Quicksort

Estrutura de Dados — QXD0010

UNIVERSIDADE FEDERAL DO CEARÁ
CAMPUS QUIXADÁ

Prof. Atílio Gomes Luiz
gomes.atilio@ufc.br

Universidade Federal do Ceará

2º semestre/2023

Introdução

Vimos dois algoritmos de ordenação $O(n^2)$:

- `insertionsort`
- `selectionsort`

Introdução

Vimos dois algoritmos de ordenação $O(n^2)$:

- `insertionsort`
- `selectionsort`

Nessa aula veremos um algoritmo de ordenação $O(n \log n)$ no caso médio e $O(n^2)$ no pior caso

Introdução

Universidade Federal do Ceará
Campus Quixadá

Vimos dois algoritmos de ordenação $O(n^2)$:

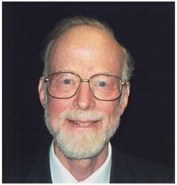
- `insertionsort`
- `selectionsort`

Nessa aula veremos um algoritmo de ordenação $O(n \log n)$ no caso médio e $O(n^2)$ no pior caso

Ele é baseado em uma técnica de projeto de algoritmos chamada Divisão e Conquista ou Dividir para Conquistar

Introdução

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



- É o algoritmo de ordenação 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).

UNIVERSIDADE FEDERAL DO CEARÁ
CAMPUS QUIXADÁ

Divisão e Conquista

Técnica de Divisão e Conquista

Observação:

- A recursão parte do princípio que é mais fácil resolver problemas menores.
- Para certos problemas, podemos dividi-los em duas ou mais partes.

Técnica de Divisão e Conquista

Observação:

- A recursão parte do princípio que é mais fácil resolver problemas menores.
- Para certos problemas, podemos dividi-los em duas ou mais partes.

  Etapas do paradigma de Divisão e Conquista:

  - Dividir: Quebramos o problema em vários subproblemas menores.
    - ex: quebramos um vetor a ser ordenado em dois.

Técnica de Divisão e Conquista

Observação:

- A recursão parte do princípio que é mais fácil resolver problemas menores.
- Para certos problemas, podemos dividi-los em duas ou mais partes.

  Etapas do paradigma de Divisão e Conquista:

  - Dividir: Quebramos o problema em vários subproblemas menores.
    - ex: quebramos um vetor a ser ordenado em dois.
  - Conquistar: Os subproblemas são resolvidos recursivamente. Se eles forem pequenos o bastante, eles são resolvidos usando o próprio algoritmo que está sendo definido.
    - ex: um subvetor com um único elemento já está ordenado.

Técnica de Divisão e Conquista

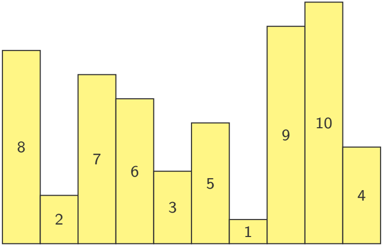
Observação:

- A recursão parte do princípio que é mais fácil resolver problemas menores.
- Para certos problemas, podemos dividi-los em duas ou mais partes.

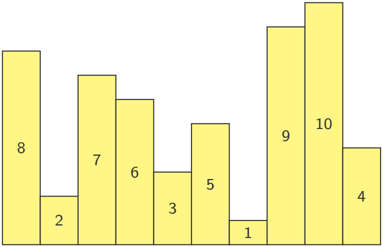
  Etapas do paradigma de Divisão e Conquista:

  - Dividir: Quebramos o problema em vários subproblemas menores.
    - ex: quebramos um vetor a ser ordenado em dois.
  - Conquistar: Os subproblemas são resolvidos recursivamente. Se eles forem pequenos o bastante, eles são resolvidos usando o próprio algoritmo que está sendo definido.
    - ex: um subvetor com um único elemento já está ordenado.
  - Combinar: Combinamos a solução dos problemas menores a fim de obter a solução para o problema maior.
    - ex: intercalamos os dois vetores ordenados ou, se todos os elementos de um vetor forem menores que o do outro, apenas concatenamos os vetores.

Quicksort - Ideia

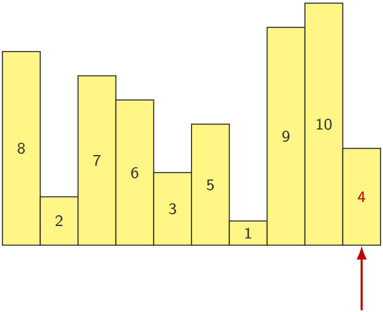


Quicksort - Ideia



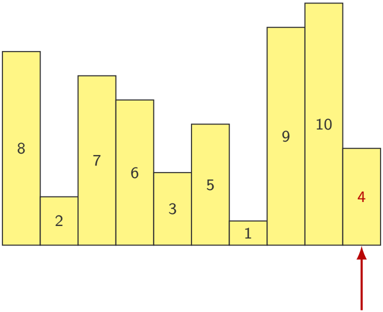
- Escolhemos um pivô (ex: 4)

Quicksort - Ideia



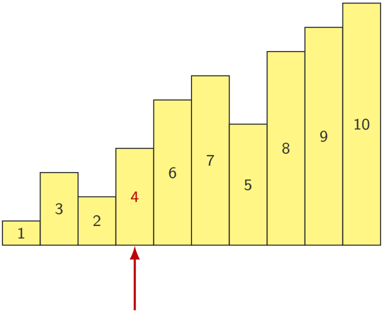
- Escolhemos um pivô (ex: 4)

Quicksort - Ideia



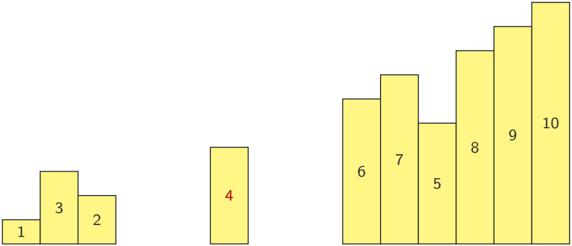
- Escolhemos um pivô (ex: 4)
- Colocamos os elementos menores que o pivô à esquerda dele
- e os elementos maiores que o pivô à direita

Quicksort - Ideia



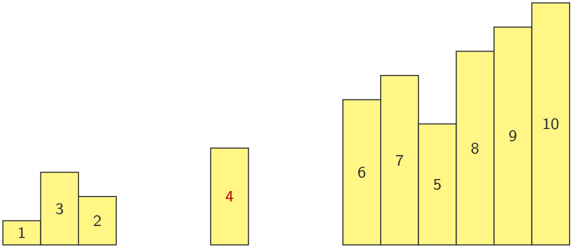
- Escolhemos um pivô (ex: 4)
- Colocamos os elementos menores que o pivô à esquerda dele
- e os elementos maiores que o pivô à direita

Quicksort - Ideia



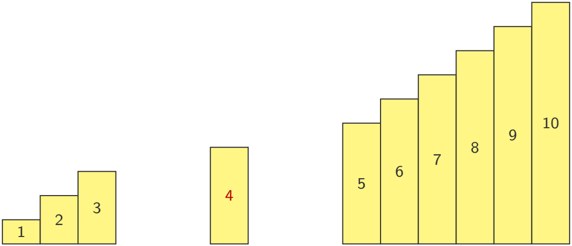
- Escolhemos um pivô (ex: 4)
- Colocamos os elementos menores que o pivô à esquerda dele
- e os elementos maiores que o pivô à direita
- Com isso, o pivô já está na posição correta

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

Quicksort aplica o paradigma de Divisão e Conquista:

Divisão e Conquista no Quicksort

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

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

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.

Um problema subjacente: Particionar um vetor

O problema da partição

UNIVERSIDADE FEDERAL DO CEARÁ
CAMPUS QUIXADÁ

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

- 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 $p \leq j \leq r$.

O problema da partição

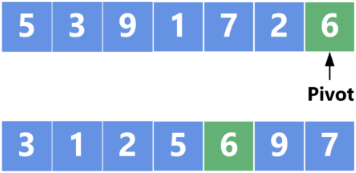
O núcleo do algoritmo Quicksort é o seguinte problema da partição:

- 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 $p \leq j \leq r$.

- Exemplo: aqui, 6 é o pivô.



O problema da partição

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

O problema da partição

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

O problema da partição

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

O problema da partição

- O ponto de partida para a solução deste problema é a escolha de um pivô, digamos p.
- Os elementos do vetor que forem maiores que p serão considerados grandes e os demais serão considerados pequenos.
- É importante escolher p 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 partição de maneira rápida sem usar muito espaço de trabalho.

O algoritmo da partição

O algoritmo da partição

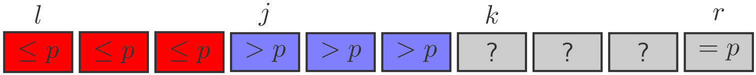
Recebe um vetor $A[l..r]$ com $l \leq r$. Rearranja os elementos do vetor e devolve j tal que $A[l..j-1] \leq A[j] \leq A[j+1..r]$

```
int partition (int A[], int l, int r) {
    int pivo = A[r];
    int j = l;
    for (int k = l; k < r; k++) {
        if (A[k] <= pivo) {
            int aux = A[k];
            A[k] = A[j];
            A[j] = aux;
            j++;
        }
    }
    A[r] = A[j];
    A[j] = pivo;
    return j;
}
```

Corretude do algoritmo da separação

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

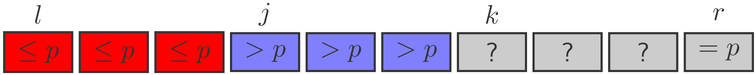
(a) $A[l...r]$ é uma permutação do vetor original,
(b) $A[l...j-1] \leq$ pivo $< A[j...k-1]$,
(c) $A[r] = p$
(d) $l \leq j \leq k \leq r$.



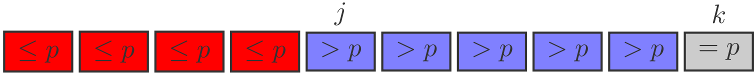
Início de uma iteração do laço `for`

Corretude do algoritmo da separação

- No início de cada iteração do laço **for** valem os seguintes invariantes:
  (a) $A[l...r]$ é uma permutação do vetor original,
  (b) $A[l...j-1] \leq$ pivo $< A[j...k-1]$,
  (c) $A[r] = p$
  (d) $l \leq j \leq k \leq r$.



Início de uma iteração do laço `for`



Última iteração do laço `for`.

Corretude do algoritmo da separação

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

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

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

Início de uma iteração do laço `for`

				j					k
$\leq p$	$\leq p$	$\leq p$	$\leq p$	$> p$	$> p$	$> p$	$> p$	$> p$	$= p$

Última iteração do laço `for`.

				j					k
$\leq p$	$\leq p$	$\leq p$	$\leq p$	$= p$	$> p$	$> p$	$> p$	$> p$	$> p$

Passagem pela última linha da função `separa`.

Desempenho do algoritmo da partição

```c
int partition (int A[], int l, int r) {
    int pivo = A[r];
    int j = l;
    for (int k = l; k < r; k++) {
        if (A[k] <= pivo) {
            int aux = A[k];
            A[k] = A[j];
            A[j] = aux;
            j++;
        }
    }
    A[r] = A[j];
    A[j] = pivo;
    return j;
}
```

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

Desempenho do algoritmo da partição

UNIVERSIDADE FEDERAL DO CEARÁ
CAMPUS QUIXADÁ

```c
int partition (int A[], int l, int r) {
    int pivo = A[r];
    int j = l;
    for (int k = l; k < r; k++) {
        if (A[k] <= pivo) {
            int aux = A[k];
            A[k] = A[j];
            A[j] = aux;
            j++;
        }
    }
    A[r] = A[j];
    A[j] = pivo;
    return j;
}
```

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

Quicksort

Quicksort

UNIVERSIDADE FEDERAL DO CEARÁ
CAMPUS QUIXADÁ

```
/**
 * Esta funcao rearranja o vetor A[l..r],
 * de modo que ele fique em ordem crescente.
 */
void quicksort (int A[], int l, int r) {
    if (l < r) {
        int j = partition(A, l, r);
        quicksort(A, l, j-1);
        quicksort(A, j+1, r);
    }
}
```

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 em tempo $O(n \lg n)$.
- 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.

- Vamos considerar que esse particionamento não balanceado surja em cada chamada recursiva.

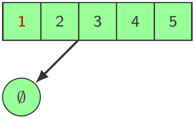
Particionamento no pior caso

1	2	3	4	5

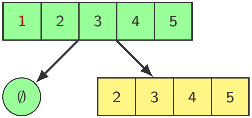
Particionamento no pior caso

1	2	3	4	5

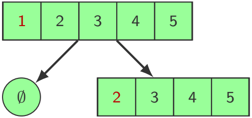
Particionamento no pior caso



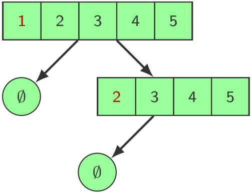
Particionamento no pior caso



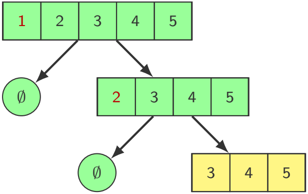
Particionamento no pior caso



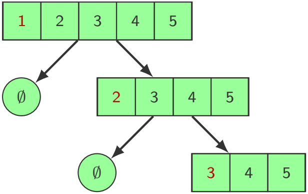
Particionamento no pior caso



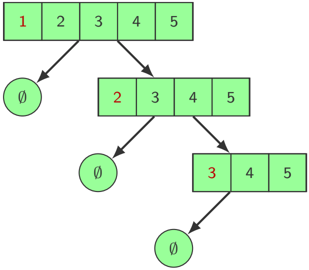
Particionamento no pior caso



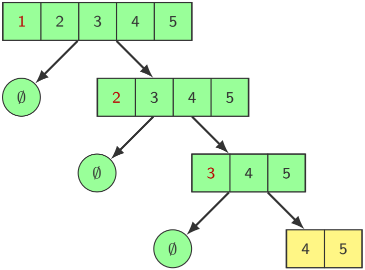
Particionamento no pior caso



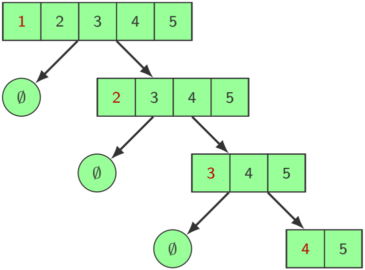
Particionamento no pior caso



Particionamento no pior caso

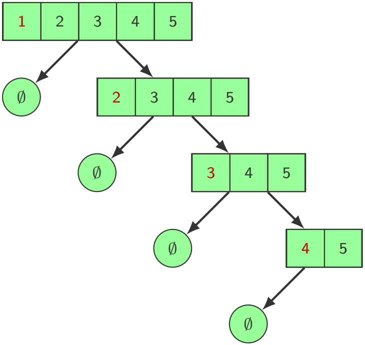


Particionamento no pior caso

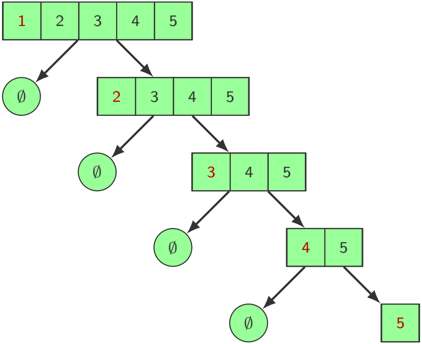


Particionamento no pior caso

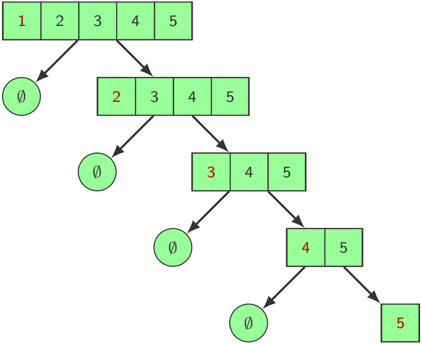
UNIVERSIDADE FEDERAL DO CEARÁ
CAMPUS QUIXADÁ



Particionamento no pior caso

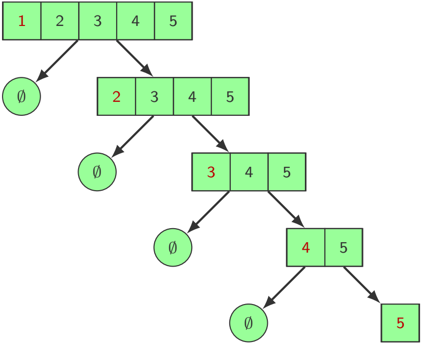


Particionamento no pior caso



$c \cdot n$

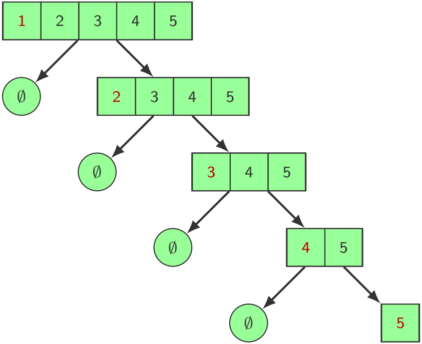
Particionamento no pior caso



$c \cdot n$

$c \cdot (n-1)$

Particionamento no pior caso

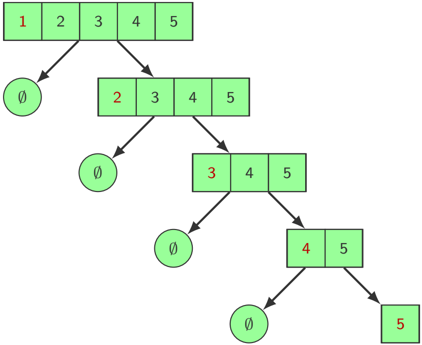


$c \cdot n$

$c \cdot (n-1)$

$c \cdot (n-2)$

Particionamento no pior caso



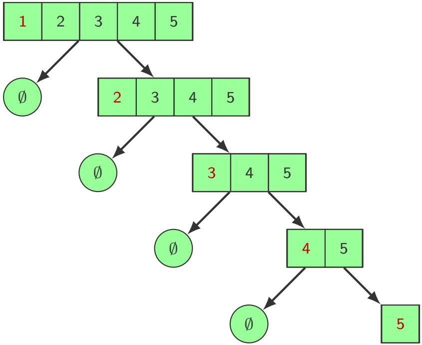
$c \cdot n$

$c \cdot (n-1)$

$c \cdot (n-2)$

$\cdots$

Particionamento no pior caso



$c \cdot n$

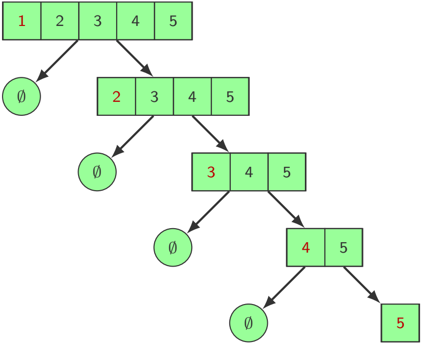
$c \cdot (n-1)$

$c \cdot (n-2)$

$\cdots$

c

Particionamento no pior caso



$c \cdot n$

$c \cdot (n-1)$

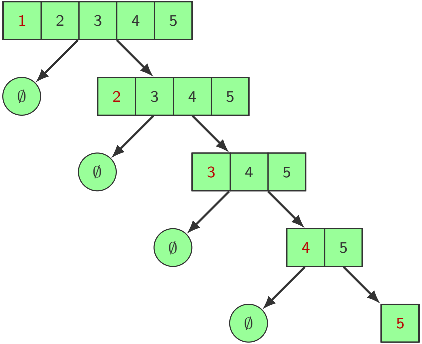
$c \cdot (n-2)$

$\cdots$

c

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

Particionamento no pior caso



$c \cdot n$

$c \cdot (n-1)$

$c \cdot (n-2)$

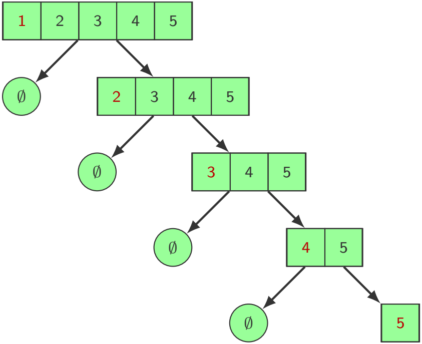
$\cdots$

c

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

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

Particionamento no pior caso



$c \cdot n$

$c \cdot (n-1)$

$c \cdot (n-2)$

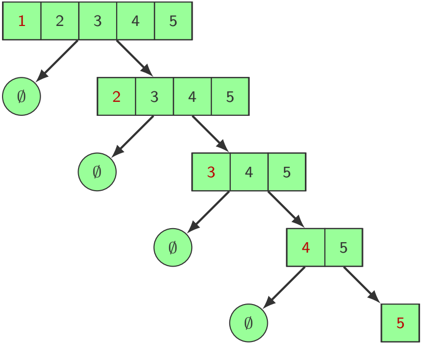
$\cdots$

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



$c \cdot n$

$c \cdot (n-1)$

$c \cdot (n-2)$

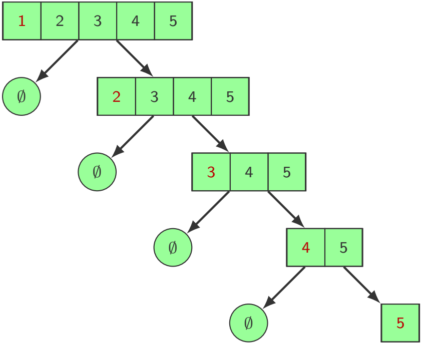
$\cdots$

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 `partition` produz dois subproblemas, cada um de tamanho não maior que $n/2$.

Particionamento no melhor caso

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

$$T(n) = 2T(n/2) + O(n).$$

Particionamento no melhor caso

UNIVERSIDADE FEDERAL DO CEARÁ
CAMPUS QUIXADÁ

- Na divisão mais equitativa possível, a função `partition` produz dois subproblemas, cada um de tamanho não maior que $n/2$.
- Nesse caso, teríamos a seguinte recorrência para o tempo de execução:
$$T(n) = 2T(n/2) + O(n).$$
- Resolvendo a recorrência, temos que $T(n) = O(n \log n)$.

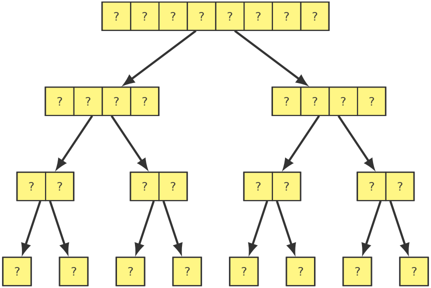
Particionamento no melhor caso

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

$$T(n) = 2T(n/2) + O(n).$$

- 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.

Tempo de execução para $n = 2^l$



$\leq c \cdot n$

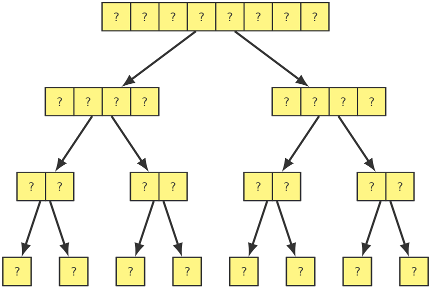
$\leq c \cdot 2(n/2)$

$\leq c \cdot 4(n/4)$

$\leq c \cdot 8(n/8)$

- Em cada nível gastamos tempo $\leq c \cdot n$

Tempo de execução para $n = 2^l$



$\leq c \cdot n$

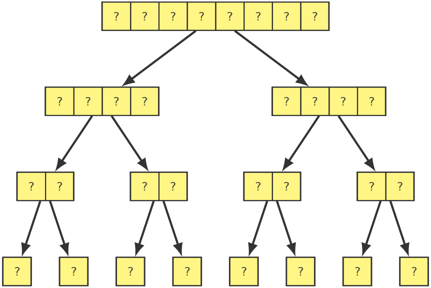
$\leq c \cdot 2(n/2)$

$\leq c \cdot 4(n/4)$

$\leq c \cdot 8(n/8)$

- Em cada nível gastamos tempo $\leq c \cdot n$
- Quantos níveis temos?

Tempo de execução para $n = 2^l$



$\leq c \cdot n$

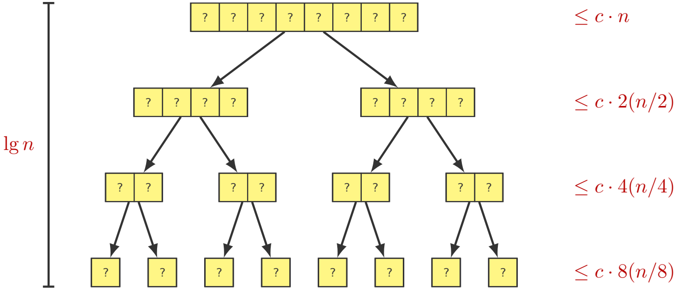
$\leq c \cdot 2(n/2)$

$\leq c \cdot 4(n/4)$

$\leq c \cdot 8(n/8)$

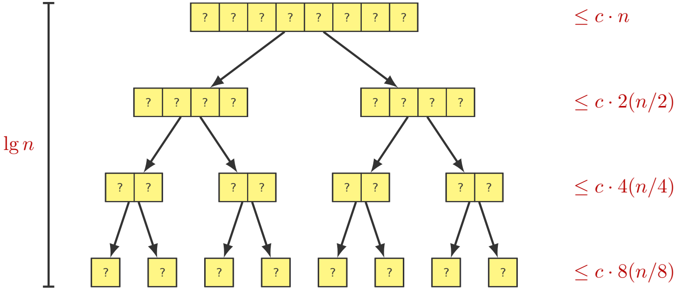
- Em cada nível gastamos tempo $\leq c \cdot n$
- Quantos níveis temos?
  - Dividimos n por 2 até que fique menor ou igual a 1

Tempo de execução para $n = 2^l$



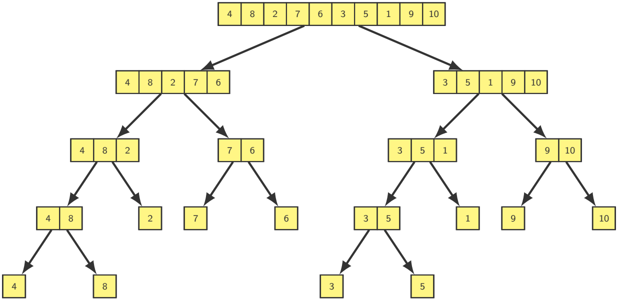
- Em cada nível gastamos tempo $\leq c \cdot n$
- Quantos níveis temos?
  - Dividimos n por 2 até que fique menor ou igual a 1
  - Ou seja, $l = \lg n$

Tempo de execução para $n = 2^l$

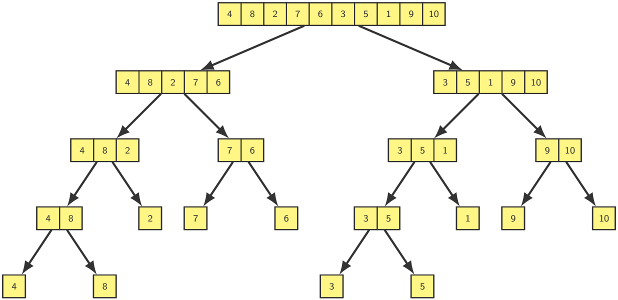


- Em cada nível gastamos tempo $\leq c \cdot n$
- Quantos níveis temos?
  - Dividimos n por 2 até que fique menor ou igual a 1
  - Ou seja, $l = \lg n$
- Tempo total: $c\, n \lg n = O(n \lg n)$

Tempo de execução para n qualquer

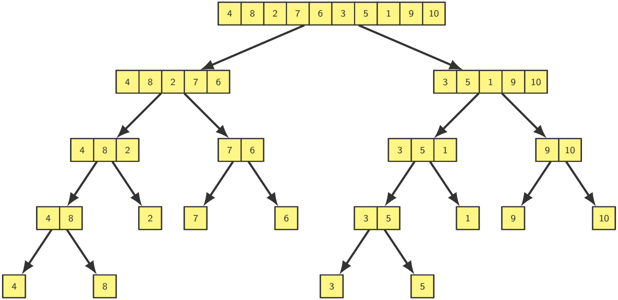


Tempo de execução para n qualquer



Qual o tempo de execução para n que não é potência de 2?

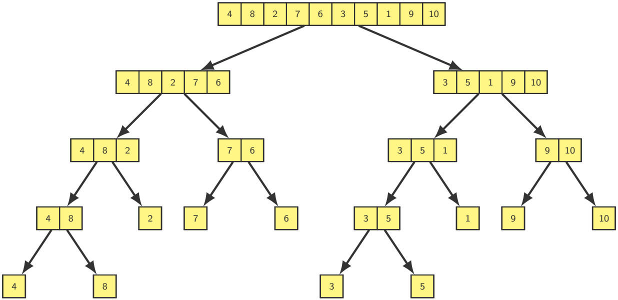
Tempo de execução para n qualquer



Qual o tempo de execução para n que não é potência de 2?

- Seja 2^k a próxima potência de 2 depois de n

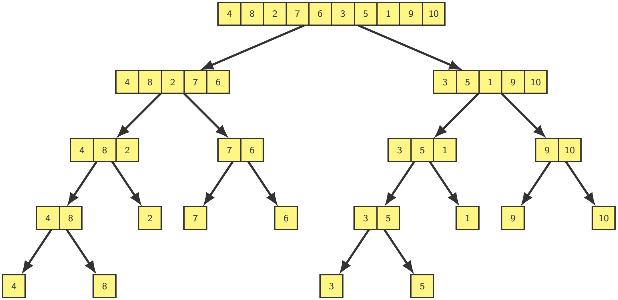
Tempo de execução para n qualquer



Qual o tempo de execução para n que não é potência de 2?

- Seja 2^k a próxima potência de 2 depois de n
  - Exemplo: Se $n = 3000$, a próxima potência é $4096 = 2^{12}$

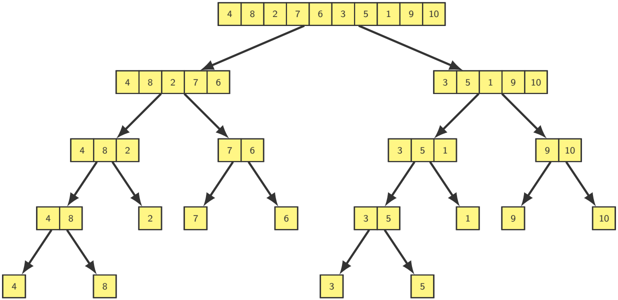
Tempo de execução para n qualquer



Qual o tempo de execução para n que não é potência de 2?

- Seja 2^k a próxima potência de 2 depois de n
  - Exemplo: Se $n = 3000$, a próxima potência é $4096 = 2^{12}$
- Temos que $2^{k-1} < n < 2^k$. Ou seja, $2^k < 2n$.

Tempo de execução para n qualquer

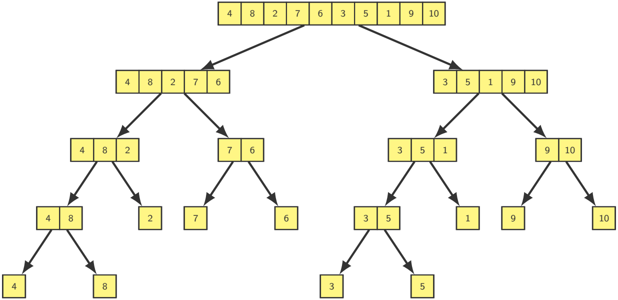


Qual o tempo de execução para n que não é potência de 2?

- Seja 2^k a próxima potência de 2 depois de n
  - Exemplo: Se $n = 3000$, a próxima potência é $4096 = 2^{12}$
- Temos que $2^{k-1} < n < 2^k$. Ou seja, $2^k < 2n$.
- O tempo de execução para n é menor do que

Tempo de execução para n qualquer

UNIVERSIDADE FEDERAL DO CEARÁ
CAMPUS QUIXADÁ

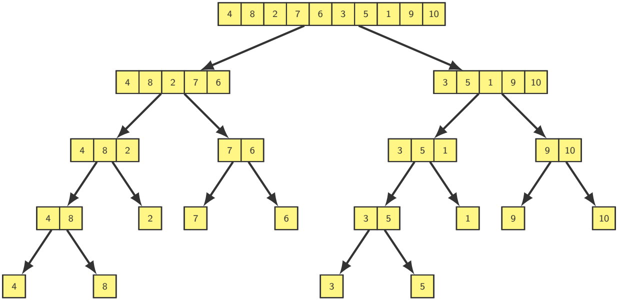


Qual o tempo de execução para n que não é potência de 2?

- Seja 2^k a próxima potência de 2 depois de n
  - Exemplo: Se $n = 3000$, a próxima potência é $4096 = 2^{12}$
- Temos que $2^{k-1} < n < 2^k$. Ou seja, $2^k < 2n$.
- O tempo de execução para n é menor do que

$$c\,2^k \lg 2^k$$

Tempo de execução para n qualquer

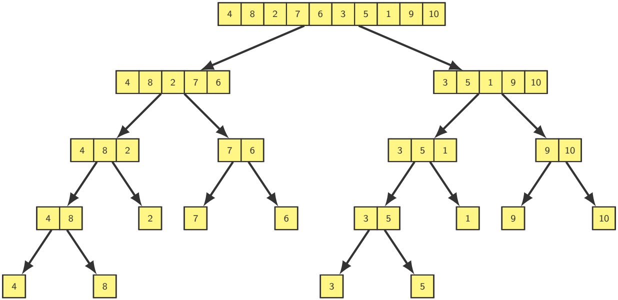


Qual o tempo de execução para n que não é potência de 2?

- Seja 2^k a próxima potência de 2 depois de n
  - Exemplo: Se $n = 3000$, a próxima potência é $4096 = 2^{12}$
- Temos que $2^{k-1} < n < 2^k$. Ou seja, $2^k < 2n$.
- O tempo de execução para n é menor do que

$$c\,2^k \lg 2^k$$

Tempo de execução para n qualquer



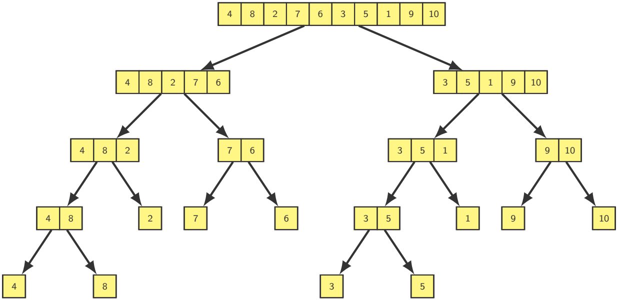
Qual o tempo de execução para n que não é potência de 2?

- Seja 2^k a próxima potência de 2 depois de n
  - Exemplo: Se $n = 3000$, a próxima potência é $4096 = 2^{12}$
- Temos que $2^{k-1} < n < 2^k$. Ou seja, $2^k < 2n$.
- O tempo de execução para n é menor do que

$$c\,2^k \lg 2^k \leq c2n \lg(2n)$$

Tempo de execução para n qualquer

UNIVERSIDADE
FEDERAL DO CEARÁ
CAMPUS QUIXADÁ

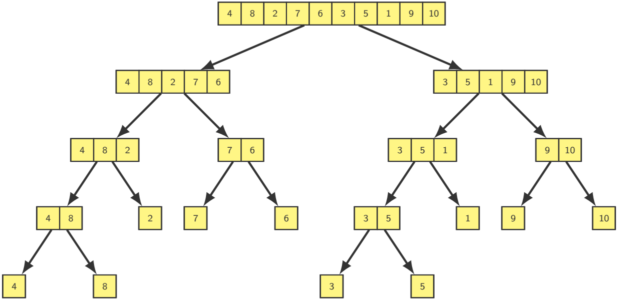


Qual o tempo de execução para n que não é potência de 2?

- Seja 2^k a próxima potência de 2 depois de n
  - Exemplo: Se $n = 3000$, a próxima potência é $4096 = 2^{12}$
- Temos que $2^{k-1} < n < 2^k$. Ou seja, $2^k < 2n$.
- O tempo de execução para n é menor do que

$$c\,2^k \lg 2^k \leq c2n \lg(2n) = 2cn(\lg 2 + \lg n)$$

Tempo de execução para n qualquer



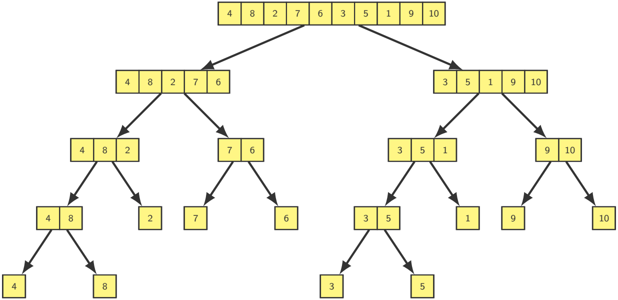
Qual o tempo de execução para n que não é potência de 2?

- Seja 2^k a próxima potência de 2 depois de n
  - Exemplo: Se $n = 3000$, a próxima potência é $4096 = 2^{12}$
- Temos que $2^{k-1} < n < 2^k$. Ou seja, $2^k < 2n$.
- O tempo de execução para n é menor do que

$$c\,2^k \lg 2^k \leq c2n \lg(2n) = 2cn(\lg 2 + \lg n) = 2cn + 2cn \lg n$$

Tempo de execução para n qualquer

UNIVERSIDADE FEDERAL DO CEARÁ
CAMPUS QUIXADÁ



Qual o tempo de execução para n que não é potência de 2?

- Seja 2^k a próxima potência de 2 depois de n
  - Exemplo: Se $n = 3000$, a próxima potência é $4096 = 2^{12}$
- Temos que $2^{k-1} < n < 2^k$. Ou seja, $2^k < 2n$.
- O tempo de execução para n é menor do que

$$c\,2^k\,\lg 2^k \leq c2n\,\lg(2n) = 2cn(\lg 2 + \lg n) = 2cn + 2cn\lg n = O(n\lg n)$$

Conclusão

UNIVERSIDADE FEDERAL DO CEARÁ
CAMPUS QUIXADÁ

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

Conclusão

UNIVERSIDADE FEDERAL DO CEARÁ
CAMPUS QUIXADÁ

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

- Mas ele pode ser rápido na prática

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

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
- Precisa de espaço adicional $O(n)$ para a pilha de recursão

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;
}
```

FIM