# Introdução a Programação Dinâmica

Professor Gabriel Tostes

# O que é?

- É um técnica de otimização combinatória. Baseia-se na ideia de que, ao calcular recursivamente a resposta de um problema, alguns estados anteriores podem coincidir. Com isso, memorizamos as respostas desses estados a fim de não calcularmos eles mais de uma vez na chamada recursiva.
- Desse modo, resolver um problema com Programação Dinâmica (PD) é definir estados para sua contagem e conseguir fazer a transição de estados anteriores para estados mais complexos.

# Como fazer?

1: Caracterize a estrutura de uma solução ideal.

2: Defina recursivamente o valor de uma solução ideal.

3: Calcule o valor e memorize-o.

4: Construa a solução ideal a partir de informações computadas.

# Exemplo

Um exemplo de programação dinâmica é no cálculo recursivo do binomial de Newton:

Sabemos que $\binom{n}{k} = \binom{n-1}{k} + \binom{n-1}{k-1}$ e que $\binom{n}{k}$ = 1 se k=0.

Então, vemos que tanto a chamada recursiva para (n-1,k) quanto (n-1,k-1) tem o binomial (n-2,k-1). A ideia é memorizar a resposta para esse binomial e só calcular o valor dele apenas uma vez, assim como para todos binomiais que aparecem mais de uma vez.

# Exemplo - Análise de Complexidade

Para o caso sem a otimização da programação dinâmica:

- Se T(n,k) for a complexidade, pela lei recursiva o tempo usado para calcular T(n,k) é T(n-1,k) + T(n-1,k-1), que obedece a mesma lei recursiva do binomial e, portanto, a complexidade é exponencial : O(c(n,k)).
- Se usarmos a otimização da PD, para o cálculo do valor de c(n,k) usamos apenas valores já calculados de c(a,b) com a<=n e b<=k. Temos, portanto, que calcular no máximo (n+1)*(k+1) valores anteriores, e a complexidade é O(n*k).

# Exemplo - Implementação.

Aqui segue a implementação das duas maneiras de calcular o binomial, junto com a saída do programa e o tempo gasto para esse cálculo. Repare que na implementação com PD memorizamos os valores já calculados de C(a,b) em memo[a][b]. A contagem de tempo é em segundos.

```cpp
#include<bits/stdc++.h>
using namespace std;

long long c(int a, int b){
    if(b==0) return 1;
    if(a==0) return 0;
    return c(a-1,b-1) + c(a-1,b);
}
int main(){
    cout << c(30,12) << "\n";
    cout << "contagem de tempo: ";
    cout << (double) clock()/CLOCKS_PER_SEC;
}
```

C:\Users\Primetek\Documents\CS\1.exe

```
86493225
contagem de tempo: 0.851
--------------------------------
```

```cpp
#include<bits/stdc++.h>
using namespace std;
long long memo[100][50];

long long c(int a, int b){
    if(b==0) return 1;
    if(a==0) return 0;
    if(memo[a][b]!=0) return memo[a][b];
    memo[a][b]=c(a-1,b-1) + c(a-1, b);
    return c(a-1,b-1) + c(a-1,b);
}
int main(){
    cout << c(30,12) << "\n";
    cout << "contagem de tempo: ";
    cout << (double) clock()/CLOCKS_PER_SEC;
}
```

C:\Users\Primetek\Documents\CS\1.exe

```
86493225
contagem de tempo: 0.031
--------------------------------
```

# PD's clássicas.

Existem alguns problemas clássicos que são resolvidos com Programação Dinâmica. Vamos falar de 3 deles nessa aula:

- **Problema da Mochila ( Knapsack problem) :** Se temos uma mochila que aguenta um peso máximo (P) e objetos a serem colocados dentro dela que pesam W e valem V. Qual é o valor máximo que podemos colocar dentro da mochila tal que a soma dos pesos do objeto é <= P ?
- **Problema do troco:** Se temos moedas de valor $a_1$, $a_2$, ... , $a_n$ e queremos retornar um valor V de troco, qual é o menor número de moedas que utilizamos?
- **Kadane:** Se temos uma sequência $a_1$, $a_2$, ... , $a_n$ com valores positivos ou negativos. Qual é o máximo valor que a soma dos elementos de uma subsequência $a_i$, $a_{i+1}$, ... , $a_j$ pode ter?

# PD's clássicas - Problema da Mochila

Entrada:

Leia um valor P para o peso máximo que uma mochila aguenta. E um valor N para o número de objetos disponíveis. Em seguida leia N linhas, na i-ésima delas teremos dois números: $w_i$ e $v_i$, referentes ao peso(w) e ao valor(v) do objeto i.

Saída:

Retorne o valor máximo que pode estar contido dentro da mochila, sem que o peso dos objetos dentro dela seja maior que P.

# PD’s clássicas - Problema da Mochila

- Solução trivial: Teste todas as $2^n$ combinações de objetos dentro e fora da mochila. Dessas combinações a que tiver peso total <= P e maior valor, retorne o valor dela.
- Solução com PD: Vamos definir um jeito de colocar objetos dentro da mochila: colocamos um por um e para cada valor w<=P guardamos uma memorização Knap[i][w] que recebe o máximo valor possível de se colocar na mochila se escolhermos alguns objetos de 1 até i cuja soma dos pesos deles é w. A resposta vai ser $\max_{1 =< w <= P}$ (Knap[n][w]). Porque isso otimiza? Digamos que os 3 primeiros objetos são {5,3}, {1,3} e {4,1} (na forma {w,v}). Então se pegarmos o objeto 1 ou o objeto e 2 e 3 temos o mesmo estado: Knap[3][5]=4. (O máximo vem do caso de pegar o objeto 2 e 3). Vemos então que “descartamos” pegar o objeto 1.

# PD’s clássicas  - Problema da Mochila

Agora, já fizemos o passo 1 da PD de caracterizar a estrutura da solução (definimos os estados que vão ser calculados da PD e como vamos obter a resposta a partir disso). Vamos agora ver como se resolve recursivamente o problema e como obter a resposta de Knap[i][w] a partir de valores anteriores. Repare que, se já calculamos todos Knap[a][b] com a<i. Então, ao chegar na fase i, temos duas possibilidades:

- não colocamos o objeto i: Então atualizamos todos os valores de Knap[i][w] iguais a Knap[i-1][w].
- colocamos o objeto i: Se colocamos o objeto i, o máximo valor será Knap[i-1][w-$w_i$]+$v_i$

# PD's clássicas - Problema da Mochila

A recursão fica da seguinte maneira então

Knap[i][w] = max(Knap[i-1][w], Knap[i-1][w-$w_i$]+$v_i$)

Knap[i][w] é o máximo valor entre colocar ou não colocar o objeto i.

O caso base da recursão é Knap[0][w]=0 e Knap[i][0]=0;

Análise complexidade: temos no máximo n*P estados (1<=i<=n e 0<=w<=P).
Então a complexidade é O(nP)

# PD's clássicas - Problema da Mochila (implementação)

```
#include<bits/stdc++.h>
using namespace std;
int P, N;
int w[5000], v[5000];
int Knap[5000][5000];

int KnapSack(){
    for(int i=1; i<=N; i++){
        for(int W=0; W<=P; W++){
            if(W-w[i]>=0) Knap[i][W]=max(Knap[i-1][W], Knap[i-1][W-w[i]]+v[i]);
            else Knap[i][W]=Knap[i-1][W];
            /* chamamos recursivarmente, se W-w[i]<0 não podemos colocar o objeto
            então Knap[i][w]=Knap[i-1][w]*/
        }
    }

    int ans=0;
    for(int W=1; W<=P; W++){
        if(ans<Knap[N][W]) ans=Knap[N][W];//pegamos o máximo de Knap[N][i]
    }
    return ans;
}
int main(){
    cin >> P >> N;
    for(int i=1; i<=N; i++) cin >> w[i] >> v[i];
    cout << KnapSack() << endl;
}
```

Esse código lê o valor máximo P da mochila, o número N de elementos e N valores w_i e v_i. Ele retorna o valor máximo que pode-se colocar numa mochila cujos objetos tenham peso no máx P.

# PD’s clássicas - Problema do Troco

Entrada:

Leia um valor V que será dado de troco e um número N de moedas. Em seguida leia N números $a_1$, $a_2$, … , $a_N$, que são os valores das N moedas.

Saída:

Retorne o menor número de moedas possíveis para entregar um valor V de troco. Ou retorne -1 caso não seja possível somar V com as moedas disponíveis.

# PD’s clássicas - Problema do Troco

Solução gulosa:

Enquanto não chegamos ainda ao valor V de troco, pegamos a maior moeda que não ultrapasse o valor que temos que dar para o troco.

Essa solução resolve para o sistema de moedas do Real: 1,5,10,25,50,100. Mas se as moedas forem 1, 4 e 5, por exemplo. Esse algoritmo retornaria 4 moedas para o valor 8(5+1+1+1), mas há um jeito com menos moedas de entregar o troco: 8=4+4.

# PD’s clássicas - Problema do Troco

Solução com PD:

Vamos definir um estado simples: Troco[v]. O menor número de moedas que precisamos para dar um troco de valor v. A resposta é Troco[V].

Transição: Troco[v] é $\min_{1<=i<=N}$[Troco[v-a[i]]+1]. Ou seja, o menor número de moedas para chegar em um valor v é o mínimo do menor número de moedas para chegar em um valor v-a[i] dentro todos i. Devemos tomar alguns cuidados pois alguns valores podem não ser atingidos qual quaisquer quantidades de moeda e não devem entrar no cálculo desse mínimo. Para cuidar desse caso vamos definir inicialmente que Troco[0]=0 e Troco[v]=-1 para todos outros valores. Se Troco[v-a[i]]==-1 não levamos ele em consideração para o cálculo de Troco[v]

# PD’s clássicas - Problema do Troco

Análise de Complexidade:

Repare que temos que calcular V estados e para cada estados testamos N moedas diferentes. Então a complexidade é O(NV)

# PD’s clássicas - Problema do Troco (implementação)

```
#include<bits/stdc++.h>
using namespace std;

int V, N, a[5000];

int CC(){
    vector < int > Troco(V+1,-1); /* inicializamos um
    vetor com V+1 espaços (0 a V) e todos valor inicialmente
    iguais a 1 */
    Troco[0]=0;
    for(int v=1; v<=V; v++){
        int minimo=2*V; /* inicializamos min com um valor grande,
        maior que qualquer resposta para o problema */
        for(int i=1; i<=N; i++){
            if(Troco[v-a[i]]!=-1 && minimo>Troco[v-a[i]]+1){
                minimo=Troco[v-a[i]]+1;
            }
        }
        if(minimo!=2*V) Troco[v]=minimo; /* so atualizamos o valor
        de Troco[v] se achamos algum Troco[v-a[i]] != -1, caso contrario
        nao é possivel também entregar um valor v em moedas e Troco[v]=-1*/
    }
    return Troco[V];
}

int main(){
    cin >> V >> N;
    for(int i=1; i<=N; i++) cin >> a[i];
    cout << CC() << endl;
}
```

Esse código retorna o menor número de moedas que precisamos usar para chegar em um valor V;

# PD's clássicas - Kadane

Entrada:

Leia um número N. Em seguida leia N números $a_1$, $a_2$, … , $a_N$.

Saída:

Retorne o maior valor possível da soma dos termos de uma subsequência $a_i, a_{i+1}, ..., a_j$.

# PD's clássicas - Kadane

Solução trivial: Escolha i<j entre 1 e N. E calcule a soma $a_i+a_{i+1}+...+ a_j$, retorne o maior valor achado entre todos pares (i,j). A complexidade nesse caso é $n^2$ para o número de escolha de pares, e para cada par devemos percorrer j-i termos, complexidade O(n). Então a complexidade total é $O(n^3)$. Há um jeito de otimizar um pouco a solução trivial que é pré computar Soma[i] = $a_1,a_2,...,a_i$. Então $a_i,a_{i+1},...,a_j$= Soma[j]-Soma[i-1] e a complexidade para achar essa soma é O(1). O algoritmo total teria complexidade então $O(n^2)$. Vamos ver que conseguimos uma complexidade ainda melhor, de O(n), se usarmos PD:

# PD’s clássicas - Kadane

Solução com PD:

Guardamos um vetor Kadane[i] que guarda o maior valor possível da soma de uma subsequência cuja último termo é a[i].

Então a resposta é $\max_{1<=i<=N}$(Kadane[i]).

Transição: Repare que, a maior sequência terminada em i ou vai ser a maior sequência terminada em i-1 + a[i]. Ou ela vai ser apenas a sequência de um termo a[i].

Então: Kadane[i]=max(a[i], a[i]+Kadane[i-1])

# PD’s clássicas - Kadane

Solução com PD:

Otimização de memória:

Não precisamos guardar Kadane[i] para todos os valores de i. Vemos que Kadane[i] só é igual a a[i] se Kadane[i-1]<0. Então, podemos iterar apenas uma variável Kadane, ao passo que guardamos o maior valor já atingido por essa variável e retornamos ela para 0 se Kadane<0 em algum momento.

A seguir veja a implementação de Kadane com e sem memória otimizada:

# PD's clássicas - Kadane (implementação)

Sem otimização:

```
#include<bits/stdc++.h>
using namespace std;

int N, a[5000], Kadane[5000];

int AlgKadane(){
    for(int i=1; i<=N; i++){
        Kadane[i]=max(a[i], a[i]+Kadane[i-1]);
    }
    int maximo=Kadane[1]; // inicializamos com um valor
    // aleatório do vetor Kadane.
    for(int i=1; i<=N; i++){
        if(maximo < Kadane[i]) maximo=Kadane[i];
    }
    return maximo;
}

int main(){
    cin >> N;
    for(int i=1; i<=N; i++) cin >> a[i];
    cout << AlgKadane() << endl;
}
```

Memória otimizada:

```
#include<bits/stdc++.h>
using namespace std;

int N, a[5000];

int AlgKadane(){
    int Kadane=0, maximo=a[1];
    for(int i=1; i<=N; i++){
        Kadane+=a[i];
        if(Kadane>maximo) maximo=Kadane;
        if(Kadane<0) Kadane=0;
    }
    return maximo;
}

int main(){
    cin >> N;
    for(int i=1; i<=N; i++) cin >> a[i];
    cout << AlgKadane() << endl;
}
```

# PROBLEMAS

Seguem problemas de 3 juízes onlines diferentes:

- URI
- SPOJ
- Codeforces

Os problemas do Codeforces são os mais difíceis.

# Problemas (URI)

**Fibonacci, Quantas Chamadas?** https://www.urionlinejudge.com.br/judge/pt/problems/view/1029

**Caixas e Pedras:** https://www.urionlinejudge.com.br/judge/pt/problems/view/1283

**Remendo:** https://www.urionlinejudge.com.br/judge/pt/problems/view/1475

# Problemas (SPOJ)

**AlphaCode:** https://www.spoj.com/problems/ACODE/

**The Double Helix:** https://www.spoj.com/problems/ANARC05B/

**Batman 1:** https://www.spoj.com/problems/BAT1/

# Problemas ( Codeforces)

Os problemas estão em ordem de dificuldade, os mais acima são os mais fáceis:

Gas Pipeline: https://codeforces.com/problemset/problem/1207/C

Cow and Message: https://codeforces.com/problemset/problem/1307/C

Anfisa the Monkey: https://codeforces.com/problemset/problem/44/E

Buns: https://codeforces.com/problemset/problem/106/C