Técnicas e Análise de Algoritmos

Busca Completa e Programação Dinâmica - Parte 01

Professor: **Jeremias Moreira Gomes**
E-mail: jeremias.gomes@idp.edu.br

Introdução

Introdução

- Ao resolver problemas computacionais, existem diversos paradigmas que auxiliam na criação eficientes de soluções:
  - Busca Completa
  - Algoritmos Gulosos
  - Divisão e Conquista
  - Programação Dinâmica
  - Busca Aleatória
  - Recursividade
  - Algoritmos Aproximados

Introdução

- Ao resolver problemas computacionais, existem diversos paradigmas que auxiliam na criação eficientes de soluções:
  - Busca Completa
  - Algoritmos Gulosos (visto indiretamente (MST))
  - Divisão e Conquista (visto indiretamente (mergesort))
  - Programação Dinâmica
  - Busca Aleatória
  - Recursividade (visto indiretamente (DFS))
  - Algoritmos Aproximados

Introdução

- Ao resolver problemas computacionais, existem diversos paradigmas que auxiliam na criação eficientes de soluções:
  - **Busca Completa**
  - Algoritmos Gulosos (visto indiretamente (MST))
  - Divisão e Conquista (visto indiretamente (mergesort))
  - **Programação Dinâmica**
  - Busca Aleatória
  - Recursividade (visto indiretamente (DFS))
  - Algoritmos Aproximados

Busca Completa

Busca Completa

- A busca completa, também denominada força bruta, consiste em avaliar todo o espaço de (possíveis) soluções do problema em busca de uma solução
- A complexidade de soluções de busca completa, em geral, são determinadas pelo tamanho do espaço de soluções

Busca Completa

- Este espaço tende a ter um grande número de elementos, de modo que a força bruta é aplicada, com eficiência, em problemas cujo contradomínio seja computacionalmente tratável
- Algoritmos de força bruta, por outro lado, tendem a ter uma implementação simples e direta

Busca Completa

- Exemplo de problema cujo a abordagem pode ser a busca completa:
  - Deseja-se saber, dado um número N, quantos são os inteiros menores que N que são múltiplos de A e B.
    - Exemplo: N = 20, A = 3 e B = 5
    - Limites: N < 1000, A < N, B < N

Busca Completa - Geradores e Filtros

- O uso busca completa é apropriado, dadas as dimensões do espaço de busca
- Na resolução utilizando a busca, existem dois tipos de abordagem
  - Utilizando **filtros**:
    - Olhar todos os elementos e verificar as condições
  - Utilizando **geradores**:
    - Gerar apenas soluções válidas no espaço de busca

Busca Completa - Geradores e Filtros

```
int filtro(int N, int A, int B)
{
    int ans = 0;
    for (int i = 1; i <= N; i++) {
        if (i % A == 0 || i % B == 0) {
            ans++;
        }
    }
    return ans;
}
```

Busca Completa - Geradores e Filtros

```cpp
int filtro(int N, int A, int B)
{
    int ans = 0;
    for (int i = 1; i <= N; i++) {
        if (i % A == 0 || i % B == 0) {
            ans++;
        }
    }
    return ans;
}
```

```cpp
int gerador(int N, int A, int B)
{
    unordered_set<int> s;
    for (int i = 1; i <= N; i += A) {
        s.insert(i);
    }

    for (int i = 1; i <= N; i += B) {
        s.insert(i);
    }

    return s.size();
}
```

Busca Completa

- Problemas de busca completa exigem explorar o espaço de busca
  - Calcular tempo é essencial
  - Possibilidade de aplicar podas quando possível
- Essa classe de problemas também se relaciona com
  - Produto cartesiano
  - Permutações
  - Etc
- Entendimento para otimizar é essencial

Programação Dinâmica

Programação Dinâmica

- A programação dinâmica é um paradigma de solução de problemas que combina características dos outros paradigmas
- É aplicável em problemas que possuem subestrutura ótima
- Ela também resolve o problema através da combinação das soluções dos subproblemas, o que se assemelha à etapa de fusão da divisão e conquista

Programação Dinâmica

- De forma semelhante a busca completa, ela avalia todas as alternativas disponíveis igualmente
- Ela, porém, difere dos demais paradigmas porque evita recalcular um subproblema múltiplas vezes por meio da técnica de memorização, e por optar por uma ou mais alternativas apenas após avaliar todas elas

Programação Dinâmica - Características

- A programação dinâmica é aplicável em problemas que possuem duas características:
  1. Subestrutura ótima - solução do problema pode ser formada a partir das soluções ótimas de seus subproblemas
  2. Sobreposição de subproblemas - problemas compartilham subproblemas em comum (necessita resolver um subproblema múltiplas vezes)

Programação Dinâmica - Características

- Se não houver sobreposição de subproblemas, esse problema é um caso para apenas uma busca completa
- Se a solução é formada a partir da solução de subproblemas, esta solução pode ser descrita por meio de uma relação de recorrência
- Os subproblemas que não podem mais serem subdivididos e que são necessários para a solução dos demais constituem os casos-base do problema

Programação Dinâmica

- Como o objetivo é explorar principalmente a característica de sobreposição de subproblemas, troca-se a utilização de mais memória no programa para ganho de velocidade
  - Todo problema já resolvido é salvo em memória para ser reaproveitado quando for necessário

Programação Dinâmica - Exemplo

- Considere o problema de encontrar o n-ésimo número da sequência de Fibonacci
- A sua definição recursiva é dada por:

$$F(n) = \begin{cases} 0, & \text{se } n = 0 \\ 1, & \text{se } n = 1 \\ F(n-1) + F(n-2), & \text{caso contrário} \end{cases}$$

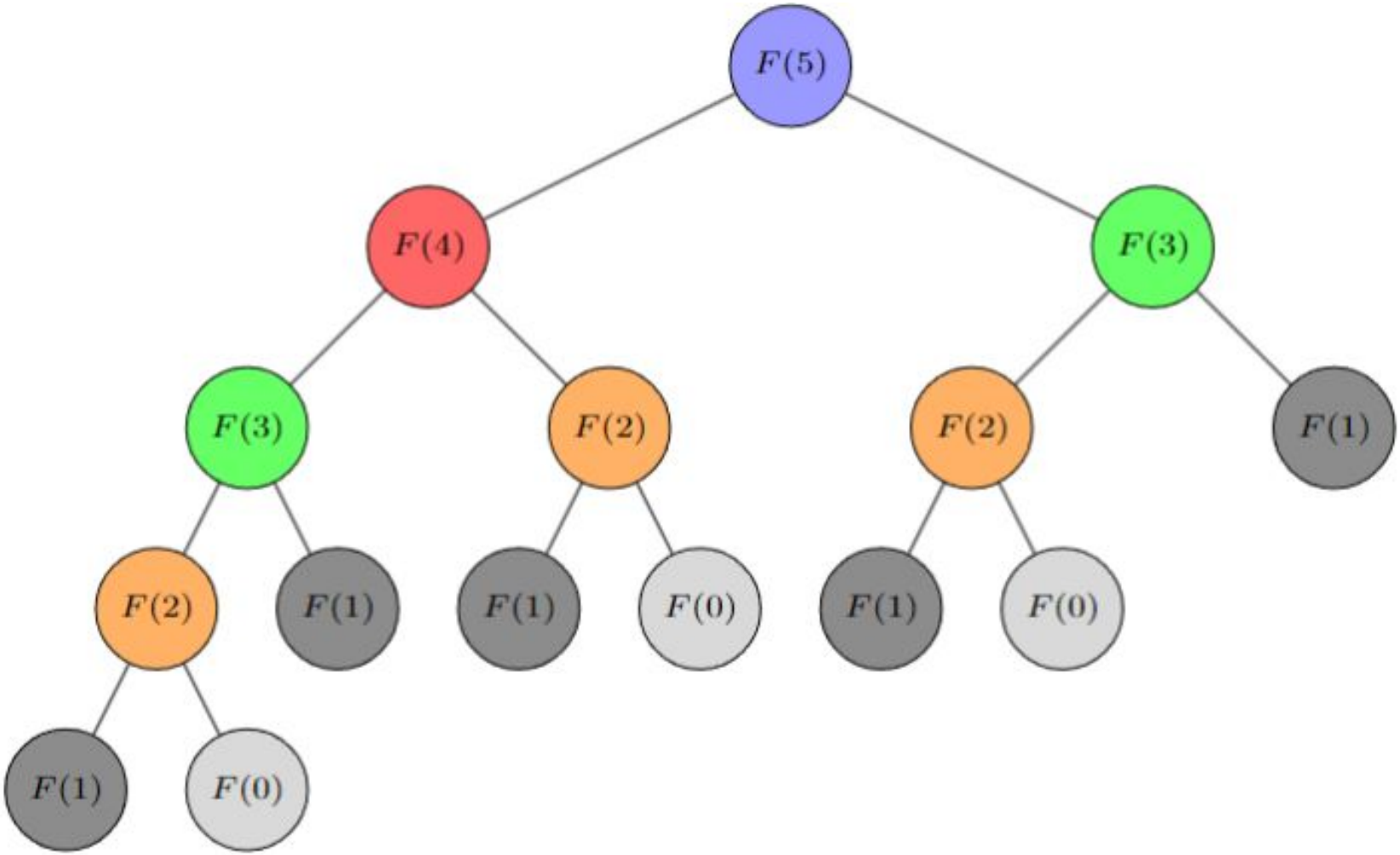
Programação Dinâmica - Exemplo

- A forma simples e recursiva de resolver esse problema, é utilizar uma busca completa, calculando o número de fibonacci de todos os termos menores do que o n procurado

Programação Dinâmica - Exemplo

```
unsigned long long fibonacci(unsigned long long N)
{
    if (N < 2) {
        return N;
    }

    return fibonacci(N - 1) + fibonacci(N - 2);
}
```


F(5)
F(4)
F(3)
F(3)
F(2)
F(2)
F(1)
F(2)
F(1)
F(1)
F(0)
F(1)
F(0)
F(1)
F(0)

Programação Dinâmica - Exemplo

- A solução utilizando a busca completa é da ordem de complexidade de $O(2^n)$
- Pode-se reparar que, a medida que a recursão avança, alguns estados são computados repetidas vezes

Programação Dinâmica - Exemplo

- Esse problema possui as seguintes características:
  - Subestrutura ótima: o n-ésimo número de Fibonacci pode ser computado a partir de números de Fibonacci anteriores
  - Sobreposição de subproblemas: o algoritmo para um mesmo valor é computado repetidas vezes
- Dessa forma, esse é um candidato natural para um algoritmo de programação dinâmica

Programação Dinâmica - Exemplo

- Então, para melhorar a velocidade desse algoritmo, utilizando programação dinâmica, deve-se passar a memorizar cada solução calculada e utilizar essa memória para evitar recálculos

Programação Dinâmica - Exemplo

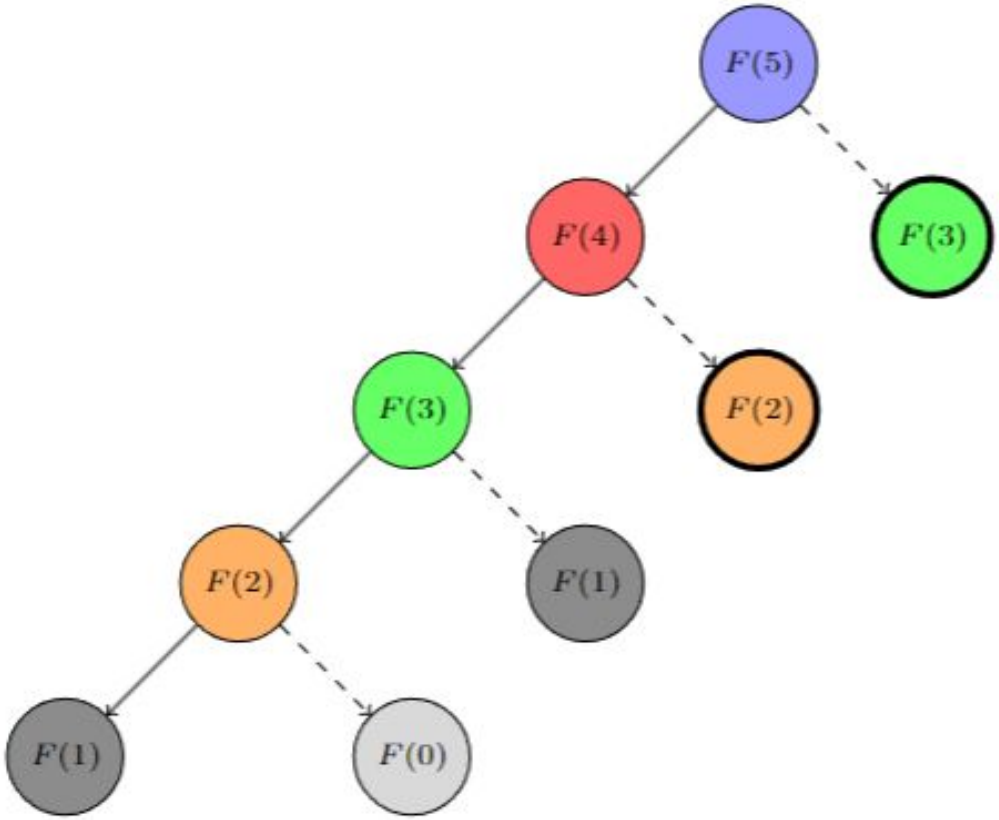
```
unordered_map<unsigned long long, unsigned long long> memo;

unsigned long long fibonacci(unsigned long long N)
{
    if (N < 2) {
        return N;
    }
    if (memo.count(N)) {
        return memo[N];
    }
    memo[N] = fibonacci(N - 1) + fibonacci(N - 2);
    return memo[N];
}
```

Programação Dinâmica - Exemplo

- A implementação utilizando programação dinâmica visita cada estado, no máximo, duas vezes
- Na primeira visita o estado é computado recursivamente, utilizando a mesma recorrência da solução de busca completa
- Na segunda visita o estado já foi computado, e o valor armazenado na tabela é retornado imediatamente

Programação Dinâmica - Exemplo



Programação Dinâmica - Exemplo

- O simples fato de se utilizar alguma memória para guardar cálculos, auxilia na redução da complexidade do algoritmo de exponencial para linear (O(n))
- Observe que, partiu-se de um problema grande (calcular F(N)) e foi-se reduzindo o tamanho do problema até se chegar em um caso caso básico (F(0) ou F(1)) sem recalcular repetições
    - Esse tipo de abordagem é chamado de top-down

Abordagem Top Down

- Uma implementação top-down de um algoritmo de programação dinâmica parte da relação de recorrência para produzir uma função recursiva
- Após a verificação dos casos-base, a tabela é consultada para se determinar se o estado já foi computado ou não

Abordagem Top Down

- Se o estado já tiver sido computado, o valor armazenado na tabela é retornado
- Caso contrário, o subproblema associado ao estado é solucionado por meio de recursão
- Em seguida, a solução obtida é armazenada na tabela e então retornado

Tabela de Memorização

- A tabela de memorização deve ter tamanho suficiente para armazenar todos os estados possíveis (dentro dos limites das variáveis que compõem o estado)
- Além disso, esta tabela deve ser iniciada com um valor que sinalize que o estado associado não foi computado ainda
    - Nesse caso, foi utilizado um hashmap (mas é comum ser uma tabela)

Tabela de Memorização

- Este valor sentinela deve corresponder a um valor que não pode ser uma solução de nenhum estado (-1, ∞, etc)
- Assim, a complexidade em memória do algoritmo será, no mínimo, O(S), onde S é o total de estados possíveis

Abordagem Top-Down - Características

- A principal vantagem da implementação top-down é a simplicidade
  - É uma tradução literal da relação de recorrência
- Outra vantagem é que apenas os estados necessários à solução são computados

Abordagem Top-Down - Características

- Por outro lado, a tabela deve ter dimensões que comportem todos os estados possíveis
- Além custo de execução associado à realização de sucessivas chamadas recursivas
  - Uma forma de se evitar essas chamadas recursivas, é a implementação utilizando a abordagem bottom-up

Abordagem Bottom-Up

- Assim como nas implementações top-down, uma implementação bottom-up também se baseia na relação de recorrência e nos casos-base do problema
- A primeira diferença é que na implementação bottom-up todos os estados intermediários, necessários ou não, são computados

Abordagem Bottom-Up

- Inicialmente os casos-base são preenchidos
- Em seguida, todos os estados que dependem apenas dos casos-base são computados
- Após eles, os estados que podem ser computados a partir dos estados já computados

Abordagem Bottom-Up

```
unsigned long long fibonacci(unsigned long long N)
{
    unsigned long long memo[N + 1];

    // Preenchimento dos casos base
    memo[0] = 0;
    memo[1] = 1;

    for (unsigned long long i = 2; i <= N; i++) {
        memo[i] = memo[i - 1] + memo[i - 2];
    }

    return memo[N];
}
```

Abordagem Bottom-Up

- No caso do problema do fibonacci, a solução para o problema N depende apenas de N - 1 e N - 2, então nesse caso não seria necessário nem guardar todos os estados de cálculos, sendo possível reduzir a memória para constante
  - Em outros problemas, talvez seja necessário manter a memória

Conclusão