maratona de programação

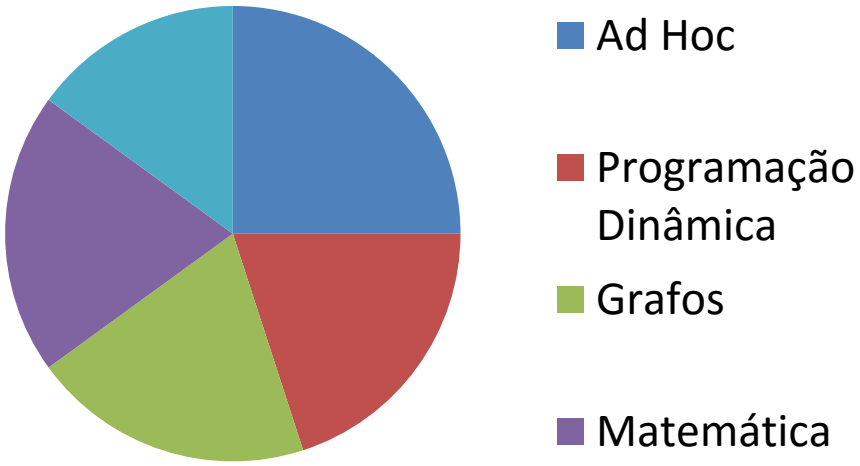
UNIFEI

8 – Paradigmas III

Programação Dinâmica

Prof. **João Paulo** R. R. Leite
joaopaulo@**unifei**.edu.br
ECOE44 & ECOE45 – Maratona de Programação
sites.google.com/site/unifeimaratona/

Uma importante parte dos problemas em uma competição pode ser resolvida através da Programação Dinâmica – Que chamaremos de PD daqui em diante, para simplificar.



Conhecer bem este paradigma de programação – talvez o mais difícil deles – irá aumentar significativamente sua habilidade e suas chances em uma competição

Uma PD é uma técnica de programação, normalmente baseada em uma **fórmula de recursão** e **alguns estados iniciais**.

Nela, a solução de um problema é construída a partir de outras sub-soluções computadas anteriormente.

Soluções com PD possuem complexidade polinomial, garantindo um **desempenho muito melhor** do que soluções "diretas" como backtracking e força-bruta, que, muitas vezes, executam em tempo exponencial.

Um algoritmo que utilize Programação Dinâmica **resolve cada sub-problema uma única vez**.

Grava seu resultado em uma tabela, de maneira que não é necessário resolvê-lo novamente toda vez que houver uma instância idêntica daquele subproblema.

Portanto:

PD é um nome bonito para recursão utilizando uma tabela. Ao invés de resolver subproblemas recursivamente, iremos resolvê-los sequencialmente, guardando suas soluções em uma tabela (um vetor ou matriz).

O truque é **resolver os problemas na ordem correta**. Assim, sempre que uma solução para um subproblema é necessária, ela já se encontrará na tabela.

> **Calcularemos cada solução pela primeira vez** e a armazenaremos em uma tabela, para uso futuro, ao invés de recalcular a mesma coisa recursivamente cada vez que necessário.

O **desempenho** do algoritmo será, em geral, **proporcional ao tamanho da tabela** (linear para um vetor, quadrático para uma matriz, etc.).

O termo “programação” de Programação Dinâmica tem muito pouco a ver com escrever código.

Foi cunhado por **Richard Bellman** nos anos 50, quando a programação de computadores era algo tão místico e esotérico que mal tinha um nome.

Naquela época, programação significava “**planejamento**”.

Nesse caso, “Programação” refere-se à construção da tabela que armazena as soluções das subinstâncias, o planejamento “**otimizado**” para um processo multiestágio.

Why "Dynamic Programming"?

"The 1950s were not good years for mathematical research. We had a very interesting gentleman in Washington named Wilson. He was Secretary of Defense, and he actually had a pathological fear and hatred of the word, research. I'm not using the term lightly; I'm using it precisely. His face would suffuse, he would turn red, and he would get violent if people used the term, research, in his presence. You can imagine how he felt, then, about the term, mathematical. The RAND Corporation was employed by the Air Force, and the Air Force had Wilson as its boss, essentially. Hence, I felt I had to do something to shield Wilson and the Air Force from the fact that I was really doing mathematics inside the RAND Corporation. What title, what name, could I choose? In the first place I was interested in planning, in decision making,... But planning, is not a good word for various reasons. I decided therefore to use the word, "programming"... [Dynamic] has a very interesting property as an adjective, and that is its impossible to use the word, dynamic, in a pejorative sense. Try thinking of some combination that will possibly give it a pejorative meaning. Its impossible. Thus, I thought dynamic programming was a good name. It was something not even a Congressman could object to. So I used it as an umbrella for my activities."

Richard Bellman, "Eye of the Hurricane: an autobiography" 1984.

Programação Dinâmica é, portanto, sobre:

Planejar

Encontre uma subestrutura ótima para o problema. Como resolvê-lo a partir de instâncias menores do mesmo subproblema (uma fórmula de recorrência).

Eliminar a Redundância

Abordagens “diretas” poderiam recalcular a resposta para um mesmo subproblema dezenas (até milhares) de vezes. É preciso evitar este retrabalho.

Mas o que queremos dizer com retrabalho?

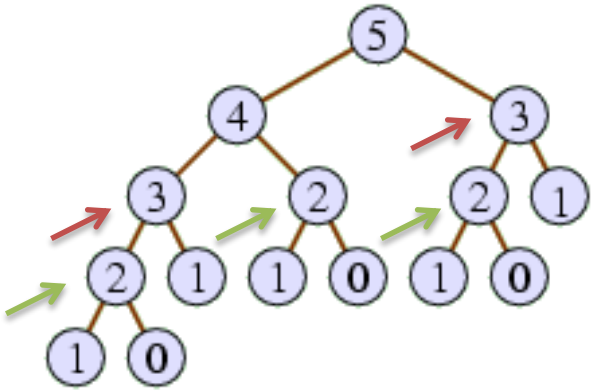
Vamos analisar um caso absolutamente simples de problema onde a programação dinâmica eliminaria um caminhão de redundância: Fibonacci!

Fibonacci é um problema claramente recursivo e sua implementação mais direta seria:

```
int fibonacciRecursiva(int r)
{
    if(r == 0) return 0;
    if(r == 1) return 1;

    return fibonacciRecursiva(r-1) + fibonacciRecursiva(r-2);
}
```

O algoritmo recursivo, apesar de resolver corretamente o problema, possui uma **complexidade no tempo bastante indesejável**.

Repare que cada chamada da função dispara outras duas chamadas, fazendo com que o tempo para a execução do algoritmo aumente exponencialmente com a entrada n, tornando-se **O(2^n)**.



Olhe as setas: repare que grande parte da culpa pelo desempenho exponencial jaz no fato de que muito do **trabalho é refeito** várias vezes.

Agora, utilizemos uma tabela para auxiliar a solução do algoritmo, **planejando de uma maneira mais dinâmica** seu funcionamento:

```
int fibonacciAlt(int r)
{
    if(r == 0) return 0;

    int vet[r+1];
    vet[0] = 0;
    vet[1] = 1;

    for(int i = 2; i <= r; i++)
        vet[i] = vet[i-1] + vet[i-2];

    return vet[r];

}
```

Repare que:
Os resultados intermediários são **armazenados** em uma tabela.

O evento mais custoso da função é um loop interno, que executa apenas **n-1 vezes** (proporcional ao tamanho da tabela).

Tempo **Linear**!! $\Theta(n)$

Cada subproblema é computado uma única vez. **Elimina TODA a redundância**.

Quando devo “ficar esperto” e utilizar a PD?

Em geral, aplicamos a PD a problemas de otimização e contagem.

- Sempre que for pedido para “**maximizar**” ou “**minimizar**” algum valor ou “**contar** as maneiras de se fazer algo”, há uma grande chance de que o problema seja de PD.
- A maioria dos problemas quer saber **apenas o valor ótimo** (máximo, mínimo, ou a quantidade), mas não está interessado na solução ótima em si (normalmente).

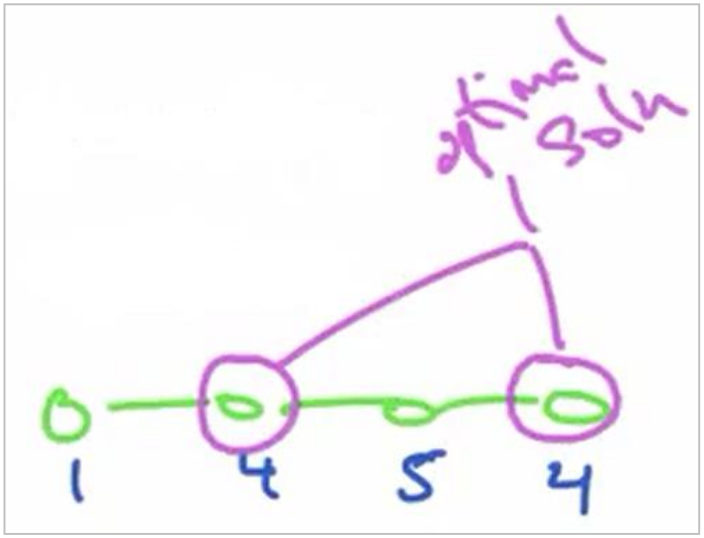
Para utilizá-la, basta que o problema possua:

- Subestrutura ótima:
  Soluções ótimas dos subproblemas **>>** Solução ótima global.
- Sobreposição de problemas:
  Uso de recursão implica em recálculo de subproblemas.

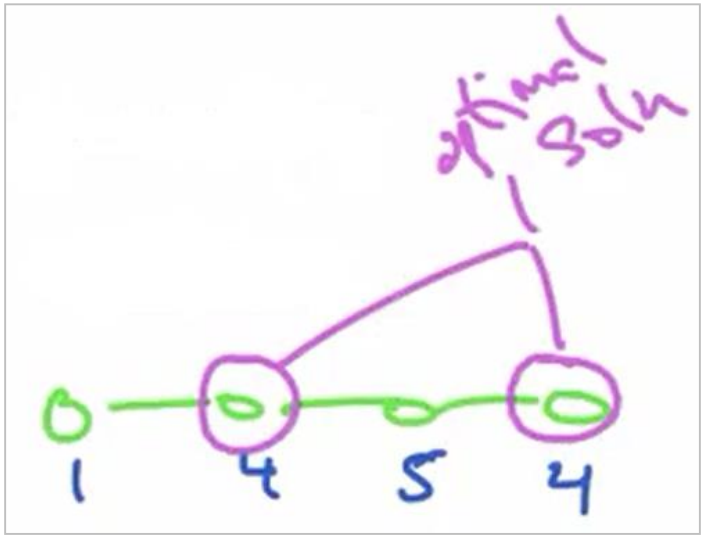
Agora vamos ver o que ocorre em alguns exemplos

Knapsack Problem

Wedding Shopping



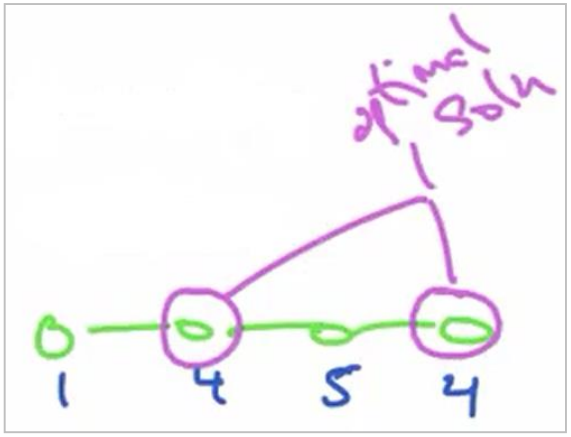
WIS in Path Graphs



Weighted Independent Set in Path Graphs

WIS in Path Graphs

Dado um conjunto de vértices V descrevendo um caminho (path) em um grafo, sendo que cada vértice possui um peso associado a ele, o Máximo Conjunto Independente Ponderado (*Weighted Independent Set*) é o subconjunto de vértices cuja soma dos pesos é máxima (otimização) sem que dois vértices do conjunto sejam adjacentes (e, por isso, são chamados de independentes).



Este problema não é tão comum, mas é um ótimo ilustrativo de Programação Dinâmica

WIS in Path Graphs

Porque utilizar Programação Dinâmica? **TEMPO**!

Um algoritmo de força bruta funcionaria corretamente, mas levaria tempo exponencial em sua execução (cálculo de cada uma das combinações de vértices).

O algoritmo "direto" examinaria cada subconjunto de vértices, checaria se é independente e encontraria o maior deles. Rodaria com tempo $O(n^2\, 2^n)$.

bem horrível

WIS in Path Graphs

Para resolvê-lo com PD, precisamos pensar da seguinte maneira:

Uma solução ótima precisa ter uma de duas propriedades:

1) Último elemento do caminho NÃO é parte do conjunto.

   Portanto, a solução é igualmente válida para o subgrafo V' formado ao retirar o último vértice do caminho.

2) O último elemento É parte do conjunto.

   Nesse caso, sabemos que seu predecessor não pode ser parte do conjunto, e a solução menos o último vértice é válida para o subgrafo V' formado ao se retirar os dois últimos vértices do caminho.

Agora, basta percorrer o caminho, da esquerda para a direita, memorizando em uma tabela o conhecimento do melhor caminho até o momento, decidindo qual das propriedades acima utilizar.

```c
int maximum_weight_set(int *vet, int siz)
{
    int memo[siz+1]; // vetor de PD
    memset(memo, 0, sizeof memo); // Inicialização (opcional)

    // Casos base (como em uma recursão)
    memo[0] = 0;
    memo[1] = vet[0];

    // Apenas percorremos vetor (tempo linear)
    for(int i = 2; i <= siz; i++)
    {
        // Testamos qual das uas opções seguir
        if(memo[i-1] > memo[i-2]+vet[i-1])
            memo[i] = memo[i-1];
        else
            memo[i] = memo[i-2]+vet[i-1];
    }

    return memo[siz];
}
```

The Infamous Knapsack Problem

Problema da Mochila (*Knapsack problem*)

Durante um roubo, o ladrão encontra muito mais objetos do que esperava, e tem que decidir o que levar.

- Sua mochila pode carregar um peso total de no máximo **W quilos**.
- Existem *n* itens entre os quais escolher, de pesos $\mathbf{w_1}$, $\mathbf{w_2}$, ..., $\mathbf{w_n}$ e valor, em reais, $\mathbf{v_1}$, $\mathbf{v_2}$, ..., $\mathbf{v_n}$.
- Qual a **combinação mais valiosa** (otimização!) de itens que ele pode colocar na sua mochila?

Este problema poderá aparecer em muitas ocasiões, mas substituirá peso, mochila e valor por outras grandezas.

Utilizemos o seguinte exemplo:

Item	Peso	Valor
1	6	R$ 30
2	3	R$ 14
3	4	R$ 16
4	2	R$ 9

Existem **duas versões desse problema**:

Se existirem quantidades ilimitadas de cada item, a escolha ótima seria selecionar o item 1 e dois do item 4 (total: R$ 48). Caso só haja um item de cada, a solução ótima contém os itens 1 e 3 (total: R$ 46) .

Problema da Mochila com repetição

Quais são os subproblemas?

Podemos definir K(w) como o valor máximo alcançável por uma mochila de capacidade w.

Caso a solução ótima K(w) inclua o item i, ao removermos esse item da mochila, teremos a solução ótima para $K(w - w_i)$ e, portanto:

$$\mathbf{K(w) = K(w - w_i) + v_i}$$

Vejamos o **algoritmo**, implementado em **C++**, que podemos tirar dessa conclusão.

```
int mochila_repeat(int* pesos, int* valores)
{
    // Valor máximo alcançavel por uma mochila de w quilos
    int k[PESO_MAX + 1];

    // Inicializa pesos
    for(int i = 0; i < PESO_MAX+1; i++)
        k[i] = 0;

    // Verifica para cada peso
    for(int w = 1; w < PESO_MAX+1; w++)
    {
        for(int j = 0; j < QUANT_ITENS; j++)
        {
            if(pesos[j] <= w)
            {
                if(k[w - pesos[j]] + valores[j] > k[w])
                    k[w] = k[w - pesos[j]] + valores[j];
            }
        }
    }

    cout << "Matriz de PD" << endl;
    for(int i = 0; i < PESO_MAX+1; i++)
        cout << k[i] << " ";
    cout << endl << endl;

    return k[PESO_MAX];
}
```

A parte final apenas imprime o vetor de PD, para melhor entendimento.

Problema da Mochila com repetição

Esse algoritmo preenche uma tabela unidimensional de comprimento W + 1, da esquerda para a direita.

Cada célula pode tomar tempo até O(n) para ser computada, portanto o tempo de execução é igual a **O(nW)**.

Muito mais rápido que a força bruta, onde seria necessário computar o resultado para cada combinação de objetos, pesos e valores.

Problema da Mochila sem repetição

Quais são os subproblemas?

Podemos definir K(w, j) como o valor máximo alcançável por uma mochila de capacidade w e itens 1, ..., j. Portanto, a resposta que procuramos é K(W, n).

Nesse caso, ou o item j é necessário para alcançar um valor ótimo ou não é:

$$K(w, j) = \max\{ K(w-w_j, j-1) + v_j, K(w, j-1) \}$$

O algoritmo, portanto, consiste em preencher uma tabela bidimensional, com W+1 linhas e n+1 colunas. Cada célula da tabela toma tempo apenas constante, assim, muito embora ela seja muito maior que no caso anterior, o tempo de execução permanece o mesmo, **O(nW)**. Veja o código:

```
int mochila_norepeat(int* pesos, int* valores)
{
    int k[PESO_MAX+1][QUANT_ITENS+1];

    // Inicializa matriz de PD
    for(int w = 0; w <= PESO_MAX; w++)
        k[w][0] = 0;
    for(int j = 0; j <= QUANT_ITENS; j++)
        k[0][j] = 0;

    // Percorrendo
    // Como a matriz começa a ter dados em 1, precisamos subtrair 1
    // quando utilizarmos os vetores pesos e valores
    for(int j = 1; j <= QUANT_ITENS; j++)
    {
        for(int w = 1; w <= PESO_MAX; w++)
        {
            if(pesos[j-1] > w)
                k[w][j] = k[w][j-1];
            else
            {
                if((k[w - pesos[j-1]][j - 1] + valores[j - 1]) > k[w][j-1])
                    k[w][j] = k[w - pesos[j-1]][j - 1] + valores[j - 1];
                else
                    k[w][j] = k[w][j-1];
            }
        }
    }

    return k[PESO_MAX][QUANT_ITENS];
}
```

Wedding Shopping

UVa Online Judge

11450 Wedding Shopping

One of our best friends is getting married and we all are nervous because he is the first of us who is doing something similar. In fact, we have never assisted to a wedding, so we have no clothes or accessories, and to solve the problem we are going to a famous department store of our city to buy all we need: a shirt, a belt, some shoes, a tie, etcetera.

Patricia Smith and **José Antonio Sánchez**
request the pleasure of
the company of
Mr and Mrs James and Sarah Student
at their wedding,
at St Mary's Church, Espinardo
on Saturday, May 17th, 2008 at 9 o'clock
and afterwards at
Cantina Hall, CSU.
RSVP
Universitary Campus, Espinardo, WT8 4EG

We are offered different models for each class of garment (for example, three shirts, two belts, four shoes, ...). We have to buy one model of each class of garment, and just one.

As our budget is limited, we cannot spend more money than it, but we want to spend the maximum possible. It's possible that we cannot buy one model of each class of garment due to the short amount of money we have.

Input

The first line of the input contains an integer, N, indicating the number of test cases. For each test case, some lines appear, the first one contains two integers, M and C, separated by blanks ($1 \leq M \leq 200$, and $1 \leq C \leq 20$), where M is the available amount of money and C is the number of garments you have to buy. Following this line, there are C lines, each one with some integers separated by blanks; in each of these lines the first integer, K ($1 \leq K \leq 20$), indicates the number of different models for each garment and it is followed by K integers indicating the price of each model of that garment.

Output

For each test case, the output should consist of one integer indicating the maximum amount of money necessary to buy one element of each garment without exceeding the initial amount of money. If there is no solution, you must print '`no solution`'.

Como podemos verificar no livro "*Competitive Programming*", para que consigamos resolver este problema, é necessário utilizar PD.

Algoritmo Guloso: Wrong Answer.
Divisão e Conquista: Wrong Answer.
Busca Completa (*Recursive Backtracking*): Time Limit Exceeded.
Correta, mas muito lenta. No pior caso, 20^{20} operações!

Wedding Shopping

O conjunto de parâmetros necessários para compor um determinado estado do meu problema são: a quantidade de dinheiro restante e a peça de roupa atual.

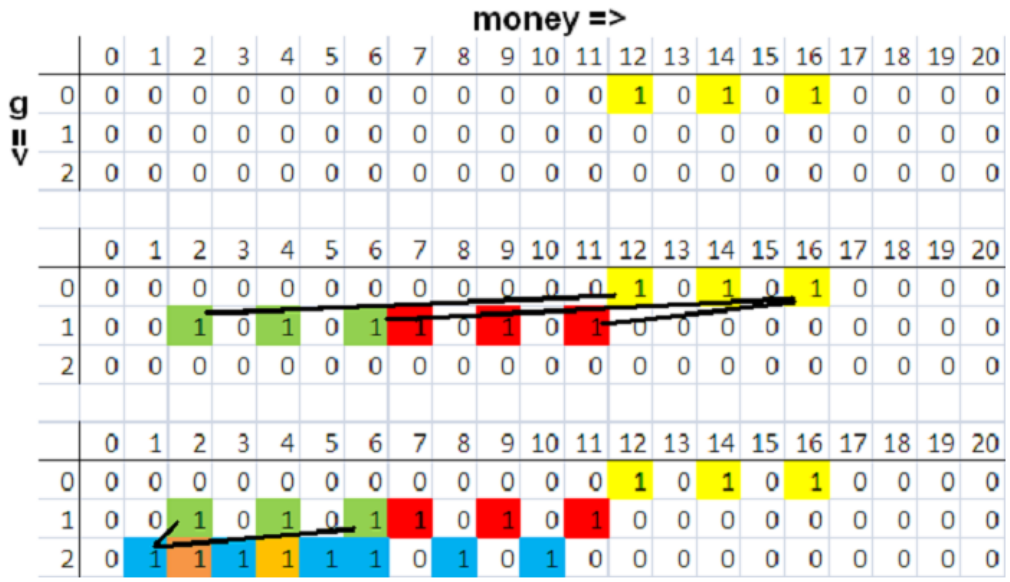
Portanto, para guardar os estados é preciso utilizar uma matriz bidimensional, onde as linhas representam cada estado possível para o dinheiro restante (200 linhas) e as colunas representam cada peça de roupa necessária (20).

A esta matriz, nós chamaremos “can_reach” e será preenchida inicialmente com “false” para todos os seus membros.

No decorrer do algoritmo, todos os estados em se se é possível chegar serão marcados com true. Assim, como obtemos a resposta final?

```
int main()
{
    int n, max_money;

    scanf("%d", &n);
    while(n--)
    {
        scanf("%d %d", &m, &c); // quantidade de dinheiro e de itens

        for(int i = 0; i < c; i++)
        {
            scanf("%d", &prices[i][0]); // quantidade de itens desse tipo
            for(int j = 1; j <= prices[i][0]; j++)
                scanf("%d", &prices[i][j]); // preço de cada
        }

        memset(can_reach, false, sizeof can_reach); // inicialmente nao sabemos nada...
        max_money = shop(m, c);

        if(max_money > 0)
            printf("%d\n", max_money);
        else
            printf("no solution\n");
    }

    return 0;
}
```

```
int shop(int money, int clothes)
{
    for(int i = 1; i <= prices[0][0]; i++)
        can_reach[money-prices[0][i]][0] = true; // inicializa com primeiros valores

    for(int j = 1; j < clothes; j++) // para cada tipo de roupa restante
        for(int i = 0; i < money; i++)
            if(can_reach[i][j-1] == true) // se for alcançavel no nivel anterior
                for(int l = 1; l <= prices[j][0]; l++)
                    if(i - prices[j][l] >= 0) // se não ultrapassar o dinheiro total
                        can_reach[i-prices[j][l]][j] = true;

    for(int i = 0; i < money; i++)
        if(can_reach[i][clothes-1] == true)
            return (m-i); // verifica qual o maximo alcançado na ultima coluna e retorna a diferença

    return -1;
}
```



Para a seguinte entrada:

```
20 3
3 4 6 8
2 5 10
4 1 3 5 5
```

Finalizando...

Programação dinâmica não é fácil.
Requer muito estudo e prática. Os primeiros problemas demorarão a sair, mas não desanime.

Logo os problemas elementares se tornarão muito fáceis e você estará procurando outros problemas mais difíceis para resolver.