Dividir para Conquistar

José Elias Claudio Arroyo

Departamento de Informática
Universidade Federal de Viçosa

INF 332 - 2022/2

Outline

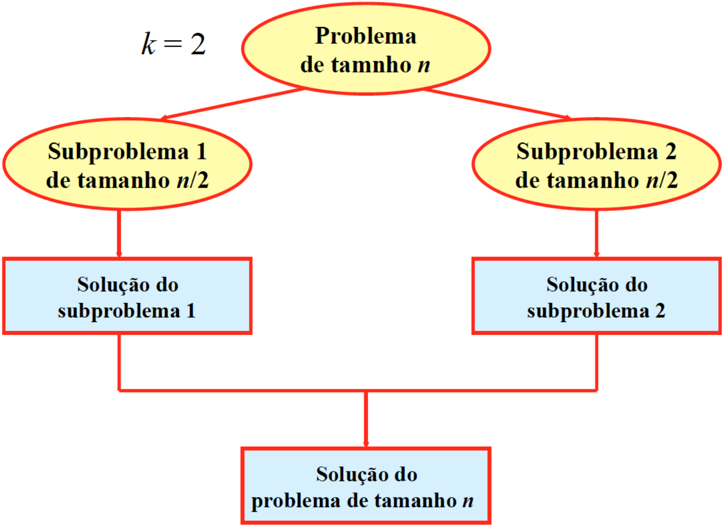
Dividir para Conquistar (DC)

Divisão e Conquista é um dos paradigmas mais conhecidos de projeto de algoritmos.

Os **passos principais** da técnica DC são:

- **Dividir** o problema em $k \geq 2$ problemas menores (subproblemas);
- Solucionar os problemas menores (**conquistar**);
- **Combinar** as soluções dos subproblemas para obter a solução do problema original.

Dividir para Conquistar (DC)



Exemplos de algoritmos que utilizam a técnica DC.

1. Ordenação: Mergesort e Quicksort;
2. Caminhamento em árvore;
3. Subsequência consecutiva máxima;
4. Par de pontos mais próximo;
5. Problema da envoltória convexa;
6. Algoritmo de Strassen para multiplicação de matrizes;
7. Multiplicação de inteiros grande.

Exemplo Simples de DC: Soma de Elementos

Determinar a soma dos elementos de uma lista de n números $\{a_1, \cdots, a_n\}$,
Ou seja, S = **Soma** $(a_1, \cdots, a_n)$

Exemplo Simples de DC: Soma de Elementos

Determinar a soma dos elementos de uma lista de n números $\{a_1, \cdots, a_n\}$,
Ou seja, S = **Soma** $(a_1, \cdots, a_n)$

1. Se $n = 1$ (lista com um único elemento), **retorne** o valor desse elemento. Caso contrário, faça os passos 2, 3 e 4.
2. Dividir a lista em 2 partes: $\{a_1, \cdots, a_{\lfloor n/2 \rfloor}\}$ e $\{a_{\lfloor n/2 \rfloor + 1}, \cdots, a_n\}$;
3. Recursivamente, somar os elementos de cada parte:
   S_1 = **Soma** $(a_1, \cdots, a_{\lfloor n/2 \rfloor})$
   S_2 = **Soma** $(a_{\lfloor n/2 \rfloor + 1}, \cdots, a_n)$
4. $S = S_1 + S_2$. //Conquistar
5. **Retorne** S.

Exemplo Simples de DC: Soma de Elementos

```
Soma(A[i..f]){
//Entrada: array A[0..n-1], onde i=0 e f=n-1

if (i == f):  return A[i]
else:
        meio = (i+f)/2
        S1 = Soma(A[i..meio])
        S2 = Soma(A[meio+1..f])
        return S1 + S2
}
```

Recorrência Soma de Elementos

$T(n) = 2T(n/2) + 1,\ T(1) = 0.$

Pelo teorema mestre:

$$T(n) = aT(n/b) + f(n)\,, \text{onde } f(n) \in \Theta(n^k)$$

$$\begin{array}{ll} T(n) \in \Theta(n^k)\,, & \text{se } a < b^k \\ T(n) \in \Theta(n^k \log n)\,, & \text{se } a = b^k \\ T(n) \in \Theta(n^{\log_b a})\,, & \text{se } a > b^k \end{array}$$

$\Rightarrow$ A soma de n elementos por divisão e conquista é $\Theta(n)$

Recorrência Soma de Elementos

$T(n) = 2T(n/2) + 1,\ T(1) = 0.$

Pelo teorema mestre:

$$T(n) = aT(n/b) + f(n)\,, \text{onde } f(n) \in \Theta(n^k)$$

$$\begin{aligned} T(n) &\in \Theta(n^k)\,, && \text{se } a < b^k \\ T(n) &\in \Theta(n^k \log n)\,, && \text{se } a = b^k \\ T(n) &\in \Theta(n^{\log_b a})\,, && \text{se } a > b^k \end{aligned}$$

$\Rightarrow$ A soma de n elementos por divisão e conquista é $\Theta(n)$

Para somar os n elementos, o algoritmo baseado em divisão e conquista é mais eficiente que o algoritmo de força bruta?

Mergesort

Ordenar um arranjo $A[0..n-1]$ de n elementos.

Mergesort

Ordenar um arranjo $A[0..n-1]$ de n elementos.

Algoritmo Mergesort

1. Se $n > 1$, divida o arranjo $A[0..n-1]$ em dois: $A[0..\lfloor n/2 \rfloor - 1]$ e $A[\lfloor n/2 \rfloor..n-1]$;
2. Recursivamente ordene $A[0..\lfloor n/2 \rfloor - 1]$ e $A[\lfloor n/2 \rfloor..n-1]$;
3. Combine (**merge**) os arranjos ordenados $A[0..\lfloor n/2 \rfloor - 1]$ e $A[\lfloor n/2 \rfloor..n-1]$ obtendo o arranjo $A[0..n-1]$ ordenado.

Mergesort

Ordenar um arranjo $A[0..n-1]$ de n elementos.

Algoritmo Mergesort

1. Se $n > 1$, divida o arranjo $A[0..n-1]$ em dois: $A[0..\lfloor n/2 \rfloor - 1]$ e $A[\lfloor n/2 \rfloor..n-1]$;
2. Recursivamente ordene $A[0..\lfloor n/2 \rfloor - 1]$ e $A[\lfloor n/2 \rfloor..n-1]$;
3. Combine (**merge**) os arranjos ordenados $A[0..\lfloor n/2 \rfloor - 1]$ e $A[\lfloor n/2 \rfloor..n-1]$ obtendo o arranjo $A[0..n-1]$ ordenado.

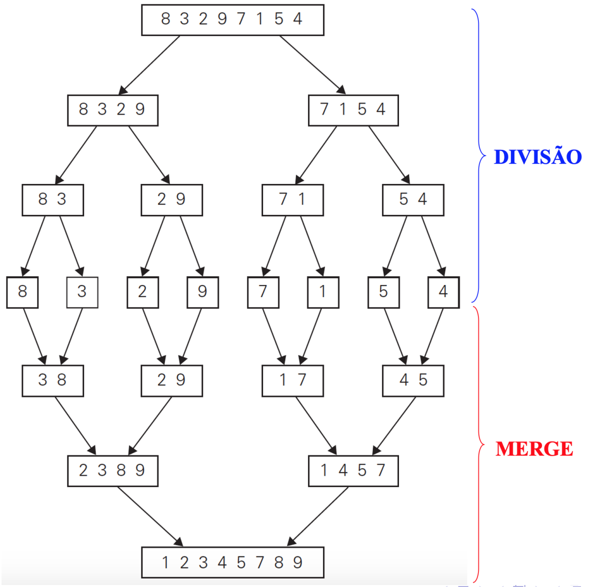
Note que, combinar (intercalar) dois arranjos ordenados é facilmente feito em tempo $\Theta(n)$.

Exemplo

Ordene a lista $\{8, 3, 2, 9, 7, 1, 5, 4\}$ com Mergesort ($n = 8$).

A seguir mostra-se o gráfico das etapas de divisão e merge para a ordenação dessa lista.

Mergesort


8 3 2 9 7 1 5 4
8 3 2 9
7 1 5 4
8 3
2 9
7 1
5 4
8
3
2
9
7
1
5
4
3 8
2 9
1 7
4 5
2 3 8 9
1 4 5 7
1 2 3 4 5 7 8 9
DIVISÃO
MERGE

ALGORITHM *Mergesort*$(A[0..n-1])$

//Sorts array $A[0..n-1]$ by recursive mergesort
//Input: An array $A[0..n-1]$ of orderable elements
//Output: Array $A[0..n-1]$ sorted in nondecreasing order
if $n > 1$
  copy $A[0..\lfloor n/2 \rfloor - 1]$ to $B[0..\lfloor n/2 \rfloor - 1]$
  copy $A[\lfloor n/2 \rfloor..n-1]$ to $C[0..\lceil n/2 \rceil - 1]$
  Mergesort$(B[0..\lfloor n/2 \rfloor - 1])$
  Mergesort$(C[0..\lceil n/2 \rceil - 1])$
  Merge(B, C, A) //see below

Pseudocódigo retirado de: *Introduction to the design & analysis of algorithms / Anany Levitin*

ALGORITHM $Merge(B[0..p-1], C[0..q-1], A[0..p+q-1])$

//Merges two sorted arrays into one sorted array
//Input: Arrays $B[0..p-1]$ and $C[0..q-1]$ both sorted
//Output: Sorted array $A[0..p+q-1]$ of the elements of B and C
$i \leftarrow 0$; $j \leftarrow 0$; $k \leftarrow 0$
while $i < p$ **and** $j < q$ **do**
  if $B[i] \leq C[j]$
    $A[k] \leftarrow B[i]$; $i \leftarrow i+1$
  else $A[k] \leftarrow C[j]$; $j \leftarrow j+1$
  $k \leftarrow k+1$
if $i = p$
  copy $C[j..q-1]$ to $A[k..p+q-1]$
else copy $B[i..p-1]$ to $A[k..p+q-1]$

Mergesort

Análise de complexidade.

- A operação de ***Merge*** faz, no pior caso, $n - 1$ comparações
- Em cada iteração do algoritmo ***Mergesort*** soluciona-se dois problemas com a metade do tamanho do problema original. Então, o tempo do Mergesort é:

$$T(n) = 2T(n/2) + n - 1$$

Pelo teorema mestre, Mergesort é da classe $O(n \log n)$.

Mergesort

Exercício

Determinar o valor exato da recorrência (fórmula fechada):

$T(1) = 0$

$T(n) = 2T(n/2) + n - 1$

Algoritmo Quicksort

Ordenar uma lista $A[l..r]$, onde $l = 0$ e $r = n - 1$:

1. Selecione um pivô p em $A[l..r]$ (por exemplo, o primeiro elemento: $p = A[l]$).
2. Divida A em duas partes tal que, os elementos da primeira parte (sub-lista $A[l..j]$) sejam menores ou iguais a $A[l]$, e os elementos da segunda parte (sub-lista $A[j+1..r]$) sejam maiores ou iguais a $A[l]$.
   j é a posição da divisão.
3. Troque $A[l]$ com $A[j]$ (último elemento da sub-lista $A[l..j]$).
4. O pivô $A[s]$ = $A[j]$ estará na sua posição correta.
5. Ordene $A[l..s-1]$ e $A[s+1..r]$ recursivamente.

$$\underbrace{A[0]\dots A[s-1]}_{\text{all are } \le A[s]} \quad A[s] \quad \underbrace{A[s+1]\dots A[n-1]}_{\text{all are } \ge A[s]}$$

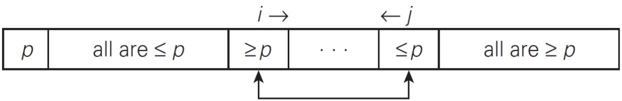
Algoritmo para Particionar $A[l..r]$ (Algoritmo de Hoare):

1. Escolha o pivô $p = A[l]$.
2. Percorrendo da esquerda-para-direita (iniciando em $i = l + 1$), procure um elemento $A[i]$ **maior** que p.
3. Percorrendo da direita-para-esquerda (iniciando em $j = r$), procure um elemento $A[j]$ **menor** que p.
4. Troque os elementos $A[i]$ e $A[j]$.
5. Repita enquanto $i < j$.

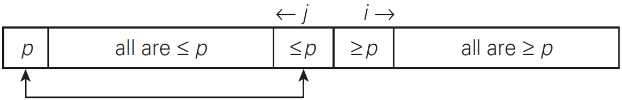
Quicksort

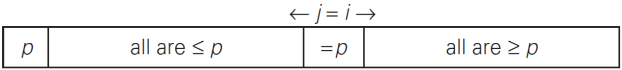
Casos da Partição:

Se $i < j$: trocar $A[i]$ com $A[j]$



Se $j \leq i$: trocar $A[l]$ com $A[j]$





ALGORITHM $Quicksort(A[l..r])$

//Sorts a subarray by quicksort
//Input: Subarray of array $A[0..n-1]$, defined by its left and right
// indices l and r
//Output: Subarray $A[l..r]$ sorted in nondecreasing order
if $l < r$
  $s \leftarrow Partition(A[l..r])$ //s is a split position
  $Quicksort(A[l..s-1])$
  $Quicksort(A[s+1..r])$

Pseudocódigo retirado de: *Introduction to the design & analysis of algorithms / Anany Levitin*

Quicksort

ALGORITHM $\quad HoarePartition(A[l..r])$

```
//Partitions a subarray by Hoare's algorithm, using the first element
//        as a pivot
//Input: Subarray of array A[0..n − 1], defined by its left and right
//        indices l and r (l < r)
//Output: Partition of A[l..r], with the split position returned as
//        this function's value
p ← A[l]
i ← l; j ← r + 1
repeat
    repeat i ← i + 1 until A[i] ≥ p
    repeat j ← j − 1 until A[j] ≤ p
    swap(A[i], A[j])
until i ≥ j
swap(A[i], A[j])    //undo last swap when i ≥ j
swap(A[l], A[j])
return j
```

Exemplo

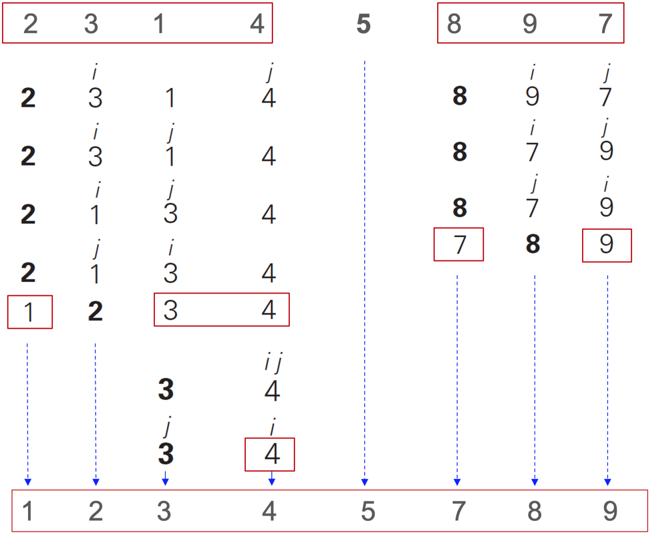
Ordene de forma crescente a lista $A[0..7] = \{5, 3, 1, 9, 8, 2, 4, 7\}$ com Quicksort.

A seguir mostra-se os gráficos das etapas do Quicksort para a ordenação dessa lista.

Quicksort

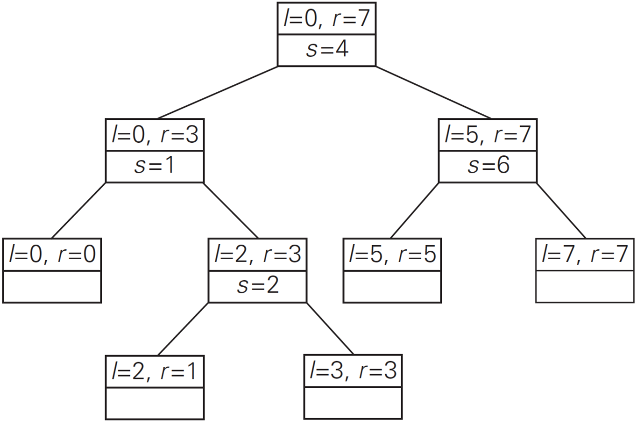
0	1	2	3	4	5	6	7
	i						j
5	3	1	9	8	2	4	7
			i			j	
5	3	1	9	8	2	4	7
			i			j	
5	3	1	4	8	2	9	7
				i	j		
5	3	1	4	8	2	9	7
				i	j		
5	3	1	4	2	8	9	7
				j	i		
5	3	1	4	2	8	9	7
2	3	1	4	**5**	8	9	7

Quicksort


2 3 1 4 5 8 9 7
i j i j
2 3 1 4 8 9 7
i j i j
2 3 1 4 8 7 9
i j j i
2 1 3 4 8 7 9
j i
2 1 3 4 7 8 9
1 2 3 4
i j
3 4
j i
3 4
1 2 3 4 5 7 8 9

Quicksort

Árvore de chamadas recursivas do algoritmo **Quicksort** para ordenar $A[0..7] = \{5, 3, 1, 9, 8, 2, 4, 7\}$

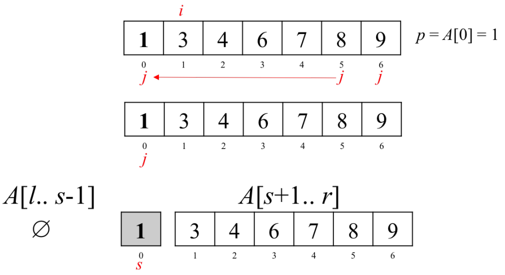


Análise de complexidade do Quicksort

- Para $n = 1$ ($l = r$): $T(1) = 0$ (base da recursão)
- Para $n > 1$: Analisar a complexidade das 3 chamadas:
  - **Partition**($A[l..r]$)
    Na divisão, os índices i e j percorrem todo o arranjo até que se cruzam. São realizadas $O(n)$ comparações.
  - **Quicksort**($A[l..s-1]$) e
  - **Quicksort**($A[s+1..r]$).

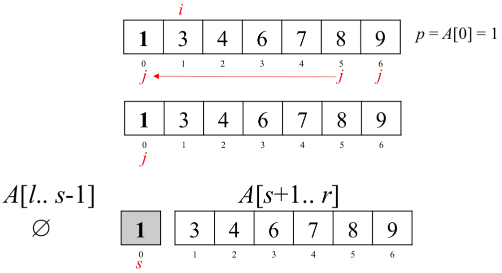
$CT(n) = O(n) +$ Tempo Quicksort($A[l..s-1]$) +
Tempo Quicksort($A[s+1..r]$)

O **Pior Caso** ocorre quando o pivô é o menor ou o maior elemento do arranjo. Neste caso a partição cria um arranjo com $n - 1$ elementos e outro com nenhum elemento (vazio).



Análise de complexidade do Quicksort

O **Pior Caso** ocorre quando o pivô é o menor ou o maior elemento do arranjo. Neste caso a partição cria um arranjo com $n-1$ elementos e outro com nenhum elemento (vazio).



$$\Rightarrow T(n) = n + T(0) + T(n-1) = n + T(n-1) \in O(n^2)$$

Análise de complexidade do Quicksort

Melhor caso

- O procedimento de partição divide a entrada n em **duas partes do mesmo tamanho** $n/2$. Isto ocorre quando o pivô, é sempre, a **mediana** do arranjo.
- A partição realiza n comparações.

$$T(n) = 2T(n/2) + n\,, \text{ com } T(1) = 0\,.$$

$\Rightarrow$ pelo teorema Mestre: $T(n) \in O(n \log n)$.

Análise de complexidade do Quicksort

Caso médio

- Assume-se que a chance de particionar $A[0..n-1]$ em qualquer uma das n posições é a mesma: $\frac{1}{n}$

Caso médio

- Assume-se que a chance de particionar $A[0..n-1]$ em qualquer uma das n posições é a mesma: $\frac{1}{n}$

$T(n) = \frac{1}{n}[n + T(0) + T(n-1)] + \frac{1}{n}[n + T(1) + T(n-2)] + \frac{1}{n}[n + T(2) + T(n-3)] + ... + \frac{1}{n}[n + T(n-1) + T(0)]$

$T(n) = \frac{1}{n}\sum_{k=0}^{n-1}[n + T(k) + T(n-k-1)]$

$T(n) \approx 2n \ln n \approx 1.39 n \log_2 n \in O(n \log_2 n)$
(provado por Sedgewick e Flajolet, 1996).

Sedgewick, R. and Flajolet, P. An Introduction to the Analysis of Algorithms. Addison-Wesley Professional, 1996.

Algumas melhorias do Quicksort:

- Use ordenacao por inserção (*InsertionSort*) quando a lista ficar pequena o suficiente (por exemplo, com 15 elementos). No próximo slide está o algoritmo *InsertionSort*.
- Defina o pivô como sendo a mediana de três elementos escolhidos aleatoriamente.
- Para listas maiores, defina o pivô como a mediana de três medianas.

ALGORITHM *InsertionSort*$(A[0..n-1])$

//Sorts a given array by insertion sort
//Input: An array $A[0..n-1]$ of n orderable elements
//Output: Array $A[0..n-1]$ sorted in nondecreasing order
for $i \leftarrow 1$ **to** $n-1$ **do**
    $v \leftarrow A[i]$
    $j \leftarrow i-1$
    while $j \geq 0$ **and** $A[j] > v$ **do**
        $A[j+1] \leftarrow A[j]$
        $j \leftarrow j-1$
    $A[j+1] \leftarrow v$

Pseudocódigo retirado de: *Introduction to the design & analysis of algorithms / Anany Levitin*

Exercícios

Exercício 1

Ordene as sequências utilizando o Mergesort e o Quicksort.

1. 5 3 1 9 8 2 4 7
2. 8 3 2 9 7 1 5 4

Par de Pontos mais Próximo

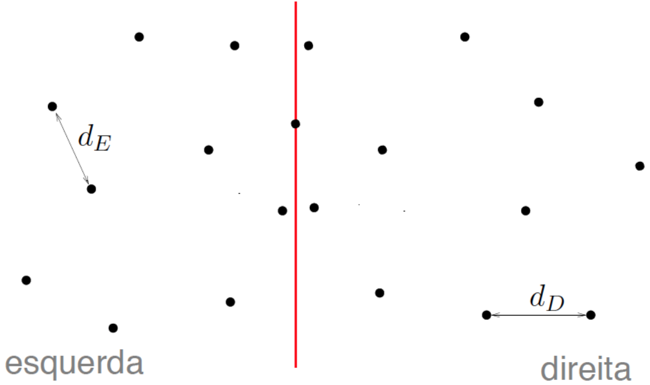
- Dado um conjunto P de $n \geq 2$ pontos no plano, obter o par de pontos mais próximo.
- Supor que os pontos em P estão **ordenados** em ordem crescente da coordenada x (Caso não estejam ordenados, eles podem ser ordenados pelo algoritmo **mergesort** em tempo $O(n \log n)$).
- Também considerar o conjunto Q dos mesmos pontos **ordenados** em ordem crescente da coordenada y.

Solução utilizando DC:

1. Se $2 \leq n \leq 3$, resolver por força bruta.
2. **Dividir**: Se $n > 3$, dividir o conjunto P em 2 subconjuntos P_E e P_D (P_E contem os primeiros $\lceil n/2 \rceil$ pontos de P e P_D contem os $\lfloor n/2 \rfloor$ pontos restantes).
3. **Conquistar**: Recursivamente, determinar o par de pontos mais próximos em P_E e P_D . Sejam d_E e d_D as menores distâncias nos conjuntos P_E e P_D, respectivamente.
4. **Combinar**: Determinar a menor distância d_{ED} entre um ponto da esquerda (P_E) e um ponto da direita (P_D) e retorne $min\{d_E, d_D, d_{ED}\}$ (o mínimo entre d_E, d_D e d_{ED}).

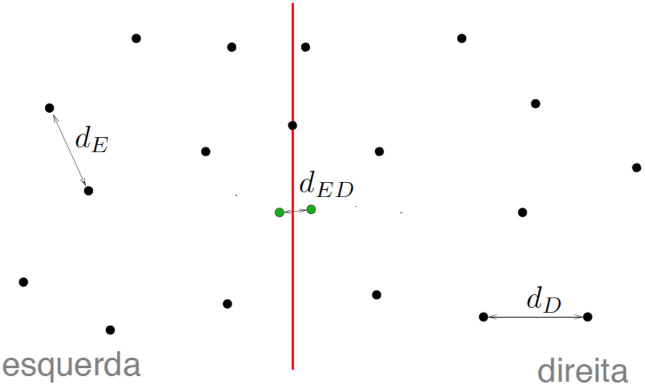
Par de Pontos mais Próximo

Divisão e **Conquista**:



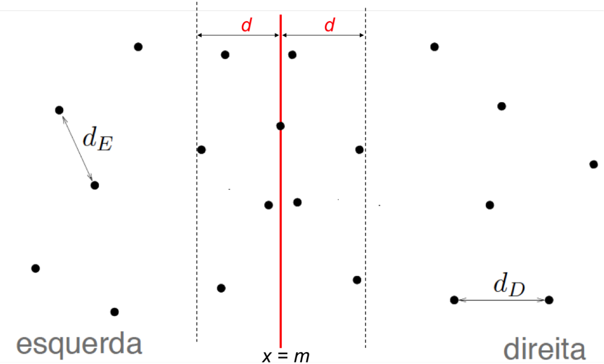
Par de Pontos mais Próximo

Combina:



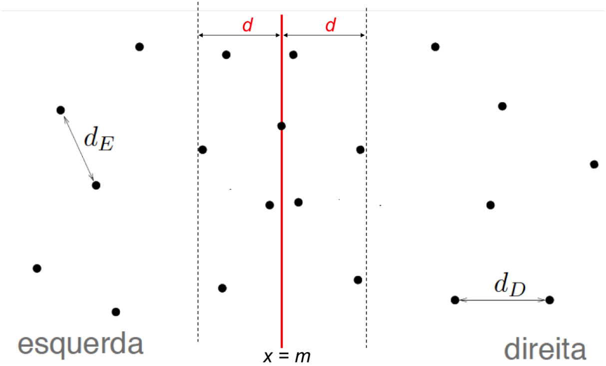
Como fazer a combinação?

- Considerar apenas pontos que estão a uma distância menor que $d = min\{d_E, d_D\}$ da reta vertical $x = m$.



Como fazer a combinação?

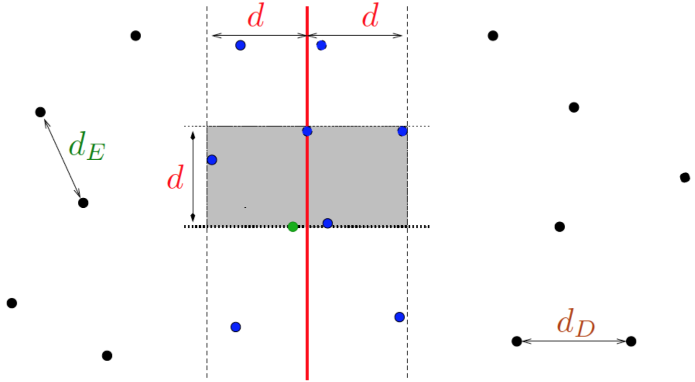
- Considerar apenas pontos que estão a uma distância menor que $d = min\{d_E, d_D\}$ da reta vertical $x = m$.



Note que todos os *n* pontos podem estar nessa faixa *F*.

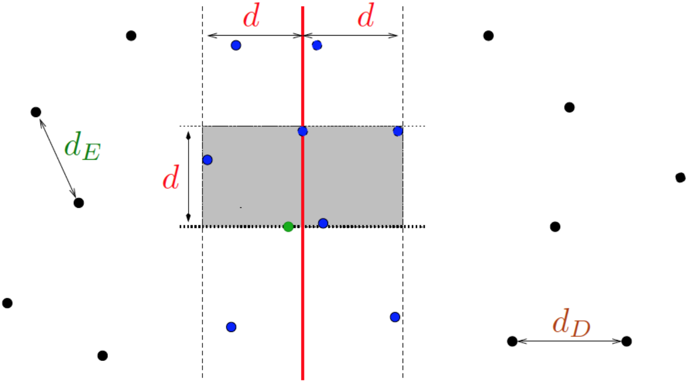
Par de Pontos mais Próximo

- Para cada **ponto** p' na faixa, consideramos apenas os pontos que estejam acima de p' e até uma distância d (ou seja, os pontos da faixa que tenham coordenada y no máximo d mais que o **ponto** p').



Par de Pontos mais Próximo

- Para cada **ponto** p' na faixa, consideramos apenas os pontos que estejam acima de p' e até uma distância d (ou seja, os pontos da faixa que tenham coordenada y no máximo d mais que o **ponto** p').



Em cada um dos quadrados de lado d, haverá no máximo 4 pontos, porque $d \leq d_E$ e $d \leq d_D$.

Par de Pontos mais Próximo

- Logo, em cada **retangulo** $2d \times d$ haverá no máximo 8 pontos.
- Como ter acesso rápido a esses pontos?
  Os pontos também devem estar ordenados em ordem crescente da coordenada y (conjunto Q).
- Assim, para cada ponto p', o algoritmo de combinação analisará somente os próximos 7 pontos da conjunto ordenado Q.
- No pior caso, se todos os n pontos estiverem na faixa F, o algoritmo de **combinação** gastará $7n = O(n)$ (para cada ponto da faixa serão analizados 7 pontos).

Par de Pontos mais Próximo

```
PontosProximos(P, Q)
\\P = pontos ordenados em ordem crescente de x
\\Q = pontos ordenados em ordem crescente de y
Se(n <= 3) return a menor distância obtido por forca bruta
Senao
   (Pe, Pd) = Divide(P);   (Qe, Qd) = Divide(Q);
   dE = PontosProximos(Pe, Qe);
   dD = PontosProximos(Pd, Qd);
   d = min{dE, dD};    m = P[n/2].x;
   F[1,...,num] = {p em Q/ |p.x - m| < d};
   dmin = d;
   Para i = 1  até num-1:
          k = i+1;
          Enquanto(k<=num e |F[i].y - F[k].y|<dmin)
                dmin = min{dist(F[i],F[k]), dmin};
                k = k+1;
return dmin;
```

Par de Pontos mais Próximo

- O algoritmo gasta tempo $O(n)$ para fazer a **divisão**, quanto para **combinar** as soluções obtidas.
- Portanto, assumindo que n é uma potência de 2, temos a seguinte recorrência para o tempo de execução do algoritmo:
- $T(n) = 2T(n/2) + f(n)$, onde $f(n) \in \Theta(n)$.
- Aplicando o Teorema Mestre (com $a = 2$, $b = 2$, and $k = 1$), $T(n) \in \Theta(n \log_2 n)$.

Subsequência Consecutiva Máxima

Dada uma sequência $X = [x_1, x_2, ..., x_n]$ de n números reais.
Encontrar uma subsequência consecutiva $Y = [x_i, x_{i+1}, ..., x_j]$ de X, $1 \leq i \leq j \leq n$, tal que a soma dos elementos de Y seja máxima.

Exemplos:

- $X = [4, 2, -7, 3, 0, -2, 1, 5, -2] \Rightarrow$ soma = 7.
- $X = [-2, 11, -4, 13, -5, 2] \Rightarrow$ soma = 20.
- $X = [-1, -2, 0] \Rightarrow$ soma = 0.
- $X = [4, 2, 8, 1] \Rightarrow$ soma = 15.

Aplicação da técnica DC para determinar a SCM:

- Dividir a sequencia X em duas partes: $X1$ e $X2$.
  Pode acontecer **3 casos**:
- A SCM pode estar completamente em $X1$;
- A SCM pode estar completamente em $X2$;
- A SCM pode iniciar em $X1$ e terminar em $X2$;.

Subsequência Consecutiva Máxima

Caso 3:

- $X = [4, -7, 3, 0, -2, 1, 5, -2]$
- Divisão e Conquista:
  $X1 = [4, -7, 3, 0] \Rightarrow$ Resolvendo: $Y1 = [4]$, somaEsq = 4
  $X2 = [-2, 1, 5, -2] \Rightarrow$ Resolvendo: $Y2 = [1, 5]$, somaDir = 6
- A SCM é : Y = [3, 0,-2, 1, 5], inicia em $X1$ e termina em $X2$

Subsequência Consecutiva Máxima

Para determinar a SCM no **caso 3**:

1. Percorrer $X1$ do final para o início e determinar a maior soma:
   $X1 = [4, -7, 3, 0] \Rightarrow MaxSoma1 = 3$
2. Percorrer $X2$ do inicio para o final e determinar a maior soma:
   $X2 = [-2, 1, 5, -2] \Rightarrow MaxSoma2 = 4$
3. Juntar:
   $Y3 = [3, 0, -2, 1, 5]$, $MaxSoma1 + MaxSoma2 = 7$
4. Comparar com as soluções $Y1$, $Y2$ e $Y3$ e escolher o máximo:
   $SomaMax = max\{somaEsq, somaDir, MaxSoma1 + MaxSoma2\} = 7$.

Subsequência Consecutiva Máxima

Para determinar a SCM no **caso 3**:

1. Percorrer $X1$ do final para o início e determinar a maior soma:
   $X1 = [4, -7, 3, 0] \Rightarrow MaxSoma1 = 3$
2. Percorrer $X2$ do inicio para o final e determinar a maior soma:
   $X2 = [-2, 1, 5, -2] \Rightarrow MaxSoma2 = 4$
3. Juntar:
   $Y3 = [3, 0, -2, 1, 5]$, $MaxSoma1 + MaxSoma2 = 7$
4. Comparar com as soluções $Y1$, $Y2$ e $Y3$ e escolher o máximo:
   $SomaMax = max\{somaEsq, somaDir, MaxSoma1 + MaxSoma2\} = 7$.

Tempo: $O(n)$

Subsequência Consecutiva Máxima

```
SCM (X[], ini, fim){
  if (X = ∅) return (SomaMax, i, j) = (0, 0, 0);
  else{
    if (ini == fim) return (X[ini], ini, ini);

    soma1 = soma2 = 0;
    MaxSoma1 = MaxSoma2 = -∞;
    meio = (ini + fim)/2;

    (somaEsq, i1, j1) = SCM (X, ini, meio);
    (somaDir, i2, j2) = SCM (X, meio+1, fim);

    for (k = meio; k>=ini; k–) {//Percorrer X1 do final para o início
      soma1 = soma1 + X[k];
      if(soma1 > MaxSoma1){ MaxSoma1 = soma1; i3 = k; }
    }
```

Subsequência Consecutiva Máxima

```
    for (k = meio+1; k<=fim; k++){ //Percorrer X2 do início para o final
       soma2 = soma2 + X[k];
      if (soma2 > MaxSoma2){ MaxSoma2 = soma2; j3 = k; }
    }

     SomaMax = max(somaEsq, somaDir, MaxSoma1+ MaxSoma2 );

     (i, j) = DeterminaIndices(i1, j1, i2, j2, i3, j3);

     return (SomaMax, i, j);
  } //end else
}
```

Subsequência Consecutiva Máxima

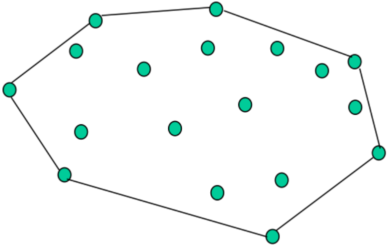
Complexidade:

$T(1) = 1$;
$T(n) = 2T(n/2) + n$; para $n > 1$.

Pelo Teorema Mestre: $T(n) \in (n \log_2 n)$.

Envoltória Convexa

Seja um conjunto S contendo n pontos do plano R^2. Determinar a envoltória convexa formada pelos pontos extremos do conjunto S.

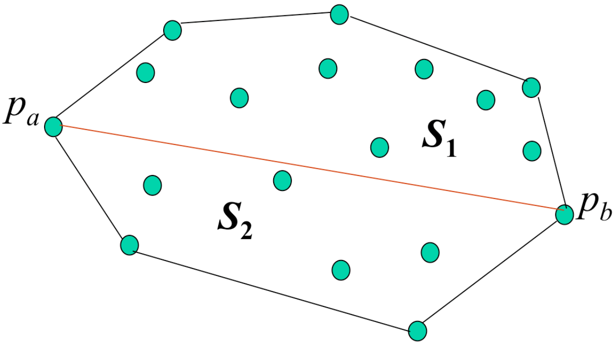


Algoritmo Inspirado no **Quicksort**:

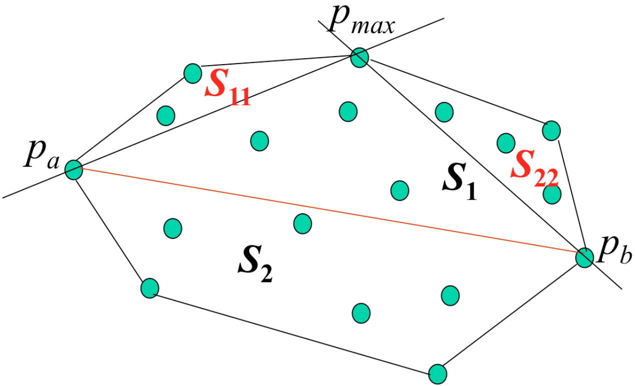
AlgoritmoQuick(S, n)

1. **Se** $n \geq 2$:
2. Determinar os pontos p_a e p_b, que possuem, respectivamente, o menor e maior valor da coordenada x (os pontos *mais à esquerda* e *mais à direita*).
3. $EC = \{p_a, p_b\}$
4. **Dividir** S em dois subconjuntos $S1$ e $S2$:
   - $S1$ é o conjunto de pontos p à esquerda da reta $\overline{p_a p_b}$ ($p \neq p_a, p \neq p_b$)
   - $S2$ é o conjunto de pontos q à direita de $\overline{p_a p_b}$ ($q \neq p_a, q \neq p_b$)
5. $EC1$ = **PoligonoConvexo**($\overline{p_a p_b}, S1$)
6. $EC2$ = **PolgonoConvexo**($\overline{p_a p_b}, S2$)
7. **return** $EC \cup EC1 \cup EC2$.

Envoltória Convexa



Envoltória Convexa



PoligonoConvexo($\overline{p_a p_b}, S1$)

1. **Se** $S1 = \emptyset$: **return** $\emptyset$
2. **Senão:**
3. Encontre em $S1$ o ponto p_{max} mais distante da reta $\overline{p_a p_b}$;
4. Determine o conjunto $S11$ formado pelos pontos do **lado esquerdo** de $\overline{p_a p_{max}}$
5. Determine $S22$ o conjunto formado pelos pontos do **lado direito** de $\overline{p_b p_{max}}$ ($S22$ também pode ser considerado como o conjunto de pontos do **lado esquerdo** de $\overline{p_{max} p_b}$)
6. $Pol1$ = **PoligonoConvexo**($\overline{p_a p_{max}}, S11$)
7. $Pol2$ = **PolgonoConvexo**($\overline{p_b p_{max}}, S22$)
8. **return** $Pol1 \cup Pol2 \cup \{p_{max}\}$.

Fórmulas Geométricas a serem usadas:

Sejam $p_1=(x_1, y_1)$, $p_2 = (x_2, y_2)$ e $p_3 = (x_3, y_3)$

Equação da reta p_1p_2:

$ax + by + c = 0$

Onde, $a = (y_2 - y_1)$ $b = (x_1 - x_2)$ e $c = -(x_1y_2 - y_1x_2)$

Distancia de um ponto p_3 à reta p_1p_2:

$$d(p_3, p_1p_2) = \frac{|ax_3 + by_3 + c|}{\sqrt{a^2 + b^2}}$$

Envoltória Convexa

O ponto p_3 está ao lado esquerdo da reta p_1p_2 se e somente:

$$det = \begin{vmatrix} x_1 & y_1 & 1 \\ x_2 & y_2 & 1 \\ x_3 & y_3 & 1 \end{vmatrix} = x_1y_2 + x_3y_1 + x_2y_3 - x_3y_2 - x_2y_1 - x_1y_3 > 0$$

Se $det = 0$, o ponto p_3 está sobre a reta p_1p_2

Envoltória Convexa

- O AlgoritmoQuick tem a mesma **conplexidade de pior caso** que o quicksort: $O(n^2)$.
- O pior caso acontece quando na primeira divisão, $S1$ contém todos os $n-2$ pontos e $S2 = \emptyset$. Também, nas próximas divisões um dos conjuntos S11 ou S22 sempre é vazio (um deles contem todos os pontos). Ou seja, a cada passo, o problema será reduzido em um ponto.
- O **melhor caso** acontece quando $S1 = \emptyset$ e $S2 = \emptyset$, na primeira divisão. Ou seja, quando todos os pontos pertencem à reta $\overline{p_a p_b}$. O tempo será $O(n)$, gasto para determiar os pontos p_a e p_b, e os conjuntos $S1$ e $S2$.
- No caso médio o algoritmo apresenta um bom desempenho. O algoritmo, na média, faz uma divisão equilibrada em dois subproblemas menores. Uma fração significativa dos pontos são eliminados (aqueles pontos dentro do triângulo $p_a p_{max} p_b$).

Multiplicação de Matrizes

Deseja-se multiplicar duas matrizes quadradas A e B (ordem $n \times n$).

A multiplicação das matrizes pode ser realizada aplicando as **fórmulas de Strassen** que são baseadas na técnica D&C.

Multiplicação de Matrizes

Deseja-se multiplicar duas matrizes quadradas A e B (ordem $n \times n$).

A multiplicação das matrizes pode ser realizada aplicando as **fórmulas de Strassen** que são baseadas na técnica D&C.

Para $n = 2$:

$$\begin{bmatrix} a_{00} & a_{01} \\ a_{10} & a_{11} \end{bmatrix} * \begin{bmatrix} b_{00} & b_{01} \\ b_{10} & b_{11} \end{bmatrix} = \begin{bmatrix} m_1 + m_4 - m_5 + m_7 & m_3 + m_5 \\ m_2 + m_4 & m_1 + m_3 - m_2 + m_6 \end{bmatrix}$$

- $m_1 = (a_{00} + a_{11}) * (b_{00} + b_{11})$
- $m_2 = (a_{10} + a_{11}) * b_{00}$
- $m_3 = a_{00} * (b_{01} - b_{11})$
- $m_4 = a_{11} * (b_{10} - b_{11})$
- $m_5 = (a_{00} + a_{01}) * b_{11}$
- $m_6 = (a_{10} - a_{00}) * (b_{00} + b_{01})$
- $m_7 = (a_{01} - a_{11}) * (b_{10} + b_{11})$

Multiplicação de Matrizes

Fórmulas de Strassen para multiplicar duas matrizes $n \times n$:

$$\begin{pmatrix} C_{00} & C_{01} \\ C_{10} & C_{11} \end{pmatrix} = \begin{pmatrix} A_{00} & A_{01} \\ A_{10} & A_{11} \end{pmatrix} * \begin{pmatrix} B_{00} & B_{01} \\ B_{10} & B_{11} \end{pmatrix}$$

$$= \begin{pmatrix} M_1 + M_4 - M_5 + M_7 & M_3 + M_5 \\ M_2 + M_4 & M_1 + M_3 - M_2 + M_6 \end{pmatrix}$$

Cada matriz é dividida em quatro submatrizes $n/2 \times n/2$. Se n não é potencia de 2, as matrizes podem ser completadas com linhas ou colunas de zeros. Similar à multiplicação de matrizes 2×2, para obter $C = A * B$ será necessário obter as submatrizes M_i $(i = 1, ..., 7)$ usando as fórmulas de Strassen.

Multiplicação de Matrizes

Fórmulas de Strassen

- $M_1 = (A_{00} + A_{11}) * (B_{00} + B_{11})$
- $M_2 = (A_{10} + A_{11}) * B_{00}$
- $M_3 = A_{00} * (B_{01} - B_{11})$
- $M_4 = A_{11} * (B_{10} - B_{11})$
- $M_5 = (A_{00} + A_{01}) * B_{11}$
- $M_6 = (A_{10} - A_{00}) * (B_{00} + B_{01})$
- $M_7 = (A_{01} - A_{11}) * (B_{10} + B_{11})$

Note que, são realizadas 7 multiplicações de matrizes de tamanho $n/2 \times n/2$ e 18 somas.

Complexidade:

- Seja $T(n)$ o número de multiplicações feitas para multiplicar duas matrizes $n \times n$.
- $T(n) = 7T(n/2)$, para $n > 1$. $T(1) = 1$.
  Desenvovendo:
- $T(n) = 7^2 T(n/2^2) = 7^3 T(n/2^3) = ... = 7^k T(n/2^k)$.
  $n/2^k = 1$, então $k = \log_2 n$
- Logo: $T(n) = 7^{\log_2 n} T(1) = 7^{\log_2 n} = n^{\log_2 7} \approx n^{2,807}$.

Para $n = 200$, são realizadas $T(200) = 2.877.340$ multiplicações.
Um algoritmo força bruta, que é $O(n^3)$, realiza 8.000.000 multiplicações.

Busca Binária

Seja uma lista A com n elementos ordenados (ordem crescente). Buscar um elemento K nesta lista.

- O algoritmo de busca binária compara K (chave da busca) com o elemento do meio da lista $A[m]$.
- Se $K = A[m]$, o algoritmo para.
- Caso contrário, a busca é repetida recursivamente na primeira metade da lista se $K < A[m]$, ou na segunda metade se $K > A[m]$:

$$\begin{array}{c} K \\ \updownarrow \\ \underbrace{A[0] \ldots A[m-1]}_{\substack{\text{search here if} \\ K < A[m]}} \quad A[m] \quad \underbrace{A[m+1] \ldots A[n-1]}_{\substack{\text{search here if} \\ K > A[m]}}. \end{array}$$

Busca Binária

```
ALGORITHM  BinarySearch(A[0..n − 1], K)
    //Implements nonrecursive binary search
    //Input: An array A[0..n − 1] sorted in ascending order and
    //        a search key K
    //Output: An index of the array's element that is equal to K
    //        or −1 if there is no such element
    l ← 0;   r ← n − 1
    while l ≤ r do
        m ← ⌊(l + r)/2⌋
        if K = A[m] return m
        else if K < A[m] r ← m − 1
        else l ← m + 1
    return −1
```

Pseudocódigo retirado de: *Introduction to the design & analysis of algorithms / Anany Levitin*

Busca Binária

Complexidade de **pior caso**:

- Operação básica: comparação da chave K.
- Recorrência: $\begin{cases} T(n) = 1 + T(\lfloor n/2 \rfloor), & n > 1 \\ T(1) = 1 \end{cases}$
- Supondo n potência de 2: $\lfloor n/2 \rfloor = n/2$.
- Pelo teorema mestre: $T(n) \in O(log_2 n)$

Complexidade de **melhor caso**: $T(n) \in O(1)$.