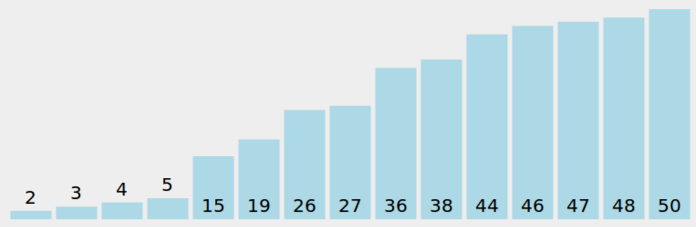
Ordenação

Pedro Ribeiro

DCC/FCUP

2021/2022



Ordenação

- A ordenação é um **passo inicial** para muitos outros algoritmos
  - Ex: encontrar a mediana
- Quando não sabes o que fazer... **ordena**!
  - Ex: encontrar repetidos fica mais fácil depois de ordenar
- **Diferentes tipos de ordenação** podem ser adequados para diferentes tipos de dados
  - Ex: para casos menos gerais, existem algoritmos lineares
- É importante conhecer as funções de ordenação disponíveis nas **bibliotecas** da vossa linguagem
  - Ex: qsort (C), STL sort (C++), Arrays.sort (Java)
  - Será um dos temas da próxima aula prática

Sobre a complexidade da ordenação

- Qual é a menor complexidade possível para um algoritmo geral de ordenação? $\boldsymbol{\Theta(n \log n)}$... mas apenas no **modelo comparativo**.
  - **Modelo comparativo:** para distinguir elementos apenas posso usar comparações $(<, >, =, \geq, \leq)$. Quantas comparações preciso?

- Um esboço da **prova** de que ordenação comparativa é $\boldsymbol{\Omega(n \log n)}$
  - Input de tamanho n tem $\mathbf{n!}$ **permutações possíveis** (apenas uma é a ordenação desejada)
  - Uma comparação tem **dois resultados posíveis** (consegue distinguir entre 2 permutações)
  - Seja $f(n)$ a função que mede o **número de comparações**
  - $f(n)$ comparações: consegue **distinguir** entre $2^{f(n)}$ permutações
  - Precisamos que $2^{f(n)} \geq n!$, ou seja, $\mathbf{f(n) \geq \log_2(n!)}$
  - Usando a **aproximação de Stirling**, sabemos que $\mathbf{f(n) \geq n \log_2 n}$

Alguns algoritmos de ordenação

- **Algoritmos Comparativos**
  - **BubbleSort** (trocar elementos)
  - **SelectionSort** (seleccionar o maior/menor)
  - **InsertionSort** (inserir na posição correta)
  - **MergeSort** (dividir em dois, ordenar metades e depois juntar)
  - **QuickSort** "naive" (dividir segundo um pivot e ordenar)
  - **QuickSort** "aleatorizado" (escolher pivot de forma aleatória)

- **Algoritmos Não Comparativos**
  - **CountingSort** (contar nº de elementos de cada tipo)
  - **RadixSort** (ordenar segundo os "dígitos")

Alguns algoritmos de ordenação

Existem muitos mais!

Exchange sorts	Bubble sort · Cocktail shaker sort · Odd–even sort · Comb sort · Gnome sort · Quicksort · Slowsort · Stooge sort · Bogosort
Selection sorts	Selection sort · Heapsort · Smoothsort · Cartesian tree sort · Tournament sort · Cycle sort · Weak-heap sort
Insertion sorts	Insertion sort · Shellsort · Splaysort · Tree sort · Library sort · Patience sorting
Merge sorts	Merge sort · Cascade merge sort · Oscillating merge sort · Polyphase merge sort
Distribution sorts	American flag sort · Bead sort · Bucket sort · Burstsort · Counting sort · Interpolation sort · Pigeonhole sort · Proxmap sort · Radix sort · Flashsort
Concurrent sorts	Bitonic sorter · Batcher odd–even mergesort · Pairwise sorting network · Samplesort
Hybrid sorts	Block merge sort · Kirkpatrick-Reisch sort · Timsort · Introsort · Spreadsort · Merge-insertion sort
Other	Topological sorting (Pre-topological order) · Pancake sorting · Spaghetti sort

(fonte da imagem: http://en.wikipedia.org/wiki/Sorting_algorithm)

Algumas considerações gerais

- Para os próximos slides vamos **assumir o seguinte**:
  - Queremos ordenar por **ordem crescente**
  - Estamos a ordenar por um conjunto de **n** items
  - Os items estão guardados num array **v[n]** (nas posições $0..n-1$)
  - Os items são **comparáveis** (através de $<, >, =, \geq, \leq$)

BubbleSort

- **Ideia-chave:** trocar elementos que estão fora de posição

Código para BubbleSort

Fazer
    $existem\_trocas \leftarrow false$
    Para $i \leftarrow 1$ **até** $n-1$ **fazer**
        Se $v[i-1] > v[i]$ **então**
            Trocar $v[i-1]$ com $v[i]$
            $existem\_trocas \leftarrow verdadeiro$
Enquanto ($existem\_trocas$)

Vamos ver uma animação no VisuAlgo

BubbleSort

- Melhorar não indo sempre até à última posição

Código para BubbleSort - v2

Fazer
    $existem\_trocas \leftarrow false$
    Para $i \leftarrow 1$ **até** $n - 1$ **fazer**
        Se $v[i-1] > v[i]$ **então**
            Trocar $v[i-1]$ com $v[i]$
            $existem\_trocas \leftarrow verdadeiro$
    $n - -$
Enquanto ($existem\_trocas$)

BubbleSort

- Melhorar indo até à última posição em que houve troca

Código para BubbleSort -v3

Fazer
  $ultima\_posicao \leftarrow 0$
  Para $i \leftarrow 1$ **até** $n-1$ **fazer**
    Se $v[i-1] > v[i]$ **então**
      Trocar $v[i-1]$ com $v[i]$
      $ultima\_posicao \leftarrow i$
  $n \leftarrow ultima\_posicao$
Enquanto $(n > 0)$

- Nenhuma das alterações/optimizações mexeu no pior caso: $\mathcal{O}(\mathbf{n^2})$

SelectionSort

- **Ideia-chave:** escolher o mínimo e colocar na posição dele

Código para SelectionSort

Para $i \leftarrow 0$ **até** $n-2$ **fazer**
    $pos\_min \leftarrow i$ (posição do menor elemento)
    Para $j \leftarrow i+1$ **até** $n-1$ **fazer**
        Se $v[j] < v[pos\_min]$ **então**
            $pos\_min \leftarrow j$
    Trocar $v[i]$ com $v[pos\_min]$

Vamos ver uma animação no VisuAlgo

- Tem complexidade $\Theta(n^2)$

InsertionSort

- **Ideia-chave:** inserir cada elemento na sua posição correta

Código para InsertionSort

Para $i \leftarrow 1$ **até** $n-1$ **fazer**
    $x \leftarrow v[i]$ (elemento que vamos inserir)
    $j \leftarrow i$
    Enquanto $j > 0$ e $v[j-1] > x$ **fazer**
        $v[j] \leftarrow v[j-1]$
        $j--$
    $v[j] \leftarrow x$

Vamos ver uma animação no VisuAlgo

- Tendo em conta o pior caso: $\mathcal{O}(\mathbf{n^2})$

MergeSort

- **Ideia-chave:** dividir em dois, ordenar metades e depois juntá-las
- Já vimos este algoritmo em detalhe anteriormente:

MergeSort com Dividir para Conquistar

Dividir: partir o array inicial em 2 arrays com metade do tamanho inicial

Conquistar: ordenar recursivamente as 2 metades. Se o problema for ordenar um array de apenas 1 elemento, basta devolvê-lo.

Combinar: fazer uma junção (*merge*) das duas metades ordenadas para um array final ordenado.

Vamos ver uma animação no VisuAlgo

- Tem complexidade $\Theta(n \log n)$

QuickSort (naive)

- **Ideia-chave:** dividir segundo um pivot e ordenar recursivamente

QuickSort (naive)

1. Escolher um elemento (primeiro, por ex.) como sendo o pivot
2. Partir o array em dois: elementos menores do que pivot e elementos maiores do que o pivot
3. Ordenar recursivamente cada uma das duas partições

Vamos ver uma animação no VisuAlgo

- A escolha do pivot é determinante
- Se a escolha "dividir" bem o algoritmo demora $n \log n$
- No pior caso, no entanto... $\mathbf{\Theta(n^2)}$

QuickSort (aleatorizado)

- **Ideia-chave:** dividir segundo um pivot e ordenar recursivamente

QuickSort (aleatorizado)

1. Escolher aleatoriamente um elemento como sendo o pivot
2. Partir o array em dois: elementos menores do que pivot e elementos maiores do que o pivot
3. Ordenar recursivamente cada uma das duas partições

Vamos ver uma animação no VisuAlgo

- Em média demora $n \log n$
- Não conseguimos arranjar um caso que obrigue (sempre) a $\mathbf{n^2}$!

Algoritmos Não Comparativos

- Para simplificar vamos assumir que os **items são números**
- Ideia pode ser **generalizada** para outros tipos de dados

CountingSort

- **Ideia-chave:** Contar número de elementos de cada "tamanho"

CountingSort

$conta[max\_tamanho]$ ← `array para contagem`
Para $i \leftarrow 0$ **até** $n-1$ **fazer**
    $conta[v[i]]++$ `(mais um elemento v[i])`
$i = 0$
Para $j \leftarrow min\_tamanho$ **até** $max\_tamanho$ **fazer**
    Enquanto $conta[j] > 0$ **fazer**
        $v[i] \leftarrow j$ `(coloca elemento no array)`
        $conta[j]--$ `(menos um elemento desse tamanho)`
        $i++$ `(incrementa posição a colocar no array)`

Vamos ver uma animação no VisuAlgo

- Seja k o maior número
- Vamos demorar $\mathcal{O}(\mathbf{n}+\mathbf{k})$

RadixSort

- **Ideia-chave:** Ordenar dígito a dígito

Um possível RadixSort (começando no dígito menos significativo)

$bucket[10]$ $\leftarrow$ array de listas de números (um por dígito)
Para pos $\leftarrow$ 1 **até** $max\_numero\_digitos$ **fazer**
    Para i $\leftarrow$ 0 **até** $n-1$ **fazer** (para cada número)
        Colocar $v[i]$ em $bucket[digito\_posicao\_pos(v[i])]$
    Para i $\leftarrow$ 0 **até** 9 **fazer** (para cada dígito possível)
        Enquanto $tamanho(bucket[i]) > 0$ **fazer**
            Retirar 1º número de $bucket[i]$ e adicioná-lo a $v[]$

Vamos ver uma animação no VisuAlgo

- Seja k o maior número de dígitos de um número
- Vamos demorar $\mathcal{O}(\mathbf{k} \times \mathbf{n})$

Uma visão global

- Existem **muitos** algoritmos de ordenação
- O **"melhor"** algoritmo depende do caso em questão
- É possível **combinar** vários algoritmos (híbridos)
  - Ex: RadixSort pode ter como passo interno um outro algoritmo, desde que seja um **stable sort** (em caso de empate, manter ordem inicial)
- Na prática, em **implementações reais**, é isso que é feito (combinar): (Nota: implementação depende do compilador e da sua versão)
  - **Java:** usa **Timsort** (MergeSort + InsertionSort)
  - **C++ STL**: usa **IntroSort** (QuickSort + HeapSort) + InsertionSort

Problema: encontrar elementos **repetidos**

Input

```
 9 21 27 38 34 53 19 38 43
51  1  9 10 39 50  6 26 44
 5 32 16 20 50 22 41 30 39
 3 32 30 31 40 50 56 13 19
46 32 56 26 20 57 32 27 31
17 32 54 61 34 22 14 54  9
34 30 38 10 30  5 37 61 44
```

Input

```
 1| 3| 5  5| 6| 9  9  9|10
10|13|14|16|17|19 19|20 20|
21|22 22|26 26|27 27|30 30
30 30|31 31|32 32 32 32 32|
34 34 34|37|38 38 38|39 39|
40|41|43|44 44|46|50 50 50|
51|53|54 54|56 56|57|61 61
```

Elementos iguais ficam juntos!

Exemplos de Aplicações de Ordenação

Vários

Problema: encontrar **frequência** de elementos
(ordenar e elementos ficam juntos)

Problema: encontrar par de números **mais próximo**
(ordenar e ver diferenças entre números consecutivos)

Problema: encontrar k-**ésimo** número
(ordenar e ver posição k)

Problema: seleccionar o **top**-k
(ordenar e ver os primeiros k)

Problema: **união** de conjuntos
(ordenar e juntar - parecido com o "merge")

Problema: **intersecção** de conjuntos
(ordenar e percorrer - parecido com o "merge")

Exemplos de Aplicações de Ordenação

Anagramas

Problema: Descobrir anagramas
(palavras/conjuntos de palavras que usam as mesmas letras)

Exemplos:

- **amor**, **ramo**, **mora**, **Roma** [amor]
- **Ricardo**, **criador** e **corrida** [acdiorr]
- **algoritmo** e **logaritmo** [agilmoort]
- **Tom Marvolo Riddle** e **I am Lord Voldemort** [addeillmmooorrtv]
- **Clint Eastwood** e **Old West action** [acdeilnoosttw]

Problema: Pesquisar elementos em arrays ordenados

Pesquisa Binária - $\Theta(\log n)$

Pesquisa Binária

Um definição

Pesquisa binária num array ordenado (*bsearch*)

Input:

- um array **v**[] de **n** números ordenados de forma crescente
- uma chave **key** a procurar

Output:

- **Posição** da *key* no array *v*[] (se número existir)
- **-1** (se número não for encontrado)

Exemplo:

v =

2	5	6	8	9	12

bsearch(v, 2) = 0
bsearch(v, 4) = -1
bsearch(v, 8) = 3
bsearch(v, 14) = -1

Pesquisa binária num array ordenado

```
bsearch(v, low, high, key)
    Enquanto (low ≤ high ) fazer
        middle ← low + (high − low)/2
        Se (key = v[middle])              retorna(middle)
        Senão se (key < v[middle])        high ← middle − 1
        Senão                             low ← middle + 1
    retorna(-1)
```

v =

2	5	6	8	9	12

bsearch(v, 0, 5, 8)

$low = 0$, $high = 5$, $middle = 2$

Como $8 > v[2]$: $low = 3$, $high = 5$, $middle = 4$

Como $8 < v[4]$: $low = 3$, $high = 3$, $middle = 3$

Como $8 = v[3]$: **retorna(3)**

Podemos generalizar a **pesquisa binária** para casos onde temos algo como:

não	não	não	não	não	sim	sim	sim	sim	sim	sim

Queremos encontrar o **primeiro sim** (ou nalguns casos o **último não**)

Exemplo:

- Procurar menor número maior ou igual a *key* (**lower_bound** do C++)

2	5	6	8	9	12
não	não	não	sim	sim	sim

lower_bound(7) $\rightarrow$ condição: $v[i] >= 7$
[o menor número maior que 7 neste array é o 8]

Pesquisa binária para condição *condicao*

bsearch(low, high, condicao)
- **Enquanto** $(low < high)$ **fazer**
  - $middle \leftarrow low + (high - low)/2$
  - **Se** $(condicao(middle) = sim)$ $high \leftarrow middle$
  - **Senão** $low \leftarrow middle + 1$
- **Se** $(condicao(low) = nao)$ retorna(-1)
- **retorna**(low)

v =

2	5	6	8	9	12
não	não	não	sim	sim	sim

bsearch(0, 5, $\geq$ 7)

$low = 0$, $high = 5$, $middle = 2$

Como $v[2] \geq 7$ é não: $low = 3$, $high = 5$, $middle = 4$

Como $v[4] \geq 7$ é sim: $low = 3$, $high = 4$, $middle = 3$

Como $v[3] \geq 7$ é sim: $low = 3$, $high = 3$ (sai do while)

Como $v[3] \geq 7$ é sim: **retorna(3)**

Problema da partição equilibrada

Input: uma sequência $\langle a_1, \ldots, a_n \rangle$ de n inteiros positivos e um inteiro k
Output: uma maneira de partir a sequência em k subsequências contíguas, minimizando a soma da maior partição

Exemplo:
7 9 3 8 2 2 9 4 3 4 7 9 9 $\qquad k = 4$ (4 partições)

7 9 3|8 2 2|9 4 3|4 7 9 9 $\rightarrow$ 19 + 12 + 16 + 29
7 9 3 8|2 2 9|4 3 4 7|9 9 $\rightarrow$ 27 + 13 + 18 + 18
7 9|3 8 2 2|9 4 3 4|7 9 9 $\rightarrow$ 16 + 15 + 20 + 25
...

Qual a melhor (com menor máximo)?

- Pesquisa exaustiva teria de testar todas as partições possíveis! (conseguem estimar quantas são?)
- Noutra aula voltaremos eventualmente a este problema para resolver com **programação dinâmica**
- Nesta aula vamos resolver com... **pesquisa binária!**

Pesquisa Binária

Um exemplo diferente - Partição Equilibrada

Vamos pensar num problema "parecido":

É possível criar alocação onde soma da maior partição seja $\leq X$?

Ideia *"greedy"*: ir estendendo partição enquanto soma for menor que X!

Exemplos:

Seja $X = 21$ e $k = 4$

7 9 3|8 2 2 9 4 3 4 7 9 9

7 9 3|8 2 2 9|4 3 4 7 9 9

7 9 3|8 2 2 9|4 3 4 7|9 9

7 9 3|8 2 2 9|4 3 4 7|9 9 - OK!

Seja $X = 20$ e $k = 4$

7 9 3|8 2 2 9 4 3 4 7 9 9

7 9 3|8 2 2|9 4 3 4 7 9 9

7 9 3|8 2 2|9 4 3 4|7 9 9

7 9 3|8 2 2|9 4 3 4|7 9|9 - Falhou! Precisava de mais do que 4 partições

É possível criar partição onde soma da maior partição seja $\leq X$?

Se pensarmos nos X para os quais a resposta é sim, temos um espaço de procura onde acontece:

não	não	...	não	não	sim	sim	sim	...	sim	sim

Posso aplicar **pesquisa binária no X**!

- Seja s a soma de todos os números
- No mínimo X será 1 (ou em alternativa o maior a_i)
- No máximo X será s
- Verificar resposta para um dado X: $\Theta(n)$
- Pesquisa binária em X: $\Theta(\log s)$
- Tempo global: $\Theta(n \log s)$

Exemplo: 7 9 3 8 2 2 9 4 3 4 7 9 9 $\qquad k = 4$ (4 partições)

low = 1, high = 76, middle = 38 $\rightarrow$ é_possível(38)? Sim
low = 1, high = 38, middle = 19 $\rightarrow$ é_possível(19)? Não
low = 20, high = 38, middle = 29 $\rightarrow$ é_possível(29)? Sim
low = 20, high = 29, middle = 24 $\rightarrow$ é_possível(24)? Sim
low = 20, high = 24, middle = 22 $\rightarrow$ é_possível(22)? Sim
low = 20, high = 22, middle = 21 $\rightarrow$ é_possível(21)? Sim
low = 20, high = 21, middle = 20 $\rightarrow$ é_possível(20)? Não
low = 21, high = 21

Sai do ciclo e verifica que **é_possível(21)**, sendo essa a resposta!

7 9 3|8 2 2 9|4 3 4 7|9 9 $\rightarrow 19 + 21 + 18 + 18$

2º Exemplo: 7 9 3 8 2 2 9 4 3 4 7 9 9 $k = 3$ (3 partições)

low = 1, high = 76, middle = 38 → é_possível(38)?Sim
low = 1, high = 38, middle = 19 → é_possível(19)?Não
low = 20, high = 38, middle = 29 → é_possível(29)?Sim
low = 20, high = 29, middle = 24 → é_possível(24)?Não
low = 25, high = 29, middle = 27 → é_possível(27)?Sim
low = 25, high = 27, middle = 26 → é_possível(26)?Não
low = 27, high = 27

Sai do ciclo e verifica que **é_possível(27)**, sendo essa a resposta!

7 9 3 8|2 2 9 4 3 4|7 9 9 → 27 + 24 + 25

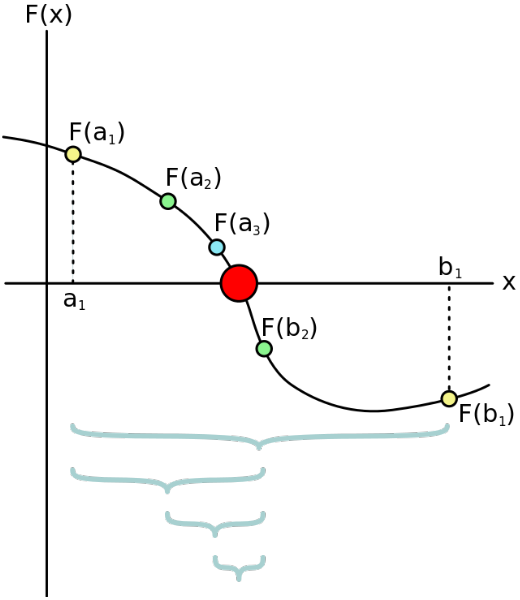
Método da Bisseção

Uma ideia semelhante a pesquisa binária pode ser usada para encontrar **raízes de funções**

- Seja $f(n)$ uma função **contínua** definida num intervalo $[\mathbf{a}, \mathbf{b}]$ e onde $f(a)$ e $f(b)$ têm **sinais opostos**
- $f(n)$ tem de ter **pelo menos uma raíz** no intervalo $[a, b]$

- Começando em $[a, b]$, ver o **ponto médio** c e consoante o sinal de $f(c)$ **reduzir o intervalo** a $[a, c]$ ou $[c, b]$

Método da Bisseção



(imagem da Wikipedia)

Método da Bisseção

Exemplo: $\mathbf{f(x) = x^3 - x - 2}$

(1) Encontrar um *a* e um *b* com sinais opostos:

$f(\mathbf{1}) = 1^3 - 1 - 2 = -2$ $\qquad$ $f(\mathbf{2}) = 2^3 - 2 - 2 = 4$

(2) Fazer divisões sucessivas

#	a	b	c	f(c)
1	1.0	2.0	1.5	-0.125
2	1.5	2.0	1.75	1.6093750
3	1.5	1.75	1.625	0.6660156
4	1.5	1.625	1.5625	0.2521973
5	1.5	1.5625	1.5312500	0.0591125
6	1.5	1.5312500	1.5156250	-0.0340538
7	1.5156250	1.5312500	1.5234375	0.0122504
8	1.5156250	1.5234375	1.5195313	-0.0109712
9	1.5195313	1.5234375	1.5214844	0.0006222
10	1.5195313	1.5214844	1.5205078	-0.0051789
11	1.5205078	1.5214844	1.5209961	-0.0022794
12	1.5209961	1.5214844	1.5212402	-0.0008289
13	1.5212402	1.5214844	1.5213623	-0.0001034

Método da Bisseção

- Parar quando atingir **precisão definida**
  ou
- Parar quando atingir um certo **número de iterações**
- Existem outros métodos que **convergem mais rapidamente**
  - Método de Newton
  - Método das Secantes
- Um **exemplo de problema** que podia ser resolvido com isto:
  Qual o maior n para o qual uma função $f(n)$ demora menos que tempo t, assumindo tempo op de cada operação ?
  $\mathbf{f(n) * op = t \iff f(n) * op - t = 0}$

  Ex: $n! * 10^{-8} - 60 = 0$
  (maior n para 1 minuto de $\Theta(n!)$ assumindo cada op. demorar 10^{-8})

Pesquisa Binária

- Pesquisa binária é muito **útil** e **flexível**
- Pode ser usado num **vasto leque de aplicações**
- Existem muitas outras **variações**, para além das faladas.
  - Pesquisa binária interpolada
    (em vez de ir para o meio, estimar posição)
  - Pesquisa (binária) exponencial
    (Começar por tentar fixar intervalo em $low = 2^a$ e $high = 2^{a+1}$)
  - Pesquisa ternária
    (máximo ou mínimo em função unimodal)
  - ...