Ordenação

Marco A L Barbosa
malbarbo.pro.br

Departamento de Informática
Universidade Estadual de Maringá

Junto com o problema de busca, o problema de ordenação é um dos mais estudados da Computação.

O problema de ordenação consiste em, dado uma sequência de n números $\langle a_1, a_2, \ldots, a_n \rangle$, determinar uma permutação (reordenação) $\langle a'_1, a'_2, \ldots, a'_n \rangle$ da sequência de entrada tal que, $a'_1 \leq a'_2 \leq \cdots \leq a'_n$.

Para avaliarmos os algoritmos de ordenação, além da complexidade de tempo, consideramos o uso extra de memória e se a ordenação é estável.

Um algoritmo é **in-loco** se a quantidade de memória que ele precisa para executar é $O(1)$, ou seja, não depende do tamanho da entrada.

Um algoritmo de ordenação é **estável** se a ordem relativamente dos elementos com a mesma chave é preservado.

Veremos agora o projeto de diversos algoritmos de ordenação. Eles são baseados nas seguintes técnicas de projeto:

- Incremental
- Divisão e conquista
- *Ad hoc* (específicos, sem técnica geral)

Projeto de algoritmos incremental

A ideia de um algoritmo incremental é:

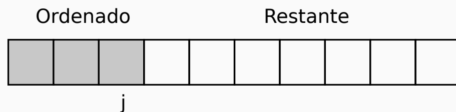
- Iniciar com a solução para um problema trivial;
- Estender a solução iterativamente para um problema maior até obter a solução do problema que queremos resolver;

Como projetar um algoritmo incremental para somar os elementos de um arranjo?

- Começamos com a soma do arranjo vazio que é 0;
- Estendemos a soma adicionando um elemento do arranjo por vez até que todos os elementos tenham sido somados.

Como projetar um algoritmo incremental para ordenar os elementos de um arranjo?

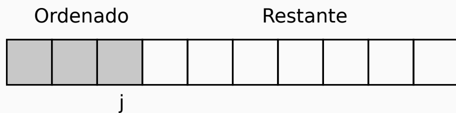
- Iniciamos com um subarranjo vazio já ordenado;
- Estendemos o subarranjo já ordenado selecionando um elemento por vez até que todos os elementos tenham sido selecionados.



Temos que tomar duas decisões para tornar o processo concreto:

- Como selecionar o próximo elemento?
- Como estender o subarranjo ordenado?

Ordenação por inserção



Como selecionar o próximo elemento?

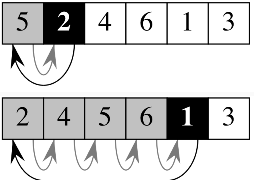
- Pegamos o primeiro elemento do restante.
- Qual é o custo? $O(1)$

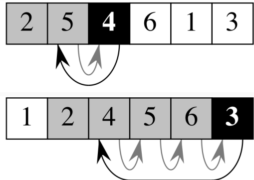
Como estender o subarranjo ordenado?

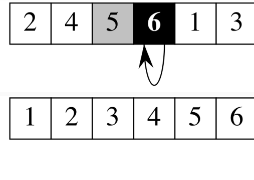
- Inserindo o elemento selecionado na parte ordenada.
- Qual é o custo? $O(j)$

Este algoritmo é conhecido como **ordenação por inserção** (*insertion sort*).

Ordenação por inserção







Ordenação por inserção

Projete uma função que implemente o algoritmo de ordenação por inserção.

```
def ordena_insercao(lst: list[int]):
    '''
    Ordena *lst* em ordem não decrescente usando
    o algoritmo de ordenação por inserção.
    Exemplos
    >>> lst = [5, 2, 4, 6, 1, 3]
    >>> ordena_insercao(lst)
    >>> lst
    [1, 2, 3, 4, 5, 6]
    '''

    for i in range(1, len(lst)):
        j = i
        while j > 0 and lst[j - 1] > lst[j]:
            lst[j - 1], lst[j] = lst[j], lst[j - 1]
            j -= 1
```

Qual é a complexidade de tempo da ordenação por inserção?

Qual é o melhor caso? `lst` está em ordem não decrescente, o corpo do **`while`** não é executado nenhuma vez. A complexidade de tempo é $O(n)$.

Qual é o pior caso? `lst` está em ordem não crescente, na iteração `i` o corpo do **`while`** é executado `i` vezes. A complexidade de tempo é $\sum_{i=1}^{n-1} i = O(n^2)$.

A implementação é in-loco? Sim.

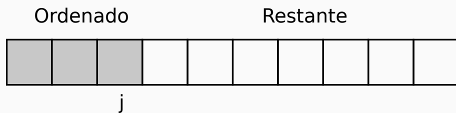
A ordenação é estável? Sim.

Podemos melhor o tempo?

Temos dois custos, o de seleção, que é $O(1)$, e o de inserção, que é $O(j)$. Podemos melhor o tempo da inserção?

Considerando que o subarranjo `lst[:j]` está ordenado, poderíamos usar uma busca binária para encontrar a posição de inserção em $O(\lg j)$. No entanto, a inserção continuaria sendo $O(j)$, pois os elementos precisar ser deslocados.

Podemos fazer uma “inserção” sem fazer o deslocamento dos elementos? Sim!



Como selecionar o próximo elemento?

- Pegamos o menor elemento do restante.
- Qual é o custo? $O(n - j)$

Como estender o subarranjo ordenado?

- Trocando de posição o menor elemento com o primeiro do restante.
- Qual é o custo? $O(1)$

Este algoritmo é conhecido como **ordenação por seleção** (*selection sort*).

Projete uma função que implemente o algoritmo de ordenação por seleção.

```
def ordena_selecao(lst: list[int]):
    '''
    Ordena *lst* em ordem não decrescente usando
    o algoritmo de ordenação por inserção.
    Exemplos
    >>> lst = [5, 2, 4, 6, 1, 3]
    >>> ordena_insercao(lst)
    >>> lst
    [1, 2, 3, 4, 5, 6]
    '''

    for i in range(len(lst)):
        m = i # índice do menor em lst[i:]
        for j in range(i + 1, len(lst)):
            if lst[j] < lst[m]:
                m = j
        lst[i], lst[m] = lst[m], lst[i]
```

Qual é a complexidade de tempo da ordenação por seleção?

Cada iteração `i` o corpo do segundo **`for`** é executado `n - i - 1` vezes. A complexidade de tempo é

$$\sum_{i=0}^{n-1} n - i - 1 = O(n^2).$$

A implementação é in-loco? Sim.

A ordenação é estável? Não.

A ordenação por seleção é mais eficiente do que a ordenação por inserção? Não...

Quando projetamos o algoritmo de ordenação por seleção nós movemos o custo de inserção para a seleção e vice e versa... Parece que não ganhamos nada!

Mas isso não é verdade, agora podemos abordar o problema por outro ângulo!

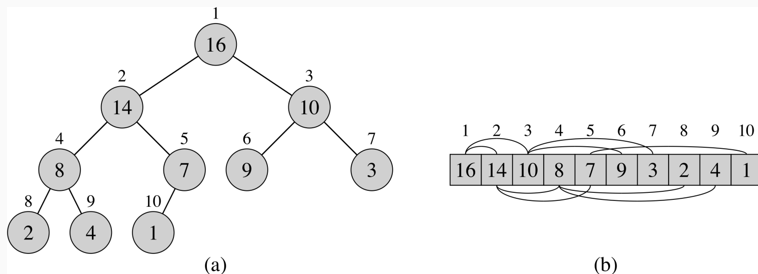
Antes

Tentamos diminuir o tempo para inserir em um arranjo ordenado (parece muito rígido).

Agora

Vamos tentar diminuir o tempo para selecionar o valor mínimo de um arranjo (parece mais flexível).

Um **heap** (binário) é um arranjo que pode ser visto como uma árvore binária quase completa:



Cada nó da árvore corresponde ao elemento do arranjo que armazena o valor do nó.

A árvore está preenchida em todos os níveis, exceto talvez no nível mais baixo, que é preenchido a partir da esquerda.

Note que nesse exemplo o arranjo é indexado a partir de 1!

Como o heap pode ser visto como árvore, ele também tem uma altura, que é $O(\lg n)$.

É essa característica que permite que as operações em um heap sejam eficientes.

Para arranjos indexados a partir de 0, a raiz do heap está na posição 0. Além disso, para cada nó no índice i, os índices do pai e dos filhos a direita e a esquerda podem ser caculado da seguinte forma:

$\textsc{Pai}(i) = \lfloor (i-1)/2 \rfloor$

$\textsc{Esq}(i) = 2i + 1$

$\textsc{Dir}(i) = 2i + 2$

Propriedade do heap

Em um **heap máximo** armazenado em um arranjo A, para cada nó i diferente da raiz

$$A[\textsc{pai}(i)] \geq A[i]$$

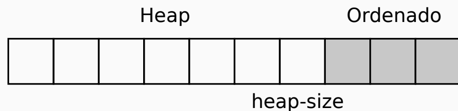
Em um **heap mínimo** armazenado em um arranjo A, para cada nó i diferente da raiz

$$A[\textsc{pai}(i)] \leq A[i]$$

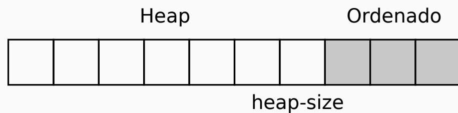
Em um heap máximo, onde está o maior elemento? Na raiz.

Em um heap mínimo, onde está o menor elemento? Na raiz.

Como utilizar um heap máximo em um processo de ordenação incremental?



- Mantemos a porção ordenada no final do arranjo;
- E o heap na porção inicial.



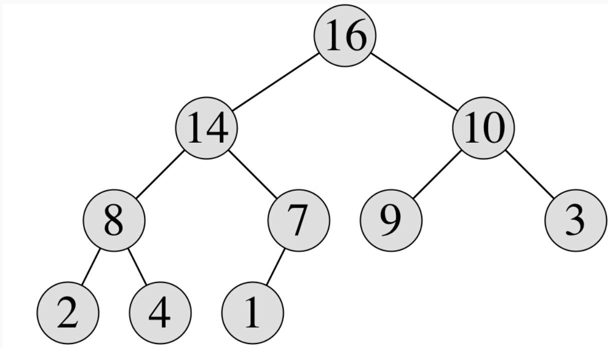
Como selecionar o próximo elemento?

- Pegamos o maior elemento do heap.
- Qual é o custo? $O(\lg(heap - size))$ – veremos isso a seguir.

Como estender o subarranjo ordenado?

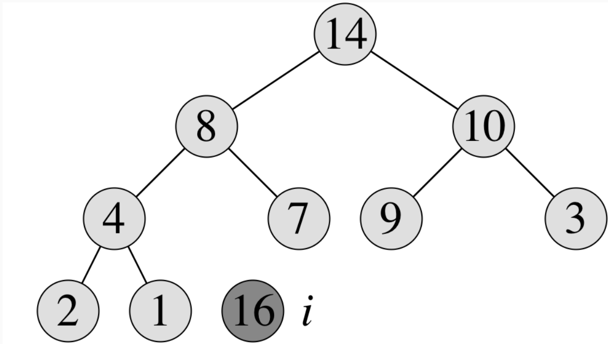
- Trocando de posição o maior elemento com o último do restante.
- Qual é o custo? $O(1)$.

Este algoritmo é conhecido como **ordenação por heap** (*heap sort*).

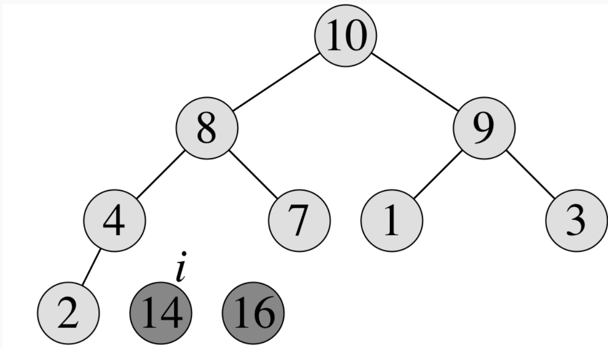



```
| 16 | 14 | 10 |  8 |  7 |  9 |  3 |  2 |  4 |  1 |
\_____________________ heap _____________________/
```

Exemplo da ordenação por heap

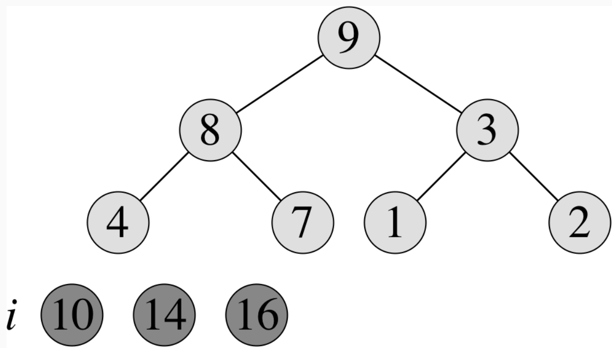


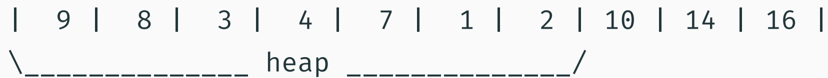
Exemplo da ordenação por heap



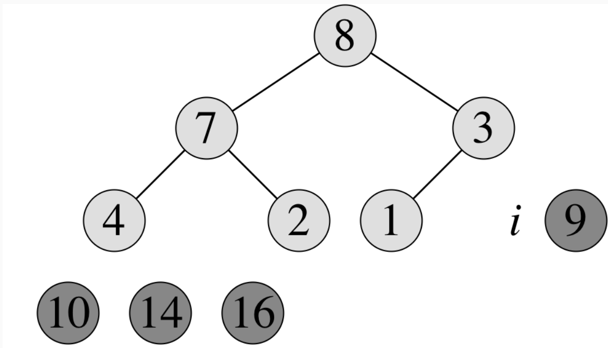

```
| 10 |  8 |  9 |  4 |  7 |  1 |  3 |  2 | 14 | 16 |
\________________ heap ________________/
```

Exemplo da ordenação por heap

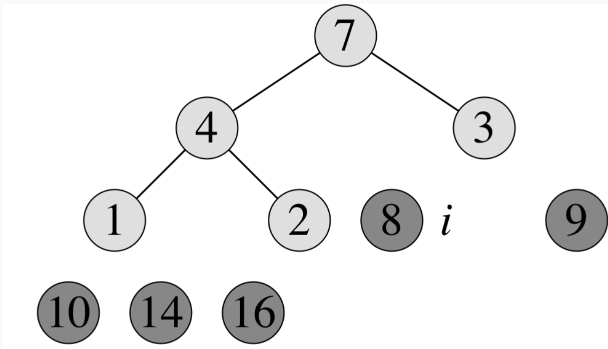




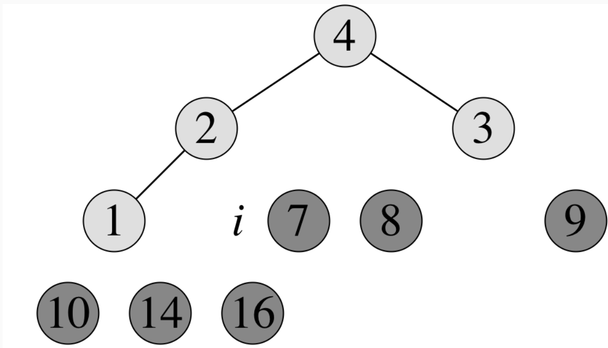
Exemplo da ordenação por heap



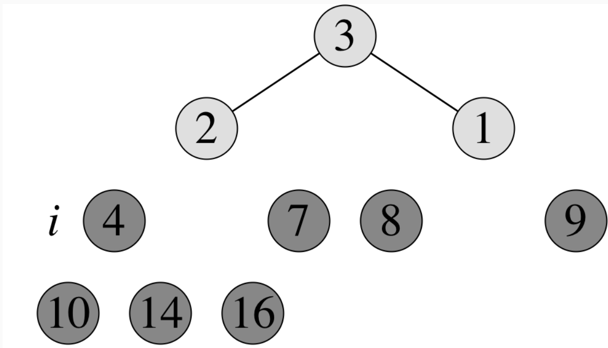

```
|  8 |  7 |  3 |  4 |  2 |  1 |  9 | 10 | 14 | 16 |
\___________ heap ____________/
```



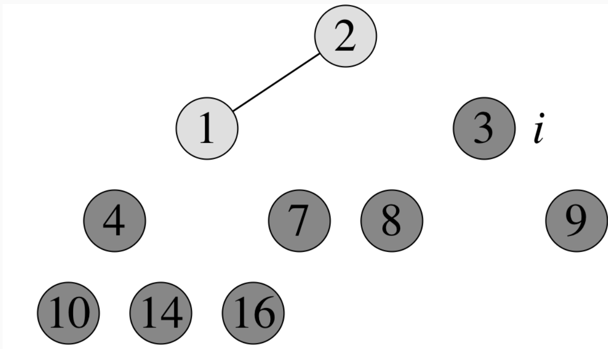

```
|  7 |  4 |  3 |  1 |  2 |  8 |  9 | 10 | 14 | 16 |
\_________ heap _________/
```



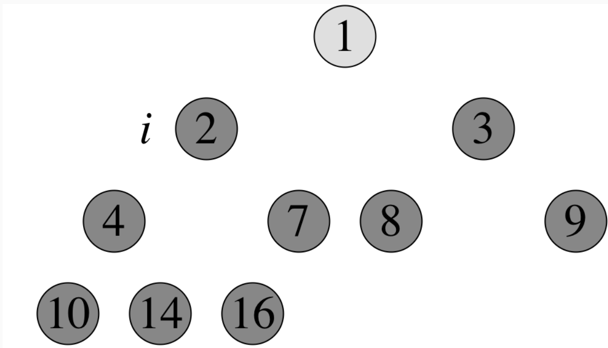

```
|  4 |  2 |  3 |  1 |  7 |  8 |  9 | 10 | 14 | 16 |
\______ heap _______/
```




```
|  3 |  2 |  1 |  4 |  7 |  8 |  9 | 10 | 14 | 16 |
\____ heap ____/
```




```
|  2 |  1 |  3 |  4 |  7 |  8 |  9 | 10 | 14 | 16 |
\_ heap __/
```



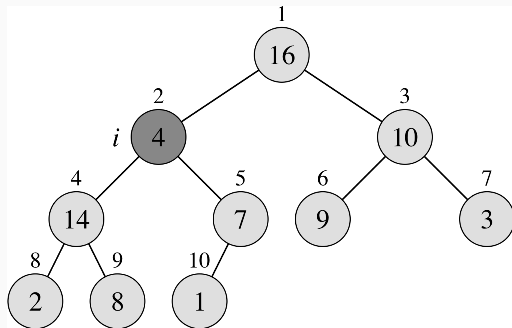

```
|   1 |   2 |   3 |   4 |   7 |   8 |   9 | 10 | 14 | 16 |
\heap/
```

Operações com heap

Que operações precisamos para implementar a ordenação por heap?

- Inicialização do heap
- Remoção do máximo

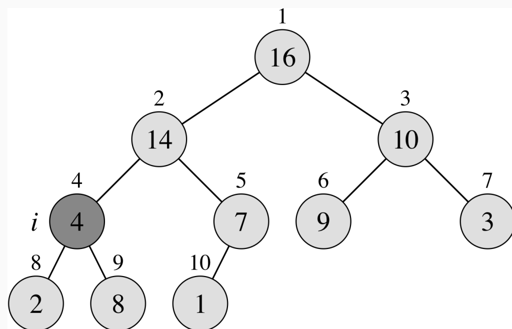
Para implementar essas funções, vamos precisar de uma operação auxiliar, que “concerta” um heap.



Seja A um arranjo que armazena um heap máximo.

Considerando que o elemento da posição i foi alterado, como podemos verificar se a propriedade do heap se mantém, e caso contrário, como podemos "concertar" o heap?

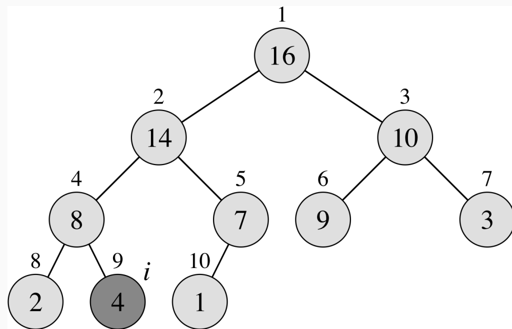
Verificamos se $A[i]$ é menor que algum dos dois filhos, se sim, trocamos $A[i]$ de lugar com o maior filho,



Seja A um arranjo que armazena um heap máximo.

Considerando que o elemento da posição i foi alterado, como podemos verificar se a propriedade do heap se mantém, e caso contrário, como podemos “concertar” o heap?

Verificamos se $A[i]$ é menor que algum dos dois filhos, se sim, trocamos $A[i]$ de lugar com o maior filho, depois executamos o processo recursivamente para o filho que foi trocado.



Seja A um arranjo que armazena um heap máximo.

Considerando que o elemento da posição i foi alterado, como podemos verificar se a propriedade do heap se mantém, e caso contrário, como podemos "concertar" o heap?

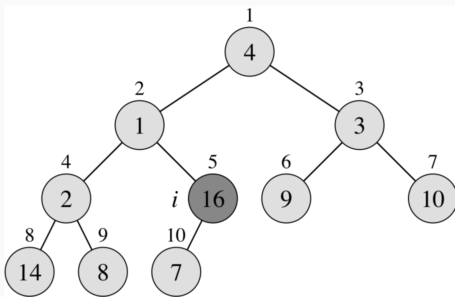
Verificamos se $A[i]$ é menor que algum dos dois filhos, se sim, trocamos $A[i]$ de lugar com o maior filho, depois executamos o processo recursivamente para o filho que foi trocado.

Projete uma função que receba com parâmetro um arranjo A, a quantidade de elementos n de A que estão sendo usados e um índice i, onde os elementos `esq(i)` e `dir(i)` são raízes de heap máximo, e "concerte" o arranjo, se necessário, para que a árvore com raiz i seja um heap máximo.

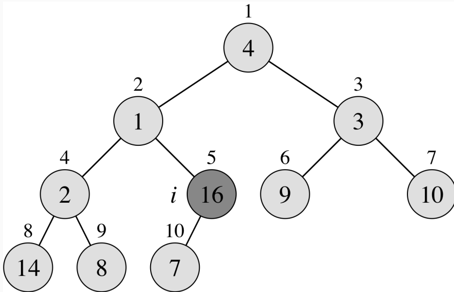
```
def concerta_heap(A: list[int], n: int, i: int):
    assert i < n <= len(A)
    # Encontra o índice do maior entre
    # A[i], A[esq(i)] e A[dir(i)]
    fesq = esq(i)
    fdir = dir(i)
    imax = i
    if fesq < n and A[fesq] > A[imax]:
        imax = fesq
    if fdir < n and A[fdir] > A[imax]:
        imax = fdir
    # Se o maior não é A[i], ajusta e repete
    # o processo.
    if imax != i:
        A[i], A[imax] = A[imax], A[i]
        concerta_heap(A, n, imax)
```

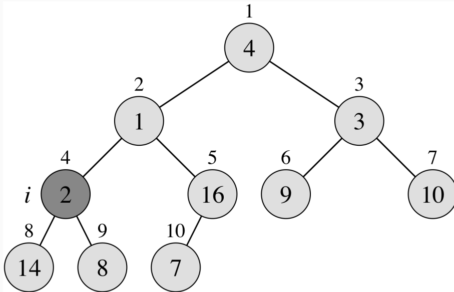
Qual é a complexidade de tempo? $O(h)$, onde h é a altura do heap, ou seja, $O(\lg n)$.

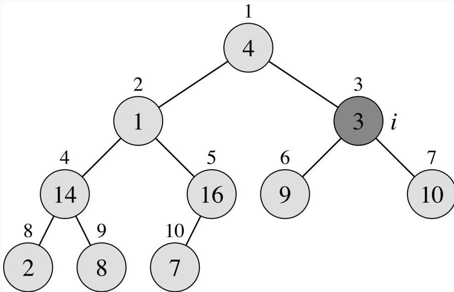
Como construir um heap? Vamos começar com o que está certo e ir "consertando" até que todo o heap fique certo.

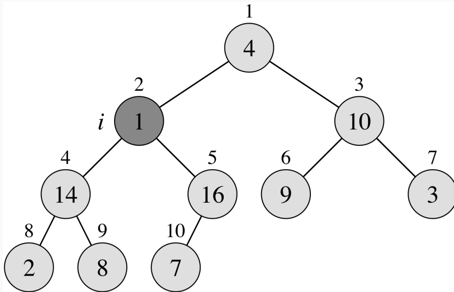


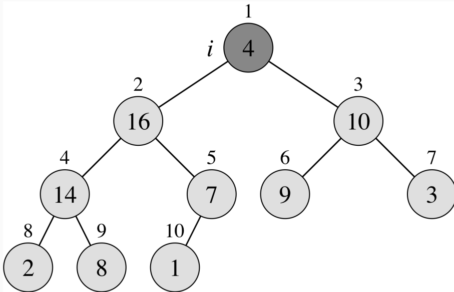
Dado um arranjo qualquer, que vamos transformar em um heap, quais elementos sabemos que são raízes de heap válidos? As folhas. Note que em um heap o número de folhas nunca é menor do que o número de nós internos.


1
4
2
1
3
3
4
2
5
i
16
6
9
7
10
8
14
9
8
10
7

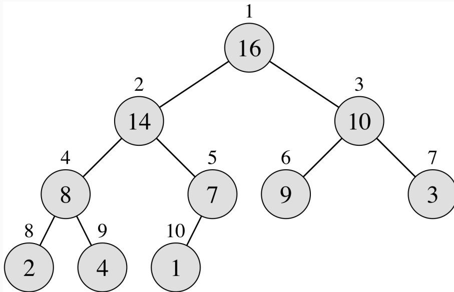

1
4
2
1
3
3
4
5
6
7
i
2
16
9
10
8
9
10
14
8
7


1
4
2
1
3
3
i
4
14
5
16
6
9
7
10
8
2
9
8
10
7


1
4
2
3
i
1
10
4
5
6
7
14
16
9
3
8
9
10
2
8
7


1
i
4
2
16
3
10
4
14
5
7
6
9
7
3
8
2
9
8
10
1

Construindo um heap



Projete uma função que receba com parâmetro um arranjo A, e rearranje os elementos de A para formar um heap máximo.

```
def inicializa_heap(A: list[int]):
    for i in reversed(range(len(A) // 2)):
        concerta_heap(A, len(A), i)
```

Qual é a complexidade de tempo?

- Limite simples: a função é `concerta_heap` tem tempo $O(\lg n)$ e é chamada $n/2$, portanto, $O(n \lg n)$;
- Limite estrito: $O(n)$ – discutido em sala.

Implementação ordenação por heap

Projete uma função que implemente a ordenação por heap.

```
def ordena_heap(lst: list[int]):
    inicializa_heap(lst)
    for n in reversed(range(1, len(lst))):
        # Troca o maior do heap com
        # o elemento da última posição do heap
        lst[0], lst[n] = lst[n], lst[0]
        # Concerta a raiz do heap
        concerta_heap(lst, n, 0)
```

Qual é a complexidade de tempo? $O(n \lg n)$.

A implementação é in-loco? Sim (se `concerta_heap` não for recursiva)

A implementação é estável? Não.

A ideia de um algoritmo divisão e conquista é:

- Resolver o problema diretamente se ele for trivial, senão **dividir** o problema em dois ou mais subproblemas do mesmo tipo;
- **Conquistar** os subproblemas resolvendo-os recursivamente;
- **Combinar** as soluções dos subproblemas para obter a solução do problema original

Como projetar um algoritmo de divisão e conquista para somar os elementos de um arranjo? (Note que esse algoritmo não traz nenhum vantagem, é apenas uma ilustração)

- Se o arranjo for vazio, a soma é 0. Senão dividir o arranjo na metade e calcular a soma de cada metade recursivamente;
- Obter a soma do arranjo somando o resultado de cada metade;

Ordenação incremental

Como projetar um algoritmo de divisão e conquista para ordenar os elementos de um arranjo?

- Se o arranjo tiver mais que um elemento, separamos os elementos em dois subarranjos;
- Ordenamos cada subarranjo recursivamente;
- Combinamos os dois subarranjos para obter a ordenação do arranjo inicial.

Temos que tomar duas decisões para tornar o processo concreto:

- Como separar os elementos em dois subarranjos?
- Como combinar os dois subarranjos ordenados?

Como separar os elementos em dois subarranjos?

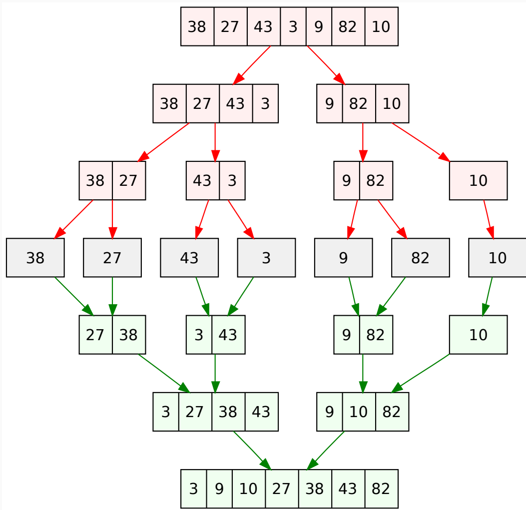
- Dividindo o arranjo ao meio;
- Qual é o custo? $O(1)$

Como combinar os dois subarranjos ordenados?

- Fazendo a intercalação em ordem dos elementos dos subarranjos;
- Qual é o custo? $O(n)$ – discutido a seguir

Este algoritmo é conhecido como **ordenação por intercalação** (*merge sort*).

Ordenação por intercalação



Projete uma função que implemente o algoritmo de ordenação por intercalação.

```
def ordena_intercalacao(lst: list[int]):
    # Se o problema não é trivial
    if len(lst) > 1:

        # Divide em dois subproblemas
        m = len(lst) // 2
        a = lst[:m]
        b = lst[m:]

        # Conquista recursivamente
        ordena_intercalacao(a)
        ordena_intercalacao(b)

        # Combina as soluções
        intercala(lst, a, b)
```

Projete uma função que implemente a intercalação.

```
def intercala(lst: list[int], a: list[int], b: list[int]):
    '''
    Faz a intercalação em ordem não decrescente dos
    elementos de *a* e *b* em *lst*.

    Requer que len(lst) = len(a) + len(b).
    Requer que a e b estejam em ordem não decrescente.

    Exemplos
    >>> lst = [0, 0, 0, 0, 0, 0, 0]
    >>> intercala(lst, [1, 6], [3, 5, 6, 8, 10])
    >>> lst
    [1, 3, 5, 6, 6, 8, 10]
    >>> intercala(lst, [3, 5, 6, 8, 10], [1, 6])
    >>> lst
    [1, 3, 5, 6, 6, 8, 10]
    '''
```

Ordenação por intercalação

Projete uma função que implemente o algoritmo de ordenação por intercalação.

```
def ordena_intercalacao(lst: list[int]):
    # Se o problema não é trivial
    if len(lst) > 1:

        # Divide em dois subproblemas
        m = len(lst) // 2
        a = lst[:m]
        b = lst[m:]

        # Conquista recursivamente
        ordena_intercalacao(a)
        ordena_intercalacao(b)

        # Combina as soluções
        intercala(lst, a, b)
```

```
def intercala(lst: list[int], a: list[int], b: list[int]):
    assert len(lst) == len(a) + len(b)
    i, j, k = 0, 0, 0
    while i < len(a) and j < len(b):
        if a[i] <= b[j]:
            lst[k] = a[i]
            i += 1
        else:
            lst[k] = b[j]
            j += 1
        k += 1
    while i < len(a): # Copia o restante de a
        lst[k] = a[i]
        i += 1
        k += 1
    while j < len(b): # Copia o restante de b
        lst[k] = b[j]
        j += 1
        k += 1
```

Projete uma função que implemente o algoritmo de ordenação por intercalação.

```
def ordena_intercalacao(lst: list[int]):
    # Se o problema não é trivial
    if len(lst) > 1:

        # Divide em dois subproblemas
        m = len(lst) // 2
        a = lst[:m]
        b = lst[m:]

        # Conquista recursivamente
        ordena_intercalacao(a)
        ordena_intercalacao(b)

        # Combina as soluções
        intercala(lst, a, b)
```

Qual é a complexidade de tempo de `intercala`? $O(n)$.

A implementação da ordenação por intercalação é in-loco? Não.

É estável? Sim.

Qual é a complexidade de tempo?

$$T(n) = \begin{cases} c & \text{se } n \leq 1 \\ 2T(n/2) + cn & \text{caso contrário} \end{cases}$$

$$T(n) = O(n \lg n)$$

Podemos fazer melhor? Vamos tentar!

Mas antes, vamos ver a forma mais comum de implementar a ordenação por intercalação, que é utilizando índices para informar o intervalo de ordenação / intercalação.

Note que nessa versão a cópia dos subarranjos geralmente é feita na função `intercala` e não em `ordena_intercala` como fizemos anteriormente.

Projete essa versão de `intercala`.

```python
def ordena_intercalacao(lst: list[int], ini: int, fim: int):
    '''
    Ordena o subarranjo lst[ini:fim] em ordem não
    decrescente.
    Requer que 0 <= i <= fim <= len(lst).
    '''

    # Se o problema não é trivial
    if ini < fim - 1:

        # Divide em dois subproblemas
        meio = (ini + fim) // 2

        # Conquista recursivamente
        ordena_intercalacao(lst, ini, meio)
        ordena_intercalacao(lst, meio, fim)

        # Combina as soluções
        intercala(lst, ini, meio, fim)
```

No projeto de um algoritmo de divisão e conquista temos que tomar duas decisões:

- Como separar os elementos em dois subarranjos?
- Como combinar os dois subarranjos ordenados?

Na ordenação por intercalação, a etapa de divisão tem tempo constante e a combinação tempo linear. Então, se queremos mudar o tempo de execução, precisamos pensar em como melhorar a combinação.

Como combinar dois arranjos ordenados sem precisar passar por todos os elementos? Parece que não tem com...

Se não podemos melhorar, será que podemos eliminar a etapa de combinação?

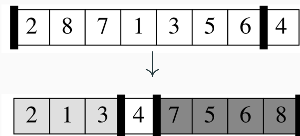
Supondo que o arranjo de entrada `lst[0:n]` seja dividido em `lst[0:p]` e `lst[p:n]`, o que é necessário para que após a ordenação de `lst[0:p]` e `lst[p:n]` o arranjo `lst[0:n]` fique ordenado sem precisarmos fazer mais nada?

Os elementos de `lst[0:p]` devem ser menores ou iguais aos elementos de `lst[p:n]`!

Então, o que precisamos fazer?

Projetar uma função que **particione** um arranjo em duas partes, uma com os “menores” e outra com os demais elementos (“maiores”).

Para isso precisamos de um “pivô” para determinar em que parte cada elemento deve ficar.



Faça a especificação da função que faz o particionamento de um arranjo. A função deve devolver o índice que separa as duas partes.

```
def particiona(lst: list[int], ini: int, fim: int) -> int:
    '''
    Reorganiza os elementos de lst[ini:fim] e devolve um
    índice p de maneira que os elementos de lst[ini:p]
    são menores ou iguais que lst[p:fim].

    Exemplos
    >>> lst = [2, 8, 7, 1, 3, 5, 6, 4]
    >>> particiona(lst, 0, len(lst))
    3
    >>> lst
    [2, 1, 3, 4, 8, 7, 5, 6]
    '''
```

O algoritmo de ordenação de divisão e conquista baseado na função de particionamento é chamado de **ordenação por particionamento** ou **quick sort**.

Implemente a ordenação por particionamento.

Ordenação por partição

```python
def ordena_particionamento(lst: list[int], ini: int, fim: int):
    '''
    Ordena o subarranjo lst[ini:fim] em ordem não
    decrescente.

    Requer que 0 <= i <= fim <= len(lst).
    '''

    # Se o problema não é trivial
    if ini < fim - 1:

        # Divide em dois subproblemas
        p = particiona(lst, ini, fim)

        # Conquista recursivamente
        ordena_particionamento(lst, ini, p)
        ordena_particionamento(lst, p, fim)

        # As soluções já estão combinadas!
```

```python
def ordena_intercalacao(lst: list[int], ini: int, fim:

    # Se o problema não é trivial
    if ini < fim - 1:

        # Divide em dois subproblemas
        meio = (ini + fim) // 2

        # Conquista recursivamente
        ordena_intercalacao(lst, ini, meio)
        ordena_intercalacao(lst, meio, fim)

        # Combina as soluções
        intercala(lst, ini, meio, fim)
```

```python
def ordena_particionamento(lst: list[int],
                           ini: int,
                           fim: int):

    # Se o problema não é trivial
    if ini < fim - 1:

        # Divide em dois subproblemas
        p = particiona(lst, ini, fim)

        # Conquista recursivamente
        ordena_particionamento(lst, ini, p)
        ordena_particionamento(lst, p, fim)

        # As soluções já estão combinadas!
```

Veremos como implementar `particiona` com tempo $O(n)$.

A ordenação por particionamento é in-loco? Se particiona é in-loco, sim.

É estável? Depende de particiona, mas o comum é que não seja.

Qual é a complexidade de tempo? Depende de como o arranjo é particionado.

O pior caso ocorre quando as partições são desbalanceadas (1 e $n - 1$), nesse caso a complexidade de tempo é $O(n^2)$.

Quando as partições são balanceadas a complexidade de tempo é $O(n \lg n)$.

Como podemos implementar a função `particiona`?

Uma forma simples é usar arranjos auxiliares para armazenar as duas partições enquanto elas são construídas.

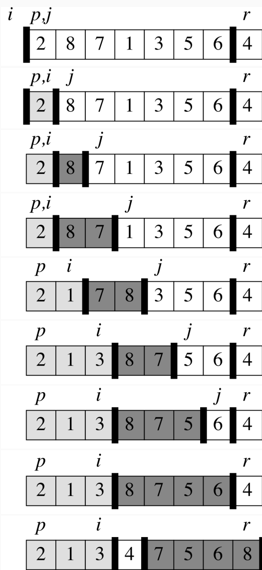
```python
def particiona(lst: list[int], ini: int, fim: int) -> int:
    pivo = lst[fim - 1]
    menores = []
    maiores_iguais = []
    for i in range(ini, fim - 1):
        if lst[i] < pivo:
            menores.append(lst[i])
        else:
            maiores_iguais.append(lst[i])
    # Copia os menores do que o pivo para lst
    for i in range(len(menores)):
        lst[i] = menores[i]
    # Copia o pivo para lst
    p = len(menores)
    lst[p] = pivo
    # Copias os maiores ou iguais ao pivo para lst
    for j in range(len(maiores_iguais)):
        lst[p + j + 1] = maiores_iguais[j]
    return p # Retorna o índice do pivo
```

As duas formas mais comum de fazer o particionamento in-loco são:

A forma sugerida por Tony Hoare, criador do quick sort, é manter dois índices, um para a partição do início do arranjo com os menores, e outro para a partição no final com os maiores. Os índices movem em direção ao meio e os elementos são trocados de lugar quando necessário.

A outra forma é o particionamento de Lomuto. Nesse esquema tanta a partição dos menores fica no início do arranjo e a dos maior logo em seguida.

Particionamento de Lomuto




```python
def particiona(lst: list[int], ini: int, fim: int) -> int:
    pivo = lst[fim - 1]
    i = ini - 1
    for j in range(ini, fim - 1):
        if lst[j] <= pivo:
            i += 1
            lst[i], lst[j] = lst[j], lst[i]
    lst[i + 1], lst[fim - 1] = lst[fim - 1], lst[i + 1]
    return i + 1
```

Algoritmo	Estável?	Local?	Melhor	Médio	Pior
Inserção	Sim	Sim	$O(n)$	$O(n^2)$	$O(n^2)$
Seleção	Não	Sim	$O(n^2)$	$O(n^2)$	$O(n^2)$
Heap	Não	Sim	$O(n \lg n)$	$O(n \lg n)$	$O(n \lg n)$
Intercalação	Sim	Não	$O(n \lg n)$	$O(n \lg n)$	$O(n \lg n)$
Particionamento	Não	Sim	$O(n \lg n)$	$O(n \lg n)$	$O(n^2)$

Parece que mesmo usando diversas técnicas não conseguimos um algoritmo melhor que $O(n \lg n)$...

Os algoritmos de ordenação que vimos até agora são baseados em comparações, isto é, a ordem dos elementos é determinada usando apenas comparações entre os elementos.

Um resultado conhecido diz que não existe algoritmo de ordenação baseado em comparação que tenha tempo melhor que $O(n \lg n)$, então, já temos algoritmos ótimos.

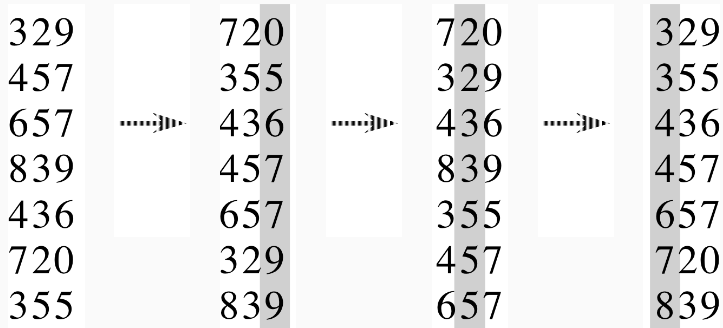
Mas ainda podemos melhorar fazendo ordenação sem comparação!

Cada um dos n valores de entrada é uma sequência de tamanho d, onde cada valor da sequência pode ser um de k valores distintos. Exemplos:

- Nomes com 50 caracteres, onde cada caractere pode $a, b, \ldots, z$ ($d = 50$, $k = 26$)
- Números com 8 dígitos, onde cada dígito pode ser $0, 1, \ldots, 9$ ($d = 8$, $k = 10$)

Podemos usar a restrição dos valores de entrada para projetar um algoritmo de ordenação mais eficiente? Sim!

A ideia é ordenar os valores pelos dígitos, começando com o menos significativos.



Se d é constante e $k = O(n)$, então é possível implementar a ordenação por dígito com complexidade de tempo de $O(n)$.

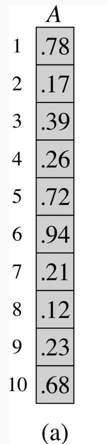
Ordenação por balde

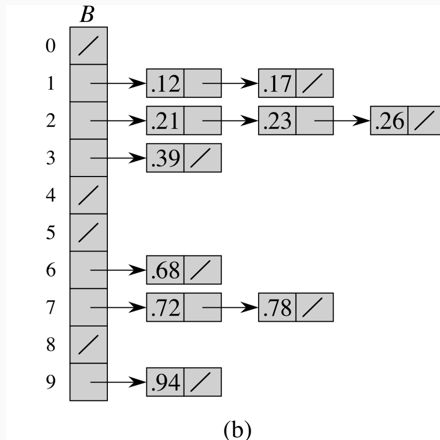
Os n valores de entrada estão distribuídos uniformemente no intervalo $[0, 1)$.

Podemos usar a restrição dos valores de entrada para projetar um algoritmo de ordenação mais eficiente? Sim!

A ideia é:

- Dividir o intervalo $[0, 1)$ em n segmentos (baldes) e distribuir cada um dos n elementos em seu respectivo segmento;
- Ordenar os elementos de cada segmento;
- Juntos os elementos de cada segmento





Uma implementação direta da ordenação por balde tem tempo de execução no pior cado de $O(n^2)$, mas o tempo esperado é de $O(n)$.

Thomas H. Cormen et al. Introduction to Algorithms. 3rd edition. Capítulos 6, 7 e 8.