Construção e Análise de Algoritmos

aula 14: O problema do subconjunto máximo

1 Introdução

Imagine que L é uma lista que contém números inteiros positivos.

Por exemplo,

L	8	2	5	11	29	23	9	16	8	10	14	11

A tarefa consiste em

→ *Selecionar um subconjunto de elementos cuja soma é a maior possível*

Mas, é preciso obedecer a seguinte restrição

◦ *Não se pode selecionar dois elementos consecutivos*

E agora?

Como é que a gente faz isso?

1.1 A pior solução do mundo

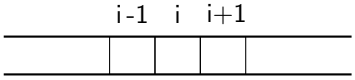
→ *O algoritmo porcaria ...*

2 Divisão e conquista

Bom, a única coisa que a gente sabe fazer é divisão e conquista.

E a seguinte observação nos dá uma primeira estratégia de decomposição

- Em qualquer grupo de 3 elementos consecutivos



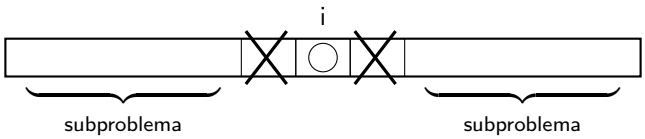
ao menos um deles vai estar na solução

— (*porque uma solução que não contém nenhum dos três pode ser melhorada adicionando o elemento do meio*)

Quer dizer, agora nós podemos considerar em separado

- as soluções que contém o elemento `L[i − 1]`
- as soluções que contém o elemento `L[i]`
- as soluções que contém o elemento `L[i + 1]`

Em cada um desses casos, a inclusão do elemento na solução quebra o problema em dois



A ideia é que essa estratégia pode ser aplicada recursivamente.

E isso já nos dá um algoritmo de divisão e conquista

```
T(n)   =   Algoritmo  SSMax-DC ( L, inicio, fim )
           {
              Se ( fim < inicio + 2 )                //  menos do que 3 elementos
    O(1)
                  Retorna ( Max{ L[inicio..fim] } )

              meio  <--  (inicio + fim) / 2

              // caso 1: L[meio-1] é selecionado
2·T(n/2) ←    S1  <--   SSMax-DC ( L, inicio, meio-3 )
                    +
                        SSMax-DC ( L, meio+1, fim )  +  L[meio-1]

              // caso 2: L[meio] é selecionado
2·T(n/2) ←    S2  <--   SSMax-DC ( L, inicio, meio-2 )
                    +
                        SSMax-DC ( L, meio+2, fim )  +  L[meio]

              // caso 3: L[meio+1] é selecionado
2·T(n/2) ←    S3  <--   SSMax-DC ( L, inicio, meio-1 )
                    +
                        SSMax-DC ( L, meio+3, fim )  +  L[meio+1]

    O(1)      Retorna ( Max{ S1, S2, S3 } )
           }
```

A partir das anotações, nós obtemos a seguinte recorrência para o tempo de execução do algoritmo

$$T(n) \quad = \quad 6 \cdot T\left(\tfrac{n}{2}\right) \quad + \quad O(1)$$

E para obter a solução, nós desenrolamos a recorrência

$$\begin{aligned} T(n) &= 6 \cdot T\left(\tfrac{n}{2}\right) + 1 \\ &= 6 \cdot \left[6 \cdot T\left(\tfrac{n}{2^2}\right) + 1\right] + 1 \\ &= 6^2 \cdot T\left(\tfrac{n}{2^2}\right) + 6 + 1 \\ &= 6^3 \cdot T\left(\tfrac{n}{2^3}\right) + 6^2 + 6 + 1 \\ &= \ldots \\ &= 6^{\log_2 n} \cdot T(1) + 6^{\log_2 n - 1} + \ldots + 6 + 1 \end{aligned}$$

Quer dizer, a gente obteve uma soma de PG.

E isso é uma coisa que a gente resolve assim

$$\begin{aligned} \mathsf{S} &= 1 + 6 + 6^2 + \ldots + 6^k \\ &= 1 + 6 \cdot \Big[\underbrace{1 + 6 + \ldots + 6^{k-1}}_{\mathsf{S} \; - \; 6^k}\Big] \end{aligned}$$

donde

$$\mathsf{S} = 1 + 6 \cdot \mathsf{S} - 6^{k+1}$$

donde

$$\mathsf{S} = \frac{6^{k+1} - 1}{5}$$

Daí que,

$$T(n) = \frac{6^{\log_2 n + 1} - 1}{5} = O\big(6^{\log_2 n}\big) = O\big(n^{\log_2 6}\big) = O\big(n^{2{,}58}\big)$$

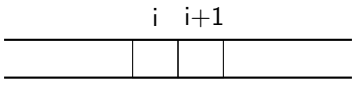
Legal.

Agora, a gente já tem um algoritmo porcaria para o problema.

Mas, como é que a gente faz melhor do que isso ?

Outra estratégia de decomposição

Bom, agora a gente pode olhar para dois elementos consecutivos

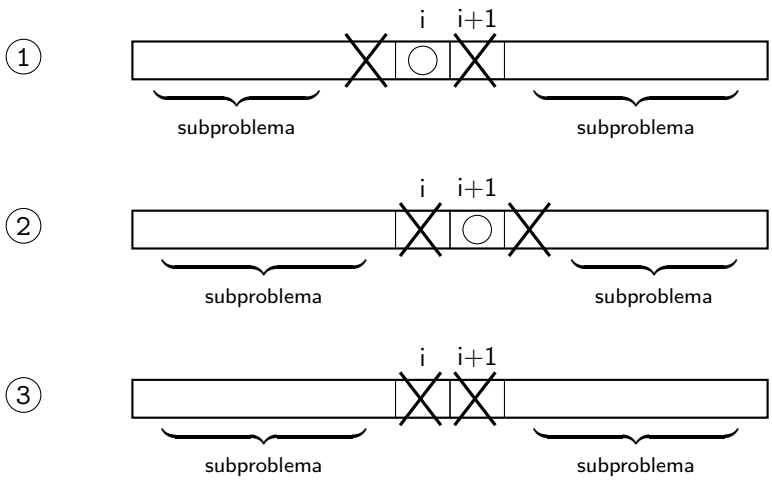


E daí, a gente nota que das 3 uma

1. ou o elemento `L[i]` está na solução

2. ou o elemento `L[i + 1]` está na solução
3. ou nenhum dos dois elemento está na solução

A seguir, a gente vê o que acontece em cada caso



Sim! os subproblemas que aparecem no caso 3 já apareceram nos casos anteriores.

Logo, eles não precisam ser resolvidos outra vez.

E daí, o algoritmo fica assim

```
T(n)   =   Algoritmo  SSMax-DC2 ( L, inicio, fim )
            {
              Se ( fim < inicio + 1 )                 //  menos do que 2 elementos
O(1)            Retorna ( Max{ L[inicio..fim] } )
              meio  <--  (inicio + fim) / 2

              // caso 1: L[meio] é selecionado
              sa  <--   SSMax-DC ( L, inicio, meio-2 )
2·T(n/2)  ←   sb        SSMax-DC ( L, meio+2, fim )
O(1)          S1  <--   sa  +  L[meio]  +  sb

              // caso 2: L[meio+1] é selecionado
              sc  <--   SSMax-DC ( L, inicio, meio-1 )
2·T(n/2)  ←   sd  <--   SSMax-DC ( L, meio+3, fim )
O(1)          S2  <--   sc  +  L[meio+1]  +  sd

              // caso 3: L[meio+1] é selecionado
O(1)          S3  <--   sc  +  sb

O(1)          Retorna ( Max{ S1, S2, S3 } )
            }
```

A partir das anotações, a gente obtém a seguinte recorrência para o tempo de execução do algoritmo

$$T(n) \quad = \quad 4 \cdot T\left(\tfrac{n}{2}\right) \quad + \quad O(1)$$

Daí, olhando para o padrão da recorrência, a gente advinha que ela desenrola assim

$$T(n) \quad = \quad 4^{\log_2 n} \cdot T(1) \quad + \quad 4^{\log_2 n - 1} \quad + \quad \ldots \quad + \quad 4 \quad + \quad 1$$

O que tem solução

$$T(n) \quad = \quad O\big(n^{\log_2 4}\big) \quad = \quad O\big(n^2\big)$$

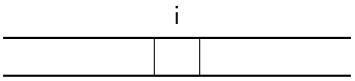
Quer dizer, o algoritmo melhorou um pouquinho.

Mas, ele ainda é uma porcaria.

E agora?

Mais uma estratégia de decomposição

Bom, agora a gente olha para um único elemento da lista



Em ambos os casos, o problema é quebrado em dois



E aqui a gente reconhece os mesmos subproblemas que apareceram na estratégia anterior.

Quer dizer, o algoritmo baseado nessa decomposição também vai fazer 4 chamadas recursivas.

E o seu tempo de execução também vai ser descrito pela recorrência

$$T(n) \quad = \quad 4 \cdot T\left(\tfrac{n}{2}\right) \quad + \quad O(1)$$

Logo, o seu tempo de execução também vai ser

$$T(n) \quad = \quad O(n^2)$$

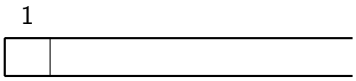
E agora?

3 Programação dinâmica

Bom, agora a gente vai tentar pensar de uma maneira diferente.

Quer dizer, ainda é possível fazer uma outra observação

- Focando no primeiro elemento da lista



só existem duas opções: ou `L[1]` está na solução, ou ele não está

Quer dizer, agora nós podemos considerar em separado

- as soluções que contém o elemento `L[1]`
- as soluções que não contém o elemento `L[1]`

Em ambos os casos, a decisão sobre `L[1]` produz um único subproblema um pouco menor



A ideia é que essa estratégia pode ser aplicada recursivamente.

E isso nos dá mais um algoritmo de divisão e conquista para o problema

```
        Algoritmo SubconjMax-DC3 ( L, inicio, n )
         {
O(1)  |    Se ( inicio = n )   Retorna ( L[n] )

           // caso 1: L[inicio] é selecionado

T(n-2) <-  S1  <--   SubconjMax-DC ( L, inicio+2, n )
                   +
                     L[inicio]

           // caso 2: L[inicio] não é selecionado

T(n-1) <-  S2  <--   SubconjMax-DC ( L, inicio+1, n )

O(1)  |    Retorna ( Max{ S1, S2 } )
         }
```

A partir das anotações, nós obtemos a seguinte recorrência para o tempo de execução do algoritmo

$$T(n) \quad = \quad T(n-1) \quad + \quad T(n-2) \quad + \quad O(1)$$

Mas, isso é quase a recorrência de Fibonacci — (*é um pouco mais que isso ...*)

Daí que,

$$T(n) \quad = \quad O(\mathsf{F}_n)$$

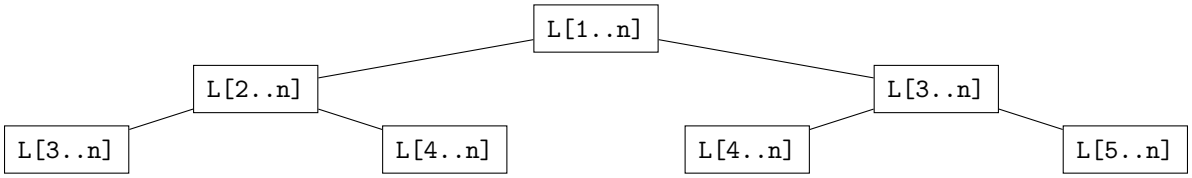
Só que F_n é uma quantidade exponencial em n.

O que significa que esse algoritmo é horrível.

Mas, ainda há esperança ...

Quer dizer, existe uma razão para o tempo de execução ser tão ruim assim.

E essa razão aparece quando a gente examina a árvore de recursão do algoritmo



Sim! o algoritmo está fazendo a mesma chamada recursiva múltiplas vezes.

Só que isso é um problema bem fácil de resolver.

Quer dizer, basta armazenar o resultado das chamadas recursivas em uma tabela auxiliar.

Daí, na hora em que a chamada está para ser realizada, a gente verifica se o seu resultado já está na tabela.

E se o resultado estiver lá, a chamada não é realizada.

Essa técnica é conhecida como a *programação dinâmica*.

E abaixo, nós temos a nova versão do algoritmo

```
Algoritmo SubconjMax-PD ( L, inicio, n )
 {
    Se ( inicio = n )   Retorna ( L[n] )

    // caso 1: L[inicio] é selecionado
    Se ( T[inicio+2] != Nulo )
        S1  <--  T[inicio+2]  +  L[inicio]
    Senão
        S1  <--  SubconjMax-DC ( L, inicio+2, n )  +  L[inicio]

    // caso 2: L[inicio] não é selecionado
    Se ( T[inicio+2] != Nulo )
        S2  <--  T[inicio+1]  +  L[inicio]
    Senão
        S2  <--  SubconjMax-DC ( L, inicio+1, n )  +  L[inicio]

    Retorna ( Max{ S1, S2 } )
 }
```

Análise

Legal.

Mas, quanto tempo leva isso ?

Bom, a primeira observação é que

1. Ignorando o tempo gasto com as chamadas recursivas, a função executa em tempo $O(1)$
   — (*i.e., esse é o tempo dentro da caixinha ...*)

E a segunda observação é a seguinte

2. Não ocorrem duas chamadas recursivas com o mesmo valor do parâmetro `início`
   — (*i.e., cada caixinha só executa uma vez*)

Com base nessas observações, é fácil ver que

$$T(n) \quad = \quad \#\ \mathsf{caixinhas} \quad \times \quad O(1)$$

Mas, nós temos uma chamada para cada valor do parâmetro `início`.

E o parâmetro `início` varia de 1 até n.

Daí que,

$$T(n) \quad = \quad O(n) \quad \times \quad O(1) \quad = \quad O(n)$$

Não é legal?

3.1 Mais programação dinâmica

Sim.

Mas agora, a gente pode tentar otimizar os dois primeiros algoritmos também.

Quer dizer, não é (muito) difícil ver que eles também realizam chamadas recursivas repetidas
— (*você consegue ver?*)

Daí que, a gente pode tentar reduzir o seu tempo de execução utilizando a técnica da programação dinâmica.

Abaixo, nós temos a versão adaptada do primeiro algoritmo

```
Algoritmo SubconjMax-DC ( L, inicio, fim )
 {
    Se ( fim < inicio + 2 )            // i.e., menos do que 3 elementos
     {
        Retorna ( Max{ L[inicio..fim] } )
     }

    meio  <--  (inicio + fim) / 2


    // caso 1: L[meio-1] é selecionado

    S1  <--   L[meio-1]

    Se ( T[inicio,meio-3] != Nulo )  S1  <--  S1 + T[inicio,meio-3]
    Senão                            S1  <--  S1 + SubconjMax-DC ( L, inicio, meio-3 )

    Se ( T[meio+1,fim] != Nulo   )   S1  <--  S1 + T[meio+1,fim]
```

```
    Senão                           S1  <--  S1 + SubconjMax-DC ( L, meio+1, fim )


    // caso 2: L[meio] é selecionado

    S2  <--   L[meio-1]

    Se ( T[inicio,meio-2] != Nulo )  S2  <--  S2 + T[inicio,meio-2]
    Senão                            S2  <--  S2 + SubconjMax-DC ( L, inicio, meio-3 )

    Se ( T[meio+2,fim] != Nulo   )   S2  <--  S2 + T[meio+2,fim]
    Senão                            S2  <--  S2 + SubconjMax-DC ( L, meio+2, fim )



    // caso 3: L[meio+1] é selecionado

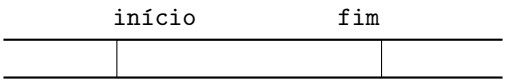
    S3  <--   L[meio-1]

    Se ( T[inicio,meio-1] != Nulo )  S3  <--  S3 + T[inicio,meio-1]
    Senão                            S3  <--  S3 + SubconjMax-DC ( L, inicio, meio-1 )

    Se ( T[meio+3,fim] != Nulo   )   S3  <--  S3 + T[meio+3,fim]
    Senão                            S3  <--  S3 + SubconjMax-DC ( L, meio+3, fim )


    Retorna ( Max{ S1, S2, S3 } )
}
```

Bom, a novidade aqui é que os subproblemas são definidos por uma faixa da lista.



Daí que, ocorre uma chamada recursiva para cada par de valores dos parâmetros `início` e `fim`.

E dessa vez, a tabela `T` é indexada por duas variáveis.

Por outro lado, não é difícil ver que

- o tempo dentro da caixinha é $O(1)$ — (*porque não há laços ...*)

Daí que, o tempo de execução do algoritmo pode ser estimado assim

$$T(n) \quad = \quad \#\,\text{caixinhas} \quad \times \quad O(1)$$

Só que, nós temos uma caixinha para cada par de valores dos parâmetros `início` e `fim`.

E cada um desses parâmetros varia de 1 a n.

Logo, o tempo de execução do algoritmo fica assim

$$T(n) \quad = \quad O(n^2) \quad \times \quad O(1) \quad = \quad O(n^2)$$

Quer dizer, o tempo de execução do algoritmo foi reduzido de $O(n^{2,58})$ para $O(n^2)$.

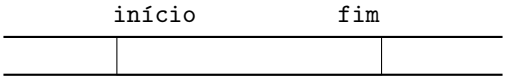
E isso nos deixa otimistas para otimizar o segundo algoritmo também.

Mas, pensando um pouquinho sobre o assunto, a gente vê que dessa vez isso não vai ser possível.

Quer dizer, o algoritmo já executa em tempo $O(n^2)$.

E ele também quebra a lista no meio.

Daí que, os subproblemas também tem a forma



E será preciso fazer uma chamada recursiva para cada par de valores dos parâmetros `início` e `fim`.

Só isso já leva tempo $O(n^2)$ — (*mesmo eliminando as chamadas recursivas repetidas*)

Logo, a técnica da programação dinâmica não nos dá um ganho significativo nesse caso.

Nota: *Devido ao padrão de quebra do problema, essa análise talvez possa ser melhorada ...*

3.2 Um algoritmo iterativo

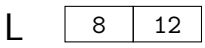
Agora nós vamos considerar ainda uma outra forma de pensar sobre o problema.

Quer dizer, imagine que a lista contém apenas um elemento



Então, é claro que a solução consiste em selecionar esse elemento para o subconjunto.

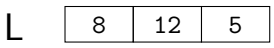
Agora, imagine que nós adicionamos outro elemento à lista



Bom, como esse elemento é maior do que o elemento que já estava lá, ele passa a ser a solução do problema.

Certo.

Agora a gente adiciona mais um



Dessa vez, o novo elemento não é maior do que o anterior.

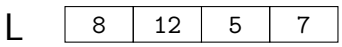
Mas, juntamente com o primeiro elemento ele produz uma soma maior.

Então, a nova solução é: $\{8, 5\}$.

E então, já deu para perceber qual é o padrão?

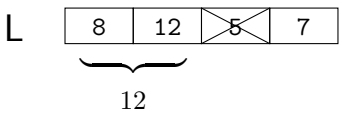
Ainda não?

Então, a gente vai adicionar mais um elemento

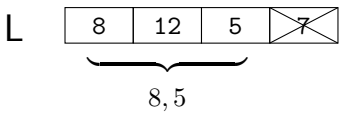


E daí, a gente raciocina assim

- `Das duas uma:  ou o 7 vai fazer parte da solução ou ele não vai.`
- `Se ele faz parte da solução, então o 5 não vai poder entrar.`
  `E daí, a gente considera a solução ótima do restante da lista`



- `Se o 7 não faz parte da solução, então nós podemos considerar a solução ótima`
  `do restante da lista`

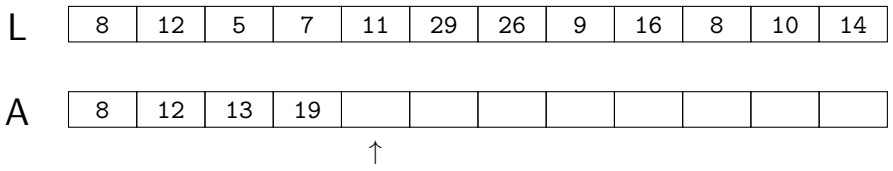


Sim! é o mesmo raciocínio outra vez.

Só que dessa vez, a gente tem as soluções dos subproblemas, e pode escolher a melhor opção.

Agora, a gente pode transformar essa ideia em um algoritmo iterativo para o problema.

Para fazer isso, a gente utiliza uma lista auxiliar



A ideia é que para preencher a próxima posição da lista auxiliar, a gente considera as duas possibilidades para o elemento 11

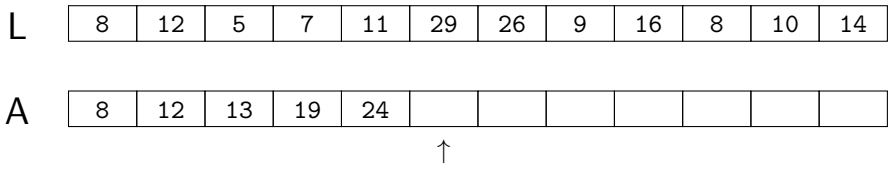
- Se o 11 faz parte da solução, então o 7 não pode fazer.
  E daí, a gente considera a solução ótima para o restante da lista

$$13 \quad + \quad 11 \quad = \quad 24$$

- Se o 11 não faz parte da solução, então a gente considera a solução ótima até o elemento anterior

19

Como a primeira opção é melhor, a gente preenche a lista auxiliar assim



Agora, é só continuar fazendo a mesma coisa.

No final, a última posição da lista auxiliar nos dá o valor da solução ótima para o problema

```
Algoritmo SSMax-It ( L )
 {

    // Inicialização

       A  <--  lista auxiliar de tamanho N

    A[1]  <--  L[1]

    A[2]  <--  Max { L[1], L[2] }


    // Preenchimento da lista

    Para  i <-- 3  Até  N
     {
           sa  <--  L[i]  +  A[i-2]

           sb  <--  A[i-1]

         A[i]  <--  Max { sa, sb }
     }

    Retorna ( A[N] )
 }
```