Conteúdo

Chapter 1

Recursão

1.1 Introdução

A decomposição é umas das quatro habilidades principais da Pensamento Computacional. A decomposição é uma habilidade que consiste em dividir um problema complexo em partes menores que são mais fáceis de entender.

A recursão pode ser entendida como uma caso particular de decomposição. Na recursão, um problema complexo é decomposto em parte(s) menores do mesmo problema. Nós podemos usar a recursão para definir sequências, funções, conjuntos e algoritmos.

O processo de recursão aplicado a um problema pode ser definido da seguinte maneira:

- Se um problema pode ser resolvido diretamente, então resolva-o.
- Caso contrário, reduza o problema em uma ou mais instâncias menores do mesmo problema.

1.2 Definições recursivas de funções

Considere o seguinte exemplo:

Exemplo 1.1. Uma linha de quadrados é construída usando palitos de fósforo, como mostrado na figura abaixo.

Quantos palitos são necessários para a construção de uma linha de n quadrados?

Seja $f(n)$ o número de palitos necessários para construir uma linha de n quadrados.

A definição recursiva pode ser separada em dois casos:

Figura 1.1 Quadrados de palitos

7

- **Passo Base:** Especifica o valor da função para o menor valor.
- **Passo Recursivo:** Apresenta uma regra para resolver o problema maior a partir de casos menores do mesmo problema.

Observando a Figura 1.2, notamos que toda vez que retiramos 3 palitos de uma linha com n quadrados, obtemos uma linha com $n-1$ quadrados e com 4 palitos obtemos uma linha com 1 quadrado. Acabamos de especificar as regras para tratar os casos complexos e os casos simples.

A definição recursiva da função $f(n)$ é:

- **Passo Base:** $f(1) = 4$
- **Passo Recursivo:** $f(n) = f(n-1) + 3$

Exemplo 1.2. João trabalha no supermercado, e seu gerente pediu que ele empilhasse latas de ervilhas como na figura abaixo.

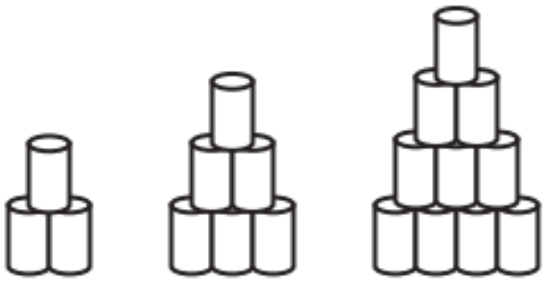


Figura 1.2 Pilha de latas no supermercardo

Proponha uma definição recursiva para o número de latas necessárias para construir uma pilha de latas nesse formato com altura de n latas.

Seja $p(n)$ o número de latas necessárias para construir uma pilhas de latas no formato acima com uma altura n.

Novamente, precisamos identificar o(s) caso(s) mais simples e como reduzir uma caso complexo em um ou mais casos simples do mesmo problema. A pilha mais simples de ser construída é a pilha de altura 1. Observe também que uma pilha de latas com altura n possui n latas na base. A remoção dessas n latas da base reduz a altura da pilha de latas em 1 unidade.

Dessa maneira, a definição recursiva para $p(n)$ será:

- **Passo Base:** $p(1) = 1$
- **Passo Recursivo:** $p(n) = p(n-1) + n$

Exemplo 1.3. Imagine que você queira construir uma parede de tijolos de comprimento n e altura 2 com tijolos com o comprimento 1 e altura 2 que podem ser rotacionados. Calcule de quantos padrões podemos construir uma parede com comprimento n [1].

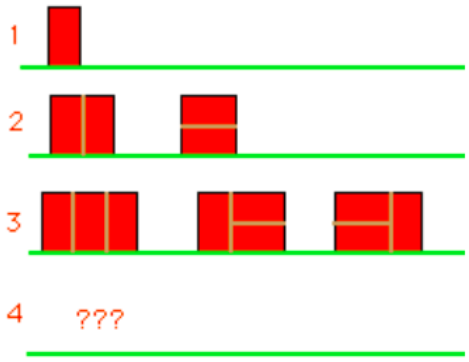
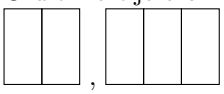
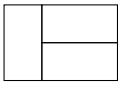


Figura 1.3 Padrões de construções da parede com tijolos 1 x 2

Seja $t(n)$ o número de maneira de construir uma parede $n \times 2$ de comprimento n com altura 2 usando apenas o tijolo 2x1 que podem ser rotacionados.

[1] Problema UVA 900

Primeiramente, vamos investigar como reduzir esse problema para obter instâncias menores do mesmo problema. Observando os padrões de construção da parede 3 x 2 na Figura 1.3, podemos concluir que os últimos tijolos podem ser colocado de duas maneiras diferente:

- O último tijolo em pé

- Os dois últimos tijolos deitados


No primeiro caso, removendo o último tijolo, reduzimos o comprimento da parede em 1 unidade. No segundo caso, removendo os dois últimos tijolos, reduzimos o comprimento da parede em 2 unidades.

Neste problema, os problemas que podem ser resolvidos diretamente sem o uso das regras acima como caso simples. Observe, novamente na Figura 1.3, as paredes de comprimento 1 e 2 podem ser resolvidas diretamente.

Dessa maneira, a definição recursiva para $t(n)$ será:

- **Passo Base:** $t(1) = 1$
- **Passo Base:** $t(2) = 2$
- **Passo Recursivo:** $t(n) = t(n-1) + t(n-2)$

Exemplo 1.4. Você passa em uma loja perto de sua casa e vê a seguinte oferta: Uma garrafa de chococola para cada 3 garrafas vazias devolvidas. Você decide comprar algumas garrafas de cola nessa loja, agora você quer saber quantas garrafas de chococola você pode beber se você aproveitar essa promoção sempre que possível?[2]

A Figura 1.2 mostra quantas garrafas de chococola você pode beber comprando inicialmente 8 garrafas de chococola.

Neste problema, utilizaremos a variável n para guardar o número de garrafas cheias e a variável m para guardar o número de garrafas vazias.

O nosso problema é encontrar uma definição recursiva para a função $cola(n, m)$ que devolve o número de garrafas que você pode beber aproveitando essa promoção sempre que possível considerando que você tem inicialmente n garrafas cheias e m garrafas vazias.

[2]UVA 11877 - The Coco-Cola Store

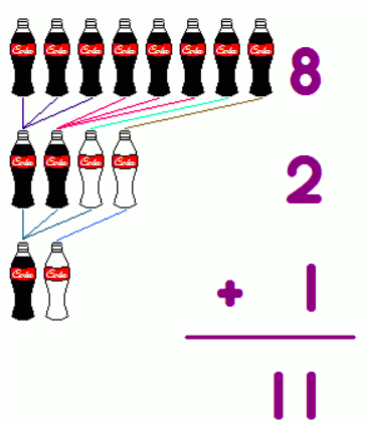


Figura 1.4 Promoção chococola

Agora, precisamos encontrar uma maneira genérica para reduzir uma entrada do problema para uma entrada mais simples do mesmo problema. Considere que você tem 5 garrafas cheias e 3 garrafas vazias, você pode beber 5 garrafas cheias. Depois, você vai ficar com 8 garrafas vazias. Utilizando a promoção, você pode conseguir mais 2 garrafas cheias e 2 vazias.

Os casos mais simples desse problema ocorre quando $n+m < 3$. Nestes casos, você pode beber apenas n garrafas de chococola.

Dessa maneira, a definição recursiva para $cola(n, m)$ será:

- **Passo Base:** $cola(n, m) = n$, se $n + m < 3$
- **Passo Recursivo:** $cola(n, m) = n + coca(\lfloor (n + m)/3 \rfloor, (n + m) \mod 3)$

Exemplo 1.5. De quantas maneiras distintas n objetos distintos podem ser organizados em uma sequência de tamanho n?

- 1 objeto pode ser organizado em uma sequência de tamanho 1 de 1 maneira: A;
- 2 objetos podem ser organizados em uma sequência de tamanho 2 de 2 maneiras: BA, AB;
- 3 objetos podem ser organizados em uma sequência de tamanho 3 de 6 maneiras: BCA, CBA, ACB, CAB, ABC e BAC;

Encontraremos a definição recursiva para a função $p(n)$ que devolve o

número de maneiras que podemos organizar n objetos distintos em uma sequência de tamanho n.

O caso mais simples desse problema ocorre quando temos apenas 1 objeto distinto para organizar em uma sequência de tamanho 1. Novamente, precisamos descobrir uma maneira de reduzir um problema complexo em uma versão mais simples do mesmo problema. Propositalmente, as sequências de tamanho 3 foram listadas de uma maneira peculiar. Começamos listando as sequências terminadas em A, depois B, e assim por diante. Observe que se temos n objetos distintos, temos n opções para o último objeto. Toda vez que utilizamos 1 objeto na última posição, temos que resolver o problema de organizar $n-1$ objetos distintos em uma sequência de tamanho $n-1$.

Dessa maneira, a definição recursiva para $p(n)$ será:

- **Passo Base:** $p(1) = 1$
- **Passo Recursivo:** $p(n) = n * p(n-1)$

Exemplo 1.6. De quantas maneiras distintas n objetos distintos podem ser organizados em uma sequência de tamanho k?

- 4 objetos (A,B,C e D) podem ser organizados em uma sequência de tamanho 2 de 12 maneiras: BA, CA, DA, AB, CB, DB, AC, BC, DC, AD, BD e CD. ;

Encontraremos a definição recursiva para a função $p(n, k)$ que devolve o número de maneiras que podemos organizar n objetos distintos em uma sequência de tamanho k.

O caso mais simples desse problema ocorre quando temos apenas n objeto distinto para organizar em uma sequência de tamanho 1, obtendo n maneiras distintas.

Novamente, precisamos descobrir uma maneira de reduzir um problema complexo em uma versão mais simples do mesmo problema. Propositalmente, as sequências de tamanho 3 foram listadas de uma maneira peculiar. Começamos listando as sequências terminadas em A, depois B, e assim por diante. Observe que se temos n objetos distintos, temos n opções para o último objeto. Toda vez que utilizamos 1 objeto na última posição, temos que resolver o problema de organizar $n-1$ objetos distintos em uma sequência de tamanho $k-1$.

Dessa maneira, a definição recursiva para $p(n)$ será:

- **Passo Base:** $p(n, 1) = n$
- **Passo Recursivo:** $p(n, k) = n * p(n - 1, k - 1)$

Utilizando a definição acima, temos que:

- p(4,2) = 4 * p(3,1) = 4 * 3 = 12

Exemplo 1.7. Ao subir a escada de seu prédio, José às vezes sobe dois degraus de uma vez e às vezes sobe um de cada vez. Sabendo que a escada tem n degraus, de quantas maneiras diferentes José pode subir a escada?

Primeiramente, vamos pensar nos casos menores:

- Uma escada de 1 degrau, podemos subir as escadas de 1 maneira (1).
- Uma escada de 2 degrau, podemos subir as escadas de 2 maneiras (1+1,2).
- Uma escada de 3 degraus, podemos subir as escadas de 3 maneiras (1+1+1,1+2,2+1).

Seja $f(n)$ número de maneira de subir uma escada de n degraus. O primeiro passo para subir uma escada de n degraus pode ser dado de duas maneiras:

- Se você subir apenas um degrau, então teremos $f(n - 1)$ maneiras de subir uma escada de $n - 1$ degraus.
- Se você subir dois degraus, então teremos $f(n - 2)$ maneiras de subir uma escada de $n - 2$ degraus.

Consideramos como case mais simples, os casos que não precisamos usar a regra acima para resolver o problema.

Dessa maneira, a definição recursiva para $f(n)$ será:

- **Passo Base:** $f(1) = 1$
- **Passo Base:** $f(2) = 2$
- **Passo Recursivo:** $f(n) = f(n - 1) + f(n - 2)$

1.3 Definições recursivas de conjuntos

Muitos conjuntos podem ser definidos com o auxílio da recursão.

Exemplo 1.8. Dado um alfabeto Σ, encontre uma definição do conjunto de todas as palavras formada no alfabeto Σ com tamanho n, denotado por Σ^n. Por exemplo, se $\Sigma = \{a, b, c\}$ então

- $\Sigma^0 = \{\varepsilon\}$, onde ε denota a string vazia.
- $\Sigma^1 = \{a, b, c\}$
- $\Sigma^2 = \{aa, ab, ac, ba, bb, bc, ca, cb, cc\}$

Naturalmente, o caso mais simples acontece quando $n = 0$. Novamente, precisamos encontrar uma maneira de reduzir o problema, note que em uma palavra de tamanho n, o primeiro caractere é um símbolo de Σ seguido de uma palavra de tamanho $n - 1$.

Dessa maneira, a definição recursiva para Σ^n será:

- **Passo Base:** $\Sigma^0 = \{\varepsilon\}$
- **Passo Recursivo:** $\Sigma^n = \{ax : a \in \Sigma, x \in \Sigma^{n-1}\}$

Exemplo 1.9. Dado um alfabeto Σ, encontre uma definição do conjunto de todas as palavras formada no alfabeto Σ com tamanho n, denotado por Σ^*. Por exemplo, se $\Sigma = \{a, b\}$ então $\Sigma^* = \{\varepsilon, a, b, aa, ab, ba, bb, \ldots\}$

Neste caso, a definição recursiva indica uma regra para a construção de novos elementos a partir de elementos anteriores. A definição recursiva de Σ^* é :

- **Passo Base:** $\varepsilon \in \Sigma^*$
- **Passo Recursivo:** Se $x \in \Sigma^*$ então $ax \in \Sigma^*, \forall a \in \Sigma$

Exemplo 1.10. Uma palavra é dita palíndroma quando ela pode ser lida da esquerda para a direita ou vice-versa sem alteração na palavra: osso, ana, renner.

Dado um alfabeto Σ, encontre uma definição do conjunto de todas as palavras palíndroma formada no alfabeto Σ com tamanho n , denotado por Σ^n. Seja $Pal(\Sigma, n)$ o conjunto de todas as palavras palíndroma no alfabeto Σ de tamanho n. Por exemplo, se $\Sigma = \{a, b, c\}$ então

- $Pal(\Sigma, 0) = \{\varepsilon\}$, onde ε denota a string vazia.
- $Pal(\Sigma, 1) = \{a, b, c\}$
- $Pal(\Sigma, 2) = \{aa, bb, cc\}$

Primeiramente, vamos investigar como reduzir esse problema. Em palavra palíndroma de tamanho n, o primeiro e o último caractere devem ser

iguais e palavra obtida pela remoção desses caracteres deve ser palíndroma. Note que este processo pode ser realizado até chegar em um string vazia no caso da palavra osso ou em uma string de tamanho 1 no caso da palavra ana.

Dessa maneira, a definição recursiva para $Pal(\Sigma, n)$ será:

- **Passo Base:** $Pal(\Sigma, 0) = \{\varepsilon\}$
- **Passo Base:** $Pal(\Sigma, 1) = \Sigma$
- **Passo Recursivo:** $Pal(\Sigma, n) = \{axa : a \in \Sigma, x \in Pal(\Sigma, n-2)\}$

Exemplo 1.11. Dado um alfabeto Σ, encontre uma definição do conjunto de todas as palavras palíndroma formada no alfabeto Σ. Por exemplo, se $\Sigma = \{a, b\}$ então $Pal(\Sigma) = \{\varepsilon, a, b, aa, bb, aba, aaa, bab, bbb, \ldots\}$

Note que o conjunto $Pal(\Sigma)$ não é finito. Neste caso, a definição recursiva vai apresentar uma regra para construção de novas palavras palíndromas a partir de palavras palíndromas prévias.

A definição recursiva de $Pal(\Sigma)$ é :

- **Passo Base:** $\varepsilon \in Pal(\Sigma)$
- **Passo Recursivo:** Se $x \in Pal(\Sigma)$ então $axa \in Pal(\Sigma), \forall a \in \Sigma$

Exemplo 1.12. Dado um cojunto S, o conjunto de todos os subconjuntos de S será denotado por $\mathcal{P}(S)$. Por exemplo,

- $\mathcal{P}(\emptyset) = \{\emptyset\}$
- $\mathcal{P}(\{1\}) = \{\emptyset, \{1\}\}$
- $\mathcal{P}(\{1, 2\}) = \{\emptyset, \{2\}, \{1\}, \{1, 2\}\}$
- $\mathcal{P}(\{1, 2, 3\}) = \{\emptyset, \{2\}, \{1\}, \{1, 2\}, \{3\}, \{2, 3\}, \{1, 3\}, \{1, 2, 3\}\}$

Primeiramente, vamos tentar pensar em como reduzir esse problema. Note que os subconjuntos de cada conjunto foram listados de maneira particular para facilitar enxergar o padrão de formação do conjunto $\mathcal{P}(S)$. Precisamos fazer duas observações importantes sobre subconjuntos de $\mathcal{P}(S \cup \{x\})$:

- Todo subconjunto de S é um subconjunto de $S \cup \{x\}$
- Todo subconjunto de S unido com o elemento x é um subconjunto de $S \cup \{x\}$.

Com essas duas observações, podemos definir recursivamente $\mathcal{P}(S)$ da seguinte maneira:

- **Passo Base:** $\mathcal{P}(\emptyset) = \{\emptyset\}$
- **Passo Recursivo:** $\mathcal{P}(S \cup \{x\}) = \mathcal{P}(S) \cup \{X \cup \{x\} | X \subseteq \mathcal{P}(S)\}$

1.4 Definições recursivas de operações

Até o momento, definimos de maneira recursiva funções e conjunto. Nesta seção, vamos definir operações sobre conjuntos definidas de maneira recursiva. Os números naturais são um exemplo de uma estrutura definida de maneira recursiva:

- **Passo Base:** $0 \in \mathbb{N}$
- **Passo Recursivo:** Se $x \in \mathbb{N}$ então $x + 1 \in \mathbb{N}$

Exemplo 1.13. Dado dois números naturais a e n, a^n é o resultado da multiplicação de n cópias de a. Por exemplo,

- $2^0 = 1$
- $2^1 = 2$
- $2^2 = 2 * 2 = 4$
- $2^3 = 2 * 2 * 2 = 8$

Novamente, precisamos descobrir uma maneira de reduzir um problema mais complexo em uma instância mais simples do mesmo problema. Observe que 2^3 pode ser escrito como $2^2 * 2$. O mesmo vale para todo expoente, ou seja, a^{n+1} pode ser escrito como $a^n * a$.

Dessa maneira, a definição recursiva para a^n será:

- **Passo Base:** $a^0 = 1$
- **Passo Recursivo:** $a^{n+1} = a^n * a$

Exemplo 1.14. Dado uma palavra $x \in \Sigma^*$, defina recursivamente a função $l : \Sigma^* \to \mathbb{N}$ tal que $l(x)$ devolve o tamanho da palavra x.

A definição recursiva da função l será:

- **Passo Base:** $l(\varepsilon) = 0$
- **Passo Recursivo:** se $x \in \Sigma^*$ e $a \in \Sigma$ então $l(xa) = l(x) + 1$

Exemplo 1.15. Dado duas palavra $x \in \Sigma^*$ e $y \in \Sigma^*$, defina recursivamente a operação $x \circ y$ representando a concatenação das duas palavras x e y. Por exemplo,

- se $x = 010$ e $y = 1100$ então $x \circ y = 0101100$
- se $x = 010$ e $y = 1100$ então $y \circ x = 1100010$

Neste texto, utilizamos como processo de construção de uma nova palavra indicando o primeiro caractere e o restante sendo uma palavra em Σ^*. No processo de construção do resultado da concatenação, usaremos o mesmo padrão de construção.

Observando os exemplos apresentados, o primeiro caractere da concatenação de $x \circ y$ é o primeiro caractere de x e o restante da concatenação seria o resultado da concatenação do restante de x com a palavra y. Utilizando essa observação, podemos definir a operação de concatenação da seguinte maneira:

- **Passo Base:** $\varepsilon \circ y = y$
- **Passo Recursivo:** Se $x \in \Sigma^*$ e $a \in \Sigma$ então $ax \circ y = a(x \circ y)$

1.5 Algoritmos recursivos

Um algoritmo recursivo é dito recursivo quando ele resolve uma instância de um problema reduzindo para uma instância do mesmo problema com uma instância menor.

Exemplo 1.16. Construa um algoritmo recursivo para calcular $n!$. O algoritmo do fatorial de n é definido da seguinte maneira:

- $fat(n) = 1$, se $n = 0$
- $fat(n) = n * fat(n - 1)$, caso contrário.

Agora, precisamos transformar essa definição recursiva de maneira que o computador possa executar. Os algoritmos são independentes de uma linguagem de programação específica. Contudo, optamos por apresentar os algoritmos na linguagem C++:

```
int fat(int n){
        if(n==0) return 1;
        else return n*fat(n-1);
}
```

Internamente, o computador utilizar uma estrutura de dados chamada de pilha para executar um algoritmo recursivo.

O site Pythontutor permite também visualizar o processo de execução de um algoritmo recursivo de maneira simplificada. Durante a execução, várias instâncias da função fact ficam ativas. Bem parecido com o processo de resolução executado manualmente.

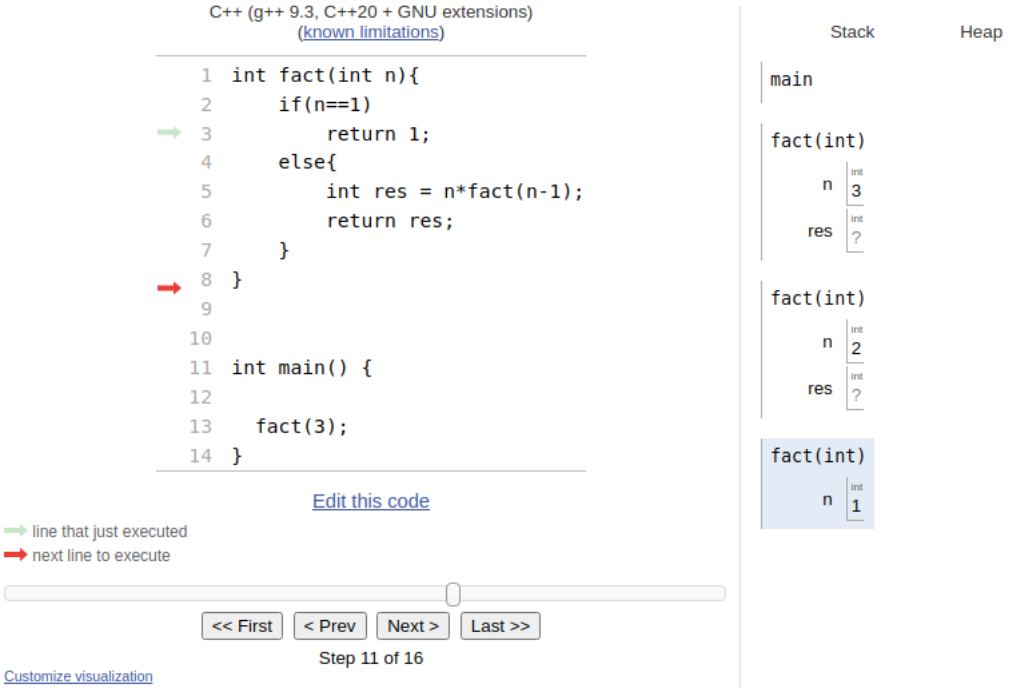


Figura 1.5 site pythontutor.com

Exemplo 1.17. O algoritmo de Euclides é um algoritmo que foi descrito no seu livro Elementos (300 A.C). O problema original era encontrar a maior medida comum capaz de medir dois comprimentos BA e DC.

A Figura 1.5 ilustra o processo sugerido por Euclides para encontrar o maior medida comum de dois comprimentos BA e DC. Inicialmente, você começa com dois comprimentos iniciais BA e DC, ambos definidos como múltiplos de uma unidade de comprimento comum. Queremos encontrar o valor do maior comprimento capaz de medir os dois comprimentos. Inicialmente, tentamos medir o comprimento BA usando o comprimento DC.

Nesse processo de medição, obtemos uma sobra que vale EA. Nesse momento, podemos perceber que se encontramos uma medida que mede EA e DC, podemos usar a mesma medida para medir BA e DC.

Aplicado aos números naturais, o algoritmo de Euclides para encontrar o máximo divisor comum de a e b, denotado por mdc(a,b), pode ser definido recursivamente da seguinte maneira:

- $mdc(a, b) = a$, se $b = 0$
- $mdc(a, b) = mdc(b, a \mod b)$, caso contrário.

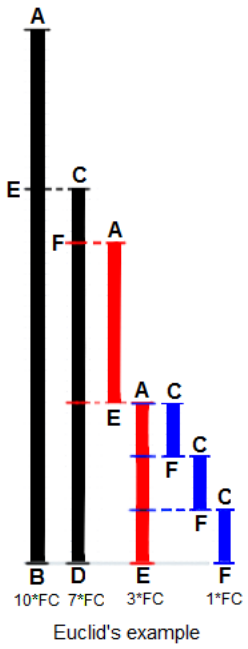


Figura 1.6 Algoritmo de Euclides

O algoritmo recursivo para calcular o máximo divisor comum de a e b é:

```
int mdc(int a, int b){
        if(b==0) return a;
        else return mdc(b, a%b);
}
```

Exemplo 1.18. O algoritmo binário do máximo divisor comum foi publicado pelo físico e programador Josef Stein em 1967, mas pode ter sido conhecido no século II AC, na China Antiga.

A Figura 1.5 mostra a visualização geométrica do algoritmo binário do máximo divisor comum. O algoritmo binário do máximo divisor comum de a e b pode ser descrito pelas seguintes regras:

- mdc(a, 0) = a
- mdc(0, b) = b
- mdc(2u, 2v) = mdc(u,v)
- mdc(2u, v) = mdc(u,v), se v é ímpar.
- mdc(u, 2v) = mdc(u,v), se u é ímpar.
- mdc(u, v) = mdc(|u-v|, min(u,v)), se u e v são ímpares

Uma implementação do algoritmo binário do máximo divisor comum em C++:

```
int binarygcd(int a, int b){
    if(b == 0) return a;
    if(a == 0) return b;
    if( a % 2 == 0){
        if( b%2 == 0){
            return 2*binarygcd(a/2, b/2);
        }else{
            return binarygcd(a/2, b);
        }
    }else{
        if( b%2 == 0){
            return binarygcd(a, b/2);
        }else{
            return binarygcd( abs(a-b), min(a,b));
        }
    }

}
```

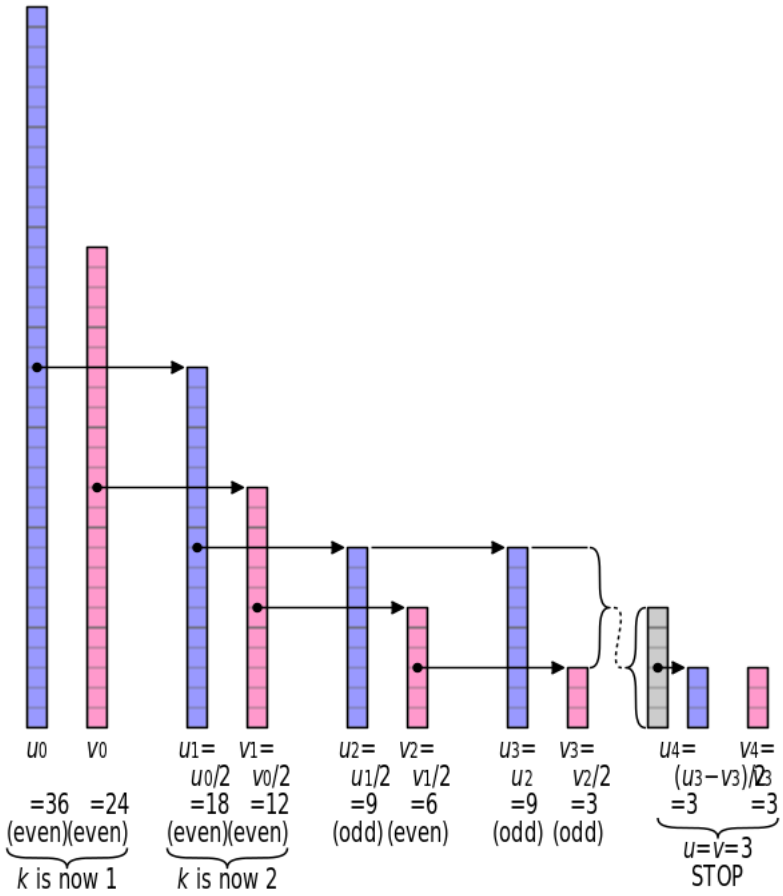


Figura 1.7 Algoritmo de Euclides

Chapter 2

Divisão e Conquista

A divisão e conquista é uma das técnicas mais utilizadas para resolução de problemas computacionais. O esquema geral para resolução de problemas utilizando essa técnica consiste nos seguintes passos:

- Divisão: Um problema é dividido em vários subproblemas do mesmo tipo com aproximadamente o mesmo tamanho.
- Os subproblemas são resolvidos de maneira recursiva.
- Se necessário, as soluções do subproblemas são combinadas para conseguir uma solução do problema original.

2.1 Mergesort

Exemplo 2.1. Dado um vetor não ordenado $A[0 \dots n-1]$, ordene o vetor utilizando o algoritmo de mergesort.

No algoritmo mergesort, um vetor $A[start \dots end]$ de tamanho maior que 1 é ordenado da seguinte maneira:

- O vetor é dividido em duas metades:
  - $A[start \dots \lfloor (start + end)/2 \rfloor]$
  - $A[\lceil (start + end)/2 \rceil \dots end]$
- Cada metade é resolvida utilizando o próprio algoritmo mergesort.
- Em seguida, os dois vetores ordenados são combinados para produzir um único vetor ordenado.

O algoritmo do mergesort é descrito em pseudocódigo pelo Algoritmo 2.1

Algorithm 2.1 Mergesort

Require: Um vetor $A[start \ldots end]$ com elementos que podem ser comparados e dois inteiros $start$ e end

Ensure: O subvetor $A[start \ldots end]$ ordenado em ordem não decrescente.

```
if q − p > 1 then
    mid ← ⌊(p + q)/2⌋
    mergesort(A, start, mid)
    mergesort(A, mid + 1, end)
    merge(A, start, mid, end)
end if
```

O processo de entrelaçamento dos dois vetores ordenado pode ser descrito da seguinte maneira: dois índices são inicializados no ínicio de cada vetor ordenado. Os elementos apontados pelos índices são comparados e o menor deles é adicionado no novo vetor, o índice do menor elemento é incrementado. Essa operação é realizada até que um dos vetores acabe seus elementos. Em seguida, os elementos do vetor não vazio são copiados para o novo vetor. No final, o novo vetor é copiado para o vetor orginal. O algoritmo de merge dos dois vetores ordenados está descrito em pseudocódigo no Algoritmo 2.4.

O vetor é modificado durante a execução do algoritmo de merge, vamos mostrar o processo de ordenação mostrando todas as transformações realizadas pelo algoritmo merge considerando a seguinte entrada $[1, 2, 1, 3, 5, 3, 2, 1]$:

Função	Entrada	Vetor resultado
merge	[1], [2]	[1,2]
merge	[1] ,[3]	[1,3]
merge	[1,2], [1,3]	[1,1,2,3]
merge	[5] ,[3]	[3,5]
merge	[2] ,[1]	[1,2]
merge	[3,5], [1,2]	[1,2,3,5]
merge	[1,1,2,3] ,[1,2,3,5]	[1,1,1,2,2,3,3,5]

Muitos problemas podem ser resolvidos facilmente quando recebemos um vetor já ordenado. No próximo exemplo, adaptaremos o algoritmo de mergesort para executar uma tarefa que pode ser realizada de maneira bastante rápida quando consideramos o vetor ordenado.

Algorithm 2.2 Merge

Require: Um vetor $A[start \ldots end]$ com elementos que podem ser comparados e três inteiros *start*, *mid* e *end*

Ensure: O subvetor $A[start \ldots end]$ ordenado em ordem não decrescente.

```
start1 ← start
start2 ← mid + 1
start3 ← 0
Seja B um vetor de tamanho end − start + 1
while start1 <= mid and start2 <= end do
    if A[start1] < A[start2] then
        B[start3] ← A[start1]; start1 ← start1 + 1
    else
        B[start3] ← A[start2]; start2 ← start2 + 1
    end if
    start3 ← start3 + 1
end while
while start1 <= mid do
    B[start3] ← A[start1]; start1 ← start1 + 1; start3 ← start3 + 1
end while
while start2 <= mid do
    B[start3] ← A[start2]; start2 ← start2 + 1; start3 ← start3 + 1
end while
k ← 0
for i ← start to end do
    A[i] ← B[k]; k ← k + 1
end for
```

2.2 Mergecount

Exemplo 2.2. Dado um vetor não ordenado $A[0 \ldots n-1]$, devolva a quantidade elementos repetidos no vetor. Por exemplo,

- Dado A = 1,2,1,3,5,3,2,1, temos 4 repetições.
- Dado A = 1,2,1,3,5,3,2,1,2, temos 5 repetições.

No algoritmo mergecount, um vetor $A[start \ldots end]$ de tamanho maior que 1 é ordenado da seguinte maneira:

- O vetor é dividido em duas metades:

- $A[start \dots \lfloor (start + end)/2 \rfloor]$
- $A[\lceil (start + end)/2 \rceil \dots end]$

- Em cada metade, contaremos a quantidade de repetições de elementos em cada metade.
- Em seguida, contaremos a quantidade de repetições de elementos que aparecem em metades distintas.

Algorithm 2.3 Mergecount

Require: Um vetor $A[start \dots end]$ com elementos que podem ser comparados e dois inteiros *start* e *end*

Ensure: O subvetor $A[start \dots end]$ ordenado em ordem não decrescente e devolve a quantidade de repetições no subvetor $A[start \dots end]$.

```
if q − p > 1 then
    mid ← ⌊(p + q)/2⌋
    p1 ← mergecount(A, start, mid)
    p2 ← mergecount(A, mid + 1, end)
    p3 ← merge(A, start, mid, end)
    return p1 + p2 + p3
else
    return 0
end if
```

Agora, precisamos adaptar o algoritmo de merge para contar a quantidade de elementos repetidos entre vetores ordenados.

Todas as repetições são contadas pelo algoritmo de merge, vamos mostrar o resultado de todas as contagem realizada pelo algoritmo merge considerando a seguinte entrada $[1, 2, 1, 3, 5, 3, 2, 1]$:

Função	Entrada	Número de repetições
merge	[1], [2]	0
merge	[1] ,[3]	0
merge	[1,2], [1,3]	1
merge	[5] ,[3]	0
merge	[2] ,[1]	0
merge	[3,5], [1,2]	0
merge	[1,1,2,3] ,[1,2,3,5]	3

O total de repetições calculada pelo Algoritmo é 4.

Algorithm 2.4 Merge

Require: Um vetor $A[start \dots end]$ com elementos que podem ser comparados e três inteiros $start$, mid e end

Ensure: O subvetor $A[start \dots end]$ ordenado em ordem não decrescente.

```
start1 ← start
start2 ← mid + 1
start3 ← 0
Seja B um vetor de tamanho end − start + 1
cont ← 0
while start1 <= mid and start2 <= end do
    if A[start1] < A[start2] then
        B[start3] ← A[start1]; start1 ← start1 + 1
    else
        if A[start1] == A[start2] then
            cont ← cont + 1
        end if
        B[start3] ← A[start2]; start2 ← start2 + 1
    end if
    start3 ← start3 + 1
end while
while start1 <= mid do
    B[start3] ← A[start1]; start1 ← start1 + 1; start3 ← start3 + 1
end while
while start2 <= mid do
    B[start3] ← A[start2]; start2 ← start2 + 1; start3 ← start3 + 1
end while
k ← 0
for i ← start to end do
    A[i] ← B[k]; k ← k + 1
end for
return cont
```

2.3 Quicksort

Exemplo 2.3. Dado um vetor não ordenado $A[0 \dots n-1]$, ordene o vetor utilizando o algoritmo de quicksort.

O algoritmo de quicksort é um outro exemplo de algoritmo de divisão e conquista. O algoritmo do quicksort pode ser descrito da seguinte maneira:

- Utilizando um elemento particular do vetor chamado de pivô,o vetor é particionado em duas partes: os elementos menores que o pivô e os elementos maiores que o pivô.
- Cada parte é ordenado de maneira recursiva.

O pseudocódigo do algoritmo do quicksort é o seguinte:

Algorithm 2.5 Quicksort

Require: Um vetor $A[start \dots end]$ com elementos que podem ser comparados e dois inteiros *start* e *end* representando índices do vetor A.
Ensure: O subvetor $A[start \dots end]$ ordenado em ordem não decrescente.

```
if q − p > 1 then
    j ← particiona(A, start, end)
    quicksort(A, start, j − 1)
    quicksort(A, j + 1, end)
end if
```

Algorithm 2.6 particiona(A, start, end)

Require: Um vetor $A[start \dots end]$ com elementos que podem ser comparados e dois inteiros *start* e *end* representando índices do vetor A.
Ensure: devolve o índice j tal que $A[i] \leq piv \forall i \leq j$ e $A[i] \geq piv \forall i \geq j$

```
j ← start
pivo ← A[end]
for k ← start to end-1 do
    if A[k] ≤ pivo then
        A[k] ↔ A[j]
        j ← j + 1
    end if
end for
A[j] ↔ A[end]
return j
```

2.4 Quickcount

Exemplo 2.4. Dado um vetor não ordenado $A[0 \dots n-1]$, devolva a quantidade elementos repetidos no vetor adaptando o algoritmo do quicksort. Por exemplo,

- Dado A = 1,2,1,3,5,3,2,1, temos 4 repetições.
- Dado A = 1,2,1,3,5,3,2,1,2, temos 5 repetições.

O adaptação do algoritmo do quicksort pode ser descrito da seguinte maneira:

- Utilizando um elemento particular do vetor chamado de pivô,o vetor é particionado em três partes: os elementos menores que o pivô, os elementos iguais ao pivô e os elementos maiores ou iguais que o pivô.
- Cada parte é ordenado de maneira recursiva e devolve a quantidade de elementos repetidos em cada parte.

Algorithm 2.7 Quickcount

Require: Um vetor $A[start \ldots end]$ com elementos que podem ser comparados e dois inteiros *start* e *end* representando índices do vetor A.
Ensure: O subvetor $A[start \ldots end]$ ordenado em ordem não decrescente.

```
if end − start > 1 then
    particiona2(A, start, end, start1, end1)
    p1 ← quicksort(A, start, start1 − 1)
    p2 ← quicksort(A, end1 + 1, end)
    return p1 + p2 + end1 - start1
else
    return 0
end if
```

O algoritmo particiona2 chama o algoritmo particiona para separa em duas partes. Em seguida, particiona a segunda parte em mais duas partes. O ínicio e o fim da parte central é devolvida pela função particiona2 através das variáveis start1 e end1.

Todas as repetições são contadas pelo algoritmo particiona2, vamos mostrar o resultado de todas as contagem realizada pelo algoritmo particiona2 considerando a seguinte entrada $[1, 2, 1, 3, 5, 3, 2, 1]$:

Algorithm 2.8 particiona2(A, start, end, start1, end1)

Require: Um vetor $A[start \ldots end]$ com elementos que podem ser comparados e dois inteiros *start* e *end* representando índices do vetor A.

Ensure: devolve o índice j tal que $A[i] \leq piv \forall i \leq j$ e $A[i] \geq piv \forall i \geq j$

```
start1 ← particiona(A, start, end)
j ← start1
pivo ← A[j]
for k ← start1 to end do
    if A[k] == pivo then
        A[k] ↔ A[j]
        j ← j + 1
    end if
end for
end1 ← j
```

Entrada	Pivô	Particões	Número de repeticões
$[1,2,1,3,5,3,2,1]$	1	$[], [1,1,1], [3,5,3,2,2]$	2
$[3,5,3,2,2]$	2	$[], [2,2], [3,5,3]$	1
$[3,5,3]$	5	$[3,3], [5], []$	0
$[3,3]$	3	$[], [3,3], []$	1

O total de repetições calculada pelo Algoritmo é 4.

Chapter 3

Algoritmo Guloso

Um algoritmo é dito guloso quando ele consegue construir uma solução ótima fazendo a cada passo uma decisão gulosa. Em geral, os algoritmos gulosos realizam uma ordenação da entrada para tornar a decisão gulosa mais eficiente.

Exemplo 3.1. Dado um sistema de moedas M = <1,5,10,25,50> um inteiro N. Encontre $(\alpha_1, \alpha_2, \ldots, \alpha_m)$ tal que

$$\begin{aligned} & min \sum_{i=1}^{m} \alpha_i \\ & st. \sum_{i=1}^{m} \alpha_i * c_i = N \\ & \alpha_i \in \mathbb{Z} \end{aligned}$$

Uma estratégia gulosa para o problema acima seria tentar utilizar primeiro as moedas com o maior valor primeiro.

Bibliography