Paradigmas de Projeto de Algoritmos

Tópicos II

Thiago Silva Vilela

Recursividade

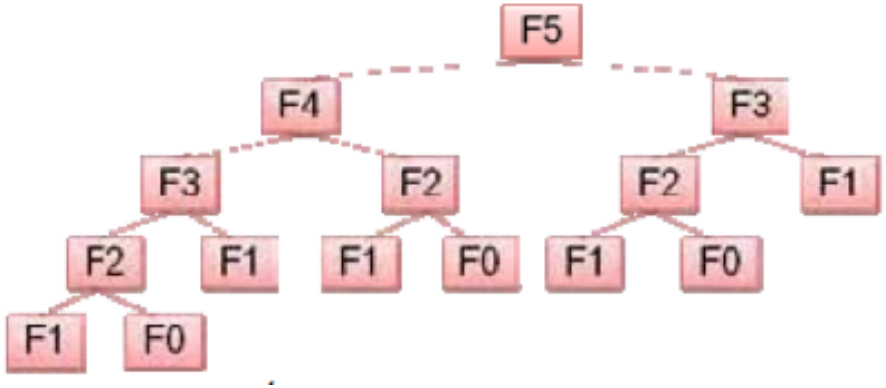
- Abaixo, apresenta-se uma implementação em linguagem funcional para um programa que calcula números de Fibonacci:

```
fib :: Integer -> Integer
fib 0 = 0
fib 1 = 1
fib n = fib(n - 1) + fib(n - 2)
```

Considerando que o programa acima não reutilize resultados previamente computados, quantas chamadas são feitas à função `fib` para computar `fib` 5?

(A) 11 **(B)** 12 **(C)** 15 **(D)** 24 **(E)** 25

Recursividade



- Podemos construir a árvore de chamadas da função gerada pela recursão sobre a função fib para encontrar o resultado.

- Para o argumento 5, teríamos então 15 chamadas à função fib.

Divisão e Conquista

- Consiste em dividir o problema em partes menores, encontrar soluções para as partes, e combiná-las em uma solução global.
- Três etapas:
  - **Divisão**: o problema é dividido em um certo número de subproblemas que são instâncias menores do problema original.
  - **Conquista**: os subproblemas são resolvidos recursivamente (em geral). Se o tamanho do subproblema for pequeno o bastante, no entanto, o subproblema é resolvido de forma direta.
  - **Combinação**: as soluções dos subproblemas são combinadas, formando a solução do problema original.

Divisão e Conquista

- Equações de recorrência estão diretamente relacionadas com o paradigma de divisão e conquista.
- Fornecem uma maneira natural de caracterizar o tempo de execução dos algoritmos de divisão e conquista.
- **Recorrência:** é uma equação ou desigualdade que descreve uma função em termos de de seus valores em entradas menores.

Divisão e Conquista

- A complexidade do Mergesort é dada pela seguinte equação de recorrência:

$$T(n) = \begin{cases} O(1) & \text{if } n = 1 \\ 2T(n/2) + O(n) & \text{if } n > 1 \end{cases}$$

Divisão e Conquista

- No geral, algoritmos de divisão e conquista possuem recorrências da forma:

$$T(n) = aT(n/b) + f(n),$$

- *a* é o número de subproblemas gerados, *b* o tamanho de cada um deles, e *f(n)* o custo de dividir ou combinar o problema.

- Exemplos: Mergesort, pesquisa binária, problema do subarray máximo.

Divisão e Conquista

- Exemplo: Dado um vetor de *n* elementos, determine o maior e o menor elementos desse vetor usando divisão e conquista.

Divisão e Conquista

```
MaxMin(Linf, Lsup, Max, Min)
    if (Lsup – Linf <= 1)        #Condição da parada recursiva
        if (A[Linf] < A[Lsup])
            Max ← A[Lsup]
            Min ← A[Linf]
        else
            Max ← A[Linf]
            Min ← A[Lsup]

    else (Meio ← (Linf+Lsup)/2) #Encontra maior e menor elemento de cada partição
        MaxMin(Linf, Meio, Max1, Min1)
        MaxMin(Meio+1, Lsup, Max2, Min2)
        if (Max1 > Max2)
            Max ← Max1
        else
            Max ← Max2
        if (Min1 < Min2)
            Min ← Min1
        else
            Min ← Min2
```

Divisão e Conquista

- Qual a complexidade do algoritmo anterior?

- Equação de recorrência:

$$f(n) = \begin{cases} 1, & n = 1, \\ f(n/2)+f(n/2), & n > 2 \end{cases}$$

Divisão e Conquista

$T(n) = 2\,T(n/2)$

$= 2\,[2\,T(n/4)]$

$= 4\,T(n/4)$

$= 4\,[2\,T(n/8)]$

$= 8\,T(n/8)$

$= 2^k\,T(n/2^k)$

$n/2^k = 1$ ou $n = 2^k$ ou $\log_2 n = k$

$T(n) = 2^k\,T(n/2^k) = 2^{\log n}\,T(1) = n$

Divisão e Conquista

- Nesse caso, nossa solução por divisão e conquista é pior que uma solução iterativa.
  - Ambas são O(n), mas a recursão acarreta custos adicionais: salva Linf, Lsup, Max e Min, além do endereço de retorno da chamada para o procedimento.

Balanceamento

- É sempre importante tentar manter o balanceamento em problemas de divisão e conquista para que a complexidade seja menor.
  - Exemplo: Ordenação.

Balanceamento

- Exemplo de ordenação (Selection sort):
  - Seleciona o menor elemento de A[1..n] e troca-o com o primeiro elemento A[1].
  - Repete o processo com os n − 1 elementos, resultando no segundo maior elemento, o qual é trocado com o segundo elemento A[2].
  - Repetindo para n−2, n−3, ... ,2 ordena a seqüência.

Balanceamento

- O selection sort pode ser visto como uma aplicação recursiva de divisão e conquista:
  - Encontre o maior elemento de um vetor de *n* elementos;
  - Resolva agora o problema para um vetor de n – 1 elementos.
- O algoritmo possui complexidade $O(n^2)$

Balanceamento

- Como melhorar a eficiência?
  - Dividir o problema em subproblemas de tamanhos parecidos.

- Mergesort:
  - Dividir recursivamente o vetor a ser ordenado em dois, até obter n vetores de 1 único elemento.
  - Aplicar a intercalação tendo como entrada 2 vetores de um elemento, formando um vetor ordenado de dois elementos.
  - Repetir este processo formando vetores ordenados cada vez maiores até que todo o vetor esteja ordenado.

- Possui complexidade O($n$log$n$)

Divisão e Conquista

- Este paradigma não é aplicado apenas a problemas recursivos, apesar de ser normalmente definido em termos de recursão.
- Existem dois principais cenários onde divisão e conquista é aplicada:
  - Processar independentemente partes do conjunto de dados. Exemplo: Mergesort.
  - Eliminar partes do conjunto de dados a serem examinados. Exemplo: Pesquisa binária.

Programação Dinâmica

- É uma espécie de tradução iterativa inteligente da recursão.
  - "Recursão com apoio de uma tabela".

- Cada instância do problema é resolvida a partir da solução de instâncias menores.
  - Ou melhor, de subinstâncias da instância original.

Programação Dinâmica

- Aplicada principalmente a problemas de otimização.
  - Uma série de escolhas deve ser feita;
  - Deseja-se obter a solução ótima.

- Resolve problemas combinando a solução de subproblemas.

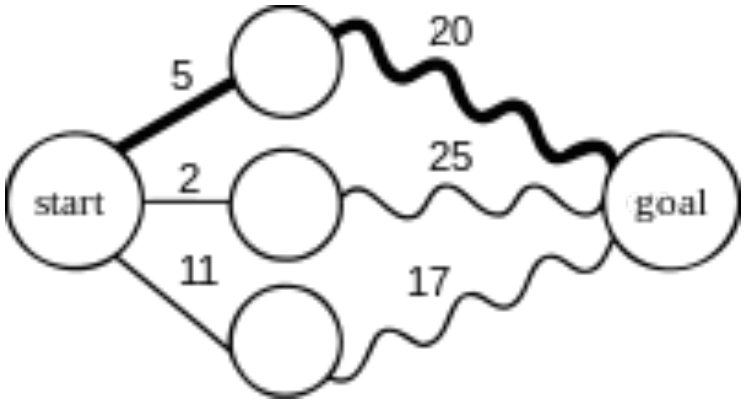
- Resolve cada subproblema somete uma vez e armazena o resultado.
  - Não é necessário recalcular soluções de subproblemas.
  - O fato de possuir memória e ser implementado, geralmente, de forma iterativa, são as maiores diferenças entre programação dinâmica e divisão e conquista.

Programação Dinâmica

- Quando utilizamos programação dinâmica?
  - Número total de instâncias do problema a ser resolvido deve ser pequeno;
  - Problema possui subestrutura ótima;
  - Problema possui subproblemas sobrepostos.

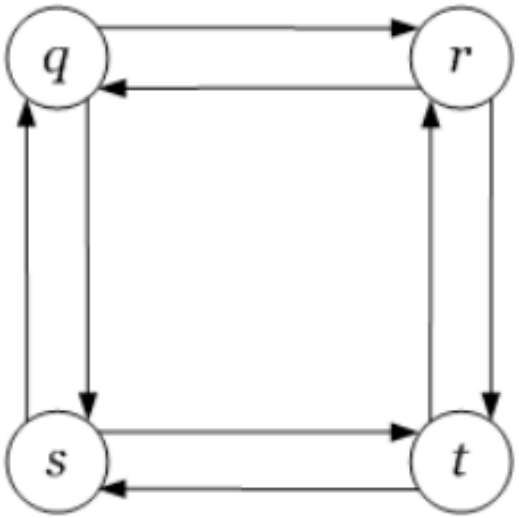
Programação Dinâmica

- Subestrutura ótima:
  - Uma solução ótima para um problema pode ser obtida combinando soluções ótimas de seus subproblemas.
- Exemplo: encontrar o menor caminho entre duas cidades.



Programação Dinâmica

- Devemos tomar cuidado ao analisar a subestrutura ótima
  - Alguns problemas parecem possuir tal estrutura, mas não possuem.
- Exemplo: Caminho mais longo.



Programação Dinâmica

- Subproblemas sobrepostos
  - Segunda característica importante para utilizar programação dinâmica;
  - Acontece quando um algoritmo recursivo revisita o mesmo problema repetidamente;
  - Os resultados de subproblemas são armazenados e reutilizados, de forma que não precisam ser recalculados.

Programação Dinâmica

- Exemplo: problema da soma do subconjunto (subset sum).
  - Temos um conjunto C de inteiros;
  - Desejamos saber se, nesse conjunto, existe um subconjunto cuja soma dos valores é *sum*.

- Se temos: C = {3, 34, 4, 12, 5, 2} e soma = 9:
  - A resposta para o problema é que sim, existe um subconjunto cuja soma dos elementos é 9 ({4, 5}).

Programação Dinâmica

- O problema tem subestrutura ótima:
  - Uma solução para o problema depende de soluções para casos menores do problema;
  - Suponha que um elemento, *x*, do subconjunto solução, seja conhecido;
  - Precisamos agora somente resolver o subset sum para um problema com *n – 1* elementos, cuja soma seja *sum – x*.

Programação Dinâmica

- O problema apresenta subproblemas sobrepostos:
  - Suponha que temos dois subproblemas, um com subconjunto S1 e soma T1, e outro com subconjunto S2 e soma T2;
  - S1 != S2 e T1 == T2;
  - Esses dois subproblemas podem ser sobrepostos!
  - Se resolvermos um deles, não é necessário resolver o outro.

Programação Dinâmica

- Sempre podemos dividir o problema em dois subproblemas:
  - Inclua o último elemento *xj* do conjunto *C* na solução, e resolva o problema com *C – {xj}* e soma total *sum – xj*.
  - Ignore o último elemento *xj* do conjunto *C*, e resolve o problema com *C – {xj}* e soma total *sum*.

Programação Dinâmica

- Usamos uma tabela para armazenar as soluções dos subproblemas:
  - T[i][j] = *true* se existe uma solução para o subproblema com conjunto C = {x1, …, xi} com soma j, e *false* caso contrário.
  - A solução do problema está na tabela, na posição T[n][sum]!

Programação Dinâmica

- Para resolver o problema, precisamos computar a tabela:
- Exemplo: sum = 5, C = {4, 2, 1, 3}

	0	1	2	3	4	5
0	1	0	0	0	0	0
1	1	0	0	0	1	0
2	1	0	1	0	1	0
3	1	1	1	1	1	1
4	1	1	1	1	1	1

Algoritmos Gulosos

- Aplicados geralmente a problemas de otimização.
- Monta uma solução para um problema de forma incremental, fazendo escolhas ótimas em um dado momento.
- Independente do que possa acontecer mais tarde, nunca reconsidera a decisão.
- Não faz uso de procedimentos sofisticados para desfazer decisões tomadas previamente.

Algoritmos Gulosos

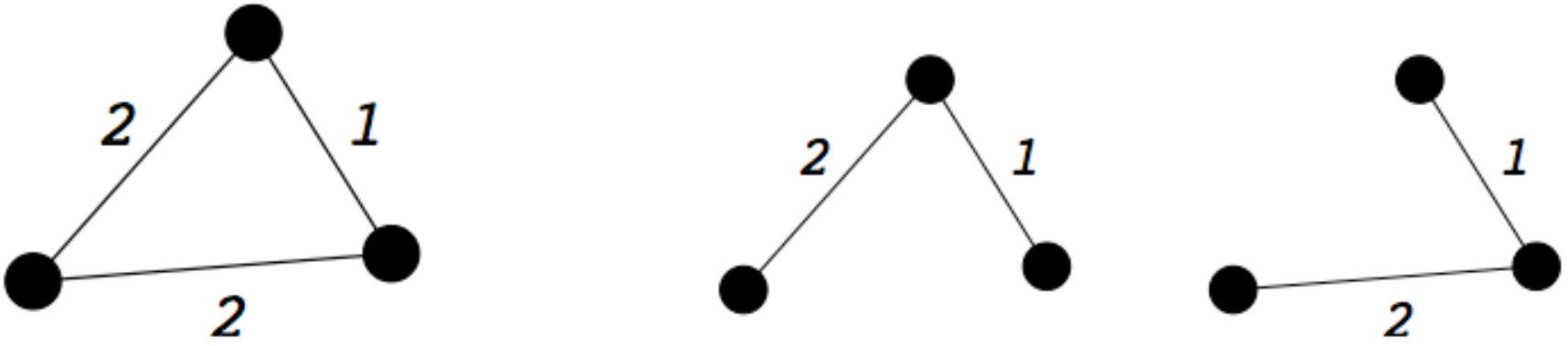
- Características dos algoritmos gulosos:
  - Para construir a solução ótima existe um conjunto ou lista de candidatos (escolhas possíveis).
  - Uma função de seleção indica a qualquer momento quais dos candidatos restantes é o mais promissor.
  - Uma função objetivo fornece o valor da solução encontrada, como o comprimento do caminho construído.

Algoritmos Gulosos

- Características dos algoritmos gulosos:
  - A função de seleção é geralmente relacionada com a função objetivo.
  - Um candidato escolhido e adicionado à solução passa a fazer parte dessa solução permanentemente.
  - Um candidato excluído do conjunto solução, não é mais reconsiderado.

Algoritmos Gulosos

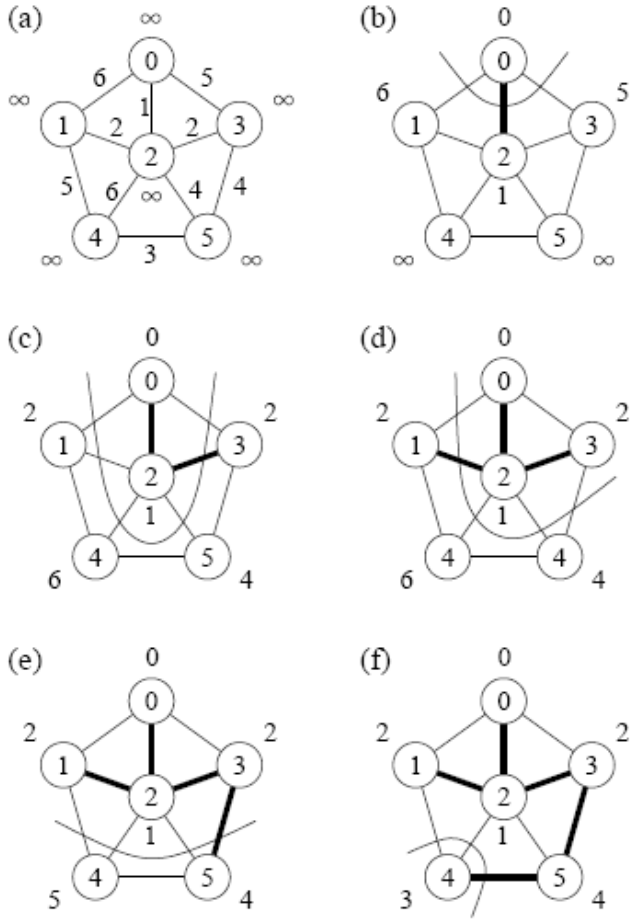
- Exemplo: Encontrar a árvore geradora mínima de um grafo.



Algoritmos Gulosos

- Algoritmo de Prim.
- Classifica vértices em 3 categorias:
  - Vértices da árvore;
  - Vértices da borda;
  - Vértices não vistos.
- O algoritmo seleciona vértices da borda de forma gulosa, até encontrar a solução do problema.

Algoritmos Gulosos



Algoritmos Gulosos

- Algoritmos gulosos fazem a melhor escolha a cada momento.

- Sempre encontraremos a solução ótima?
  - Não. Exemplo: problema da mochila

- Em geral adequados quando:
  - Problema possui subestrutura ótima;
  - Não há possibilidade de "arrependimento" depois de tomar uma decisão.

Algoritmos Aproximados

- Problemas que somente possuem algoritmos exponenciais para resolvê-los são considerados “difíceis”.

- Problemas considerados intratáveis ou difíceis são muito comuns, tais como:
  - Problema do caixeiro viajante cuja complexidade de tempo é O(n!).

- Diante de um problema difícil é comum remover a exigência de que o algoritmo tenha sempre que obter a solução ótima.

- Neste caso procuramos por algoritmos eficientes que não garantem obter a solução ótima, mas uma que seja a mais próxima possível da solução ótima.

Algoritmos Aproximados

- Tipos de algoritmos aproximados:
  - **Heurística**: é um algoritmo que pode produzir um bom resultado, ou até mesmo obter a solução ótima, mas pode também não produzir solução alguma ou uma solução que está distante da solução ótima.
  - **Algoritmo aproximado**: é um algoritmo que gera soluções aproximadas dentro de um limite para a razão entre a solução ótima e a produzida pelo algoritmo aproximado (comportamento monitorado sob o ponto de vista da qualidade dos resultados).

Exemplos de Questões

O problema do escalonamento de intervalos pode ser resolvido com o algoritmo descrito a seguir. O conjunto de intervalos dados inicialmente é R e o conjunto de intervalos escolhidos, A, começa vazio.

```
enquanto R não estiver vazio,
    seja x o intervalo de R com menor tempo de término, e que não tenha interseção com algum intervalo em A
    retire x de R e adicione ao conjunto A
retorne A
```

O algoritmo encontra a solução ótima?

Exemplos de Questões

- O algoritmo mostrado é um algoritmo guloso.
- Como verificar que ele está correto e encontra a solução ótima?
  - O problema possui subestrutura ótima;
  - Basta percebermos que, no começo de cada iteração, a solução parcial faz parte de uma solução ótima.

Exemplos de Questões

Considerando os diferentes paradigmas e técnicas de projeto de algoritmos, analise as afirmações abaixo.

I. A técnica de tentativa e erro (backtracking) efetua uma escolha ótima local, na esperança de obter uma solução ótima global.

II. A técnica de divisão e conquista pode ser dividida em três etapas: dividir a instância do problema em duas ou mais instâncias menores; resolver as instâncias menores recursivamente; obter a solução para as instâncias originais (maiores) por meio da combinação dessas soluções.

III. A técnica de programação dinâmica decompõe o processo em um número finito de subtarefas parciais que devem ser exploradas exaustivamente.

IV. O uso de heurísticas (ou algoritmos aproximados) é caracterizado pela ação de um procedimento chamar a si próprio, direta ou indiretamente.

É correto apenas o que se afirma em

A) I.
B) II.
C) I e IV.
D) II e III.
E) III e IV.