Algoritmos de Classificação de Tabelas - Sorting

A classificação de tabelas de dados é uma operação bastante usada em qualquer área da computação. Os algoritmos de classificação estão dentre os algoritmos fundamentais. Operações em bancos de dados estruturados sempre precisam que os dados estejam numa determinada ordem para facilitar a sua recuperação.

Existem várias formas de se classificar uma tabela. Veremos algumas delas. Vamos nos restringir a classificação interna, isto é, tabelas que estão na memória. A classificação externa, de arquivos, segue a mesma linha, com algumas adaptações para minimizar a quantidade de operações de entrada e saída.

Trocar os valores de 2 elementos de uma lista

Uma tabela é em última análise uma lista em Python. Nos algoritmos de classificação ocorre com frequência a troca de valores de dois elementos da lista.

Trocar x com y, ou a[i] com a[j], resume-se no comando:

```
x, y = y, x
```

ou

```
a[i], a[j] = a[j], a[i]
```

1. Classificação – método da seleção

O algoritmo imediato para se ordenar uma tabela com `n` elementos é o seguinte:

1. Determinar o mínimo a partir do primeiro e trocar com o primeiro
2. Determinar o mínimo a partir do segundo e trocar com o segundo
3. Determinar o mínimo a partir do terceiro e trocar com o terceiro

. . . .
. . . .
n-1. Determinar o mínimo a partir do (n-1)-ésimo e trocar com o (n-1)-ésimo

Exemplo:

```
6    2    2    2
8    8    4    4
4    4    8    5
2    6    6    6
5    5    5    8
```

Abaixo a função Selecao que recebe um lista e a classifica em ordem crescente :

```
def Selecao(a):
    n = len(a)
    # i = 0, 1, 2, ..., n - 2
    for i in range(n - 1):
        # determina o índice do menor elemento a partir de i
        imin = i
        for j in range(i + 1, n):
            if (a[imin] > a[j]): imin = j
        # troca a posição do menor com a posição de i-ésimo
        a[i], a[imin] = a[imin], a[i]
```

O programa abaixo exemplifica o uso da função Selecao. Gera uma tabela com números randômicos e classifica essa tabela.

```
from random import randrange

def Selecao(a):
    n = len(a)
    # i = 0, 1, 2, ..., n - 2
    for i in range(n - 1):
        # determina o índice do menor elemento a partir de i
        imin = i
        for j in range(i + 1, n):
            if a[imin] > a[j]: imin = j
        # troca a posição do menor com a posição de i-ésimo
        a[i], a[imin] = a[imin], a[i]

def geratab(n):
    # gera tabela com n numeros randômicos entre 0 999
    tab = []
    for i in range(n):
        tab.append(randrange(1000))
    return tab

# gera tabela com N números
N = 100
tabela = geratab(N)
print("\n\ntabela original:\n", tabela)

# classifica pelo método da seleção
Selecao(tabela)
print("\n\ntabela classificada:\n", tabela)
```

Veja a saída:

```
tabela original:
 [935, 815, 134, 931, 923, 797, 342, 398, 838, 248, 204, 2, 352, 669,
910, 398, 615, 734, 541, 565, 499, 386, 981, 282, 868, 121, 648, 612,
382, 241, 292, 379, 827, 221, 639, 250, 438, 891, 743, 924, 422, 986,
509, 409, 887, 845, 596, 692, 365, 988, 858, 591, 77, 333, 961, 936,
538, 185, 189, 57, 419, 967, 871, 589, 756, 140, 934, 359, 922, 721,
326, 715, 695, 261, 489, 600, 646, 791, 890, 791, 697, 153, 536, 441,
909, 521, 104, 417, 703, 254, 693, 17, 406, 749, 937, 72, 43, 20,
723, 308]


tabela classificada:
 [2, 17, 20, 43, 57, 72, 77, 104, 121, 134, 140, 153, 185, 189, 204,
221, 241, 248, 250, 254, 261, 282, 292, 308, 326, 333, 342, 352, 359,
365, 379, 382, 386, 398, 398, 406, 409, 417, 419, 422, 438, 441, 489,
499, 509, 521, 536, 538, 541, 565, 589, 591, 596, 600, 612, 615, 639,
646, 648, 669, 692, 693, 695, 697, 703, 715, 721, 723, 734, 743, 749,
756, 791, 791, 797, 815, 827, 838, 845, 858, 868, 871, 887, 890, 891,
909, 910, 922, 923, 924, 931, 934, 935, 936, 937, 961, 967, 981, 986,
988]
```

1.1 Classificação – método da seleção – análise

Número de trocas: é sempre n–1. Não seria necessário trocar se o mínimo fosse o próprio elemento.

Número de comparações: é sempre (n-1)+(n-2)+...+2+1 = n(n-1)/2

O número de comparações representa a quantidade de vezes que o laço principal do algoritmo é repetido. Assim, o tempo deste algoritmo é proporcional a n^2, ou seja, o método da seleção é $O(n^2)$.

2. Classificação – Método Bubble (da Bolha)

Outro método para fazer a classificação é pegar cada um dos elementos a partir do segundo (a[1] até a[n-1]) e subi-lo até que encontre o seu lugar.

```
6    6    4    4    4    2    2    2
8    4    6    6    2    4    4    4
4    8    8    2    6    6    6    5
2    2    2    8    8    8    5    6
5    5    5    5    5    5    8    8
```

Observe como cada elemento sobe até encontrar o seu lugar. Como uma bolha de ar num recipiente de água. Só que para quando encontrar o seu lugar, ou seja, encontrou um elemento menor ou igual ou chegou ao início da tabela.

```
def Bolha(a):
    n = len(a)
    # i = 1, 2, ..., n - 1
    for i in range(1, n):
        # sobe com a[i] até encontrar o lugar adequado
        j = i
        while j > 0 and a[j] < a[j - 1]:
            # troca com o seu vizinho
            a[j], a[j - 1] = a[j - 1], a[j]
            # continua subindo
            j = j - 1
```

O programa abaixo exemplifica o uso da função Bolha. Gera uma tabela com números randômicos e classifica essa tabela.

```
from random import randrange

def Bolha(a):
    n = len(a)
    # i = 1, 2, ..., n - 1
    for i in range(1, n):
        # sobe com a[i] até encontrar o lugar adequado
        j = i
        while j > 0 and a[j] < a[j - 1]:
            # troca com o seu vizinho
            a[j], a[j - 1] = a[j - 1], a[j]
            # continua subindo
            j = j - 1

def geratab(n):
    # gera tabela com n numeros randômicos entre 0 999
    tab = []
    for i in range(n):
        tab.append(randrange(1000))
    return tab

# gera tabela com N números
N = 100
tabela = geratab(N)
print("\n\ntabela original:\n", tabela)
```

```
# classifica pelo método da seleção
Bolha(tabela)
print("\n\ntabela classificada:\n", tabela)
```

Veja a saída:

```
tabela original:
 [277, 99, 95, 476, 315, 887, 699, 522, 694, 40, 656, 794, 211, 33,
426, 6, 136, 22, 750, 715, 85, 95, 17, 325, 207, 278, 194, 455, 34,
861, 910, 200, 22, 769, 416, 648, 148, 914, 544, 251, 380, 750, 617,
443, 163, 957, 839, 221, 745, 227, 801, 265, 959, 253, 731, 809, 742,
969, 470, 917, 791, 750, 356, 111, 495, 119, 857, 621, 639, 323, 163,
766, 657, 722, 595, 732, 14, 672, 812, 535, 539, 555, 499, 133, 500,
236, 555, 980, 328, 224, 441, 618, 750, 694, 145, 627, 939, 795, 28,
710]


tabela classificada:
 [6, 14, 17, 22, 22, 28, 33, 34, 40, 85, 95, 95, 99, 111, 119, 133,
136, 145, 148, 163, 163, 194, 200, 207, 211, 221, 224, 227, 236, 251,
253, 265, 277, 278, 315, 323, 325, 328, 356, 380, 416, 426, 441, 443,
455, 470, 476, 495, 499, 500, 522, 535, 539, 544, 555, 555, 595, 617,
618, 621, 627, 639, 648, 656, 657, 672, 694, 694, 699, 710, 715, 722,
731, 732, 742, 745, 750, 750, 750, 750, 766, 769, 791, 794, 795, 801,
809, 812, 839, 857, 861, 887, 910, 914, 917, 939, 957, 959, 969, 980]
```

Exercícios

Considere as 120 (5!) permutações de 1 2 3 4 5:

1) **Encontre algumas que precisem exatamente de 5 trocas para classificá-la pelo método da bolha.**
2) **Idem para 7 trocas**
3) **Qual a sequência que possui o número máximo de trocas e quantas trocas são necessárias?**

2.1 Classificação - método bubble (da bolha) – análise

O comportamento do método da bolha depende se a tabela está mais ou menos ordenada, mas em média, o seu tempo também é proporcional a n^2 como no método anterior.

Quantas vezes o comando de troca (`a[j], a[j - 1] = a[j - 1], a[j]`) é executado?

O pior caso ocorre quando a sequência está invertida e os elementos terão que subir sempre até a primeira posição.

Nesse caso, a troca é executada i vezes para cada valor de i = 1, 2, 3, ..., n-1. Portanto 1 + 2 + 3 + ... + n – 1 = n.(n – 1)/2.

Portanto, no pior caso o método da bolha é proporcional a n^2. O método é então $O(n^2)$.

E quanto ao número médio? Será que considerarmos o número médio teremos um limitante menor que $O(n^2)$?

2.2 Inversões

Seja `P = a1 a2 ... an`, uma permutação de `1 2 ... n`.
O par `(i,j)` é uma inversão quando `i < j` e `ai > aj`.
Exemplo: `1 3 5 4 2` tem `4` inversões: `(3,2) (5,4) (5,2) e (4,2)`

No método Bubble o número de trocas é igual ao número de inversões da sequência.
O algoritmo se resume a:

```
for (i = 1; i < n; i++) {
   /* elimine as inversões de a[i] até a[0] */
   . . .
}
```

Veja também o exemplo:

```
6 8 4 2 5 – elimine as inversões do 8 – 0
6 8 4 2 5 – elimine as inversões do 4 – 2
4 6 8 2 5 – elimine as inversões do 2 – 3
2 4 6 8 5 – elimine as inversões do 5 – 2
2 4 5 6 8
```

Total de 7 inversões que é exatamente a quantidade de trocas na sequência.

Para calcularmos então o número de trocas do bubble, basta calcular o número de inversões da sequência.

É equivalente a calcular o número de inversões de uma permutação de `1 2 ... n`.

Qual o número médio de inversões em todas as sequências possíveis?

É equivalente a calcular o número médio de inversões em todas as permutações de 1 2 ... n.

Seja P_n o conjunto das permutações 1 2 3 ... n. P_n tem n! elementos.

Seja $p \in P_{n-1}$ (p é permutação de 1 2 3 ... n-1). Pensando recursivamente, p gera n outras permutações de n elementos, pela introdução de n em todas as n posições possíveis.

Seja I(p) a quantidade de inversões em p. Se I(p) = k, então a n permutações geradas por p terão a seguinte quantidade de inversões:

$(k+0) + (k+1) + (k+2) + ... + (k+n-1) = n.k + n.(n-1)/2$

Seja J_n o total de inversões em todas as permutações possíveis de n elementos.

$J_n = \sum I(p)\ (p \in P_n)$

$J_n = \sum n.I(p)+n.(n-1)/2\ (p \in P_{n-1}) = n\ .\ \sum I(p)+ (n-1)/2\ (p \in P_{n-1})$

Como temos (n-1)! permutações em P_{n-1}

$J_n = n\ .\ (I(p_1) + I(p_2) + ... I(p_{n-1!}) + (n-1)!\ .\ (n-1)/2)$

$J_n = n!\ .\ (n-1)/2 + n\ .\ J_{n-1}$

$J_n = n!\ .\ (n-1)/2 + n\ .\ ((n-1)!\ .\ (n-2)/2 + (n-1)\ .\ J_{n-2})$

$J_n = n!\ .\ ((n-1)/2 + (n-2)/2) + n\ .\ (n-1)\ .\ J_{n-2})$

...

$J_n = n!\ .\ ((n-1)/2 + (n-2)/2 + ... + 1) + n\ .\ (n-1)\ .\ (n-2)\ ...\ 1\ .\ J_0$

$J_n = n!\ .\ (n\ .\ (n-1)/4) + n!\ .\ J_0$

J_0 e J_1 são zeros.

Portanto, $J_n = n!\ .\ (n\ .\ (n-1)/4)$

Estamos interessados no número médio de inversões. Como há n! permutações em P_n:

Média = J_n / n! = (n . (n-1) / 4)

Portanto o número médio de trocas no algoritmo da Bolha é n.(n-1)/4.
Note que é exatamente a média entre o mínimo e o máximo, o que não surpreende. Intuitivamente, a quantidade de permutações com muitas inversões é igual à quantidade com poucas inversões.

Assim, o algoritmo da Bolha é definitivamente $O(n^2)$.

3. Classificação – método da inserção

O método da inserção é uma variação do bubble.

A idéia é pegar cada um dos elementos a partir do segundo e abrir um espaço para ele (inserir), deslocando de uma posição para baixo, todos os maiores que estão acima. Observe que é até um pouco melhor que trocar com o vizinho de cima.

```
def insercao(a):
    n = len(a)
    # todos a partir do segundo elemento
    for i in range(1, n):
        x = a[i] # guarda a[i]
        # desloca todos os necessários para
        # liberar um lugar para a[i]
        j = i - 1
        while j >= 0 and a[j] > x:
            a[j + 1] = a[j]
            j = j - 1
        # a posição j + 1 ficou livre para receber a[i]
        a[j + 1] = x
```

A análise é a mesma que o bubble.

4. Classificação - método Shell

Este método é um pouco melhor que o bubble.

A idéia é tentar eliminar inversões de maneira mais rápida.
Repete-se então o bubble com uma sequência de passos: n/2, n/4, ... 1.
Quanto o passo é 1, temos o próprio bubble. Porém, neste momento a sequência já está com menos inversões. Portanto teremos menos trocas.

Veja um exemplo numa sequência de 5 elementos. Vamos executar o bubble com passos 2 e 1:

```
h=2          h=1
6   4   4    4   2   2
8   8   2    2   4   4
4   6   6    5   5   5
2   2   8    8   8   6
5   5   5    6   6   8
```

Observe que tivemos 5 trocas. No caso do bubble para esta mesma sequência tivemos 7.

A implementação abaixo é parecida com o bubble. Tem um loop a mais para gerar o passo h que vale: n / 2, n / 4, n / 8, ..., 1. Observe que se substituir o valor de h por 1, o algoritmo é exatamente o bubble.

```
def shell(a):
    h = len(a) // 2
    # repita até que h fique 1
```

```
    while h > 0:
        # subir a[i], i = h, h + 1, ..., n -1
        for i in range(h, len(a)):
            # subir a[i] até encontrar menor ou chegar em a[0]
            j = i
            while j >= h and a[j] < a[j - h]:
                # troca a[j] e a[j - h]
                a[j], a[j - h] = a[j - h], a[j]
                j = j - h # continua subindo
        # muda o passo
        h = h // 2
```

Outra variação parecida com o método da inserção. Observe que se substituir o valor de h por 1, o algoritmo é exatamente o da inserção:

```
def shell(a):
    h = len(a) // 2
    # repita até que h fique 1
    while h > 0:
        # deslocar a partir de a[i], i = h, h+1, ..., n-1
        for i in range(h, len(a)):
            # abrir espaço para a[i]
            j = i
            v = a[i]
            while j >= h and v < a[j - h]:
                # desloca a[j - h]
                a[j] = a[j - h]
                j = j - h # continua subindo
            # colocar v no seu lugar
            a[j] = v
        # muda o passo
        h = h // 2
```

4.1 Classificação - método shell – análise

A análise do método shell depende do passo h que é usado.
É uma análise bastante complexa e não existe uma análise geral para ao mesmo.
Existem alguns resultados parciais que sugerem que o shell é melhor que $O(n^2)$.
Alguns resultados parciais:

1) O método shell faz menos que $O(N^{3/2})$ quando se usa a sequência 1, 4, 13, 40, 121, ..., ou seja, a sequência $h_i = 3.h_{i-1}+1$. Esse resultado é de Knuth-1969.

2) O método shell faz menos que $O(N^{4/3})$ quando se usa a sequência 1, 8, 23, 77, 281, ..., ou seja, a sequência $4^{i+1} + 3.2^i + 1$, para $i=0,1,2, ...$

3) A sequência 1, 2 , 4, 8, ..., proposta por Shell-1959 não é a melhor sequência. Em alguns casos pode levar ao tempo $O(N^2)$. Com esta sequência de passos, os elementos de posições pares não são comparados com elementos de posições ímpares. Considere o caso em que temos a metade dos elementos menores nas posições ímpares e a metade dos elementos maiores nas posições pares. Exemplo – sequência de 16 elementos:

```
1 9 2 10 3 11 4 12 5 13 6 14 7 15 8 16
```

Veja o que acontece com os passos 8, 4 e 2. Nada é feito. Tudo fica para o passo 1. (bubble).

A versão abaixo usa a sequência sugerida por Knuth (primeiro caso).

```
def shell(a):
    # determina o valor inicial de h
    n = len(a)
    h = 1
    while h <= n // 3: h = 3 * h + 1

    # repita até que h fique 1
    while h > 0:
        # deslocar a partir de a[i], i = h, h+1, ..., n-1
        for i in range(h, len(a)):
            # abrir espaço para a[i]
            j = i
            v = a[i]
            while j >= h and v < a[j - h]:
                # desloca a[j - h]
                a[j] = a[j - h]
                j = j - h # continua subindo
            # colocar v no seu lugar
            a[j] = v
        # muda o passo
        h = h // 3
```

5. Classificação - método Merge

Considere o seguinte problema. Dados dois vetores **a** e **b** de **n** e **m** elementos já ordenados, construir outro vetor **c** de **m+n** elementos também ordenado com os elementos de **a** e **b**.

Uma primeira solução seria colocar em **c** os elementos de **a**, seguidos dos elementos de **b** e ordenar o vetor **c**. Existe forma melhor, que é fazer a intercalação (merge) dos elementos de **a** e **b** em **c**. Vejamos:

```
a = 2 5 8 9
b = 1 3 4
```

c = 1 2 3 4 5 8 9

Basta pegar o menor entre **a[i]** e **b[j]** e colocar em **c[k].** A cada passo incrementar **i** ou **j** e **k**.

```
def intercala(a, b):
    # Intercala duas listas já ordenadas.
    n = len(a)
    m = len(b)
    # o tamanho de c é (n + m)
    c = (n + m) * [None]
    # i percorre a, j percorre b e k percorre c
    i, j, k = 0, 0, 0
    # move para c, elemento de a ou de b
    while k < n + m:
        if i == n: # terminou a - então move b[j]
            c[k] = b[j]; j += 1
        elif j == m: # terminou b - então move a[i]
            c[k] = a[i]; i += 1
        elif a[i] < b[j]: # move o menor entre a[i] e b[j]
            c[k] = a[i]; i += 1
        else:
            c[k] = b[j]; j += 1
        # avança k
        k += 1
    return c
```

Outra forma:

```
def intercala(a, b):
    ''' Intercala duas listas já ordenadas.'''
    n = len(a)
    m = len(b)
    # o tamanho de c é (n + m)
    c = (n + m) * [None]
    # i percorre a, j percorre b e k percorre c
    i, j, k = 0, 0, 0
    # move para c, elemento de a ou de b
    while i < n and j < m:
        if a[i] < b[j]: # move a[i]
            c[k] = a[i]; i += 1
        else:           # move b[j]
            c[k] = b[j]; j += 1
        k += 1
    # Neste ponto, ou chegou ao final de a ou de b
    # Basta mover o que restou de a e o que restou de b
```

```
    # De um deles restou 0 elementos
    while i < n:
        c[k], k, i = a[i], k + 1, i + 1
    while i < n:
        c[k], k, j = b[j], k + 1, j + 1

    return c
```

Apenas para evitar que a cada chamada da função intercala haja a criação de uma nova lista para receber a lista intercalada, vamos modificar um pouco, passando também a lista destino como parâmetro.

```
def intercala(a, b, c):
    # supor len(c) >= len(a) + len(b)
    n, m = len(a), len(b)
    i, j, k = 0, 0, 0
    # Coloque em c[k] o menor entre a[i] e b[j]
    while i < n and j < m:
        if a[i] < b[j]:
            c[k] = a[i]
            i = i + 1
        else:
            c[k] = b[j]
            j = j + 1
        # avança k
        k = k + 1
    # Neste ponto, esgotou a lista a ou a lista b
    # Basta então mover os itens restantes de a ou b
    # De uma delas nada será movido
    while i < n:
        c[k] = a[i]
        i, k = i + 1, k + 1
    while j < m:
        c[k] = b[j]
        j, k = j + 1, k + 1
```

Importante notar que o algoritmo acima sempre precisa de memória auxiliar para intercalar. Mesmo que os elementos das duas listas sejam estejam em trechos contíguos de uma mesma lista, será necessário usar uma lista auxiliar para conter a lista intercalada.

A função **`intercala`** acima em qualquer dos casos tem complexidade linear.
O(n+m) é o mesmo que O(2.n) que é o mesmo que O(n).

Com a função intercala, podemos construir um algoritmo interessante de classificação. O algoritmo é recursivo e divide a sequência em duas metades, estas em outras duas metades, etc., até que cada

sequência tenha 0 ou 1 elementos, caso em que nada é feito. Após isso, intercala as sequências vizinhas.

A função merge_sort abaixo classifica uma lista da maneira proposta acima.

```
def merge_sort(lista):
        if len(lista) > 1:
             meio = len(lista) // 2
             lista_esquerda = lista[:meio]
             lista_direita = lista[meio:]
             # classifica as listas esquerda e direita
             merge_sort(lista_esquerda)
             merge_sort(lista_direita)
             # intercala as listas esquerda e direita
             intercala(lista_esquerda, lista_direita, lista)
```

Considere o teste abaixo, onde usamos as funções idênticas apenas acrescentando alguns print() para mostrar o funcionamento.

```
def merge_sort(lista):
    print("Classificando:", lista)
    if len(lista) > 1:
        meio = len(lista) // 2
        lista_esquerda = lista[:meio]
        lista_direita = lista[meio:]
        # classifica as listas esquerda e direita
        merge_sort(lista_esquerda)
        merge_sort(lista_direita)
        # intercala as listas esquerda e direita
        intercala(lista_esquerda, lista_direita, lista)


def intercala(a, b, c):
    print("intercalando:", a, " e ", b, " em ", c)
    # supor len(c) >= len(a) + len(b)
    n, m = len(a), len(b)
    i, j, k = 0, 0, 0
    # Coloque em c[k] o menor entre a[i] e b[j]
    while i < n and j < m:
        if a[i] < b[j]:
            c[k] = a[i]
            i = i + 1
        else:
            c[k] = b[j]
            j = j + 1
```

```
            # avança k
            k = k + 1
        # Neste ponto, esgotou a lista a ou a lista b
        # Basta então mover os itens restantes de a ou b
        # De uma delas nada será movido
        while i < n:
            c[k] = a[i]
            i, k = i + 1, k + 1
        while j < m:
            c[k] = b[j]
            j, k = j + 1, k + 1
        print("resultado:", c)

# teste
minha_lista = [72, 32, 12, 45, 38, 23, 86, 17]
merge_sort(minha_lista)
print(minha_lista)
```

Seria impresso:

```
Classificando: [72, 32, 12, 45, 38, 23, 86, 17]
Classificando: [72, 32, 12, 45]
Classificando: [72, 32]
Classificando: [72]
Classificando: [32]
intercalando: [72]  e  [32]  em  [72, 32]
resultado: [32, 72]
Classificando: [12, 45]
Classificando: [12]
Classificando: [45]
intercalando: [12]  e  [45]  em  [12, 45]
resultado: [12, 45]
intercalando: [32, 72]  e  [12, 45]  em  [72, 32, 12, 45]
resultado: [12, 32, 45, 72]
Classificando: [38, 23, 86, 17]
Classificando: [38, 23]
Classificando: [38]
Classificando: [23]
intercalando: [38]  e  [23]  em  [38, 23]
resultado: [23, 38]
Classificando: [86, 17]
Classificando: [86]
Classificando: [17]
intercalando: [86]  e  [17]  em  [86, 17]
resultado: [17, 86]
```

```
intercalando: [23, 38]  e  [17, 86]  em  [38, 23, 86, 17]
resultado: [17, 23, 38, 86]
intercalando: [12, 32, 45, 72]  e  [17, 23, 38, 86]  em  [72, 32, 12,
45, 38, 23, 86, 17]
resultado: [12, 17, 23, 32, 38, 45, 72, 86]
[12, 17, 23, 32, 38, 45, 72, 86]
```

5.1 Classificação - método Merge – análise simplificada

No caso da função **intercala**, não tem comparações ou trocas para basearmos a contagem do tempo. Já vimos que o tempo necessário para fazer a intercalação de duas sequências com um total de **n** elementos é proporcional a **n**. Digamos **c.n**, onde **c** é uma constante.

O tempo necessário para fazer o mergesort de **n** elementos **T(n)**, é o tempo necessário para fazer o merge de **2** sequências de **n/2** elementos mais o tempo necessário para se fazer a intercalação de **n** elementos. Ou seja

$$T(n) = 2.T(n/2) + c.n$$
$$T(n) = 2.(2.T(n/4)+c.n/2)+c.n = 4.T(n/4) + 2.c.n$$
$$T(n) = 4.(2.T(n/8)+c.n/4)+2.c.n = 8.T(n/8)+3.c.n$$
$$....$$
$$T(n) = 2^k.T(n/2^k)+ k.c.n$$

Supondo $n = 2^k$, e portanto $k = lg(n)$ (base 2). Se **n** não for potência de **2**, considere como limitante superior o menor **n'** maior que **n** que seja desta forma.

$$T(n) = n.T(1) + lg(n).c.n = n.(1 + c.lg(n))$$

Assim, **T(n)** é proporcional a **n.lg(n)** ou **O(n.log(n))**.

É um algoritmo melhor que os primeiros (seleção, bolha e inserção) que eram O(n^2).

Sempre é bom lembrar que esse método necessita de memória auxiliar de tamanho igual a lista original.

6. Classificação - método Quick

Existem algoritmos que particionam uma sequência em duas, determinando um elemento pivô, tal que todos à esquerda são menores e todos à direita são maiores. Com isso usa-se a mesma técnica do Merge, isto é, aplica-se o algoritmo recursivamente na parte esquerda e na parte direita.

6.1 O algoritmo para particionar a sequência

O algoritmo abaixo trabalha com dois apontadores para "pivotar" a sequência. O apontador **`i`** começa com o primeiro elemento enquanto que o apontador **`j`** começa com o último. Apenas um deles é modificado a cada comparação (**`i += 1`** ou **`j -= 1`**), ou seja, há dois sentidos, incrementando **`i`** ou decrementando **`j`**. Quando a direção é no sentido de incrementar **`i`** e **`A[i] > A[j]`** trocam-se os elementos, fixa-se o **`i`** e inverte-se a direção. Quando a direção é no sentido de decrementar **`j`** e **`A[i] > A[j]`** trocam-se os elementos, fixa-se o **`j`** e inverte-se a direção. O processo continua até que **`i`** fique igual a **`j`**. Quando isso acontece, temos a seguinte situação:

`A[k] < A[i]` para **`k=0,1,...,i-1`**
`A[k] >= A[i]` para **`k=i+1,...,n-1`**
`A[i]` já está em seu lugar definitivo.

Veja a seguinte sequência de passos:

```
i             j
→             ←
5 3 9 7 2 8 6 (Avança i)
5 3 6 7 2 8 9 (6 é o maior da esquerda e todos menores que 9. Avança j)
5 3 2 7 6 8 9 (6 é o menor da direita e todos maiores. Avança i)
5 3 2 6 7 8 9 (6 é o maior da esquerda e todos menores. Avança j)
              (i ficou igual a j. 6 já está em seu lugar; todos da esquerda
               são menores; todos da direita são maiores; os da direita já estão em
               ordem neste exemplo mas é coincidência)
              (Vamos repetir o mesmo para as partes esquerda e direita)
5 3 2         (Avança i)
2 3 5         (Avança j. 2 está no lugar)
  3 5         (Avança i. 5 já está no lugar)
        7 8 9 (Avança i. 9 já está no lugar)
        7 8   (Avança i. 8 já está no lugar)
2 3 5 6 7 8 9 (tudo no lugar)

Outro exemplo:

6 3 9 2 5 4 1 8 7 (avança i)
6 3 7 2 5 4 1 8 9 (todos a esquerda de i menores que j;avança j)
6 3 1 2 5 4 7 8 9 (todos a direita de j maiores que todos menores que i; avança i)
                  (i=j no 7; todos a esquerda menores e todos a direita maiores)

6 3 1 2 5 4       (avança i)
4 3 1 2 5 6       (avança j)
2 3 1 4 5 6       (avança i - i fica igual a j no 4)

2 3 1             (avança i)
1 3 2             (avança j - i fica igual a j no 1)

  3 2             (avança i)
  2 3             (avança j - i fica igual a j no 2)

            8 9   (avança i - i fica igual a j no 9)
```

A sequência ficou classificada.

Vejamos primeiro a função particiona da maneira proposta acima:

```
def particiona(lista, inicio, fim):
    # Particiona a lista de lista{inicio] até lista[fim]
    i, j = inicio, fim
    # Direção - dir == 1 esquerda-direita e dir == -1 ao contrário
    dir = 1
    while i < j:
        if lista[i] > lista[j]:
            lista[i], lista[j] = lista[j], lista[i]
            # muda a direção
            dir = - dir
        # incrementa i ou decrementa j
        if dir == 1: i = i + 1
        else: j = j - 1
    # Devolve o índice do elemento pivô
    return i
```

6.2 Outra forma de particionar

Um variação, talvez mais intuitiva do algoritmo acima é pensar no último elemento da sequência como o pivô. Este pivô fica alterando de posição, deslocando os menores para a sua esquerda e os maiores para a sua direita. No final ele estará na posição definitiva.

```
def particiona(lista, inicio, fim):
    i, j = inicio, fim
    pivo = lista[fim]
    while True:
        # aumentado i
        while i < j and lista[i] <= pivo: i = i + 1
        if i < j:
            lista[i], lista[j] = pivo, lista[i]
        else: break
        # diminuindo j
        while i < j and lista[j] >= pivo: j = j - 1
        if i < j: lista[i], lista[j] = lista[j], pivo
        else: break
    return i
```

6.3 O método Quick - recursivo

Agora o Quick recursivo que fica parecido com o Merge.

Vamos também usar com parâmetro o inicio e o fim da lista. Note que na solução abaixo, não vamos criar novas listas auxiliares. Todo o trabalho será feito sobre trechos internos da lista original. Por isso, vamos usar sempre os índices de início e fim dentro da lista.
A primeira chamada é Quick_Sort **(lista, 0, n-1)**.

```
def Quick_Sort(lista, inicio, fim):
    # Se a lista tem mais de um elemento, ela será
    # particionada e as duas partições serão classificadas
    # pelo mesmo método Quick Sort
    if inicio < fim:
        k = particiona(lista, inicio, fim)
        print("pivo:", lista[k])
        Quick_Sort(lista, inicio, k - 1)
        Quick_Sort(lista, k + 1, fim)
```

6.4 Testes – Quick recursivo

Veja agora o teste abaixo. Usamos exatamente as mesmas funções e só adicionamos alguns print() para mostrar o funcionamento.

```
def Quick_Sort(lista, inicio, fim):
    # Se a lista tem mais de um elemento, ela será
    # particionada e as duas partições serão classificadas
    # pelo mesmo método Quick Sort
    print("\nLista Total:", lista)
    print("Classificando Trecho:", lista[inicio:fim + 1])
    if inicio < fim:
        k = particiona(lista, inicio, fim)
        print("pivo:", lista[k])
        print("Trecho Particionado:", lista[inicio:fim + 1])
        Quick_Sort(lista, inicio, k - 1)
        Quick_Sort(lista, k + 1, fim)

def particiona(lista, inicio, fim):
    i, j = inicio, fim
    pivo = lista[fim]
    while True:
        # aumentado i
        while i < j and lista[i] <= pivo: i = i + 1
        if i < j:
            lista[i], lista[j] = pivo, lista[i]
        else: break
        # diminuindo j
        while i < j and lista[j] >= pivo: j = j - 1
        if i < j: lista[i], lista[j] = lista[j], pivo
```

```
        else: break
    return i

# testes
print("\n* * * Teste - 1")
minha_lista = [72, 17, 12, 45, 38, 23, 86, 32]
k = Quick_Sort(minha_lista, 0, len(minha_lista) - 1)
print(minha_lista)

print("\n* * * Teste - 2")
minha_lista = [37, 19, 49, 27]
k = Quick_Sort(minha_lista, 0, len(minha_lista) - 1)
print(minha_lista)

print("\n* * * Teste - 3")
minha_lista = [8, 15, 39, 53]
k = Quick_Sort(minha_lista, 0, len(minha_lista) - 1)
print(minha_lista)

print("\n* * * Teste - 4")
minha_lista = [53, 39, 15, 8]
k = Quick_Sort(minha_lista, 0, len(minha_lista) - 1)
print(minha_lista)
```

Será impresso:

```
* * * Teste - 1

Lista Total: [72, 17, 12, 45, 38, 23, 86, 32]
Classificando Trecho: [72, 17, 12, 45, 38, 23, 86, 32]
pivo: 32
Trecho Particionado: [23, 17, 12, 32, 38, 45, 86, 72]

Lista Total: [23, 17, 12, 32, 38, 45, 86, 72]
Classificando Trecho: [23, 17, 12]
pivo: 12
Trecho Particionado: [12, 17, 23]

Lista Total: [12, 17, 23, 32, 38, 45, 86, 72]
Classificando Trecho: []

Lista Total: [12, 17, 23, 32, 38, 45, 86, 72]
Classificando Trecho: [17, 23]
pivo: 23
Trecho Particionado: [17, 23]
```

```
Lista Total: [12, 17, 23, 32, 38, 45, 86, 72]
Classificando Trecho: [17]

Lista Total: [12, 17, 23, 32, 38, 45, 86, 72]
Classificando Trecho: []

Lista Total: [12, 17, 23, 32, 38, 45, 86, 72]
Classificando Trecho: [38, 45, 86, 72]
pivo: 72
Trecho Particionado: [38, 45, 72, 86]

Lista Total: [12, 17, 23, 32, 38, 45, 72, 86]
Classificando Trecho: [38, 45]
pivo: 45
Trecho Particionado: [38, 45]

Lista Total: [12, 17, 23, 32, 38, 45, 72, 86]
Classificando Trecho: [38]

Lista Total: [12, 17, 23, 32, 38, 45, 72, 86]
Classificando Trecho: []

Lista Total: [12, 17, 23, 32, 38, 45, 72, 86]
Classificando Trecho: [86]
[12, 17, 23, 32, 38, 45, 72, 86]

* * * Teste - 2

Lista Total: [37, 19, 49, 27]
Classificando Trecho: [37, 19, 49, 27]
pivo: 27
Trecho Particionado: [19, 27, 49, 37]

Lista Total: [19, 27, 49, 37]
Classificando Trecho: [19]

Lista Total: [19, 27, 49, 37]
Classificando Trecho: [49, 37]
pivo: 37
Trecho Particionado: [37, 49]

Lista Total: [19, 27, 37, 49]
Classificando Trecho: []
```

```
Lista Total: [19, 27, 37, 49]
Classificando Trecho: [49]
[19, 27, 37, 49]

* * * Teste - 3

Lista Total: [8, 15, 39, 53]
Classificando Trecho: [8, 15, 39, 53]
pivo: 53
Trecho Particionado: [8, 15, 39, 53]

Lista Total: [8, 15, 39, 53]
Classificando Trecho: [8, 15, 39]
pivo: 39
Trecho Particionado: [8, 15, 39]

Lista Total: [8, 15, 39, 53]
Classificando Trecho: [8, 15]
pivo: 15
Trecho Particionado: [8, 15]

Lista Total: [8, 15, 39, 53]
Classificando Trecho: [8]

Lista Total: [8, 15, 39, 53]
Classificando Trecho: []

Lista Total: [8, 15, 39, 53]
Classificando Trecho: []

Lista Total: [8, 15, 39, 53]
Classificando Trecho: []
[8, 15, 39, 53]

* * * Teste - 4

Lista Total: [53, 39, 15, 8]
Classificando Trecho: [53, 39, 15, 8]
pivo: 8
Trecho Particionado: [8, 39, 15, 53]

Lista Total: [8, 39, 15, 53]
Classificando Trecho: []

Lista Total: [8, 39, 15, 53]
```

```
Classificando Trecho: [39, 15, 53]
pivo: 53
Trecho Particionado: [39, 15, 53]

Lista Total: [8, 39, 15, 53]
Classificando Trecho: [39, 15]
pivo: 15
Trecho Particionado: [15, 39]

Lista Total: [8, 15, 39, 53]
Classificando Trecho: []

Lista Total: [8, 15, 39, 53]
Classificando Trecho: [39]

Lista Total: [8, 15, 39, 53]
Classificando Trecho: []
[8, 15, 39, 53]
```

Observe principalmente os dois últimos testes, onde a sequência original está classificada e invertida respectivamente.

6.5 Classificação – método Quick - versão não recursiva

A recursão é usada só para guardar a ordem das subsequências que devem ser classificadas. Podemos em vez de recursão usar uma pilha para guardar as tais subsequências.

Vamos usar a nossa ADT de pilha PilhaLista já definida. A pilha irá conter o inicio e fim (dupla) de cada sub-lista ainda não classificada.
A função Quick_Sort não precisa agora receber os parâmetros inicio e fim da lista:

```
def Quick_Sort(lista):
    # Cria a pilha de sub-listas e inicia com lista completa
    Pilha = PilhaLista()
    Pilha.push((0, len(lista) - 1))
    # Repete até que a pilha de sub-listas esteja vazia
    while not Pilha.is_empty():
        inicio, fim = Pilha.pop()
        # Só particiona se há mais de 1 elemento
        if fim - inicio > 0:
            k = particiona(lista, inicio, fim)
            # Empilhe as sub-listas resultantes
            Pilha.push((inicio, k - 1))
            Pilha.push((k + 1, fim))
```

6.6 Testes – Quick não recursivo

Veja abaixo alguns testes da versão não recursiva. Acrescentamos alguns print(), apenas para mostrar o funcionamento do algoritmo.
Vamos importar a classe PilhaLista do módulo PilhaLista. Portanto dentro do diretório corrente deve estar o módulo PilhaLista.py e também um módulo vazio de nome módulo __init__.py. Esse último indica ao sistema Python a permissão para importar desse diretório.

```
from PilhaLista import PilhaLista
def Quick_Sort(lista):
    # Cria a pilha de sub-listas e inicia com lista completa
    Pilha = PilhaLista()
    Pilha.push((0, len(lista) - 1))
    # Repete até que a pilha de sub-listas esteja vazia
    while not Pilha.is_empty():
        print("\nPilha = ", Pilha)
        print("Lista = ", lista)
        inicio, fim = Pilha.pop()
        # Só particiona se há mais de 1 elemento
        if fim - inicio > 0:
            k = particiona(lista, inicio, fim)
            # Empilhe as sub-listas resultantes
            Pilha.push((inicio, k - 1))
            Pilha.push((k + 1, fim))

def particiona(lista, inicio, fim):
    i, j = inicio, fim
    pivo = lista[fim]
    while True:
        # aumentado i
        while i < j and lista[i] <= pivo: i = i + 1
        if i < j:
            lista[i], lista[j] = pivo, lista[i]
        else: break
        # diminuindo j
        while i < j and lista[j] >= pivo: j = j - 1
        if i < j: lista[i], lista[j] = lista[j], pivo
        else: break
    return i

# testes
print("\n* * * Teste - 1")
minha_lista = [72, 17, 12, 45, 38, 23, 86, 32]
k = Quick_Sort(minha_lista)
print(minha_lista)
```

```
print("\n* * * Teste - 2")
minha_lista = [37, 19, 49, 27]
k = Quick_Sort(minha_lista)
print(minha_lista)

print("\n* * * Teste - 3")
minha_lista = [8, 15, 39, 53]
k = Quick_Sort(minha_lista)
print(minha_lista)

print("\n* * * Teste - 4")
minha_lista = [53, 39, 15, 8]
k = Quick_Sort(minha_lista)
print(minha_lista)
```

A saída será:

```
* * * Teste - 1

Pilha =  [(0, 7)]
Lista =  [72, 17, 12, 45, 38, 23, 86, 32]

Pilha =  [(0, 2), (4, 7)]
Lista =  [23, 17, 12, 32, 38, 45, 86, 72]

Pilha =  [(0, 2), (4, 5), (7, 7)]
Lista =  [23, 17, 12, 32, 38, 45, 72, 86]

Pilha =  [(0, 2), (4, 5)]
Lista =  [23, 17, 12, 32, 38, 45, 72, 86]

Pilha =  [(0, 2), (4, 4), (6, 5)]
Lista =  [23, 17, 12, 32, 38, 45, 72, 86]

Pilha =  [(0, 2), (4, 4)]
Lista =  [23, 17, 12, 32, 38, 45, 72, 86]

Pilha =  [(0, 2)]
Lista =  [23, 17, 12, 32, 38, 45, 72, 86]

Pilha =  [(0, -1), (1, 2)]
Lista =  [12, 17, 23, 32, 38, 45, 72, 86]

Pilha =  [(0, -1), (1, 1), (3, 2)]
Lista =  [12, 17, 23, 32, 38, 45, 72, 86]
```

```
Pilha =  [(0, -1), (1, 1)]
Lista =  [12, 17, 23, 32, 38, 45, 72, 86]

Pilha =  [(0, -1)]
Lista =  [12, 17, 23, 32, 38, 45, 72, 86]
[12, 17, 23, 32, 38, 45, 72, 86]

* * * Teste - 2

Pilha =  [(0, 3)]
Lista =  [37, 19, 49, 27]

Pilha =  [(0, 0), (2, 3)]
Lista =  [19, 27, 49, 37]

Pilha =  [(0, 0), (2, 1), (3, 3)]
Lista =  [19, 27, 37, 49]

Pilha =  [(0, 0), (2, 1)]
Lista =  [19, 27, 37, 49]

Pilha =  [(0, 0)]
Lista =  [19, 27, 37, 49]
[19, 27, 37, 49]

* * * Teste - 3

Pilha =  [(0, 3)]
Lista =  [8, 15, 39, 53]

Pilha =  [(0, 2), (4, 3)]
Lista =  [8, 15, 39, 53]

Pilha =  [(0, 2)]
Lista =  [8, 15, 39, 53]

Pilha =  [(0, 1), (3, 2)]
Lista =  [8, 15, 39, 53]

Pilha =  [(0, 1)]
Lista =  [8, 15, 39, 53]

Pilha =  [(0, 0), (2, 1)]
Lista =  [8, 15, 39, 53]
```

```
Pilha =  [(0, 0)]
Lista =  [8, 15, 39, 53]
[8, 15, 39, 53]

* * * Teste - 4

Pilha =  [(0, 3)]
Lista =  [53, 39, 15, 8]

Pilha =  [(0, -1), (1, 3)]
Lista =  [8, 39, 15, 53]

Pilha =  [(0, -1), (1, 2), (4, 3)]
Lista =  [8, 39, 15, 53]

Pilha =  [(0, -1), (1, 2)]
Lista =  [8, 39, 15, 53]

Pilha =  [(0, -1), (1, 0), (2, 2)]
Lista =  [8, 15, 39, 53]

Pilha =  [(0, -1), (1, 0)]
Lista =  [8, 15, 39, 53]

Pilha =  [(0, -1)]
Lista =  [8, 15, 39, 53]
[8, 15, 39, 53]
```

6.7 Método Quick – análise simplificada

O processo de partição da tabela é proporcional a **n**. O problema é saber quantas partições serão necessárias.
No **melhor caso**, cada partição é dividida em duas de igual tamanho ou de tamanhos que diferem apenas de um. Teremos assim um caso análogo ao do mergesort e o tempo é proporcional a **n.log(n)**.
No **pior caso**, quando a tabela já está classificada ou invertida, cada partição é dividida em uma partição de 0 e outra de n-1 elementos. Portanto o tempo é proporcional a $\mathbf{n^2}$.

No caso médio, a partição pode estar em qualquer lugar, mas o tempo também é proporcional a **n.log(n)**. Veja a seguir a demonstração.

Seja C(n) a quantidade de comparações necessárias para uma sequência de n elementos.
São necessárias n-1 comparações para realizar a primeira partição que pode ocorrer em qualquer lugar da sequência. O cálculo de C(n) deve levar em conta a média das comparações de cada partição.

$C(n) = (n-1) + (1/n).(C(0) + C(n-1) + C(1) + C(n-2) + ... + C(n-1) + C(0))$
$C(n) = (n-1) + (2/n).(C(0) + C(1) + ... + C(n-1)) = (n-1) + (2/n).C(n-1) + (2/n).\sum C(k)\ (k = 0, n-2)$

$n.C(n) = n.(n-1) + 2.C(n-1) + 2.\sum C(k)\ (k = 0, n-2)$ [A]

Da mesma forma:
$C(n-1) = (n-2) + (2/(n-1)).(C(0) + C(1) + ... + C(n-2)) = (n-2) + (2/(n-1)).\sum C(k)\ (k = 0, n-2)$
$(n-1).C(n-1) = (n-1).(n-2) + 2.\sum C(k)\ (k = 0, n-2)$ [B]

Subtraindo [B] de [A]:

$n.C(n) – (n-1).C(n-1) = 2.(n-1) + 2.C(n-1)$
$n.C(n) = 2.(n-1) + (n+1).C(n-1) \leq 2.n + (n+1).C(n-1)$

Dividindo ambos os lados por n.(n+1):

$C(n)/(n+1) \leq 2/(n+1) + C(n-1)/n \leq 2/(n+1) + 2/n + C(n-2)/(n-1)$

$\leq 2/(n+1) + 2/n + 2/(n-1) + C(n-3)/(n-2)$
$\leq 2/(n+1) + 2/n + 2/(n-1) + 2/(n-2) + C(n-4)/(n-3)$
$\leq 2/(n+1) + 2/n + 2/(n-1) + 2/(n-2) + ... C(2)/3$ (C(1) = C(0) = 0 e C(2) = 1)
$= 2/(n+1) + 2/n + 2/(n-1) + 2/(n-2) + ... + 1/3$
$\leq 2/n + 2/(n-1) + 2/(n-2) + ... + 1/3 + 1/2$
$= 2.\sum 1/k\ (k = 2, n)$

Observe o gráfico abaixo e conclua veja que a área de 1.(1/2) + 1.(1/3) + ... + 1.(1/n) é menor que a integral de 1/x calculada de 1 a n.

Assim:

$C(n)/(n+1) \leq \int (1/x).dx\ (x = 1, n) = \ln(n) – \ln(1) = \ln(n)$

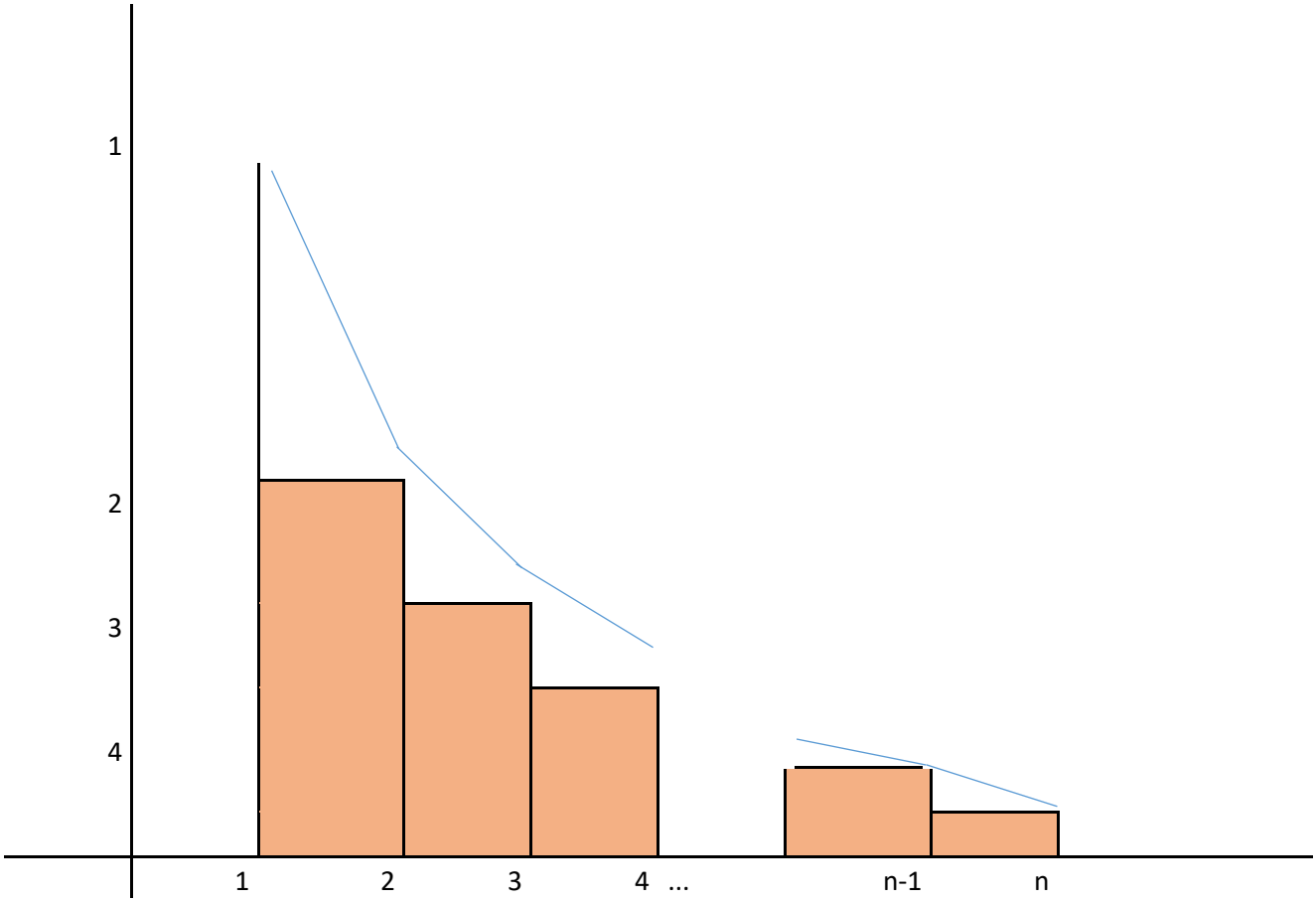
Portanto:

$C(n) \leq (n+1).\ln(n).$

Portanto, o Quick é na média também O(n.log(n))

Em resumo a complexidade do método Quick é:

pior caso	**O(n^2)**
melhor caso	**O(n.log n)**
caso médio	**O(n.log n)**



6.8 Merge e Quick

O algoritmo que particiona a sequência, não usa um vetor auxiliar como no caso do algoritmo de intercalação de sequências ordenadas. Isso torna o Quick melhor para sequências muito grandes, pois menos memória é usada.

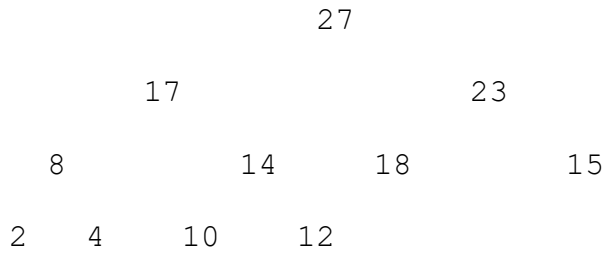
Se o uso de memória não for um problema, podemos também construir o Quick de forma análoga ao Merge. Basta escolher como pivô o elemento do meio da tabela (na verdade pode ser qualquer um) e dividir a tabela em três partes ou 3 tabelas distintas: os menores que o pivô, os iguais e os maiores. Em seguida aplica-se o mesmo método nos menores e nos maiores.

```
def Quick_Sort(lista):
    # lista com zero ou 1 elementos já está classificada
    if len(lista) <= 1:
        return lista
    pivot = lista[len(lista) // 2]          # elemento do meio
```

```
    # constrói as novas listas
    left = [x for x in lista if x < pivot]    # menores
    middle = [x for x in lista if x == pivot] # iguais
    right = [x for x in lista if x > pivot]   # maiores
    # retorna uma nova lista com as 3 partes, aplicando
    # o Quick nas partes dos menores e dos maiores
    return Quick_Sort(left) + middle + Quick_Sort(right)
```

7. Classificação - método Heap

Uma árvore binária é um Heap se e só se cada nó que não é folha armazena um elemento maior que seus filhos. A árvore binária abaixo é Heap.
Outra característica de uma árvore Heap é que deve ser construída, preenchendo totalmente cada nível da esquerda para a direita.



Armazenando os elementos numa lista na ordem em que aparecem nível a nível:

índice	1	2	3	4	5	6	7	8	9	10	11
valor	27	17	23	8	14	18	15	2	4	10	12

Cada nó **i** tem filhos **2i** e **2i+1** e **a[i] ≥ a[2i]** e **a[i] ≥ a[2i+1]**.

As folhas estão a partir do elemento **a[n/2+1]** até **a[n]**.

Para o caso do algoritmo Heap (Heapsort) vamos considerar a lista **a[1..n]** e não **a[0..n-1]**. O que em Python, implica em declarar o vetor com um elemento a mais que o máximo e ignorar **a[0]**.

Abaixo a função **Heap(a, k, n)** que arruma todos os descendentes (filhos, netos, bisnetos, etc.) de **a[k]** até o final da lista.

A função Heap - versão recursiva

Nessa versão, basta verificar se os dois filhos de **a[k]** são menores ou iguais e chamar Heap recursivamente para esses filhos:

```
# versão recursiva
# aplica o Heap em a[k] na lista de n elementos
def Heap(a, k, n):
    # compara o filho esquerdo
    if 2*k <= n and a[k] < a[2*k]:
        # troca com o pai e aplica Heap ao novo filho
        a[k], a[2*k] = a[2*k], a[k]
        Heap(a, 2*k, n)
    # compara com o filho direito
    if 2*k+1 <= n and a[k] < a[2*k+1]:
        # troca com o pai e aplica Heap ao novo filho
        a[k], a[2*k+1] = a[2*k+1], a[k]
        Heap(a, 2*k+1, n)
```

A função Heap - versão não recursiva

```
# versão não recursiva
# aplica o Heap em a[k] na lista de n elementos
def Heap(a, k, n):
    # verifica a situação Heap com os filhos, netos, etc.
    while 2 * k <= n:
       j = 2 * k
       # decide o maior entre a[j] e a[j+1] (filhos)
       if j < n and a[j] < a[j + 1]: j += 1
       # neste ponto a[j] é o maior entre os 2 filhos
       # verifica se deve trocar com o pai
       if a[j] > a[k]:
           a[j], a[k] = a[k], a[j]
       else: break; # retorna pois nada mais a fazer
       # se trocou pode ser que a troca introduziu algo não Heap
       # verifica portanto o filho que foi trocado
       k = j
```

Classificação - método Heap

Note agora que se a tabela é Heap e **a[1]** é o maior. Portanto, para ordenar a tabela:

a) troca-se **a[1]** com **a[n]**
b) **a[n]** já está em seu lugar
c) **a[1]** não está em seu lugar mas abaixo dele <u>todos estão arrumados em Heap.</u>
d) aplica-se o Heap em **a[1],** pois é o único que está em posição errada
e) repetir a partir do início com a tabela agora com **n-1** elementos

```
# O método Heapsort - Neste método, a lista tem n + 1 elementos
# a[1], a[2], ..., a[n]. O elemento a[0] é ignorado
def Heapsort(a):
    n = len(a) - 1
    # aplica o Heap aos elementos acima da metade
```

```
    for k in range(n // 2, 0, -1):
        Heap(a, k, n)
    # a[1] é o maior. Troca com o último que já fica em seu lugar
    # aplica Heap em a[1] numa tabela com 1 elemento a menos
    while n > 1:
        a[1], a[n] = a[n], a[1]
        Heap(a, 1, n - 1)
        n -= 1
```

Classificação - método Heap – análise simplificada

A função Heap acima, a partir de **k**, verifica **a[2k]** e **a[2k+1]**, **a[4k]** e **a[4k+1]** e assim por diante, até **a[2^ik]** e **a[2^ik + 1]** até que estes índices sejam menores ou iguais a **n**. Portanto o tempo é proporcional a **log n** (base 2).

A função **Heapsort** acima chama a **Heap n/2+n = 3n/2** vezes (o **for** é executado **n/2** vezes e o **while** é executado **n** vezes)
Dessa forma, o tempo é proporcional a **3n/2.log(n)**, ou seja proporcional a **n.log(n)**. Assim, o método Heap é **O(n.log(n))**.

Animação

Vejam a animação dos vários algoritmos no site: http://www.nicholasandre.com.br/sorting/

Classificação - Exercícios

1. Classifique a sequência abaixo pelos métodos Merge, Quick, Heapsort e Bubble. Em cada um dos casos diga qual o número de comparações e de trocas feitas para a classificação.

12 23 5 9 0 4 1 12 21 2 5 14

2. Encontre permutações de (1, 2, 3, 4, 5) que ocasionem:

a. Número de trocas máximo no Quick.
b. Número de trocas máximo no Bubble.

3. Dado uma lista **V** de **N** elementos, em ordem decrescente, escreva um algoritmo que a deixe classificado em ordem crescente.
Quantas trocas foram necessárias?

4. Em cada situação abaixo discuta sobre qual algoritmo de ordenação usar:

a. Um vetor de inteiros de tamanho ≤ 8.

b. Uma lista de nomes parcialmente ordenada.
c. Uma lista de nomes em ordem aleatória.
d. Uma lista de inteiros positivos distintos menores que 100.
e. Um arquivo que não cabe na memória principal.

5. Sejam os seguintes procedimentos de classificação de um vetor **a** de **n** elementos:

```
def ordena(a):
    n = len(a)
    for i in range(n):
        imin = i
        # comando I
        for j in range(i + 1, n):
            # comando II
            if a[j] < a[imin]: imin = j
        # comando III
        a[i], a[imin] = a[imin], a[i]


def classifica(a):
    n = len(a)
    for i in range(2, n):
        # comando I
        for j in range(i, 0, -1):
            # comando II
            if a[j] < a[j - 1]:
                # comando III
                a[j], a[j-1] = a[j - 1], a[j]  }
```

Para cada um dos algoritmos responda às seguintes perguntas:

a. O comando (I) é executado iterativamente para **i = 1, 2, ... n-2** em **ordena** e para **i = 2, 3, ..., n-1** em **classifica**. Ao final da i-ésima iteração qual é a situação da lista **a**? Justifique.

b. Baseado em sua resposta, justifique que cada um dos procedimentos realmente ordena ou classifica a lista a.

c. Qual o número máximo e mínimo de vezes que a comparação (II) é executada e em que situações ocorre?

d. Idem para o comando (III) que troca elementos?

6. Faça um procedimento recursivo que ordena uma sequência **A** de **n > 1** elementos baseado no seguinte algoritmo:

Obtenha um número **p**, **1 ≤ p ≤ n**. Divida a sequência em duas partes, a primeira com **p** elementos e a segunda com **n-p**. Ordene cada uma das duas partes. Após isso, supondo que:

a[1] ≤ a[2] ≤ ... ≤ a[p]
e **a[p+1] ≤ a[p+2] ≤ ... ≤ a[n]**

Intercale as duas sequências de forma a obter o vetor **a** ordenado (pode usar vetores auxiliares).

7. Considere o algoritmo da questão 6 e os seguintes valores de **p**:

(i) **1**
(ii) **n/3**
(iii) **n/2**

a. Você reconhece o algoritmo com algum dos valores de **p** ?
b. Qual das três escolhas sugeridas para p é a mais eficiente?
c. Por quê ?

8. Encontre uma permutação de 1, 2, 3, 4, 5 em quem sejam necessárias 5 ou mais trocas para ordená-la pelo método Quick.

9. Seja $a_1, a_2, ..., a_n$ uma permutação qualquer de 1, 2, ..., n. Chama-se inversão a situação em que $i < j$ e $a_i > a_j$. Assim, para n=3 a permutação 2 3 1 tem 2 inversões.

Considere a permutação $a_1\ a_2 ... a_i ... a_j ... a_n$ onde $a_i > a_j$.

A permutação $a_1\ a_2 ... a_i ... a_j ... a_n$ pode ter mais inversões que a anterior?

10. Justifique porque no algoritmo Bubble o número de trocas necessárias para ordenar a sequência coincide com o número de inversões.

O módulo time() do Python

Um programa em Python pode manipular diversas medidas de tempo. As várias funções do módulo time() são as responsáveis por essas medidas. O tempo é medido em segundos.

O módulo possui várias funções. Abaixo exemplos das mais importantes:

```
# necessário importar o módulo
import time

# segundos desde 01/01/1970
ticks = time.time()
print("Numero de segundos desde as 0 hs de 01/01/1970:", ticks)

# dia e hora local - como uma tupla
localtime = time.localtime(time.time())
print("Data/Hora Local:", localtime)

# idem como um string
localtime = time.asctime( time.localtime(time.time()) )
print("Local current time :", localtime)

# idem como um string - sem parâmetro
localtime = time.ctime()
print("Local current time :", localtime)

# tempo de CPU deste processo
cpu_t1 = time.clock()

# esperar 5 segundos
time.sleep(5)

# tempo de CPU deste processo - inclui o sleep
cpu_t2 = time.clock()

print("Tempo de CPU:", cpu_t2 - cpu_t1)
```

Como medir o tempo gasto de CPU por um algoritmo?

Fica claro então que para medir o tempo gasto por um algoritmo, usamos a seguinte sequência:

```
t1 = time.clock()
# Algoritmo
. . .
t2 = time.clock()
```

```
# tempo de CPU
tempo_cpu = t2 - t1
```