



Maratona**CIn**



SELETIVA AULA 2

Buscas e
Backtracking

Lua Guimarães <lgf>
João Luís <jlga>

rev. 1.1



LINKS E OBS



<https://app.codeimage.dev/> – aqui o link do negócio de fazer foto do código. Se tiver dificuldade, me fala que eu te mostro como usar.

LEMBRAR DE DELETAR ESSE SLIDE!!!



Maratona**CIn**



Binary Search

Busca linear



MaratonaCIn

Centro de
Informática
UFPE



Dada uma lista de N números, descubra se o número X está ou não nessa lista.

Uma possibilidade de resolução é procurar linearmente por toda a lista:

- **Melhor caso: O(1)**, o número X é o primeiro da lista.
- **Pior caso: O(N)**, o número X não se encontra na lista.

Como em programação competitiva nos preocupamos **sempre com o pior caso**, tratamos a complexidade dessa busca como **linear**.

- A maioria dos problemas pede para que N buscas sejam realizadas.
- **N × N = N²**. O problema provavelmente não vai passar no tempo limite.

Como melhorar?

Supondo que a **lista esteja ordenada**, é possível realizar a busca de forma muito mais ótima.

Iniciamos procurando no item do meio, caso o item seja maior ou menor do que o número X, descartamos uma metade da lista

A cada iteração, **metade da lista é descartada**.

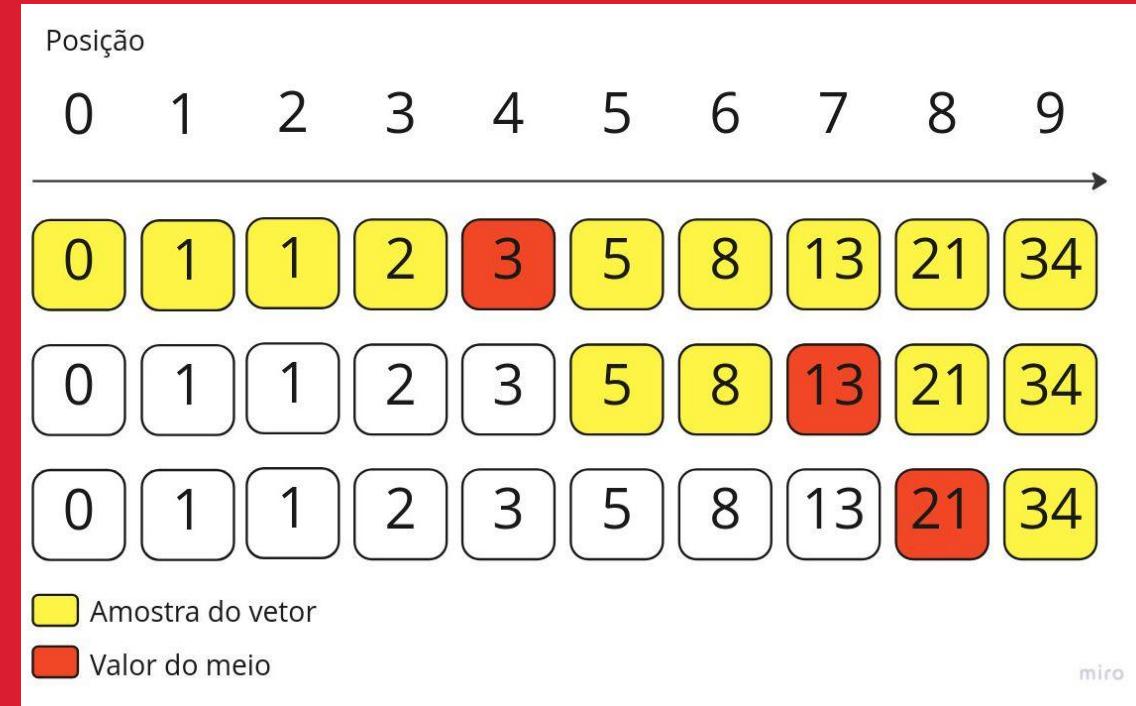
Complexidade: **O(Log(N))**.



MaratonaCIn

Centro de
Informática
UFPE

UNIVERSIDADE
FEDERAL
DE PERNAMBUCO
VIRTUS IMPAVIDA



Busca Binária



MaratonaCIn

Centro de
Informática
UFPE

UNIVERSIDADE
FEDERAL
DE PERNAMBUCO


Isso é o que chamamos de **Busca Binária**:

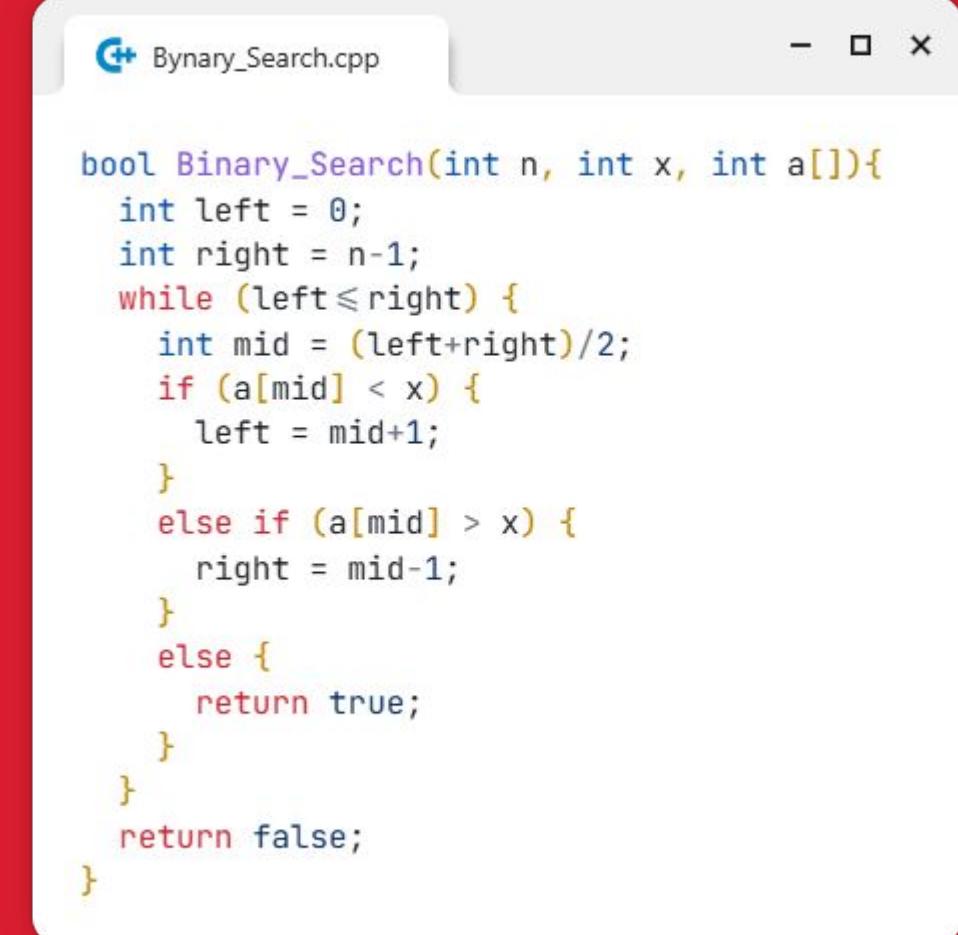
- Primeiro ordenamos a lista com um **Sort: $O(N \times \log(N))$**
- A seguir, separamos a lista em dois intervalos **[Left, Mid]** e **[Mid, Right]**
- Com base no valor de Mid, reduzimos o nosso espaço de busca.
- Caso **Mid < X**, continuamos a busca no intervalo **[Mid+1, Right]**
- Caso **Mid > X**, continuamos a busca no intervalo **[Left, Mid-1]**
- Repetimos o processo até que $\text{Mid} = \text{X}$ ou o intervalo seja inválido.

Código de BS

Com isso, a complexidade do pior caso para realizar N buscas se torna:

- $O(N \times \log(N)) + O(N \times \log(N)) = O(N \times \log(N))$
- Boa complexidade até 5×10^6

A seguir, um código simples de uma função com o Binary Search básico implementado.

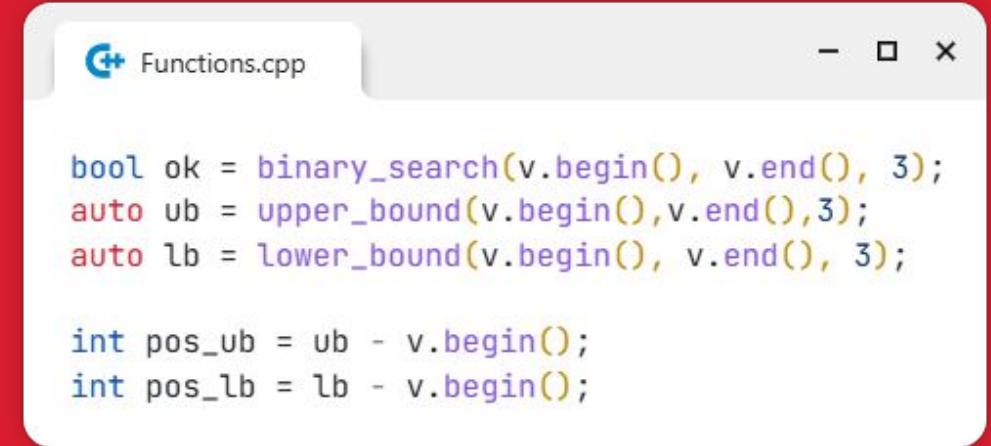


```
bool Binary_Search(int n, int x, int a[]){
    int left = 0;
    int right = n-1;
    while (left <= right) {
        int mid = (left+right)/2;
        if (a[mid] < x) {
            left = mid+1;
        }
        else if (a[mid] > x) {
            right = mid-1;
        }
        else {
            return true;
        }
    }
    return false;
}
```

Funções do C++

Algumas funções com Binary Search já existem dentro de funções da biblioteca padrão de C++.

- A função **binary_search** retorna se o valor existe ou não na lista.
- A função **upper_bound** retorna um ponteiro para o primeiro item $>$ que o valor na lista
- A função **lower_bound** retorna um ponteiro para o primeiro item \geq ao valor na lista
- **É necessário que a lista esteja ordenada.**



```
Functions.cpp

bool ok = binary_search(v.begin(), v.end(), 3);
auto ub = upper_bound(v.begin(), v.end(), 3);
auto lb = lower_bound(v.begin(), v.end(), 3);

int pos_ub = ub - v.begin();
int pos_lb = lb - v.begin();
```

Lower X Upper

Lower Bound (Limite Inferior)

O que faz? Encontra o primeiro elemento que não é menor que o alvo.

Em outras palavras: O primeiro elemento \geq ao alvo.

1	2	3	3	3	4	5
---	---	---	---	---	---	---



**Lower
Bound**

Upper Bound (Limite Superior)

O que faz? Encontra o primeiro elemento estritamente maior que o alvo.

Em outras palavras: O primeiro elemento $>$ ao alvo.

1	2	3	3	3	4	5
---	---	---	---	---	---	---



**Upper
Bound**

Funções Monótonas

Temos uma função monótona e queremos saber qual o **maior x tal que $f(x) < k$.**

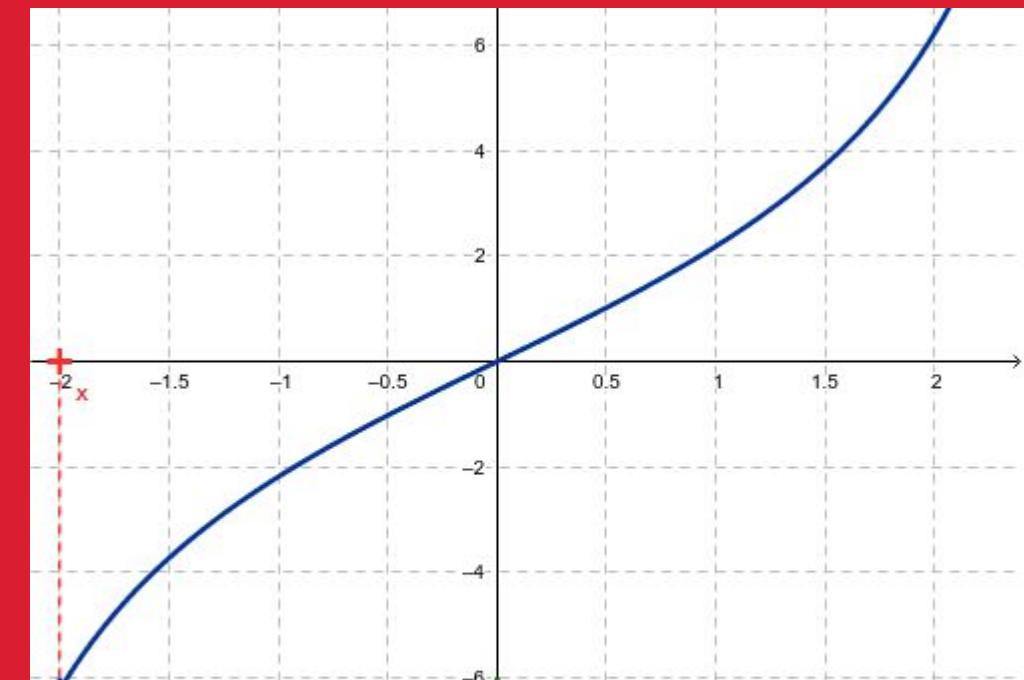
Uma função monótona é uma função cujo valor só cresce ou só decresce conforme o argumento aumenta.

Todo problema que envolve uma função monótona pode ser solucionado com busca binária.

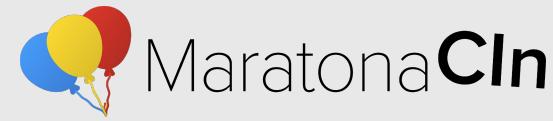


Maratona

CIn Centro de
Informática
UFPE



Factory Machines



A factory has n machines which can be used to make products. Your goal is to make a total of t products.

For each machine, you know the number of seconds it needs to make a single product. The machines can work simultaneously, and you can freely decide their schedule.

What is the shortest time needed to make t products?

Input

The first input line has two integers n and t : the number of machines and products.

The next line has n integers k_1, k_2, \dots, k_n : the time needed to make a product using each machine.

Output

Print one integer: the minimum time needed to make t products.

Constraints

- $1 \leq n \leq 2 \cdot 10^5$
- $1 \leq t \leq 10^9$
- $1 \leq k_i \leq 10^9$

Menor tempo: 1
Maior tempo: 10^{18}

BS na resposta

O problema se resume a descobrir o **menor x tal qual $F(x) \geq t$.**

Como **$F(x)$ é monótona**, podemos transformar ela em um booleano.

Basta pensar que, se é possível construir k brinquedos em t segundos, também é possível construir em $t+1$, $t+2$, $t+3$, etc.



MaratonaCIn

Centro de
Informática
UFPE

UNIVERSIDADE
FEDERAL
DE PERNAMBUCO
 VIRTUS IMPAVIDA

F	F	F	F	T	T	T	T
1	2	3	4	5	6	7	8

Resolução

Setamos “l” para o menor valor possível, e “r” para o máximo.

Caso “f” consiga ser resolvida em **tempo linear**, a complexidade do código se torna **O(N×Log(r))**.

Como $r = 10^{18}$ e $\log(r) = 60$, a complexidade total se torna **60 × 2×10⁵ = 1.2×10⁷**.

O código roda em menos de um segundo!



MaratonaCIn

Centro de
Informática
UFPE

UNIVERSIDADE
FEDERAL
DE PERNAMBUCO
 VIRTUS IMPAVIDA



```
ll l = 1;
ll r = 1e18;
ll ans;
while (l <= r) {
    ll m = (l+r)/2;
    if (f(m,a,t)) {
        ans = m;
        r = m-1;
    }
    else {
        l = m+1;
    }
}
```



MaratonaCIn



UNIVERSIDADE
FEDERAL
DE PERNAMBUCO

Dúvidas?



Maratona**CIn**



Two Pointers

Problema



Dado um array com N inteiros e um valor S, qual o maior segmento contínuo do array com a soma $\leq S$.

Exemplo:

- **N** = 7, **S** = 20
- 2 6 4 3 6 8 9

Como resolver em **tempo linear**?

Two Pointers



MaratonaCIn

Centro de
Informática
UFPE



UNIVERSIDADE
FEDERAL
DE PERNAMBUCO

Iniciamos com dois ponteiros no primeiro elemento e começamos a armazenar a soma e o tamanho de cada intervalo válido.

I

2	6	4	3	6	8	9
---	---	---	---	---	---	---

r

Sum = 2

Ans = 1

Two Pointers



Se a soma for menor do que S, movemos j para a direita e adicionamos v[r] a soma.



Sum = 8
Ans = 2

Two Pointers



Se a soma for menor do que S, movemos j para a direita e adicionamos v[r] a soma.

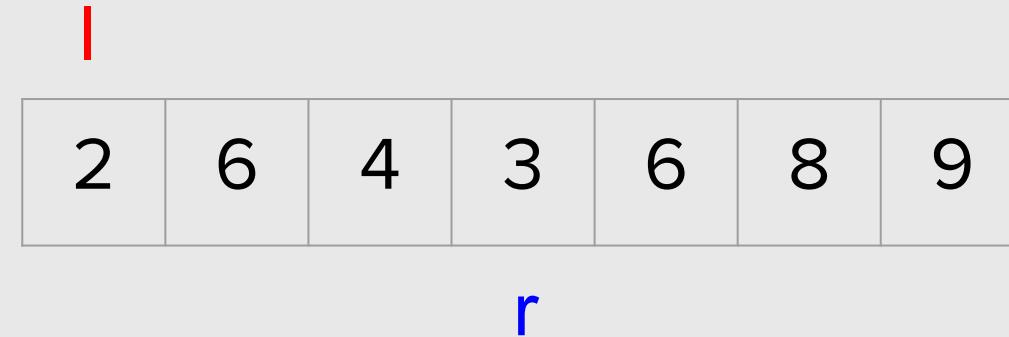


Sum = 12
Ans = 3

Two Pointers



Se a soma for menor do que S, movemos j para a direita e adicionamos v[r] a soma.



Sum = 15
Ans = 4

Two Pointers



MaratonaCIn

Centro de
Informática
UFPE



Já quando a soma supera S, retiramos $v[l]$ da soma e movemos l para a direita.



Sum = 21

Ans = 4

Two Pointers



Já quando a soma supera S, retiramos $v[l]$ da soma e movemos l para a direita.



Sum = 19
Ans = 4

Two Pointers



Continuamos o processo até r sair do array.



Sum = 27
Ans = 4

Two Pointers



Continuamos o processo até r sair do array.

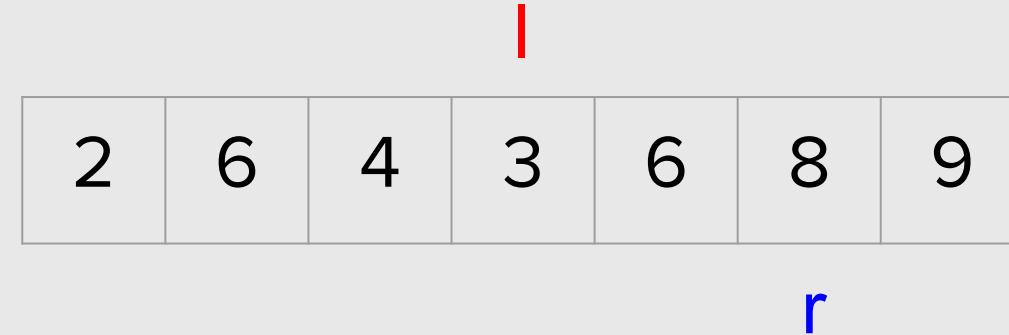


Sum = 21
Ans = 4

Two Pointers



Continuamos o processo até r sair do array.



Sum = 17
Ans = 4

Two Pointers



Continuamos o processo até r sair do array.



Sum = 26
Ans = 4

Two Pointers



Continuamos o processo até r sair do array.



Sum = 23
Ans = 4

Two Pointers



Continuamos o processo até r sair do array.



Sum = 17
Ans = 4

Two Pointers



Continuamos o processo até r sair do array.



r

Sum = 17
Ans = 4

Código



Maratona

CIn

Centro de
Informática
UFPE



UNIVERSIDADE
FEDERAL
DE PERNAMBUCO

```
Two_Pointers.cpp - □ ×

int l = 0;
int sum = 0;
int best = 0;

for (int r=0; r<n; r++) {
    sum += a[r];
    while (sum > s) {
        sum -= a[l];
        l++;
    }
    best = max(best, r-l+1);
}
```



MaratonaCIn



Dúvidas?



Maratona**CIn**



Sweep Line

Problema



Dados N intervalos abertos e Q queries, descubra para cada query em quantos intervalos ela se encontra.

Exemplo:

- **N** = 3
- 1 5, 3 7, 4 10
- **Q** = 3
- 6, 2, 4

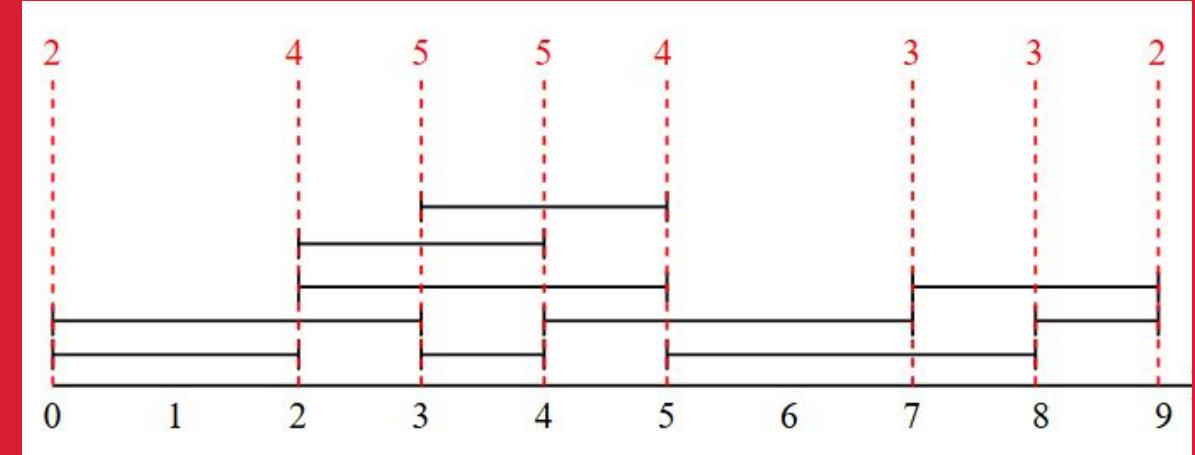
A resolução bruteforce é procurar em todos os intervalos para cada query.

- **Complexidade: O(N×Q)**, N buscas para cada query Q.

Sweep Line

Dados os intervalos, atravessamos todo o range com uma linha vertical, **somando todas as intersecções.**

Então, armazenamos essas intersecções para que possamos obter as respostas das queries de forma muito mais rápida.

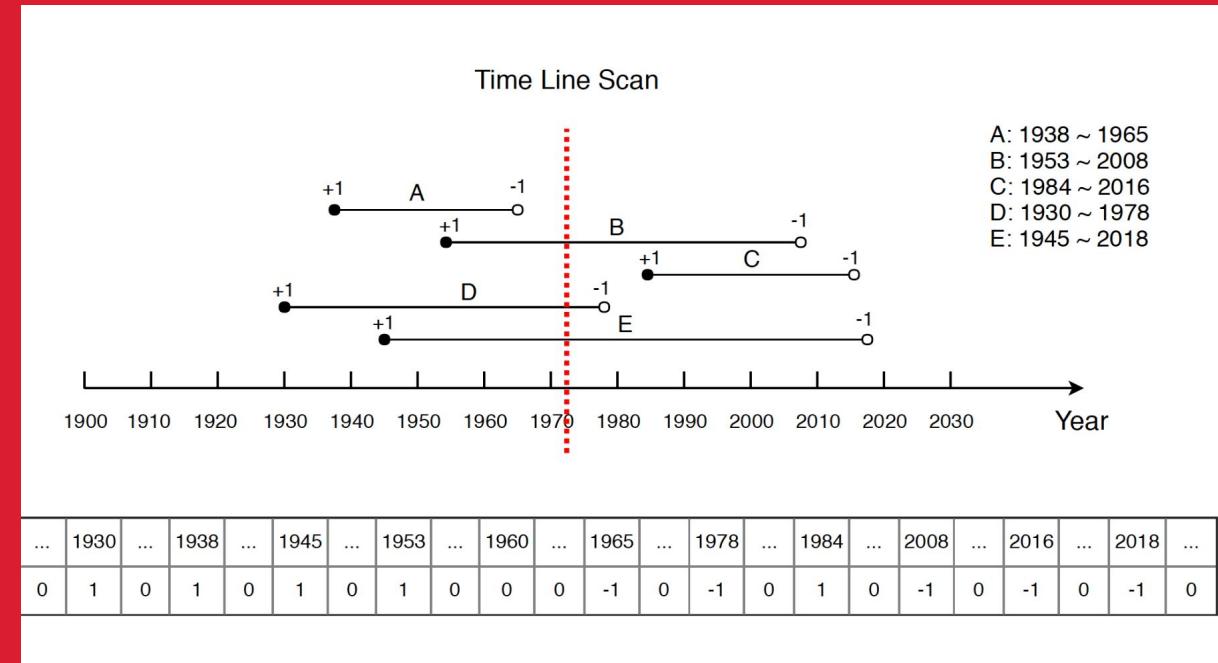


Passo a passo

Primeiro passo: Armazenamos os intervalos, utilizando, por exemplo, um vetor de pairs.

Segundo passo: criamos um vetor onde somamos 1 em cada posição onde um intervalo foi iniciado, e subtraímos 1 em cada posição onde um intervalo foi encerrado.

Qual estrutura utilizamos a seguir?

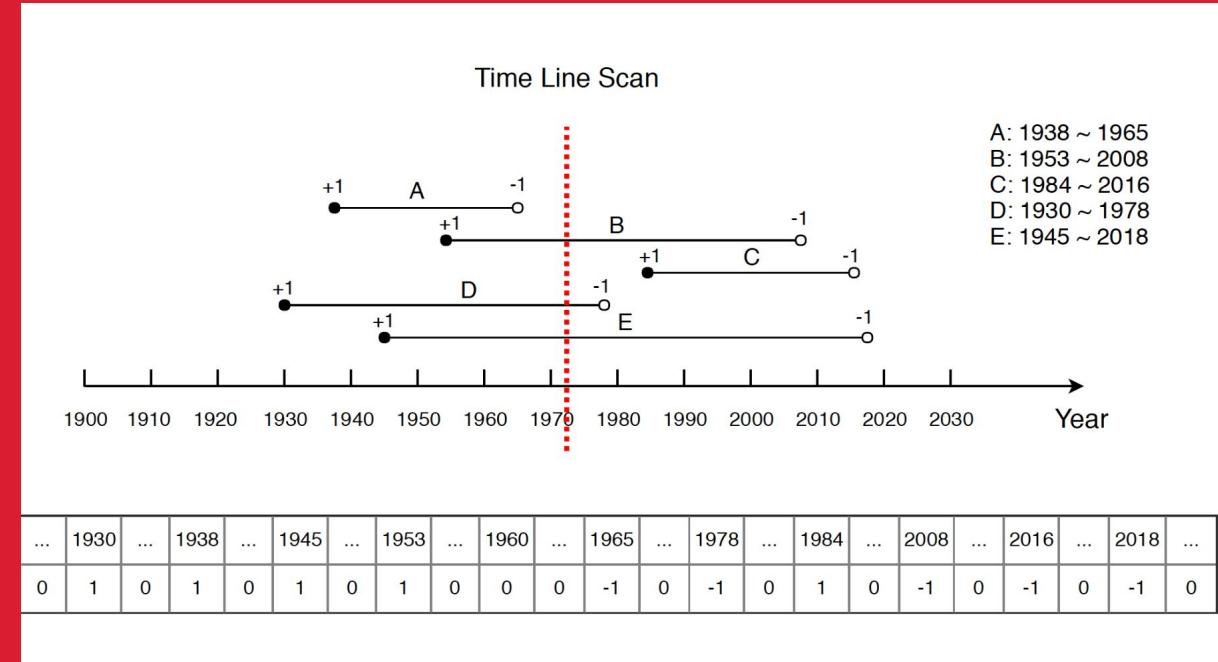


Passo a passo

Primeiro passo: Armazenamos os intervalos, utilizando, por exemplo, um vetor de pairs.

Segundo passo: criamos um vetor onde somamos 1 em cada posição onde um intervalo foi iniciado, e subtraímos 1 em cada posição onde um intervalo foi encerrado.

Qual estrutura utilizamos a seguir? **Prefix Sum**



Passo a passo

Com o vetor de prefix sum conseguimos obter todas as queries em **tempo linear**.

Intervalos do exemplo:

- 1 - 5
- 3 - 7
- 4 - 10

Ao lado, consta como ficaria o **vetor de contagem**, o **prefix sum** e os **índices**.



MaratonaCIn

Centro de
Informática
UFPE



													$m = -1$
1	0	1	1	m	0	m	0	0	m				
0	1	0	2	3	2	2	1	1	1	0			
1	2	3	4	5	6	7	8	9	10				

Passo a passo

Com o vetor de prefix sum conseguimos obter todas as queries em **tempo linear**.

Intervalos do exemplo:

- 1 - 5
- 3 - 7
- 4 - 10

Ao lado, consta como ficaria o **vetor de contagem**, o **prefix sum** e os **índices**.



MaratonaCIn

Centro de
Informática
UFPE



UNIVERSIDADE
FEDERAL
DE PERNAMBUCO

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

Código

Com base no que vimos,
podemos implementar o seguinte
código utilizando dois vetores de
tamanho do **maior número
possível de um dos pontos do
intervalo.**

Entretanto, vetores só
conseguem armazenar por volta
de **1 milhão de números.**

O que fazer se **MAXN = 1×10^9** ?

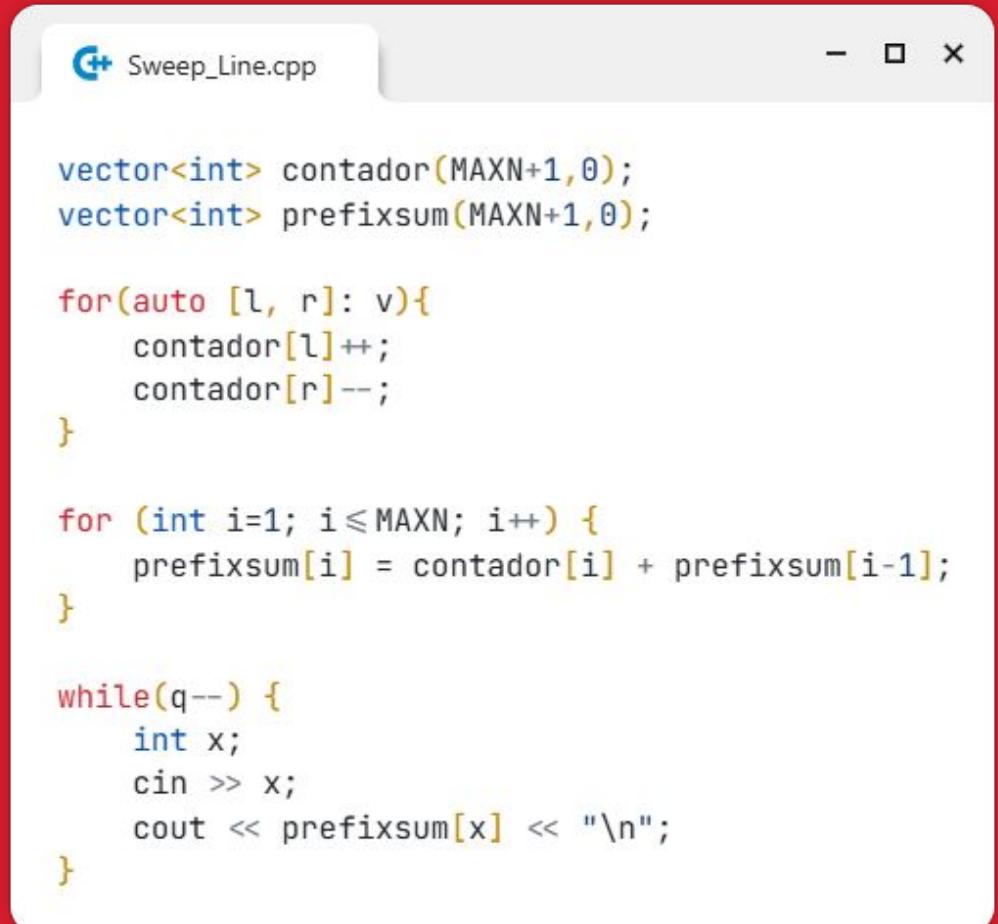


Maratona

CIn

Centro de
Informática
UFPE

UNIVERSIDADE
FEDERAL
DE PERNAMBUCO
 VIRTUS IMPAVIDA



```
vector<int> contador(MAXN+1, 0);
vector<int> prefixsum(MAXN+1, 0);

for(auto [l, r]: v){
    contador[l]++;
    contador[r]--;
}

for (int i=1; i≤MAXN; i++) {
    prefixsum[i] = contador[i] + prefixsum[i-1];
}

while(q--) {
    int x;
    cin >> x;
    cout << prefixsum[x] << "\n";
}
```

Compressão de Coordenadas



Em dada questão, os valores dos intervalos podem chegar até 10^9 , **mas sua quantidade chegará somente até 10^5 ou 10^6** . Com isso, podemos utilizar a compressão de coordenadas para tirarmos as duplicatas e deixarmos um vetor apenas com os valores utilizados.

Para isso, sortamos e depois aplicamos a **função erase com unique no vetor**.

A screenshot of a code editor window titled "Cc.cpp". The code inside the editor is:

```
sort(v.begin(), v.end());
v.erase(unique(v.begin(), v.end()), v.end());
```

The code uses the standard library functions `sort` and `unique` from the `<algorithm>` header to sort a vector and then remove consecutive duplicates, effectively compressing the coordinates.

Lower e Upper

Alteramos o vetor de contagem utilizando o limite inferior.

Acessamos o valor das queries com o limite superior - 1.

1	3	5	6	7	8	9
---	---	---	---	---	---	---

↑
**Lower
Bound**

1	3	5	6	7	8	9
---	---	---	---	---	---	---

↑
**Upper
Bound**

Código



Maratona

CIn

Centro de
Informática
UFPE



UNIVERSIDADE
FEDERAL
DE PERNAMBUCO

```
CC Cc.cpp - □ ×

for(auto: [l, r]: events) {
    int pos1 = lower_bound(v.begin(), v.end(), l) - v.begin();
    int pos2 = lower_bound(v.begin(), v.end(), r) - v.begin();
    contador[pos1]++, contador[pos2]--;
}

for (int i=1; i≤sz; i++) {
    prefixsum[i] = contador[i-1] + prefixsum[i-1];
}

while(q--) {
    ll pos;
    cin >> pos;
    cout << prefixsum[(upper_bound(v.begin(), v.end(), pos) - v.begin())-1] << "\n";
}
```



MaratonaCIn



UNIVERSIDADE
FEDERAL
DE PERNAMBUCO

Dúvidas?



Busca exaustiva
Testando toda possibilidade

O que é?



Complete search, ou busca completa, ou ainda busca de força bruta, trata-se de uma técnica de procurar uma resposta testando todos os cenários possíveis e escolhendo o melhor (ou um qualquer) dentre eles.

Analogia Principal: Quebrar uma senha de 4 números

- Você testa um valor (0000).
- Se conseguiu, para.
- Se não conseguiu, testa outro valor (0001).

Objetivo: Explorar sistematicamente todas as possíveis soluções candidatas até que uma solução, ou a melhor solução, seja encontrada.

Quando usar?



Maratona

CIn

Centro de
Informática
UFPE



A Busca Completa só é viável quando o "espaço de busca" (o número total de cenários) é pequeno o suficiente para ser verificado em tempo hábil.

O Fator Decisivo: As Constraints (Limites de Entrada)

- Um computador moderno executa cerca de 10^8 operações/segundo.
- A busca deve ter menos casos que esse limite.

	Limite de N	Exemplo de Problema
$O(N!)$	$N \leq 11$	Permutações (Ex. Caixeiro Viajante - TSP)
$O(N \times 2^N)$	$N \leq 22$	Subconjuntos (Ex. Problema da Mochila)
$O(N^K)$	$N \leq 464$ (para $K=3$)	Loops aninhados (Ex: Achar uma tripla)
$O(K^N)$	Depende de k e N	Backtracking geral (Ex. Sudoku)

Permutações

Como gerar? Podemos usar o `next_permutation`, que transforma uma sequência na sua próxima permutação lexicograficamente.

Ele Retorna `true` se uma próxima permutação foi encontrada, e `false` se a sequência já está na maior permutação possível (ex: [3, 2, 1]).

Complexidade? $O(N)$ por chamada $O(N!)$ no loop `do-while`.



Maratona

CIn



Centro de
Informática
UFPE



UNIVERSIDADE
FEDERAL
DE PERNAMBUCO



next_permutation.cpp

```
1 vector<int> v = {1, 2, 3};  
2 // 1. Garanta que o vetor esteja ordenado  
3 sort(v.begin(), v.end());  
4  
5 // 2. Use o loop do-while para iterar  
6 // por todas as permutações  
7 do {  
8     for (auto e : v) cout << e << ' ';  
9     cout << endl;  
10 } while (next_permutation(v.begin(), v.end()));
```

Subconjuntos

Como gerar? Podemos usar Bitmasks para gerar todos os subconjuntos, testando todas as combinações de pegar ou não pegar um item.

Exemplo: Se $i=5$ (binário 101), significa "pegar o item 0" e "pegar o item 2".

Complexidade? $O(N \times 2^N)$, pois precisamos iterar pelos bits para ver quais estão setados.



MaratonaCIn



UNIVERSIDADE
FEDERAL
DE PERNAMBUCO

C++ substet mask.cpp

```
1  for (int mask = 0; mask < (1 << n); mask++) {  
2      for (int bit = 0; bit < n; bit++) {  
3          if (mask & (1 << (bit))) {  
4              // bit está no subconjunto  
5          }  
6      }  
7  }
```

Loops aninhados

Como gerar? Podemos encontrar todos os pares, triplas, ..., como loops aninhados. Essa estratégia é usada quando o número de "escolhas" é fixo e pequeno.

Complexidade? $O(N^K)$ onde N é o número de elementos no conjunto de escolha e K é o número de elementos que serão escolhidos.

Mais especificamente $C(N, K)$ operações se não tiver repetição.



Maratona

CIn

Centro de
Informática
UFPE

UNIVERSIDADE
FEDERAL
DE PERNAMBUCO
 VIRTUS IMPAVIDA

 loops.cpp

```
1  for (int i = 0; i < n; i++) {  
2      for (int j = i + 1; j < n; j++) {  
3          for (int k = j + 1; k < n; k++) {  
4              // Testa o trio (i, j, k)  
5          }  
6      }  
7  }
```



Backtracking

Busca “inteligente”

O que é?

Backtracking trata-se de uma técnica de força bruta que é “inteligente”.

Analogia Principal: Resolver um labirinto.

- Você avança por um caminho (constrói uma solução parcial).
- Você chega em uma bifurcação (tem várias escolhas).
- Você escolhe um caminho.
- Se bater num beco sem saída (solução inválida), você volta atrás (backtracks) até a última bifurcação e tenta o outro caminho.

Objetivo: Explorar sistematicamente todas as possíveis soluções candidatas, mas descartando (podando) caminhos que você já sabe previamente que não levarão a uma resposta.

Como pensar?

Todo backtracking é uma função recursiva que segue, mais ou menos, o seguinte modelo:

- Testa se a solução tá certa.
- Para cada decisão:
 - Faz uma escolha
 - Chama para o novo estado
 - Desfaz a escolha

backtracking.cpp

```
1 void backtrack( /* estado atual */ ) {
2     // 1. Caso Base: É uma solução completa e válida?
3     if (é_solucao_completa(estado_atual)) {
4         return armazenar_solucao(estado_atual);
5     }
6
7     // 2. Iterar sobre todas as próximas escolhas (Transições)
8     for (cada_proxima_escolha) {
9
10        // 3. Poda: A escolha é válida?
11        if (é_valida(proxima_escolha)) {
12
13            // 4. FAZER (Aplicar a transição)
14            aplica_escolha(proxima_escolha);
15
16            // 5. Recursão (Avançar para o próximo estado)
17            backtrack( /* novo estado */ );
18
19            // 6. DESFAZER (O Backtrack real)
20            desfaz_escolha(proxima_escolha);
21        }
22    }
23}
24}
```

Estados

Pergunta-chave: "O que eu preciso saber agora para tomar a próxima decisão?"

Definição: Um Estado é uma "foto" da sua solução parcial. São as informações mínimas necessárias para continuar a busca.

Como definir: Geralmente, são os parâmetros da sua função recursiva.

EXEMPLOS:

- N-Rainhas:
 - Estado: (int linha_atual, vector<vector<bool>>& board)
- Subset Sum (Soma dos Subconjuntos):
 - Estado: (int idx, int soma_atual)

Transições

Pergunta-chave: "Quais são minhas próximas escolhas a partir do estado atual?"

Definição: Uma Transição é a ação ou escolha que te leva de um estado para o próximo.

Como definir: Geralmente, é o loop for dentro da sua função recursiva.



MaratonaCIn



EXEMPLOS:

- N-Rainhas
 - Transição: "Onde posso colocar a rainha nesta linha?"
- Subset Sum:
 - Transição: "Para o elemento $v[index]$, eu tenho duas escolhas:"
 - Incluir ou não incluir $v[index]$ na soma.

A Mágica

O "Fazer" (Aplicar Transição) é o que modifica o estado.

O "Desfazer" (Backtrack) é o que restaura o estado exatamente como era antes.



MaratonaCIn

Centro de
Informática
UFPE

UNIVERSIDADE
FEDERAL
DE PERNAMBUCO
 VIRTUS IMPAVIDA

Por que desfazer?

Para que na próxima iteração do loop (ex: tentar (linha, col+1)), o estado esteja limpo, como se a tentativa anterior nunca tivesse acontecido.

Poda (Pruning)

É o que torna o backtracking "inteligente" e não só um força bruta cego.

Conceito: "Cortar" galhos da árvore de busca que nunca levarão a uma solução.

Exemplo: Subset Sum (com números positivos)

Estado: (index, soma_atual)

Poda: if (soma_atual > K) {
return; }

Por quê? Se a soma já passou de K e só temos números positivos para adicionar, é impossível voltar para K. Parar a busca agora economiza tempo exponencial.

Receita de bolo



Maratona

CIn



UNIVERSIDADE
FEDERAL
DE PERNAMBUCO

Identifique: O problema pede para "gerar todos...", "encontrar um caminho...", "verificar se é possível..."? As restrições são pequenas (ex: $N \leq 20$)? -> Provavelmente backtracking.

Defina o Estado: "Quais parâmetros minha função backtrack() precisa?" (Ex: index, soma_atual, linha_atual).

Defina as Transições: "Quais são minhas escolhas em cada estado?" (Ex: "tentar todas as colunas", "incluir ou não incluir o item", "tentar todos os vizinhos no grafo"). Isso será seu loop for ou suas chamadas recursivas.

Receita de bolo



MaratonaCIn



Defina a Poda (Pruning): "Quando uma escolha é obviamente inválida?"

(Ex: "coluna já usada", "soma atual > alvo"). Isso é o if (`é_valida(...)`) antes da recursão.

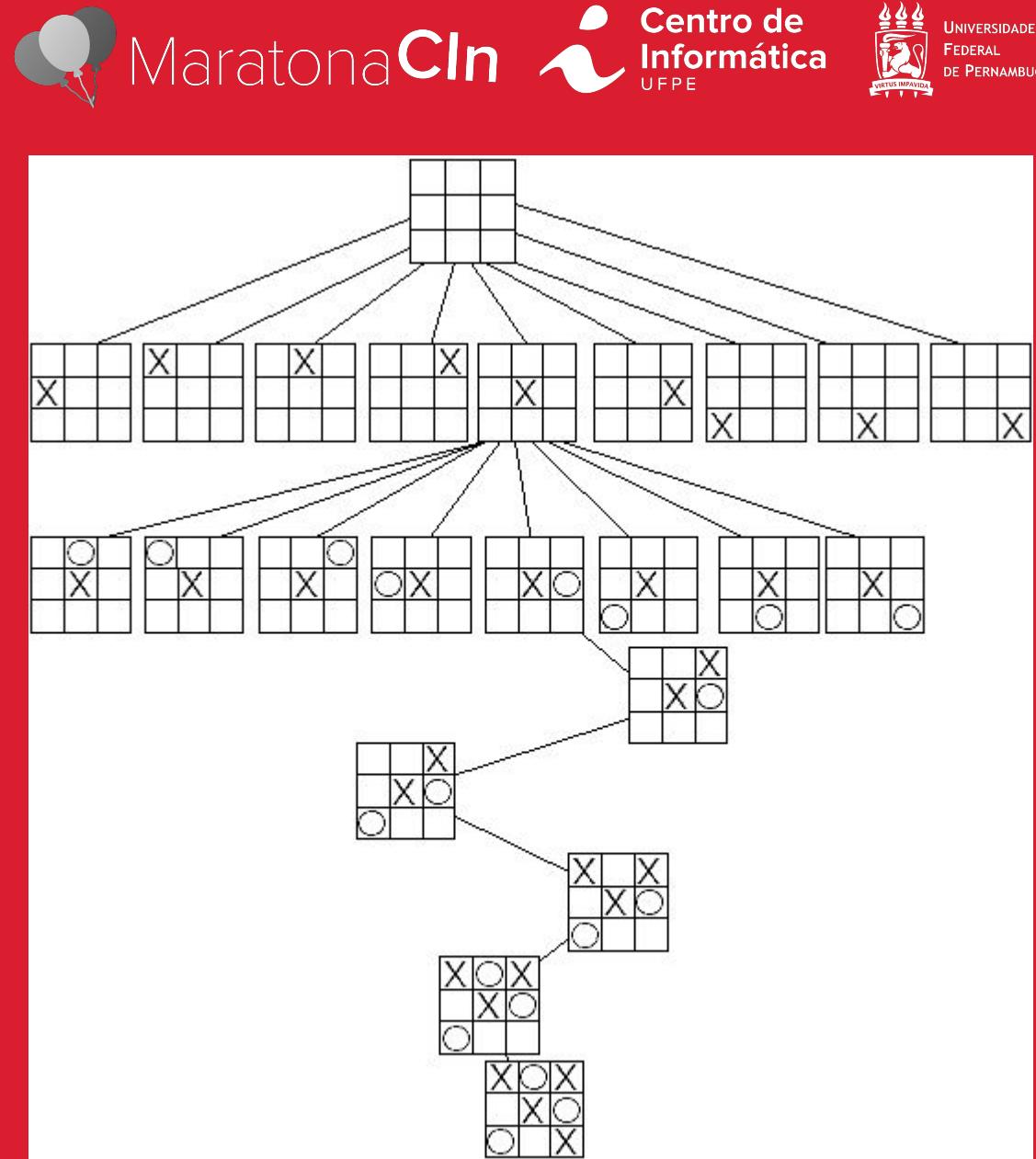
Defina o Caso Base: "Quando eu paro?"

- Sucesso: "Encontrei uma solução completa." (Ex: `linha == N` nas `N-Rainhas`, `soma_atual == alvo`).
- Falha: "Cheguei ao fim sem solução" (Ex: `index == N` no Subset Sum, mas `soma != alvo`).

Complexidade

A pergunta: "Meu backtracking é rápido o suficiente?"

- Depende do número total de estados que sua recursão pode visitar!
- A complexidade raramente é polinomial. Ela é quase sempre Exponencial ou Fatorial.
- A complexidade é ditada pela forma da Árvore de Estados.



Complexidade



MaratonaCIn

Centro de
Informática
UFPE



A complexidade de tempo do seu backtracking é (aproximadamente):

Complexidade = (Número Total de Estados na Árvore) × (Custo por Transição)

Custo por Transição: É o custo de "Fazer" e "Desfazer".

- Ideal: $O(1)$ (ex: usado[i] = true, soma += valor).
- Ruim: $O(N)$ (ex: copiar um vetor inteiro em cada chamada). Tente evitar isso!

Número Total de Estados: É o que realmente domina. É o número de nós na sua árvore de recursão.

Decisão Binária



Problema: Gerar Subconjuntos de N elementos.

Estados: (int index) - "Estou decidindo sobre o elemento index".

Transições: Para cada estado (índice), temos 2 escolhas (transições):

Incluir o elemento.

Não incluir o elemento.

Análise da Árvore:

Nível 0 (raiz): 1 estado

Nível 1: 2 estados

Nível 2: 4 estados

...

Nível N: $2N$ estados (folhas)

Número Total de Estados:

$$1+2+4+\cdots+2N = 2^N$$

Decisão Binária

Análise da Árvore:

Nível 0 (raiz): 1 estado

Nível 1: 2 estados

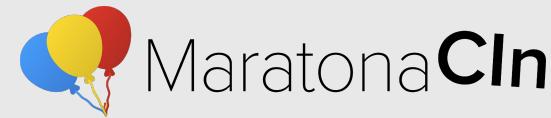
Nível 2: 4 estados

...

Nível N: 2^N estados (folhas)

Número Total de Estados:

$$1+2+4+\dots+2^N=2^{N+1}-1$$



Complexidade: $O(2^N)$ estados ×
 $O(1)$ por transição = $O(2^N)$

Moral: Se em cada passo N você tem k escolhas independentes, a complexidade será da ordem de $O(k^N)$.

Escolha e Reduza



Maratona

CIn



Problema: Gerar Permutações de N elementos.

Estados: (int k) - "Estou escolhendo o k-ésimo elemento da permutação".

Transições: O número de escolhas diminui a cada nível.

Análise da Árvore:

Nível 1 (Raiz): Temos N escolhas (qualquer um dos N números).

Nível 2: Temos N-1 escolhas (qualquer um, exceto o já usado).

Nível 3: Temos N-2 escolhas.

...

O número de folhas (soluções completas) é $N \times (N-1) \times \cdots \times 1 = N!$

Escolha e Reduza



Maratona



Análise da Árvore:

Nível 1 (Raiz): Temos N escolhas (qualquer um dos N números).

Nível 2: Temos N-1 escolhas (qualquer um, exceto o já usado).

Nível 3: Temos N-2 escolhas.

...

O número de folhas (soluções completas) é $N \times (N-1) \times \cdots \times 1 = N!$

Número Total de Estados: O número total de nós na árvore é $1+N+N(N-1)+\cdots+N!$. Isso é dominado pelo número de folhas.

Complexidade: $O(\text{Número de Folhas})$ é uma boa aproximação. A complexidade é $O(N!)$.

Análise x poda

Impacto da poda na complexidade:

No pior caso, a poda pode não ajudar (ex: se alvo for gigante). A complexidade teórica continua $O(2^N)$.

Na prática, a poda "corta" (prunes) galhos inteiros da árvore de estados. Isso é o que torna o backtracking viável!

Conclusão:

Pense na complexidade do pior caso (sem poda) para saber se a ideia geral é relativamente viável.

Use a Poda para otimizar o caso médio e passar nos testes.
Experiência em problemas semelhantes é bom para ter o feeling.

Exemplo

Você recebe um tabuleiro de sudoku parcialmente preenchido.

Responda se é possível completá-lo.

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

Exemplo

Ideia naïve (Força bruta):

Para cada quadrado não preenchido ($up\ to\ n \times n$), tente cada uma das 10 possibilidades.

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

Exemplo

Ideia naïve (Força bruta):

Para cada quadrado não preenchido ($\text{up to } n \times n$), tente cada uma das 9 possibilidades.

Complexidade: $9^{n \times n}$
(passa para n até 3)

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

Exemplo

Ideia com BackTracking

Para cada quadrado não preenchido, tente uma possibilidade e veja se o jogo continua possível. Se não, volta pra o último estado válido e tenta outra escolha.

Complexidade teórica: $9^{n \times n}$

Na prática: Cada corte economiza $9^{n \times n - 1}$ estados!



MaratonaCIn

Centro de
Informática
UFPE

UNIVERSIDADE
FEDERAL
DE PERNAMBUCO


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

Exemplo

Ideia com BackTracking

Para cada quadrado não preenchido, tente uma possibilidade e veja se o jogo continua possível. Se não, volta pra o último estado válido e tenta outra escolha.

Complexidade teórica: $9^{n \times n}$

Na prática: Cada corte economiza $9^{n \times n - 1}$ estados!



MaratonaCIn

Centro de
Informática
UFPE

UNIVERSIDADE
FEDERAL
DE PERNAMBUCO

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

Exemplo

Ideia com BackTracking

Para cada quadrado não preenchido, tente uma possibilidade e veja se o jogo continua possível. Se não, volta pra o último estado válido e tenta outra escolha.

Complexidade teórica: $9^{n \times n}$

Na prática: Cada corte economiza $9^{n \times n - 1}$ estados!



MaratonaCIn

Centro de
Informática
UFPE

UNIVERSIDADE
FEDERAL
DE PERNAMBUCO


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

SAMPLE:4....8.... ..57.34.. ..4.2.5.. 92.1.4.87 ..1.3.2.. ..84.69..1....9....

Exemplo



MaratonaCIn

Centro de
Informática
UFPE

UNIVERSIDADE
FEDERAL
DE PERNAMBUCO


```
 1  bool check(const vector<string>& board, int row, int col, char c) {
 2      for (int i = 0; i < 9; i++) {
 3          if (board[row][i] == c) return false;
 4          if (board[i][col] == c) return false;
 5          if (board[3 * (row / 3) + i / 3][3 * (col / 3) + i % 3] == c)
 6              return false;
 7      }
 8      return true;
 9  }
10
11 bool solveSudoku(vector<string>& board, int i = 0, int j = 0) {
12     if (i == 9) return true;
13     if (j == 9) return solveSudoku(board, i + 1, 0);
14
15     if (board[i][j] != '.') return solveSudoku(board, i, j + 1);
16
17     for (char c = '1'; c <= '9'; c++) {
18         if (check(board, i, j, c)) {
19             board[i][j] = c;
20             if (solveSudoku(board, i, j + 1)) return true;
21             board[i][j] = '.';
22         }
23     }
24
25     return false;
26 }
```

Exemplo



backtracking.cpp

```
1 void backtrack( /* estado atual */ ) {
2     // 1. Caso Base: É uma solução completa e válida?
3     if (é_solucao_completa(estado_atual)) {
4         return armazenar_solucao(estado_atual);
5     }
6
7     // 2. Iterar sobre todas as próximas escolhas (Transições)
8     for (cada_proxima_escolha) {
9
10        // 3. Poda: A escolha é válida?
11        if (é_valida(proxima_escolha)) {
12
13            // 4. FAZER (Aplicar a transição)
14            aplica_escolha(proxima_escolha);
15
16            // 5. Recursão (Avançar para o próximo estado)
17            backtrack( /* novo estado */ );
18
19            // 6. DESFAZER (O Backtrack real)
20            desfaz_escolha(proxima_escolha);
21        }
22    }
23}
24}
```

sudoku.cpp

```
1 bool check(const vector<string>& board, int row, int col, char c) {
2     for (int i = 0; i < 9; i++) {
3         if (board[row][i] == c) return false;
4         if (board[i][col] == c) return false;
5         if (board[3 * (row / 3) + i / 3][3 * (col / 3) + i % 3] == c)
6             return false;
7     }
8     return true;
9 }
10
11 bool solveSudoku(vector<string>& board, int i = 0, int j = 0) {
12     if (i == 9) return true;
13     if (j == 9) return solveSudoku(board, i + 1, 0);
14
15     if (board[i][j] != '.') return solveSudoku(board, i, j + 1);
16
17     for (char c = '1'; c <= '9'; c++) {
18         if (check(board, i, j, c)) {
19             board[i][j] = c;
20             if (solveSudoku(board, i, j + 1)) return true;
21             board[i][j] = '.';
22         }
23     }
24
25     return false;
26 }
```

Exemplo

backtracking.cpp

```
1 void backtrack( /* estado atual */ ) {
2     // 1. Caso Base: É uma solução completa e válida?
3     if (é_solucao_completa(estado_atual)) {
4         return armazenar_solucao(estado_atual);
5     }
6
7     // 2. Iterar sobre todas as próximas escolhas (Transições)
8     for (cada_proxima_escolha) {
9
10        // 3. Poda: A escolha é válida?
11        if (é_valida(proxima_escolha)) {
12
13            // 4. FAZER (Aplicar a transição)
14            aplica_escolha(proxima_escolha);
15
16            // 5. Recursão (Avançar para o próximo estado)
17            backtrack( /* novo estado */ );
18
19            // 6. DESFAZER (O Backtrack real)
20            desfaz_escolha(proxima_escolha);
21        }
22    }
23}
24}
```

sudoku.cpp

```
1 bool check(const vector<string>& board, int row, int col, char c) {
2     for (int i = 0; i < 9; i++) {
3         if (board[row][i] == c) return false;
4         if (board[i][col] == c) return false;
5         if (board[3 * (row / 3) + i / 3][3 * (col / 3) + i % 3] == c)
6             return false;
7     }
8     return true;
9 }
10
11 bool solveSudoku(vector<string>& board, int i = 0, int j = 0) {
12     if (i == 9) return true;
13     if (j == 9) return solveSudoku(board, i + 1, 0);
14
15     if (board[i][j] != '.') return solveSudoku(board, i, j + 1);
16
17     for (char c = '1'; c <= '9'; c++) {
18         if (check(board, i, j, c)) {
19             board[i][j] = c;
20             if (solveSudoku(board, i, j + 1)) return true;
21             board[i][j] = '.';
22         }
23     }
24
25     return false;
26 }
```

Exemplo

backtracking.cpp

```
1 void backtrack( /* estado atual */ ) {
2     // 1. Caso Base: É uma solução completa e válida?
3     if (é_solucao_completa(estado_atual)) {
4         return armazenar_solucao(estado_atual);
5     }
6
7     // 2. Iterar sobre todas as próximas escolhas (Transições)
8     for (cada_proxima_escolha) {
9
10        // 3. Poda: A escolha é válida?
11        if (é_valida(proxima_escolha)) {
12
13            // 4. FAZER (Aplicar a transição)
14            aplica_escolha(proxima_escolha);
15
16            // 5. Recursão (Avançar para o próximo estado)
17            backtrack( /* novo estado */ );
18
19            // 6. DESFAZER (O Backtrack real)
20            desfaz_escolha(proxima_escolha);
21        }
22    }
23}
24}
```

sudoku.cpp

```
1 bool check(const vector<string>& board, int row, int col, char c) {
2     for (int i = 0; i < 9; i++) {
3         if (board[row][i] == c) return false;
4         if (board[i][col] == c) return false;
5         if (board[3 * (row / 3) + i / 3][3 * (col / 3) + i % 3] == c)
6             return false;
7     }
8     return true;
9 }
10
11 bool solveSudoku(vector<string>& board, int i = 0, int j = 0) {
12     if (i == 9) return true;
13     if (j == 9) return solveSudoku(board, i + 1, 0);
14
15     if (board[i][j] != '.') return solveSudoku(board, i, j + 1);
16
17     for (char c = '1'; c <= '9'; c++) {
18         if (check(board, i, j, c)) {
19             board[i][j] = c;
20             if (solveSudoku(board, i, j + 1)) return true;
21             board[i][j] = '.';
22         }
23     }
24
25     return false;
26 }
```

Exemplo

backtracking.cpp

```
1 void backtrack( /* estado atual */ ) {
2     // 1. Caso Base: É uma solução completa e válida?
3     if (é_solucao_completa(estado_atual)) {
4         return armazenar_solucao(estado_atual);
5     }
6
7     // 2. Iterar sobre todas as próximas escolhas (Transições)
8     for (cada_proxima_escolha) {
9
10        // 3. Poda: A escolha é válida?
11        if (é_valida(proxima_escolha)) {
12
13            // 4. FAZER (Aplicar a transição)
14            aplica_escolha(proxima_escolha);
15
16            // 5. Recursão (Avançar para o próximo estado)
17            backtrack( /* novo estado */ );
18
19            // 6. DESFAZER (O Backtrack real)
20            desfaz_escolha(proxima_escolha);
21        }
22    }
23}
24}
```

sudoku.cpp

```
1 bool check(const vector<string>& board, int row, int col, char c) {
2     for (int i = 0; i < 9; i++) {
3         if (board[row][i] == c) return false;
4         if (board[i][col] == c) return false;
5         if (board[3 * (row / 3) + i / 3][3 * (col / 3) + i % 3] == c)
6             return false;
7     }
8     return true;
9 }
10
11 bool solveSudoku(vector<string>& board, int i = 0, int j = 0) {
12     if (i == 9) return true;
13     if (j == 9) return solveSudoku(board, i + 1, 0);
14
15     if (board[i][j] != '.') return solveSudoku(board, i, j + 1);
16
17     for (char c = '1'; c ≤ '9'; c++) {
18         if (check(board, i, j, c)) {
19             board[i][j] = c;
20             if (solveSudoku(board, i, j + 1)) return true;
21             board[i][j] = '.';
22         }
23     }
24
25     return false;
26 }
```

Exemplo

backtracking.cpp

```
1 void backtrack( /* estado atual */ ) {
2     // 1. Caso Base: É uma solução completa e válida?
3     if (é_solucao_completa(estado_atual)) {
4         return armazenar_solucao(estado_atual);
5     }
6
7     // 2. Iterar sobre todas as próximas escolhas (Transições)
8     for (cada_proxima_escolha) {
9
10        // 3. Poda: A escolha é válida?
11        if (é_valida(proxima_escolha)) {
12
13            // 4. FAZER (Aplicar a transição)
14            aplica_escolha(proxima_escolha);
15
16            // 5. Recursão (Avançar para o próximo estado)
17            backtrack( /* novo estado */ );
18
19            // 6. DESFAZER (O Backtrack real)
20            desfaz_escolha(proxima_escolha);
21        }
22    }
23
24 }
```

sudoku.cpp

```
1 bool check(const vector<string>& board, int row, int col, char c) {
2     for (int i = 0; i < 9; i++) {
3         if (board[row][i] == c) return false;
4         if (board[i][col] == c) return false;
5         if (board[3 * (row / 3) + i / 3][3 * (col / 3) + i % 3] == c)
6             return false;
7     }
8     return true;
9 }
10
11 bool solveSudoku(vector<string>& board, int i = 0, int j = 0) {
12     if (i == 9) return true;
13     if (j == 9) return solveSudoku(board, i + 1, 0);
14
15     if (board[i][j] != '.') return solveSudoku(board, i, j + 1);
16
17     for (char c = '1'; c <= '9'; c++) {
18         if (check(board, i, j, c)) {
19             board[i][j] = c;
20             if (solveSudoku(board, i, j + 1)) return true;
21             board[i][j] = '.';
22         }
23     }
24
25     return false;
26 }
```

Exemplo

backtracking.cpp

```
1 void backtrack( /* estado atual */ ) {
2     // 1. Caso Base: É uma solução completa e válida?
3     if (é_solucao_completa(estado_atual)) {
4         return armazenar_solucao(estado_atual);
5     }
6
7     // 2. Iterar sobre todas as próximas escolhas (Transições)
8     for (cada_proxima_escolha) {
9
10        // 3. Poda: A escolha é válida?
11        if (é_valida(proxima_escolha)) {
12
13            // 4. FAZER (Aplicar a transição)
14            aplica_escolha(proxima_escolha);
15
16            // 5. Recursão (Avançar para o próximo estado)
17            backtrack( /* novo estado */ );
18
19            // 6. DESFAZER (O Backtrack real)
20            desfaz_escolha(proxima_escolha);
21        }
22    }
23}
24}
```

sudoku.cpp

```
1 bool check(const vector<string>& board, int row, int col, char c) {
2     for (int i = 0; i < 9; i++) {
3         if (board[row][i] == c) return false;
4         if (board[i][col] == c) return false;
5         if (board[3 * (row / 3) + i / 3][3 * (col / 3) + i % 3] == c)
6             return false;
7     }
8     return true;
9 }
10
11 bool solveSudoku(vector<string>& board, int i = 0, int j = 0) {
12     if (i == 9) return true;
13     if (j == 9) return solveSudoku(board, i + 1, 0);
14
15     if (board[i][j] != '.') return solveSudoku(board, i, j + 1);
16
17     for (char c = '1'; c <= '9'; c++) {
18         if (check(board, i, j, c)) {
19             board[i][j] = c;
20             if (solveSudoku(board, i, j + 1)) return true;
21             board[i][j] = '.';
22         }
23     }
24
25     return false;
26 }
```

Exemplo

backtracking.cpp

```
1 void backtrack( /* estado atual */ ) {
2     // 1. Caso Base: É uma solução completa e válida?
3     if (é_solucao_completa(estado_atual)) {
4         return armazenar_solucao(estado_atual);
5     }
6
7     // 2. Iterar sobre todas as próximas escolhas (Transições)
8     for (cada_proxima_escolha) {
9
10        // 3. Poda: A escolha é válida?
11        if (é_valida(proxima_escolha)) {
12
13            // 4. FAZER (Aplicar a transição)
14            aplica_escolha(proxima_escolha);
15
16            // 5. Recursão (Avançar para o próximo estado)
17            backtrack( /* novo estado */ );
18
19            // 6. DESFAZER (O Backtrack real)
20            desfaz_escolha(proxima_escolha);
21        }
22    }
23
24 }
```

sudoku.cpp

```
1 bool check(const vector<string>& board, int row, int col, char c) {
2     for (int i = 0; i < 9; i++) {
3         if (board[row][i] == c) return false;
4         if (board[i][col] == c) return false;
5         if (board[3 * (row / 3) + i / 3][3 * (col / 3) + i % 3] == c)
6             return false;
7     }
8     return true;
9 }
10
11 bool solveSudoku(vector<string>& board, int i = 0, int j = 0) {
12     if (i == 9) return true;
13     if (j == 9) return solveSudoku(board, i + 1, 0);
14
15     if (board[i][j] != '.') return solveSudoku(board, i, j + 1);
16
17     for (char c = '1'; c <= '9'; c++) {
18         if (check(board, i, j, c)) {
19             board[i][j] = c;
20             if (solveSudoku(board, i, j + 1)) return true;
21             board[i][j] = '.';
22         }
23     }
24
25     return false;
26 }
```



MaratonaCIn



UNIVERSIDADE
FEDERAL
DE PERNAMBUCO

Dúvidas?



MaratonaCIn

Centro de
Informática
UFPE

UNIVERSIDADE
FEDERAL
DE PERNAMBUCO


Dúvidas?

Próximo passo:
Fazer o Homework!



MaratonaCIn



UNIVERSIDADE
FEDERAL
DE PERNAMBUCO

That's all, Folks!

