Construção e Análise de Algoritmos

discussão 21a: Programação dinâmica $\simeq$ Algoritmos gulosos

(ou, comendo pelas beiradas ...)

Na aula 21, nós apresentamos a técnica da programação dinâmica como uma variante mais sofisticada do método da divisão e conquista.

De fato, nós podemos ver a programação dinâmica como uma adaptação da divisão e conquista aos problemas de otimização.

Quer dizer, em geral, um problema de otimização pode ser quebrado de muitas maneiras diferentes.

E cada maneira de quebrar o problema, em geral, produz uma solução de qualidade diferente.

Então, para encontrar uma solução ótima para o problema, nós podemos quebrá-lo de todas as maneiras possíveis, e escolher a melhor solução produzida por todas as possibilidades de quebra.

Em geral, esse tipo de estratégia produz algoritmos exponenciais.

Mas, nos casos em que o número de subproblemas diferentes gerados pelo processo é relativamente pequeno, nós obtemos um algoritmo elegante e eficiente para o problema.

Legal! foi isso o que nós aprendemos na aula 21.

Mas, a técnica da programação dinâmica também pode ser vista como uma variante mais sofisticada dos algoritmos gulosos.

Para ver isso, basta olhar para as coisas de maneira ligeiramente diferente.

Considere mais uma vez o problema da seleção de atividades com prioridades.

Nós já vimos que se uma atividade a_j é escolhida para fazer parte da solução, e nós eliminamos todas as atividades incompatíveis com ela, então o problema é decomposto em dois subproblemas independentes que podem ser resolvidos separadamente.

Mas, e se a atividade a_j não pertence a nenhuma solução ótima?

Bom, nesse caso, nós não precisamos nos preocupar com ela, e essa atividade pode ser simplesmente removida do problema.

Essa é a ideia nova aqui!

Esse passo é como uma escolha gulosa ao contrário ...

Quer dizer, uma estratégia gulosa vai escolhendo atividades uma a uma para fazer parte da solução, e a cada passo o problema vai ficando um pouquinho menor.

Mas, nós também podemos pensar em uma estratégia que vai escolhendo atividades uma a uma para ficar fora da solução e, dessa maneira, o problema também vai ficando um pouquinho menor a cada passo*.

Essa ideia sozinha ainda não é suficiente para resolver problemas como a seleção de atividades com prioridades de maneira ótima.

Quer dizer, em geral, não há como saber de antemão se uma atividade a_j faz ou não faz parte de alguma solução ótima.

E se a gente se compromete com uma dessas opções no início do processo de solução, a gente pode terminar com uma solução subótima.

Mas então, nós podemos aplicar de novo a ideia da programação dinâmica: *examine as duas possibildades!*

A maneira mais eficiente de implementar essa ideia consiste em fazer como o algoritmo guloso: ir comendo pelas beiradas ...

Quer dizer, nós vamos considerar as atividades em ordem.

Só que, ao invés de ordená-las pelo tempo de encerramento, nós vamos ordená-las pelo tempo de início.

Para cada atividade a_j na lista, existem basicamente duas opções:

1. ou a_j é escolhida para fazer parte da solução, e todas as atividades incompatíveis com ela são removidas do problema
2. ou a_j não fará parte da solução, e ela é simplesmente removida do problema

Em ambos os casos nós obtemos um problema menor do que o original, e isso já nos dá uma estratégia recursiva de solução do problema.

```
Procedimento  Ativ-Pri-PD-2.0 ( A: atividades ordenadas p/ início )
 {
    Se ( A está vazio )   Retorna ( vazio, 0 )

    aj <--  primeira atividade da lista A

   A1  <--  remove aj e incompativeis da lista A
(S1,P1)  <--  Ativ-Pri-PD-2.0 ( A1 )

   A2  <--  remove aj da lista A
(S2,P2)  <--  Ativ-Pri-PD-2.0 ( A2 )
```

*No problema 2 da lista 16, nós vimos um algoritmo guloso que funciona dessa maneira.

```
    Se ( P1 + pj  >  P2 )  Retorna ( S1 + aj, P1 + pj )
    Senão                  Retorna ( S2, P2 )
}
```

A primeira coisa que chama a atenção nesse procedimento, em comparação com a versão `1.0` que vimos na aula 21, é que ele não tem um laço.

Quer dizer, para fazer a ideia da divisão e conquista funcionar, nós precisamos examinar todas as possibilidades de quebra, e isso requer um laço.

Mas aqui, ao fazer as coisas como o algoritmo guloso e examinar as atividades em ordem, a cada passo só existem 2 opções: ou a atividade a_j fará parte da solução ou ela não fará.

Isso por si só já reduz o tempo de execução do procedimento (sem levar em conta as chamadas recursivas).

Mas, ainda existe um ganho adicional.

Quer dizer, nós vamos ver a seguir que essa estratégia também reduz o número de subproblemas diferentes que são produzidos ao longo do processo.

Para ver isso, considere a chamada inicial do procedimento, onde nós ainda temos todas as atividades na lista

$$A \;=\; \{a_1,\; a_2,\; a_3,\; \ldots,\; a_{n-1},\; a_n\}$$

Caso a atividade a_1 seja escolhida para fazer parte da solução, a_1 e todas as atividades que começam antes do término de a_1 são removidas do problema, produzindo um subproblema da forma

$$A_1 \;=\; \{a_j,\; a_{j+1},\; \ldots,\; a_{n-1},\; a_n\}$$

(porque as atividades estão ordenadas por tempo de início)

Caso a atividade a_1 fique fora da solução, o subproblema que é produzido é o seguinte

$$A_2 \;=\; \{a_2,\; a_3,\; \ldots,\; a_{n-1},\; a_n\}$$

A observação importante é que, em ambos os casos, nós temos uma subsequência contígua de atividades.

E não é difícil ver que essa propriedade é mantida durante toda a execução do procedimento.

Dessa maneira, cada subproblema que aparece no processo de solução é caracterizado pela primeira atividade a_j na lista (porque a última atividade sempre é a_n).

E isso significa que existem no máximo n subproblemas diferentes (que correspondem aos diferentes valores que o índice j pode assumir).

Para concluir a análise de complexidade do algoritmo, basta estimar o tempo de execução de cada chamada ao procedimento `Ativ-Pri-PD-2.0`.

E aqui surge a oportunidade para uma pequena esperteza.

Quer dizer, com a exceção de

```
A1  <--  remove aj e incompativeis da lista A
```

todas as instruções do procedimento executam em tempo $O(1)$.

A ideia então é pré-calcular o resultado dessa operação para toda atividade a_j.

Na prática, isso corresponde ao índice k da primeira atividade que se inicia após o término de a_j.

Caso a lista de atividades já esteja ordenada (pelo tempo de início), esse índice pode ser encontrado em tempo $O(\log n)$, através de uma busca binária.

Portanto, antes de iniciar as chamadas recursivas, nós gastamos tempo $O(n \log n)$ para ordenar as atividades por tempo de início.

E gastamos mais $O(n \log n)$ para pré-calcular os resultados da operação de remoção da atividade a_j e incompatíveis da sublista que começa em a_j.

Fazendo isso, cada chamada ao procedimento `Ativ-Pri-PD-2.0` executa em tempo $O(1)$.

E a complexidade total do algoritmo é dada por

$$O(n \log n) \;+\; O(n \log n) \;+\; O(n) \times O(1) \quad = \quad O(n \log n)$$

Quer dizer, nós conseguimos reduzir a complexidade do algoritmo de $O(n^3)$ para $O(n \log n)$, o que é um ganho realmente significativo.