Universidade Federal de Viçosa
Campus Rio Paranaíba
Instituto de Ciências Exatas e Tecnológicas

SIN 343
Desafios de Programação

João Batista Ribeiro

joao42lbatista@gmail.com

Slides baseados no material do prof. Guilherme C. Pena

Universidade Federal de Viçosa
Campus Rio Paranaíba
Instituto de Ciências Exatas e Tecnológicas

Aula de Hoje

Projeto de Algoritmos

Backtracking

Backtracking:

É um método para gerar/enumerar/percorrer sistematicamente todas as alternativas de um espaço de soluções.

Pode ser considerado uma versão melhorada da busca exaustiva.

Backtracking

Backtracking:
O método constrói soluções candidatas incrementalmente e abandona (***backtrack***) uma solução parcial quando identifica que tal solução parcial não levará a uma solução do problema.

Em cada passo do processo de enumeração são tentadas todas as possíveis alternativas recursivamente.

É um método de tentativa e erro: tenta um caminho; se não der certo, volta e tenta outro.

Backtracking

Backtracking (Algumas Aplicações):

- Quais são todos os subconjuntos de {2, 5, 8, 9, 13} ?
- Em quantos destes a soma dos elementos é ≤ 17 ?

- Quais são todas as permutações de [2, 5, 8, 9, 13] ?
- Em quantas destas os itens 8 e 9 não aparecem juntos ?

- Dado um labirinto, como achar um caminho para sair dele ?

- Como colocar 8 rainhas num tabuleiro de xadrez sem que nenhuma ataque outra?

- Sudoku!

Backtracking

Problema: Gerar todas as permutações.

```
void geraPerm(int a[], int k, int n){
    if(k == n){ // se terminou de gerar a permutação
        imprimePerm(a, n);
        return;
    }
    for(int i = 1; i <= n; i++) // para todo elemento i
        if(!pertence(a, k, i)){ // i não está na permutação parcial
            a[k] = i; // inclui o elemento na permutação
            geraPerm(a, k+1, n); // gera o restante da permutação
        }
}
```

Ver código: **geraPerm.cpp**

Backtracking

Problema: Gerar todos os subconjuntos.

```
void geraSub(bool a[], int k, int n){
   if(k == n){ // se terminou de gerar um subconjunto
      imprimeSub(a, n);
      return;
   }
   a[k] = true; // subconjuntos com o elemento k
   geraSub(a, k+1, n);
   a[k] = false;
   geraSub(a, k+1, n); // subconjuntos sem o elemento k
}
```

Ver código: **geraSub.cpp**

Backtracking

Algoritmo Geral:

```
// int a[] - solução parcial, de a[0] até a[k-1]
// int k - posição para inserir o próximo passo
// int n - tamanho máximo da solução
void backtrack(int a[], int k, int n){
   if ( ## terminou ##) { // se chegou numa possível solução
      ## processa solucao ##;
      return;
   }
   for ( ## toda alternativa ## ) // para toda alternativa
      if ( ## alternativa viável ## ) { // se pode incluí-la
         a[k] = ## alternativa ##; // inclui na solução parcial
         backtrack(a, k+1, n); // gera o restante
   }
}
```

Backtracking

Algoritmo Geral: É importante cortar a busca tão logo se descubra que não levará a uma solução

```
// int k - posição para inserir o próximo passo
// int a[] - solução parcial, de a[0] até a[k-1]
// int n - tamanho máximo da solução
void backtrack(int a[], int k, int n){
   if ( ## não há como continuar ##) // corte
      return;
   if ( ## terminou ##) { // se chegou numa possível solução
      ## processa solucao ##;
      return;
   }
   for ( ## toda alternativa ## ) // para toda alternativa
      if ( ## alternativa viável ## ) { // se pode incluí-la
         a[k] = ## alternativa ##; //inclui na solução parcial
         backtrack(a, k+1, n); // gera o restante
   }
}
```

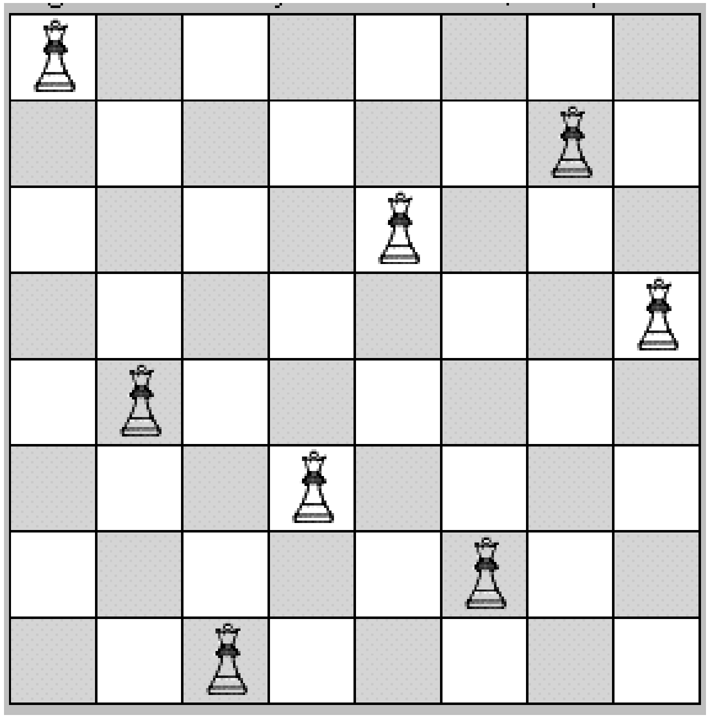
Backtracking

- Conceitualmente, é como uma árvore de busca
- As soluções candidatas são nós da árvore
- Cada nó é pai das soluções parciais com um passo a mais
- As folhas são soluções que não podem mais crescer

Backtracking (Exemplo)

Problema das 8 rainhas:

Dado um tabuleiro de xadrez (com 8x8 casas), o objetivo é distribuir 8 rainhas sobre este tabuleiro de modo que nenhuma delas fique em posição de ser atacada por outra rainha.



Backtracking (Exemplo)

Problema das 8 rainhas:

1ª ideia: (Gerar todas) p = 64 * 63 * ... * 58 * 57 =

$1{,}784629876 \times 10^{14}$ possibilidades

2ª ideia: (Supondo uma rainha por linha) p = 8^8 =

16.777.216 possibilidades

3ª ideia: (Uma rainha por linha e por coluna) p = 8! =

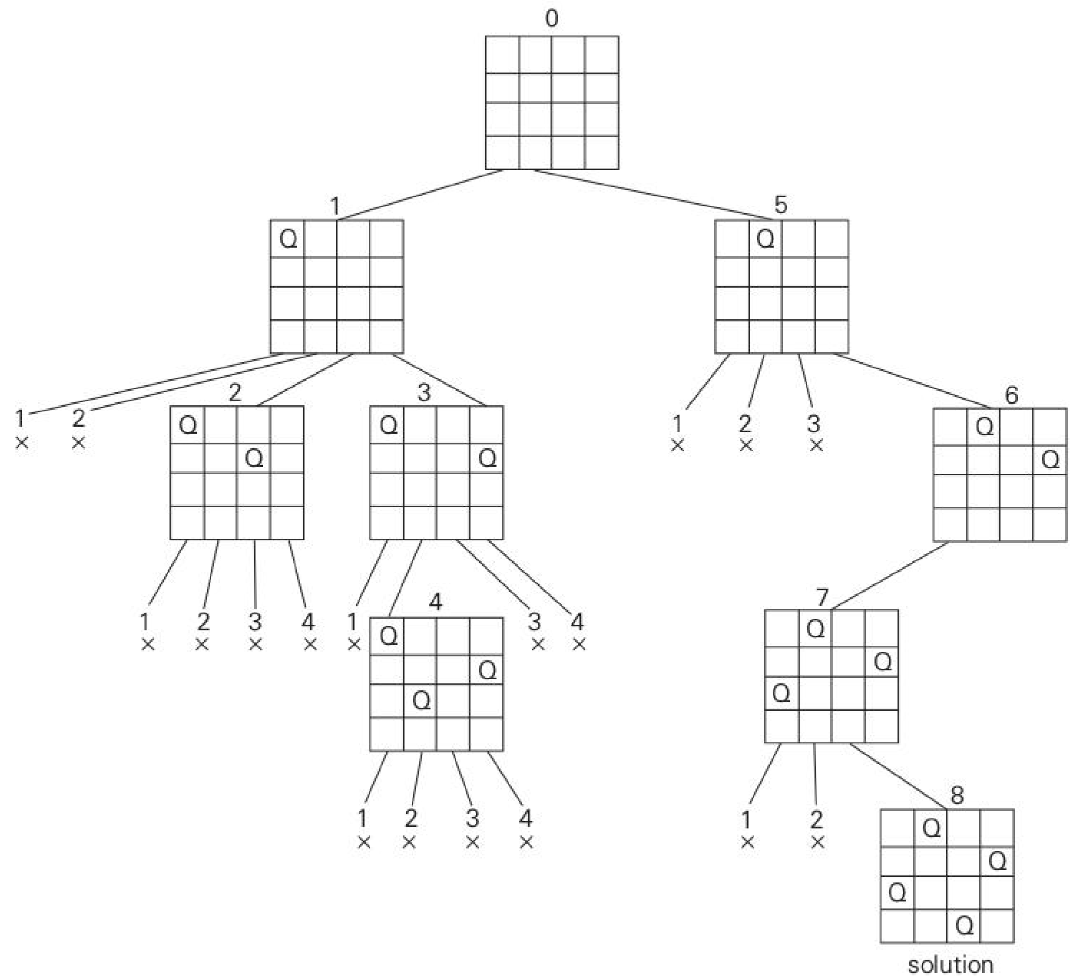
40320 possibilidades

Utilizando ***backtracking*** pode-se evitar o teste de diversas configurações que não levam a uma solução

Backtracking (Exemplo)

Problema das 8 rainhas:

Supondo 4 rainhas num tabuleiro 4x4.



Backtracking (Exemplo)

Problema das 8 rainhas:

A solução pode ser representada por um vetor Q (de inteiros) com 8 posições, sendo que:

$$\mathbf{Q[i] = j,}$$

indica que há uma rainha na posição (i, j) do tabuleiro

Essa abordagem já evita o teste de uma mesma linha. Ao inserir uma nova rainha, verifica-se as colunas e as diagonais:

Duas rainhas nas posições (l_1, c_1) e (l_2, c_2) estão numa mesma diagonal se e somente se $| l_1 - l_2 | = | c_1 - c_2 |$

Backtracking (Exemplo)

Problema das 8 rainhas:

Ver arquivo: **nRainhas.cpp**

Backtracking (Exemplo)

Labirinto:

Encontrar a saída de um labirinto usando backtracking.

```
Tamanho do mapa: 9 8
   ########
   #.#.#.##
   #......#
   #.#.#.##
   #.###..#
   #...##..
   ##.###.#
   #...o#.#
   ########
Posicao inicial: 7 4
```

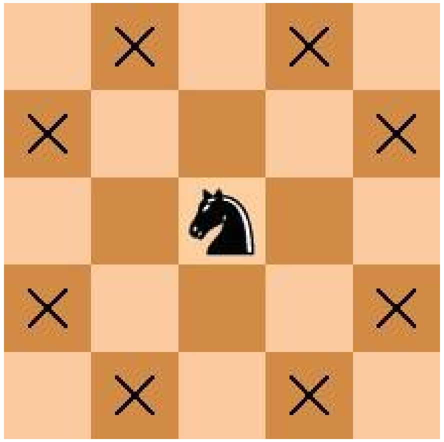
```
Tamanho do mapa: 5 6
   ######
   #o...#
   #....#
   #....#
   ######
Posicao inicial: 1 1
```

Ver arquivo: **labirinto.cpp**

Backtracking (Exemplo)

Passeio do Cavalo:

Dado um tabuleiro de xadrez com dimensões n × n, considere o cavalo posicionado numa casa de coordenadas (x_0 , y_0). Deseja-se obter, **se existir**, uma sequência de movimentos (seguindo as regras de movimento desta peça) de modo que todas as posições do tabuleiro sejam visitadas exatamente uma única vez.



Backtracking (Exemplo)

Sudoku:

O objetivo do Sudoku é preencher uma matriz 9x9 com números de 1 a 9, de forma que em cada linha, cada coluna e cada seção 3x3 não pode conter dígitos iguais.

	6			1				2
	4		6	2		7		
			3		7		8	
	2			4		9		
		1				2		
		3		8			5	
	9		8		4			
		4		3	2		9	
2				7			4	

Branch and Bound

É uma extensão do método **Backtracking**.

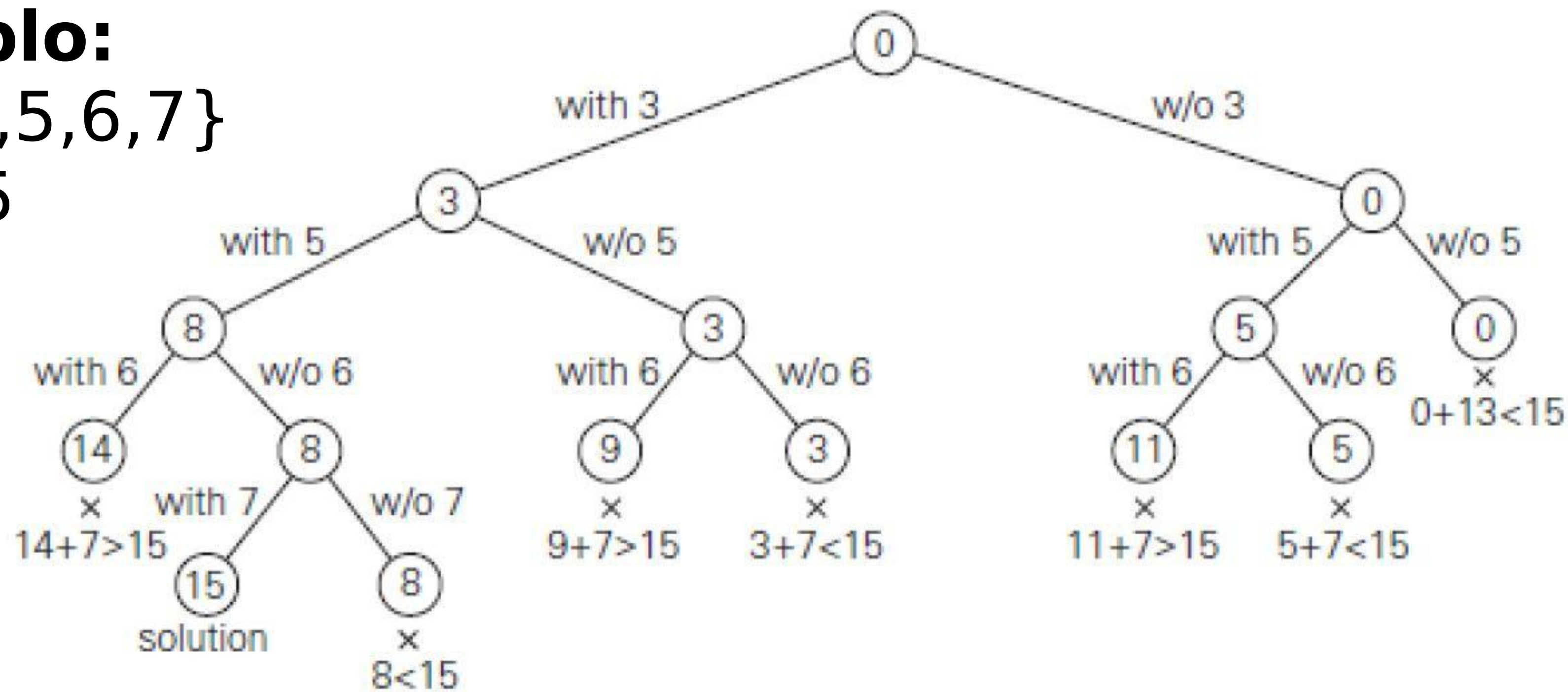
Em geral, aplicável a **problemas de otimização**.

O **branch-and-bound** é aplicado através de um **corte** da sequência de geração de candidatos, pois a partir daquele ponto, não há necessidade de continuar, visto que só vai levar a soluções “piores”.

Branch and Bound (Exemplo)

Soma de Subconjuntos: $\Theta(2^n)$ (somaSub.cpp)
Dado um conjunto S com n valores positivos e um valor m também positivo, determine todos os subconjuntos de S cuja soma é m.

Exemplo:
S = {3,5,6,7}
m = 15



Programação Dinâmica

Programação Dinâmica

A ***programação dinâmica***, assim como na Divisão e Conquista, resolve problemas combinando as soluções de subproblemas.

Nesse contexto, "***programação***" se refere a um método tabular, não ao processo de escrever códigos.

Programação Dinâmica

A ***DC*** subdivide o problema em subproblemas independentes, os resolve recursivamente e depois combina suas soluções para resolver o problema original.

Ao contrário, a ***PD*** se aplica quando os subproblemas se sobrepõem, ela **resolve cada subproblema só uma vez** e **guarda sua resposta em uma tabela**, evitando assim o trabalho de recalcular uma mesma resposta várias vezes.

Programação Dinâmica

A **programação dinâmica** pode acelerar bastante a resolução de alguns problemas.

Pré-requisitos

- Solução ótima do problema pode ser obtida a partir da solução ótima de subproblemas
- Vários problemas maiores precisam da solução dos mesmos subproblemas

Idéia geral

- ***Memoization***: manter uma tabela com resultados de problemas menores e utilizá-los para solucionar problemas maiores

Programação Dinâmica

Observação (Maratonas de Programação):

- Pode ser a chave para resolver um problema
- Pode ser o truque para resolver problemas que não passariam no tempo limite com outras técnicas
- Geralmente um ou mais problemas precisam de PD

Programação Dinâmica

Características (Programação Dinâmica):

Solução "top down":

- Dividir o problema em subproblemas menores
- Guardar soluções dos subproblemas em uma tabela à medida que são resolvidos
- Só resolver um subproblema se sua solução ainda não está guardada na tabela
- Recursão + ***memoization***

Programação Dinâmica

Características (Programação Dinâmica):

Solução “bottom up”:

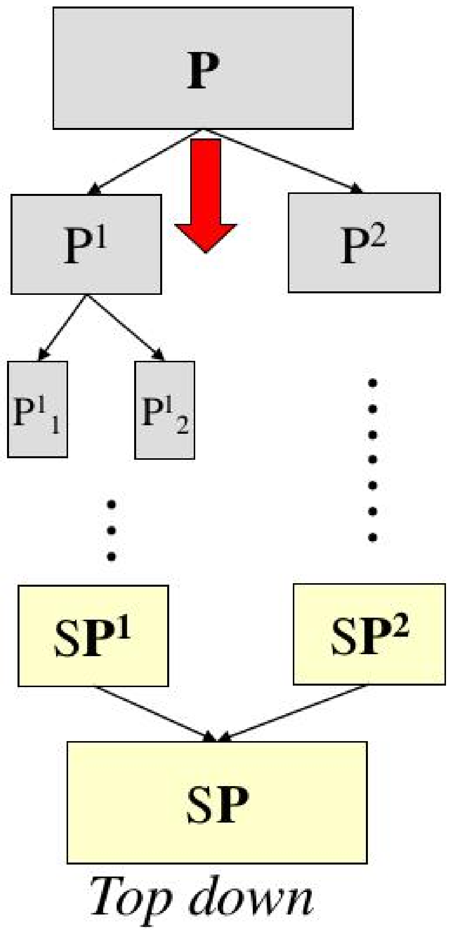
- Resolver antecipadamente os subproblemas que podem ser necessários em “**ordem de tamanho**” guardando os resultados em uma tabela

- Ao resolver um problema, seus subproblemas já foram resolvidos

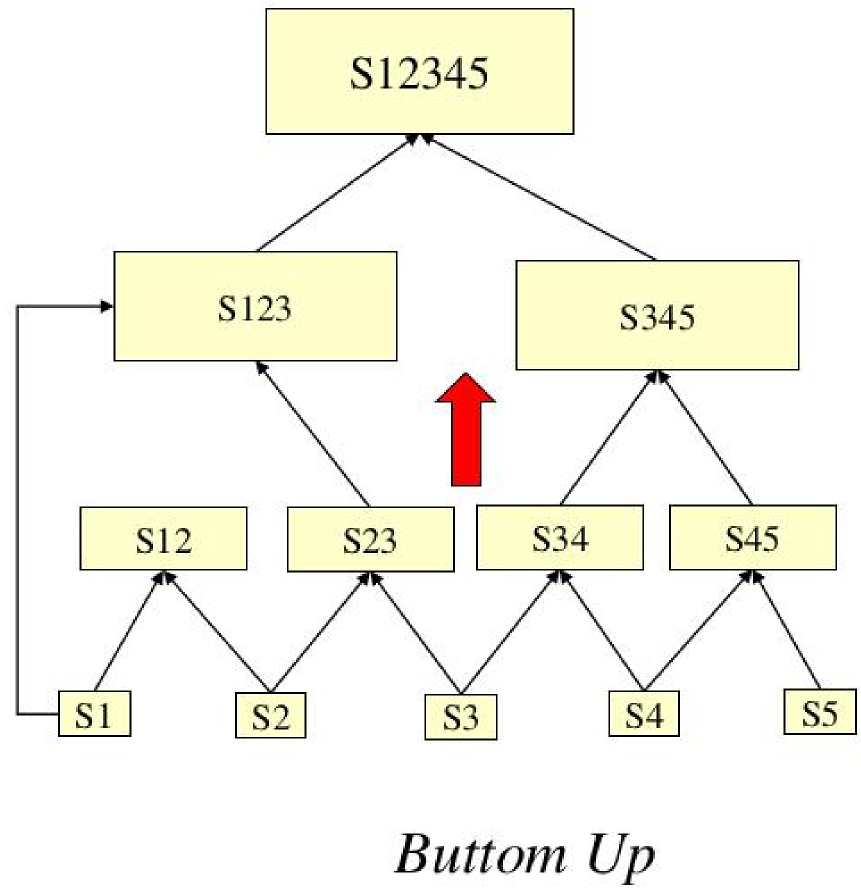
Ou seja, as soluções parciais armazenadas na tabela são usadas para obter soluções de problemas cada vez maiores, até gerar a solução do problema original.

Programação Dinâmica

Divisão e Conquisa



Programação Dinâmica



Programação Dinâmica

Cálculo de Fibonacci (sem PD):

```
int fibo (int n){
if (n <= 1)
   return n;
else
   return fibo(n-1) + fibo(n-2);
}
```

- Muitos cálculos desnecessários
- Para *n* grande pode ficar lento.

Programação Dinâmica

Cálculo de Fibonacci (com PD top down):

```
int tab[MAX]; //contendo 0,1,-1,-1,-1,-1,-1,...

int fibo (int n) {
   if (tab[n] == -1)
      tab[n] = fibo(n-1) + fibo(n-2);
   return tab[n];
}
```

- fibo(n) é calculado uma única vez para cada valor de n

Ver arquivo: **fibo.cpp**

Programação Dinâmica

Cálculo de Fibonacci (com PD bottom up):

```
int tab[MAX];

int fibo (int n) {
   tab[0] = 0;
   tab[1] = 1;
   for(int i = 2; i <= n; i++)
      tab[n] = tab[n-1] + tab[n-2];
   return tab[n];
}
```

Elimina a recursividade

Ver arquivo: **fibo.cpp**

Programação Dinâmica

Comparação (de uma forma geral)

“Botom up”

- elimina a recursividade, mas preenche a tabela inteira

“Top down”

- Só calcula realmente o que precisa, utilizando recursividade

Qual delas utilizar?

- A que você tiver mais habilidade
- A que você conseguir codificar o problema
- A que for mais fácil para codificar o problema
- A que for mais rápida para o problema
- A que funcionar primeiro...

Programação Dinâmica

O problema do Subarranjo Máximo:

Dado um conjunto de valores, seu objetivo é determinar um subarranjo contíguo deste conjunto cujos valores resultem na maior soma.

Day	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Price	100	113	110	85	105	102	86	63	81	101	94	106	101	79	94	90	97
Change		13	−3	−25	20	−3	−16	−23	18	20	−7	12	−5	−22	15	−4	7

Força Bruta: $\Theta(n^2)$

Divisão e Conquista: $\Theta(n \log n)$

Programação Dinâmica

O problema do Subarranjo Máximo:

Programação Dinâmica: O algoritmo consiste em armazenar os resultados de cálculos intermediários numa tabela para evitar que eles sejam repetidos.

Na idéia, a cada passo, ele acumula a soma com os anteriores. Se a soma anterior for negativa, ele recomeça.

Day	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Price	100	113	110	85	105	102	86	63	81	101	94	106	101	79	94	90	97
Change		13	−3	−25	20	−3	−16	−23	18	20	−7	12	−5	−22	15	−4	7
Soma		13	10	-15	20	17	1	-22	18	38	31	**43**	38	16	31	27	34

Complexidade: **Θ(n)**

Ver arquivo: **maxSubArranjo.cpp**

Programação Dinâmica

O problema da Mochila:

Dados n itens (objetos) cujos pesos são w_1 , w_2 , ..., w_n e os custos são c_1 , c_2 , ..., c_n , determine o subconjunto de itens mais valiosos (de maior custo total) que caibam dentro de uma mochila de capacidade W.

Exemplo

- Capacidade: 17
- Peso: **5** 7 3 **4** **8**
- Valor: **11** 14 6 **9** **17** – Total: 37

Programação Dinâmica

O problema da Mochila:

Algoritmo Recursivo (Top-Down): $\Theta(2^n)$

Seja C[i][j] o valor máximo considerando itens 1 até **i** numa mochila de capacidade **j**, $\mathbf{w_i}$ (peso de i), $\mathbf{v_i}$ (valor de i)**:**

Se i == 0 ou j == 0 // se não há itens ou não cabe nada
  C[i][j] = 0

Se w_i > j // se item i não cabe nessa mochila
  C[i][j] = C[i-1][j] // considera somente os itens anteriores

Se w_i <= j
  C[i][j] = max { C[i-1][j], // mochila sem o item
                 C[i-1][j-w_i] + v_i } // mochila com o item

Solução: C[n][W]

Programação Dinâmica

O problema da Mochila:

Mas note, há, no máximo, ***n*** *x* ***W*** instâncias distintas de (sub)problemas.

O que nos permite concluir que na estratégia anterior, uma mesma instância dos (sub)problemas foram resolvidas repetidas vezes.

Para evitar este fato, vamos lançar mão da programação dinâmica.

Programação Dinâmica

O problema da Mochila (Programação Dinâmica):

Vamos criar uma matriz **V** de dimensão ***(n+1) × (W+1)*** onde cada linha representa um item e cada coluna a capacidade de uma mochila.

Isto é, **V(0, j) = 0**, para todo j e **V(i, 0) = 0,** para todo i; Intuitivamente, **V(0, j)** equivale a dizer que não há nenhum item e **V(i, 0)** equivale a uma mochila de capacidade 0.

A cada passo, a matriz será preenchida com valores de subproblemas já resolvidos.

Programação Dinâmica

O problema da Mochila (Programação Dinâmica):

Exemplo

- Capacidade: 17
- Peso: **5** 7 3 **4 8**
- Valor: **11** 14 6 **9 17** – Total: 37

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	11	11	11	11	11	11	11	11	11	11	11	11	11
2	0	0	0	0	0	11	11	14	14	14	14	14	25	25	25	25	25	25
3	0	0	0	6	6	11	11	14	17	17	20	20	25	25	25	31	31	31
4	0	0	0	6	9	11	11	15	17	20	20	23	26	26	29	31	34	34
5	0	0	0	6	9	11	11	15	17	20	20	23	26	28	29	32	34	37

Programação Dinâmica

O problema da Mochila (Programação Dinâmica):

Complexidade: $\Theta(nW)$

Ver arquivo: **mochila01.cpp**

Programação Dinâmica

O problema da Maior Subsequência Comum *(Longest Common Subsequence)*:

Dada duas sequências $X=x_1x_2 \ldots x_m$ e $Y=y_1y_2 \ldots y_n$, o problema da maior subsequência comum (LCS) consiste em determinar a subsequência de X e Y de maior comprimento.

Exemplos:

X = abcdefg   Y = hfbcdka

Subsequência comum de maior comprimento: S = bcd

X = heroically   Y = scholarly

Subsequência comum de maior comprimento: S = holly

Programação Dinâmica

O problema da Maior Subsequência Comum:

Aplicações: Sequenciamento de DNA, Busca em texto

Força Bruta:

- Para cada subsequência de X verificar se é uma subsequência de Y.
- Se o tamanho da sequência X é n, então o número de subsequências de X é 2^n.
- Para verificar se uma subsequência de X é subsequência de Y, é gasto $\Theta(m)$. (varrer Y)

Complexidade: $\Theta(m2^n)$

Programação Dinâmica

O problema da Maior Subsequência Comum:

Algoritmo Recursivo (Top-Down):

Seja LCS(X,Y) o comprimento da maior subsequência comum entre as sequências X e Y.

Se o último caracter é igual

- Então é 1 + a LCS dos strings sem esse caracter

Se o último caracter não é igual

- Calcular a LCS da string s1 inteira e s2 sem o último caracter
- Calcular a LCS da string s1 sem o último caracter e s2 inteira
- A solução é o que for maior

Complexidade: $\Theta(m2^n)$

Programação Dinâmica

O problema da Maior Subsequência Comum:

Algoritmo Recursivo (Top-Down):

LCS (“ACTGT**T**”, “CATG**T**”) = 1 + LCS (“ACTGT”, “CATG”)

LCS (“ACTGT”, “CATG”) = max (LCS (“ACTG”, “CATG”),
LCS (“ACTGT”, “CAT”))

Novamente existem apenas **Θ(mn)** instâncias diferentes do problema. Portanto, uma mesma instância está sendo resolvida várias vezes.

Programação Dinâmica

O problema da Maior Subsequência Comum:

Programação Dinâmica:
A versão *bottom-up* faz o caminho inverso.

Vamos criar uma matriz L de dimensão (n+1) × (m+1).

L[i][0] = 0 e L[0][j] = 0, subsequencias de tamanho 0.

L[i][j] = 1+ L[i-1][j-1], se $x_i == y_j$
max(L[i-1][j], L[i][j-1]), se x_i != y_j

Complexidade: **Θ(mn)**

Programação Dinâmica

O problema da Maior Subsequência Comum:

Programação Dinâmica:

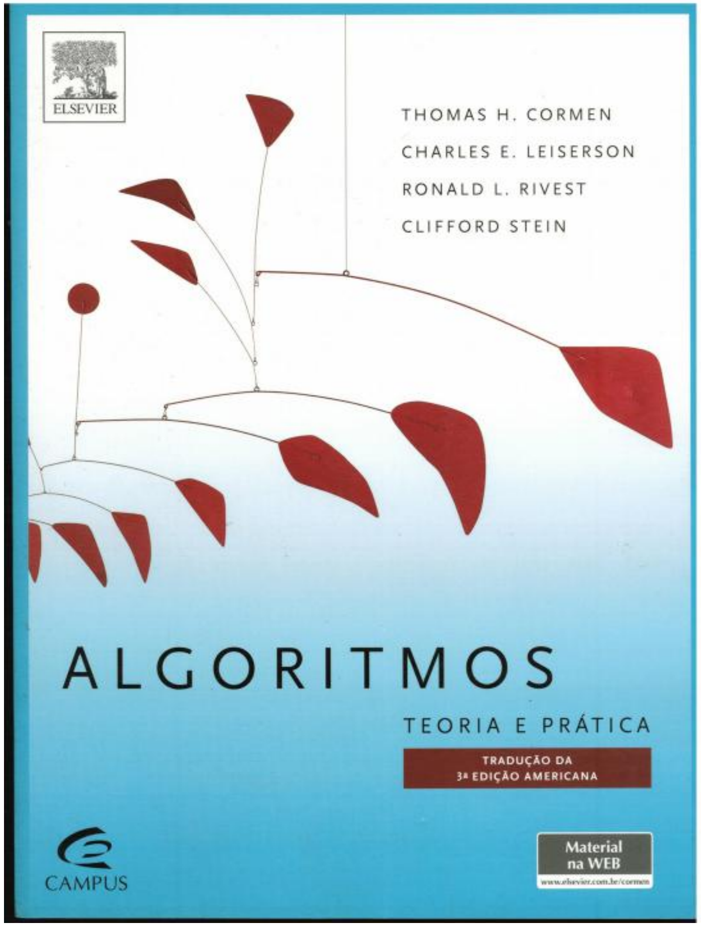
		A	C	G	T	T	G	C	C	A	G
	0	0	0	0	0	0	0	0	0	0	0
C	0	0	1	1	1	1	1	1	1	1	1
A	0	1	1	1	1	1	1	1	1	2	2
T	0	1	1	1	2	2	2	2	2	2	2
G	0	1	1	2	2	2	3	3	3	3	3
T	0	1	1	2	3	3	3	3	3	3	3
G	0	1	1	2	3	3	4	4	4	4	4
A	0	1	1	2	3	3	4	4	4	5	5
T	0	1	1	2	3	4	4	4	4	5	5
T	0	1	1	2	3	4	4	4	4	5	5

Ver arquivo: **LCS.cpp**

Complexidade: **Θ(mn)**

Referência Bibliográfica

- CORMEN, T. H.; LEISERSON, C. E.; RIVEST, R. L.; STEIN, C. Algoritmos: teoria e prática. Tradução da 2. ed. Americana. Rio de Janeiro: Campus, 2002.



Referência Bibliográfica

- SKIENA, S.S. REVILLA, M. A. Programming challenges: the programming contest training manual. Springer, 2003.