ACH2002 – IAA

Aula 11
Técnicas de programação baseadas em indução - Programação Dinâmica (indução forte)

(baseada nos slides de aula do Prof. Marcos L. Caim)

Aulas passadas

- Aula 3: prova de corretude de algoritmos iterativos (usando **indução matemática (fraca)** e invariante)
- Aula 6: recursão - que tem tudo a ver com **indução matemática** também! Não só para a prova de corretude mas para a própria concepção do algoritmo – ex: fatorial – mesmo que implementado de forma iterativa!

Fatorial

$$F(n) = \begin{cases} 1 & \text{, se } n = 0 \\ n * F(n-1), & \text{se } n > 0 \end{cases}$$

```
fatorial (n)
 se n < 2
      retorna 1
 senão
      retorna n*fatorial (n-1)
 fim se
```

Provando que o algoritmo (recursivo) está **correto**:

(com base em seus retornos)

- Base da indução:

  Se n=0, a função retorna 1 (o valor correto)

- Passo da indução
  - Assumindo que a função retorne o valor correto (n!) para n > 0
  - fatorial(n+1) irá retornar (n+1)*fatorial(n) = (n+1)*n! = (n+1)!
  - Logo, fatorial(n+1) retorna o valor **correto**!

```
int fatorial (int n)
{
  if (n == 0)
    return 1;
  else
    return n*fatorial (n-1);
}
```

- Algoritmo iterativo:

```
fatorial (n)
 fat = 1
 para i = 2 até n
         fat = fat * i
 fim para
 retorna fat
```

Aulas passadas

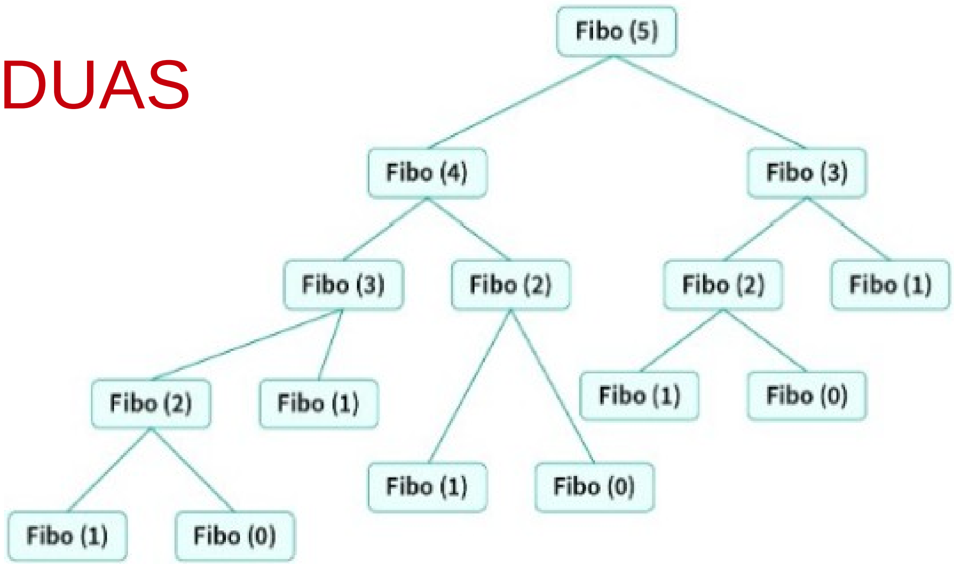
- Aula 3: prova de corretude de algoritmos iterativos (usando **indução matemática (fraca)** e invariante)
- Aula 6: recursão - que tem tudo a ver com **indução matemática** também! Não só para a prova de corretude mas para a própria concepção do algoritmo – ex: fatorial – mesmo que implementado de forma iterativa!
  - Fibonacci também, mas como os **subproblemas não são disjuntos, há muito retrabalho** em fazer as chamadas recursivas (exponencial no tempo) – melhor fazer iterativo ARMAZENANDO o que já foi calculado e precisará ser usado (linear no tempo)

Fibonacci

$$\begin{cases} f_0 = 0, f_1 = 1, \\ f_n = f_{n-1} + f_{n-2}, n \geq 2 \end{cases}$$

```
fibonacci(n)
 se n < 2
     retorna n
 senão
     retorna fibonacci(n-1) + fibonacci(n-2)
 fim se
```

Note que aqui precisamos definir DUAS bases de indução!!!! (n=0 e n=1)




```
int fibonacci (int n)
{
  int fib, fib_1, fib_2; /* fib_1 significa 'fib-1' */
  if (n < 2)
    return n;
  else
  {
    fib_2 = 0;
    fib_1 = 1;
    for (int i = 2; i <= n; i++)
    {
      /* i-esimo elemento da série sendo calculado */
      /* INVARIANTE: */
      /* fib_1 tem o elemento anterior, e fib_2 tem o anterior do anterior */
      fib = fib_1 + fib_2;
      fib_2 = fib_1;
      fib_1 = fib;
    }
    return fib;
  }
}
```

Aulas passadas

- Aula 3: prova de corretude de algoritmos iterativos (usando **indução matemática (fraca)** e invariante)
- Aula 6: recursão - que tem tudo a ver com **indução matemática** também! Não só para a prova de corretude mas para a própria concepção do algoritmo – ex: fatorial – mesmo que implementado de forma iterativa!
  - Fibonacci também, mas como os **subproblemas não são disjuntos, há muito retrabalho** em fazer as chamadas recursivas (exponencial no tempo) – melhor fazer iterativo ARMAZENANDO o que já foi calculado e precisará ser usado (linear no tempo)
- Aula 7: divisão e conquista – que tem tudo a ver com **indução matemática** também! (corretude e concepção) – ex: mergesort – mesmo que implementado de forma iterativa!
  - Útil quando os subproblemas são **disjuntos**

Divisão e conquista

```
mergeSort (A,i,f)
se i < f
        m ← ⌊(i+f)/2⌋
        mergeSort(A, i, m)
        mergeSort(A, m+1,f)
        merge(A,i,m,f)
```

- Três passos em cada nível de recursão:
  - Dividir – o problema em um determinado número de subproblemas **DISJUNTOS**.
  - Conquistar – os subproblemas, resolvendo-os recursivamente. Se os tamanhos dos subproblemas forem pequenos o suficiente, resolvê-los diretamente.
  - Combinar – as soluções dadas aos subproblemas para formar solução procurada para problema original.

Aulas passadas

- Aula 3: prova de corretude de algoritmos iterativos (usando **indução matemática (fraca)** e invariante)
- Aula 6: recursão - que tem tudo a ver com **indução matemática** também! Não só para a prova de corretude mas para a própria concepção do algoritmo – ex: fatorial – mesmo que implementado de forma iterativa!
  - Fibonacci também, mas como os **subproblemas não são disjuntos, há muito retrabalho** em fazer as chamadas recursivas (exponencial no tempo) – melhor fazer iterativo ARMAZENANDO o que já foi calculado e precisará ser usado (linear no tempo)
- Aula 7: divisão e conquista – que tem tudo a ver com **indução matemática (fraca)** também! (corretude e concepção) – ex: mergesort – mesmo que implementado de forma iterativa!
  - Útil quando os subproblemas são **disjuntos**
- Aulas 8 e 10: dado um algoritmo cuja “alma é recursiva”, podemos descrever sua complexidade de tempo como uma recorrência
  - Vimos como calcular a complexidade assintótica a partir da recorrência (expansão, substituição e mestre)
  - Também usamos **indução matemática (fraca)**

Aulas passadas

- Aula 3: prova de corretude de algoritmos iterativos (usando **indução matemática (fraca)** e invariante)
- Aula 6: recursão - que tem tudo a ver com **indução matemática** também! Não só para a prova de corretude mas para a própria concepção do algoritmo – ex: fatorial – mesmo que implementado de forma iterativa!
  - Fibonacci também, mas como os **subproblemas não são disjuntos, há muito retrabalho** em fazer as chamadas recursivas (exponencial no tempo) – melhor fazer iterativo ARMAZENANDO o que já foi calculado e precisará ser usado (linear no tempo)
- Aula 7: divisão e conquista – que tem tudo a ver com **indução matemática (fraca)** também! (corretude e concepção) – ex: mergesort – mesmo que implementado de forma iterativa!
  - Útil quando os subproblemas são **disjuntos**
- Aulas
descre
  -
  -

Quando a soma dos tamanhos dos subproblemas é aprox. n (problemas disjuntos), é provável que a complexidade da solução recursiva seja **polinomial**.

Quando a divisão de um problema de tamanho n resulta em mais de um subproblema de tamanho n-1 (subproblemas não disjuntos), é provável que a complexidade da solução recursiva seja **exponencial**... Então algo melhor deveria ser feito!

Conclusão e *spoiler* da aula de hoje

- **Indução matemática (fraca)** é uma ferramenta para criar a solução de um problema (algoritmo), que de quebra já vem com a prova de corretude (se a implementação for recursiva); mas pode-se fazer também uma implementação iterativa
- A indução matemática (que é a que vimos na aula 3 e usamos até agora) é chamada **indução fraca**
- Dependendo do problema, a solução exige que a hipótese de indução seja incrementada (veremos isso hoje)
- Para certos problemas a indução fraca não é suficiente ou remete a solução ineficientes
  - Para esses problemas é melhor usar **indução forte**
  - Que tem tudo a ver com a técnica de programação dinâmica

Conclusão e *spoiler* da aula de hoje

- **Indução matemática (fraca)** é uma ferramenta para criar a solução de um problema (algoritmo), que de quebra já vem com a prova de corretude (se a implementação for recursiva); mas pode-se fazer também uma implementação iterativa
- A indução matemática (que é a que vimos na aula 3 e usamos até agora) é chamada **indução fraca**
- Dependendo do problema, a solução exige que a hipótese de indução seja incrementada (veremos isso hoje)
- Para certos problemas a indução fraca não é suficiente ou remete a solução ineficientes
  - Para esses problemas é melhor usar **indução forte**
  - Que tem tudo a ver com a técnica de programação dinâmica

Subsequência consecutiva máxima

Problema:

- Dada uma sequência $x_1, x_2, \ldots, x_n$ de número reais (não necessariamente positivos) encontre uma subsequência $x_i, x_{i+1}, \ldots, x_j$ (de elementos consecutivos) tal que a soma dos números nela seja máxima em relação a todas subsequências de elementos consecutivos.
- Nós chamamos tal subsequência de **subsequência máxima**.
- Exemplo: 2, -3, 1.5, -1, 3, -2, -3, 3
- A subsequência máxima é (1.5, -1, 3). A soma é 3,5.
- Pode haver mais de uma subsequência máxima em uma da sequência.
- Se todos os número são negativos, então a subsequência é vazia (por definição a soma de uma subsequência vazia é zero).

Subsequência consecutiva máxima (SCM)

- Seria interessante um algoritmo que resolvesse o problema lendo a sequência em ordem uma única vez (ou seja, não queremos primeiro ver se apenas há números negativos).
- Base:

Subsequência consecutiva máxima (SCM)

- Seria interessante um algoritmo que resolvesse o problema lendo a sequência em ordem uma única vez (ou seja, não queremos primeiro ver se apenas há números negativos).
- Base: n = 1

SCM = é próprio número,   se ele for positivo
vazia, c.c

Subsequência consecutiva máxima (SCM)

- Seria interessante um algoritmo que resolvesse o problema lendo a sequência em ordem uma única vez (ou seja, não queremos primeiro ver se apenas há números negativos).
- Base: n = 1

SCM = { é próprio número, se ele for positivo
vazia, c.c

- Hipótese da indução:

Subsequência consecutiva máxima (SCM)

- Seria interessante um algoritmo que resolvesse o problema lendo a sequência em ordem uma única vez (ou seja, não queremos primeiro ver se apenas há números negativos).
- Base: n = 1

SCM = { é próprio número,   se ele for positivo
vazia, c.c

- Hipótese da indução: sabemos calcular a SCM para uma sequência de tamanho n-1.

Subsequência consecutiva máxima (SCM)

- Passo da indução:

Considere a subsequência $S = \{x_1, x_2, \ldots, x_n\}$ onde $n > 1$.

- Pela hipótese de indução, sabe-se como determinar a SCM de $S' = \{x_1, x_2, \ldots, x_{n-1}\}$.
- Se a SCM de S' é vazia, todos os números de S' são negativos, logo, apenas x_n precisa ser considerado.
- Assume-se então que a SCM de S' é $SCM(S') = \{x_i, x_{i+1}, \ldots, x_j\}$ sendo $1 \leq i \leq j \leq n-1$.
- Se $j = n-1$ (isto é, a SCM(S') é um sulfixo de S'), então é fácil estender a solução para S:
  - Se x_n é positivo, então ele estende a SCM(S') → SCM(S) = SCM(S') mais x_n.
  - Caso contrário, SCM(S) = SCM(S')
- Mas se $j < n - 1$, então há duas possibilidades:
  - SCM(S) = SCM (S'), ie, continua sendo a subsequência consecutiva máxima
  - ou existe uma outra subsequência, que não é máxima para S', mas é para S, quando x_n é adicionada a ela

Subsequência consecutiva máxima (SCM)

- Passo da indução (cont):
- A solução para esse problema é fortalecer a hipótese de indução.
- O problema com a hipótese de indução trivial é que x_n pode estender uma subsequência que não é máxima para S' e mesmo assim criar uma subsequência máxima consecutiva para S.
- Portanto, saber apenas a subsequência máxima consecutiva de S' não é suficiente.
- Suponha que a hipótese de indução possa ser fortalecida para incluir o conhecimento sobre o sufixo de valor máximo, denotado por SuffixMax = $(x_k, x_{k+1}, \ldots, x_{n-1})$.

Subsequência consecutiva máxima (SCM)

Hipótese de indução Fortalecida: Nós sabemos como encontrar, em sequências de tamanho < n, a subsequência máxima no geral e também a máxima subsequência que é um sufixo.

- O algoritmo agora fica mais fácil:

- Adiciona-se x_n ao sufixo máximo.

  - Se a soma é maior do que a SCM(S') (que é geral), então tem-se uma nova subsequência geral máxima (SCM(S) = SCM(S')), e também um novo sufixo máximo.
  - Caso contrário, mantém-se a subsequência geral máxima atual.

- No entanto, o algoritmo não está terminado. Precisa-se calcular o novo sufixo máximo. Pode ser que x_n seja negativo e o sufixo máximo fique negativo. Nesse caso, é melhor utilizar como novo sufixo máximo a sequência vazia (com soma zero) como novo sufixo máximo.

Subsequência consecutiva máxima (SCM)

```
Input: X (um vetor de tamanho n)
Output: GlobalMax (a soma da subsquência máxima)
GlobalMax := 0;
SufixMax := 0;
for i = 0 to n do
    if x[i]+SufixMax > GlobalMax then
        SufixMax := SufixMax + x[i];
        GlobalMax := SufixMax;
    else
        if x[i]+SufixMax > 0 then
            SufixMax := SufixMax + x[i];
        else
            SufixMax := 0;
return GlobalMax
```

Observações

- Seja P(n) o problema de calcular uma subsequência máxima para um vetor v de tamanho n.
- Por que o problema de solucionar P(n) requer fortalecer a indução?

  - Indução fraca: sabemos solucionar P(< n) cujo resultado é uma SCM(S')

  - Porém, saber como determinar SCM(S') não é suficiente para resolver P(n).
- O truque é fortalecer a hipótese de indução:

  $[P \text{ and } Q](< n) \Rightarrow [P \text{ and } Q](n)$

  Q(n): problema de calcular a subsequência máxima que é sulfixo.
- Fortalecer a hipótese de indução requer alguns cuidados:

  - A cada passo do algoritmo temos calcular a subsequência máxima e a subsequência máxima que é sufixo.

  - Erro comum: esquecer de calcular Q(< n).

Complexidade

1) O algoritmo lê o vetor apenas uma vez e em ordem?

2) Qual a complexidade do algoritmo?

Complexidade

1) O algoritmo lê o vetor apenas uma vez e em ordem?

SIM!

2) Qual a complexidade do algoritmo?

O(n)

Exercícios

1) É possível escrever o algoritmo acima utilizando recursão? Se sim, escreva um algoritmo recursivo para o problema de determinar o valor das subsequências máximas (lembrando que elas não necessariamente são únicas).

2) Modifique o algoritmo para não só calcular o valor da SCM (GlobalMax) mas também os índices i e j (início e fim) que a definem.

Conclusão e *spoiler* da aula de hoje

- **Indução matemática (fraca)** é uma ferramenta para criar a solução de um problema (algoritmo), que de quebra já vem com a prova de corretude (se a implementação for recursiva); mas pode-se fazer também uma implementação iterativa
- A indução matemática (que é a que vimos na aula 3 e usamos até agora) é chamada **indução fraca**
- Dependendo do problema, a solução exige que a hipótese de indução seja incrementada (veremos isso hoje)
- Para certos problemas a indução fraca não é suficiente ou remete a solução ineficientes
  - Para esses problemas é melhor usar **indução forte**
  - Que tem tudo a ver com a técnica de programação dinâmica

Problema da mochila

Um problema de otimização combinatória (base do algoritmo de criptografia por chaves públicas)

- Suponha que você tenha recebido uma mochila e quer enchê-la completamente com determinados itens.
- Pode haver itens com diferentes tamanhos e formatos. Nosso objetivo é deixar a mochila o mais cheia possível.
- A mochila pode ser um caminhão, um navio ou um chip de silício e o problema é empacotar os itens.
- Há muitas variantes desse problema; porém, nós vamos tratar apenas de um problema simples com itens de uma única dimensão.

O problema

- Dado um inteiro K ("tamanho" da mochila) e n itens de diferentes "tamanhos" tais que o i-ésimo item possui um tamanho k_i, encontre o subconjunto de itens cujos tamanhos somam exatamente K ou determine que esse subconjunto não existe.
- O problema é representado por P(n, K), tal que n denota o número de itens e K denota o "tamanho" da mochila.
- Nós iremos implicitamente assumir que os n itens são aqueles que foram dados como entrada do problema e não iremos incluir os seus tamanhos na notação do problema.
- Assim, P(i, k) denota o problema de inserção dos primeiros i itens e a mochila de tamanho k ($k \leq K$).
- Inicialmente, nós nos concentraremos apenas no problema de decisão, que é determinar se uma solução existe.

A solução (primeira tentativa)

Hipótese de indução: Nós sabemos como resolver P(n-1, K).

- O caso base é fácil (n = 1): há uma solução apenas se o elemento é do tamanho K.
- Passo da indução:

- Se há uma solução para P(n-1, K) – isto é, se há um jeito de empacotar alguns dos n-1 na mochila – então estamos feitos. Basta não usar o n-ésimo item.

- Suponha, porém, que não há solução para P(n-1, K). Podemos usar esse resultado negativo?
    - Sim. Ele significa que o n-ésimo elemento pode ser incluído (se couber).
    - Neste caso, o resto dos itens devem caber em uma mochila menor de tamanho $K\text{-}k_n$.

- Nós reduzimos o problema em dois subproblemas melhores: P(n-1, K) e P(n-1, $K–k_n$).

- Nós precisamos resolver o problema não apenas para as mochilas de tamanho K, mas também para as mochilas de todos os tamanhos de no máximo K. Ou seja, precisamos fortalecer a hipótese.

A solução (segunda tentativa)

Hipótese de indução: Nós sabemos como resolver P(n-1, k) para todo $0 \leq k \leq K$.

- O caso base P(1, k) é fácil: se $k = 0$, então sempre há uma solução trivial (não há como encher a mochila)
- Passo da indução: Caso contrário, há uma solução que consiste do primeiro item igual a k.
- Agora nós reduzimos P(n, k) em dois problemas P(n-1, k) e P(n-1, $k-k_n$). Se $k-k_n < 0$ então ignoramos o segundo problema

A solução (segunda tentativa)

- Ambos os problemas podem ser resolvidos por indução fraca.
- É uma redução válida e isso nos dá um algoritmo, embora ineficiente.
- Nós reduzimos um problema de tamanho n para dois problemas de tamanho n-1. Nós também reduzimos o valor de k em um subproblema.
- Cada um desses dois subproblemas podem ainda ser reduzidos para outros dois subproblemas levando a um algoritmo exponencial.
- Felizmente, é possível em muitos casos melhorar o tempo de execução para esses tipos de problemas.
- A restrição é que o número de possíveis problemas não ser muito grande.

A solução (terceira tentativa)

- Seja P(i, k) a representação do problema de colocar i elementos em uma mochila de tamanho k.
- Se existe n possíveis valores para i e K valores para k então existem nK problemas diferentes!
- Portanto, muitos dos problemas P(i,k) que ocorreriam em um algoritmo exponencial se repetem (lembrem-se da árvore de Fibonacci).
- A solução é salvar todas as soluções e nunca resolver um problema duas vezes.
- A solução baseia-se em:

  - Indução forte, isto é, assumir que todas as soluções para os casos menores, e não apenas para n-1, são conhecidas.

A solução (terceira tentativa)

- Nós salvamos todos os resultados conhecidos em uma matriz $n_x K$.
- O elemento (i, k) da matriz contém a solução para P(i, k).
- Cada elemento (i, k) da matriz contém dois booleanos: *belongs* e *exists*.

- P[i,k].exists vai indicar se existe um subconjunto dos i primeiros números que preenche completamente uma mochila de tamanho k (no caso, k minúsculo).

- Já P[i,k].belongs indica se o i-ésimo elemento faz parte do subconjunto solução.

Solução

Input: S — um arranjo de tamanho n que armazena os tamanhos dos itens, e K

Output: P — uma matriz tal que P[i,j].exist = true, se existe uma solução para o problema da mochila com os primeiros i elementos e uma mochila de tamanho k; e P[i,j].belong = true, se o i-ésimo elemento pertence à solução.

```
P[0,0].exist = true;
for k:=1 to K do
    P[0,k].exist = false;
for i:=1 to n do
    for k:=0 to K do
        P[i,k].exist = false;
        if P[i-1,k].exist then
            P[i,k].exist = true;
            P[i,k].belong = false;
        else
            if k-S[i]≥0 then     /* se couber, ou seja, se k ≥ S[i] */
                if P[i-1,k-S[i]].exist then
                    P[i,k].exist = true;
                    P[i,k].belong = true;
return P
```

Exemplo

- Considere uma entrada de quatro itens de tamanhos iguais a 2, 3, 5 e 6, e uma mochila de tamanho 9.
- O algoritmo tem como entrada um arranjo S com os quatro (n = 4) itens que podem ser inseridos na mochila. É introduzido o item $i_0 = 0$ para representar um item de tamanho zero. A tabela abaixo representa o arranjo S de entrada.

S	i_1	i_2	i_3	i_4
Valor	2	3	5	6

Exemplo

S	i_1	i_2	i_3	i_4
Valor	2	3	5	6

```
P[0,0].exist = true;
for k:=1 to K do
    P[0,k].exist = false;
for i:=1 to n do
    for k:=0 to K do
        P[i,k].exist = false;
        if P[i-1,k].exist then
            P[i,k].exist = true;
            P[i,k].belong = false;
        else
            if k-S[i]≥0 then
                if P[i-1,k-S[i]].exist then
                    P[i,k].exist = true;
                    P[i,k].belong = true;
return P
```

Seguindo o algoritmo obtemos a matriz P a seguir. Nesta matriz, cada célula possui dois valores booleanos separados por uma barra invertida. Os valores booleanos referem-se, respectivamente, a P[i,k].exists e a P[i,k].belong.

P	0	1	2	3	4	5	6	7	8	9
$i_0 = 0$	T/	F/	F/	F/	F/	F/	F/	F/	F/	F/
$i_1 = 2$	T/F	F/	T/T	F/	F/	F/	F/	F/	F/	F/
$i_2 = 3$	T/F	F/	T/F	T/T	F/	T/T	F/	F/	F/	F/
$i_3 = 5$	T/F	F/	T/F	T/F	T/T	T/F	T/T	T/T	F/	T/T
$i_4 = 6$	T/F	F/	T/F	T/F	T/F	T/F	T/F	T/F	T/T	T/F

Exemplo

- No exemplo, P[4,7].exists é verdadeiro (true), mas o quarto elemento não faz parte do subconjunto pois P[4,7].belongs é falso (false).

- Isto porque é possível encontrar a solução com um subconjunto de somente os três primeiros itens (k_1 = 2 e k_3 = 5) uma vez que P[3,7].exists e P[3,7].belongs são verdadeiros. Dessa maneira, o algoritmo deixa de procurar por outras soluções.

P	0	1	2	3	4	5	6	7	8	9
$i_0 = 0$	T/	F/	F/	F/	F/	F/	F/	F/	F/	F/
$i_1 = 2$	T/F	F/	T/T	F/	F/	F/	F/	F/	F/	F/
$i_2 = 3$	T/F	F/	T/F	T/T	F/	T/T	F/	F/	F/	F/
$i_3 = 5$	T/F	F/	T/F	T/F	T/T	T/F	T/T	T/T	F/	T/T
$i_4 = 6$	T/F	F/	T/F	T/F	T/F	T/F	T/F	T/F	T/T	T/F

Complexidade?

```
P[0,0].exist = true;
for k:=1 to K do
    P[0,k].exist = false;
for i:=1 to n do
    for k:=0 to K do
        P[i,k].exist = false;
        if P[i-1,k].exist then
            P[i,k].exist = true;
            P[i,k].belong = false;
        else
            if k-S[i]≥0 then
                if P[i-1,k-S[i]].exist then
                    P[i,k].exist = true;
                    P[i,k].belong = true;
return P
```

P	0	1	2	3	4	5	6	7	8	9
$i_0 = 0$	T/	F/	F/	F/	F/	F/	F/	F/	F/	F/
$i_1 = 2$	T/F	F/	T/T	F/	F/	F/	F/	F/	F/	F/
$i_2 = 3$	T/F	F/	T/F	T/T	F/	T/T	F/	F/	F/	F/
$i_3 = 5$	T/F	F/	T/F	T/F	T/T	T/F	T/T	T/T	F/	T/T
$i_4 = 6$	T/F	F/	T/F	T/F	T/F	T/F	T/F	T/F	T/T	T/F

Complexidade:

O(nK)

```
P[0,0].exist = true;
for k:=1 to K do
    P[0,k].exist = false;
for i:=1 to n do
    for k:=0 to K do
        P[i,k].exist = false;
        if P[i-1,k].exist then
            P[i,k].exist = true;
            P[i,k].belong = false;
        else
            if k-S[i]≥0 then
                if P[i-1,k-S[i]].exist then
                    P[i,k].exist = true;
                    P[i,k].belong = true;
return P
```

P	0	1	2	3	4	5	6	7	8	9
$i_0 = 0$	T/	F/	F/	F/	F/	F/	F/	F/	F/	F/
$i_1 = 2$	T/F	F/	T/T	F/	F/	F/	F/	F/	F/	F/
$i_2 = 3$	T/F	F/	T/F	T/T	F/	T/T	F/	F/	F/	F/
$i_3 = 5$	T/F	F/	T/F	T/F	T/T	T/F	T/T	T/T	F/	T/T
$i_4 = 6$	T/F	F/	T/F	T/F	T/F	T/F	T/F	T/F	T/T	T/F

Conclusões

- Todos os problemas resolvidos por indução requer que saibamos resolver o caso base.
- Indução fraca requer que saibamos a solução para P(< n).

  - O problema da subsequência consecutiva máxima é resolvido por meio do fortalecimento da hipótese de indução (ainda fraca).

  - Uma condição Q(n) é adicionada; no caso, a subsequência máxima que é um sufixo.
- O problema da mochila é resolvido por meio de **indução forte**.

  - **Todas** as soluções para os casos menores, isto é, P(< n), e não apenas para P(n-1), são conhecidas.

Conclusões

- A solução apresentada para o problema da mochila é conhecido **programação dinâmica**.
- A essência da programação dinâmica é construir tabelas grandes com os resultados anteriores conhecidos.
- As tabelas são construídas iterativamente.
- Cada entrada da tabela (i, k) é construída a partir da combinação de outras entradas acima ou à esquerda de (i,k).
- O maior problema é construir a matriz da maneira mais eficiente.

Fibonacci

- A solução iterativa de Fibonacci pode ser vista como um caso bem simples de programação dinâmica (que você na verdade só precisa saber/armazenar para n-1 e n-2)

Exercício

- Escreva um algoritmo para encontrar a solução do problema da mochila a partir da matriz gerada pelo algoritmo fornecido.

Um outro exemplo

- Alinhamento de sequências:

  https://www.youtube.com/watch?v=F-8FS295gM8

Referências

- Udi Manber. Introduction to Algorithms: A Creative Approach. Addison-Wesley Professional, 1a. ed., 1989 (páginas 106-111).