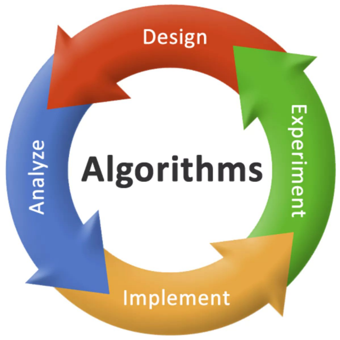
AED2 - Aula 09
Projeto e análise de algoritmos, segmento de soma máxima

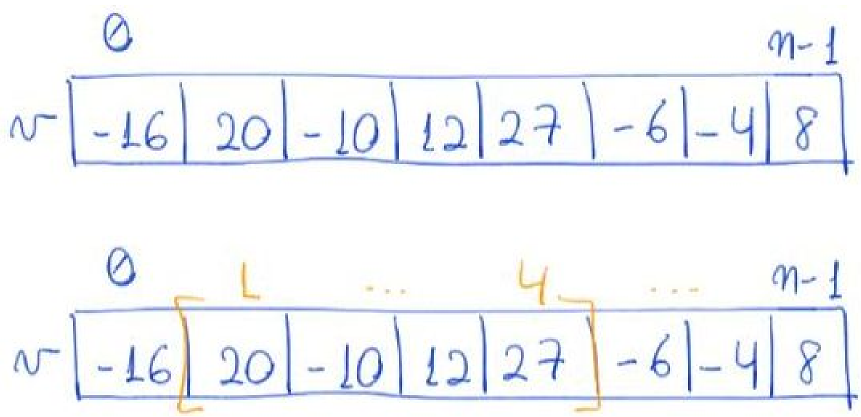
"Podemos fazer melhor?"
- mote do projetista de algoritmos



Compreensão, verificação da corretude e análise da eficiência de algoritmos iterativos para o problema do segmento de soma máxima

Problema do segmento de soma máxima

- dado um vetor v[0 .. n - 1]
- um segmento de v é qualquer subvetor de v da forma
  - v[e .. d], com 0 <= e <= d < n
  - se e > d então o segmento é vazio
- considerando que os elementos de v são inteiros
  - a soma de um segmento corresponde à soma dos seus elementos
- assim, desejamos encontrar um segmento de soma máxima
  - i.e., um segmento cuja soma dos elementos seja >= que a soma dos elementos de qualquer outro segmento de v
- exemplo



v[e..d] = v[1..4] = 49
é o seg. de soma máxima

Observem que um segmento é determinado pelos seus extremos (e, d)

- portanto, dado um vetor de tamanho v, existem
  - (n escolhe 2) = n (n - 1) / 2 segmentos diferentes

Além disso, podemos calcular o valor de um segmento

- em tempo linear no tamanho do segmento,
  - i.e., proporcional a (d - e),
- usando a seguinte rotina

```
void somaSeg(int v[], int e, int d, int *s)
{
   int i;
   *s = 0;
   for (i = e; i <= d; i++)
       *s += v[i];
}
```

Corretude de algoritmos iterativos:

- enunciar relações invariantes que valem ao longo das iterações
  - *s é a soma dos elementos em v[e .. i - 1]
- mostrar que as relações valem no início da primeira iteração
  - no início *s = 0 e v[e .. i - 1] = v[e .. e - 1] é vazio
    - portanto, o resultado vale trivialmente
- mostrar que, se as relações valem no início de uma iteração qualquer
  - então elas continuam valendo no início da próxima iteração.
  - no início da i-ésima iteração *s = v[e] + … + v[i - 1]
    - depois da instrução *s += v[i] temos
      - *s = v[e] + … + v[i - 1] + v[i]
    - e, ao final da iteração, ocorre i++
    - portanto, no início da próxima iteração
      - *s = v[e] + … + v[i - 1]
- verificar que, quando os laços terminam,
  - os invariantes implicam a corretude do algoritmo.
  - quando o laço termina i = d + 1
  - pelo invariante, *s = v[e] + … + v[i - 1] = v[e] + … + v[d]
    - portanto, o algoritmo devolve a soma do segmento v[e .. d].

Combinando as ideias, podemos

- verificar a soma de cada um dos (n escolhe 2) segmentos e pegar o maior
- esta ideia é implementada no nosso primeiro algoritmo

```
void segMax3(int v[], int n, int *e, int *d, int *sMax)
{
   int i, j, k, sAux;
   *sMax = 0;
   *e = *d = -1;
   for (i = 0; i < n; i++)
```

```
        for (j = i; j < n; j++)
        {
            sAux = 0;
            for (k = i; k <= j; k++)
                sAux += v[k];
            if (sAux > *sMax)
            {
                *sMax = sAux;
                *e = i;
                *d = j;
            }
        }
}
```

Invariantes e corretude:

- observe que na i-ésima iteração do laço externo
  - calculamos a soma de todos os segmentos que começam em i
- de modo semelhante, na j-ésima iteração do segundo laço
  - calculamos a soma dos segmentos que terminam em j
- e o laço mais interno é responsável apenas por somar os valores em v[i .. j]
- os principais invariantes do laço externo são
  - v[*e .. *d] é um segmento de soma máxima com *e < i
  - *sMax = v[*e] + … + v[*d]

Eficiência:

- tem número de operações no pior caso da ordem de n^3
  - por conta dos três laços aninhados,
  - cada um podendo ter tamanho da ordem de n.

Observando o problema, podemos perceber que

- a soma de um segmento v[e .. d] corresponde
  - à soma do seguimento v[e .. d - 1] com v[d]
- analisando o algoritmo segMax3 atentos à essa observação, percebemos que
  - ele recalcula muitas vezes a soma dos mesmos segmentos
  - eliminando esses recálculos temos nosso segundo algoritmo

```
void segMax2(int v[], int n, int *e, int *d, int *sMax)
{
   int i, j, sAux;
   *sMax = 0;
   *e = *d = -1;
   for (i = 0; i < n; i++)
   {
       sAux = 0;
       for (j = i; j < n; j++)
```

```
        {
            sAux += v[j];
            if (sAux > *sMax)
            {
                *sMax = sAux;
                *e = i;
                *d = j;
            }
        }
    }
}
```

Invariantes e corretude:

- observe que na i-ésima iteração do laço externo
  - calculamos a soma de todos os segmentos que começam em i
- de modo semelhante, na j-ésima iteração do laço interno
  - calculamos a soma dos segmentos que terminam em j
- os principais invariantes do laço externo são
  - v[*e .. *d] é um segmento de soma máxima com *e < i
  - *sMax = v[*e] + … + v[*d]

Eficiência:

- tem número de operações no pior caso da ordem de n^2
  - por conta dos dois laços aninhados,
  - cada um podendo ter tamanho da ordem de n.

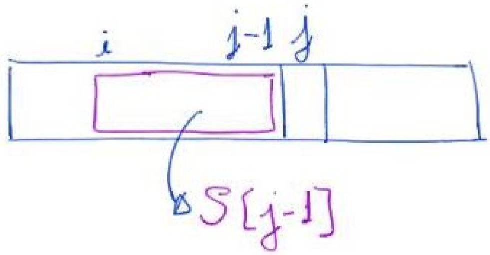
Será que conseguimos fazer melhor?

Vamos considerar um raciocínio recursivo (ou indutivo)

- seja S[j - 1] = v[i .. j - 1] um segmento de soma máxima
  - que termina e contém j - 1
- queremos calcular S[j],
  - i.e., o segmento de soma máxima que termina e contém j
  - podemos usar S[j - 1] = v[i .. j - 1] para tanto?
    - note que nenhum outro segmento terminado em j - 1
      - pode contribuir tanto quanto S[j - 1]
  - se S[j - 1] >= 0 então acrescentamos a ele v[j]
    - i.e., S[j] = v[i .. j] é o segmento de soma máxima que termina e contém j
  - caso contrário, então todo segmento que termina e contém j - 1 tem soma < 0,
    - i.e., nenhum destes segmentos contribui para S[j]

        - portanto, S[j] = v[j] é o segmento de soma máxima que termina e contém j

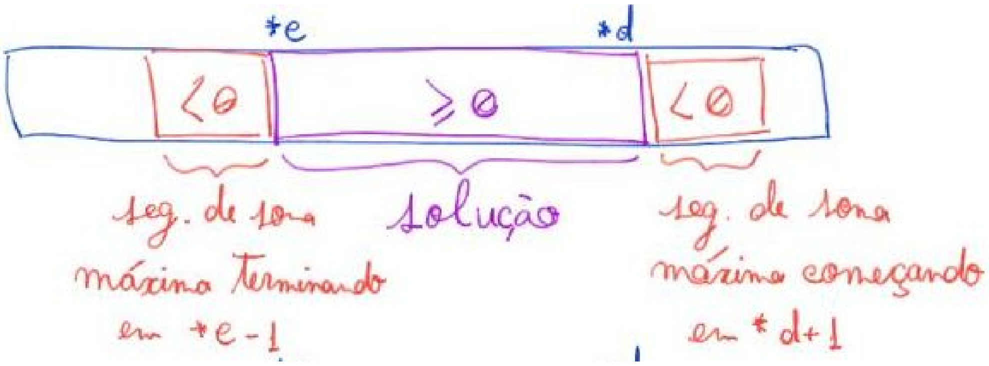
Seja $S[j]$ o seg. de soma máx. que termina e contém j



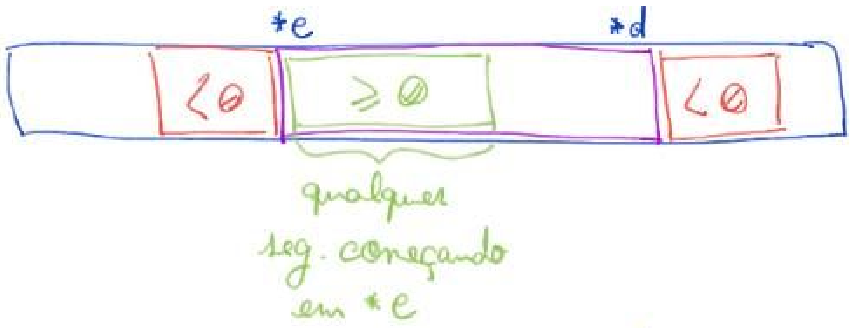
se $S[j-1] < 0$ então
$S[j] = v[j]$ e $i = j$,
pois $S[j-1]$ não contribui

se $S[j-1] > 0$ então
$S[j] = v[j] + S[j-1]$

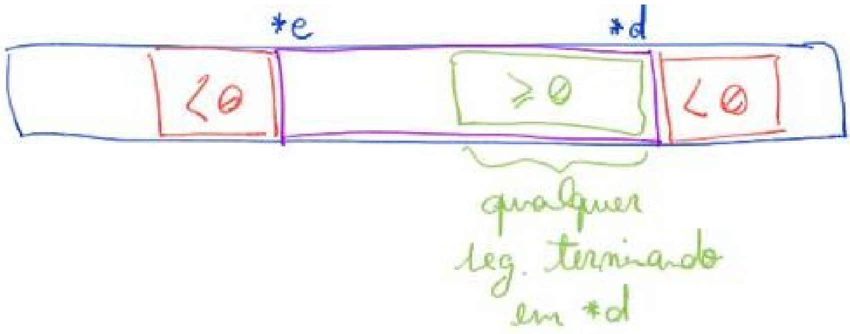
- Observe que a generalização deste raciocínio recursivo nos permite concluir algumas propriedades interessantes sobre uma solução ótima qualquer:
    - considerando uma solução no segmento v[*e .. *d]
    - qualquer segmento que termina em *e - 1, i.e., da forma v[k .. *e - 1]
        - tem soma < 0
            - caso contrário este teria sido adicionado à solução
    - o mesmo vale para qualquer segmento que começa em *d + 1, i.e., da forma v[*d + 1 .. k]



    - de modo complementar, qualquer segmento que começa em *e e está contido no subvetor da solução, i.e., da forma v[*e .. k] com k <= *d
        - tem soma >= 0
            - caso contrário este teria sido excluído da solução



- o mesmo vale para qualquer segmento que termina em *d e está contido no subvetor da solução, i.e., da forma v[k .. *d] com k >= *e



- essa ideia está por trás do seguinte algoritmo

```
void segMax1(int v[], int n, int *e, int *d, int *sMax)
{
   int i, j, sAux;
   *sMax = 0;
   *e = *d = -1;
   sAux = 0;
   for (i = j = 0; j < n; j++)
   {
       if (sAux >= 0)
           sAux += v[j];
       else // sAux < 0
       {
           sAux = v[j];
           i = j;
       }
       if (sAux > *sMax)
       {
           *sMax = sAux;
           *e = i;
           *d = j;
       }
   }
}
```

Invariante e corretude:

- observe que na j-ésima iteração do laço

- v[*e .. *d] é um segmento de soma máxima com *d < j
- *sMax = v[*e] + … + v[*d]
- v[i .. j - 1] é um segmento de soma máxima com término em j - 1
- sAux = v[i] + … + v[j - 1]

Eficiência:

- tem número de operações no pior caso da ordem de n.

Uma versão levemente diferente, mas equivalente ao algoritmo anterior

```
void segMax0(int v[], int n, int *e, int *d, int *sMax)
{
   int i, j, sAux;
   *sMax = 0;
   *e = *d = -1;
   sAux = 0;
   for (i = j = 0; j < n; j++)
   {
       if (sAux + v[j] > 0)
           sAux += v[j];
       else // sAux + v[j] <= 0
       {
           sAux = 0;
           i = j + 1;
       }
       if (sAux > *sMax)
       {
           *sMax = sAux;
           *e = i;
           *d = j;
       }
   }
}
```

- note que este algoritmo usa uma variante da ideia recursiva descrita anteriormente
    - na qual S[j - 1] = v[i .. j - 1] é um segmento de soma máxima
        - que termina em j - 1, mas não necessariamente o contém
            - i.e., o seguimento vazio é uma opção