# SIN5013 - Análise de Algoritmos e Estruturas de Dados

## Algoritmos de ordenação

Prof. Flávio Luiz Coutinho

# Algoritmos de ordenação

Definição do problema: dado uma sequência (desorganizada) de valores, determinar uma permutação dos valores que os coloque em uma determinada ordem, respeitando um certo critério (crescente ou decrescente, por exemplo).

Importância prática: organização da informação, para viabilizar a recuperação eficiente da mesma, e/ou para fins de apresentação.

Uma definição um pouco mais concreta do problema: ordenar um vetor (*array*) **a** de valores inteiros em ordem crescente. Após concluída a ordenação devemos ter `a[i] <= a[i + 1]`, para `0 <= i < n-1`.

# Algoritmos de ordenação

Quadráticos:

- Selection sort
- insertion sort
- bubble sort

# Algoritmos de ordenação

$\Theta(n \lg n)$:

- Merge sort
- Quicksort
- Heapsort

# Selection sort

```
public static void selection_sort(int [] a, int ini){

    if(ini == a.length - 1) return;

    int min_index = min(a, ini);

    troca(a, ini, min_index);

    selection_sort(a, ini + 1);

}
```

# Selection sort

```
public static void selection_sort(int [] a, int ini){

    if(ini == a.length - 1) return;

    int min_index = min(a, ini);

    troca(a, ini, min_index);

    selection_sort(a, ini + 1);

}
```

# Selection sort

```
public static void selection_sort(int [] a, int ini){
    if(ini == a.length - 1) return;                         Θ(1)
    int min_index = min(a, ini);
    troca(a, ini, min_index);
    selection_sort(a, ini + 1);
}
```

# Selection sort

```
public static void selection_sort(int [] a, int ini){
    if(ini == a.length - 1) return;                    Θ(1)
    int min_index = min(a, ini);
    troca(a, ini, min_index);
    selection_sort(a, ini + 1);
}
```

# Selection sort

```
public static void selection_sort(int [] a, int ini){
    if(ini == a.length - 1) return;                    Θ(1)
    int min_index = min(a, ini);                        Θ(n)
    troca(a, ini, min_index);
    selection_sort(a, ini + 1);
}
```

# Selection sort

```
public static void selection_sort(int [] a, int ini){

    if(ini == a.length - 1) return;                          Θ(1)

    int min_index = min(a, ini);                             Θ(n)

    troca(a, ini, min_index);

    selection_sort(a, ini + 1);

}
```

# Selection sort

```
public static void selection_sort(int [] a, int ini){
    if(ini == a.length - 1) return;                    Θ(1)
    int min_index = min(a, ini);                       Θ(n)
    troca(a, ini, min_index);                          Θ(1)
    selection_sort(a, ini + 1);
}
```

# Selection sort

```
public static void selection_sort(int [] a, int ini){
    if(ini == a.length - 1) return;                  Θ(1)
    int min_index = min(a, ini);                     Θ(n)
    troca(a, ini, min_index);                        Θ(1)
    selection_sort(a, ini + 1);
}
```

# Selection sort

```
public static void selection_sort(int [] a, int ini){
    if(ini == a.length - 1) return;                       Θ(1)
    int min_index = min(a, ini);                          Θ(n)
    troca(a, ini, min_index);                             Θ(1)
    selection_sort(a, ini + 1);                           T(n - 1)
}
```

# Selection sort

```
public static void selection_sort(int [] a, int ini){
    if(ini == a.length - 1) return;              Θ(1)
    int min_index = min(a, ini);                 Θ(n)
    troca(a, ini, min_index);                    Θ(1)
    selection_sort(a, ini + 1);                  T(n - 1)
}
```

$T(n) = T(n - 1) + \Theta(n) + \Theta(1)$

$T(n) = T(n - 1) + \Theta(n)$

$\mathbf{T(n) = \Theta(n^2)}$

# Merge sort

Base do algoritmo: método/função **merge**

- dois subvetores já ordenados com n/2 elementos.
- une os dois subvetores em um vetor ordenado com n elementos.
- compara apenas os primeiros elementos de cada subvetor.
- n/2 =< #comparações <= n - 1
- merge é $\Theta(n)$ [ onde n é o total de elementos juntados no processo ]

# Merge sort (motivação)

Temos o selection sort que é Θ(n^2). Logo o custo de ordenação é ~ cn^2

Temos o merge que é Θ(n). Portanto, o custo de juntar n elementos é ~ kn

E se, para ordenar um vetor a com n elementos, o partirmos em duas partes iguais (n/2 elementos), ordenar cada uma delas com o selection sort, e uní-las em seguida com o merge?

# Merge sort (motivação)

```
{ 5, 3, 4, 7, 2, 0, 1, 6 }

{ 5, 3, 4, 7 }        {2, 0, 1, 6 }

{ 3, 4, 5, 7 }        {0, 1, 2, 6 }

{ 0, 1, 2, 3, 4, 5, 6, 7 }
```

# Merge sort (motivação)

{ 5, 3, 4, 7, 2, 0, 1, 6 }

{ 5, 3, 4, 7 }　　{2, 0, 1, 6 }

{ 3, 4, 5, 7 }　　{0, 1, 2, 6 }

{ 0, 1, 2, 3, 4, 5, 6, 7 }

Θ(1)

2[ c(n/2)^2 ] = 2c(n^2/4) = cn^2 / 2

kn

Total: cn^2 / 2 + kn + Θ(1) = O(n^2)

# Merge sort (motivação)

{ 5, 3, 4, 7, 2, 0, 1, 6 }

{ 5, 3, 4, 7 }   {2, 0, 1, 6 }

{ 3, 4, 5, 7 }   {0, 1, 2, 6 }

{ 0, 1, 2, 3, 4, 5, 6, 7 }

Θ(1)

2[ c(n/2)^2 ] = 2c(n^2/4) = cn^2 / 2

kn

Total: cn^2 / 2 + kn + Θ(1) = O(n^2)

# Merge sort (motivação)

Temos o selection sort que é Θ(n^2). Logo o custo de ordenação é ~ cn^2

Temos o merge que é Θ(n). Portanto, o custo de juntar n elementos é ~ kn

E se, para ordenar um vetor a com n elementos, o partirmos em duas partes iguais (n/2 elementos), ordenar cada uma delas com o selection sort, e uní-las em seguida com o merge? Não muda a complexidade, mas na prática o tempo é reduzido pela metade.

# Merge sort (motivação)

Temos o selection sort que é $\Theta(n^2)$. Logo o custo de ordenação é ~ $cn^2$

Temos o merge que é $\Theta(n)$. Portanto, o custo de juntar n elementos é ~ $kn$

E se, para ordenar um vetor a com n elementos, o partirmos em duas partes iguais (n/2 elementos), ordenar cada uma delas com o selection sort, e uní-las em seguida com o merge? Não muda a complexidade, mas na prática o tempo é reduzido pela metade.

E se, para ordenar um vetor a com n elementos, o partirmos em 4 partes iguais (n/4 elementos), ordenar cada uma delas com o selection sort, e uní-las em seguida com o merge?

# Merge sort (motivação)

{ 5, 3, 4, 7, 2, 0, 1, 6 }

{ 5, 3 } { 4, 7 } { 2, 0 } { 1, 6 }

{ 3, 5 } { 4, 7 } { 0, 2 } { 1, 6 }

{ 3, 4, 5, 7 } {0, 1, 2, 6 }

{ 0, 1, 2, 3, 4, 5, 6, 7 }

# Merge sort (motivação)

{ 5, 3, 4, 7, 2, 0, 1, 6 }

$\Theta(1)$

{ 5, 3 } { 4, 7 } { 2, 0 } { 1, 6 }

4[ c(n/4)^2 ] = cn^2 / 4

{ 3, 5 } { 4, 7 } { 0, 2 } { 1, 6 }

kn

{ 3, 4, 5, 7 } {0, 1, 2, 6 }

kn

{ 0, 1, 2, 3, 4, 5, 6, 7 } Total: cn^2 / 4 + 2kn + $\Theta(1)$ = $\Theta$(n^2)

# Merge sort (motivação)

{ 5, 3, 4, 7, 2, 0, 1, 6 }

$\Theta(1)$

{ 5, 3 } { 4, 7 } { 2, 0 } { 1, 6 }

4[ c(n/4)^2 ] = cn^2 / 4

{ 3, 5 } { 4, 7 } { 0, 2 } { 1, 6 }

kn

{ 3, 4, 5, 7 } {0, 1, 2, 6 }

kn

{ 0, 1, 2, 3, 4, 5, 6, 7 }

Total: cn^2 / 4 + 2kn + $\Theta$(1) = $\Theta$(n^2)

# Merge sort (motivação)

Temos o selection sort que é $\Theta(n^2)$. Logo o custo de ordenação é ~ $cn^2$

Temos o merge que é $\Theta(n)$. Portanto, o custo de juntar n elementos é ~ $kn$

E se, para ordenar um vetor a com n elementos, o partirmos em duas partes iguais (n/2 elementos), ordenar cada uma delas com o selection sort, e uní-las em seguida com o merge? Não muda a complexidade, mas na prática o tempo é reduzido pela metade.

E se, para ordenar um vetor a com n elementos, o partirmos em 4 partes iguais (n/4 elementos), ordenar cada uma delas com o selection sort, e uní-las em seguida com o merge? Não muda a complexidade, mas na prática o tempo é reduzido a 1/4.

# Merge sort

Perceba que, quanto menor o tamanho do subvetor a ser ordenado pelo selection sort, maior o fator de redução do termo quadrático da função que representa o tempo total da ordenação.

Perceba também que se os subvetores chegarem ao tamanho mínimo de 1, o selection sort se torna desnecessário ao processo de ordenação.

Neste caso, o que ordenaria, de fato, o vetor são sucessivos merges:

$$n \times 1 \rightarrow (n/2) \times 2 \rightarrow (n/4) \times 4 \rightarrow (n/8) \times 8 \rightarrow \ldots \rightarrow 2 \times (n/2) \rightarrow 1 \times n$$

com um custo combinado do merge de O(n) por nível. Como, no total, a quantidade de níveis é $\Theta(\log_2 n)$, então temos um algoritmo de ordenação que é $O(n\log_2 n)$.

# Merge sort

Perceba que, quanto menor o tamanho do subvetor a ser ordenado pelo selection sort, maior o fator de redução do termo quadrático da função que representa o tempo total da ordenação.

Perceba também que se os subvetores chegarem ao tamanho mínimo de 1, o selection sort se torna desnecessário ao processo de ordenação.

Neste caso, o que ordenaria, de fato, o vetor são sucessivos merges:

$$n \times 1 \rightarrow (n/2) \times 2 \rightarrow (n/4) \times 4 \rightarrow (n/8) \times 8 \rightarrow \ldots \rightarrow 2 \times (n/2) \rightarrow 1 \times n$$

com um custo combinado do merge de $O(n)$ por nível. Como, no total, a quantidade de níveis é $\Theta(\log_2 n)$, então temos um algoritmo de ordenação que é $O(n\log_2 n)$. E este é o famoso Merge sort! :)

# Merge sort (código)

```
public static void merge_sort(int [] a, int ini, int fim){

    if(ini < fim){

        int med = (ini + fim) / 2;
        merge_sort(a, ini, med);
        merge_sort(a, med + 1, fim);
        merge(a, ini, med, fim);
    }
}
```

# Merge sort (código)

```
public static void merge_sort(int [] a, int ini, int fim){

    if(ini < fim){                                   Θ(1)

        int med = (ini + fim) / 2;                   Θ(1)
        merge_sort(a, ini, med);                     T(n/2)
        merge_sort(a, med + 1, fim);                 T(n/2)
        merge(a, ini, med, fim);                     Θ(n)
    }
}
```

# Merge sort (código)

```
public static void merge_sort(int [] a, int ini, int fim){

    if(ini < fim){                                  Θ(1)

        int med = (ini + fim) / 2;                  Θ(1)
        merge_sort(a, ini, med);                    T(n/2)
        merge_sort(a, med + 1, fim);                T(n/2)
        merge(a, ini, med, fim);                    Θ(n)
    }
}
```

$T(n) = T(n/2) + T(n/2) + \Theta(n) + \Theta(1)$

$T(n) = 2T(n/2) + \Theta(n)$

$\mathbf{T(n) = \Theta(n\log_2 n)}$ (segundo o Teorema Mestre)

# Quicksort

Base do algoritmo: método/função particiona

- particiona um vetor com n elementos em duas partições (partes).
- não necessariamente do mesmo tamanho (ao menos 1 elemento p/ partição).
- de modo que qualquer elemento da primeira partição seja menor ou igual a qualquer elemento da segunda partição.
- leva tempo $\Theta(n)$

# Quicksort

Base do algoritmo: método/função **particiona**

- particiona um vetor com n elementos em duas partições (partes).
- não necessariamente do mesmo tamanho (ao menos 1 elemento p/ partição).
- de modo que qualquer elemento da primeira partição seja menor ou igual a qualquer elemento da segunda partição.
- leva tempo $\Theta(n)$

Exemplo:

```
{ 5, 3, 8, 2, 7, 10, 1, 4,  0 }
```

# Quicksort

Base do algoritmo: método/função **particiona**

- particiona um vetor com n elementos em duas partições (partes).
- não necessariamente do mesmo tamanho (ao menos 1 elemento p/ partição).
- de modo que qualquer elemento da primeira partição seja menor ou igual a qualquer elemento da segunda partição.
- leva tempo Θ(n)

Exemplo:

```
{ 5, 3, 8, 2, 7, 10, 1, 4,  0 }
{ 3, 2, 1, 4, 0,  5, 8, 7, 10 }       q = 4
```

# Quicksort

Base do algoritmo: método/função particiona

- particiona um vetor com n elementos em duas partições (partes).
- não necessariamente do mesmo tamanho (ao menos 1 elemento p/ partição).
- de modo que qualquer elemento da primeira partição seja menor ou igual a qualquer elemento da segunda partição.
- leva tempo Θ(n)

Exemplo:

```
{ 5, 3, 8, 2, 7, 10, 1, 4,  0 }
{ 3, 2, 1, 4, 0,  5, 8, 7, 10 }        q = 4

a[i] <= a[j], para todo 0 <= i <= q e (q + 1) <= j <= (n - 1)
```

# Quicksort

Um particionamento apenas não é suficiente para ordenar um vetor…

Mas já é um primeiro passo importante no processo de ordenação.

Embora os elementos dentro de uma partição ainda precisem ser reorganizados para que o vetor como um todo fique ordenado, perceba que eles não mais sairão daquela partição.

Como dar um novo passo no processo de ordenação? Simples, basta chamar o método/função particiona para cada partição obtida.

O que nos leva ao seguinte algoritmo recursivo...

# Quicksort

```
public static void quicksort(int [] a, int ini, int fim){

    if(ini < fim){

        int q = particiona(a, ini, fim);
        quicksort(a, ini, q);
        quicksort(a, q + 1, fim);
    }
}
```

# Quicksort

```
public static void quicksort(int [] a, int ini, int fim){

    if(ini < fim){                                   Θ(1)

        int q = particiona(a, ini, fim);             Θ(n)
        quicksort(a, ini, q);                        T(x)
        quicksort(a, q + 1, fim);                    T(n - x)
    }
}
```

# Quicksort

```
public static void quicksort(int [] a, int ini, int fim){

    if(ini < fim){                                  Θ(1)

        int q = particiona(a, ini, fim);            Θ(n)
        quicksort(a, ini, q);                       T(x)
        quicksort(a, q + 1, fim);                   T(n - x)
    }
}
```

1 <= x <= (n - 1): tamanho da primeira partição

$$T(n) = T(x) + T(n - x) + \Theta(n) + \Theta(1)$$

$$T(n) = T(x) + T(n - x) + \Theta(n)$$

# Quicksort

Dois cenários extremos para o Quicksort:

Melhor caso: partição balanceada (em todos os particionamentos) $\longrightarrow$ `x = n/2`

```
T(n) = T(n/2) + T(n/2) + Θ(n)
T(n) = 2T(n/2) + Θ(n)                [ mesma recorrência do Mergesort!
]    T(n) = Θ(nlog₂n)
```

# Quicksort

Dois cenários extremos para o Quicksort:

Pior caso: partição o mais desbalanceada possível ⟶ `x = 1 (ou n - 1)`

$$T(n) = T(1) + T(n - 1) + \Theta(n)$$

$$T(n) = \Theta(1) + T(n - 1) + \Theta(n)$$

$$T(n) = T(n - 1) + \Theta(n)$$

$$\mathbf{T(n) = \Theta(n^2)}$$

# Quicksort

Sem especificar um cenário particular, temos:

$$T(n) = \Omega(n\log_2 n)$$

$$T(n) = O(n^2)$$

O que quer dizer que o Quicksort pode ser tão bom quanto Merge sort, mas tão ruim quanto o Selection sort (e outros algoritmos mais simples que são quadráticos).

Na prática (caso médio), o que podemos esperar?

# Quicksort

Sem especificar um cenário particular, temos:

$$T(n) = \Omega(n\log_2 n)$$
$$T(n) = O(n^2)$$

O que quer dizer que o Quicksort pode ser tão bom quanto Merge sort, mas tão ruim quanto o Selection sort (e outros algoritmos mais simples que são quadráticos).

Na prática (caso médio), o que podemos esperar? Contrariando o que costuma acontecer, o caso médio do Quicksort está mais próximo do melhor caso do que do pior caso. Logo, o Quicksort tem complexidade média $\Theta(n\log_2 n)$.

# Quicksort

Por que a complexidade média é $\Theta(n\log_2 n)$? Algumas intuições...

- Um particionamento desbalanceado mas que, partindo de um problema de tamanho n, gera subproblemas de tamanhos `(n/b)` e `((b - 1)*n)/b` (exemplo: n/10 e 9n/10), e supondo que tal proporção ocorra em todas as chamadas ao particiona, ainda irá resultar em uma árvore de recursão com profundidade $\Theta(\log_2 n)$ e em uma complexidade $\Theta(n\log_2 n)$ em relação ao tempo de execução.

- Particionamentos ruins não devem acontecer sempre, e um ou outro particionamentos ruins podem ser “absorvidos” por particionamentos bons.

# Heapsort

Heapsort pode ser visto como um Selection sort melhorado.

Selection sort: seleção do menor valor (varredura linear)
menor valor colocado no início do vetor

Heapsort: seleção do maior valor (usando um heap máximo)
maior valor colocado no final do vetor

O uso da heap (fila de prioridade) torna a seleção no Heapsort muito mais eficiente do que no Selection sort.

# Heapsort

Heapsort pode ser visto como um Selection sort melhorado.

Selection sort: seleção do menor valor (varredura linear)
menor valor colocado no início do vetor

Heapsort: seleção do maior valor (usando um heap máximo)
maior valor colocado no final do vetor

O uso do heap (fila de prioridade) torna a seleção no Heapsort muito mais eficiente do que no Selection sort. No total continuam sendo n seleções, mas ao custo $O(\log_2 n)$, ao invés de custo $\Theta(n)$ da varredura linear, o que resulta em uma complexidade $O(n \log_2 n)$.

# Heapsort (versão simplificada)

```
public static void heapsort(int [] a){

    Heap heap = new Heap(a);

    for(int i = a.length - 1; i >= 0; i--){

        a[i] = heap.extractMax();
    }
}
```

# Heapsort (versão simplificada)

```
public static void heapsort(int [] a){

    Heap heap = new Heap(a);                          O(nlog₂n)

    for(int i = a.length - 1; i >= 0; i--){

        a[i] = heap.extractMax();                     n * [ O(log₂n) + Θ(1) ]
    }
}
```

$O(n\log_2 n)$

$n * [\ O(\log_2 n) + \Theta(1)\ ]$

# Heapsort (versão simplificada)

```
public static void heapsort(int [] a){

    Heap heap = new Heap(a);                          O(nlog₂n)

    for(int i = a.length - 1; i >= 0; i--){

        a[i] = heap.extractMax();                     n * [ O(log₂n) + Θ(1) ]
    }
}
```

$T(n) = O(n\log_2 n) + n * [\ O(\log_2 n) + \Theta(1)\ ]$

$T(n) = O(n\log_2 n) + O(n\log_2 n) + \Theta(n)$

$\mathbf{T(n) = O(n\log_2 n)}$

# Heapsort (versão simplificada)

```
public static void heapsort(int [] a){

    Heap heap = new Heap(a);                          O(nlog₂n)

    for(int i = a.length - 1; i >= 0; i--){

        a[i] = heap.extractMax();                     n * [ O(log₂n) + Θ(1) ]
    }
}
```

$T(n) = O(n\log_2 n) + n * [\ O(\log_2 n) + \Theta(1)\ ]$

$T(n) = O(n\log_2 n) + O(n\log_2 n) + \Theta(n)$

$\mathbf{T(n) = O(n\log_2 n)}$

Esta versão poderia ser melhorada ao transformar o próprio vetor **a** em um heap, e usar a porção final do vetor para ir montando a sequência ordenada.

# Limite inferior no número de comparações

Todos os algoritmos mencionados e descritos até aqui possuem algo em comum: são todos algoritmos baseados em comparação.

# Limite inferior no número de comparações

Todos os algoritmos mencionados e descritos até aqui possuem algo em comum: são todos algoritmos baseados em comparação.

Isto é, a ordenação é obtida pela realização de diversas comparações entre pares de elementos presentes na sequência a ser ordenada.

# Limite inferior no número de comparações

Todos os algoritmos mencionados e descritos até aqui possuem algo em comum: são todos algoritmos baseados em comparação.

Isto é, a ordenação é obtida pela realização de diversas comparações entre pares de elementos presentes na sequência a ser ordenada.

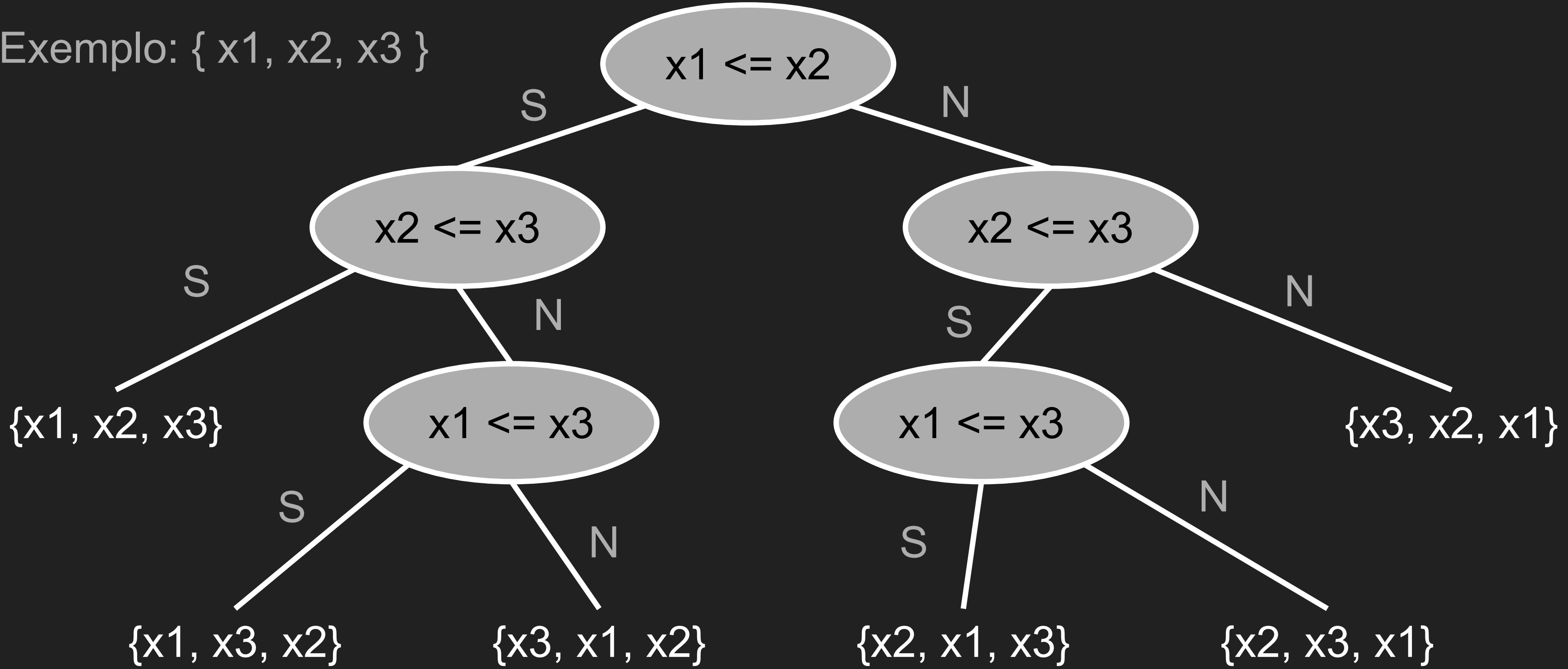
É possível determinar a quantidade mínima de comparações necessárias para ordenar uma sequência de tamanho n?

# Limite inferior no número de comparações

Ordenação de uma sequência de tamanho n é análoga a navegar por uma árvore de decisão, onde cada nó representa uma comparação entre dois valores.

# Limite inferior no número de comparações

Exemplo: { x1, x2, x3 }



# Limite inferior no número de comparações

Ordenação de uma sequência de tamanho n é análoga a navegar por uma árvore de decisão, onde cada nó representa uma comparação entre dois valores.

Na pior das hipóteses (ou seja, quando mais comparações são necessárias), temos que o número de comparações será igual à altura da árvore (**h**).

Sabendo que temos **n!** permutações de sequências de tamanho n, e cada uma dessas permutações deve aparecer como uma folha da árvore de decisão, que é uma árvore binária, temos que:

# Limite inferior no número de comparações

O número máximo de folhas que uma árvore de altura h comporta é $2^h$. Logo:

$n! <= 2^h$ -----> $h >= lg(n!)$

Sabendo que $n! > (n / e)^n$ (aproximação de Stirling):

$h >= lg(n!) > lg( (n / e)^n )$

$h > n [ lg(n) - lg(e) ]$ -----> $h > nlg(n) - nlg(e)$

$h > \Theta(nlgn)$

**$h = \Omega(nlgn)$**

# Limite inferior no número de comparações

Logo, não é possível ordenar uma sequência de n elementos, fazendo menos do que $\Theta(n\lg n)$ comparações.

Portanto, tanto o *merge sort* quanto o *heapsort* são ótimos neste sentido. E o *quicksort* é ótimo na média.

# Algoritmos de ordenação sem comparação

Existem algoritmos de ordenação com complexidade menor que $\Theta(n \lg n)$?

Sim, há algoritmos que possuem complexidade linear, mas eles usam abordagens um pouco diferentes, sem envolver realizar comparações entre elementos presentes na sequência sendo ordenados:

Exemplos: counting sort, radix sort.