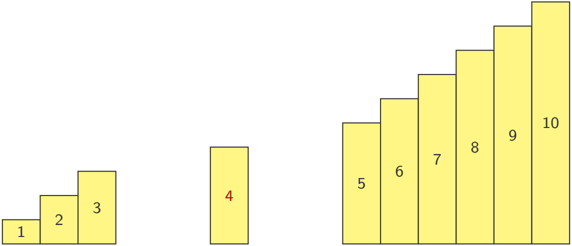
- Escolhemos um pivô (ex: 4)
- Colocamos os elementos menores que o pivô à esquerda dele
- e os elementos maiores que o pivô à direita
- Com isso, o pivô já está na posição correta

Quicksort - Ideia



- Escolhemos um pivô (ex: 4)
- Colocamos os elementos menores que o pivô à esquerda dele
- e os elementos maiores que o pivô à direita
- Com isso, o pivô já está na posição correta
- O lado esquerdo e o direito podem ser ordenados independentemente

Quicksort - Ideia



- Escolhemos um pivô (ex: 4)
- Colocamos os elementos menores que o pivô à esquerda dele
- e os elementos maiores que o pivô à direita
- Com isso, o pivô já está na posição correta
- O lado esquerdo e o direito podem ser ordenados independentemente

Divisão e Conquista no Quicksort

Assim como o Mergesort, o Quicksort aplica o paradigma de Divisão e Conquista:

- Dividir: rearranja o vetor $A[p..r]$ em dois subvetores (possivelmente vazios) $A[p..q-1]$ e $A[q+1..r]$ tal que $A[p..q-1] \leq A[q] < A[q+1..r]$. O índice q é calculado como parte deste procedimento de separação.

Divisão e Conquista no Quicksort

Assim como o Mergesort, o Quicksort aplica o paradigma de Divisão e Conquista:

- Dividir: rearranja o vetor $A[p..r]$ em dois subvetores (possivelmente vazios) $A[p..q-1]$ e $A[q+1..r]$ tal que $A[p..q-1] \leq A[q] < A[q+1..r]$. O índice q é calculado como parte deste procedimento de separação.
- Conquistar: Ordena os subvetores $A[p..q-1]$ e $A[q+1..r]$ por meio de chamadas recursivas ao quicksort.

Divisão e Conquista no Quicksort

Assim como o Mergesort, o Quicksort aplica o paradigma de Divisão e Conquista:

- Dividir: rearranja o vetor $A[p..r]$ em dois subvetores (possivelmente vazios) $A[p..q-1]$ e $A[q+1..r]$ tal que $A[p..q-1] \leq A[q] < A[q+1..r]$. O índice q é calculado como parte deste procedimento de separação.
- Conquistar: Ordena os subvetores $A[p..q-1]$ e $A[q+1..r]$ por meio de chamadas recursivas ao quicksort.
- Combinar: Os subvetores já estão ordenados, não há o que fazer: o vetor $A[p..r]$ encontra-se ordenado.

O algoritmo Quicksort

```
int separa(int A[], int p, int r);
```

O algoritmo Quicksort

```c
int separa(int A[], int p, int r);
```

- escolhe um pivô

O algoritmo Quicksort

```
int separa(int A[], int p, int r);
```

- escolhe um pivô
- coloca os elementos menores à esquerda do pivô

O algoritmo Quicksort

```
int separa(int A[], int p, int r);
```

- escolhe um pivô
- coloca os elementos menores à esquerda do pivô
- coloca os elementos maiores à direita do pivô

O algoritmo Quicksort

```
int separa(int A[], int p, int r);
```

- escolhe um pivô
- coloca os elementos menores à esquerda do pivô
- coloca os elementos maiores à direita do pivô
- devolve a posição final do pivô

O algoritmo Quicksort

```c
int separa(int A[], int p, int r);
```

- escolhe um pivô
- coloca os elementos menores à esquerda do pivô
- coloca os elementos maiores à direita do pivô
- devolve a posição final do pivô

```c
void quicksort(int A[], int p, int r) {
    if (p < r) {
        int i = separa(A, p, r);
        quicksort(A, p, i-1);
        quicksort(A, i+1, r);
    }
}
```

O algoritmo Quicksort

```
int separa(int A[], int p, int r);
```

- escolhe um pivô
- coloca os elementos menores à esquerda do pivô
- coloca os elementos maiores à direita do pivô
- devolve a posição final do pivô

```
void quicksort(int A[], int p, int r) {
    if (p < r) {
        int i = separa(A, p, r);
        quicksort(A, p, i-1);
        quicksort(A, i+1, r);
    }
}
```

- Basta particionar o vetor em dois

O algoritmo Quicksort

```
int separa(int A[], int p, int r);
```

- escolhe um pivô
- coloca os elementos menores à esquerda do pivô
- coloca os elementos maiores à direita do pivô
- devolve a posição final do pivô

```
void quicksort(int A[], int p, int r) {
    if (p < r) {
        int i = separa(A, p, r);
        quicksort(A, p, i-1);
        quicksort(A, i+1, r);
    }
}
```

- Basta particionar o vetor em dois
- e ordenar o lado esquerdo e o direito

O problema da separação

O núcleo do algoritmo Quicksort é o seguinte problema da separação:

- rearranjar um vetor $A[p \ldots r]$ de modo que

$$A[p \ldots j-1] \leq A[j] < A[j+1 \ldots r]$$

para algum j tal que $p \leq j < r$.

O problema da separação

O núcleo do algoritmo Quicksort é o seguinte problema da separação:

- rearranjar um vetor $A[p \dots r]$ de modo que

$$A[p \dots j-1] \leq A[j] < A[j+1 \dots r]$$

para algum j tal que $p \leq j < r$.

- Exemplo: aqui, $A[j]$ é o pivô.

777	222	111	777	999	444	555	666	555	888

							j		
666	222	111	777	555	444	555	777	999	888

O problema da separação

- O ponto de partida para a solução deste problema é a escolha de um pivô, digamos c.

O problema da separação

- O ponto de partida para a solução deste problema é a escolha de um pivô, digamos c.
- Os elementos do vetor que forem maiores que c serão considerados grandes e os demais serão considerados pequenos.

O problema da separação

- O ponto de partida para a solução deste problema é a escolha de um pivô, digamos c.
- Os elementos do vetor que forem maiores que c serão considerados grandes e os demais serão considerados pequenos.
- É importante escolher c de tal modo que as duas partes do vetor rearranjado sejam estritamente menores que o vetor todo.

O problema da separação

- O ponto de partida para a solução deste problema é a escolha de um pivô, digamos c.

- Os elementos do vetor que forem maiores que c serão considerados grandes e os demais serão considerados pequenos.

- É importante escolher c de tal modo que as duas partes do vetor rearranjado sejam estritamente menores que o vetor todo.

- A dificuldade está em resolver o problema da separação de maneira rápida sem usar muito espaço de trabalho.

O algoritmo da separação

```
/* Recebe um vetor A[p..r] com p <= r.
 * Rearranja os elementos do vetor e devolve
 * j em p..r tal que A[p..j-1] <= A[j] < A[j+1..r].
 */
int separa (int A[], int p, int r) {
    int c = A[r];
    int j = p;
    for (int k = p; k < r; k++) {
        if (A[k] <= c) {
            std::swap(A[k], A[j]);
            j++;
        }
    }
    A[r] = A[j];
    A[j] = c;
    return j;
}
```

Corretude do algoritmo da separação

- No início de cada iteração do laço **for** valem os seguintes invariantes:

(a) $A[p...r]$ é uma permutação do vetor original,
(b) $A[p...j-1] \leq c < A[j...k-1]$,
(c) $A[r] = c$
(d) $p \leq j \leq k \leq r$.

p			j			k			r
$\leq c$	$\leq c$	$\leq c$	$> c$	$> c$	$> c$	?	?	?	$= c$

Início de uma iteração do laço `for`

Corretude do algoritmo da separação

- No início de cada iteração do laço **for** valem os seguintes invariantes:

(a) $A[p...r]$ é uma permutação do vetor original,
(b) $A[p...j-1] \leq c < A[j...k-1]$,
(c) $A[r] = c$
(d) $p \leq j \leq k \leq r$.

p			j			k			r
$\leq c$	$\leq c$	$\leq c$	$> c$	$> c$	$> c$	?	?	?	$= c$

Início de uma iteração do laço `for`

				j					k
$\leq c$	$\leq c$	$\leq c$	$\leq c$	$> c$	$> c$	$> c$	$> c$	$> c$	$= c$

Última iteração do laço `for`.

Corretude do algoritmo da separação

- No início de cada iteração do laço **for** valem os seguintes invariantes:

(a) $A[p...r]$ é uma permutação do vetor original,
(b) $A[p...j-1] \leq c < A[j...k-1]$,
(c) $A[r] = c$
(d) $p \leq j \leq k \leq r$.

p			j			k			r
$\leq c$	$\leq c$	$\leq c$	$> c$	$> c$	$> c$	?	?	?	$= c$

Início de uma iteração do laço `for`

				j					k
$\leq c$	$\leq c$	$\leq c$	$\leq c$	$> c$	$> c$	$> c$	$> c$	$> c$	$= c$

Última iteração do laço `for`.

				j					k
$\leq c$	$\leq c$	$\leq c$	$\leq c$	$= c$	$> c$	$> c$	$> c$	$> c$	$> c$

Passagem pela última linha da função `separa`.

Desempenho do algoritmo da separação

UNIVERSIDADE FEDERAL DO CEARÁ
CAMPUS QUIXADÁ

```
int separa (int A[], int p, int r) {
    int c = A[r];
    int j = p;
    for (int k = p; k < r; k++) {
        if (A[k] <= c) {
            std::swap(A[k], A[j]);
            j++;
        }
    }
    A[r] = A[j];
    A[j] = c;
    return j;
}
```

- O consumo de tempo da função `separa` é proporcional ao número de iterações.

Desempenho do algoritmo da separação

UNIVERSIDADE FEDERAL DO CEARÁ
CAMPUS QUIXADÁ

```
int separa (int A[], int p, int r) {
    int c = A[r];
    int j = p;
    for (int k = p; k < r; k++) {
        if (A[k] <= c) {
            std::swap(A[k], A[j]);
            j++;
        }
    }
    A[r] = A[j];
    A[j] = c;
    return j;
}
```

- O consumo de tempo da função `separa` é proporcional ao número de iterações.
- Como o número de iterações é $r - p + 1$, podemos dizer que o consumo de tempo é proporcional ao número de elementos do vetor.

Quicksort

```
void quicksort(int A[], int p, int r) {
    if (p < r) {
        int i = separa(A, p, r);
        quicksort(A, p, i-1);
        quicksort(A, i+1, r);
    }
}
```

O desempenho do Quicksort

- O tempo de execução do quicksort depende do particionamento ser balanceado ou não ser balanceado.

O desempenho do Quicksort

- O tempo de execução do quicksort depende do particionamento ser balanceado ou não ser balanceado.
- Se o particionamento é balanceado, o algoritmo é executado assintoticamente tão rápido quanto o mergesort.

O desempenho do Quicksort

- O tempo de execução do quicksort depende do particionamento ser balanceado ou não ser balanceado.
- Se o particionamento é balanceado, o algoritmo é executado assintoticamente tão rápido quanto o mergesort.
- Contudo, se o particionamento é não balanceado, ele pode ser executado assintoticamente tão lento quanto a ordenação por inserção.

Pior caso do Quicksort

- O comportamento do pior caso para o quicksort ocorre quando a rotina de separação produz um subproblema com $n - 1$ elementos e um com 0 elementos. (isso acontece, por exemplo, se o vetor já estiver ordenado ou quase ordenado.)

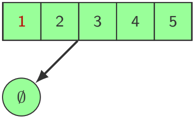
Particionamento no pior caso

1	2	3	4	5

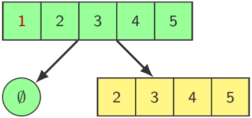
Particionamento no pior caso

1	2	3	4	5

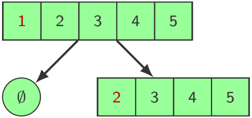
Particionamento no pior caso



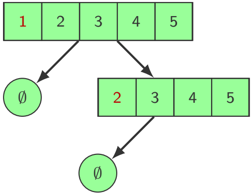
Particionamento no pior caso



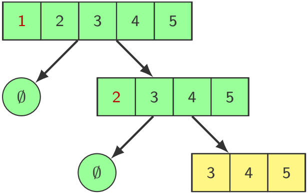
Particionamento no pior caso



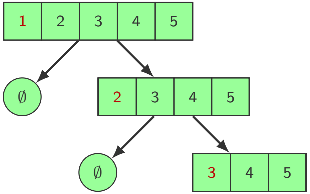
Particionamento no pior caso



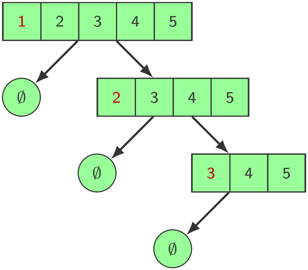
Particionamento no pior caso



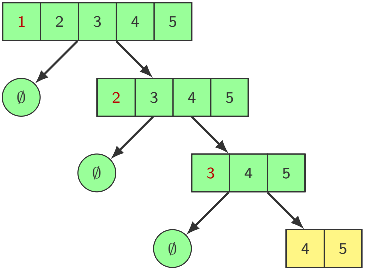
Particionamento no pior caso



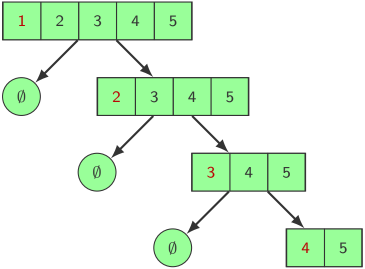
Particionamento no pior caso



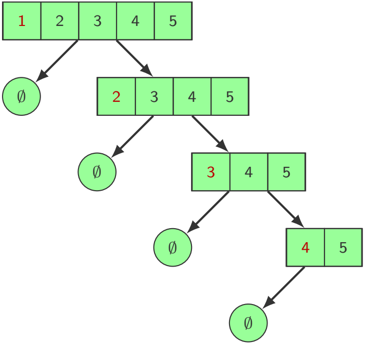
Particionamento no pior caso



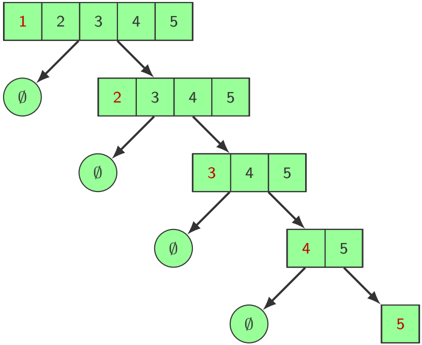
Particionamento no pior caso



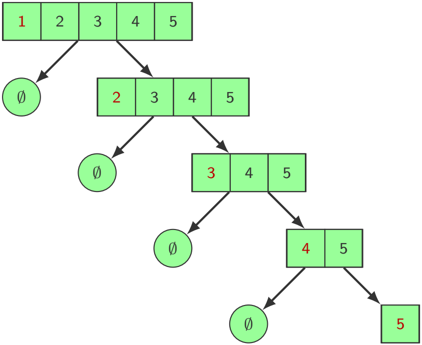
Particionamento no pior caso



Particionamento no pior caso

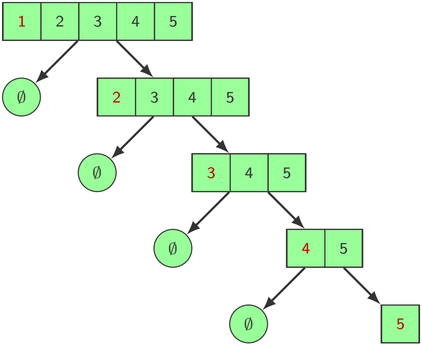


Particionamento no pior caso



$c \cdot n$

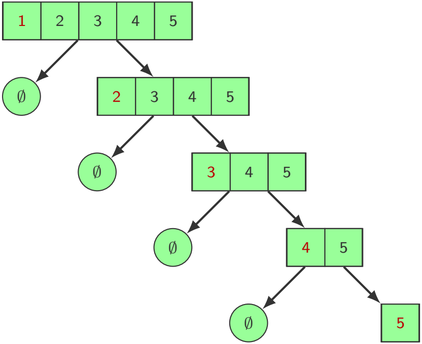
Particionamento no pior caso



$c \cdot n$

$c \cdot (n-1)$

Particionamento no pior caso

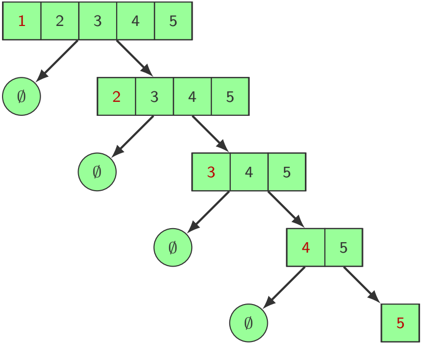


$$c \cdot n$$

$$c \cdot (n-1)$$

$$c \cdot (n-2)$$

Particionamento no pior caso



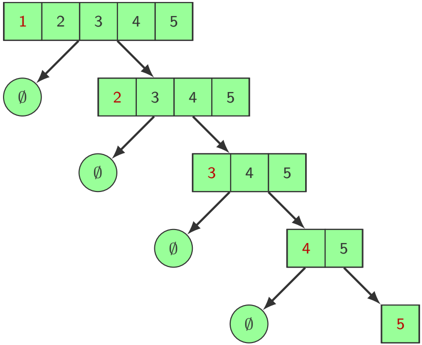
$$c \cdot n$$

$$c \cdot (n-1)$$

$$c \cdot (n-2)$$

$$\cdots$$

Particionamento no pior caso



$c \cdot n$

$c \cdot (n-1)$

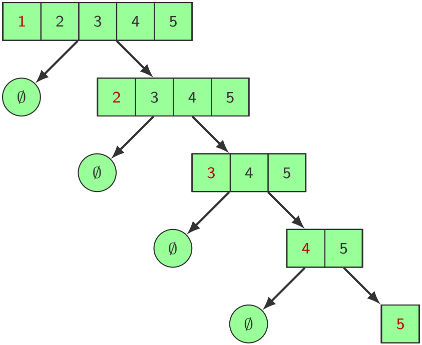
$c \cdot (n-2)$

$\cdots$

c

Particionamento no pior caso

UNIVERSIDADE FEDERAL DO CEARÁ
CAMPUS QUIXADÁ



$c \cdot n$

$c \cdot (n-1)$

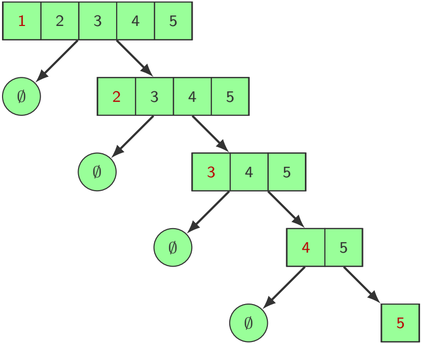
$c \cdot (n-2)$

$\cdots$

c

O tempo de execução do Quicksort é, no pior caso:

Particionamento no pior caso



$c \cdot n$

$c \cdot (n-1)$

$c \cdot (n-2)$

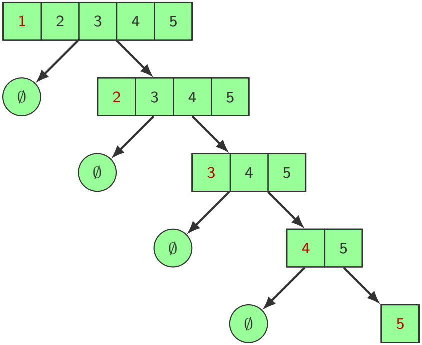
$\cdots$

c

O tempo de execução do Quicksort é, no pior caso:

$$c \cdot n + c \cdot (n-1) + \cdots + c$$

Particionamento no pior caso



$c \cdot n$

$c \cdot (n-1)$

$c \cdot (n-2)$

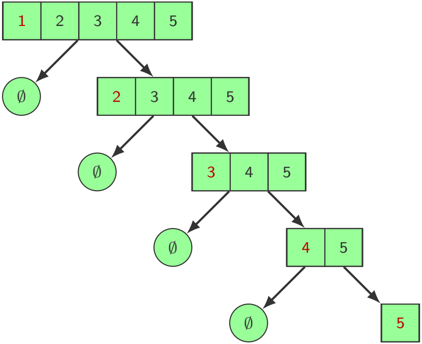
$\cdots$

c

O tempo de execução do Quicksort é, no pior caso:

$$c \cdot n + c \cdot (n-1) + \cdots + c = c \sum_{j=1}^{n} j$$

Particionamento no pior caso



$c \cdot n$

$c \cdot (n-1)$

$c \cdot (n-2)$

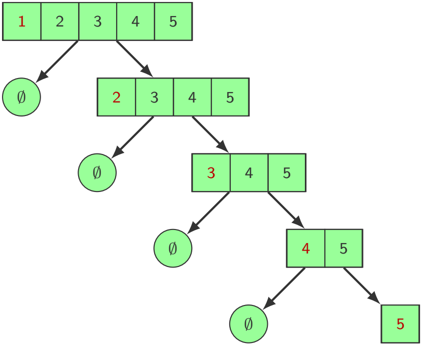
$\cdots$

c

O tempo de execução do Quicksort é, no pior caso:

$$c \cdot n + c \cdot (n-1) + \cdots + c = c \sum_{j=1}^{n} j = c\frac{n(n+1)}{2}$$

Particionamento no pior caso



$c \cdot n$

$c \cdot (n-1)$

$c \cdot (n-2)$

$\cdots$

c

O tempo de execução do Quicksort é, no pior caso:

$$c \cdot n + c \cdot (n-1) + \cdots + c = c \sum_{j=1}^{n} j = c \frac{n(n+1)}{2} = O(n^2)$$

Particionamento no melhor caso

- Na divisão mais equitativa possível, a função `separa` produz dois subproblemas, cada um de tamanho não maior que $n/2$.

Particionamento no melhor caso

UNIVERSIDADE FEDERAL DO CEARÁ
CAMPUS QUIXADÁ

- Na divisão mais equitativa possível, a função `separa` produz dois subproblemas, cada um de tamanho não maior que $n/2$.

- Nesse caso, teríamos a seguinte recorrência para o tempo de execução:

$$T(n) = 2T(n/2) + \theta(n).$$

Particionamento no melhor caso

Universidade Federal do Ceará
Campus Quixadá

- Na divisão mais equitativa possível, a função `separa` produz dois subproblemas, cada um de tamanho não maior que $n/2$.

- Nesse caso, teríamos a seguinte recorrência para o tempo de execução:

$$T(n) = 2T(n/2) + \theta(n).$$

- Resolvendo a recorrência, temos que $T(n) = O(n \log n)$.

Particionamento no melhor caso

- Na divisão mais equitativa possível, a função `separa` produz dois subproblemas, cada um de tamanho não maior que $n/2$.
- Nesse caso, teríamos a seguinte recorrência para o tempo de execução:

$$T(n) = 2T(n/2) + \theta(n).$$

- Resolvendo a recorrência, temos que $T(n) = O(n \log n)$.
- Balanceando igualmente os dois lados da partição em todo nível da recursão, obtemos um algoritmo assintoticamente mais rápido.

Conclusão

UNIVERSIDADE FEDERAL DO CEARÁ
CAMPUS QUIXADÁ

O QuickSort é um algoritmo de ordenação $O(n^2)$

Conclusão

O QuickSort é um algoritmo de ordenação $O(n^2)$

- Mas ele pode ser rápido na prática

Conclusão

O QuickSort é um algoritmo de ordenação $O(n^2)$

- Mas ele pode ser rápido na prática
- Leva tempo $O(n \lg n)$ (em média) para ordenar uma permutação aleatória

Conclusão

UNIVERSIDADE FEDERAL DO CEARÁ
CAMPUS QUIXADÁ

O QuickSort é um algoritmo de ordenação $O(n^2)$

- Mas ele pode ser rápido na prática
- Leva tempo $O(n \lg n)$ (em média) para ordenar uma permutação aleatória
- Precisa de espaço adicional $O(n)$ para a pilha de recursão

Comparação Assintótica

Algoritmo	Melhor Caso	Caso Médio	Pior Caso	Memória
BubbleSort	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$
SelectionSort	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$
InsertionSort	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$

Comparação Assintótica

Algoritmo	Melhor Caso	Caso Médio	Pior Caso	Memória
BubbleSort	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$
SelectionSort	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$
InsertionSort	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$
MergeSort	$O(n \lg n)$	$O(n \lg n)$	$O(n \lg n)$	$O(n)$
QuickSort	$O(n \lg n)$	$O(n \lg n)$	$O(n^2)$	$O(n)$

Exercícios

Exercícios

(1) Escreva uma função que rearranje um vetor $V[p..r]$ de números inteiros de modo que os elementos negativos e nulos fiquem à esquerda e os positivos fiquem à direita. Em outras palavras, rearranje o vetor de modo que tenhamos $V[p..j-1] \leq 0$ e $V[j..r] > 0$ para algum j em $\{p, \ldots, r+1\}$.

- Procure escrever uma função eficiente que não use vetor auxiliar.

(2) Digamos que um vetor $V[p..r]$ está arrumado se existe j em $p..r$ que satisfaz:

$$V[p..j-1] \leq V[j] < V[j+1..r]$$

Escreva um algoritmo que decida se $V[p..r]$ está arrumado. Em caso afirmativo, o seu algoritmo deve devolver o valor de j.

Exercícios

UNIVERSIDADE FEDERAL DO CEARÁ
CAMPUS QUIXADÁ

(3) Escreva uma implementação do algoritmo Quicksort que evite aplicar a função a vetores com menos do que dois elementos.

(4) A função `separa` produz um rearranjo estável do vetor?

(5) A função `quicksort` produz uma ordenação estável?

(6) (recursão em cauda) Mostre que a segunda invocação da função `quicksort` pode ser eliminada se trocarmos o `if` por um `while` apropriado.

(7) Escreva uma versão do algoritmo quicksort que rearranje uma lista duplamente encadeada de modo que ela fique em ordem crescente. Sua função não deve alocar novas células na memória.

Exercícios

Faça uma versão do QuickSort que seja boa para quando há muitos elementos repetidos no vetor.

- A ideia é particionar o vetor em três partes: menores, iguais e maiores que o pivô

Corretude do algoritmo da separação

- Provar: No início de cada iteração do laço **for** valem os seguintes invariantes:

(a) $A[p..r]$ é uma permutação do vetor original,
(b) $A[p..j-1] \leq c < A[j..k-1]$,
(c) $A[r] = c$
(d) $p \leq j \leq k$.

```
int separa (int A[], int p, int r) {
    int c = A[r];
    int j = p;
    for (int k = p; k < r; k++) {
        if (A[k] <= c) {
            std::swap(A[k], A[j]);
            j++;
        }
    }
    A[r] = A[j];
    A[j] = c;
    return j;
}
```

UNIVERSIDADE FEDERAL DO CEARÁ
CAMPUS QUIXADÁ

FIM