Programação Procedimental

Ordenação (parte 2)

Aula 14

Prof. Felipe A. Louza

Roteiro

1. Merge Sort
2. Quick Sort
3. Referências

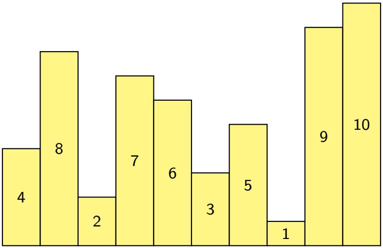
Na unidade anterior...

Vimos três algoritmos de ordenação $\mathrm{O}(n^2)$:

- `selectionsort`
- `bubblesort`
- `insertionsort`

Nessa aula veremos um novo algoritmo de ordenação

Estratégia: recursão

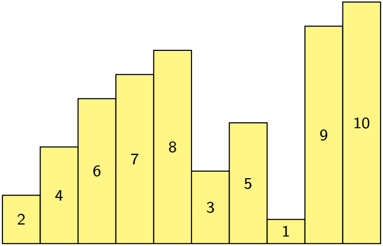


Como ordenar a primeira metade do vetor?

- usamos uma função `ordenar(int *v, int l, int r)`
  - ordena o vetor das posições `l` a `r` (inclusive)
  - poderia ser um dos algoritmos vistos anteriormente
  - mas faremos algo mais simples e melhor
- executamos `ordenar(v, 0, 4);`

E se quiséssemos ordenar a segunda parte?

Ordenando a segunda parte



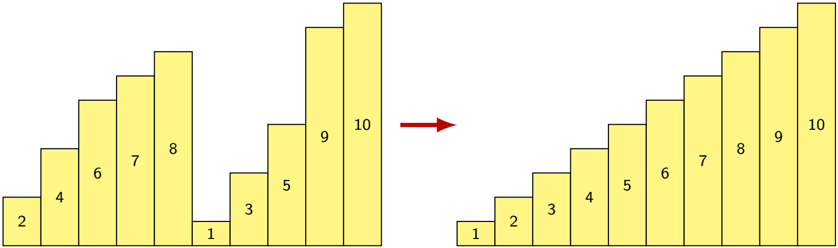
Para ordenar a segunda metade:

- executamos `ordenar(v, 5, 9);`

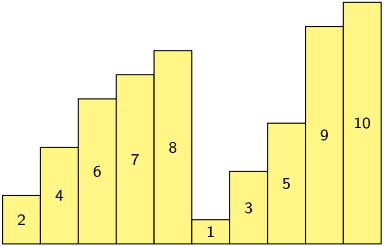
Ordenando todo o vetor

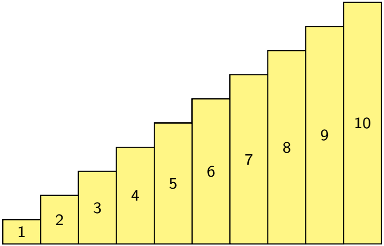
Se temos um vetor com as suas duas metades já ordenadas

- Como ordenar todo o vetor?



Intercalando





- Percorremos os dois subvetores
- Pegamos o mínimo e inserimos em um vetor auxiliar
- Depois copiamos o restante
- No final, copiamos do vetor auxiliar para o original

Divisão e conquista

Observação:

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

Divisão e conquista:

- Divisão: Quebramos o problema em vários subproblemas menores
  - ex: quebramos um vetor a ser ordenado em dois
- Conquista: Combinamos a solução dos problemas menores
  - ex: intercalamos os dois vetores ordenados

Ordenação por intercalação (*Merge Sort*)

Intercalação:

- Os dois subvetores estão armazenados em `v`:
  - O primeiro nas posições de `l` até `m`
  - O segundo nas posições de `m + 1` até `r`
- Precisamos de um vetor auxiliar do tamanho do vetor
- Vamos considerar que o maior vetor tem tamanho `MAX`
  - Exemplo `#define MAX 100`

Ordenação por intercalação (*Merge Sort*)

```c
void merge(int *v, int l, int m, int r) {
  int aux[MAX];
  int i = l, j = m + 1, k = 0;
  /*intercala*/
  while (i <= m && j <= r)
    if (v[i] <= v[j])
      aux[k++] = v[i++];
    else
      aux[k++] = v[j++];
  /*copia o resto do subvetor que não terminou*/
  while (i <= m)
    aux[k++] = v[i++];
  while (j <= r)
    aux[k++] = v[j++];
  /*copia de volta para v*/
  for (i = l, k = 0; i <= r; i++, k++)
      v[i] = aux[k];
}
```

Quantas comparações são feitas?

- a cada passo, aumentamos um em `i` ou em `j`
- no máximo $n := r - \ell + 1$

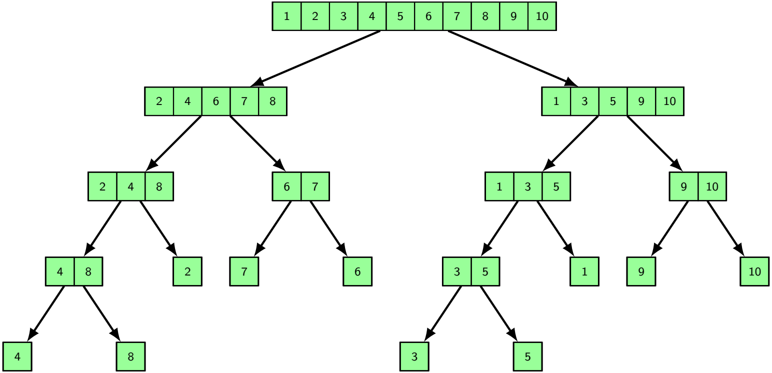
Ordenação por intercalação (*Merge Sort*)

Ordenação:

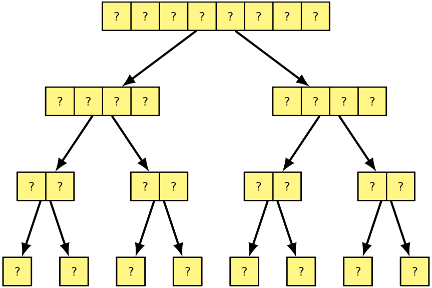
- Recebemos um `vetor` de tamanho n com limites:
  - O vetor começa na posição `vetor[l]`
  - O vetor termina na posição `vetor[r]`
- Dividimos o vetor em dois subvetores de tamanho $n/2$
- O caso base é um vetor de tamanho 0 ou 1

```c
void mergesort(int *v, int l, int r) {
  int m = (l + r) / 2;
  if (l < r) {
    /*divisão*/
    mergesort(v, l, m);
    mergesort(v, m + 1, r);
    /*conquista*/
    merge(v, l, m, r);
  }
}
```

Simulação



Tempo de execução para $n = 2^\ell$



$\leq c \cdot n$

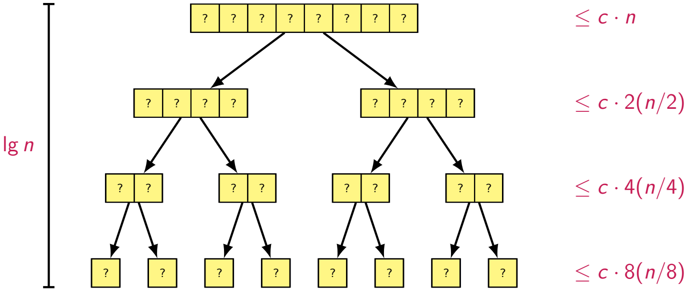
$\leq c \cdot 2(n/2)$

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

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

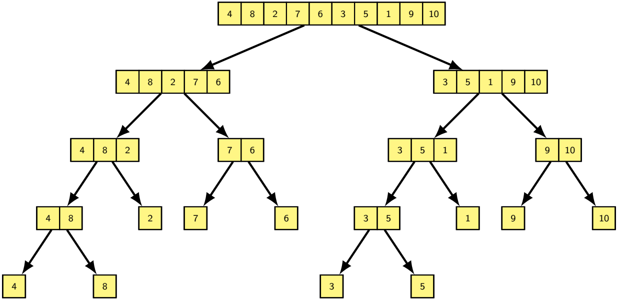
- No primeiro nível fazemos um `merge` com n elementos
- No segundo fazemos dois `merge` com $n/2$ elementos
- No $(k-1)$-ésimo fazemos 2^k `merge` com $n/2^k$ elementos
- No último gastamos tempo constante n vezes

Tempo de execução para $n = 2^\ell$



- No nível k gastamos tempo $\leq c \cdot n$
- Quantos níveis temos?
  - Dividimos n por 2 até que fique menor ou igual a 1
  - Ou seja, $\ell = \lg n$
- Tempo total: $c\, n \lg n = \mathrm{O}(n \lg n)$

Tempo de execução para n qualquer

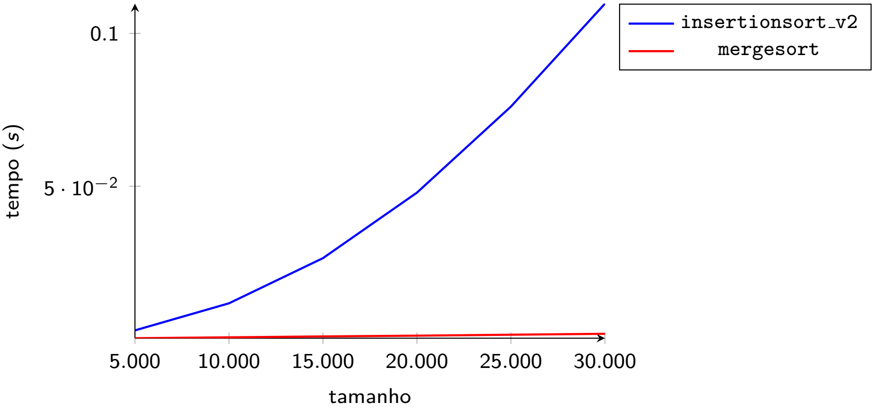


Qual o tempo de execução para n que não é potência de 2?

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

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

Gráfico de comparação dos algoritmos



`mergesort` é muito mais rápido do que o `insertionsort`

- mas precisa de memória adicional
  - tanto para o vetor auxiliar - $\mathrm{O}(n)$
  - quanto para a pilha de recursão - $\mathrm{O}(\lg n)$

Roteiro

1 Merge Sort

2 Quick Sort

3 Referências

Na unidade anterior...

Vimos três algoritmos de ordenação $\mathrm{O}(n^2)$:

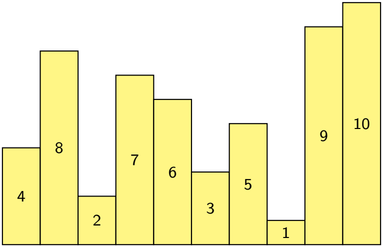
- `selectionsort`
- `bubblesort`
- `insertionsort`

E um algoritmo de ordenação $\mathrm{O}(n \lg n)$

- `mergesort`

Nessa aula veremos outro algoritmo baseado na técnica de **divisão-e-conquista**

Quicksort - Ideia



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

Quicksort

```c
int partition(int *v, int l, 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

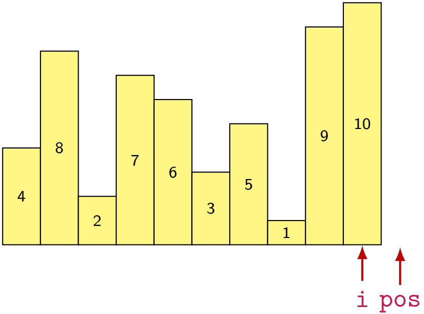
```c
void quicksort(int *v, int l, int r) {
  int i;
  if (r <= l) return;
  i = partition(v, l, r);
  quicksort(v, l, i-1);
  quicksort(v, i+1, r);
}
```

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

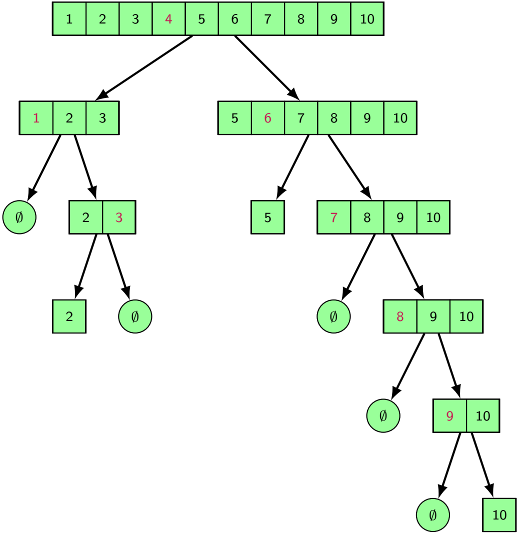
Como particionar um vetor?

- Andamos da direita para a esquerda com um índice `i`
  - De `i` até `pos - 1` ficam os menores do que o pivô
  - De `pos` até `r` ficam os maiores ou iguais ao pivô
- Sempre que o elemento em `i` for maior ou igual ao pivô
  - diminuímos `pos` e realizamos uma troca de `i` com `pos`
- No final, o pivô está em `pos`

```
int partition(int *v, int l, int r) {
  int i, pivo = v[l], pos = r + 1;
  for (i = r; i >= l; i--) {
    if (v[i] >= pivo) {
      pos--;
      troca(&v[i], &v[pos]);
    }
  }
  return pos;
}
```



Simulação do Quicksort

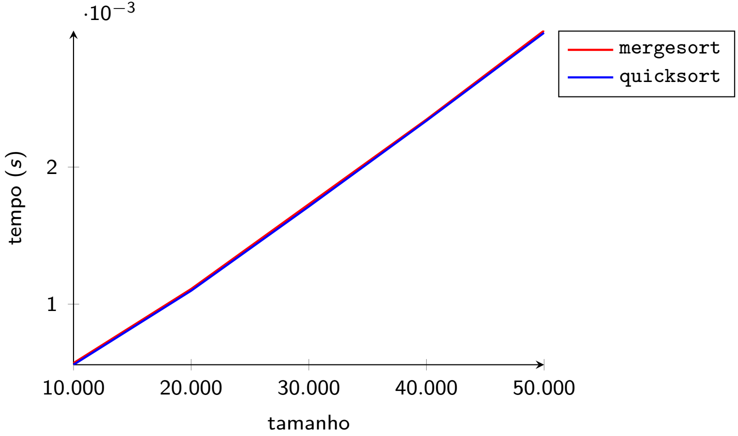


Quick-Sort

Importante:

- Note a similaridade entre o Quick-Sort e o Merge-Sort.
- Porém, o maior trabalho do Merge-Sort está na fase de conquista onde é necessário fazer a fusão.
- No Quick-Sort o maior trabalho está na fase de Divisão pois é necessário fazer um particionamento do vetor.

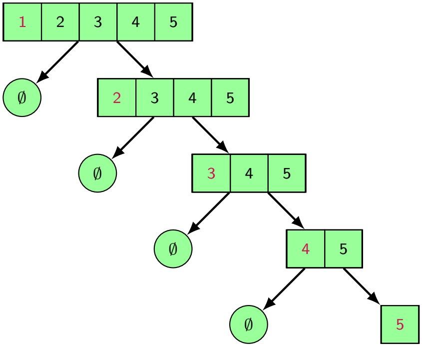
Comparação com o `mergesort` e `quicksort`



O `quicksort` foi levemente mais rápido do que o `mergesort`

- Mas ainda poderíamos otimizar o código..
- Ou seja, um poderia ficar melhor do que o outro

Pior caso do QuickSort



$c \cdot n$

$c \cdot (n-1)$

$c \cdot (n-2)$

$\cdots$

c

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

$$c \cdot n + c \cdot (n-1) + \cdots + c = c \sum_{i=0}^{n-1} (n-i) = c \sum_{j=1}^{n} j = c \frac{n(n+1)}{2} = \mathrm{O}(n^2)$$

Caso médio do QuickSort

Se o QuickSort é $\mathrm{O}(n^2)$, como ele foi melhor que o Merge-Sort no experimento?

- Se o vetor for uma permutação aleatória de n números
- então o tempo médio (esperado) do QuickSort é $\mathrm{O}(n \lg n)$
  - Nesse caso, o pivô particiona bem o vetor

Ou seja, o pior caso do QuickSort é "raro" nesse experimento

- Isso nem sempre é verdade
  - as vezes, os dados estão parcialmente ordenados
  - exemplo: inserção em blocos em um vetor ordenado

Vamos ver duas formas de mitigar esse problema

Mediana de Três

No `quicksort` escolhemos como pivô o elemento da esquerda

- Poderíamos escolher o elemento da direita ou do meio
- Melhor ainda, podemos escolher a mediana dos três
  - já que a mediana do vetor particiona ele no meio

```c
void quicksort_mdt(int *v, int l, int r) {
  int i;
  if(r <= l) return;
  troca(&v[(l+r)/2], &v[l+1]);
  if(v[l] > v[l+1])
    troca(&v[l], &v[l+1]);
  if(v[l] > v[r])
    troca(&v[l], &v[r]);
  if(v[l+1] > v[r])
    troca(&v[l+1], &v[r]);
  i = partition(v, l+1, r-1);
  quicksort_mdt(v, l, i-1);
  quicksort_mdt(v, i+1, r);
}
```

- trocamos `v[(l+r)/2]` com `v[l+1]`
- ordenamos `v[l]`, `v[l+1]` e `v[r]`
- particionamos `v[l+1]`, $\cdots$, `v[r-1]`
  - `v[l]` já é menor que o pivô
  - `v[r]` já é maior que o pivô

Quicksort Aleatorizado

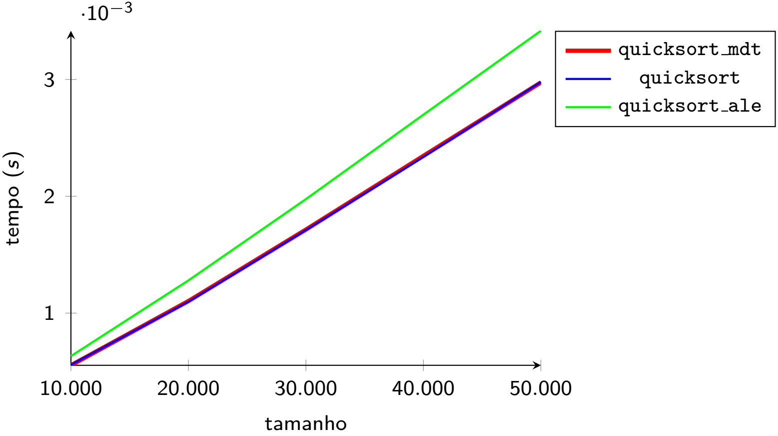
```c
int pivo_aleatorio(int l, int r) {
  //escolhe aleatoriamente um índice entre l e r
  return l + (int)((r-l+1)*(rand() / ((double)RAND_MAX + 1)));
}

void quicksort_ale(int *v, int l, int r) {
  int i;
  if(r <= l) return;
  troca(&v[pivo_aleatorio(l,r)], &v[l]);
  i = partition(v, l, r);
  quicksort_ale(v, l, i-1);
  quicksort_ale(v, i+1, r);
}
```

O tempo de execução depende dos pivôs sorteados

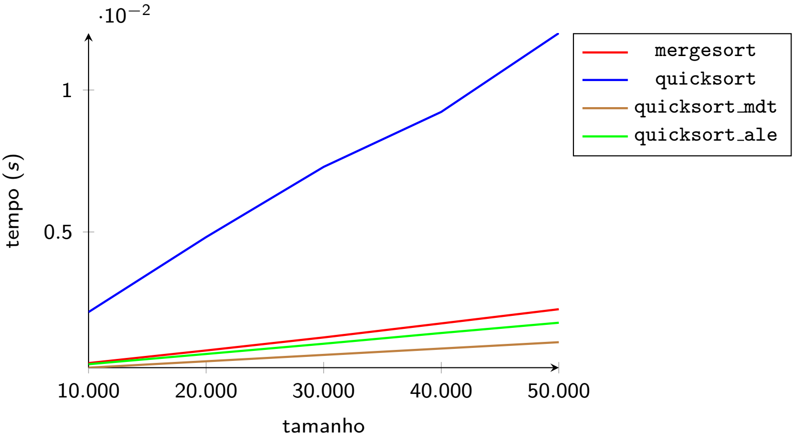
- O tempo médio é $O(n \lg n)$
  - as vezes é lento, as vezes é rápido
  - mas não depende do vetor dado

Experimentos - vetores aleatórios



`quicksort_ale` adiciona um overhead desnecessário

Experimentos - vetores quase ordenados



- `quicksort_mdt` é melhor
  - é esperado já que para vetores ordenados ele é $\mathrm{O}(n \lg n)$

Conclusão

O MergeSort é um algoritmo de ordenação $\mathrm{O}(n \lg n)$

- Melhor do que o InsertionSort, SelectionSort e BubbleSort.
- Mas precisa de espaço adicional $\mathrm{O}(n)$

O QuickSort é um algoritmo de ordenação $\mathrm{O}(n^2)$

- Mas ele pode ser rápido na prática
- Leva tempo $\mathrm{O}(n \lg n)$ (em média) para ordenar uma permutação aleatória
- Precisar de espaço adicional $\mathrm{O}(n)$ 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	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$
MergeSort	$O(n \lg n)$	$O(n \lg n)$	$O(n \lg n)$	$O(n)$
QuickSort	$O(n \lg n)$	$O(n \lg n)$	$O(n^2)$	$O(n)$

Fim

Dúvidas?

Roteiro

1. Merge Sort
2. Quick Sort
3. Referências

Referências

1. Materiais adaptados dos slides dos Profs. Rafael Schouery e Lehilton L. C. Pedrosa, da UNICAMP.