



UNIVERSIDADE DE SÃO PAULO
INSTITUTO DE CIÊNCIAS MATEMÁTICAS E DE COMPUTAÇÃO

Departamento de Ciências de Computação

<http://www.icmc.usp.br>

SCC-211 - Capítulo 11

Programação Dinâmica

João Luís Garcia Rosa¹

¹Departamento de Ciências de Computação
Instituto de Ciências Matemáticas e de Computação
Universidade de São Paulo - São Carlos
<http://www.icmc.usp.br/~joaoluiz>

2011

Sumário

- 1 Paradigmas de Resolução de Problemas
 - Busca Completa (*backtracking*)
 - Algoritmos divisão-e-conquista
 - Algoritmos gulosos

- 2 Programação Dinâmica
 - PD
 - Exemplo

Os quatro paradigmas

- Os quatro paradigmas de resolução de problemas mais comumente usados em competições de programação são [3]:
 - 1 Busca Completa (*backtracking*)
 - 2 Divisão e Conquista
 - 3 Busca Gulosa
 - 4 Programação Dinâmica

Sumário

1 Paradigmas de Resolução de Problemas

- Busca Completa (*backtracking*)
- Algoritmos divisão-e-conquista
- Algoritmos gulosos

2 Programação Dinâmica

- PD
- Exemplo

Algoritmos tentativa-e-erro

- Um algoritmo tentativa-e-erro é aquele que testa **exaustivamente** todas as soluções possíveis de um problema, de modo a obter a desejada,
- As soluções são testadas **indiscriminadamente**:
 - Não utiliza critérios para eliminar outras soluções que não poderão ser melhores que a obtida no estágio considerado.
- As soluções são enumeradas de modo semelhante ao percurso em uma árvore que possua todas as soluções,
- Muitas vezes a “árvore” de soluções cresce exponencialmente [8]!

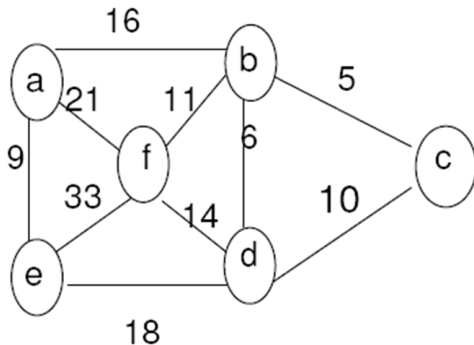


Algoritmos tentativa-e-erro [8]

- Algoritmos tentativa e erro não seguem regra fixa de computação:
 - Passos em direção à solução final são tentados e registrados,
 - Caso esses passos tomados não levem à solução final, eles podem ser retirados e apagados do registro.
- Quando a pesquisa na árvore de soluções cresce rapidamente é necessário usar **algoritmos aproximados** ou **heurísticas** que não garantem a solução ótima mas são rápidas.

Algoritmos tentativa-e-erro

- Exercício:
 - Qual o menor caminho da cidade **a** até a **c**?



Algoritmos tentativa-e-erro

- Exercício:
 - **TODOS** os caminhos são enumerados:
 - $a \Rightarrow b \Rightarrow c$: **21**
 - $a \Rightarrow b \Rightarrow d \Rightarrow c$: **32**
 - $a \Rightarrow b \Rightarrow f \Rightarrow d \Rightarrow c$: **51**
 - ...

Sumário

1 Paradigmas de Resolução de Problemas

- Busca Completa (*backtracking*)
- Algoritmos divisão-e-conquista
- Algoritmos gulosos

2 Programação Dinâmica

- PD
- Exemplo

Algoritmos divisão-e-conquista

- O paradigma divisão-e-conquista consiste em:
 - 1 Dividir o problema a ser resolvido em subproblemas menores e independentes;
 - 2 Encontrar soluções para as partes;
 - 3 Combinar as soluções obtidas em uma solução global.
- Os algoritmos utilizam **recursão** para dividir e combinar.
- Processo recursivo :
 - dada uma entrada, se ela é suficientemente **simples**, obtém-se **diretamente** uma saída correspondente;
 - caso contrário, ela é **decomposta** em entradas mais simples, para as quais se aplica o mesmo processo, obtendo saídas correspondentes que são então combinadas em uma saída para a entrada original.

Algoritmos divisão-e-conquista [8]

- **Exemplo:** encontrar o maior e o menor elemento de um vetor de inteiros, $A[1..n]$, $n \geq 1$:

```
void MaxMin4 (int Linf, int Lsup, int *Max, int *Min)
{
    int Max1, Max2, Min1, Min2, Meio;
    if (Lsup - Linf <= 1)
    { if (A[Linf-1] < A[Lsup-1])
      { *Max = A[Lsup-1]; *Min = A[Linf-1]; }
      else { *Max = A[Linf-1]; *Min = A[Lsup-1]; } }
    else {
        Meio = (Linf+Lsup)/2;
        MaxMin4(Linf, Meio, &Max1, &Min1);
        MaxMin4(Meio+1, Lsup, &Max2, &Min2);
        if (Max1 > Max2) *Max = Max1; else *Max = Max2;
        if (Min1 < Min2) *Min = Min1; else *Min = Min2; }
}
```

- Cada chamada de *MaxMin4* atribui à *Max* e *Min* o maior e o menor elemento em $A[Linf]$, $A[Linf + 1]$, ..., $A[Lsup]$.

Divisão-e-conquista: análise do exemplo [8]

- Seja $f(n)$ o número de comparações entre os elementos de A , se A contiver n elementos,

$$f(n) = \begin{cases} 1 & \text{para } n \leq 2 \\ f(\lfloor n/2 \rfloor) + f(\lceil n/2 \rceil) + 2 & \text{para } n > 2 \end{cases}$$

- Quando $n = 2^i$ para algum inteiro positivo i :

$$\begin{aligned} f(n) &= 2f(n/2) + 2 \\ 2f(n/2) &= 4f(n/4) + 2 \times 2 \\ 4f(n/4) &= 8f(n/8) + 2 \times 2 \times 2 \\ &\vdots \\ 2^{i-2}f(n/2^{i-2}) &= 2^{i-1}f(n/2^{i-1}) + 2^{i-1} \end{aligned}$$

Divisão-e-conquista: análise do exemplo [8]

- Adicionando lado a lado, obtém-se:

$$\begin{aligned}f(n) &= 2^{i-1}f(n/2^{i-1}) + \sum_{k=1}^{i-1} 2^k \\&= 2^{i-1}f(2) + 2^i - 2 \\&= 2^{i-1} + 2^i - 2 \\&= \frac{3n}{2} - 2\end{aligned}$$

- Logo, $f(n) = 3n/2 - 2$ para o melhor caso, pior caso e caso médio (solução **ótima**).

Algoritmos divisão-e-conquista

- Exemplos de algoritmos:
 - Busca binária;
 - *SelectionSort*, *MergeSort*, *Quicksort*;
 - Maior elemento de uma sequência;
 - Fibonacci recursivo.

Sumário

1 Paradigmas de Resolução de Problemas

- Busca Completa (*backtracking*)
- Algoritmos divisão-e-conquista
- Algoritmos gulosos

2 Programação Dinâmica

- PD
- Exemplo

Algoritmos gulosos

- São tipicamente usados para resolver problemas de **otimização**,
- Por exemplo, o algoritmo para encontrar o caminho mais curto entre duas cidades:
 - Um algoritmo guloso escolhe a estrada que **parece mais promissora** no instante atual e **nunca muda essa decisão**, independentemente do que possa acontecer depois.
- A cada iteração:
 - seleciona um elemento conforme uma função gulosa,
 - marca-o para não considerá-lo novamente nos próximos estágios,
 - atualiza a entrada,
 - examina o elemento selecionado quanto sua viabilidade,
 - decide a sua participação ou não na solução.

Algoritmos gulosos

- Algoritmo guloso genérico:
 - C: conjunto de candidatos;
 - S: conjunto solução.
- $S = \emptyset$
- Enquanto ($C \neq \emptyset$) e (S não tem solução)
 - $x = \text{seleciona}(C)$
 - $C = C - \{x\}$
 - Se ($S + \{x\}$ é viável) então $S = S + \{x\}$
- Se (S tem solução) então retorna S
- Senão não existe solução

Características dos Algoritmos Gulosos [8]

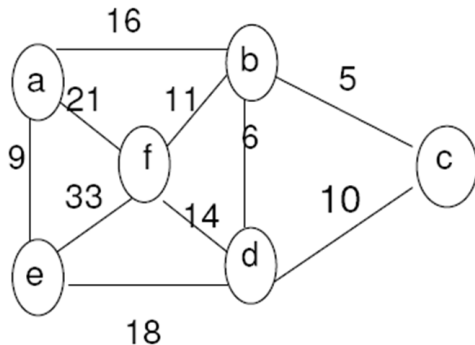
- Para construir a solução ótima existe um conjunto ou lista de candidatos,
- São acumulados um conjunto de candidatos considerados e escolhidos, e o outro de candidatos considerados e rejeitados,
- Existe uma função que verifica se um conjunto particular de candidatos produz uma **solução** (sem considerar otimalidade no momento),
- Outra função verifica se um conjunto de candidatos é **viável** (também sem se preocupar com a otimalidade),
- Uma **função de seleção** indica a qualquer momento quais dos candidatos restantes é o mais promissor,
- Uma **função objetivo** fornece o valor da solução encontrada, como o comprimento do caminho construído (não aparece de forma explícita no algoritmo guloso).

Características da Implementação de Algoritmos Gulosos [8]

- Quando funciona corretamente, a primeira solução encontrada é sempre ótima,
- A função de seleção é geralmente relacionada com a função objetivo,
- Se o objetivo é:
 - maximizar $\Rightarrow$ provavelmente escolherá o candidato restante que proporcione o maior ganho individual,
 - minimizar $\Rightarrow$ então será escolhido o candidato restante de menor custo.
- O algoritmo nunca muda de ideia:
 - Uma vez que um candidato é escolhido e adicionado à solução ele lá permanece para sempre,
 - Uma vez que um candidato é excluído do conjunto solução, ele nunca mais é reconsiderado.

Algoritmos gulosos

- Exercício:
 - Calcule o menor caminho da cidade *a* até a *c*, utilizando um algoritmo guloso.



Sumário

- 1 Paradigmas de Resolução de Problemas
 - Busca Completa (*backtracking*)
 - Algoritmos divisão-e-conquista
 - Algoritmos gulosos
- 2 Programação Dinâmica
 - PD
 - Exemplo

Introdução

- Projetistas de algoritmos e programadores: escrever programas que achem a **melhor** solução para todas as instâncias dos problemas.
- É relativamente fácil escrever um programa que forneça uma solução decente e correta, mas assegurar que **sempre** ele retorna a melhor solução...
- **Programação Dinâmica** é uma ferramenta geral muito poderosa para resolver problemas de otimização em itens ordenados da esquerda para a direita, como as cadeias de caracteres.
- É necessário trabalhar com muitos exemplos para entender PD!

Programação Dinâmica

- Quando um algoritmo recursivo tem complexidade exponencial, a programação dinâmica (**PD**) pode levar a um algoritmo mais eficiente,
- A técnica de programação dinâmica consiste em **dividir** o problema original em subproblemas **mais simples** e resolvê-los, armazenando os resultados em uma **tabela**,
- Isso é feito **iterativamente**.
- A ideia básica da programação dinâmica é construir por **etapas** uma resposta ótima **combinando** respostas já obtidas para partes menores,
- Inicialmente, a entrada é **decomposta** em partes mínimas, para as quais são obtidas respostas.

Programação Dinâmica

- Em cada passo, sub-resultados são **combinados** obtendo-se respostas para partes maiores, até que se obtenha uma resposta para o problema original,
- A decomposição é feita uma única vez e os casos menores são tratados antes dos maiores,
- Suas soluções são armazenadas para serem usadas quantas vezes for necessário.

- Estrutura geral da programação dinâmica [6]:



Programação Dinâmica

- É baseada no **princípio da otimalidade**:
 - Em uma sequência ótima de escolhas ou de decisões, cada subsequência também deve ser ótima:
 - Por exemplo: o menor caminho de São Carlos a São Paulo passando por Campinas é dado pelo menor caminho de São Carlos a Campinas **combinado** com o menor caminho de Campinas a São Paulo.
- Reduz drasticamente o número total de verificações pois evita aquelas que sabidamente não podem ser ótimas:
 - A cada passo são **eliminadas** subsoluções que certamente não farão parte da solução ótima do problema.

Programação Dinâmica

- Exemplo: Qual a melhor maneira de se fazer o produto entre n matrizes?
 - O produto de uma matriz $p \times q$ por uma matriz $q \times r$ requer $\mathcal{O}(pqr)$ operações.
- Considere o produto:
 - $M = M_1[10, 20] \times M_2[20, 50] \times M_3[50, 1] \times M_4[1, 100]$:
 - O produto na ordem $M = M_1 \times (M_2 \times (M_3 \times M_4))$ requer **125.000** operações;
 - O produto na ordem $M = (M_1 \times (M_2 \times M_3)) \times M_4$ requer **2.200** operações.

Programação Dinâmica

- Resolver esse problema por **tentativa e erro**, isto é, testar todas as ordens possíveis de fazer o produto das matrizes para se obter qual é a melhor ($f(n)$), é um processo **exponencial** em n ($f(n) \geq 2^{n-2}$ [1]).
- Usando programação dinâmica é possível obter um algoritmo $\mathcal{O}(n^3)$!

Programação Dinâmica

- Seja m_{ij} o menor custo para calcular o produto $M_i \times M_{i+1} \dots \times M_j$ para $1 \leq i \leq j \leq n$,
- Assim,

$$m_{ij} = \begin{cases} 0 & \text{se } i = j \\ \min_{i \leq k \leq j} (m_{ik} + m_{k+1,j} + b_{i-1}b_kb_j) & \text{se } j > i \end{cases}$$

- onde:
 - o termo m_{ik} representa o custo mínimo para calcular $M' = M_i \times M_{i+1} \times \dots \times M_k$,
 - o segundo termo $m_{k+1,j}$ representa o custo mínimo para calcular $M'' = M_{k+1} \times M_{k+2} \times \dots \times M_j$,
 - o terceiro termo, $b_{i-1}b_kb_j$, representa o custo de multiplicar $M'[b_{i-1}, b_k]$ por $M''[b_k, b_j]$,
 - $b_{i-1} \times b_i$ são as dimensões da matriz M_i ,
 - a equação acima diz que $m_{ij}, j > i$, representa o custo mínimo de todos os valores possíveis de k entre i e $j - 1$, da soma dos três termos.

Programação Dinâmica

- Seja m_{ij} o menor custo para calcular o produto $M_i \times M_{i+1} \dots \times M_j$ para $1 \leq i \leq j \leq n$,
- Assim,

$$m_{ij} = \begin{cases} 0 & \text{se } i = j \\ \text{Min}_{i \leq k \leq j} (m_{ik} + m_{k+1,j} + b_{i-1}b_kb_j) & \text{se } j > i \end{cases}$$

- *Min*

$$\underbrace{i \leq k \leq j}$$

princípio da otimalidade

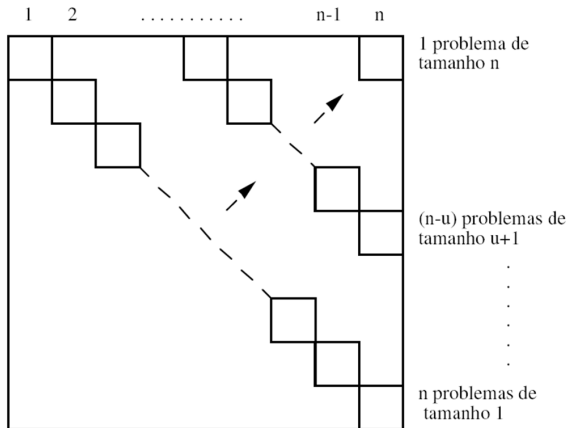
Em uma sequência ótima de escolhas ou de decisões, cada subsequência também deve ser ótima.

Programação Dinâmica

- Os valores m_{ij} são calculados na ordem crescente das diferenças nos subscritos,
- O cálculo inicia com m_{ii} para todo i , depois $m_{i,i+1}$ para todo i , depois $m_{i,i+2}$, ...
- Assim, esse método é chamado ascendente, ao contrário dos métodos recursivos, que são chamados descendentes.

Programação Dinâmica

- Hierarquia de instâncias na programação dinâmica [6]:



Programação Dinâmica

- **Exemplo:** Tabela de dados calculados pela programação dinâmica para o produto:

- $M = M_1[10, 20] \times M_2[20, 50] \times M_3[50, 1] \times M_4[1, 100]$

$m_{11} = 0$	$m_{12} = 10.000$	$m_{13} = 1.200$	$m_{14} = 2.200$
	$m_{22} = 0$	$m_{23} = 1.000$	$m_{24} = 3.000$
		$m_{33} = 0$	$m_{34} = 5.000$
			$m_{44} = 0$

- **Exercício:** Escrever um algoritmo para achar o produto de menor custo entre n matrizes, usando programação dinâmica.

Programação Dinâmica

```
#define Maxn 10
int main(int argc, char *argv[])
{
    int i, j, k, h, n, temp;
    int b[Maxn+1];
    int m[Maxn][Maxn];
    printf("Numero de matrizes n: ");
    scanf("%d", &n);
    getchar();
    printf("Dimensoes das matrizes: ");
    for (i = 0; i <= n; i++)
        scanf("%d", &b[i]);
    for (i = 0; i < n; i++)
        m[i][i] = 0;
    for (h = 1; h <= n-1; h++) {
        for (i = 1; i <= n-h; i++) {
            j = i+h;
            m[i-1][j-1] = INT_MAX;
            for (k = i; k <= j-1; k++)
            {
                temp = m[i-1][k-1] + m[k][j-1] + b[i-1] * b[k] * b[j];
                if (temp < m[i-1][j-1])
                    m[i-1][j-1] = temp;
            }
            printf("m[%d][%d] = %d\n", i-1, j-1, m[i-1][j-1]);
        }
        putchar('\n');
    }
    return 0;
}
```

Sumário

- 1 Paradigmas de Resolução de Problemas
 - Busca Completa (*backtracking*)
 - Algoritmos divisão-e-conquista
 - Algoritmos gulosos
- 2 Programação Dinâmica
 - PD
 - Exemplo

Flight Planner (10337)

- Calculating the minimal cost for a flight involves calculating an optimal flight-altitude depending on wind-strengths changing with different altitudes. It's not enough just to ask for the route with optimal wind-strength, because due to the mass of a plane you need a certain amount of fuel to rise. Moreover due to safety regulations it's forbidden to fly above a certain altitude and you can't fly under zero-level.
- In order to simplify the problem for now, we assume that for each 100 miles of flight you have only three possibilities: to climb one mile, to hold your altitude or to sink one mile.
- Climb flight requires 60 units of fuel, holding your altitude requires 30 units and sinking requires 20 units.

Flight Planner (10337)

- In the case of headwind you need more fuel while you can save fuel flying with tailwind. Wind strength w will satisfy the condition $-10 \leq w \leq 10$, where negative wind strength is meant to be headwind and positive wind strength is tailwind.
- For one unit of tailwind you can save one unit of fuel each 100 miles; each unit of headwind will cost an extra unit of fuel.
- For example to climb under conditions of wind strength $w = -5$, you need 65 units of fuel for this 100 miles.
- Given the wind strengths on different altitudes for a way from here to X , calculate the minimal amount of fuel you need to fly to X .

Flight Planner (10337)

- **Input:**

- The first line of the input file contains the number N of test cases in the file. The first line of each test case contains a single integer X , the distance to fly, with $1 \leq X \leq 100000$ miles and X is a multiple of 100. Notice that it's not allowed to fly higher than 9 miles over zero and that you have to decide whether to climb, hold your altitude or to sink only for every 100 miles.
- For every mile of allowed altitude (starting at altitude 9 down to altitude 0) there follow $X/100$ wind strengths, starting with the wind strength at your current position up to the wind strength at position $X - 100$ in steps of 100 miles. Test cases are separated by one or more blank lines.

Flight Planner (10337)

- **Output:**

- For each test case output the minimal amount of fuel used flying from your current position (at altitude 0) to X (also at altitude 0), followed by a blank line.

Flight Planner (10337)

Sample Input

```

2
400
1 1 1 1
1 1 1 1
1 1 1 1
1 1 1 1
1 1 1 1
1 1 1 1
1 1 1 1
1 1 1 1
1 1 1 1
1 1 1 1
1 9 9 1
1 -9 -9 1

1000
9 9 9 9 9 9 9 9 9 9 9
9 9 9 9 9 9 9 9 9 9 9
9 9 9 9 9 9 9 9 9 9 9
9 9 9 9 9 9 9 9 9 9 9
9 9 9 9 9 9 9 9 9 9 9
9 9 9 9 9 9 9 9 9 9 9
7 7 7 7 7 7 7 7 7 7 7
-5 -5 -5 -5 -5 -5 -5 -5 -5 -5 -5
-7 -3 -7 -7 -7 -7 -7 -7 -7 -7 -7
-9 -9 -9 -9 -9 -9 -9 -9 -9 -9 -9
    
```

Sample Output

120

354

Material do Stephen Halim

Os próximos 19 slides contêm material de Stephen Halim disponíveis em [4].

Motivation

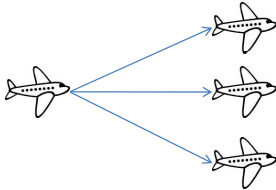


- How to solve this: [10337](#) (Flight Planner)?
 - Unit: 1 mile altitude and 1 (x100) miles distance
 - Given wind speed map
 - Fuel cost: {**climb** (+60), **hold** (+30), **sink** (+20)} - wind speed `wsp[alt][dis]`
 - Compute **min** fuel cost from (0, 0) to (0, X = 4)!

1	1	1	1		9
1	1	1	1		8
1	1	1	1		7
1	1	1	1		6
1	1	1	1		5
1	1	1	1		4
1	1	1	1		3
1	1	1	1		2
1	9	9	1		1
1	-9	-9	1		0
=====					
0	1	2	3	4	(x100)

Complete Search? (1)

- First guess:
 - Do complete search/brute force/**backtracking**
 - Find *all possible* flight paths and pick the one that yield the minimum fuel cost



Complete Search? (2)

- Recurrence of the Complete Search

- `fuel(alt, dis) =`
 `min3(60 - wsp[alt][dis] + fuel(alt + 1, dis + 1),`
 `30 - wsp[alt][dis] + fuel(alt, dis + 1),`
 `20 - wsp[alt][dis] + fuel(alt - 1, dis + 1))`

- Stop when we reach final state (base case):

- $\text{alt} = 0$ and $\text{dis} = X$, i.e. $\text{fuel}(0, X) = 0$

- Prune infeasible states (also base cases):

- $\text{alt} < 0$ or $\text{alt} > 9$ or $\text{dis} > X!$, i.e. return INF^*

- Answer of the problem is **`fuel(0, 0)`**

Complete Search Solutions (1)

- Solution 1

1	1	1	1		9
1	1	1	1		8
1	1	1	1		7
1	1	1	1		6
1	1	1	1		5
1	1	1	1		4
1	1	1	1		3
1	1	1	1		2
1	9	9	1		1
1	-9	-9	1		0
=====					
0	1	2	3	4	(x100)

29+ 39+ 39+ 29=136

- Solution 2

1	1	1	1		9
1	1	1	1		8
1	1	1	1		7
1	1	1	1		6
1	1	1	1		5
1	1	1	1		4
1	1	1	1		3
1	1	1	1		2
1	9	9	1		1
1	-9	-9	1		0
=====					
0	1	2	3	4	(x100)

29+ 39+ 69+ 19=156

Complete Search Solutions (2)

- Solution 3

1	1	1	1		9
1	1	1	1		8
1	1	1	1		7
1	1	1	1		6
1	1	1	1		5
1	1	1	1		4
1	1	1	1		3
1	1	1	1		2
1	9	9	1		1
1	-9	-9	1		0
=====					
0	1	2	3	4	(x100)

$$29+ 69+ 11+ 29=138$$

- Solution 4

1	1	1	1		9
1	1	1	1		8
1	1	1	1		7
1	1	1	1		6
1	1	1	1		5
1	1	1	1		4
1	1	1	1		3
1	1	1	1		2
1	9	9	1		1
1	-9	-9	1		0
=====					
0	1	2	3	4	(x100)

$$59+ 11+ 39+ 29=138$$

Complete Search Solutions (3)

- Solution 5

1	1	1	1		9
1	1	1	1		8
1	1	1	1		7
1	1	1	1		6
1	1	1	1		5
1	1	1	1		4
1	1	1	1		3
1	1	1	1		2
1	9	9	1		1
1	-9	-9	1		0
=====					
0	1	2	3	4	(x100)

$$29+ 69+ 21+ 19=138$$

- Solution 6

1	1	1	1		9
1	1	1	1		8
1	1	1	1		7
1	1	1	1		6
1	1	1	1		5
1	1	1	1		4
1	1	1	1		3
1	1	1	1		2
1	9	9	1		1
1	-9	-9	1		0
=====					
0	1	2	3	4	(x100)

$$59+ 21+ 11+ 29=120(\text{OPT})$$

Complete Search Solutions (4)

- Solution 7

1	1	1	1		9
1	1	1	1		8
1	1	1	1		7
1	1	1	1		6
1	1	1	1		5
1	1	1	1		4
1	1	1	1		3
1	1	1	1		2
1	9	9	1		1
1	-9	-9	1		0
=====					
0	1	2	3	4	(x100)

59+ 21+ 21+ 19=120(OPT)

- Solution 8

1	1	1	1		9
1	1	1	1		8
1	1	1	1		7
1	1	1	1		6
1	1	1	1		5
1	1	1	1		4
1	1	1	1		3
1	1	1	1		2
1	9	9	1		1
1	-9	-9	1		0
=====					
0	1	2	3	4	(x100)

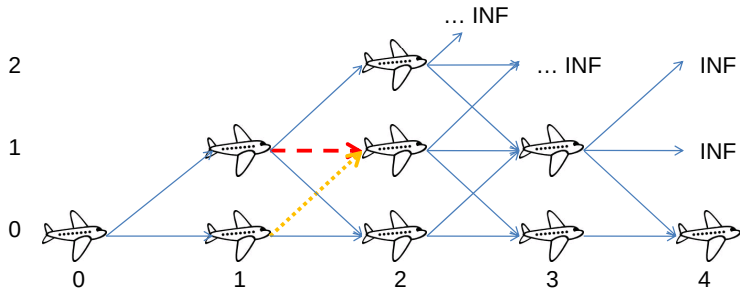
59+ 51+ 19+ 19=148

Complete Search? (3)

- How large is the search space?
 - Max distance is 100,000 miles
Each distance step is 100 miles
That means we have **1,000** distance columns!
 - Branching factor per step is 3... (climb, hold, sink)
 - That means complete search can end up performing $O(3^{1,000})$ operations...
 - Too many for 3s time limit 😞

Overlapping Sub Problem Issue

- In simple $O(3^{1,000})$ Complete Search solution, we observe **many overlapping sub problems!**
 - Many ways to reach coordinate (alt, dis)



DP to the Rescue (1)

- DP = Dynamic Programming
 - Programming here is not writing computer code, but a “tabular method”!
 - A programming paradigm that you must know!
 - And hopefully, master...

DP to the Rescue (2)

- Use DP when the problem exhibits:
 - Optimal sub structure
 - Optimal solution to the original problem contains optimal solution to sub problems
 - This is similar as the requirement of Greedy algorithm
 - Overlapping sub problems
 - Number of distinct sub problems are actually “small”
 - But they are **repeatedly computed**
 - This is different from Divide and Conquer

DP Solution (1)

- Recurrence* of the Complete Search

- `fuel(alt, dis) =`
 `min3(60 - wsp[alt][dis] + fuel(alt + 1, dis + 1),`
 `30 - wsp[alt][dis] + fuel(alt, dis + 1),`
 `20 - wsp[alt][dis] + fuel(alt - 1, dis + 1))`

- Sub-problem `fuel(alt, dis)` can be **overlapping!**

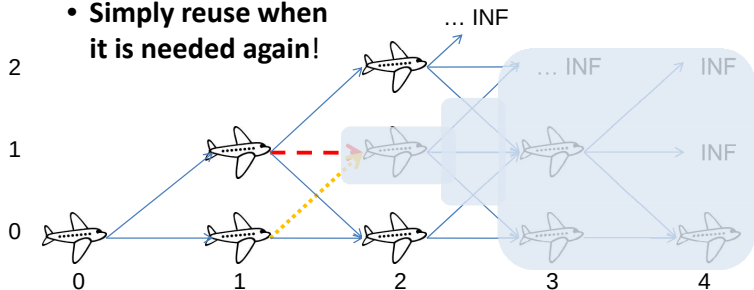
- There are only 10 alt and 1,000 dis = **10,000** states
 - A lot of time saved if these are not re-computed!
 - Exponential $O(3^{1,000})$ to polynomial $O(10 \cdot 1,000)$!

alt > 2 not shown

DP Solution (2)

2	-1	-1	-1	∞	∞
1	-1	-1	40	19	∞
0	-1	-1	-1	29	0
	0	1	2	3	4

- Create a 2-D table of size $10 * (X/100)$ ← Save Space!
 - Set “-1” for unexplored sub problems (memset)
 - Store the computation value of sub problem
- **Simply reuse when it is needed again!**



DP Solution – Implementation (1)

- There are two ways to implement DP:
 - Top-Down
 - Bottom-Up
- Top-Down (Demo):
 - Recursion as per normal + **memoization table**
 - It is just a simple change from backtracking (complete search) solution!

Turn Recursion into Memoization

initialize memo table in main function (memset?)

```
return_value recursive_function() {  
    if already calculated, simply return the result  
    calculate the result using recursion  
    save the result in the memo table  
    return the result  
}
```

DP Solution – Implementation (2)

- Another way: Bottom-Up (Demo):
 - Figure out **the order** in which the table is filled

- Recurrence relation must be modified a bit!

```
fuel(alt, dis) =  
    min3(20 - wsp[alt + 1][dis - 1] + fuel(alt + 1, dis - 1),  
         30 - wsp[alt    ][dis - 1] + fuel(alt    , dis - 1),  
         60 - wsp[alt - 1][dis - 1] + fuel(alt - 1, dis - 1))
```

– Then fill in the table using loops!

- Both DP variants use “table”!

Visualization

```
fuel(alt, dis) =
  min3(20 - wsp[alt + 1][dis - 1] + fuel(alt + 1, dis - 1),
        30 - wsp[alt ][dis - 1] + fuel(alt , dis - 1),
        60 - wsp[alt - 1][dis - 1] + fuel(alt - 1, dis - 1))
```

1	1	1	1		9
1	1	1	1		8
1	1	1	1		7
1	1	1	1		6
1	1	1	1		5
1	1	1	1		4
1	1	1	1		3
1	1	1	1		2
1	9	9	1		1
1	-9	-9	1		0
=====					
0	1	2	3	4	(x100)

Tips:
(on-the-fly)

We can reduce one storage dimension by only keeping 2 recent columns at a time...

But the time complexity is unchanged:
 $O(10 * X / 100)$

2	∞	∞			
1	∞	59			
0	0	29			
	0	1	2	3	4

2	∞	∞	110		
1	∞	59	80		
0	0	29	68		
	0	1	2	3	4

2	∞	∞	110	131	
1	∞	59	80	101	
0	0	29	68	91	
	0	1	2	3	4

2	∞	∞	110	131	-
1	∞	59	80	101	-
0	0	29	68	91	120
	0	1	2	3	4

Top-Down or Bottom-Up?

- Top-Down

- Pro:

- Natural transformation from normal recursion
- Only compute sub problems when necessary

- Cons:

- Slower if there are many sub problems due to recursive call overhead
- Use exactly $O(\text{states})$ table size (MLE?)

- Bottom Up

- Pro:

- Faster if many sub problems are visited: no recursive calls!
- Can save memory space^

- Cons:

- Not so intuitive?
- If there are X states, bottom up visits/fills the value of all these X states

If Optimal Solution(s) are Needed

- Although not often, sometimes this is asked!
- As we build the DP table, record which option is taken in each cell!
 - Usually, this information is stored in different table
 - Then, do recursive scan(s) to output solution
 - Sometimes, there are more than one solutions!
- Demo

2	∞	∞	110	131	-
1	∞	59	80	101	-
0	0	29	68	91	120
	0	1	2	3	4

Exercícios para Nota

- Optimal Binary Search Tree (10304)
- Vampires (11500)

Optimal Binary Search Tree (10304)

- Given a set $S = (e_1, e_2, \dots, e_n)$ of n distinct elements such that $e_1 < e_2 < \dots < e_n$ and considering a binary search tree of the elements of S , it is desired that higher the query frequency of an element, closer will it be to the root.
- The cost of accessing an element e_i of S in a tree ($cost(e_i)$) is equal to the number of edges in the path that connects the root with the node that contains the element. Given the query frequencies of the elements of S , ($f(e_1), f(e_2), \dots, f(e_n)$), we say that the total cost of a tree is the following summation:

$$f(e_1) * cost(e_1) + f(e_2) * cost(e_2) + \dots + f(e_n) * cost(e_n)$$

- In this manner, the tree with the lowest total cost is the one with the best representation for searching elements of S . Because of this, it is called the Optimal Binary Search Tree.

Optimal Binary Search Tree (10304)

- Input

- The input will contain several instances, one per line.
- Each line will start with a number $1 \leq n \leq 250$, indicating the size of S . Following n , in the same line, there will be n non-negative integers representing the query frequencies of the elements of S : $f(e_1), f(e_2), \dots, f(e_n)$. $0 \leq f(e_i) \leq 100$. Input is terminated by end of file.

- Output

- For each instance of the input, you must print a line in the output with the total cost of the Optimal Binary Search Tree.

Optimal Binary Search Tree (10304)

- Sample Input

```
1 5
3 10 10 10
3 5 10 20
```

- Sample Output

```
0
20
20
```

Vampires (11500)

- The Problem

- Felipinho is thrilled with his new RPG game, about wars between clans of vampires. In this game he plays a vampire that repeatedly comes into combat against vampires from other clans. Such battles are won or lost based on some characteristics of the opponents, with the help of a standard six-faced dice.
- For simplicity, we will consider only the fight between two vampires, Vampire 1 and Vampire 2. Each vampire has a vital energy (denoted respectively by EV_1 and EV_2). Besides, an attack force AT and a damage capacity D are also given.

Vampires (11500)

- The Problem (cont.)
 - The combat is fought in turns, in the following way. At each turn, the dice is rolled; if the result value is less than or equal to AT , Vampire 1 wins the turn, otherwise Vampire 2 wins. The winner then sucks the value D from the loser's vital energy. That is, D points are subtracted from the loser's vital energy and added to the winner's vital energy. The combat continues until one of the fighters has EV less than or equal to zero.
 - For example, suppose $EV_1 = 7$, $EV_2 = 5$, $AT = 2$ and $D = 4$. The dice is rolled and the result value is 3. Then, Vampire 2 wins the turn, and therefore 4 points are subtracted from EV_1 and added to EV_2 . The new values for the vital energies would be $EV_1 = 3$ and $EV_2 = 9$. Notice that, if in the next turn Vampire 2 wins again, the combat ends.

Vampires (11500)

- The Problem (cont.)
 - The values of AT and D are constant throughout the combat; only EV_1 and EV_2 vary. Despite loving the game, Felipinho thinks that the combats are too long, and suspects that, given the initial values of EV_1 , EV_2 , AT and D , it is possible to determine the probability of one of the players winning the combat, and that could help shorten the combat time.
 - Felipinho has asked your help to verify his suspicion.

Vampires (11500)

- The Input

- The input contains several test cases. Each test case is given in one single line, containing four integers EV_1 , EV_2 , AT and D separated by spaces ($1 \leq EV_1, EV_2 \leq 10$, $1 \leq AT \leq 5$ and $1 \leq D \leq 10$).
- The end of input is indicated by one line containing only four zeros, separated by spaces.

- The Output

- For each test case in the input, your program must print a single line. The line must contain a real number representing, in terms of percentages, the probability that Vampire 1 wins the combat. The result must be printed as a real number with exactly one decimal figure.

Vampires (11500)

- Sample Input

```
1 1 3 1
1 2 1 1
8 5 3 1
7 5 2 4
0 0 0 0
```

- Sample Output

```
50.0
3.2
61.5
20.0
```

Referências I



[1] Aho, A. V., Hopcroft, J. E., Ullman, J. D.
The Design and Analysis of Computer Algorithms.
Addison-Wesley, 1974.



[2] Cormen, T. H., Leiserson, C. E., Rivest, R. L., Stein, C.
Algoritmos - Teoria e Prática.
Ed. Campus, Rio de Janeiro, Segunda Edição, 2002.



[3] Halim, S., Halim, F.
Competitive Programming - Increasing the Lower Bound of Programming Contests.
Lulu, 2010.

Referências II



[4] Halim, S.

CS3233 - Competitive Programming - Week 04 Problem Solving Paradigms (Dynamic Programming 1)

Supporting Material - Competitive Programming:

<http://sites.google.com/site/stevenhalim/home/material>



[5] Horowitz, E., Sahni, S. Rajasekaran, S.

Computer Algorithms.

Computer Science Press, 1998.



[6] Munari Junior, Pedro Augusto

Paradigmas e Técnicas de Projeto de Algoritmos. SCE-181

Introdução à Ciência da Computação II.

Slides. Ciência de Computação. ICMC/USP, 2007.

Referências III



[7] Skiena, Steven S.; Revilla, Miguel A.
Programming Challenges - The Programming Contest Training Manual - chapter 11.
Springer, 2003.



[8] Ziviani, Nivio
Projeto de Algoritmos - com implementações em Java e C++.
Thomson, 2007.