Bruno Schumacher Farias Souza - 202121316

Gabriel de Souza Matheus Oliveira - 202121097

Guilherme Sampaio Borges Santana – 202220025

2162B

Exemplo de Execução

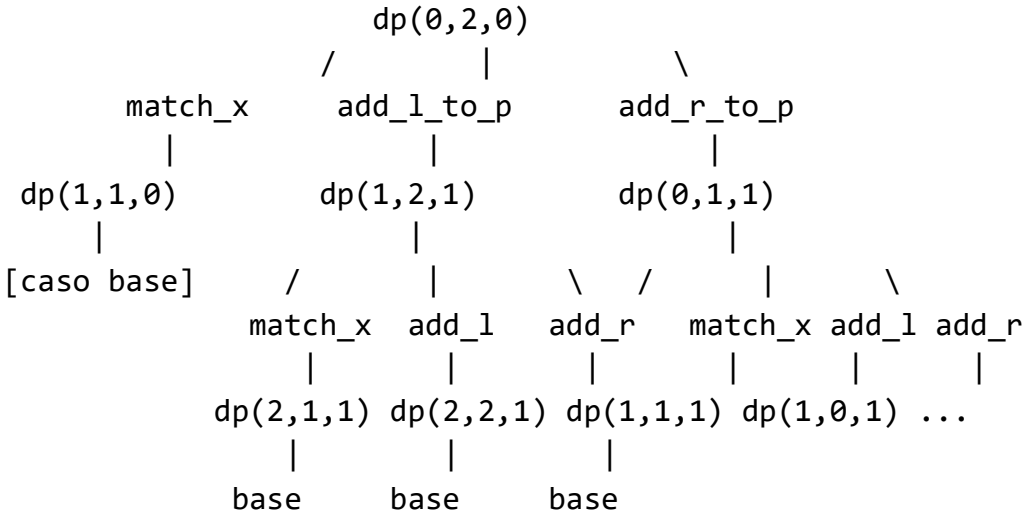
Entrada:

```
s = "101"
n = 3
```

Estado inicial:

```
dp_solve(0, 2, 0)
```

Árvore de Busca (Backtracking com Memoização)



Explicação da Primeira Camada

1. **match_x**
    a. `s[0] == s[2] == '1'` → tenta formar palíndromo x
    b. Chamada: `dp(1,1,0)`
    c. Um caractere sozinho é palíndromo válido.
2. **add_l_to_p**
    a. Coloca `s[0]='1'` em p → novo estado `p_state=1`
    b. Chamada: `dp(1,2,1)`
    c. Dentro dela:
        i. `match_x`: falha (`'0' != '1'`)
        ii. `add_l_to_p`: `'0'` inválido (pois `p_state=1`)
        iii. `add_r_to_p`: `'1'` válido → `dp(1,1,1)`
3. **add_r_to_p**
    a. Coloca `s[2]='1'` em p → novo estado `p_state=1`
    b. Chamada: `dp(0,1,1)`
    c. Dentro dela:
        i. `match_x`: falha (`'1' != '0'`)
        ii. `add_l_to_p`: `'1'` válido → `dp(1,1,1)`
        iii. `add_r_to_p`: `'0'` inválido (`1...0`)

Caminhos válidos encontrados

1. `match_x → dp(1,1,0)` → (`add_x`)
2. `add_l_to_p → add_r_to_p → dp(1,1,1)`
3. `add_r_to_p → add_l_to_p → dp(1,1,1)`

Resumo da Árvore Simplificada

```
dp(0,2,0)
├── match_x → dp(1,1,0)
│   └── add_x
├── add_l_to_p → dp(1,2,1)
│   └── add_r_to_p → dp(1,1,1)
└── add_r_to_p → dp(0,1,1)
    └── add_l_to_p → dp(1,1,1)
```

Interpretação

- O algoritmo tenta **todas as combinações possíveis** de colocar os caracteres de `s` em `p` ou `x`.
- As restrições (como `1...0` inválido) fazem a poda automática da árvore.
- A **memoização** evita recalcular os mesmos subproblemas, como `dp(1,1,1)` que aparece em múltiplos caminhos.
- Isso é um **backtracking com DP top-down** e reconstrução de caminho.

1180/A

Bruno Schumacher Farias Souza - 202121316
Gabriel de Souza Matheus Oliveira - 202121097
Guilherme Sampaio Borges Santana - 202220025

→ Prova por Indução – Alex and a Rhombus - 1180A

- Fórmula fechada

$S(n)$ = Soma do número de células até a ordem n

$S(n) = 1 + 4(1) + 4(2) + 4(3) + \ldots + 4(n-1)$

$S(n) = 1 + 4\underbrace{(1+2+3+\ldots+(n-1))}_{\frac{n(n-1)}{2}}$

$S(n) = 1 + 4\frac{n(n-1)}{2}$

$S(n) = 1 + 2n(n-1) = 1 + 2n^2 - 2n$

$\boxed{S(n) = n^2 + (n-1)^2}$

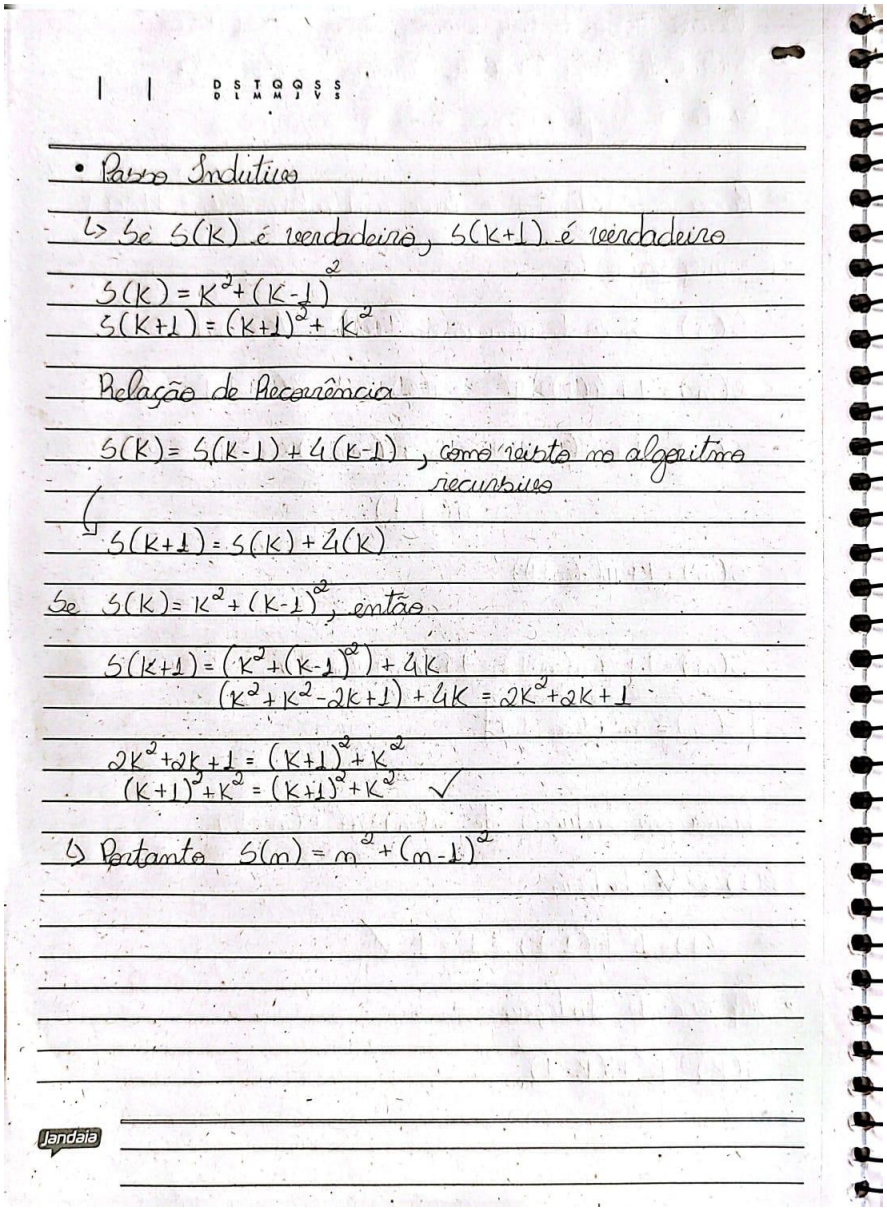
- Prova por Indução de $S(n) = n^2 + (n-1)^2$

• $S(1)$ é Verdadeiro

↳ $S(1) = 1^2 + (1-1)^2 = 1^2 = 1$ ✓

• Hipótese de Indução

$S(k) = k^2 + (k-1)^2$



1807B

Problema: Dado um array de sacos de doces, Mihai pega todos os sacos com valores pares e Bianca pega todos os sacos com valores ímpares. Podemos reorderar os

sacos. Queremos saber se existe uma ordem tal que, após cada saco ser pego (exceto no começo, quando ambos têm 0), Mihai tenha ESTRITAMENTE mais doces que Bianca.

Definições:

- sum_even = soma de todos os valores pares (doces do Mihai, independentemente da ordem).
- sum_odd = soma de todos os valores ímpares (doces da Bianca, independentemente da ordem).

Objetivo:

- Existe uma reordenação tal que, depois de cada passo k (k >= 1), vale:

Mihai_prefix(k) > Bianca_prefix(k).

Parte 1 — Necessidade:

1. Se existe uma ordem que funciona, então no final (após todos os sacos) também deve valer:

Mihai_total > Bianca_total.

2. Os totais finais não dependem da ordem, pois a paridade fixa o dono:

Mihai_total = sum_even

Bianca_total = sum_odd

3. Logo, condição necessária:

sum_even > sum_odd.

Se sum_even <= sum_odd, é impossível, pois no final Mihai não teria estritamente mais que Bianca.

Parte 2 — Suficiência (construção gulosa):

Suponha que sum_even > sum_odd.

Considere a ordem gulosa:

- Coloque TODOS os pares primeiro (em qualquer ordem entre si).
- Depois, coloque TODOS os ímpares (em qualquer ordem entre si).

Verificação:

A) Enquanto só aparecem pares:

- Em cada passo, Mihai recebe um valor par (>= 2).
- Bianca continua com 0.
- Portanto, após cada um desses passos: Mihai_prefix > 0 = Bianca_prefix.

B) Quando começam os ímpares:

- Antes do primeiro ímpar, Mihai já tem exatamente sum_even.
- Ao longo dos próximos passos, Bianca vai acumulando parte de sum_odd.
- Em qualquer prefixo de ímpares, vale: Bianca_prefix <= sum_odd.
- Como sum_even > sum_odd, então, em qualquer momento após iniciar os ímpares:

Mihai_prefix (que é sum_even) > Bianca_prefix (que é no máximo sum_odd).

- Assim, Mihai permanece estritamente à frente em todos os passos.

Conclusão:

- Se sum_even <= sum_odd: NÃO existe ordenação possível (necessidade).
- Se sum_even > sum_odd: a ordem “todos os pares, depois todos os ímpares” funciona (suficiência).
- Portanto, a regra “responder YES se e somente se sum_even > sum_odd” é correta e ótima.

Algoritmo prático:

1. Some todos os pares em sum_even.
2. Some todos os ímpares em sum_odd.
3. Se sum_even > sum_odd -> imprimir YES; caso contrário -> imprimir NO.

Exemplo rápido:

- a = [1, 2, 3, 4]

sum_even = 2 + 4 = 6

sum_odd = 1 + 3 = 4

6 > 4 -> YES.

Ordem viável (gulosa): [4, 2, 1, 3]. Mihai sempre à frente.

2126B

Passo 1 — Redução por blocos de dias bons

A sequência é dividida em blocos (máximos) de dias bons, separados por dias ruins (1). Como não é possível atravessar um dia ruim durante uma trilha, cada bloco pode ser tratado **independentemente**. A resposta total é a soma das respostas ótimas de cada bloco.

Considere um bloco de L zeros consecutivos (dias bons). Dentro desse bloco, cada trilha consome:

- k dias consecutivos (a trilha), e
- entre duas trilhas contíguas, **pelo menos 1 dia** de intervalo (descanso).

Passo 2 — Limite superior (cota) para qualquer solução

Se colocamos m trilhas dentro do bloco, então:

- usamos m·k dias de trilha, e
- precisamos de pelo menos m−1 dias de descanso entre elas.

Logo,

```
m·k + (m − 1) ≤ L
⇒ m(k + 1) − 1 ≤ L
⇒ m ≤ floor( (L + 1) / (k + 1) ).
```

Portanto, **nenhuma** solução (gulosa ou não) pode exceder ⌊(L+1)/(k+1)⌋ trilhas nesse bloco.

Passo 3 — A estratégia gulosa atinge essa cota (construtividade)

No bloco de comprimento L, o guloso faz:

- 1ª trilha começando no primeiro dia do bloco;
- depois, “reserva” 1 dia de descanso;
- repete o processo o mais cedo possível.

Isso coloca as trilhas nas posições:

```
início do bloco,
início + (k + 1),
início + 2(k + 1),
...
```

Pararemos quando faltar espaço para k dias de trilha. O número de trilhas colocadas é justamente

```
m_greedy = floor( (L + 1) / (k + 1) ),
```

que coincide com a cota superior do Passo 2. Logo, o guloso é **ótimo em cada bloco**.

Passo 4 — Argumento de troca (robustez do guloso)

Se alguém propõe uma solução ótima cujo **primeiro** início de trilha no bloco não é o mais cedo possível, podemos **deslocá-la para a esquerda** até encostar no começo do bloco (ou no fim da trilha anterior, mais 1 dia de descanso). Esse deslocamento:

- não usa dias ruins (estamos dentro do bloco),
- não viola os descansos,
- e **só aumenta** o espaço livre à direita (nunca diminui).

Repetindo esse deslocamento para cada trilha, obtemos uma solução “alinhada à esquerda”, que é exatamente a produzida pelo guloso. Portanto, qualquer solução ótima pode ser transformada na gulosa sem perder qualidade.

Conclusão

- Em cada bloco de L dias bons, o máximo é $\lfloor (L+1)/(k+1) \rfloor$.
- O guloso atinge esse máximo em cada bloco.
- A soma desses máximos sobre todos os blocos é a melhor resposta global.

Logo, **a solução gulosa é ótima**.