1. Estatísticas de Ordem

Nesta aula, estudaremos assuntos relacionados à temática de estatísticas de ordem.

1.1. Problema de Seleção

Problemas de **seleção** são comumente encontrados em nosso cotidiano. Suponha que estejamos interessados em determinar o i-ésimo menor elemento de um conjunto A com n elementos. Como alguns casos particulares importantes, podemos citar:

- Menor elemento: $i = 1$
- Maior elemento: $i = n$

Para obter o **menor** elemento de um conjunto de dados, podemos considerar a seguinte implementação:

In [4]:
```
import numpy
global counter
```

In [2]:
```
def Min(A):

    n = len(A)
    min = A[0]

    for i in range(1,n):
        if(A[i] < min):
            min = A[i]

    return min
```

No algoritmo acima, o temos $n - 1$ comparações, ou seja, encontrar o menor elemento em um conjunto de dados quaisquer possui complexidade de tempo de $\theta(n)$. Exemplo de funcionamento:

In [3]:
```
A = numpy.random.randint(0, 20, 20)
print('Vetor de entrada: '+ str(A))
print('Menor elemento: '+ str(Min(A)))
```

```
Vetor de entrada: [16  1 15 14 12 19 13  5 10  7 12  9 10 16  2  4
16  4  8 14]
20
Menor elemento: 1
```

Suponha, agora, que desejemos encontrar o **menor** e o **maior** valor de um conjunto de dados quaisquer A com n elementos. Podemos considerar a seguinte implementação:

In [112]:
```
def Min_Max(A):

    n = len(A)
    min = max = A[0]
    counter = 0 # armazena o número de comparações

    for i in range(1, n):

        counter = counter + 1
        if(A[i] < min):
            min = A[i]

        counter = counter + 1
        if(A[i] > max):
            max = A[i]

    return min,max,counter
```

No algoritmo acima, temos $2(n-1) \in \theta(n)$ comparações, ou seja, encontrar o menor e o maior elemento em um conjunto de dados quaisquer possui complexidade de tempo de $\theta(n)$. Exemplo de funcionamento:

In [113]:
```
A = numpy.random.randint(0, 20, 20)
print('Vetor de entrada: '+ str(A))
min, max, counter = Min_Max(A)
print('Menor elemento: '+ str(min))
print('Maior elemento: '+ str(max))
print('Número de elementos: '+ str(len(A)))
print('Número de comparações: '+ str(counter))
```

```
Vetor de entrada: [ 6 11  1  2  3  5 10 19  2  7 17 18 16 12  5  9
 0  4 17  9]
Menor elemento: 0
Maior elemento: 19
Número de elementos: 20
Número de comparações: 38
```

In [114]:
```
def Min_Max_Melhorado(A):

    n = len(A)
    counter = 0 # armazena o número de compações

    if(n % 2 == 1):
        min = max = A[0]
        i = 1
    else:
        i = 2
        if(A[0] < A[1]):
            min = A[0]
            max = A[1]
        else:
            min = A[1]
            max = A[0]

    while (i+1 < n):

        counter = counter + 1
        if(A[i] > A[i+1]):
            counter = counter + 1
            if(A[i] > max):
                max = A[i]
        else:
            counter = counter + 1
            if(A[i] < min):
                min = A[i]

        i = i+2

    return min,max,counter
```

Entretanto, podemos projetar uma versão mais rápida do algoritmo acima, na qual processamos os elementos por **pares**. A ideia consiste em comparar o menor atual com o menor elemento do par, bem como o maior valor atual com o maior elemento do par. Quando n é **ímpar**, inicializamos o menor e maior valores como sendo o primeiro elemento do conjunto de dados. Caso contrário, inicializamos o máximo e mínimo comparando os dois primeiros elementos do conjunto de dados.

Nesta versão melhorada, realizamos $2n/2 - 2 = n - 2 \in \theta(n)$comparações, ou seja, como o índice i é incrementado em duas unidades, o laço executa $n/2$ vezes, sendo que temos 2 comparações em cada iteração do mesmo. Desta forma, perfazemos $2n/2$ comparações. Como o primeiro par, em geral, já foi comparado antes da execução do laço, subamtraímos duas unidades do número de comparações anterior, ou seja, $2n/2 - 2 = n - 2$. Muito embora ambas versões possuam a mesma complexidade nominal, o algoritmo `Min_Max_Melhorado` realiza um número menor de comparações na prática.

```
In [115]: A = numpy.random.randint(0, 20, 20)
print('Vetor de entrada: '+ str(A))
min, max, counter = Min_Max_Melhorado(A)
print('Menor elemento: '+ str(min))
print('Maior elemento: '+ str(max))
print('Número de elementos: '+ str(len(A)))
print('Número de comparações: '+ str(counter))
```

```
Vetor de entrada: [14 19  8  4  6 18 17 16  0  4  5 17 12 12  4  9
18  9 11  3]
Menor elemento: 0
Maior elemento: 19
Número de elementos: 20
Número de comparações: 18
```

Suponha, agora, que desejemos encontrar o i-ésimo menor elemento do vetor A. Uma solução simples seria a seguinte:

1. Ordene o vetor A.
2. Retorne A[i]

A complexidade desta solução depende, então, da complexidade do algoritmo de ordenação utilizado no passo 1. Opções boas são o Mergesort e o Heapsort, os quais possuem complexidade de tempo de $\theta(n \log n)$. Como o passo 2 possui complexidade $\theta(1)$, então a complexidade de tempo final da ideia acima é de $\theta(n \log n)$.

No entanto, existem outras soluções que melhoram essa complexidade. Retomemos ao **problema da partição**, em que a ideia é: dado um vetor $A[i \dots f]$, devolver um índice p tal que $A[e \dots p-1] \leq A[p] \leq A[p+1 \dots d]$ A Figura 1 ilustra essa situação.

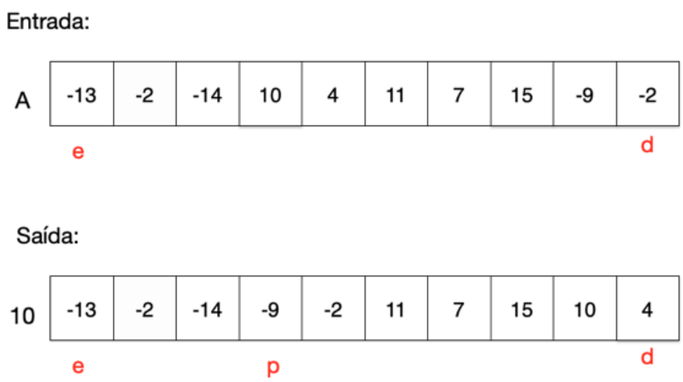

Figura 1: Problema de partição.

O algoritmo abaixo implementação a função de partição que foi utilizada na técnica Quicksort.

In [116]:
```
def Partition(A, e, d):
    pivot = A[d]
    p_index = e

    for i in range(e,d):
        if (A[i] <= pivot):
            A[i], A[p_index] = A[p_index], A[i]
            p_index = p_index + 1

    A[p_index], A[d] = A[d], A[p_index]

    return p_index

n = 10 # número de elementos do vetor A
A = numpy.random.randint(-20, 20, n)
print('Vetor de entrada: '+ str(A))
p = Partition(A, 0, n-1)
print('Vetor de saída: '+ str(A))
print('Posição do pivô: '+ str(p))
```

```
Vetor de entrada: [-20  -4   5 -13   0   4 -20  15  15  19]
Vetor de saída: [-20  -4   5 -13   0   4 -20  15  15  19]
Posição do pivô: 9
```

Note que o algoritmo acima possui um único laço que roda $\theta(n)$ vezes, como vimos na aula de Quicksort. Podemos, então, utilizá-lo para resolver o problema de selecionar o i-ésimo elemento de um vetor de elementos quaisquer. Desta forma, temos as seguintes situações após a execução da função `Partition` acima:

1. Se $p = i$, então problema resolvido!
2. Se $i < p$, então a solução do problema está em encontrar o i-ésimo menor elemento em $A[e \dots p-1]$.
3. Caso contrário, ou seja, se $i > p$, então a solução do problema está em encontrar o $(i-p)$-ésimo menor elemento em $A[p+1 \dots d]$.

Para os casos descritos nos itens 2 e 3, basta executar o algoritmo de `Partition` recursivamente até o índice desejado for atingido, conforme implementado abaixo.

In [117]:
```
def Select(A, e, d, i):
    global counter

    if(e == d):
        return A[e]

    p = Partition(A, e, d)

    k = p-e+1 # p está na k-ésima posição de A
    if(k == i):
        return A[p]
    elif(i < k):
        counter = counter + 1
        return Select(A, e, p-1, i)
    else:
        counter = counter + 1
        return Select(A, p+1, d, i-k)
```

Exemplo de funcionamento:

In [118]:
```
counter = 1 # conta quantas chamadas recursivas foram realizadas
n = 10 # número de elementos do vetor A
i = 3 # posição do i-ésimo menor elemento
A = numpy.random.randint(-20, 20, n)
print('Vetor de entrada: '+ str(A))
x = Select(A, 0, n-1, i)
print('Elemento desejado: '+ str(x))
```

```
Vetor de entrada: [ -1   8   0 -13   9 -10  19  11   8  19]
Elemento desejado: -1
```

A função `Select` possui uma chamada à função `Partition` que, como descrito anteriormente, possui complexidade de tempo de $\theta(n)$. Em seguida, caso a execução do algoritmo continue na condição $i < k$, temos que a função `Select` pode ser chamada $O(k-1)$ vezes, que seria o **tamanho do lado esquerdo** do vetor, ou seja, o lado que possui elementos menores ou iguais ao pivô. Caso contrário, a função `Select` poderia ser chamada $O(n-k)$ vezes, que corresponde ao **tamanho do lado direito** do vetor, ou seja, o lado que possui elementos maiores ou iguais ao pivô.

Assim sendo, a complexidade de tempo da função `Select` corresponde ao pior dos casos entre $O(k-1)$ e $O(n-k)$, ou seja, temos a seguinte relação de recorrência:

$$T(n) = \max_{1 \leq k \leq n} \{T(k-1), T(n-k)\} + \theta(n). \tag{1}$$

Note que, de maneira semelhante ao algoritmo Quicksort, a complexidade depende da posição do pivô. No pior caso, isto é, quando $k = n - 1$, ou seja, quando o pivô está localizado na última posição do pivô, temos que a relação de recorrência é dada por:

$$T(n) = T(n-1) + \theta(n), \tag{2}$$

a qual, como visto anteriormente, resulta em uma complexidade $O(n^2)$.

Entretanto, podemos mostrar que, no **caso médio**, a função `Select` tem complexidade de $O(n)$. A ideia é partir da premissa que o pivô possui probabilidade de estar localizado em qualquer posição do vetor, mas que na maioria das vezes ele acaba dividindo o conjunto de dados em dois subconjuntos com tamanhos similares.

Agora, vamos realizar uma pequena alteração na função `Partition` para que ela escolha o pivô de maneira **aleatória**, e não como sendo o **último** elemento do vetor. Chamaremos essa função de `Partition_Random` , bem como chamaremos de `Select_Random` sua versão modificada que faz uso do pivô escolhido de maneira aleatória.

In [119]:
```
def Partition_Random(A, e, d):

    # aqui estão as mudanças *****
    j = numpy.random.randint(e, d)
    pivot = A[j]
    A[j], A[d] = A[d], A[j]
    # ***************************

    p_index = e

    for i in range(e,d):
        if (A[i] <= pivot):
            A[i], A[p_index] = A[p_index], A[i]
            p_index = p_index + 1

    A[p_index], A[d] = A[d], A[p_index]

    return p_index

def Select_Random(A, e, d, i):
    global counter

    if(e == d):
        return A[e]

    p = Partition_Random(A, e, d)

    k = p-e+1 # p está na k-ésima posição de A
    if(k == i):
        return A[p]
    elif(i < k):
        counter = counter + 1
        return Select_Random(A, e, p-1, i)
    else:
        counter = counter + 1
        return Select_Random(A, p+1, d, i-k)
```

Exemplo de funcionamento:

In [144]:
```
counter = 1 # conta quantas chamadas recursivas foram realizadas
n = 10 # número de elementos do vetor A
i = 3 # posição do i-ésimo menor elemento
A = numpy.random.randint(-20, 20, n)
print('Vetor de entrada: '+ str(A))
x = Select(A, 0, n-1, i)
print('Número de chamadas recursivas para a função Select: '+ str(c
ounter))
print("Elemento: "+ str(x))
counter = 1
x = Select_Random(A, 0, n-1, i)
print('Número de chamadas recursivas para a função Select_Random: '
+ str(counter))
print("Elemento: "+ str(x))
```

```
Vetor de entrada: [  8  -7 -19  13 -18 -17 -14  -5 -11   8]
Número de chamadas recursivas para a função Select: 4
Elemento: -17
Número de chamadas recursivas para a função Select_Random: 1
Elemento: -17
```

Pode se constatar que, na maioria das vezes, o número de chamadas recursivas à função `Select_Random` é menor do que sua versão tradicional `Select` .