Construção e Análise de Algoritmos

aula 21: Programação dinâmica

1 Introdução

Depois de algum tempo ficou claro que a ideia da coordenação de alocar o maior número possível de atividades no laboratório não era uma boa ideia.

Quer dizer, apesar das boas intenções do coordenador, de tentar satisfazer o maior número de pessoas, a política de alocação deixou quase todos insatisfeitos.

Uns diziam: "Já faz 5 anos que eu não consigo reservar o laboratório para a minha disciplina."

Outros diziam: "O laboratório passa o dia inteiro ocupado com atividades opcionais, e eu não consigo realizar os trabalhos práticos da minha disciplina obrigatória."

Outros ainda diziam: "Não é porque uma atividade demanda muito tempo do laboratório que ela deva ser deixada de lado."

E por aí vai ...

O coordenador pensava consigo mesmo: "Ai, ai, é difícil mesmo querer agradar as pessoas ...".

Mas então ele teve uma ideia que parecia boa!

E correu para contar aos colegas.

A ideia era atribuir prioridades diferentes às diversas atividades (que podiam mudar com o tempo ou dependendo das circunstâncias).

As atividades seriam alocadas, então, levando em conta essas prioridades.

Todos gostaram da ideia, pelo menos por enquanto ...

O problema de otimização dessa historinha pode ser formalizado da seguinte maneira.

Nós temos um conjunto de atividades $A = \{a_1, a_2, \ldots, a_n\}$, cada uma com um horário de início e um horário de término.

E cada atividade a_i está associada também a uma prioridade p_i, onde um valor alto de prioridade indica que a atividade é importante.

O objetivo do problema consiste em selecionar um subconjunto de atividades S de modo que a soma

$$\mathsf{Prior}(S) \quad = \quad \sum_{a_i \in S} p_i$$

seja a maior possível.

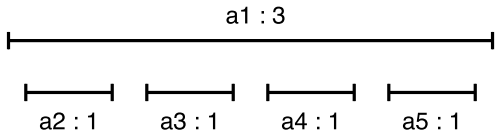
Examinando as declarações dos professores, nós podemos supor que eles imaginam que se as suas atividades tiverem uma prioridade bem alta, então eles iriam conseguir utilizar o laboratório.

A ideia implícita aqui é que as atividades deveria ser selecionadas em ordem decrescente de prioridade — o que é uma estratégia gulosa.

Mas, será que isso nos leva a uma solução ótima?

Infelizmente não ...

Veja só.



Nesse exemplo, a estratégia gulosa seleciona a atividade a_1, que possui prioridade 3, de depois não consegue selecionar mais nenhuma atividade.

Mas, é fácil ver que a solução ótima consiste em selecionar as atividades a_2, a_3, a_4, a_5, cuja soma de prioridades é igual a 4.

Outra ideia consiste em adaptar a estratégia que deu certo na aula 15.

Quer dizer, naquele dia nós ordenamos as atividades de acordo com o horário de término (do menor para o maior), e daí nós selecionamos as atividades nessa ordem, eliminando a cada passo as atividades incompatíveis.

Uma adaptação possível dessa estratégia para o problema com prioridades é a seguinte:

- Para cada atividade a_i nós calculamos a soma P_i das prioridades das atividades que começam após o término de a_i
- A seguir, nós ordenamos as atividades de acordo com esse valor (do maior para o menor)
- Finalmente, nós selecionamos as atividades nessa ordem, eliminando a cada passo as atividades incompatíveis

Mas, infelizmente, essa estratégia também não leva a uma solução ótima (em todos os casos).

(Porque? você consegue encontrar um contra-exemplo?)

Na realidade, ninguém conhece uma estratégia gulosa ótima para esse problema.

E isso significa que agora nós precisamos de novas ideias.

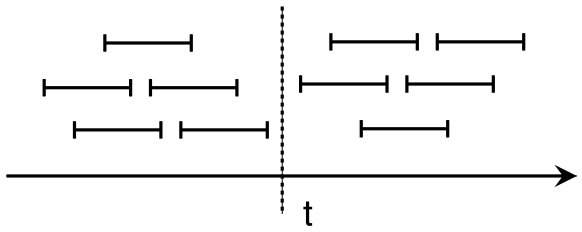
2 Divisão e conquista

Bom, se a estratégia gulosa não funcionou, nós podemos tentar a divisão e conquista — que é a outra ideia que nós já aprendemos.

Para aplicar a técnica da divisão e conquista, nós precisamos encontrar uma maneira de quebrar o problema.

Existe um caso em que isso é uma tarefa bem simples.

Veja só.



Nesse exemplo, existe um instante de tempo t em que não existe nenhuma atividade acontecendo.

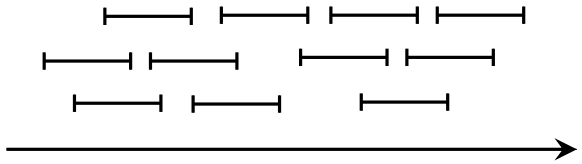
Isso significa que não existe conflito entre o subconjunto de atividades A_1 que termina antes de t, e o subconjunto de atividades A_2 que começa depois de t.

E isso significa que a solução do problema A pode ser calculada da seguinte maneira

$$S(A) \quad = \quad S(A_1) \ \cup \ S(A_2)$$

Mas, nem sempre a vida é assim tão fácil ...

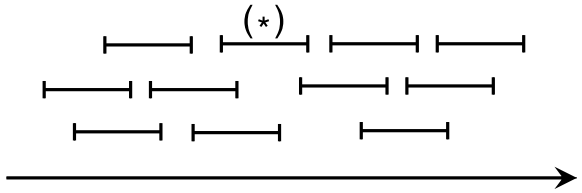
Quer dizer, na maioria dos casos (interessantes) esse instante de tempo t simplesmente não existe.



E o nosso desafio consiste em descobrir uma maneira de quebrar o problema nesse tipo de situação.

Vejamos.

Escolha uma atividade a_j qualquer, digamos, lá pelo meio do problema

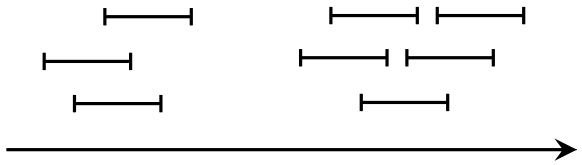


Nós não temos, é claro, a menor ideia se a atividade a_j está na solução ótima ou não.

Mas, suponha que sim.

Se isso é o caso, então nenhuma atividade que entra em conflito com ela pode estar na solução, e elas podem ser eliminadas do problema.

Mas, veja o que acontece quando nós fazemos isso



Quer dizer, ao selecionar a atividade a_j para a solução, o problema foi decomposto em dois subproblemas independentes A_1 e A_2 que podem ser resolvidos separadamente.

E, uma vez que isso tenha sido feito, a solução do problema A pode ser calculada como

$$S(A) \quad = \quad S(A_1) \ \cup \ \{a_j\} \ \cup \ S(A_2)$$

O problema é que nós não temos a menor ideia se a atividade a_j faz parte da solução ótima ou não.

E, como vimos há pouco, ninguém conhece uma regra simples para descobrir atividades que fazem parte da solução.

Mas, essa não é uma dificuldade tão grande assim.

3 Programação dinâmica

Quer dizer, nada nos garante que a solução ótima é obtida escolhendo a atividade a_j.

Mas nada nos impede de fazer outras tentativas.

Isto é, selecionando uma outra atividade qualquer a_i, e eliminando as atividades incompatíveis com ela, o problema se decompõe em outros dois subproblemas independentes A'_1 e A'_2 que podem ser resolvidos em separado para nos dar a solução

$$S_i(A) \quad = \quad S(A'_1) \ \cup \ \{a_i\} \ \cup \ S(A'_2)$$

Agora, denotando a solução anterior por $S_j(A)$, nós podemos comparar a prioridade total das duas e ficar com a melhor delas.

É claro que isso ainda não nos dá a garantia de otimalidade, mas agora nós já sabemos o que fazer.

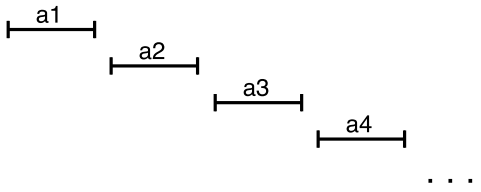
Abaixo nós temos o pseudo-código de um algoritmo que aplica essa ideia a todas as atividades e, dessa maneira, obtém a solução ótima do problema.

```
Procedimento  Ativ-Pri-PD ( A: conjunto de atividades com prioridades )
 {
     Se ( A está vazio )   Retorna ( vazio, 0 )

     Smax  <--  vazio;    Pmax  <--  0

     Para cada atividade  aj  em  A
      {
           A1,A2  <--  Quebra (A,aj)

         (S1,P1)  <--  Ativ-Pri-PD ( A1 )
         (S2,P2)  <--  Ativ-Pri-PD ( A2 )

         Se ( Pmax  <  P1 + P2 + pj )

              Smax  <--  S1 + aj + S2
              Pmax  <--  P1 + pj + P2
       }
      }
     Retorna (Smax,Pmax)
 }
```

Análise de complexidade

Mas, não é difícil ver que esse procedimento executa em tempo exponencial.

Considere, por exemplo, um problema sem conflitos



Nesse caso, apenas uma atividade é eliminada a cada quebra do problema, e nós podemos escrever a seguinte equação de recorrência para o número de chamadas recursivas realizadas pelo procedimento

$$
\begin{aligned}
C(n) &= 2 \cdot \sum_{i=0}^{n-1} C(i) \\
C(0) &= 1
\end{aligned}
$$

Abaixo, nós examinamos os valores iniciais dessa função

- $C(1) = 2 \cdot C(0) = 2$
- $C(2) = 2 \cdot \Big[C(0) + C(1)\Big] = 6$
- $C(3) = 2 \cdot \Big[C(0) + C(1) + C(2)\Big] = 18$
- $C(4) = 2 \cdot \Big[C(0) + C(1) + C(2) + C(3)\Big] = 58$
- ...

Esse negócio cresce mais rápido que a função de Fibonacci!

Quer dizer, a função $C(n)$ tem crescimento exponencial.

E isso significa que o nosso algoritmo não é realmente um algoritmo — isto é, algo que nós podemos executar para resolver um problema.

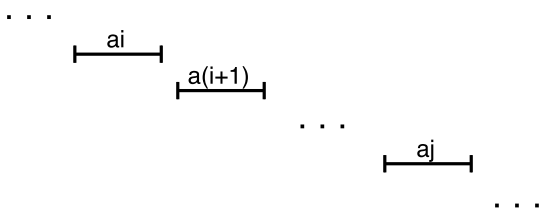
4 Memoização

Mas, vamos manter a calma.

Sim, o procedimento realmente realiza uma quantidade exponencial de chamadas recursivas.

Mas, não existe uma quantidade exponencial de subproblemas diferentes no exemplo acima.

Quer dizer, de acordo com a nossa estratégia de quebra, todo subproblema que aparece tem o seguinte formato



isto é, uma sequência contígua de atividades.

E não existe uma quantidade tão grande assim de sequências desse tipo — já já nós vamos estimar o seu número.

Isso significa que a vasta maioria das chamadas recursivas são chamadas repetidas.

Ora, uma chamada recursiva repetida não precisa ser feita de novo — se nós tivermos armazenado o seu resultado em algum lugar.

Ou seja, nós podemos transformar o procedimento `Ativ-Pri-PD` em um algoritmo eficiente utilizando uma tabela de resultados

- `T[i,j]` = resultado da chamada para o subproblema $\{a_i, \ldots, a_j\}$

No início, a tabela se encontra completamente vazia.

Quando o problema $\{a_i, \ldots, a_j\}$ é resolvido pela primeira vez, a sua solução é armazenada na tabela.

No momento em que uma chamada recursiva sobre o subproblema $\{a_i, \ldots, a_j\}$ está para ser realizada, a tabela é consultada.

Se a solução já está lá, então a chamada não é feita.

Abaixo nós temos uma nova versão do procedimento utilizando essa ideia.

```
Procedimento  Ativ-Pri-PD2.0 ( A[i,j]: conjunto de atividades com prioridades )
 {
     Se ( A[i,j] está vazio ) {   T[i,j]  <--  (vazio, 0);    Retorna  }

     Smax  <--  vazio;    Pmax  <--  0

     Para cada atividade  ak  em  A[i,j]
      {
         A1[i,g], A2[h,j]  <--  Quebra (A,ak)

         Se ( T[i,g] = Nulo )  Ativ-Pri-PD2.0 ( A1[i,g] )
         (S1,P1)  <--  T[i,g]

         Se ( T[h,j] = Nulo )  Ativ-Pri-PD2.0 ( A2[h,j] )
         (S2,P2)  <--  T[h,j]

         Se ( Pmax  <  P1 + pk + P2 )
          {
              Smax  <--  S1 + ak + S2
              Pmax  <--  P1 + pk + P2
          }
      }
     T[i,j]  <--  (Smax,Pmax)
 }
```

Note que o procedimento não tem mais valor de retorno.

Se, no momento de uma chamada recursiva a posição correspondent na tabela ainda está vazia, a chamada é feita para preencher essa posição, e a seguir a tabela é consultada para obter o resultado.

Análise de complexidade

Nós vamos analisar primeiro o tempo de execução do algoritmo no problema sem conflitos.

Como já observamos acima, nesse caso cada subproblema corresponde a uma sequência da forma $\{a_i, \ldots, a_j\}$.

Ora, cada sequência desse tipo é definida pelo seu índice incial i e pelo seu índice final j.

E tanto i como j devem ser números entre 1 e n.

Portanto, existem no máximo $n \times n = n^2$ combinações de valores para i e j (na realidade, um pouco menos do que isso).

E isso nos permite concluir que existem

$$O(n^2) \quad \text{subproblemas diferentes}$$

Finalmente, cada subproblema corresponde a uma chamada recursiva.

E cada chamada recursiva realiza um laço que percorre todas as atividades do subproblema, dando no máximo $O(n)$ voltas.

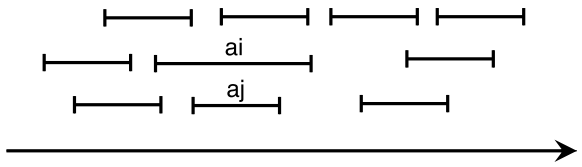
No caso sem conflitos a quebra do subproblema é muito simples, e assim todas as instruções no laço executam em tempo $O(1)$.

Portanto, no caso sem conflitos o algoritmo executa em tempo

$$\underbrace{O(n^2)}_{\substack{\text{número de} \\ \text{chamadas recursivas}}} \times \underbrace{O(n)}_{\substack{\text{tempo de cada} \\ \text{chamada recursiva}}} = O(n^3)$$

Mas, o caso geral não é muito diferente.

Quer dizer, no caso geral não é possível ordenar as atividades em uma sequência natural.

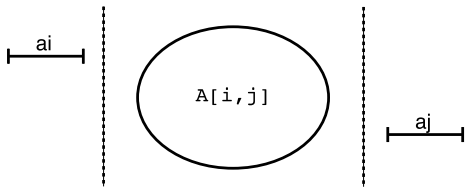


(No problema acima, a_i vem antes ou depois de a_j?)

E portanto os subproblemas não vão ter a forma de sequências $\{a_i, \ldots, a_j\}$.

Mas, ainda faz sentido falar no subproblema `A(i,j)`, com a seguinte interpretação

`A(i,j) =` atividades que começam depois do término da atividade a_i,
e terminam antes do começo da atividade a_j



Esse é exatamente o tipo de subproblema que é gerado pela nossa estratégia de quebra.

E, como antes, existem no máximo $O(n^2)$ combinações possíveis de valores para i, j.

Nesse caso, a operação de quebra é ligeiramente mais complicada, mas nós podemos assumir que os subproblemas são pré-computados antes de iniciar a recursão, de modo que todas as intruções do laço ainda executam em tempo $O(1)$.

Dessa maneira, também no caso geral, o algoritmo executa em tempo $O(n^3)$.

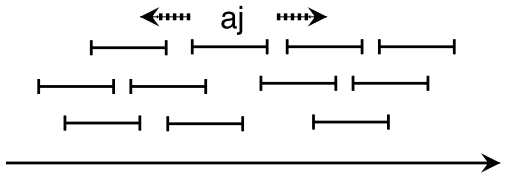
5 Uma decomposição mais esperta

É possível obter um algoritmo um pouco mais eficiente, olhando para o problema de maneira ligeiramente diferente.

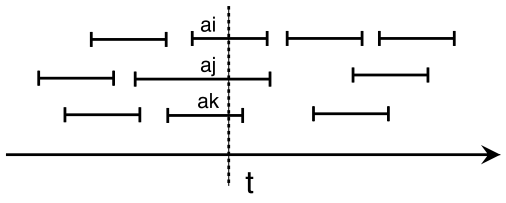
Vejamos.

A estratégia de quebra que nós utilizamos consiste em selecionar uma atividade a_j para a solução, e resolver os subproblemas resultantes recursivamente.

E, para fazer a coisa funcionar, nós examinamos todas as possíveis escolhas de a_j.



Mas, no início da aula, nós não estávamos vendo as coisas na horizontal, mas sim na vértical.



Olhando as coisas desse ângulo, nós podemos fazer as seguintes observações:

- ou a atividade a_i faz parte da solução
- ou a atividade a_j faz parte da solução
- ou a atividade a_k faz parte da solução
- ou nenhuma delas faz parte da solução

Em todos os casos, o problema pode ser quebrado em subproblemas independentes.

E esses são todos os casos possíveis!

(Você consegue ver isso?)

Ou seja, nós acabamos de encontrar uma nova maneira de organizar as quebras do problema.

E essa estratégia tende a ser mais eficiente que a anterior, pois o número de atividades que cruzam uma certa linha vertical tende a ser bem menor que o número total de atividades.

Abaixo nós temos uma implementação simples dessa ideia*

```
Procedimento  Ativ-Pri-PD3.0 ( A[i,j]: conjunto de atividades com prioridades )
 {
     Se ( A[i,j] está vazio ) {   T[i,j]  <--  (vazio, 0);     Retorna  }

     ak  <--  atividade qualquer de  A[i,j]
      B  <--  ak  +  atividades que conflitam com ak

     Smax  <--  vazio;    Pmax  <--  0

     Para cada atividade  b  em  B
      {
         (A1,A2)  <--  Quebra (A,b)

         Se ( T[A1] = Nulo )  Ativ-Pri-PD3.0 ( A1 )
         (S1,P1)  <--  T[A1]

         Se ( T[A2] = Nulo )  Ativ-Pri-PD3.0 ( A2 )
         (S2,P2)  <--  T[A2]

         Se ( Pmax  <  P1 + pb + P2 )
          {
              Smax  <--  S1 + b + S2
              Pmax  <--  P1 + pb + P2
          }
      }
     T[A]  <--  (Smax,Pmax)
 }
```

Assumindo que o tamanho do conjunto B é no máximo L, esse algoritmo executa em tempo

$$L \times O(n^2) \quad = \quad O(Ln^2)$$

*Nessa implementação, ao invés de considerar as atividades que cruzam um certo instante t do tempo, nós consideramos uma atividade a_k qualquer e todas as atividades que conflitam com ela. A observação chave, nesse caso, é que uma solução ótima deve conter alguma atividade desse grupo (se as prioridades são todas positivas) — porque?