Recursão e complexidade de algoritmos

Conceito de algoritmo e procedimentos

O conceito matemático por trás da versatilidade dos computadores é o de algoritmo, cuja origem remonta à era clássica, mas que ganhou muita força no século XIX, com a sua adoção nos programas de computador. Mas o que é um algoritmo?

O algoritmo é uma sequência finita de passos bem definidos para solucionar determinado problema.

Caracteristicas:

1. O algoritmo deve ser finito, ou seja, ele eventualmente deve parar de executar e chegar à conclusão se solucionou ou não o problema que deveria resolver.

2. O algoritmo deve ter passos bem definidos, ou seja, não pode haver dúvidas sobre o que deve ser feito a cada passo executado.

3. O algoritmo deve estar associado à resolução de um problema.

Muitas vezes, o programa de computador se utilizará de um ou mais algoritmos na sua execução. Além disso, os programas associam esses algoritmos a algum tratamento de dados de entrada, isto é, cada execução será diferente, pois os dados iniciais que servem de entrada para o algoritmo mudam também em cada uma delas.

Exemplo: algoritmo simples

Matematicamente, o problema de ordenação recebe como entrada uma sequência de números naturais distintos em qualquer ordem e como saída deve entregar a mesma sequência de números, porém ordenada em ordem crescente.

Esse mero construto matemático, entretanto, pode ser reescrito para resolver problemas mais próximos da realidade. Vamos imaginar um baralho de cartas de jogo, com 52 cartas, no qual você escreve os números naturais. Depois você pode separar um subconjunto dessas cartas e embaralhá-las. Essa seria a entrada do problema. A saída será você terminar com o mesmo baralho com as cartas sobrepostas em ordem crescente.

Confira um algoritmo simples para resolver esse problema é:

1. Pegue a carta do topo do baralho de entrada (carta atual).

2. Compare a carta atual com a carta do topo do baralho de saída:
se a da saída for maior, então coloque a carta atual no topo do baralho de saída, senão repita o passo 2 com a próxima carta do baralho de saída.

3. Se o baralho de entrada estiver vazio, então o algoritmo acabou, senão repita o passo 1.

Analisando, podemos ver que algoritmo é um conjunto finito de passos, e ele terminará quando o baralho de entrada acabar, ou seja, todas as cartas tiverem sido ordenadas. Entretanto, o “algoritmo” não está bem definido. O que acontece quando pegamos a primeira carta do baralho de entrada e o baralho de saída está vazio? Perceba que não explicamos o que acontece nesse caso. Como você pode resolver esse problema?

Existe mais de uma maneira de resolver o problema, e essa diferença mostra um pouco como a criatividade na hora de criar um algoritmo é importante. O jeito mais simples é criar mais uma cláusula de condição (**se... então**) substituindo o segundo passo:

Se o baralho de saída estiver vazio, **então** coloque a carta atual no topo do baralho de saída; **se não** , compare a carta atual com a carta do topo do baralho de saída. **Se** a da saída for maior, **então** coloque a carta atual no topo do baralho de saída; **se não** , repita o passo 2 com a próxima carta do baralho de saída.

Outro jeito envolve criar uma abstração, uma carta "virtual" de valor infinito que começa presente no baralho de saída. Quando o algoritmo termina, você entrega a saída sem essa carta de valor infinito. Você consegue ver como essa abstração resolve o problema sem alterar o algoritmo original?

Vamos ver o algoritmo em execução.
Entrada: [1,15,13,2] Saída: ∞
Carta atual: 1
Atual é menor que a do topo da saída? Sim, pois 1<∞.
Então coloque a carta atual no topo.

Entrada: [15,13,2] Saída: [1,∞]
Carta atual: 15
Atual é menor que a do topo da saída? Não, pois 15>1.
Próxima carta da saída: ∞. A carta atual é menor? Sim, pois 15<∞. Então, coloque a carta atual no topo.

Entrada: [13,2] Saída: [1,15,∞]
Carta atual: 13
Atual é menor que a do topo da saída? Não, pois 13>1.
Próxima carta da saída: 15 . A carta atual é menor? Sim, pois 13<15. Então, coloque a carta atual no topo.

Entrada: [2] Saída: [1,13,15,∞]
Carta atual 2
Atual é menor que a do topo da saída? Não, pois 2>1
Próxima carta da saída: 13 . A carta atual é menor? Sim, pois 2<13. Então, coloque a carta atual no topo.

Entrada: [] Saída: [1,2,13,15,∞]
A entrada está vazia, logo a resposta é: [1,2,13,15]
O algoritmo está completo.

Procedimentos e funções

Embora o conceito de algoritmo seja matemático, você provavelmente vai trabalhar com os construtos que utilizam os algoritmos, os programas de computador. Para facilitar o

entendimento do programa (ou dos algoritmos), os programadores costumam organizar pedaços do código que são repetidos várias vezes em um procedimento ou função.

Um procedimento é um conjunto de comandos organizado sob um "nome" que costumamos denominar de chamada de procedimento. Geralmente, fica em outra parte do código do programa, facilitando a compreensão geral do programa e podendo ser chamado em diversas ocasiões distintas pelo programa.

Já a função é um tipo de procedimento que, ao seu término, retorna algum valor para o programa que o chamou, o que denominamos "retorno da função".

Continuação do exemplo: procedimentos e funções

Considere que temos um novo programa que executa a ordenação que vimos no exemplo anterior diversas vezes em sequência. Como ele se pareceria?

1. **Embaralhe** as cartas.

2. **Ordene** as cartas.

3. **Repita** a partir do passo 1.

Esse simples programa fica embaralhando e ordenando as cartas repetidamente. Mas o que significa a frase "Ordene as cartas"? Podemos entender essa linha como uma chamada ao procedimento de ordenar as cartas, que descrevemos no exemplo anterior.

Desse modo, o programa principal fica mais simples de entender, e você pode se deparar com outro programa que precise embaralhar as cartas. Nesse caso, você também já tem o procedimento que realiza essa tarefa pronta, bastando chamá-lo novamente.

Como exemplo de função, você pode criar um procedimento que devolve como saída o menor número que está na sequência. **Lembre-se de que uma função é um tipo especial de procedimento que retorna um valor como sua saída.**

Você consegue escrever tal função? Como ela se parece se for executada sobre um baralho ordenado? E como ela se parece quando executada em um baralho desordenado?

Caso o baralho esteja ordenado, em ordem crescente, a função é muito simples, retorne a carta do topo do baralho. No caso desordenado, precisaremos percorrer todo o baralho, guardando qual foi o menor número visto nesse percurso. Esse número é a saída da função. Veja o exemplo:

1. Comece com o menor valor valendo infinito.

2. Se o baralho acabou, então retorne o menor valor.

3. Compare o menor valor com a carta do topo: se a carta for menor, então atualize o menor valor com o valor da carta.

4. Passe para a próxima carta e repita a partir do passo 2.

A diferença entre as duas formas de criar a função leva à nossa próxima discussão. Qual desses jeitos é melhor? Como podemos medir e comparar dois algoritmos, procedimentos ou funções distintas que resolvem o mesmo problema?

Complexidade de algoritmos

Conceito de complexidade de algoritmo

Complexidade de algoritmo é uma aproximação do tempo de execução do algoritmo, calculado segundo algumas simplificações para permitir comparar a qualidade de algoritmos independentemente do equipamento sobre o qual ele está executando.

Mas o que vamos comparar? No nosso algoritmo de ordenação, cada comparação pode ser considerada de custo de tempo T. Vamos calcular três tipos diferentes de complexidade para nosso exemplo.

Complexidade de melhor caso:

A complexidade de melhor caso é o tempo aproximado de execução do algoritmo quando a entrada é a melhor possível para o algoritmo.

No nosso exemplo da ordenação, imagine qual caso seria o melhor possível. Qual baralho de entrada é o mais positivo para o algoritmo?

Estranhamente, o melhor caso para o nosso algoritmo de ordenação é o baralho de entrada estar ordenado exatamente na ordem decrescente.

Imagine que o baralho de entrada tem **T** cartas. Assim, a primeira carta será comparada com o baralho de saída vazia, levando tempo **T**. A próxima carta será comparada com a do topo e será menor, logo será ali colocada, levando tempo **T**. Isso se repetirá para cada carta, levando um total de **T**+**T**+**T**…+**T** (repetido **T** vezes), isto é, **nT**. A essa complexidade chamamos de linear, pois depende do tamanho de entrada **T**, de uma forma linear (o expoente de **T** é 1).

A complexidade de melhor caso raramente é usada como parâmetro para comparação de algoritmos, principalmente porque o melhor caso por diversas vezes é de solução trivial.

Comlexidade de caso médio:

A complexidade de caso médio é o tempo aproximado de execução do algoritmo quando a entrada apresenta um comportamento próximo da média dificuldade. Esse tipo de cálculo envolve o uso de certas técnicas estatísticas, principalmente o conceito de valor esperado, também chamado de esperança ou média.

No nosso exemplo da ordenação, a cada carta avaliada do baralho de entrada, quantas comparações precisarão ser feitas?

Na primeira carta, haverá apenas uma comparação, pois o baralho de saída está vazio (tempo **T**).

A partir da segunda carta, existe uma chance de precisar de uma comparação e uma chance de precisar de duas comparações. Assumindo que as chances são iguais, a esperança é de que precisaremos de 1,5 comparações, ou seja, em média tempo 1,5 **T**.

Isso pode ser repetido para as próximas cartas, gerando tempos esperados de 2**T**, 2,5**T** e assim por diante, até **nT/**2.

O tempo total do caso médio então será representado pela soma **T**+1,5**T**+ 2**T**+…+**nT/**2.

Calculando chegamos a: **n**(**n**+1)**T/**4

A complexidade de caso médio é usada eventualmente, mas não é a mais comum por depender de avaliações estatísticas que às vezes dificultam seu cálculo.

Complexidade de pior caso:

A complexidade de pior caso é o tempo aproximado de execução do algoritmo quando a entrada é a pior possível para este.

No nosso exemplo da ordenação, imagine qual caso seria o pior possível. Qual baralho de entrada é o mais negativo para o algoritmo?

Você consegue calcular o tempo de execução nesse caso, com uma entrada de **N** cartas já ordenadas?

A primeira carta será colocada no topo, custando 1 comparação (tempo **T**). A segunda será comparada com a primeira, será maior, depois será comparada com o baralho vazio e colocada ali (tempo **2T**). Seguirá assim por diante, até que a última carta seja comparada com todas as outras, o que leva tempo **nT**. O tempo total será de 1**T**+2**T**+…+(**n**−1)**T**+**nT**. Calculando chegamos a: **n(n+1)T/2**.

Essa complexidade chamamos de quadrática, pois o termo dominante é $\mathbf{n^2}$.

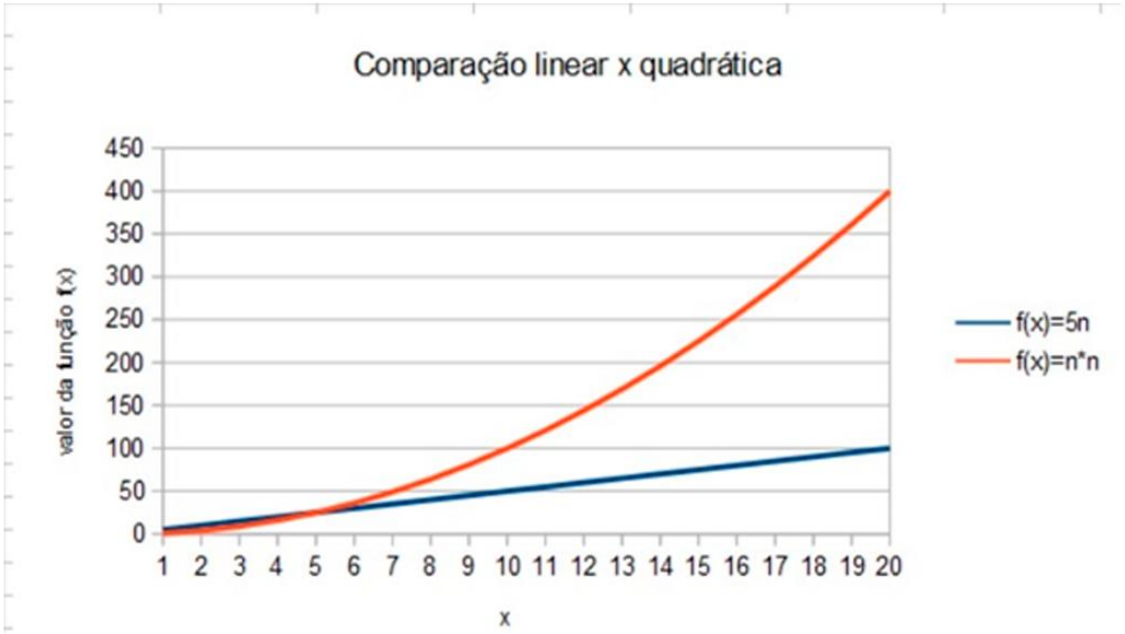
A complexidade de pior caso é a mais comumente usada na comparação de algoritmos.

Complexidade assintótica:

Quando falamos de complexidade de algoritmos, normalmente queremos analisar a dificuldade de resolver problemas com entradas grandes. Dessa forma, você pode prever o comportamento conforme o tamanho da entrada aumenta de forma arbitrária.

A diferença entre um bom algoritmo linear e um algoritmo quadrático pode ser pequena para uma entrada de tamanho pequeno (5, por exemplo), mas é excepcionalmente grande para uma entrada de tamanho 1.000.

Para ilustrar a diferença, pode-se analisar o gráfico a seguir, de uma função linear **5n** e uma quadrática $\mathbf{n^2}$.



Como você pode perceber, eventualmente há um tamanho de entrada em que a função quadrática é sempre maior que a linear, e isso vale para qualquer constante multiplicativa finita do fator linear.

Por isso, na complexidade de algoritmos, costumamos analisar o comportamento assintótico, ou seja, o comportamento a partir de um tamanho de entrada arbitrariamente grande, após o qual o comportamento das funções fica estabilizado.

No cálculo de complexidade assintótica, usamos uma aproximação, em que se descartam multiplicadores do tamanho da entrada e elementos constantes ao final do cálculo.

Por exemplo, uma complexidade calculada de 3**n**T+5 será aproximada por **nT**. Você pode dizer que o termo **n** é dominante na complexidade, pois os outros termos serão muito menores que **n** quando **n** é grande o bastante.

Notação O

Do conceito de complexidade assintótica de algoritmo surge a notação para permitir a fácil comparação entre complexidades. A notação O serve para comparar assintoticamente duas funções.

Dizemos que uma função **F** é **O**(**g**) quando **cg**(**n**)≥**F**(**n**) a partir de um determinado **n** arbitrariamente grande e uma constante **C**>0. Traduzindo, se **f** é **O**(**g**), então a função **g**(**n**) é "maior" que **F**(**n**) para um valor suficientemente grande de **n** (às vezes, é necessário multiplicar por uma constante finita **c**).

Você consegue ver como podemos utilizar isso na complexidade de algoritmos? A complexidade de um algoritmo de pior caso pode ser diretamente expressa com a notação **O**. Por exemplo, dizemos que um algoritmo é **O(g)** quando sua complexidade de pior caso é linear.

No exemplo de ordenação, nosso algoritmo de ordenação é **O**(**n²**) , pois sua complexidade de pior caso é dominada pelo termo quadrático **n²**.

Recursão

Função recursiva

Uma função recursiva é aquela que chama a si mesma dentro do seu código, com outros parâmetros de entrada.

Para utilizar recursão, você precisa tomar cuidado para não chamar a si mesmo infinitamente. Por isso, toda função recursiva deve ter um caso base, no qual ela retorna uma saída sem chamar a si mesma para alguma entrada.

Aplicando recursão – fatorial

Você pode calcular fatorial de **n** multiplicando **n**(**n**−1) ! e assim por diante, até chegar a 1!—— que é o nosso caso base, retornando o valor 1.

Ao escrever isso como uma função, você terá:

1. se n=1, então retorne 1.

2. retorne n*fatorial(n-1) }.

Perceba que no passo 2 a função chama a si mesma. A chamada da função fatorial(4), por exemplo, fará chamada a fatorial(3), que chamará fatorial(2) e que chamará fatorial(1), que por sua vez começará a retornar valores que serão multiplicados até o cálculo da chamada original.

A chamada básica retorna um número, e a chamada recursiva executa uma multiplicação de dois números.

Se você assumir que a multiplicação leva um tempo Θ fixo, basta calcular o número de chamadas recursivas feitas.

Para uma entrada **n**, são chamados fatorial de (**n**−1),(**n**−2),… até fatorial de 1. Há um total de (**n**−1) chamadas e multiplicações, logo a complexidade é de (**n**− 1) **T**, ou seja, o cálculo é **O**(**g**), ou linear.

Aplicando recursão – sequência de Fibonacci

No segundo exemplo, você deve calcular o valor do n-ésimo termo da sequência de Fibonacci. Essa sequência é [1,1,2,3,5,8,13,21,...], na qual cada elemento da sequência é igual à soma dos dois anteriores.

Como a própria definição da sequência é recursiva, fica fácil escrevê-la como uma

função recursiva:

função fibonacci(n){

1. se n=1 ou n=2, então retorne 1

2. retorne fibonacci(n-1)+ fibonacci(n-2)}

Perceba que no passo 2 a função chama a si mesma duas vezes. A chamada da função fibonacci(4), por exemplo, fará chamada a fibonacci(3) e fibonacci(2). Já fibonacci(3) chamará fibonacci(2) e fibonacci(1).

Você consegue calcular a complexidade de pior caso dessa função? Cada passo 2 envolve a soma de dois números, que consideramos levar tempo **T**. Então, basta calcular o número de chamadas de fibonacci(n) **com n**>2.

Para facilitar o cálculo, podemos assumir que a linha 1 da função também leva tempo **T**, que alterará muito pouco no cálculo final, assim basta calcularmos o número de chamadas da função como um todo.

Para fibonacci(4) há 4 chamadas, para fibonacci(5) há 8 chamadas, para fibonacci(6) há 12 chamadas, e assim por diante. Essa sequência cresce exponencialmente, motivo pelo qual dizemos que essa função é $\mathbf{O}(\mathbf{2^n})$ ou, ainda, que tem complexidade exponencial.

A complexidade exponencial torna um algoritmo impraticável de executar para tamanhos crescentes de forma acelerada. Com **n**=20, já seriam necessárias milhões de operações; com **n**=100, o tempo de execução já estaria na casa de alguns anos.

Esse é um exemplo de que o uso de recursão nem sempre é a opção correta. O valor de um termo da sequência de Fibonacci pode ser calculado diretamente por meio de uma fórmula simples.