AED1 - Aula 07
Operações em vetores, ordenação por inserção (insertionsort)

Um vetor é uma estrutura de dados que armazena uma sequência de objetos do mesmo tipo em posições consecutivas da memória.

- a principal propriedade de uma sequência (ou lista) é preservar a ordem dos seus elementos.
- queremos que nossas listas aceitem certas operações básicas:
    - Pegar o conteúdo do k-ésimo item.
    - Buscar um item pelo seu conteúdo.
    - Remover um item da posição k.
    - Inserir um item na posição k.

Implementar lista em vetor

Usar um vetor v de tamanho TAM_MAX

```
#define TAM_MAX 10000
```

O vetor pode ser declarado:

- estaticamente

```
int v[TAM_MAX];
```

- dinamicamente

```
int *v = (int *)malloc(TAM_MAX * sizeof(int));
```

Operações, implementações e eficiência:

- Pegar o conteúdo do i-ésimo item leva tempo constante, i.e., O(1).

```
int pega(int v[], int n, int k)
{
   return v[k];
}
```

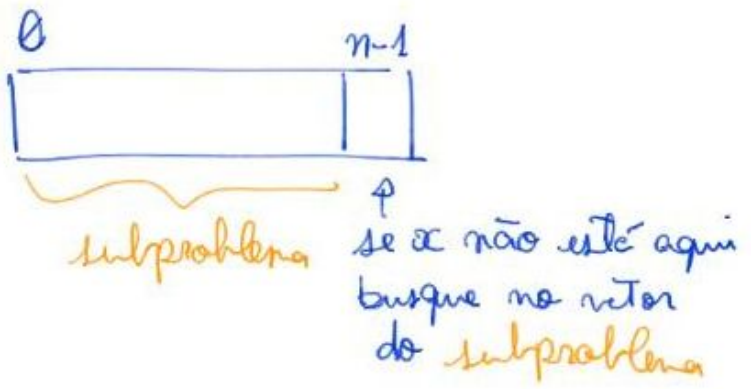
- Buscar um item (iterativa ou recursivamente) leva tempo O(n).

```
int buscaI(int v[], int n, int x)
{
   int k;
   k = n - 1;
   while (k >= 0 && v[k] != x)
       k -= 1;
   return k;
}

int buscaR(int x, int n, int v[])
{
   if (n == 0)
       return -1;
```

```
    if (x == v[n - 1])
        return n - 1;
    return buscaR(x, n - 1, v);
}
```

- o subproblema da recursão corresponde ao prefixo de tamanho n - 1 do vetor corrente, i.e., v[0 .. n - 2].

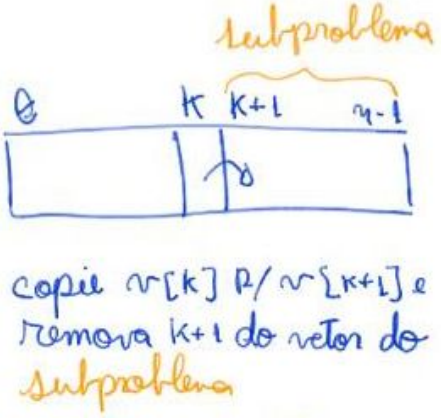


- Remover um item (iterativa ou recursivamente) leva tempo O(n).

```
int removeI(int k, int n, int v[])
{
    int j;
    for (j = k + 1; j < n; j++)
        v[j - 1] = v[j];
    return n - 1;
}

int removeR(int k, int n, int v[])
{
    if (k == n - 1)
        return n - 1;
    v[k] = v[k + 1];
    return removeR(k + 1, n, v);
}
```

- o subproblema da recursão corresponde ao sufixo de tamanho n - (k + 1) do vetor corrente, i.e., v[k + 1 .. n - 1].

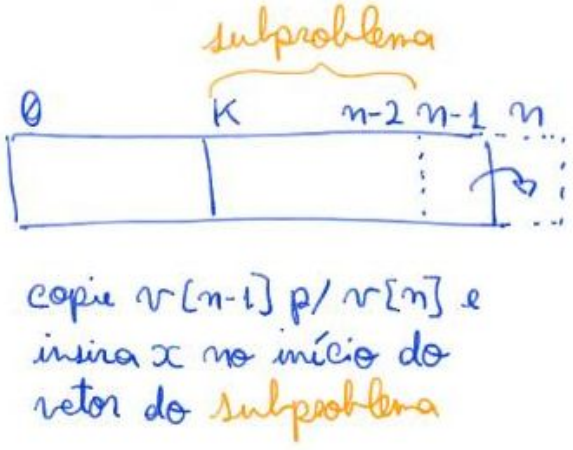


- Inserir um item (iterativa ou recursivamente) leva tempo O(n).

```
int insereI(int v[], int n, int x, int k)
{
   int j;
   for (j = n; j > k; j--)
       v[j] = v[j - 1];
   v[k] = x;
   return n + 1;
}

int insereR(int v[], int n, int x, int k)
{
   if (k == n)
       v[n] = x;
   else
   {
       v[n] = v[n - 1];
       insereR(v, n - 1, x, k);
   }
   return n + 1;
}
```

  - o subproblema da recursão corresponde ao subvetor v[k .. n - 2], que tem tamanho n - k - 1.



- bônus: remove todas as ocorrências de um elemento x.

```
int removeTodos(int v[], int n, int x)
{
   int k;
   while ((k = buscaI(v, n, x)) != -1)
       n = removeI(v, n, k);
   return n;
}
```

  - qual a eficiência de pior caso de removeTodos?

```
int removeTodos2(int v[], int n, int x)
{
   int i = 0, j;
```

```
    for (j = 0; j < n; j++)
        if (v[j] != x)
        {
            v[i] = v[j];
            i++;
        }
    return i;
}
```

- qual a eficiência de pior caso de removeTodos2?
- como mostrar que função anterior está correta?
  - qual seu invariante principal?

Problema da ordenação

Considere um vetor v de inteiros com n elementos.

Dizemos que ele está em ordem crescente se

v[0] <= v[1] <= ... <= v[n-1].

Um algoritmo para o problema da ordenação deve rearranjar (permutar) os elementos de v de modo a torná-lo crescente.

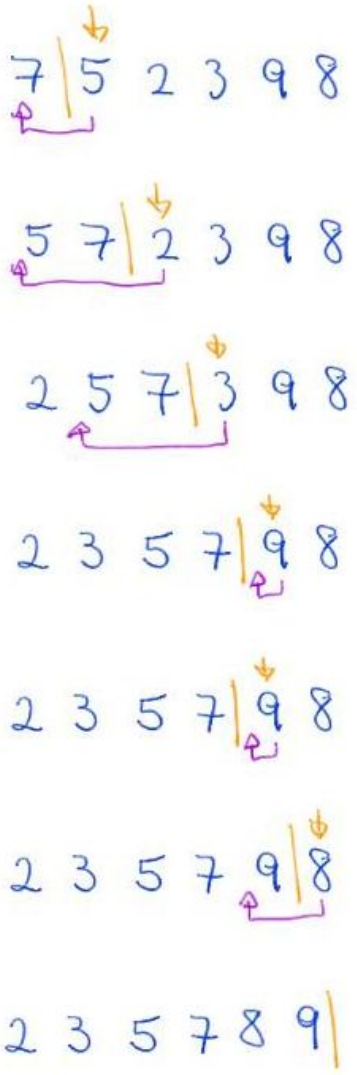
Para três algoritmos elementares veremos:

- Ideia e exemplo
- Código
- Invariante e corretude
- Eficiência de tempo:
  - no melhor e pior casos
- Estabilidade:
  - algoritmo é estável se não inverte a posição relativa de valores idênticos.
- Eficiência de espaço:
  - algoritmo é in place se não usa estruturas auxiliares (e portanto memória) com tamanho proporcional à entrada.

Insertionsort (ordenação por inserção)

Ideia e exemplo:

- Varre o vetor do início ao fim e, a cada novo elemento encontrado,
  - o coloca na posição correta no subvetor já visitado.
- Como exemplo, considere o vetor 7 5 2 3 9 8



Código:

```
void insertionSort(int v[], int n)
{
   int i, j, aux;
   for (j = 1; j < n; j++)
   {
       aux = v[j];
       for (i = j - 1; i >= 0 && aux < v[i]; i--)
           v[i + 1] = v[i];
       v[i + 1] = aux; /* por que i+1? */
   }
}
```

Invariante e corretude:

- os principais invariantes que valem no início de cada iteração são:
    - o vetor é uma permutação do original,
    - v[0 .. j - 1] está ordenado.

- invariantes do laço interno:
    - v[0..i] e v[i+2..j] são crescentes
    - v[0..i] ≤ v[i+2..j]
    - v[i+2..j] > aux
- demonstrar que esses invariantes estão corretos:
    - verificando que eles valem antes da primeira iteração
    - e que valem de uma iteração pra outra.
- verificar que, no final do laço, os invariantes implicam a corretude do algoritmo.

Eficiência de tempo:

- no melhor caso é da ordem de n, i.e., O(n).
    - Ex.: vetor está ordenado ou tem apenas adjacentes fora de ordem.
- no pior caso é da ordem de n(n-1)/2 ~= n^2/2 = O(n^2).
    - Detalhar mais...
    - Ex.: vetor está em ordem decrescente.
- Bônus:
    - já que o prefixo v[0.. j - 1] do vetor sempre está ordenado, por que não usamos busca binária para encontrar a posição de inserção do j-ésimo elemento?
        - essa variante do algoritmo funciona? Isto é, ela está correta?
        - qual sua eficiência?

Estabilidade:

- ordenação é estável.
    - por que?
- o que acontece se trocarmos "aux < v[i]" por "aux <= v[i]"?
    - Continua ordenando?
    - Continua estável?

Eficiência de espaço:

- ordenação é in place, pois só usa estruturas auxiliares (e portanto memória) de tamanho constante em relação à entrada.

Animação:

- Visualization and Comparison of Sorting Algorithms - www.youtube.com/watch?v=ZZuD6iUe3Pc