maratona de programação

UNIFEI

5 – **Paradigmas II**

Divisão & Conquista

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

Dividindo para Conquistar

Dada uma instância de um problema, devemos:

1. Dividir o problema original em **subproblemas**, que são instâncias menores do mesmo subproblema.
   Normalmente na metade ou próximo disso.

2. Resolver cada um desses novos problemas **recursivamente**.
   São menores e possuem soluções mais simples.

3. **Combinar** as soluções dos subproblemas em uma única solução global para o problema original.

Dividindo para Conquistar

Alguns exemplos de algoritmos que utilizam o paradigma:

- Quicksort
- Mergesort
- **Busca Binária**
  Ainda que este talvez possa ser classificado como “Diminuir para Conquistar” (*Decrease & Conquer*)
- **Algoritmo de Strassen** (Multiplicação de Matrizes)
- Algoritmo de Karatsuba (Multiplicação de números grandes)
- Vários algoritmos de geometria e matemática
  - **Exponenciação Binária**
  - Fecho convexo (*Convex Hull*)

A maioria dos algoritmos deste tipo são expressos naturalmente através de **códigos recursivos**.

Complexidade – Relações de Recorrência

Algoritmos de divisão e conquista resolvem um problema de tamanho **n** solucionando, recursivamente, **a** subproblemas de tamanho **n/b** e, então, combinando estas respostas em tempo **O(n^d)**.

Seu tempo de execução pode, portanto, ser capturado pela equação:

$$T(n) = \mathbf{a}*T(\mathbf{n/b}) + O(\mathbf{n}^{\mathbf{d}})$$

E como resolvemos a relação de recorrência, obtendo a complexidade?

Complexidade – Teorema Mestre

Através do **Teorema Mestre:**

– Se $T(n) = a*T(n/b) + O(n^d)$
  Para constantes $a > 0$, $b > 1$ e $d \geq 0$, temos que:

$$T(n) = \begin{cases} \mathbf{O(n^d)}, & \text{se } d > \log_b a \\ \mathbf{O(n^d \log n)}, & \text{se } d = \log_b a \\ \mathbf{O(n^{\log_b a})}, & \text{se } d < \log_b a \end{cases}$$

Este único teorema nos dá o tempo de execução da maioria dos procedimentos de divisão e conquista que provavelmente usaremos.

Complexidade – Teorema Mestre

O teorema mestre nos diz, por exemplo, que uma solução onde dividimos cada problema em duas instâncias do mesmo problema com metade de seu tamanho, combinando-as em empo linear, possui complexidade assintótica de O(n logn). Veja:

a = 2, b = 2, d = 1.

d == $\log_2 2$, e, portanto:

T(n) = O(n logn)

$$T(n) = \begin{cases} \mathbf{O(n^d)}, & \text{se } d > \log_b a \\ \mathbf{O(n^d \log n)}, & \text{se } d = \log_b a \\ \mathbf{O(n^{\log_b a})}, & \text{se } d < \log_b a \end{cases}$$

Busca Binária

O mais famoso dos algoritmos de divisão (ou diminuição) e conquista é a **busca binária**:

- Para **encontrarmos** o elemento k em um vetor grande contendo elementos n elementos ordenados:
  - Primeiro comparamos k com o elemento na posição n/2 e, dependendo do resultado, fazemos **recursão** ou na primeira metade, [0, ..., n/2 – 1], ou na segunda metade, [n/2 + 1, ..., n-1].
  - É obrigatório que o vetor esteja previamente ordenado.

Repare que se formos rígidos, o algoritmo não segue os padrões, pois abandona a parte que não interessa para focar apenas na **metade do problema**. Poderia ser chamado de “Decrease & Conquer”.

Busca Binária

Divisão: Particiona o vetor V em dois sub-vetores V1 e V2, que possuem n/2 elementos, cada.

Conquista: Verifica se o elemento do meio é o desejado.

Caso seja, encontrou-se o elemento: retorne true.

Caso o elemento seja menor que o desejado, realize a busca binária apenas na segunda metade do vetor restante.

Caso o elemento seja maior, realize a busca apenas na primeira metade do vetor restante

Caso base: Vetor com tamanho <= 1: retorne false.

```c
bool binary_search(int *vet, int ini, int end, int key)
{
    int mid = (end + ini)/2;

    if(vet[mid] == key) return true;
    if(ini >= end) return false;

    if(vet[mid] < key)
        binary_search(vet, mid+1, end, key);
    else
        binary_search(vet, ini, mid-1, key);
}
```

Na main:

```c
if(binary_search(vet, 0, SIZE-1, key))
    printf("Encontrado!");
else
    printf("Nao achei...");
```

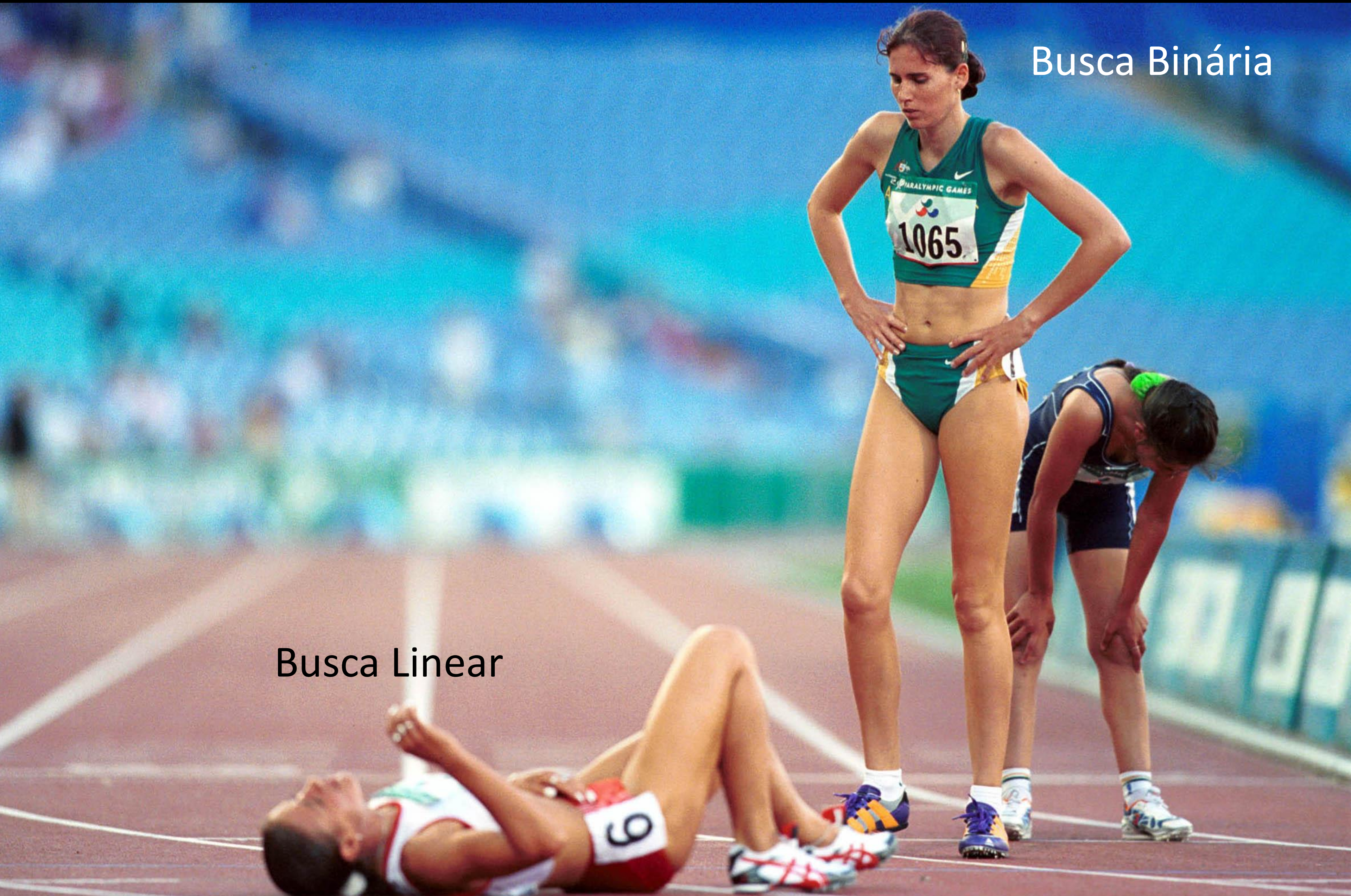
Busca Binária

A recorrência da é modelada pela equação
$T(n) = T(n/2) + O(1)$

- **a = 1, b = 2, d = 0**
  - O algoritmo sempre reparte o vetor em duas metades do mesmo tamanho (b = 2)
  - Não há necessidade de se combinar as sub-respostas (a = 1)
  - Os testes para divisão possuem tempo constante (d = 0)

Através do segundo caso do teorema mestre, temos que **sua complexidade é O(log n)**

- **Melhor que a busca sequencial**, que é linear – O(n)

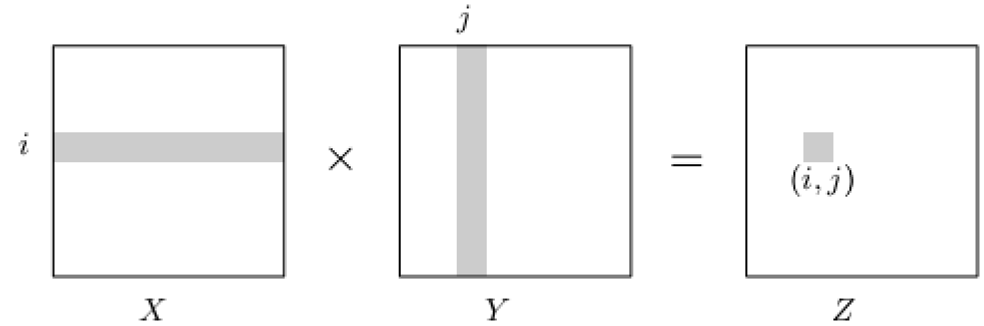

Busca Binária
Busca Linear
1065

Multiplicação de Matrizes

O produto de duas matrizes *n* x *n*, X e Y, é uma terceira matriz *n* x *n*, Z = XY, com a (i,j)-ésima célula igual a:

$$Z_{ij} = \sum_{k=1}^{n} X_{ik} Y_{kj}$$

O que significa dizer que Z_{ij} é o produto escalar da i-ésima linha de X com a j-ésima coluna de Y:



Multiplicação de Matrizes

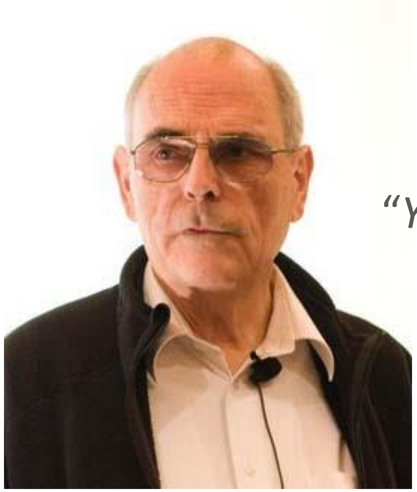
A fórmula anterior implica em um **algoritmo O(n^3)**, para matrizes de n x n, pois existem n^2 elementos para serem computados e cada um deles toma tempo n (percorrendo linha/coluna):

```
void multiplica(int** a, int** b, int** c, int tam)
{
    for (int i = 0; i < tam; i++)
        for (int j = 0; j < tam; j++)
        {
            c[i][j] = 0;
            for (int k = 0; k < tam; k++)
                c[i][j] += a[i][k] * b[k][j];
        }
}
```

Multiplicação de Matrizes

- **Volker Strassen** desenvolveu um algoritmo significativamente mais eficiente, baseado em **divisão e conquista** (1969):
  - É particularmente fácil quebrar a multiplicação de matrizes em subproblemas, porque ela pode ser realizada em blocos.
  - Podemos particionar X e Y em quatro sub-blocos de tamanho n/2 por n/2:

$$X = \begin{bmatrix} A & B \\ C & D \end{bmatrix}, \quad Y = \begin{bmatrix} E & F \\ G & H \end{bmatrix}$$



"Yes, I'm that good."

Multiplicação de Matrizes

E seu produto pode ser expresso em termos desses blocos, e **é exatamente como se esses blocos fossem simples elementos**:

$$XY = \begin{bmatrix} A & B \\ C & D \end{bmatrix} \begin{bmatrix} E & F \\ G & H \end{bmatrix} = \begin{bmatrix} AE + BG & AF + BH \\ CE + DG & CF + DH \end{bmatrix}$$

Para computar a multiplicação de XY agora, podemos computar multiplicações de 8 matrizes com tamanho n/2:
AE, BG, AF, BH, CE, DG, CF, DH.

Além disso, é necessário fazer algumas adições extras, resultando num tempo $O(n^2)$. Como ficaria a relação de recorrência?

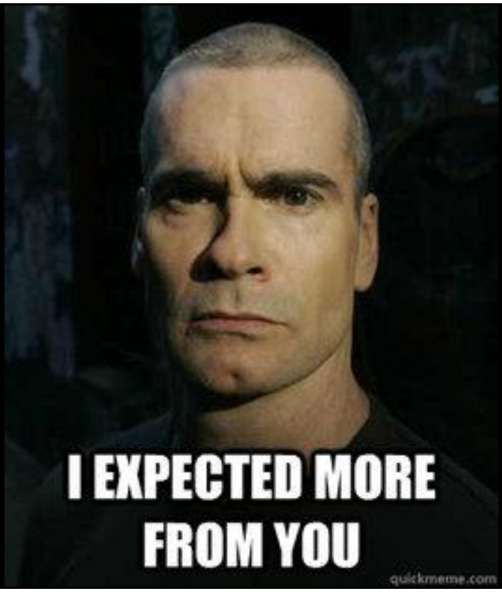
Multiplicação de Matrizes

A relação ficaria:

$T(n) = 8*T(n/2) + O(n^2)$

Segundo o teorema mestre, o algoritmo também é **$O(n^3)$**

E todos ficam desapontados :(



Multiplicação de Matrizes

No entanto, Strassen descobriu uma otimização interessante que irá diminuir a complexidade utilizando apenas uma jogada inteligente de álgebra.

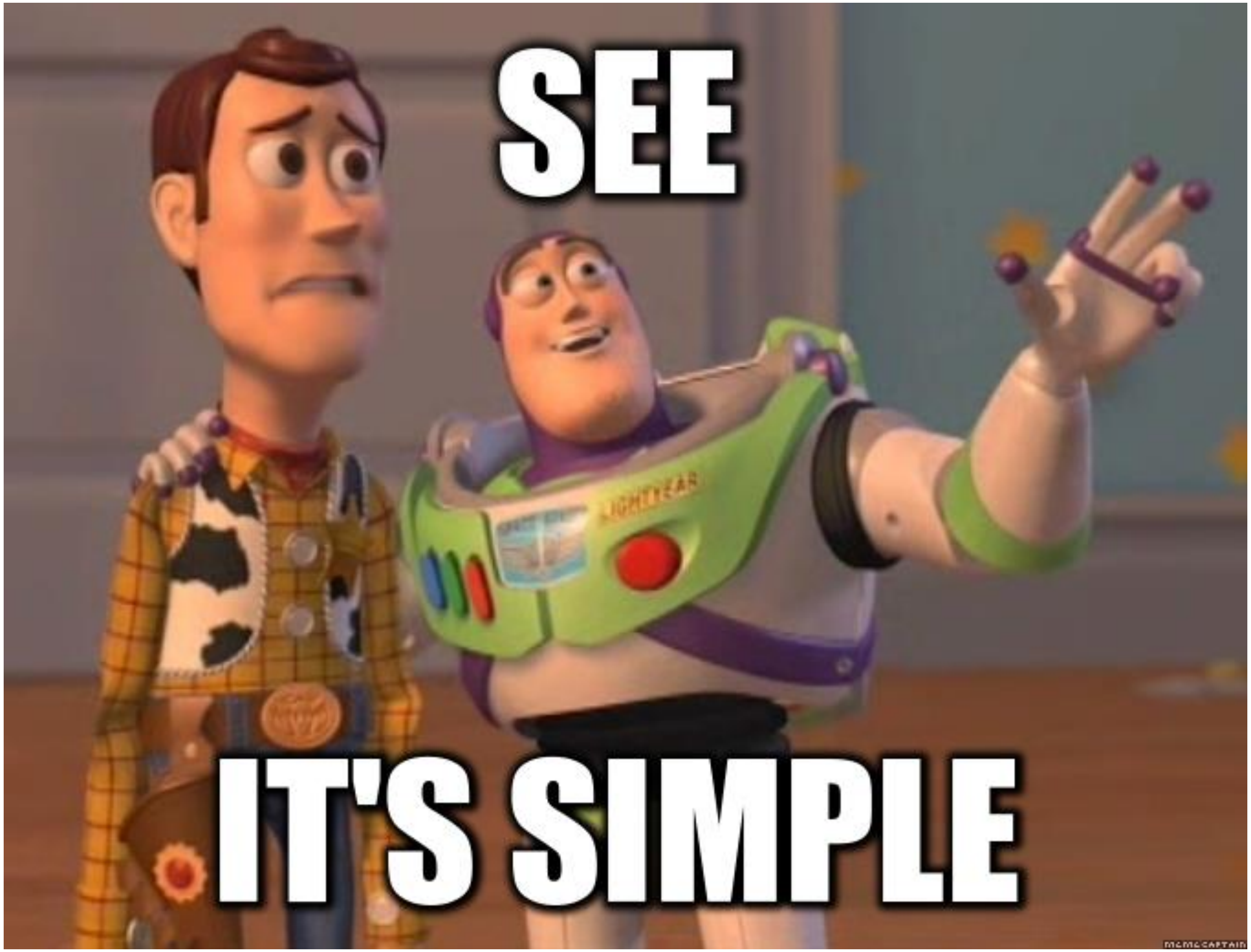
Iremos decompor a multiplicação de matrizes em algo **mais engenhoso**:

$$XY = \begin{bmatrix} P_5 + P_4 - P_2 + P_6 & P_1 + P_2 \\ P_3 + P_4 & P_1 + P_5 - P_3 - P_7 \end{bmatrix}$$

Onde:

$$\begin{aligned} P_1 &= A(F - H) \\ P_2 &= (A + B)H \\ P_3 &= (C + D)E \\ P_4 &= D(G - E) \end{aligned}$$

$$\begin{aligned} P_5 &= (A + D)(E + H) \\ P_6 &= (B - D)(G + H) \\ P_7 &= (A - C)(E + F) \end{aligned}$$


SEE
IT'S SIMPLE

Multiplicação de Matrizes - Strassen

Portanto, iremos calcular apenas 7 sub-problemas de tamanho n/2 (P_1 a P_7), diminuindo a complexidade:

$T(n) = 7*T(n/2) + O(n^2)$

Segundo o teorema mestre, no terceiro caso, a complexidade seria igual a $O(n^{\log_2 7}) \approx$ **$O(n^{2,81})$**

```
// inicializando elementos das matrizes
for(int i = 0; i < half; i++)
{
    for(int j = 0; j < half; j++)
    {
        a11[i][j] = a[i][j];
        a12[i][j] = a[i][j + half];
        a21[i][j] = a[i + half][j];
        a22[i][j] = a[i + half][j + half];

        b11[i][j] = b[i][j];
        b12[i][j] = b[i][j + half];
        b21[i][j] = b[i + half][j];
        b22[i][j] = b[i + half][j + half];
    }
}

// Calcula Ps
subtract(b12, b22, aux1, half);
strassen(a11, aux1, p1, half);

add(a11, a12, aux1, half);
strassen(aux1, b22, p2, half);

add(a21, a22, aux1, half);
strassen(aux1, b11, p3, half);
```

.
.
.

```
add(p1, p5, aux1, half);
subtract(aux1, p3, aux2, half);
subtract(aux2, p7, c22, half);

// passa para C
for (int i = 0; i < half ; i++)
{
    for (int j = 0 ; j < half ; j++)
    {
        c[i][j] = c11[i][j];
        c[i][j + half] = c12[i][j];
        c[i + half][j] = c21[i][j];
        c[i + half][j + half] = c22[i][j];
    }
}
```

.
.
.

Exponenciação Binária

Imagine que queiramos calcular x^n, onde x e n são inteiros.

Por algum motivo (tempo, por exemplo) não podemos utilizar o método pow da biblioteca padrão.

Como seria o método direto?

```
// método mais simples para expontenciacao O(n)
int naivepower(int base, int expoente)
{
    int ans = 1;
    for(int i = 0; i < expoente; i++)
        ans *= base;
    return ans;
}
```

Executa em O(n).
Mas e se quisermos suportar **grandes valores** com eficiência?

Exponenciação Binária

Vamos explorar o paradigma de divisão e conquista:

Existem algumas propriedades que podemos ter como base:

1. $X^0 = 1$
2. $X^n = x*x^{n-1}$
3. $X^n = x^{n/2}*x^{n/2}$

Podemos utilizar a propriedade 1 como caso base da recursão e dois como chamada recursiva. Vejamos como ficaria:

```
// método recursivo - primeira abordagem O(n)
int recursivepower(int base, int expoente)
{
    if(expoente == 0) return 1;
    return base*recursivepower(base, expoente-1);
}
```

Como ficaria o tempo de execução?

$T(n) = 1 + T(n-1)$
$T(n) = 1 + 1 + T(n-2)$
$T(n) = 1 + 1 + 1 + T(n-3)$
...

O algoritmo também executa em O(n)!
Tão **lento** quanto o outro, e ainda com overhead de chamadas de função recursivas.

Exponenciação Binária

E se tentarmos a terceira propriedade? Como ficaria?

$$\mathbf{X^{n} = x^{n/2} * x^{n/2}}$$

A melhor maneira, é calculando-se uma única vez $\mathbf{x^{n/2}}$. Veja:

```
// método divisão e consquista O(logn)
int dividepower(int base, int expoente)
{
    if(expoente == 0) return 1;
    if(expoente%2) return base*dividepower(base, expoente-1);
    int part = dividepower(base, expoente/2);
    return part*part;
}
```

Como ficaria a complexidade?

Exponenciação Binária

```
// método divisão e consquista O(logn)
int dividepower(int base, int expoente)
{
    if(expoente == 0) return 1;
    if(expoente%2) return base*dividepower(base, expoente-1);
    int part = dividepower(base, expoente/2);
    return part*part;
}
```

T(n) = 1 + T(n-1) se expoente for ímpar
T(n) = 1 + T(n/2) se expoente for par

Uma vez que n-1 é par quando n for ímpar...
T(n) = 1 + 1 + T((n-1)/2) se n for ímpar

Temos um subproblema apenas, de metade do tamanho: O(logn)

Exponenciação Binária

Veja que **não é necessário** que a base seja um número inteiro.

O mesmo tipo de algoritmo funcionaria para:

- Computar x^n, onde x é número de ponto flutuante;
- Computar A^n, onde A é uma matriz e * é uma multiplicação de matrizes;
- Computar x*x*x*x, onde x é qualquer elemento e * é um operador associativo.

Tudo isso com complexidade $O(\log(n)*f)$, onde f é o custo de uma única aplicação do operador *.

Considerações Finais

A forma mais utilizada da “divisão e conquista” em competições é o princípio da **busca binária**.

Gaste um tempo praticando as várias maneiras de aplicá-lo e poderá se dar muito bem quando acontecer.