Construção e Análise de Algoritmos

aula 09: O método da divisão e conquista

1 Introdução

Nas últimas 5 aulas nós construímos diversos algoritmos para o problema da ordenação.

E agora já é possível perceber um padrão.

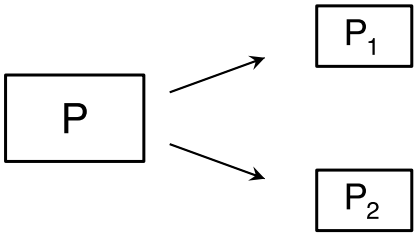
Em todos os casos, nós encontramos uma maneira de quebrar o problema em uma coleção de subproblemas (em geral sublistas com apenas uma parte dos elementos) e, ao resolver os subproblemas nós nos aproximamos da solução do problema original.

Mesmo o algoritmo radix pode ser visto dessa maneira: ao concentrar a nossa atenção sobre apenas um dígito e ignorar todos os outros, nós obtemos um subproblema que pode ser resolvido de maneira muito eficiente.

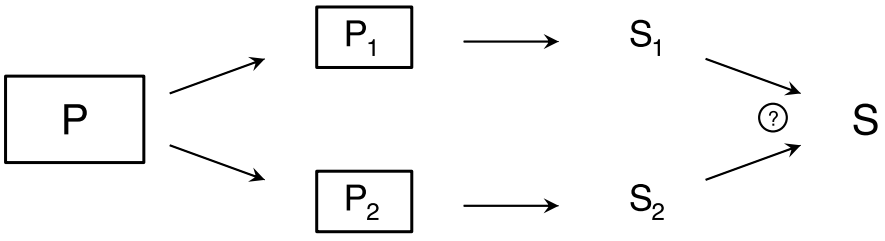
De fato, essa é uma ideia muito geral, que pode ser aplicada a qualquer problema.

Suponha que P é um problema qualquer

E suponha que nós temos uma maneira qualquer de quebrar o problema P em problemas menores, por exemplo



Então, resolvendo os problemas menores, nós podemos tentar combinar as suas soluções para obter a solução do problema original

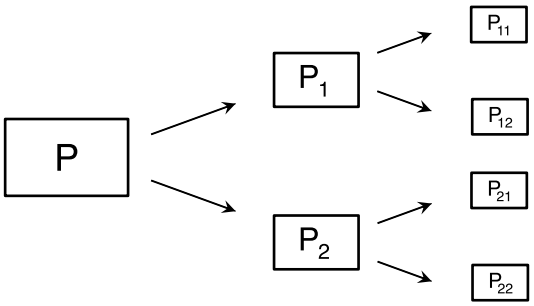


Como dissemos, essa é uma ideia muito geral.

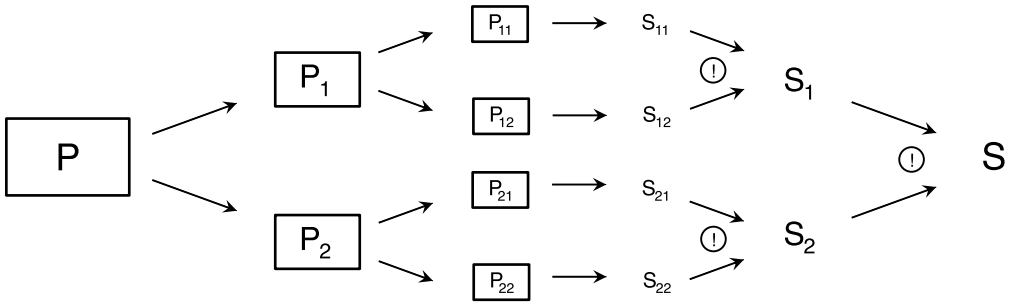
Mas ela ainda não é o método da divisão e conquista.

A coisa fica mais interessante quando P_1 e P_2 são instâncias menores do mesmo tipo que o problema original P.

Nesse caso, a mesma estratégia que foi utilizada para quebrar o problema P também pode ser utilizada para quebrar os subproblemas P_1 e P_2

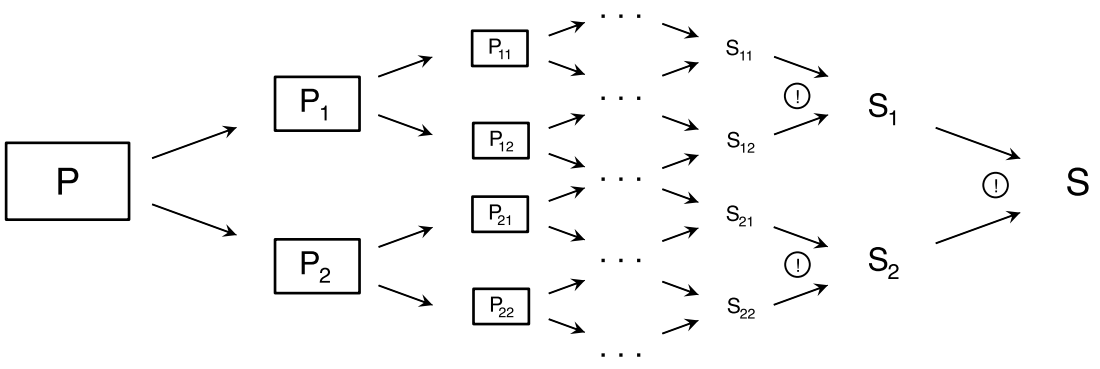


E a mesma estratégia que foi utilizada para combinar as soluções de P_1 e P_2 também pode ser utilizada para combinar as soluções de P_{11} e P_{12} e de P_{21} e P_{22}.

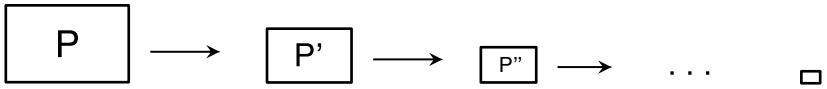


De fato, como P_1 e P_2 foram quebrados da mesma maneira que P, nós podemos concluir que P_{12}, P_{21} e P_{22} também são problemas do mesmo tipo que P.

E isso significa que eles também podem ser quebrados



Finalmente, a última observação é que, frequentemente, quando quebramos um problema sucessivamente até ele ficar bem pequenininho



no final das contas nós obtemos algo que já não é mais problema algum (por exemplo, ordenar uma lista com apenas um elemento, ou uma outra tarefa trivial qualquer).

E é aqui onde está a pequena mágica do método da divisão e conquista:

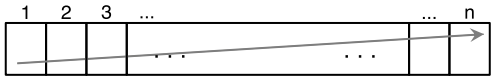
- *ao descobrir como quebrar um problema* P *em problemas menores (do mesmo tipo) e como combinar as soluções desses últimos para obter a solução de* P*, o problema* P *já foi essencialmente resolvido!*

(Daí o nome divisão e conquista ...)

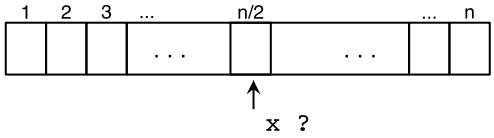
Exemplo prosaico: Suponha que você quer saber quantos feijões um saco de 5kg contém (aproximadamente). Bom, seguindo as ideias acima, você pode começar dividindo o conteúdo do saco em duas partes iguais. Se você conseguir determinar a quantidade de feijões em cada metade, então basta multiplicar esse número por 2. Mas, agora, você tem o mesmo problema que antes: determinar a quantidade de feijões em meio saco de 5kg. Logo, o seu problema já está resolvido.

2 Busca binária

Voltando aos problemas computacionais, suponha que você tem uma lista ordenada `V[1..n]` e quer saber se o número `x` está na lista ou não.



Como a lista está ordenada, a maneira natural de abordar o problema consiste em comparar o número `x` ao elemento central da lista: `V[n/2]`.



E daí, existem 3 resultados possíveis

- $\texttt{x} = \texttt{V[n/2]}$
  Nesse caso, o problema acabou.
- $\texttt{x} < \texttt{V[n/2]}$
  Nesse caso, `x` só pode estar no lado esquerdo da lista, e agora nós temos o problema de saber se `x` está na lista $\texttt{V}[\texttt{1}..(\frac{\texttt{n}}{\texttt{2}} - \texttt{1})]$ ou não.
- $\texttt{x} > \texttt{V[n/2]}$
  Nesse caso, `x` só pode estar no lado direito da lista, e agora nós temos o problema de saber se `x` está na lista $\texttt{V}[(\frac{\texttt{n}}{\texttt{2}} + \texttt{1})..\texttt{n}]$ ou não.

Portanto, se a solução não é obtida de imediato, o problema é reduzido a um subproblema do mesmo tipo com a metade do tamanho.

Ao final das quebras sucessivas (se `x` não está na lista), nós chegamos a um problema trivial: determinar se `x` está em uma lista vazia.

Portanto, o problema já foi resolvido.

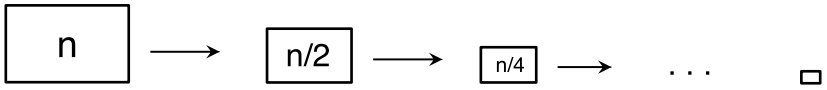
A aplicação do método da divisão e conquista tipicamente produz um algoritmo recursivo que resolve o problema.

```
Procedimento buscaBin-Rec ( x, V[i..j] )
 {
1.      Se ( i > j )   Retorna (NÃO)

2.      k ← (i+j)/2

3.      Se ( x = V[k] )   Retorna (SIM)
4.      Se ( x < V[k] )   resp ← buscaBin-Rec ( x, V[i..k-1] )
5.      Se ( x > V[k] )   resp ← buscaBin-Rec ( x, V[k+1..j] )

6.      Retorna (resp)
 }
```

Nesse algoritmo, as linhas 2-5 correspondem à etapa da divisão, enquanto que a linha 6 corresponde à etapa da conquista (ou, a combinação das soluções dos subproblemas, que nesse caso é trivial).

Como usual, para determinar o tempo de execução de um algoritmo recursivo, nós examinamos a árvore de chamadas recursivas.

Nesse caso, a árvore é apenas uma linha



onde cada chamada recursiva trabalha sobre uma porção da lista que tem a metade do tamanho da porção anterior.

Com base nessa observação, é fácil ver que serão realizadas (no máximo) $\log_2 n$ chamadas recursivas.

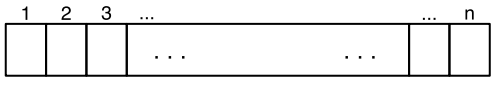
Além disso, cada chamada recursiva basicamente realiza um teste, uma operação aritmética, uma comparação e uma nova chamada recursiva, o que leva tempo $O(1)$.

Portanto, o tempo total de execução do algoritmo é

$$\log_2 n \;\times\; O(1) \quad = \quad O(\log n)$$

3 O problema da seleção

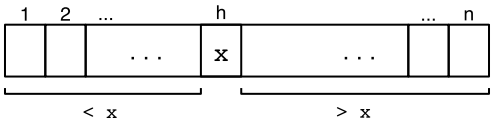
Agora, suponha que nós temos uma lista desordenada



e queremos encontrar o k-ésimo maior elemento dessa lista.

A ideia, é claro, é tentar aplicar o método da divisão e conquista.

E, para fazer isso, nós aplicamos o procedimento de partição



Isto é, nós utilizamos um elemento `x` qualquer como pivô, e movemos os elementos menores que `x` para a esquerda e o elementos maiores que `x` para a direita.

Após o procedimento de partição, o pivô `x` se encontra em alguma posição `h`.

Daí, nós temos 3 possibilidades

- se `h` = `k`, então `x` é o k-ésimo maior elemento da lista
- se `h` > `k`, então o k-ésimo maior elemento da lista se encontra do lado esquerdo — mais especificamente, ele é o k-ésimo maior elemento do lado esquerdo.
- se `h` < `k`, então o k-ésimo maior elemento da lista se encontra do lado direito — mais especificamente, ele é o $(k-h)$-ésimo maior elemento do lado direito.

Pronto, agora o problema já está resolvido.

Isto é,

Div: nós já sabemos como quebrar o problema em um problema menor do mesmo tipo

Conq: e nós já sabemos como obter a solução do problema original a partir da solução do subproblema

Abaixo nós temos o procedimento recursivo que implementa essa solução.

```
Procedimento seleção-Rec ( k, V[i..j] )
  {
      h ← Partição ( V[i..j] )

      Se ( h = k )   Retorna (V[h])
      Se ( h > k )   resp ← seleção-Rec ( k, V[i..h-1] )
      Se ( h < k )   resp ← seleção-Rec ( k, V[h+1..j] )

      Retorna (resp)
  }
```

Qual o tempo de execução desse algoritmo?

Bom, aqui nós temos uma situação semelhante à que vimos no algoritmo Quicksort.

Caso todas as chamadas ao procedimento `Partição` quebrem a lista exatamente no meio, o k-ésimo elemento é encontrado após no máximo $\log_2 n$ chamadas recursivas.

Como o tempo de execução do procedimento `Partição` é proporcional ao tamanho da lista que ele recebe, nesse caso o tempo total do algoritmo é

$$n \;+\; n/2 \;+\; n/4 \;+\; \ldots 2 \;+\; 1 \quad = \quad O(n)$$

Por outro lado, se todas as chamadas ao procedimento `Partição` produzam partições o mais desbalanceado possível, o tempo total de execução é

$$n \;+\; (n-1) \;+\; (n-2) \;+\; \ldots 2 \;+\; 1 \quad = \quad O(n^2)$$

Finalmente, se modificamos o procedimento `Partição` para que ele produza uma partição balanceada onde cada lado possui ao menos 1/3 dos elmentos da lista (veja a aula 03), então o algoritmo realiza no máximo $\log_{3/2} n$ chamadas recursivas, e seu tempo médio de execução é $O(n)$.

Exercícios