

Paradigmas e projetos de algoritmos

- Algoritmos recursivos
- Força bruta
- Gulosos

Algoritmos recursivos: Algoritmos recursivos são algoritmos que chamam a si mesmo como por exemplo uma árvore de busca.

Quando não utilizar recursão.

Em resumo, evite algoritmos recursivos quando o desempenho, uso de memória ou legibilidade forem preocupações significativas, ou quando a recursão não oferecer benefícios claros em relação a abordagens iterativas.

pythonCopy code

```
def contador_infinito():  
    print("Contando...")  
    contador_infinito() # Chama a si mesma infinitamente  
  
contador_infinito()
```

Neste exemplo, a função `contador_infinito()` chama a si mesma repetidamente sem qualquer condição de parada. Isso levará a um estouro de pilha, já que cada chamada adiciona um novo contexto de função à pilha de chamadas do programa. Eventualmente, o programa ficará sem espaço na pilha e lançará um erro de estouro de pilha.

É importante sempre ter uma condição de parada em funções recursivas para evitar esse tipo de problema. Sem uma condição de parada adequada, a recursão continuará indefinidamente, consumindo recursos até que o programa falhe.

Faz-se necessário uma *condição de parada*

- Alguma condição que, se satisfeita, finaliza as chamadas recorrentes

Um procedimento recursivo P deve estar sujeito a uma condição de parada B

- Deve-se **garantir** que B não seja satisfeita em algum momento
 - $P! = NULL$

Exemplo de algoritmo não eficiente em recursão é o **cálculo de fibonacci**

cálculo de fibonacci: sequência matemática onde número da série é somado pelos dois numero anteriores.

Algoritmos de força bruta: algoritmos de força bruta (tentativa e erro) tende a encontrar todas as soluções possíveis, ou seja suponha que tem um problema específico, o algoritmo de força bruta vai analisar todas as soluções possíveis e testar todas essas soluções para ter a solução correta final

```
#include <stdio.h>

int encontrar_maximo(int array[], int tamanho) {
    int maximo = array[0]; // Supõe que o primeiro elemento é o máximo
    for (int i = 1; i < tamanho; i++) {
        if (array[i] > maximo) {
            maximo = array[i]; // Atualiza o máximo se encontrar um número maior
        }
    }
    return maximo;
}

int main() {
    int numeros[] = {7, 12, 5, 9, 18, 3};
    int tamanho = sizeof(numeros) / sizeof(numeros[0]);

    int maximo = encontrar_maximo(numeros, tamanho);
    printf("O máximo é: %d\n", maximo);
}
```

Nesse exemplo de algoritmo de força bruta podemos observar que o resultado final vai ser o maior número do array, mas pra isso ele vai "ler" todas as soluções possíveis ou seja testar (o teste nesse caso é a comparação) e entregar o resultado correto no final.

Pergunta da prova 2 de 2023: algoritmos de força bruta são úteis para resolver problemas da classe P, apresenta uma pequena discussão sobre isso.

Sabemos que algoritmo de força bruta tem o objetivo de analisar todas as soluções possíveis e entregar o resultado final, algoritmos da classe P são problemas que podem ser resolvidos de forma polinomial em máquinas determinísticas, algoritmos de força bruta pode ser usado para resolver problemas na classe P, como por exemplo encontrar o maior número de um array, então sim, existem algoritmos que podem ser utilizados e são eficientes em algoritmos da classe P embora não seja a forma mais eficiente de ser utilizada, e principalmente se o tamanho da entrada algoritmo for grande.


pergunta bônus: algoritmos de força bruta são úteis para resolver problemas da classe NP

Sabemos que problemas da classe NP está na classe de problemas exponencial ou seja tempo de execução de forma não polinomial em comparação ao tamanho da entrada, algoritmos de força bruta são utilizados para testar todos os problemas possíveis para achar a solução, com isso percebemos que utilizar força bruta para analisar problemas da classe NP pode ser utilizado mas vai levar muito tempo para chegar em uma solução.

Algoritmos gulosos:

Imagine que você está em uma loja de doces com a quantia limitada de dinheiro e quer comprar a maior quantidade possível de doces. Você pode usar uma abordagem gulosa para alcançar seu objetivo. Aqui está como funciona:

1. **Escolha imediata:** Em cada passo, você escolhe o doce mais barato disponível. Isso é uma escolha "gulosa" porque você não está pensando no preço dos doces futuros, apenas no que é mais barato agora.
2. **Continuar escolhendo:** Você continua escolhendo os doces mais baratos até que não tenha mais dinheiro suficiente para comprar outro doce.
3. **Fim:** Quando não puder mais comprar doces, você termina e conta quantos doces conseguiu comprar.

Esse é um exemplo básico de um algoritmo guloso. Em cada etapa, você faz a escolha que parece ser a melhor naquele momento, esperando  que essa escolha leve a uma solução globalmente ótima.

O algoritmo guloso só faz a melhor escolha para o problema atual(problema imediato), o algoritmo não se importa com o problema global, ele faz escolhas ótimas a fim de chegar em uma solução ótima

Os algoritmos gulosos são como "pensamento de curto prazo". Eles não olham para o quadro completo do problema, apenas para a situação imediata. Às vezes, essa abordagem funciona e leva a uma solução ótima, mas em outros casos pode levar a uma solução subótima.

Os algoritmos gulosos são frequentemente utilizados em problemas de otimização, onde você está tentando maximizar ou minimizar algo, como no exemplo dos doces. No entanto, é importante lembrar que nem todos os problemas podem ser resolvidos com algoritmos gulosos, e é necessário cuidado para garantir que essa abordagem leve a uma solução correta e eficiente.

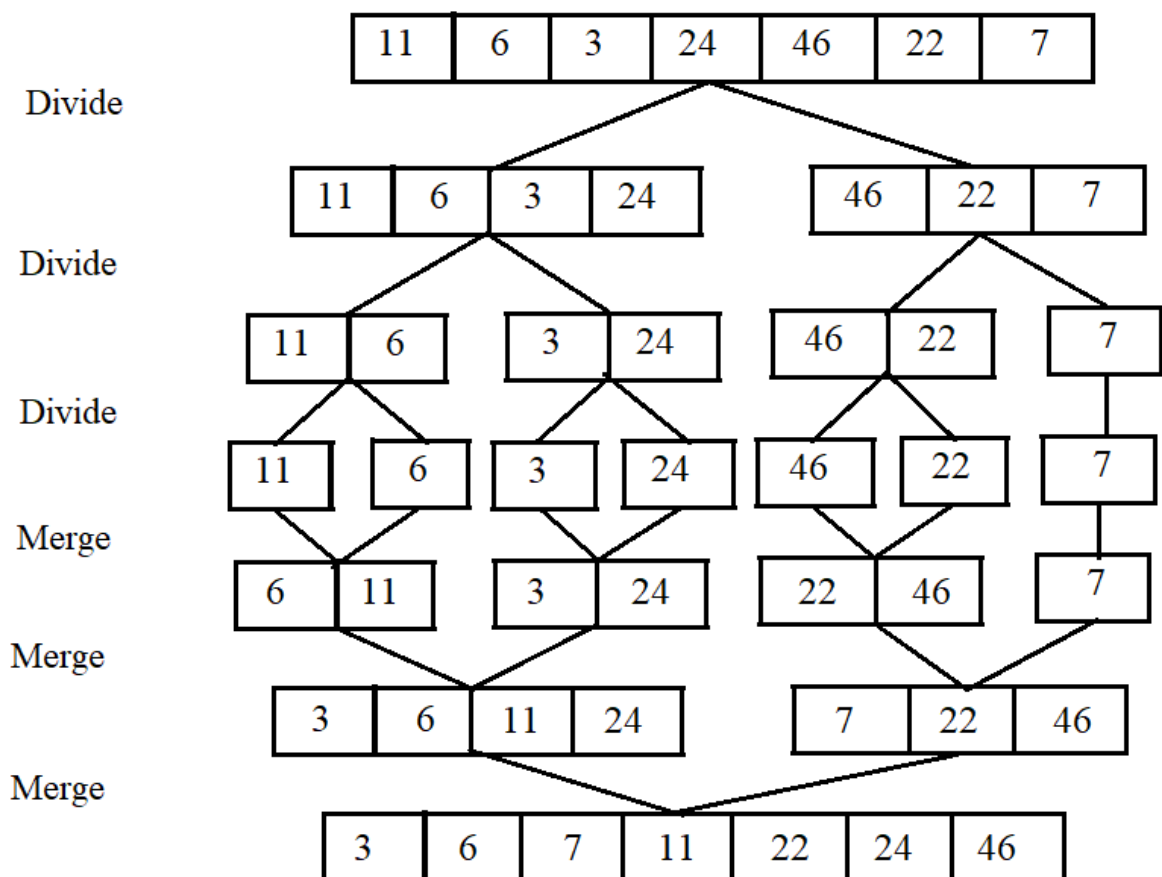
Pergunta da prova 2 de 2023: Quais são os pontos fracos e fortes de algoritmos gulosos, apresente uma pequena discussão sobre o tema

algoritmos gulosos tem o objetivo de fazer a melhor escolha de forma imediata, ou seja ele no algoritmo ele tem que fazer escolhas, ele analisa a melhor escolha no momento e aceita sem analisar os problemas futuros, o ponto forte é que racionalmente isso leva a uma solução ótima ou sub ótima e que também como são escolhas imediatas isso leva a um tempo de execução mais rapido do problema principal e são algoritmos eficientes em problemas de otimização, o ponto fraco é ele não garante que essas escolhas ótimas imediatas não vai atrapalhar escolhas futuras e também não consideram escolhas passadas feita no algoritmo e dependem do tipo de estrutura utilizado no problema para ser utilizado o método guloso.

problemas a ser analisados : problema da mochila binária, problema do caixeiro viajante problema da coloração dos grafos

- Divisão e conquista
- Programação dinâmica

Algoritmos de divisão e conquista: como o próprio nome já diz é o ato de você dividir e conquistar, esse tipo de algoritmo tem como o objetivo de você pegar um problema grande no momento e dividir em pequenas partes sendo que o resultado obtido da parte principal que foi dividida tem que ter tamanhos parecidos e o resultado deve ser simples.



vantagens e desvantagens de utilizar divisão e conquista:

vantagens: problemas muito grandes ao ser divididos tornam se exponencialmente mais simples, ou seja pega problemas considerados difíceis de forma que fique mais fácil de chegar na solução

são fáceis de entender ou seja de implementar já que é o fato de dividir o problema para chegar na solução final

desvantagens: se você vai dividir um problemas em partes isso significa que vc vai dividir várias vezes ocorrendo uma recursão, isso utilizando mais memória do computador para fazer essas recursão

Nem todos os problemas podem ser resolvidos utilizando divisão e conquista, muitas das vezes vai ser necessário pegar um problema e adaptar para utilizar a divisão ou seja transformando em outro problema para de resolvido isso gastando o tempo do que utilizando outra forma eficiente para resolver.

Programação dinâmica: programação dinâmica aparenta ser parecido com divisão e conquista mas tem detalhes específicos que se destacam em diferença a algoritmos de divisão e conquista.

A programação dinâmica pega um problema e transforma em subproblemas, com isso ao resolver esses subproblemas ele se junta para ter a resolução final.

diferença entre divisão e conquista e programação dinâmica: na divisão e conquista você pega um problema e divide e resolve cada subproblema sem se preocupar com o problema em si, já em programação dinâmica você pode utilizar a solução de um subproblema e utilizar em outros

Resumindo, na divisão e conquista, resolvemos cada subproblema independentemente, enquanto na programação dinâmica, aproveitamos soluções de subproblemas já resolvidos para economizar tempo e evitar cálculos desnecessários.

A definição da estrutura desta tabela é de extrema importância para a eficiência do algoritmo

- Quais serão as dimensões da tabela?
- O que significa cada dimensão da tabela
- Como determinar os índices de entrada da tabela?
- O que será salvo em cada entrada da tabela?
 - Valor da solução?
 - Componentes da solução?

- **Conceitos de grafos e estruturas de dados para representação de grafos**

- **Algoritmos de busca em grafos - profundidade e largura**

▶ Estrutura de Dados - Aula 27 - Grafos - Busca em largura

▶ Estrutura de Dados - Aula 26 - Grafos - Busca em profundidade