

# Aula 4 · Soluções Gananciosas e Programação Dinâmica

Desafios de Programação

Fernando Kiotheka   Victor Alflen

UFPR

29/06/2022

## Memória e Soma de Prefixos

# Memória

- Memória é um recurso essencial.
- Em muitos algoritmos podemos trocar **tempo** por **espaço**.
- Não precisamos recalcular o que já calculamos antes!
- Problemas recursivos que podem ser divididos em subproblemas geralmente são muito fáceis de serem resolvidos de forma incremental.

## Soma de prefixos

Um problema simples: Dado um vetor, qual a soma de um intervalo  $[a, b)$  do vetor indexado em  $[0, n)$ ?

- Uma consulta apenas: Solução  $\mathcal{O}(n)$  percorrendo todos os elementos no intervalo  $[a, b)$ .
- Várias consultas: Solução  $\mathcal{O}(qn)$ ? É possível fazer melhor?

Podemos utilizar a associatividade da adição e sua operação inversa, a subtração.

$$(x + y) + z = x + (y + z).$$

$$(x + y) - y = x + (y - y) = x.$$

Isso nos permite que com a soma de cada prefixo (intervalo  $[0, n]$ ) do vetor, seja possível obter a soma de qualquer intervalo  $[a, b)$ .

# Organização do nosso vetor de soma de prefixos

Vamos organizar o nosso vetor assim:

$$v := \begin{bmatrix} 0 & 5_1 & 1_2 & 3_3 & 5_4 & 9_5 & 2_6 \end{bmatrix}$$
$$ps := \begin{bmatrix} 0 & 0_1 & 5_2 & 6_3 & 9_4 & 14_5 & 23_6 & 25_7 \end{bmatrix}$$

Para que a gente não precise ficar se preocupando com o caso onde  $a$  é 0, podemos simplesmente subtrair o 0 que já tem no vetor de soma de prefixos.

# Implementação da soma de prefixo

## Código prefixsum.cpp

```
#include <bits/stdc++.h>
using namespace std;
int main() {
    int n; cin >> n; vector<int> v (n);
    for (int& x : v) { cin >> x; }
    vector<int> ps (n+1);
    ps[0] = 0;
    for (int i = 0; i < n; i++)
        ps[i+1] = ps[i] + v[i];
    int a, b; while (cin >> a >> b)
        cout << ps[b] - ps[a] << "\n";
}
```

| Entrada     | Saída |
|-------------|-------|
| 6           | 5     |
| 5 1 3 5 9 2 | 19    |
| 3 4         |       |
| 2 6         |       |

# Implementação da soma de prefixo usando partial\_sum

**Código** partialsum.cpp

```
#include <bits/stdc++.h>
using namespace std;
int main() {
    int n; cin >> n; vector<int> v (n);
    for (int& x : v) { cin >> x; }
    vector<int> ps (n+1);
    ps[0] = 0;
    partial_sum(v.begin(), v.end(), ps.begin()+1);
    int a, b; while (cin >> a >> b)
        cout << ps[b] - ps[a] << "\n";
}
```

| Entrada     | Saída |
|-------------|-------|
| 6           | 5     |
| 5 1 3 5 9 2 | 19    |
| 3 4         |       |
| 2 6         |       |

# Problema do Troco: Algoritmos Gananciosos e Programação Dinâmica



# Soluções gananciosas/gulosas

Um algoritmo guloso/ganancioso constrói a solução de um problema escolhendo sempre o caminho que pareça ser o melhor, isto é, através de ótimos locais.

- Geralmente rápidos.
- Pode ser difícil determinar a estratégia gananciosa.
- Os ótimos locais realmente levam ao ótimo global?

## Problema do Troco (conjunto fixo)

Qual o menor número de moedas do conjunto  $S$  necessárias para obter o valor  $V$ ?

$$S = \{1, 2, 5, 10\}$$

Para  $V = 25$ , por exemplo, podemos usar 3 moedas:  $\{10, 10, 5\}$ .

## Solução gananciosa do Problema do Troco (conjunto fixo)

Acontece que existe uma solução gananciosa para este problema, e é de escolher sempre a maior moeda que não passa do valor desejado (e decrementá-lo com o valor da moeda).

**Intuição:** estamos tirando o máximo possível do valor atual.

# Implementação do Problema do Troco

**Código** change1.cpp

```
#include <bits/stdc++.h>
using namespace std;
int n; vector<int> coins = {10, 5, 2, 1};
int main() {
    while (cin >> n) {
        int ans = 0;
        for (auto c : coins)
            while (c <= n) {
                n -= c;
                ans++;
            }
        cout << ans << '\n';
    } }
```

| Entrada | Saída |
|---------|-------|
| 13      | 3     |
| 27      | 4     |
| 50      | 5     |

## Solução gananciosa do Problema do Troco (conjunto fixo)

Isso não funciona sempre. Tomemos por exemplo  $S = \{1, 3, 4\}$  e  $V = 6$ :

1. Para  $V = 6$ , escolhemos a moeda 4 (`ans++`)
2. Para  $V = 2$ , escolhemos a moeda 1 (`ans++`)
3. Para  $V = 1$ , escolhemos a moeda 1 (`ans++`)

Poderíamos, no entanto, escolher a moeda 3 duas vezes!

## Subproblemas, transições e estados

Observe que quando tiramos a moeda  $x$  do valor  $v$ , estamos procurando a resposta do troco mínimo para o valor  $v - x$ . Isso significa que o nosso estado atual  $troco(v)$  pode ser dado como a função dos estados de  $troco(v - x_1), \dots, troco(v - x_n)$ .

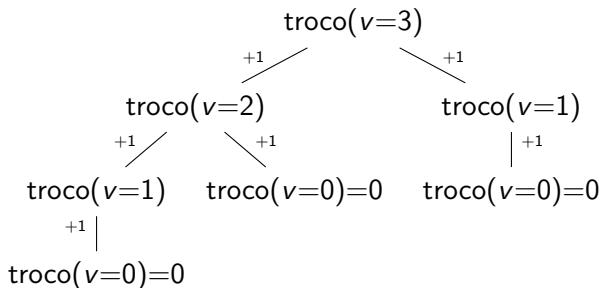
- **Estado** é um problema resolvido usando programação dinâmica. Ele tem um valor definido.
- **Funções de transição** ditam como um estado é mapeado para outros estados, isto é, como um problema é resultado da combinação de outros subproblemas.

O algoritmo se desenvolve como uma máquina de estados.

## Solução de PD para o Problema do Troco (conjunto fixo)

$$troco(v) = \begin{cases} 0 & \text{se } v = 0 \\ 1 + \min\{troco(v - x) : x \in S \wedge x \leq v\} & \text{caso contrário} \end{cases}$$

Para  $S = \{1, 2, 5\}$ :



## Uma nota sobre notação

- Podemos escrever  $10^4$  como `1e4` em C++. Porém é necessário lembrar que isso é um *float*, então as vezes será necessário convertê-lo para inteiro se ficar ambíguo para o compilador assim: `int(1e4)`.
- Outra opção é ir contando a quantidade de zeros, então por exemplo  $10^4$  fica `11234`. Usa um pouco mais de memória, mas vale a pena pra não levar Runtime Error.
- Você pode abusar da memória um pouco. É comum alocar alguns bytes a mais, por exemplo declarar um vetor com `int(1e4+15)` posições, só no caso delas serem úteis.
- E lembre-se que os literais de inteiros no C++ se adequam ao menor tamanho que se encaixa. Se você quer explicitamente um `long long` que caberia num `long`, **precisa** colocar um `LL` no seu literal assim: `1LL`. Fazer um `1 << n` ao invés de `1LL << n` já me custou um problema na competição!



# Uma nota sobre o “infinito”!

- É comum precisarmos de um valor neutro para o mínimo.
- Um valor maior que todos usados no problema pode ser usado.
- Estimar esse valor não é fácil, e é fácil errar, sendo preferível simplesmente colocar um valor “grande”.
- Os valores máximos dos inteiros,  $2^{31} - 1$ ,  $2^{63} - 1$  podem ser usados mas tem que ter muito cuidado.
- **Propriedade importante:** Somar infinito com infinito é comum, então não pode dar overflow.
- Por esse motivo geralmente se usam esses valores:
  - 987654321 e 0x3f3f3f3f para `int`.
  - 1987654321987654321 e 0x3f3f3f3f3f3f3f3f para `long long`.

# Implementação do Problema do Troco (PD)

**Código** change2.cpp

```
#include <bits/stdc++.h>
using namespace std;
int n; vector<int> coins = {1, 2, 5, 10};
const int oo = 987654321;
int main() {
    vector<int> troco(112, oo);
    troco[0] = 0;
    for (int i = 1; i <= 100; ++i)
        for (auto c : coins)
            if (c <= i)
                troco[i] = min(troco[i], 1+troco[i-c]);
    while (cin >> n) cout << troco[n] << '\n';
}
```

| Entrada | Saída |
|---------|-------|
| 0       | 0     |
| 7       | 2     |
| 100     | 10    |

## Problema da Mochila

## Problema da Mochila Fracionária

A história é de um ladrão de pequenas causas: Você quer roubar doces daquelas lojas que vendem vários tipos de doces diferentes a quilo. São  $N$  ( $1 \leq N \leq 10^4$ ) doces diferentes e a sua mochila suporta apenas  $K$  ( $1 \leq K \leq 10^4$ ) quilos, e você quer, obviamente, roubar as coisas mais valiosas do estabelecimento. Como escolher o que roubar e quanto roubar *otimamente*?

| Doce               | Peso | Valor |
|--------------------|------|-------|
| <i>Marshmallow</i> | 40   | 840   |
| Chocolate          | 30   | 600   |
| Bala de goma       | 10   | 100   |
| Bala de gelatina   | 20   | 400   |

Surpreendentemente, esse é um problema fácil. É só pegar o máximo que pudermos dos itens que tem mais custo-benefício (valor/peso).

# Implementação da Mochila Fracionária

## Código fractional.cpp

```
#include <bits/stdc++.h>
using namespace std;
struct itm { int w, v; string s; };
int main() {
    int n, k; cin >> n >> k;
    vector<itm> m (n);
    for (auto& [w, v, s] : m) { cin >> s >> w >> v; }
    sort(m.rbegin(), m.rend(), [](auto a, auto b) {
        return a.v*b.w < b.v*a.w; /* a.v/a.w < b.v/b.w */ });
    for (auto [w, v, s] : m) {
        cout << min(w, k) << "kg de " << s << "\n";
        k -= w; if (k <= 0) break; }
}
```

| Entrada                                                       | Saída                          |
|---------------------------------------------------------------|--------------------------------|
| 4 53<br>marsh 40 840 choco 30 600<br>gelat 10 100 goma 20 400 | 40kg de marsh<br>13kg de choco |

## Problema da Mochila

Roubar lojas de doces não foi o suficiente porém. A próxima parada é o museu. Mas os  $N$  ( $1 \leq N \leq 10^4$ ) itens do museu só têm valor se forem inteiros, então precisamos escolher quais itens que valem a pena, sabendo que a nossa mochila continuando suportando apenas  $K$  ( $1 \leq K \leq 10^4$ ) quilos.

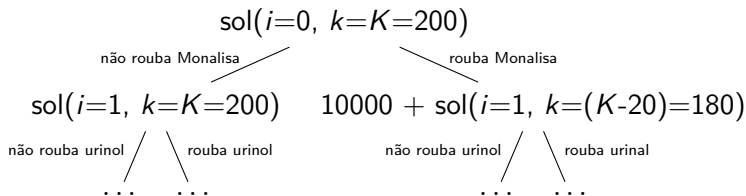
| Obra de Arte            | Peso | Valor |
|-------------------------|------|-------|
| Monalisa                | 20   | 10000 |
| Urinol                  | 30   | 3302  |
| Banana colada na parede | 5    | 1200  |
| Câmera do museu         | 20   | 300   |

Esse problema é... difícil.

## Recursão ao resgate!

| $i$ | Obra de Arte            | Peso | Valor |
|-----|-------------------------|------|-------|
| 0   | Monalisa                | 20   | 10000 |
| 1   | Urinol                  | 30   | 3302  |
| 2   | Banana colada na parede | 5    | 1200  |
| 3   | Câmera do museu         | 20   | 300   |

$$s(i, k) = \begin{cases} 0 & \text{se } i = n \\ \max\{s(i+1, k), v[i] + s(i+1, k - w[i])\} & \text{se } k \geq w[i] \\ s(i+1, k) & \text{se } k < w[i] \end{cases}$$



# Implementação do Problema da Mochila

## Código knapsackrec.cpp

```
#include <bits/stdc++.h>
using namespace std;
struct itm { int w, v; string s; };
int n; vector<itm> m (1e4);
int sol(int i, int k) {
    if (i == n) return 0;
    int best = sol(i+1, k); // skip
    if (k >= m[i].w) // get
        best = max(best, sol(i+1, k-m[i].w) + m[i].v);
    return best; }
int main() {
    int k; cin >> n >> k;
    for (auto& [w, v, s] : m) { cin >> s >> w >> v; }
    cout << "R$" << sol(0, k) << "\n"; }
```

| Entrada                                                           | Saída     |
|-------------------------------------------------------------------|-----------|
| 4 53<br>mona 20 10000 urin 30 3302<br>banana 5 1200 camera 20 300 | R\$ 13302 |



# Alerta de problema exponencial!

- A nossa solução tem complexidade  $\mathcal{O}(2^N)$ .
- Essencialmente iteramos sobre todos os subconjuntos.
- “Tecnicamente”, não tem como fazer melhor, mas a gente pode “roubar” trocando processamento por memória.
- Sabemos que
  - $K$ , o tamanho da mochila limita o parâmetro  $k$  da função.
  - $i$  pertence a  $[0, N)$ .
- Podemos usar uma matriz de tamanho  $N \times K$ , e apenas calcular o valor caso ele nunca tenha sido calculado antes.
- O nome dessa técnica é memoização.
- Nossa complexidade “roubada” é de  $\mathcal{O}(NK)$ .
- O problema da Mochila ainda é NP-Completo. O algoritmo é dito pseudopolinomial:  $\mathcal{O}(N2^{\lg K})$ .

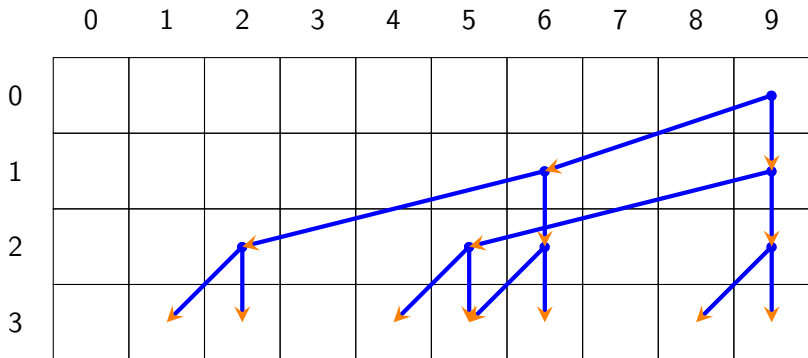
# Implementação do Problema da Mochila com memoização

**Código** knapsackmem.cpp

```
#include <bits/stdc++.h>
using namespace std;
struct itm { int w, v; string s; };
int n; vector<itm> m (1e4+1);
vector<vector<int>> dp (1e4+1, vector<int>(1e4+1, -1));
int sol(int i, int k) {
    if (i == n) { return 0; }
    if (dp[i][k] != -1) { return dp[i][k]; }
    int best = sol(i+1, k); // skip
    if (k >= m[i].w) // get
        best = max(best, sol(i+1, k-m[i].w) + m[i].v);
    return dp[i][k] = best;
}
int main() {
    int k; cin >> n >> k;
    for (auto& [w, v, s] : m) { cin >> s >> w >> v; }
    cout << sol(0, k) << "\n";
}
```

## Dependências...

Uma PD produz naturalmente um grafo de dependências, pois os valores dos problemas dependem dos valores dos subproblemas. A solução recursiva é ótima. É relativamente fácil de se pensar na sua construção e as dependências se resolvem “automaticamente”.



## Matriz... esparsa?

Em muitos problemas, a matriz resultado da PD é esparsa. Isso é importante lembrar porque dependendo do problema, a matriz inteira não cabe em memória, mas ela pode ser guardada de forma esparsa, usando por exemplo um `map`.

|   | 0  | 1  | 2  | 3  | 4  | 5  | 6  | 7  | 8  | 9  |
|---|----|----|----|----|----|----|----|----|----|----|
| 0 | -1 | -1 | -1 | -1 | -1 | -1 | -1 | -1 | -1 | 12 |
| 1 | -1 | -1 | -1 | -1 | -1 | -1 | 8  | -1 | -1 | 10 |
| 2 | -1 | -1 | 5  | -1 | -1 | 7  | 7  | -1 | -1 | 7  |
| 3 | -1 | 0  | 2  | -1 | 2  | 2  | 2  | -1 | 2  | 2  |

## Mas minha solução recursiva é lenta!

- Existe um custo associado ao uso de funções recursivas relacionado a criação e destruição de pilhas de execução.
- Com uma matriz não esparsa, esse custo começa a pesar.
- Seria bom aproveitar da *cache*, acessando posições em sequência na memória.
- Solução: Programação Dinâmica Iterativa.
- O problema: Tem que fazer na ordem correta das dependências, senão o resultado **vai ser errado**.

# Implementação do Problema da Mochila iterativo

## Código knapsackit.cpp

```
#include <bits/stdc++.h>
using namespace std;
int main() {
    int n, k; cin >> n >> k;
    struct itm { int w, v; string s; }; vector<itm> m (n);
    for (auto& [w, v, s] : m) { cin >> s >> w >> v; }
    vector<vector<int>>> dp (1e4+1, vector<int>(1e4+1));
    for (int i = n; i >= 0; i--)
        for (int k = 0; k <= 1e4; k++) {
            if (i == n) { dp[i][k] = 0; continue; }
            int best = dp[i+1][k]; // skip
            if (k >= m[i].w) // get
                best = max(best, dp[i+1][k-m[i].w] + m[i].v);
            dp[i][k] = best;
        }
    cout << dp[0][k] << "\n";
}
```

## Muito bem, mas quais são os itens da mochila mesmo?

- Sabemos responder qual o valor máximo que é possível alcançar com  $N$  itens e mochila com capacidade  $K$ .
- Mas qual seria os itens que atingem o valor máximo?
- Criamos uma matriz de *recuperação* do valor da PD.
- Ideia: Salvar qual o caminho foi tomado para cada chamada para depois reconstruímos o caminho de trás pra frente.
- Lembre-se que muitas vezes múltiplas respostas são possíveis, então precisamos escolher uma.

# Recuperando os itens da mochila

## Código knapsackre.cpp

```
#include <bits/stdc++.h>
using namespace std;
int main() {
    int n, k; cin >> n >> k;
    struct itm { int w, v; string s; }; vector<itm> m (n);
    for (auto& [w, v, s] : m) { cin >> s >> w >> v; }
    vector<vector<int>> dp (1e4+1, vector<int>(1e4+1));
    vector<vector<int>> nxt (1e4+1, vector<int>(1e4+1));
    for (int i = n; i >= 0; i--)
    for (int k = 0; k <= 1e4; k++) {
        if (i == n) { dp[i][k] = 0; continue; }
        nxt[i][k] = k; int best = dp[i+1][k]; // skip
        if (k >= m[i].w) { // get
            int get = dp[i+1][k-m[i].w] + m[i].v;
            if (get > best) { best = get; nxt[i][k] = k-m[i].w; }}
        dp[i][k] = best;
    }
    for (int i = 0; i < n; k = nxt[i][k], i++)
        if (nxt[i][k] != k) cout << m[i].s << "\n";
}
```



## Recuperando os itens da mochila (continuado)

| Entrada                                                           | Saída        |
|-------------------------------------------------------------------|--------------|
| 4 53<br>mona 20 10000 urin 30 3302<br>banana 5 1200 camera 20 300 | mona<br>urin |

## Eu não quero saber dos itens, dá pra reduzir a memória?

- Uma coisa que pode ter chamado a sua atenção é que nessa PD do Problema da Mochila só usamos o estado anterior para calcular o próximo.
- Isso significa que a gente pode usar apenas  $\mathcal{O}(K)$  memória!
- Podemos criar uma matriz  $2K$  e ficar alternando os valores usando  $\sim 1$  por exemplo.
- Porém, lembre que a gente perde a recuperação dos itens, pois a gente não guarda exatamente tudo que a gente precisa pra poder reconstruir o caminho tomado.

# Reduzindo a memória da PD

## Código knapsackred.cpp

```
#include <bits/stdc++.h>
using namespace std;
int main() {
    int n, k; cin >> n >> k;
    struct itm { int w, v; string s; }; vector<itm> m (n);
    for (auto& [w, v, s] : m) { cin >> s >> w >> v; }
    vector<vector<int>> dp (2, vector<int>(1e4+1));
    for (int i = n; i >= 0; i--)
        for (int k = 0; k <= 1e4; k++) {
            if (i == n) { dp[i&1][k] = 0; continue; }
            int best = dp[(i+1)&1][k]; // skip
            if (k >= m[i].w) // get
                best = max(best, dp[(i+1)&1][k-m[i].w] + m[i].v);
            dp[i&1][k] = best;
        }
    cout << dp[0&1][k] << "\n";
}
```

## Outros Problemas em Programação Dinâmica

## Maior subsequência crescente

Outro problema clássico resolvido com PD é o problema da maior subsequência crescente. Também chamado de LIS (Longest Increasing Subsequence), o problema busca encontrar o maior subsequência de números crescentes em um vetor.

$$v := \left[ \begin{array}{cccccccccc} & \text{10} & \text{22} & 9 & \text{33} & 21 & \text{50} & 41 & \text{60} & \text{80} \\ 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \end{array} \right]$$

Para resolver esse problema, podemos pensar em cada posição do vetor e concluir qual a maior subsequência crescente que usa aquela posição, usando as posições anteriores como apoio.

$$lis(n) = \begin{cases} 1 & \text{se } n = 0 \\ \max(\{1\} \cup \{1 + lis(i) : i \in [0, n) \wedge v[i] < v[n]\}) & \text{se } n > 0 \end{cases}$$

# Solução recursiva para o LIS

## Código lisrec.cpp

```
#include <bits/stdc++.h>
using namespace std;
vector<int> v (1e3);
int sol(int n) {
    int mx = 1;
    for (int i = 0; i < n; i++) if (v[i] < v[n])
        mx = max(mx, 1 + sol(i));
    return mx; }
int main() {
    int n; while (cin >> n) { int mx = 0;
        for (int i = 0; i < n; i++) {
            cin >> v[i]; mx = max(mx, sol(i)); }
        cout << mx << "\n";
    }
}
```

| Entrada                     | Saída |
|-----------------------------|-------|
| 9 10 22 9 33 21 50 41 60 80 | 6     |
| 6 4 5 3 2 1 0               | 2     |

## Solução *online*?

- A solução é *bottom-up*, diferente do *top-down* da mochila.
- Isso significa que a gente começa *do começo*, ao invés de começar resolvendo pelo final.
- Uma coisa legal é que isso nos permite ler e calcular o resultado para aquele vetor parcial imediatamente.
- Algoritmos assim são ditos *online*, em contraste com os algoritmos *offline* que precisam ler toda a entrada antes de poder responder o problema.
- Outro algoritmo *online* que você conhece: *Insertion Sort*.
- Os algoritmos *offline* nos permitem “xunxar” de alguns jeitos divertidos, mas a gente vai ver isso só pra frente.
- Porém, algoritmos *online* são sempre mais divertidos, afinal eles constroem a resposta iterativamente.

Ok, essa solução é lenta, cadê a PD?

**Código** lismem.cpp

```
#include <bits/stdc++.h>
using namespace std;
vector<int> v (1e3), dp (1e3);
int sol(int n) {
    if (dp[n] != -1) { return dp[n]; }
    int mx = 1;
    for (int i = 0; i < n; i++) if (v[i] < v[n])
        mx = max(mx, 1 + sol(i));
    return dp[n] = mx;
}
int main() {
    int n; while (cin >> n) {
        fill(dp.begin(), dp.end(), -1); // importante!
        int mx = 0;
        for (int i = 0; i < n; i++) {
            cin >> v[i]; mx = max(mx, sol(i));
        }
        cout << mx << "\n";
    }
}
```



## E a iterativa?

### Código lisit.cpp

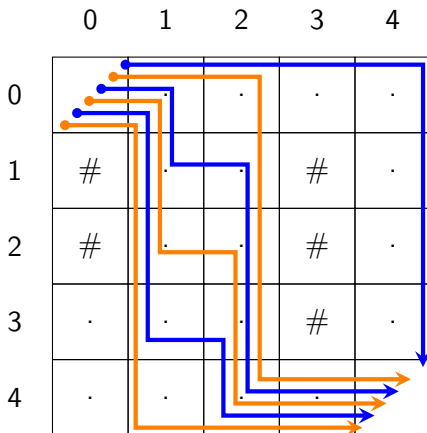
```
#include <bits/stdc++.h>
using namespace std;
vector<int> v (1e3), dp (1e3);
int main() {
    int n; while (cin >> n) {
        fill(dp.begin(), dp.end(), 1); // base
        for (int i = 0; i < n; i++) {
            cin >> v[i];
            for (int j = 0; j < i; j++) if (v[j] < v[i])
                dp[i] = max(dp[i], 1 + dp[j]);
        }
        cout << *max_element(dp.begin(), dp.end()) << "\n";
    }
}
```

Acho que já entendi. Tem mais?

- Acabamos de resolver esse problema em  $\mathcal{O}(n^2)$ .
- Porém existe uma solução em  $\mathcal{O}(n \lg n)$  que também é *online* que veremos mais pra frente no curso!

## Caminhos na Grade

Que tal um problema de contagem 2D? Lhe é dada uma matriz  $n \times n$ , e você precisa descobrir de quantos jeitos diferentes um robô pode ir da posição  $(0, 0)$  até a posição  $(n - 1, n - 1)$  (mod  $10^9 + 7$ ). O robô se move apenas para a direita e para baixo e não pode atravessar as paredes.



# Exemplos

| Entrada                                                                                 | Saída |
|-----------------------------------------------------------------------------------------|-------|
| 5<br>.....<br>#..#.<br>#..#.<br>...#.<br>.....                                          | 6     |
| 9<br>.....<br>.....<br>.....<br>.....<br>....#.....<br>.....<br>.....<br>.....<br>..... | 7970  |

# Uma solução iterativa *top-down*

## Código gridtop.cpp

```
#include <bits/stdc++.h>
using namespace std; using ll = long long;
const int P = 1e9+7;
int main() {
    int n; cin >> n;
    vector<vector<char>> grid (n+2, vector<char>(n+2));
    for (int i = 1; i <= n; i++)
        for (int j = 1; j <= n; j++)
            cin >> grid[i][j];
    vector<vector<ll>> dp (n+2, vector<ll>(n+2, 0));
    // se aproveitamos que tudo é inicializada com zero
    for (int i = n; i >= 1; i--)
        for (int j = n; j >= 1; j--) {
            if (i == n && j == n) { dp[i][j] = 1; continue; }
            if (grid[i+1][j] == '.')
                dp[i][j] = (dp[i][j] + dp[i+1][j]) % P;
            if (grid[i][j+1] == '.')
                dp[i][j] = (dp[i][j] + dp[i][j+1]) % P;
        }
    cout << dp[1][1] << "\n";
}
```

# Uma solução iterativa *bottom-up*

## Código gridbot.cpp

```
#include <bits/stdc++.h>
using namespace std; using ll = long long;
const int P = 1e9+7;
int main() {
    int n; cin >> n;
    vector<vector<char>> grid (n+2, vector<char>(n+2));
    for (int i = 1; i <= n; i++)
        for (int j = 1; j <= n; j++)
            cin >> grid[i][j];
    vector<vector<ll>> dp (n+2, vector<ll>(n+2));
    // se aproveitamos que tudo é inicializada com zero
    for (int i = 1; i <= n; i++)
        for (int j = 1; j <= n; j++) {
            if (i == 1 && j == 1) { dp[i][j] = 1; continue; }
            if (grid[i][j] == '.')
                dp[i][j] = (dp[i-1][j] + dp[i][j-1]) % P;
        }
    cout << dp[n][n] << "\n";
}
```

# Probabilidade!

Comumente resolvemos probabilidades com programação dinâmica. Isto porque geralmente o problema é de probabilidade discreta ao invés de contínua. É comum encontrarmos:

- Problema de achar o valor de esperança (geralmente envolve o conceito de linearidade da esperança).
- Probabilidade de um evento em ponto flutuante (**double**).
- Probabilidade de um evento usando uma fração inteira (usando inverso multiplicativo modular).
- Combinação de eventos condicionais.

## Problema das Moedas

Seja  $N$  ( $1 \leq N \leq 2999$ ) um número positivo ímpar. Existem  $N$  moedas numeradas  $i = 1, 2, \dots, N$ , cada uma com probabilidade  $p_i$  de ser cara, e  $1 - p_i$  de ser coroa. Qual a probabilidade de depois de jogadas as  $N$  moedas, mais moedas sejam cara do que coroa? Imprima a resposta em ponto flutuante. A saída será considerada correta se o erro absoluto não for maior que  $10^{-9}$ .



# Ponto flutuante e suas imprecisões

- Evite ponto flutuante ao máximo.
- Se precisar, sempre use no mínimo, `double`.
- Números em ponto flutuante devem sempre ser somados com valores de tamanho similar para obter menos imprecisão.
- A técnica de janela da aula anterior pode ajudar com isso.
- Problemas que pedem como saída um número em ponto flutuante geralmente toleram um certo erro.
- Imprima várias casas decimais caso exista essa tolerância de erro, ou a quantidade de casas exigida pelo problema.

## As transições do Problema das Moedas

Vamos definir a função  $f(n, h)$  como sendo a probabilidade de que dadas as  $n$  primeiras moedas,  $h$  sejam cara. Alguns valores:

- $f(0, 0) = 1$ .
- $f(1, 0) = (1 - p_1)$ .
- $f(1, 1) = p_1$ .
- $f(1, 2) = 0$ .
- $f(2, 0) = (1 - p_1)(1 - p_2) = f(1, 0) \cdot (1 - p_2)$ .
- $f(2, 1) = (1 - p_1)p_2 + p_1(1 - p_2) = f(1, 0) \cdot p_2 + f(1, 1) \cdot (1 - p_2)$ .
- $f(2, 2) = f(1, 1) \cdot p_2 + f(1, 2) \cdot (1 - p_2) = f(1, 1) \cdot p_2$ .

Genericamente então,

$$f(n, h) = \begin{cases} 1 & \text{se } n = h = 0 \\ 0 & \text{se } h > n \\ f(n-1, 0) \cdot (1 - p_n) & \text{se } h = 0 \\ f(n-1, h-1) \cdot p_n + f(n-1, h) \cdot (1 - p_n) & \text{se } 0 < h \leq n \end{cases}$$

## A solução

- Calcularemos a função  $f(n, h)$ .
- Somaremos a probabilidade do número de caras ser maior que o número de coroas, isto é, todo  $f(n, h)$  com  $h > \lfloor \frac{n}{2} \rfloor + 1$  e com  $n$  sendo a quantidade total de moedas.
- Como somaremos números pequenos com números pequenos, não vamos usar técnicas para minimizar o erro.

# Implementação do Problema das Moedas

**Código** coins.cpp

```
#include <bits/stdc++.h>
using namespace std;
int main() {
    int n; cin >> n;
    vector<double> p (n+1);
    for (int i = 1; i <= n; i++) cin >> p[i];
    vector<vector<double>> f (n+1, vector<double>(n+1));
    f[0][0] = 1.0;
    for (int i = 1; i <= n; i++) {
        f[i][0] = f[i-1][0] * (1.0 - p[i]);
        for (int h = 1; h <= i; h++)
            f[i][h] = f[i-1][h-1] * p[i]
                    + f[i-1][h] * (1.0 - p[i]);
    }
    double ans = 0.0;
    for (int h = n/2+1; h <= n; h++) ans += f[n][h];
    cout << fixed << setprecision(10) << ans << "\n";
}
```

## Implementação do Problema das Moedas (continuado)

| <b>Entrada</b>                | <b>Saída</b> |
|-------------------------------|--------------|
| 3<br>0.30 0.60 0.80           | 0.6120000000 |
| 1<br>0.50                     | 0.5000000000 |
| 5<br>0.42 0.01 0.42 0.99 0.42 | 0.3821815872 |

## E isso é tudo!

- O conteúdo que vocês precisam para fazer a competição que logo começará está todo aqui.
- Vale uma leitura extra: “DP on Broken Profile” ([https://cp-algorithms.com/dynamic\\_programming/profile-dynamics.html](https://cp-algorithms.com/dynamic_programming/profile-dynamics.html)).
- Verificando que o algoritmo ganancioso do problema do troco funciona para um determinado conjunto  $S$ : “A polynomial-time algorithm for the change-making problem” (<https://www.sciencedirect.com/science/article/abs/pii/S0167637704000823>)
- Existem muitas outras técnicas de otimização em programação dinâmica utilizando-se de simetrias geométricas, divisão e conquista, etc.
- Dá pra fazer programação dinâmica em grafos, é bem louco!
- Bons estudos!