iscte
INSTITUTO UNIVERSITÁRIO DE LISBOA

iscte
TECNOLOGIAS
E ARQUITETURA

Desenho e Análise de Algoritmos

Notas - Aulas TP 5 e 6

Análise de complexidade de algotitmos
Continuação

Ana Maria de Almeida

DCTI

2022/2023

1

iscte

"Our lives begin to end the day we become silent about things that matter."

- Martin Luther King

2

Plano para esta aula

- Análise de Algoritmos
- Algoritmos recursivos
- Outras estratégias de desenho de algoritmos

Nas últimas aulas:

- Introdução à Análise de Algoritmos:
  - as 3 questões essenciais:
    - Correção do algoritmo
    - Ordem de complexidade
    - Melhoramentos
  - análise de complexidade
  - análise no pior dos casos
  - estratégia *dividir-e-conquistar* recursiva
  - limites assintóticos

- **Estratégia dividir-e-conquistar:** esta estratégia recorre a 3 passos em cada nível recorrente:
  1. **Dividir** o problema em **k** subproblemas **iguais mas de menor dimensão**.
  2. **Enquanto não atingir** um **caso base,** resolver cada subproblema através da aplicação recursiva da divisão.
  3. **Combinar** as k soluções dos subproblemas na solução do problema da chamada anterior.

Sobre os limites assintóticos (cormen 3.2)

- Limite superior - $f(n) = \mathcal{O}(g(n))$
- Limite inferior - $f(n) = \Omega(g(n))$
- Limite exato - $f(n) = \Theta(g(n))$

Transitivity:

$$f(n) = \Theta(g(n)) \text{ and } g(n) = \Theta(h(n)) \text{ imply } f(n) = \Theta(h(n)),$$
$$f(n) = O(g(n)) \text{ and } g(n) = O(h(n)) \text{ imply } f(n) = O(h(n)),$$
$$f(n) = \Omega(g(n)) \text{ and } g(n) = \Omega(h(n)) \text{ imply } f(n) = \Omega(h(n)).$$

Reflexivity:

$$f(n) = \Theta(f(n)),$$
$$f(n) = O(f(n)),$$
$$f(n) = \Omega(f(n)).$$

Symmetry:

$$f(n) = \Theta(g(n)) \text{ if and only if } g(n) = \Theta(f(n)).$$

Transpose symmetry:

$$f(n) = O(g(n)) \text{ if and only if } g(n) = \Omega(f(n)).$$

Algoritmo: 3 questões essenciais

Relembrar da aula anterior: quando em face de um novo algoritmo, existem 3 questões que qualquer *developer* deve colocar antes de se dar por satisfeito:

1. **Este algoritmo está (sempre) correto?**
2. **Quanto tempo vai demorar a resolver (em função do tamanho do input)?**
3. ***Que podemos fazer para melhorar?***

O algoritmo está correto?

- **Problema genérico da ordenação** (ordem crescente):
  - Dados: uma sequência de objetos de valor ordenável: $a_1, a_2, ..., a_n$
  - Resultados: a permutação da sequência que os ordena por valor: $a_1 \leq a_2 \leq ... \leq a_n$

Inserção linear (Insertion sort)

```
Insertionsort(a, n):
    //Ordena uma sequência por ordem crescente
    //Input: sequência a ordenar e inteiro não negativo, n
    //Output: sequência ordenada

    for j = 2, …, n:
        este = a[j]
        i = j-1
        while i > 0 and a[i] > este:
            a[i+1] = a[i]
            i = i-1
        endwhile
        a[i+1] = este
    endfor
```

2022/2023

7

Análise de correção

- É necessária para exibirmos uma garantia (prova) que o nosso algoritmo vai sempre devolver uma resposta correta para qualquer input.
- Pode ser desafiante, mas é importante como medida de segurança.

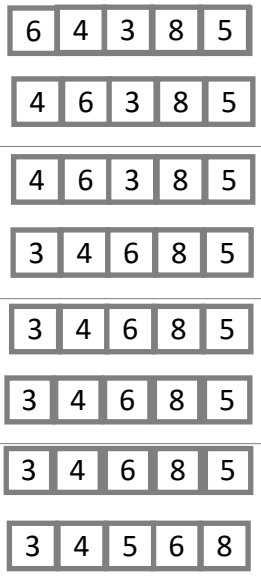
Como fazer a correção?

- **Na maioria dos casos**, podemos usar o **método de indução**, que nos permite generalizar um passo/conjunto de passos chave na construção do nosso algoritmo para qualquer input.
- Vamos ilustrar um modo de analisar um algoritmo e detetar o tal "passo chave" usando o método de Inserção Linear.

Source: Mary Wooters

9

Análise à construção da resposta de um algoritmo



O primeiro elemento, 6, sózinho, é uma lista ordenada, [6].

Logo, inserindo 4 na lista [6] na posição correta torna [4,6] uma lista ordenada.

Inserir 3 na posição correta em [4,6] torna [3,4,6] uma lista ordenada.

A inserção, na posição correta, de 8 na lista [3,4,6] devolve a lista ordenada [3,4,6,8].

A inserção, na posição correta, de 5 na lista [3,4,6,8] devolve a lista ordenada [3,4,5,6,8].

10

Generalizar

- O que acabámos de fazer foi detetar uma propriedade (**invariante**) de construção da resposta final do algoritmo.
- Para generalizar, é necessário formular essa propriedade/invariante matematicamente.
- E usar uma técnica de prova construtiva para mostrar que, para qualquer input, a construção funciona (sempre) corretamente.
- Neste caso, vamos usar a técnica de **Prova por Indução.**

11

Esquema de prova de correção do Insertionsort

Considere que A é uma sequência de tamanho n.

- **Hipótese de Indução**:

  No final da i-ésima iteração (*do for exterior*), A[1..i] está ordenada (para i ≥ 1)

- **Caso Base** (i=1, ou seja, considereamos a primeira posição da sequência):

  A[1] está ordenada.

- **Passo Indutivo**:

  Para todo o 2 < k < n, se a hipótese de indução é verdadeira até **k-1**, então é verdadeira para **i=k**.

- **Conclusão**:
  - A hipótese de indução é verdadeira para i ≥ 1.
  - Em particular, é verdade para i=n.
  - Logo, no final do algoritmo, a sequência A[1 .. n] está ordenada.

Desempenho do InsertionSort: tempo

- Os testes ao tempo de execução são úteis.

- **Contras**:
  - O "mesmo" algoritmo pode mostrar-se mais rápido ou mais lento em função da implementação.
  - Para a mesma implementação, pode ser mais rápido ou mais lento dependendo do hardware utilizado.

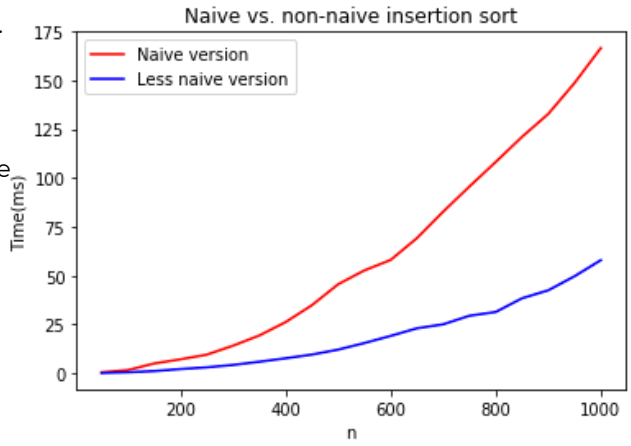


Gráfico do exemplo do notebook da aula anterior

Análise assintótica da Inserção Linear

```
Insertionsort(a, n):
    //Ordena uma sequência por ordem crescen
    //Input: sequência a ordenar e inteiro não ne
    //Output: sequência ordenada

    for j = 2, …, n:
     (i)  este = a[j]
     (ii)  i = j-1
     (iii) while i > 0 and a[i] > este:
              a[i+1] = a[i]
              i = i-1
           endwhile
     (iv) a[i+1] = este
    endfor
```

 # passos elementares explícitos:

- (i), (ii), (iii),(iv): uma vez em cada volta do for temos ___ atribuições com acesso à sequência, ___ somas e até ___ comparações
- ainda (iii): seja t_j o número de vezes que o teste do ciclo while é true para o índice j; em cada iteração, temos _____ _____.
- O tempo (ou o número de passos elementares) será:

$$T(n) = 7\,(n-1) + 3 \times \sum_2^n t_j$$

No que se segue, não usamos as constantes de implementação de operações elementares (K e C) que usámos no exemplo da aula anterior.

Análise da Inserção Linear - casos

- Melhor dos casos (*Best case*) – $B(n)$:

  a sequência já está ordenada e o corpo do while nunca é executado, ou seja, $t_j =$:

  $B(n) = 7\,(n-1) + 3 \times$, logo, $B(n) = O(\ )$

- Pior dos casos (*Worst case*) – $W(n)$:

  a sequência está ordenada por ordem inversa, logo, cada novo elemento tem de ser colocado no ínicio (1.ª posição), i.e, $t_j =$:

  $W(n) = 7\,(n-1) + 3 \times \sum_2^n$ =

  logo, $W(n) = O(\ )$

Análise da Inserção Linear - casos

- Caso médio (*Average case*) – $\boldsymbol{A(n)}$:

Assume-se uma distribuição uniforme, ou seja, a possibilidade de ocorrer desordenamento numa posição é de 50% e, nesse caso, terá de recuar em metade das posições atrás. Por excesso, temos:

$$W(n) = 7\,(n-1) + 4 \times \sum_2^n j/2 = 7n - 7 + 4 \times \frac{\left(\frac{n}{2}-1\right)^2}{2} = n^2 + 5\,n + 5$$

Depois dos cálculos, vê-se que também este caso vai ser : $\boldsymbol{A(n) = O(n^2)}$

Conclusão: A ordem geral do Insertion sort é $O(n^2)$.

De volta às 3 questões essenciais

- **Correção**: "Está correcto?"

  Fazer um estudo orientado para a análise do pior dos casos (ou seja, uma análise suficientemente geral para aplicar a qualquer input (ou seja, qualquer caso) - Worst-case Analysis – Prova por Indução

- **Análise de eficiência**: "É rápido?"

  Fazer a análise assintótica suficientemente geral. i.e., análise do pior dos casos (**se necessário**, estudar cada um dos casos -pior, melhor, médio- em particular)

- **Otimização**: "Podemos melhorar"?
  - Otimizar, mas manter o mesmo algoritmo (em geral, não altera a ordem de eficiência, mas torna a execução típica menos pesada ☺)
  - Para alterar a ordem assintótica é necessário pensar "fora da caixa"; alterar a estratégia; modificar o fluxo de instruções; ...

Desenho de algoritmos

Estratégias de construção de algoritmos:

- exaustiva
- incremental
- "diminuir-e-conquistar"

Estratégia algorítmica: **Pesquisa exaustiva**

- **Pesquisa exaustiva** ("*Brute force*"): Para alguns (muitos) problemas não triviais, existe um algoritmo de pesquisa de solução simples – procurar **entre todas as soluções possíveis** qual a usar (e devolver).
- Tipicamente, para um input de tamanho n, realiza cerca de 2^n operações (ou mais!).
- O que é, computacionalmente, inaceitável (ou não tratável).

source: Tàrdos & Kleinberg, cs.princeton.edu.

Tempo de execução (da ordem) polinomial

- Na prática, consideramos um algoritmo **eficiente** se tem um pior dos casos (limite superior) da ordem polinomial.
- Mas polinomial com constantes e expoentes limitados, no pior dos casos.

Map graphs in polynomial time

Mikkel Thorup*
Department of Computer Science, University of Copenhagen
Universitetsparken 1, DK-2100 Copenhagen East, Denmark
mthorup@diku.dk

Abstract

Chen, Grigni, and Papadimitriou (WADS'97 and STOC'98) have introduced a modified notion of planarity, where two faces are considered adjacent if they share at least one point. The corresponding abstract graphs are called map graphs. *Chen et al. raised the question of whether map graphs can be recognized in polynomial time. They showed that the decision problem is in NP and presented a polynomial time algorithm for the special case where we allow at most 4 faces to intersect in any point — if only 3 are allowed to intersect in a point, we get the usual planar graphs.*

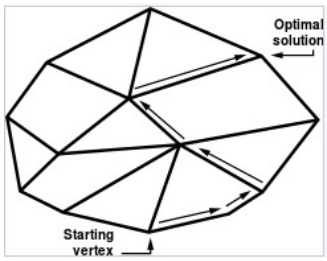
Chen et al. conjectured that map graphs can be recognized in polynomial time, and in this paper, their conjecture is settled affirmatively.

polinomial

algoritmo $O(n^{120})$!!!!

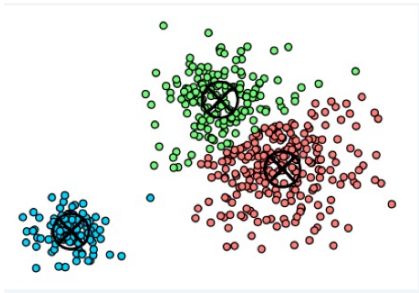
Tempo de execução exponencial no pior dos casos

- Mas existem exceções: alguns algoritmos têm tempo, no pior dos casos, exponencial mas são usados na prática
- Porque as instâncias que levam ao pior dos casos não (se espera que) surjam!



simplex algorithm

https://en.wikipedia.org/wiki/Simplex_algorithm



k-means clustering algorithm

https://en.wikipedia.org/wiki/K-means_clustering

Qual é a estratégia algorítmica nestes dois casos?

```
Insertionsort(a, n):
    //Ordena uma sequência por ordem crescente
    //Input: sequência a ordenar e inteiro não negativo, n
    //Output: sequência ordenada

   for j = 2, …, n:
        este = a[j]
        i = j-1
       while i > 0 and a[i] > este:
             a[i+1] = a[i]
             i = i-1
       endwhile
       a[i+1] =  este
   endfor
```

Inserção linear (Insertion sort)

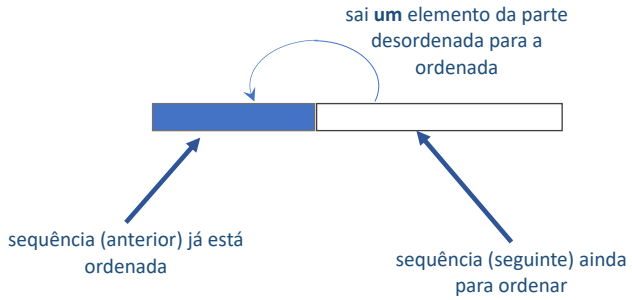
```
Selectionsort(a, n):
    //Ordena uma sequência por ordem crescente
    //Input: sequência a ordenar e inteiro não negativo, n
    //Output: sequência ordenada

    for i = 1, …, n:
        minimo = i
       for j = i+1, …, n:
           if a[j] < a[minimo]
               minimo = j
       endfor
       if i ≠ minimo:
           troca(a[i], a[minimo])
   endfor
```

Seleção linear (Selection sort)

23

Estratégia para desenho de algoritmos: **"Diminuir – e – Conquistar"**

- Quer a seleção linear, quer a inserção linear, seguem, na construção de solução/resultado, uma abordagem direta: a da **estratégia incremental**

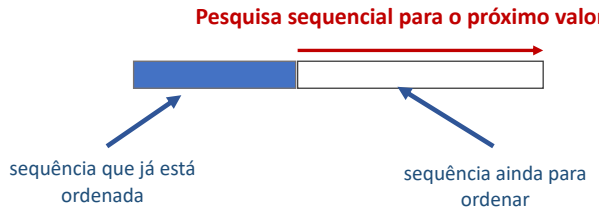


- Ambos os métodos de ordenamento usam a estratégia de resolver um problema, resolvendo instâncias sucessivamente menores num tamanho constante (em ambos os casos, um passo de cada vez)
- **Técnica "decrease and conquer"**: usa uma relação que compõe a resolução de um problema de dimensão n usando a solução de um problema de menor dimensão (num valor constante k)

24

Estratégias para desenho de algoritmos: **Pesquisa exaustiva**

- No método da seleção linear**,** a construção do ordenamento, a estratégia incremental é conseguida **após uma pesquisa sequencial** para determinar qual o elemento a colocar na próxima posição



- **Também conhecido por Método da força bruta!**

Voltando aos

Algoritmos de ordenação

continuação

Algoritmos de ordenamento: **3.ª questão essencial**

Revimos já o insertionsort e selection sort, que vimos serem ambos da ordem geral de complexidade quadrática.

- **Podemos melhorar**?

Mergesort

- Um dos primeiros algoritmos construídos (desenhados/projetados) para computadores generalistas (1945).

  Nessa altura, a maioria dos "bons" computadores que já existia eram "bons" porque eram "dedicados" – específicos para executar determinadas tarefas

- Desenvolvido por John von Neuman que, juntamente com Herman Goldstine, publica uma versão não recursiva (em 1947), que é hoje chamada ***bottom-up mergesort.***

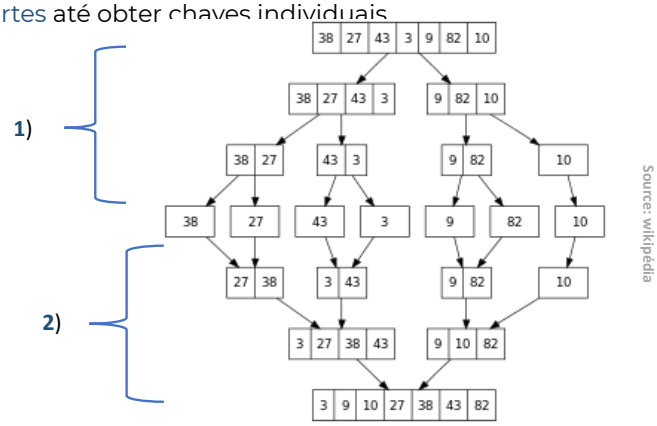
  Este é o mesmo do "Modelo de Von Neuman para representação de um computador digital"

Ordenamento por Fusão: *merge* (to) *sort*

Este algoritmo trabalha em duas fases:

1) **Separar** a sequência em 2 partes (aproximadamente) iguais e continuar recursivamente em cada uma das partes até obter chaves individuais

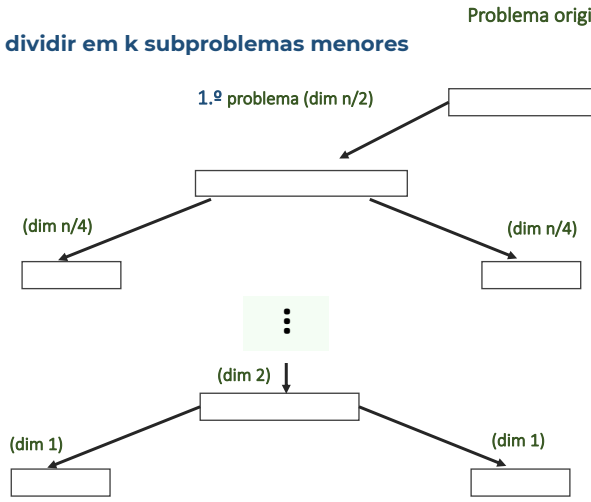
2) **Fundir** (juntar) **em ordem** cada uma das duas partes já ordenadas



Source: wikipédia

29

O mergesort segue a estratégia recursiva **divide-and-conquer**



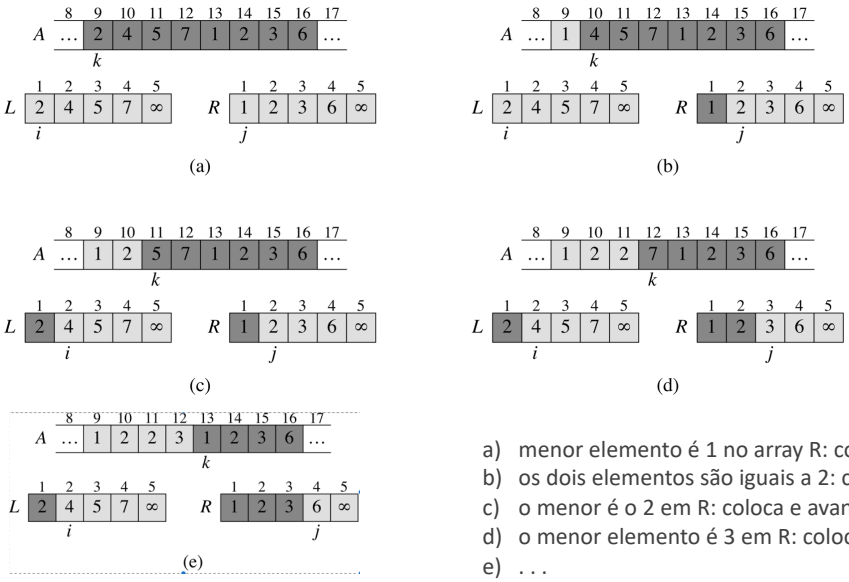
Decompõe o problema em problemas iguais mas de menor dimensão/complexidade, cuja integração, em ordem, resolve o problema original

31

Ordenamento por fusão (merge sort)

- Baseado no conceito de **fusão ordenada de duas sequências já ordenadas**
- Quando a estrutura de dados subjacente é um array, necessita de armazenamento adicional temporário (outro array) para funcionar (realmente) eficientemente
- Se os elementos a ordenar estiverem numa lista ligada, apenas atualiza ponteiros e não necessita espaço adicional

Passo intermédio na fusão



a) menor elemento é 1 no array R: coloca e avança em R
b) os dois elementos são iguais a 2: coloca e avança em L
c) o menor é o 2 em R: coloca e avança em R
d) o menor elemento é 3 em R: coloca e avança em R
e) . . .

Ordenação: **ordenamento por Fusão**

Algoritmo recursivo

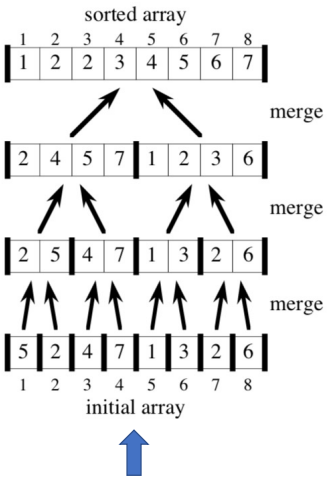
merge sort

```
mergesort(a, ini, fim):
    //Ordena uma sequência por ordem crescente
    //Input: sequência a ordenar e posições inicial e final da sequência
    //Output: sequência ordenada

    if ini < fim:          ← Se a tem tamanho 1, já está ordenado (caso base)
        meio = maior inteiro contido em (ini+fim/2)
        mergesort(a, ini, meio)      ← ordenar a metade esquerda
        mergesort(a, meio+1,  fim)   ← ordenar a metade direita
        merge(a, ini, meio, fim)     ← fundir em ordem
    endif
```

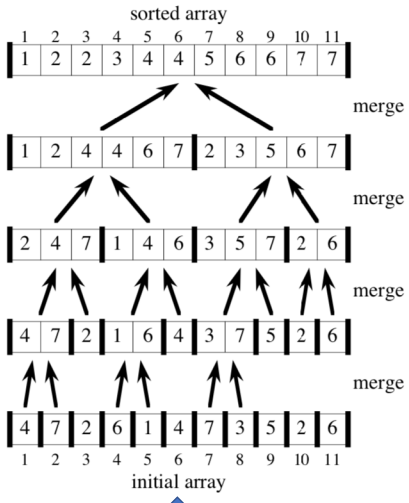
invocação inicial: `mergesort(seq, 1, n)`

(Sendo 1 a primeira posição na sequência seq)

Ordenamento por Fusão: merge - examples



Processo de junção (tamanho par)



Processo de junção (tamanho ímpar)

Algoritmo de Fusão (merge)

```
merge(a, ini, meio, fim):
    // fusão de duas sequências ordenadas:
    // a[ini .. meio] e a[meio+1 .. fim]

    i ← ini;   j ← meio+1
    for k = ini … fim:
        if j > fim:
            aux[k] ← a[i]; i←i+1
        else: if i > meio :
             aux[k] ← a[j]; j←j+1
        else: if a[i] < a[j]
             aux[k] ← a[i]; i←i+1
        else:  aux[k] ← a[j]; j←j+1
    for k = ini … fim
          a[k] ← aux[k]
```

```
mergesort(a, ini, fim):
     //Ordena uma sequência por ordem crescente
     //Input: sequência a ordenar e posições inicial e final da sequência
     //Output: sequência ordenada
    if ini < fim:
        meio = maior inteiro contido em (ini+fim/2)
        mergesort(a, ini, meio)
        mergesort(a, meio+1,  fim)
        merge(a, ini, meio, fim)
   endif
```

Algoritmo para efetuar a fusão ordenada

Questões essenciais:

1. Funciona?

O ordenamento é feito em em 2 fases:

Fase 1): partição recursiva (top-down) da sequência em duas metades até atingir casos base (sequência com um só elemento)

Fase 2) fusão iterativa de baixo para cima (bottom-up) até atingir os n elementos originais e ordenados

Parece funcionar, mas é preciso alguma evidência para afirmar que "está correto".

Correção do Mergesort

```
//pseudocódigo simplificado
MERGESORT(A):
    n = length(A)
    if n ≤ 1:
        return A
    L = MERGESORT(A[1 .. n/2])
    R = MERGESORT(A[n/2+1 .. n])
    return MERGE(L, R)con
```

- **Hipótese de indução (H.I.):**
  "O MERGESORT devolve uma sequência ordenada com i elementos."
- **Caso Base (k=1):**
- **Passo Indutivo:** Mostrar que, assumindo que a H.I. é verdadeira para k<i, então é verdadeira para k=i.
- **Conclusão:** MERGESORT devolve uma sequência ordenada para qualquer i = n >=1.

Análise do Mergesort

```
merge(a, ini, meio, fim):
    // fusão de duas sequências ordenadas:
    // a[ini .. meio] e a[meio+1 .. fim]
    i ← ini;   j ← meio+1
    for k = ini … fim:
        if j > fim:
            aux[k] ← a[i]; i←i+1
        else: if i > meio :
            aux[k] ← a[j]; j←j+1
        else: if a[i] < a[j]
            aux[k] ← a[i]; i←i+1
        else:  aux[k] ← a[j]; j←j+1
    for k = ini … fim
        a[k] ← aux[k]
```

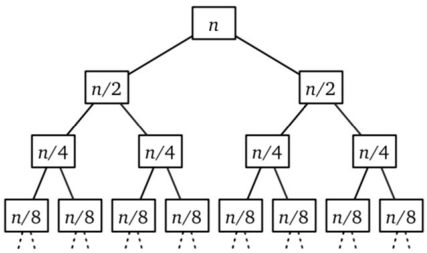
- **Merge**:
  - o ciclo **for** que efetua um trabalho constante por cada iteração
    (3 comparações + 1 soma + 2 atribuição com acesso)
  - Logo, é $O(\ )$ (com n = tamanho da sequência de input)
- O **mergesort** faz **2** chamadas **recorrentes** para (aprox.) metade dos elementos da entrada: **n/2** (na verdade, faz ⌊n/2⌋)
- Logo:

Exercício: Prove, usando indução que T(n) = O(n log n)

Análise informal

1. Na fusão em ordem, cada subarray é percorrido na totalidade uma vez, logo, é $O(n)$.
2. No total, a árvore diminui o input em metade e, em cada nível da árvore, existe o dobro de nós que no nível anterior. Então, a altura da árvore é --------
3. Donde, o trabalho para ordenar no mergesort é $O(\ \ )$

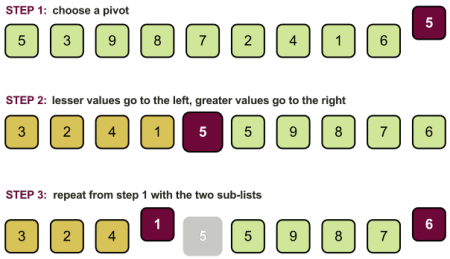
Árvore de recursão do MergeSort



O MergeSort é (assintoticamente) mais eficiente que o InsertionSort ou o SelectionSort

Ordenamento por Partição: *Quicksort*

- Método proposto por C. Hoare em 1961
- Baseado nas ideias seguintes:
  - Para ordenar uma sequência, as trocas mais produtivas são as efetuadas entre elementos mais distantes
  - Se a sequência estiver por ordem inversa, um bom método deve realizar apenas n/2 trocas



Quicksort

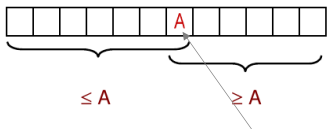
- **Método de separação** entre maiores e menores
- Processo recorrente: a separação é aplicada recursivamente a cada uma das duas partes

Descrição do algoritmo:

- "colocar" pequenos para a esquerda e grandes para a direita
- Aplicar o algoritmo à esquerda
- Aplicar o algoritmo à direita

Separação de elementos à esquerda e à direita

- Para separar é necessário um elemento – **pivot** – cujo valor serve para distinguir maiores de menores (relativamente ao valor do pivot)
- O pivot define a **fronteira de separação**:
  - Exemplo: 17 12 6 19 23 8 5 10
  - Pivot = **10**
  - Resultado da separação: **5 8 6 10 23 19 12 17**



Se o pivot pertence à sequência a ordenar, fica posicionado na sua posição definitiva

Ordenação: Algoritmo **Quicksort**

Quicksort

```
quicksort(a, ini, fim):
    //Ordena uma sequência por ordem crescente
    //Input: sequência a ordenar e posições inicial e final da sequência
    //Output: sequência ordenada

    if ini < fim:
        partir = partition(ini, fim)
        quicksort(a, ini, partir)
        quicksort(a, partir+1,  fim)
    endif
```

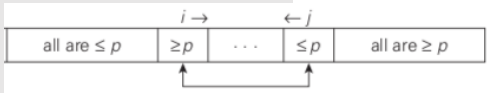
invocação inicial: `quicksort(seq, 1, n)`

(Sendo **1** a primeira posição na sequência seq)

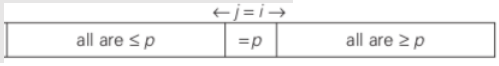
Algoritmo de partição do Quicksort

```
Hoarepartition(a, e, d)
     //Partition subarray (Hoare's algorithm) using the first element as a pivot
    //Input: array a[e..d], defined by its left (e) and right (d) indices (e < d)
    //Output: Partition of a[e..d] (returns the split position)

     p ← a[e]
     i ← e; j ← d
     while i < j
          repeat:   i ← i + 1   until a[i] ≥ p
          repeat:   j ← j – 1   until a[j] ≤ p
          swap(a[i], a[j])
      endwhile
      return j
```

passo intermédio



final



- No final de uma partição, passámos por todos os elementos do array, logo, é $O(n)$.

Exercício: Mostre que de facto a partição é O(n).

Análise do Quicksort

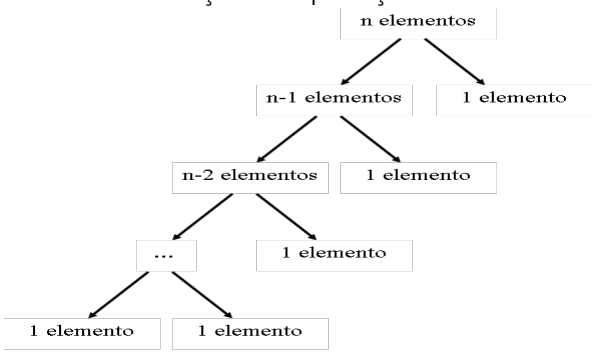
- No final de uma partição, passámos por todos os elementos da sequência, logo, (o algoritmo d)a partição é $O(n)$.
- Mais uma vez, fazemos chamadas recursivas para resolver (ordenar) toda a sequência.
- Sendo r a posição do pivot (ou correspondente à paragem da partição), separámos r-1 elementos para o lado esquerdo e n-r para o lado direito, logo,

$$T(n) =$$

- Mas … esta posição "r" não é fixa e depende da escolha do pivot

Análise de Eficiência do Quicksort no pior dos casos (***worst case***)

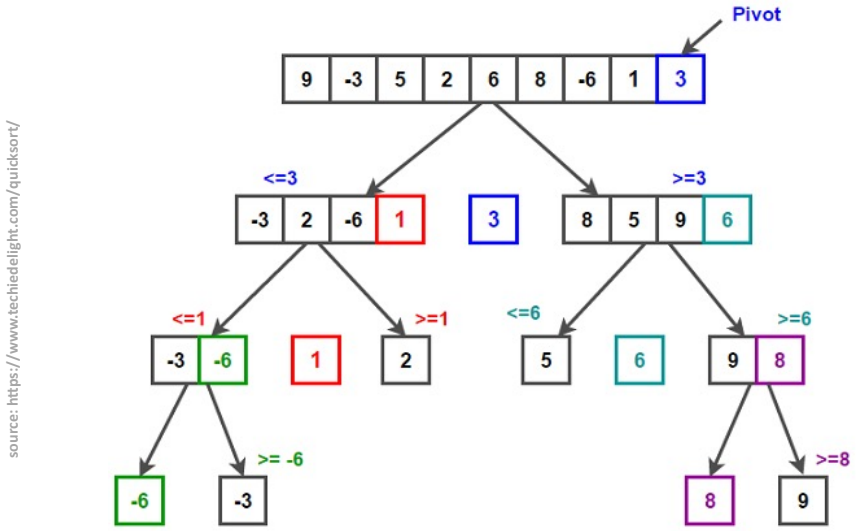
- O pior dos casos acontece-se quando a separação é o **menos** eficiente possível: a maioria dos elementos fica numa das secções de partição



- Como a árvore fica com **n-1** níveis**,** no pior dos casos o Quicksort é ***O(n^2).***

Quicksort

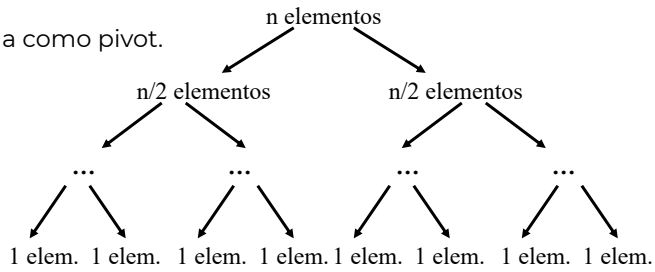
O melhor pivot possível, ou seja, que parte equilibradamente é



source: https://www.techiedelight.com/quicksort/

Análise de Eficiência do Quicksort usando a mediana como pivot

- O melhor caso consiste, em cada passo de separação, conseguir dividir o mais simetricamente possível (~ n/2)
- O que fica garantido se usarmos a mediana como pivot.



- Como a árvore fica com $log_2 n$ níveis, este caso seria ***O(n log n)***

Exemplos de algoritmos que seguem a estratégia **divide-and-conquer**

- Ambos o *mergesort* e o *quicksort* seguem um padrão recursivo de 3 fases do tipo *divide and conquer*:
  - **Dividir** (a instância d)o problema em várias instâncias do MESMO problema, mas de dimensão menor, e resolúveis independentemente umas das outras.
  - **Delegar** a resolução de cada uma das instâncias menores recursivamente.
  - **Combinar** as soluções das instâncias menores para construir a solução final.

Para reter das aulas anteriores

- Análise de algoritmos:
- Alguns algoritmos de ordenação:
  - Algoritmos quadráticos: Selection sort e Insertion sort
  - Algoritmo da fusão ordenada: *merge sort é O(n log n)*
  - *Algoritmo quicksort é quadrático no pior dos casos e é O(n log n) no melhor dos casos.*
- Estratégias de desenho de algoritmos:
  - Pesquisa Exaustiva
  - Estratégia Incremental (*Decrease-and-Conquer*)
  - Dividir para Conquistar (*Divide-and-Conquer*)

Para estudar

- **Cap. 2 e 3 – *Introduction to Algorithms*.** Cormen, Leiserson et al., 3rd ed. MIT Press, 2009
- **Cap. 2 – Algorithm *Design***, John Kleinberg, Eva Tàrdos, Addison-Wesley, 2005.