Projeto de algoritmos

1 Introdução

Neste capítulo iremos ver algumas técnicas para a construção de algoritmos. Inicialmente veremos a estratégia de força bruta, que nada mais é que fazer programas baseados unicamente no enenciado, sem maiores refinamentos. Veremos que esta estratégia é simples de ser implementada, mas leva a soluções em geral ineficientes. Depois iremos analisar duas outras estratégias de projeto, ambas baseadas na premissa de que é mais eficiente resolver vários problemas pequenos do que um grande: divisão e conquista, e programação dinâmica.

2 Força bruta

É a mais simples das estratégias de projeto, e consiste em encontrar uma solução direta para resolver um problema, geralmente baseada diretamente no enunciado do problema e nas definições dos conceitos envolvidos.

Neste caso, usa-se o poder computacional dos processadores para fazer o trabalho ao invés de tentar utilizar um solução mais inteligente. Esta característica torna esta estratégia uma das mais fáceis de serem aplicadas, embora os resultados não sejam, na maioria das vezes, brilhantes e/ou eficientes.

Apesar destas características, muitas vezes vale a pena utilizar esta estratégia. De fato, para alguns problemas importantes (ordenação, multiplicação de matrizes, casamento de strings, etc.) a técnica de força bruta fornece algoritmos razoáveis, de grande valor prático, e sem limitações quanto ao tamanho da instância.

Adicionalmente, se apenas algumas poucas instâncias de um problema necessitarem ser resolvidas, ou seja, se a velocidade é aceitável, não é necessário passar a abordagens mais sofisticadas. Ainda, pode ser útil para resolver problemas com instâncias pequenas.

Uma última consideração a ser feita sobre esta estratégia é a sua grande impotância do ponto de vista teórico e educacional. Pode ser ainda utilizada para validar estratégias mais sofisticadas e menos transparentes.

No campo dos algoritmos, alguns problemas são clássicos: ordenação, busca, *string matching*, problema do caixeiro viajante, problema da mochila, entre outros. Vamos agora ver como resolver estes problemas segundo a estratégia da força bruta.

Exemplo 1. Um exemplo clássico de algoritmo que se utiliza da estratégia de força bruta é o de ordenação por seleção direta, cujo princípio de funcionamento é o seguinte:

- A cada etapa realizada, é identificado o menor elemento dentro do segmento que contém os elementos ainda não selecionados.
- É realizada uma troca do elemento identificado na etapa anterior, com o primeiro elemento do segmento.
- O tamanho do segmento é atualizado, ou seja, subtrai-se um do tamanho do segmento (menos um elemento).
- Esse processo é interrompido no momento em que o segmento ficar com apenas um elemento.

O código de uma função que executa estes procedimentos é mostrado abaxo:

```
void selecaoDireta(int vetor[], int tamanho)
{
  int menor;
  int i,j,aux;

  for (i=0;i<(tamanho-1);i++)
  {
    menor = i;
    for (j=i+1;j<tamanho;j++)
      if (vetor[j] < vetor[menor])
        menor = j;
    aux = vetor[i];
    vetor[i] = vetor[menor];
    vetor[menor] = aux;
  }
}
```

Exemplo 2. Outro exemplo de aplicação da estratégia de força bruta é o algoritmo de busca sequencial. Basicamente, ele compara os elementos sucessivos de uma dada lista com uma chave de busca até que ocorra um dos casos abaixo:

- é encontrado um elemento igual à chave (busca bem sucedida);
- o algoritmo alcança o final da lista sem ter encontrado um elemento igual à chave (busca mal sucedida).

O código de uma função que executa estes procedimentos é mostrado abaxo:

```
int sequencial(int vetor[],int tamanho, int x)
{
  bool achou = false; // var aux p/ busca
  int i=0; // contador

  while (achou==false && i<tamanho)
    achou = vetor[i++]==x;

  if (achou)
    return (i-1);
  else
    return -1;
}
```

Exemplo 3. O problema de *string matching* pode ser enunciado como segue: "dada uma string de n caracteres chamada de `texto`, e uma string de m caracteres ($m \leq n$), chamada de `padrão`, encontrar uma substring do `texto` que coincida com o `padrão`."

Solução. Usando o método da força bruta, poderíamos resolver o problema da seguinte maneira:

1. alinha o padrão com o início do texto
2. movendo da esquerda para a direita, comparar cada caractere do padrão com o respectivo caracter no texto até que:
   - todos os caracteres sejam coincidentes (busca bem sucedida) ou
   - uma combinação errônea seja detectada
3. enquanto o padrão não for encontrado e o texto não for percorrido integralmente, realinhar o padrão uma posição à direita e repetir o passo 2.

Uma instância deste algoritmo em funcionamento é mostrada abaixo:

```
SEUS AMIGOS DE VERDADE AMAM VOCÊ DE QUALQUER JEITO
AMIGO
 AMIGO
  AMIGO
   AMIGO
    AMIGO
     AMIGO
```

Vou deixar pra você escrever um código que implemente esta solução.

2.1 Busca exaustiva

Muitos problemas importantes necessitam encontrar um elemento com propriedades especiais em um domínio que cresce exponencialmente (ou mais rapidamente) com o tamanho da instância.

Muitos destes problemas são de otimização, ou seja, consistem em encontrar valores de parâmetros que maximizem ou minimizem certas características desejáveis.

Neste caso, a busca exaustiva é simplesmente uma estratégia de força bruta para problemas combinatoriais, e pode ser descrita da seguinte maneira:

- listar todas as soluções potenciais para o problema de uma maneira sistemática de forma que:
  - todas as soluções possíveis são eventualmente listadas;
  - nenhuma solução é repetida
- avaliar as soluções uma a uma, mantendo apenas a melhor solução encontrada até o momento.
- quando a busca terminar, anunciar o vencedor.

Alguns problemas clássicos nesta área são o do caixeiro viajante e o da mochila, os quais iremos ver a seguir.

Exemplo 4. O problema do caixeiro viajante pode ser descrito da seguinte maneira: dadas n cidades com distâncias conhecidas entre cada par, encontrar o trajeto mais curto que passe por todas as cidades exatamente uma vez antes de retornar à cidade de origem. Este percurso é conhecido como percurso Hamiltoniano em um grafo conectado e ponderado (mais disso adiante).

Uma instância deste problema é vista na Figura 1:

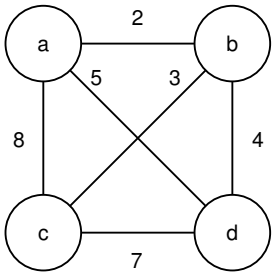


Figura 1: O problema do caixeiro viajante.

Solução. A solução deste problema utilizando a estratégia de força bruta consiste em listar todos os caminhos possíveis e então verificar qual a solução de menor custo. Para a instância apresentada, esta solução é mostrada na Tabela 1

Tabela 1: Solução do problema do caixeiro viajante usando a estratégia de força bruta.

Trajeto	Custo
$a \to b \to c \to d \to a$	$2+3+7+5=17$
$a \to b \to d \to c \to a$	$2+4+7+8=21$
$a \to c \to b \to d \to a$	$8+3+4+5=20$
$a \to c \to d \to b \to a$	$8+7+4+2=21$
$a \to d \to b \to c \to a$	$5+4+3+8=20$
$a \to d \to c \to b \to a$	$5+7+3+2=17$

Pode-se ver que este tipo de solução se torna rapidamente inviável quando aumentamos o número de cidades envolvidas. De fato, pode-se mostrar que a eficiência desta abordagem é $(n-1)!$

Exemplo 5. O último problema que vamos abordar é o da mochila, que pode ser enunciado da seguinte maneira: sejam n ítens com pesos $w_1, w_2, \cdots, w_n$ e valores $v_1, v_2, \cdots, v_n$. Seja também uma mochila com capacidade W. Qual o subconjunto mais valioso de itens que pode ser carregado na mochila?

Uma instância deste problema é mostrado na Figura abaixo:

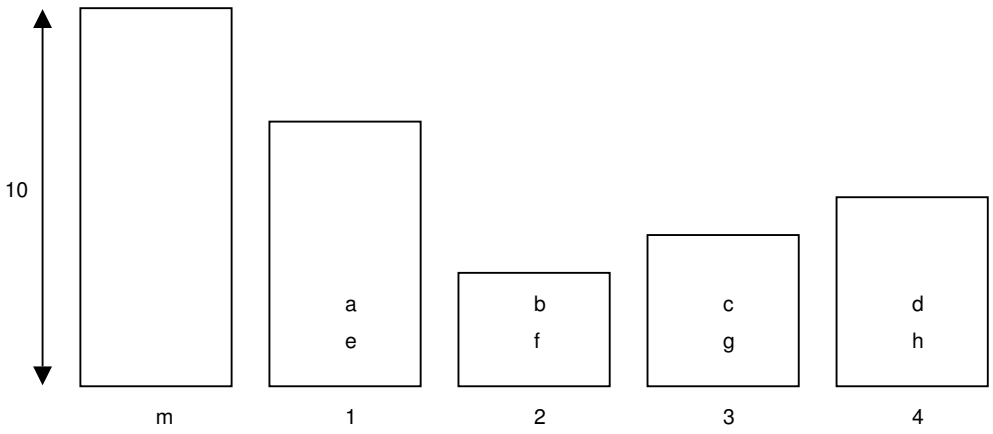


Figura 2: O problema da mochila.

Solução. A solução segundo a estratégia da força bruta consiste em listar todos os subconjuntos do conjunto de n itens dados, calculando o peso total de cada suconjunto para identificar os subconjuntos praticáveis.

Depois é só identificar o subconjunto com o valor mais elevado entre eles. Esta solução é mostrada na Tabela 2 para a instância dada:

Tabela 2: Solução do problema da mochila segundo a estratégia de força bruta. A solução encontra-se em negrito.

Subconjunto	Peso Total	Valor total
ϕ	0	\$0
$\{1\}$	7	\$42
$\{2\}$	3	\$12
$\{3\}$	4	\$40
$\{4\}$	5	\$25
$\{1,2\}$	10	\$36
$\{1,3\}$	11	excedeu o peso
$\{1,4\}$	12	excedeu o peso
$\{2,3\}$	7	\$52
$\{2,4\}$	8	\$37
$\mathbf{\{3,4\}}$	**9**	**\$65**
$\{1,2,3\}$	14	excedeu o peso
$\{1,2,4\}$	15	excedeu o peso
$\{1,3,4\}$	16	excedeu o peso
$\{2,3,4\}$	12	excedeu o peso
$\{1,2,3,4\}$	19	excedeu o peso

Como o número de subconjuntos de um conjunto de n elementos é 2^n, a busca exaustiva leva a um algoritmo $\Omega(2^n)$.

2.2 Comentários finais

A força bruta é uma estratégia direta para resolver um problema, geralmente baseada no enunciado do problema e definições dos conceitos envolvidos. A vantagem desta abordagem são a sua ampla aplicabilidade e grande simplicidade. Por outro lado, geralmente leva a soluções de baixa eficiência. Entretanto, é frequentemente possível melhorar este desempenho com uma modesta quantidade de esforço.

Os seguintes algoritmos podem ser considerados como exemplos de estratégias de força bruta:

- multiplicação de matrizes baseada na definição
- ordenação por seleção direta
- busca sequencial
- algoritmo direto de busca de string

A busca exaustiva é uma estratégia de força bruta para problemas combinatoriais que sugere gerar todos os objetos combinatoriais do problema, selecionar aqueles que satisfaçam as restrições do problema, e finalmente selecionar o objeto desejado. Exemplo de problemas típicos que podem ser resolvidos por busca exaustiva são o do caixeiro viajante e o da mochila.

Algoritmos de busca exaustiva são executados em um intervalo de tempo realista apenas para instâncias muito pequenas. Em muitos casos existem alternativas muito mais eficientes, mas em outros, infelizmente, é a única solução conhecida.

3 Divisão e conquista

Dividir a conquistar foi uma estatégia militar de grande sucesso bem antes de se tornar um paradigma de projeto de algoritmos. Os generias observaram que era mais fácil derrotar um exército de 50000 homens, seguido de um outro exército de 50000 homens do que um único exército de 100000 homens. Assim o general sábio deveria atacar o exército inimigo de forma a separá-lo em duas forças e então enfrentar um deles de cada vez.

Para usar a estratégia de divisão e conquista como uma técnica de projeto de algoritmos, precisamos dividir o problema em dois subproblemas menores, resolver cada um deles recursivamente, e então combinar as soluções parciais em uma solução para o problema completo. Se o processo de combinação das soluções tomar menos tempo do que a resolução de cada um dos subproblemas, teremos um algoritmo eficiente.

Podemos então resumir a estratégia de divisão e conquista nos passos abaixo:

1. dividir a instância do problema em duas ou mais instâncias menores;
2. resolver as instâncias menores de forma recursiva;
3. obter a solução para as instâncias originais (maiores) através da combinação destas soluções.

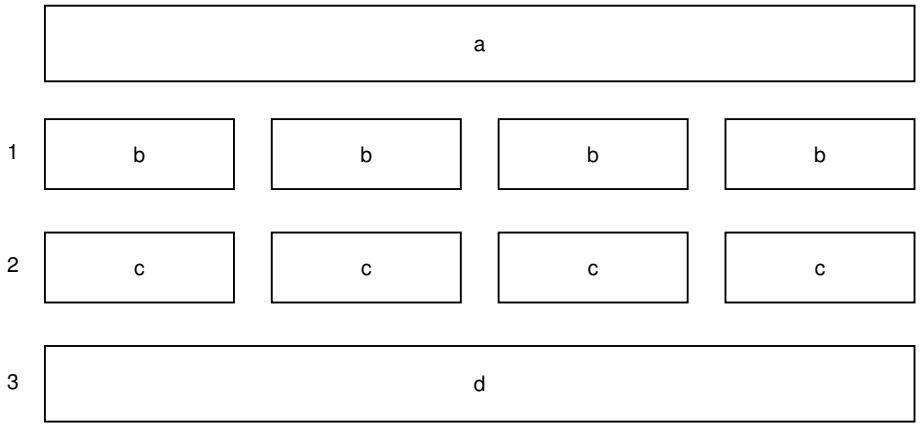


Figura 3: O método de divisão e conquista.

A técnica de divisão e conquista é responsável por alguns dos algoritmos mais importantes e eficientes da ciência da computação. De fato, o mais importante deles, o da transformada rápida de Fourier, responsável por toda a revolução digital, baseia-se nesta abordagem.

Outro atrativo desta técnica é a sua natural adequação a sistemas de computação paralela, o que faz com que seja mais eficiente ainda do ponto de vista temporal.

Vamos então a alguns exemplos clássicos desta abordagem para entendê-la melhor.

3.1 Análise de eficiência

Uma instância de tamanho n pode ser dividida em diversas instâncias de tamanho n/b, com a delas devendo ser resolvidas. Desta forma, obtemos a seguinte recorrência:

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

A análise de eficiência é feita através dos métodos vistos anteriormente.

3.2 Algoritmo Merge Sort

Suponha que temos um vetor A de n elementos para ordenar. O algoritmo *merge sort* pode ser descrito nos seguintes passos:

- dividir A em duas partes de mesmo tamanho (ou possivelmente com a diferença de um elemento) e copiar a primeira metade para o vetor B e a segunda metade para o vetor C;
- ordenar os vetores B e C;
- unir os vetores B e C ordenados de modo a gerar um vetor ordenado.

Em notação de pseudocódigo, este algoritmo fica assim:

```
algoritmo mergesort(A[0..n-1])
se (n>1)
  copie A[0..⌊n/2⌋-1] para B[0..⌊n/2⌋-1]
  copie A[⌊n/2⌋..n-1] para C[0..⌊n/2⌋-1]
  mergesort(B[0..⌊n/2⌋-1])
  mergesort(C[0..⌊n/2⌋-1])
  merge(B,C,A)
```

Este último passo (*merge*) pode ser implementado seguindo os seguintes passos:

- enquanto houver elementos em B e C:
  - se o primeiro elemento de B for menor ou igual ao primeiro elemento de C, mova o primeiro elemento de B para o vetor A;
  - caso contrário, mova o primeiro elemento de C para o vetor A;
- quando acabarem os elementos de um vetor, copiar os elementos que sobraram do outro para o vetor A.

```
algoritmo merge(B[0..p-1],C[0..q-1],A[0..p+q-1])
  i=0; j=0; k=0;
  enquanto (i<p e j<q)
    se (B[i]<=C[j])
      A[k++] = B[i++];
    senão
      A[k++] = C[j++];
  se (i==p)
    copie C[j..q-1] para A[k..p+q-1]
  senão
    copie B[i..p-1] para A[k..p+q-1]
```

3.2.1 Análise da eficiência do algoritmo *merge sort*

Assumindo que n é uma potência de 2, temos:

Número de comparações chave:

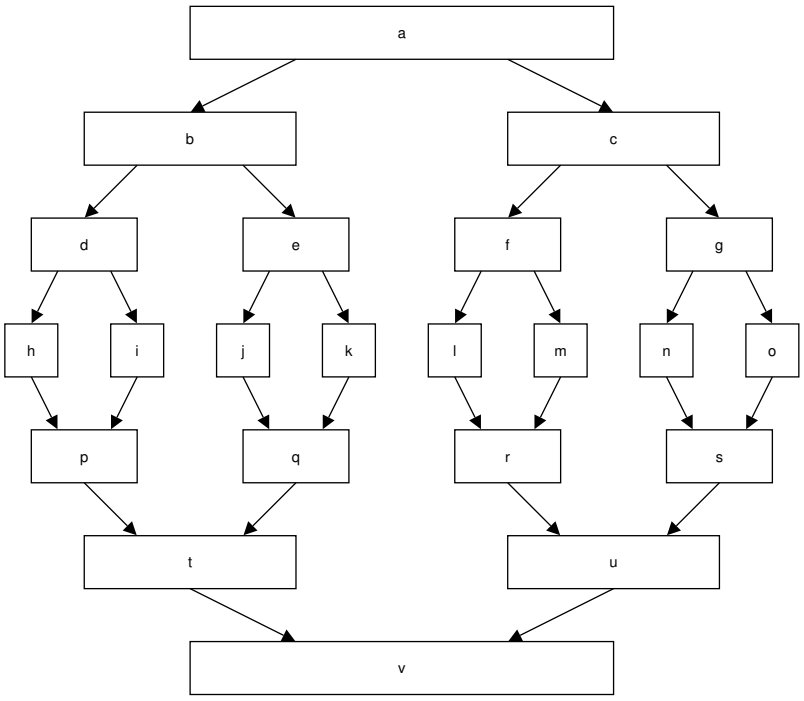
$$C(1) = 0 \text{ se } n = 1$$
$$C(n) = 2C(n/2) + C_{merge}(n) \text{ se } n > 1$$

Pior caso: $C_{merge}(n) = n - 1$

$$C_{pior}(1) = 0 \text{ se } n = 1$$
$$C_{pior}(n) = 2C(n/2) + n - 1 \text{ se } n > 1$$

Desta forma temos que, para o pior caso, este algoritmo é $O(n \log_2(n))$.

Exemplo 6. Seja um vetor com os seguintes dados: $\{8, 3, 2, 9, 7, 1, 5, 4\}$. A ordenação deste usando o algoritmo merge sort pode ser vista na figura abaixo:



O `mergesort` é um algoritmo clássico de divisão e conquista. Sempre que pudermos quebrar um problema grande em dois problemas menores, teremos vantagens porque os problemas menores são mais fáceis de serem resolvidos. O truque é tirar vantagem das soluções parciais para gerar a solução para o problema total. A operação `merge` é responsável por remontar a solução no `mergesort`, mas nem todos os problemas podem ser decompostos de forma tão engenhosa.

3.3 Método da troca e partição (*Quick Sort*)

Foi proposto por C. A. Hoare em 1962, e é considerado como o mais rápido em relação métodos apresentados anteriormente. hoare partiu da premissa de que é mais rápido classificar dois vetores de $n/2$ elementos do que um de n elementos.

Assim, este método divide o vetor a ser ordenado em três segmentos, como mostrado abaixo:

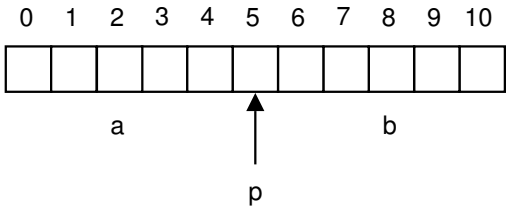


Figura 4: Segmentando o vetor a ser ordenado

O escolha do elemento **pivô**, que divide o vetor em dois segmentos pode ser realizada de forma arbitrária, mas normalmente é escolhido o elemento central, por tornar o algoritmo mais eficiente.

Em seguida é realizada uma movimentação nos elementos, de forma que no segmento 1 fiquem os elementos menores que o pivô, e no segmento 2, os elementos maiores que o pivô. Depois é a vez de aplicar o mesmo procedimento aos segmentos 1 e 2, de forma recursiva, até que o vetor original fique totalmente ordenado. Vamos ver mais de perto como isto funciona através de um exemplo. Seja então o seguinte vetor, que desejamos ordenar:

	0	1	2	3	4	5	6	7	8
`vetor`	21	37	12	13	20	19	43	15	23

O procedimento *Quick Sort* pode ser então descrito da seguinte maneira:

1. Seleciona-se um elemento para ser o pivô. Pode ser qualquer um, mas geralmente escolhe-se o elemento do meio. Este é então copiado para a variável auxiliar `pivo`.

	0	1	2	3	4	5	6	7	8
`vetor`	21	37	12	13	20	19	43	15	23

`pivo` | 20 |

2. Iniciamos dois ponteiros, `esq` e `dir`, apontando para o primeiro e último elementos do vetor, respectivamente.

	0	1	2	3	4	5	6	7	8
`vetor`	21	37	12	13	20	19	43	15	23
	`esq`								`dir`

`pivo` | 20 |

3. O vetor é percorrido a partir da esquerda até que se encontre um elemento `(vetor[esq] >= pivo)`, incrementando-se o valor da variável `esq`.

	0	1	2	3	4	5	6	7	8
`vetor`	21	37	12	13	20	19	43	15	23
		`esq`							`dir`

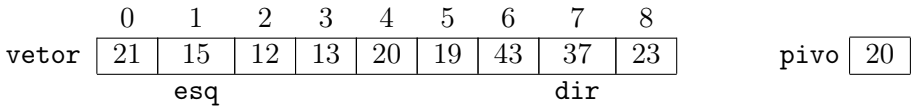
`pivo` | 20 |

4. O vetor é percorrido a partir da direita até que se encontre um elemento `(vetor[dir] < pivo)`, decrementando-se o valor da variável `dir`.

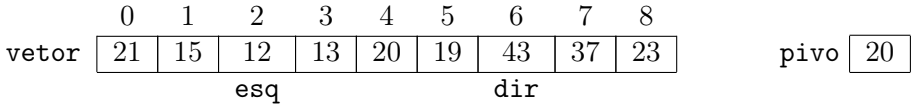
	0	1	2	3	4	5	6	7	8
`vetor`	21	37	12	13	20	19	43	15	23
		`esq`						`dir`	

`pivo` | 20 |

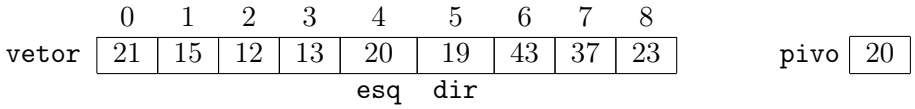
5. Os elementos `vetor[esq]` e `vetor[dir]` estão fora do lugar em relação ao pivô. Desta forma, suas posições são invertidas.

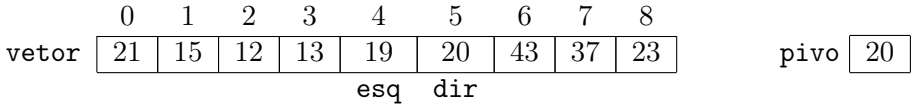


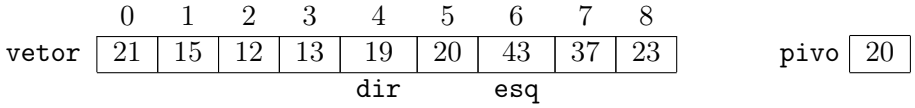
6. Incrementa-se `esq` e decrementa-se `dir`



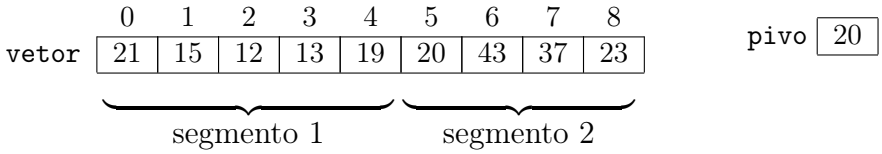
7. Este processo é repetido até que `esq` e `dir` se cruzem em algum ponto do vetor:







8. Após este passo, o pivô está na posição correta e temos dois segmentos a serem ordenados:



9. Depois, estes dois segmentos são ordenados de forma recursiva.

A seguir é mostrada uma função em C++ que implementa o algoritmo *Quick Sort* de forma recursiva. Este tem duas entradas adicionais em relação aos outros algoritmos: `i` e `j`. Estes dois parâmetros correspondem aos índices dos elementos inicial e final do vetor, respectivamente.

Mas espera um pouco: se temos um parâmetro vetorial e o tamanho do vetor na entrada, pra que precisamos ainda indicar as posições inicial e final do vetor? Isto não seria informação duplicada? A resposta é, sim e não: sim para a primeira chamada à função, quando levamos em conta o vetor inteiro. Nas chamadas recursivas (veja o passo 9), chamamos a função para processar só um pedaço do vetor, e aí sim os parâmetros `i` e `j` fazem sentido.

```
void quickSort(int vetor[], int tamanho, int i, int j)
{
  int trab,esq,dir,pivo;

  esq = i;
  dir = j;
  pivo = vetor[(int)round((esq + dir)/2.0)];
  do
  {
    while (vetor[esq] < pivo)
      esq++;
    while (vetor[dir] > pivo)
      dir--;
    if (esq <= dir)
    {
      trab = vetor[esq];
      vetor[esq] = vetor[dir];
      vetor[dir] = trab;
      esq++;
      dir--;
    }
  } while (esq <= dir);

  if (dir-i >= 0)
    quickSort(vetor,tamanho,i,dir);
  if (j-esq >= 0)
    quickSort(vetor,tamanho,esq,j);
}
```

3.4 Método de pesquisa binária.

Quando os elementos de um vetor estão previamente classificados segundo algum critério, então pesquisas muito mais eficientes podem ser conduzidas. Entre elas destaca-se o método de **pesquisa binária**.

Seu funcionamento é o seguinte: inicialmente o vetor é classificado (por exemplo, em ordem crescente). O elemento que divide o vetor ao meio (ao menos aproximadamente) com relação ao seu número de componentes é localizado e comparado ao valor procurado. Se ele for igual ao valor procurado a pesquisa é dita bem sucedida e é interrompida. No caso dele ser maior que o valor procurado, repete-se o processo na primeira metade do vetor. No caso do elemento central do vetor ser menor que o valor procurado, repete-se o processo na segunda metade do vetor. Este processo continua até que o elemento desejado seja localizado (pesquisa bem-sucedida), ou então, até que não reste mais um trecho do vetor a ser pesquisado (pesquisa mal-sucedida).

Este método tem a desvantagem de exigir que o vetor seja previamente ordenado para seu correto funcionamento, o que não acontece no caso da pesquisa sequencial. Por outro lado, é **em média** muito mais rápido que o método de pesquisa sequencial.

A seguir é apresentada uma função que implementa o método de pesquisa binária para um vetor do tipo `int`. As saídas dela são as mesmas da função anterior (pesquisa sequencial).

```
int binaria(int vetor[],int tamanho, int x)
{
  bool achou = false; // var aux p/ busca
  int baixo, meio, alto; // var aux

  baixo = 0;
  alto = tamanho-1;
  achou = false;
  while ((baixo <= alto) && (achou == false))
  {
    meio = (baixo + alto) / 2;
    if (x < vetor[meio])
      alto = meio - 1;
    else
      if (x > vetor[meio])
        baixo = meio + 1;
      else
        achou = true;
  }
  if (achou)
    return meio;
  else
    return -1;
}
```

3.5 Comentários

- Este paradigma não é aplicado apenas em problemas modelados recursivamente.
- Existem pelo menos três cenários onde divisão-e-conquista é aplicado:
  1. Eliminar partes do conjunto de dados a serem examinados. Exemplo: Pesquisa binária.
  2. Processar independentemente partes do conjunto de dados. Exemplo: Mergesort.
  3. Processar separadamente partes do conjunto de dados mas onde a solução de uma parte influencia no resultado da outra. Exemplo: Somador binário.

4 Programação dinâmica

Depois de entender como funciona, a programação dinâmica é provavelmente a técnica de projeto de algoritmo mais fácil de aplicar na prática. Entretanto, até conseguir entendê-la, a programação dinâmica parece mágica. Você precisa entender as artimanhas antes do poder usá-la.

Em algoritmos para problemas como a ordenação, a correção tende a ser mais fácil de verificar do que a eficiência. Este não é o caso para problemas de otimização, onde procuramos por uma solução que maximize ou minimize alguma função. No projeto de algoritmos para um problema de otimização, precisamos provar que nosso algoritmo sempre fornece a melhor solução possível.

Algoritmos gulosos, que tomam a melhor decisão local a cada passo, ocasionalmente produzem um ótimo global para certos problemas. Estes são tipicamente eficientes. Entretanto, precisamos de uma prova para mostrar que sempre terminaremos com a melhor resposta. Algoritmos de busca

exaustiva, que listam todas as possibilidades e selecionam a melhor, sempre encontram o resultado ótimo, mas geralmente a um custo proibitivo em termos de complexidade temporal.

A programação dinâmica combina o melhor de ambos os mundos. A técnica considera sistematicamente todas as decisões possíveis e sempre seleciona aquela que prova ser a melhor. Armazenando as consequências de todas as possíveis decisões até o momento e usando esta informação de forma sistemática, a quantidade total de trabalho é minimizada. Dynamic Programming is a technique for computing recurrence relations efficiently by sorting partial results.

4.1 Metodologia

O primeiro passo é descrever a solução para o problema inteiro em termos de soluções para problemas menores. Isto define um algoritmo recursivo.

O segundo passo é ver que o algoritmo recursivo resolve repetidamente os mesmos subproblemas diversas vezes, de modo que se armazenarmos as soluções destes subproblemas em uma tabela, podemos ter um algoritmo mais eficiente.

A recorrência da programação dinâmica tipicamente retorna apenas o custo da solução ótima, mas não retorna o caminho escolhido. Para reconstruir a solução precisamos armazenar também as decisões que tomamos em cada ponto local, para podermos recuperar o caminho que levou à solução ótima.

Como você já deve ter percebido, esta técnica é bastante difícil de compreender. A melhor maneira de aprender programação dinâmica é analizando cuidadosamente alguns exemplos até que as coisas comecem a clarear. Vamos então arregaçar as mangas e dar uma olhada nos exemplos a seguir.

4.2 Série de Fibonacci

Figura 5: Leonardo Pisano ou Leonardo de Pisa (1175-1250), também conhecido como Fibonacci.

Os números de Fibonacci foram definidos originariamente pelo matemático italiano Leonardo Pisano (depois conhecido como Fibonacci), no século XIII para modelar o crescimento de populações de coelhos. Os coelhos se reproduzem, bem, como coelhos. Fibonacci supôs que o número de pares de coelhoes que nascem em um determinado ano fosse igual ao número de pares de coelhos nascidos nos dois anos anteriores, se você iniciar com um par de coelhos no primeiro ano. Para contar o número de coelhos nascidos no n-ésimo ano, ele definiu a seguinte fórmula de recorrência:

$$F_n = F_{n-1} + F_{n-2}$$

com casos base $F_0 = 0$ e $F_1 = 1$. Desta forma, $F_2 = 1$, $F_3 = 3$. A série fica então:

$$\{0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, \dots\}$$

No final das contas, a fórmula de Fibonacci não deu muito certo para contar o número de coelhos, mas ela tem algumas aplicações e propriedades interessantes (como por exemplo, explicar programação dinâmica).

Desde que é definida por uma fórmula recursiva, é fácil escrever um programa recursivo para calcular o n-ésimo número da sequência de Fibonacci:

```
int fibonacci(int n)
{
  if (n==0)
    return 0;
  else
    if (n==1)
      return 1;
    else
      return (fibonacci(n-1)+fibonacci(n-2));
}
```

Se desenharmos a árvore de recursão para calcular o sexto elemento da sequência teríamos o seguinte:

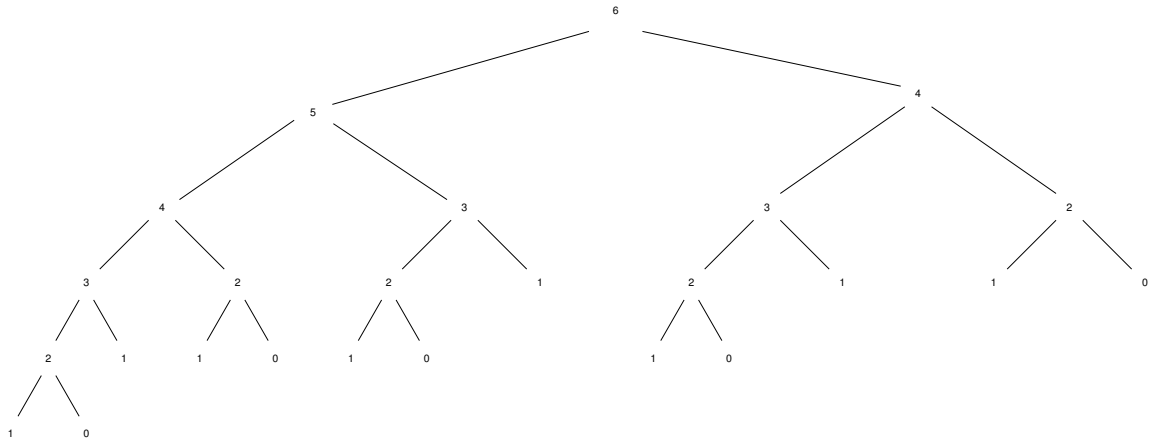


Figura 6: Árvore de recursão para cálculo da sequência de Fibonacci.

Veja que para calcular $F(6)$, calculamos $F(5)$ uma vez, $F(4)$ duas vezes, $F(3)$ 3 vezes, $F(2)$ cinco vezes, $F(1)$ 8 vezes e $F(0)$ 5 vezes! Isto é um bocado de conta jogada fora! Pode-se provar que a complexidade deste algoritmo é exponencial $(1,6^n)$, o que é pra lá de ruim.

De fato, podemos fazer muito melhor: é possível calcular $F(n)$ com complexidade temporal linear se armazenarmos os valores previamente calculados. Trocamos tempo por espaço em memória no algoritmo abaixo:

```
f[0] = 0;
f[1] = 1;

for (i=2;i<n;i++)
  f[i] = f[i-1]+f[i-2];
```

Como avaliamos os números de Fibonacci do menor para o maior e armazenamos todos os resultados, temos sempre F_{i-1} e F_{i-2} disponíveis para calcular F_i. Desta forma, cada um dos n valores é calculado como uma soma simples de dois inteiros com tempo total $O(n)$, o que é uma tremenda melhoria sobre uma complexidade exponencial.

4.2.1 Comentários

A conclusão que pode ser tirada deste exemplo é que quando os subproblemas se repetem várias vezes, a técnica de divisão e conquista não é adequada.

Se formos descrever a estratégia que usamos para resolver este problema, podemos enumerar os seguintes passos:

- definimos uma solução recursiva para o problema
- mudamos esta solução para uma forma que usa uma estrutura de repetição, e armazenamos os resultados dos subproblemas pra não ter que repetir os cálculos já efetuados.

Estes dois passos são na verdade um subconjunto dos passos necessários para estabelecer uma solução para um problema usando programação dinâmica. A receita completa é dada a seguir:

1. caracterizar a estrutura de uma solução ótima;
2. definir recursivamente o valor de uma solução ótima;
3. calcular o valor de uma solução ótima em um processo de baixo para cima (*bottom-up*);
4. construir uma solução ótima a partir de informações calculadas.

Os passos de 1 a 3 formam a base de uma solução por programação dinâmica. O passo 4 nem sempre é necessário. Confuso? Naturalmente. Vamos a mais um exemplo pra tentar clarear as idéias. Depois dele, volte a esta seção, e você verá como as coisas ficam mais claras.

4.3 Programação de linha de montagem

Uma fábrica produz automóveis em uma fábrica que tem duas linhas de montagem, como mostradas na Figura 7

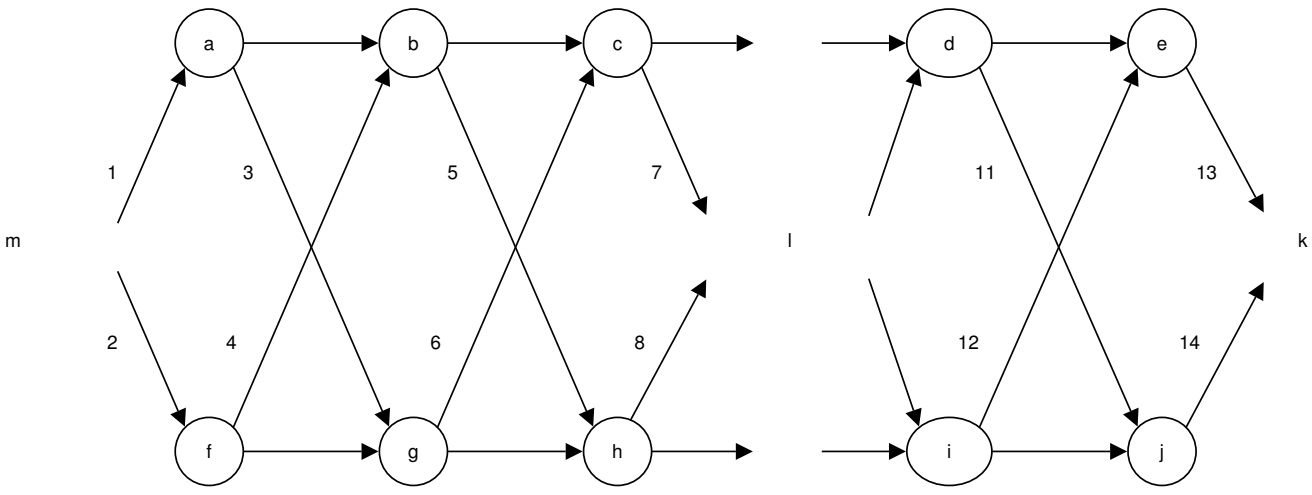


Figura 7: O problema do caminho mais rápido.

Um chassi de automóvel entra em cada linha de montagem, tem as peças adicionadas a ele em uma série de estações, e um automóvel pronto sai no final da linha.

Cada linha de montagem tem n estações, numeradas com $j = 1, 2, \cdots, n$. Denotamos a j-ésima estação da linha i (onde $i \in \{1, 2\}$) como $S_{i,j}$.

A j-ésima estação da linha 1 ($S_{1,j}$) executa a mesma função que a j-ésima estação da linha 2 ($S_{2,j}$), mas como estas foram construídas em épocas diferentes, podem ter seu tempo de execução diferentes. Denotamos o tempo de montagem exigido na estação $S_{i,j}$ por $a_{i,j}$.

Conforme mostrado na Figura 7, um chassi entra na estação 1 de uma das linhas de montagem e avança de cada estação para a seguinte. Também há um tempo de entrada e_i para o chassi entrar na linha de montagem i, e um tempo de saída para o automóvel concluído sair da linha de montagem i.

Normalmente, uma vez que um chassi entra em uma linha de montagem, ele percorre apenas esta linha. O tempo para ir de uma estação à seguinte dentro da mesma linha de montagem é desprezível. Às vezes chega um pedido especial de urgência, e o cliente quer que o automóvel seja fabricado o mais rápido possível.

Para pedidos de urgência, o chassi passa ainda pelas n estações em ordem, mas o gerente da fábrica pode passar o automóvel parcialmente concluído de uma linha de montagem para outra após qualquer estação. O tempo para transferir um chassi da linha de montagem i depois da passagem pela estação $S_{i,j}$ é $t_{i,j}$, onde $i = 1, 2$ e $j = 1, 2, \cdots, n-1$ (pois após a n-ésima estação a montagem já está completa).

O problema consiste em determinar quais estações escolher na linha 1 e quais escolher na linha 2 para minimizar o tempo total de passagem de um único automóvel pela fábrica.

A maneira óbvia de resolver o problema, utilizando a estratégia de força bruta, é listar todos os caminhos possíveis e escolher o de menor duração, o que é inviável quando temos muitas estações de trabalho. De fato, este é um problema de complexidade $\Omega(2^n)$. Vamos então a uma solução usando programação dinâmica.

Etapa 1: a estrutura do caminho mais rápido pela fábrica

Vamos considerar o modo mais rápido possível para um chassi seguir desde o ponto de partida passando pela estação $S_{1,j}$:

- se $j = 1$, só existe um caminho, e desta forma é fácil descobrir quanto tempo ele demora para passar pela estação $S_{1,j}$;
- para $j = 2, 3, \cdots, n$, há duas opções:
  - o chassi veio da estação $S_{1,j-1}$ ou
  - o chassi veio da estação $S_{2,j-1}$

Este último ponto merece uma análise mais detalhada: primeiro, vamos supor que o caminho mais rápido passando por $S_{1,j}$ passe pela estação $S_{1,j-1}$. A observação fundamental é que o chassi deve ter tomado um caminho mais rápido desde o ponto de partida e através da estação $S_{1,j-1}$. Porquê? Se houvesse um caminho mais rápido através da estação $S_{1,j-1}$ poderíamos utilizar este caminho mais rápido como substituto para produzir um caminho mais rápido pela estação $S_{1,j}$, o que é uma contradição. O mesmo raciocínio vale pelo caminho mais rápido passando por $S_{2,j}$.

Em termos gerais, podemos dizer que, no caso de programação da linha de montagem, uma solução ótima para um problema (encontrar o caminho ótimo passando pela estação $S_{i,j}$) contém em seu interior uma solução ótima para subproblemas (encontrar a passagem mais rápida por $S_{1,j-1}$ ou $S_{2,j-1}$). Vamos nos referir a esta propriedade como subestrutura ótima, e ela é uma das indicações da aplicabilidade da programação dinâmica, como vimos na Seção 4.1.

Etapa 2: uma solução recursiva

Seja $f_i(j)$ o tempo mais rápido possível para levar um chassi desde o ponto de partida até a estação $S_{i,j}$.

O objetivo final é descobrir o tempo mais rápido para levar um chassi por todo o percurso na fábrica, que denotaremos por f^*. O chassi deve passar por todo o caminho até a estação n na linha 1 ou na linha 2, e depois até a saída da fábrica. Em termos matemáticos, podemos escrever:

$$f^* = \min(f_1(n) + x_1, f_2(n) + x_2) \tag{1}$$

Também é fácil raciocinar sobre $f_1(1)$ e $f_2(1)$:

$$f_1(1) = e_1 + a_{1,1} \tag{2}$$
$$f_2(1) = e_2 + a_{2,1} \tag{3}$$

Para os demais valores de j, ou seja, $j = 2, 3, \cdots, n$, temos:

$$f_1(j) = \min(f_1(j-1) + a_{1,j}, f_2(j-1) + t_{2,j-1} + a_{1,j}) \tag{4}$$
$$f_2(j) = \min(f_2(j-1) + a_{2,j}, f_1(j-1) + t_{1,j-1} + a_{2,j}) \tag{5}$$

Etapa 3: cálculo dos tempos mais rápidos

Neste ponto, é fácil propor um algoritmo recursivo que resolva o problema. Entretanto, uma solução recursiva para este problema teria complexidade $\Theta(2^n)$!

Entretanto, podemos fazer melhor, usando a técnica de programação dinâmica: observe que a partir da segunda máquina em cada linha, o tempo total depende apenas dos tempos totais acumulados na máquina anterior, para as duas linhas. Desta forma, calculando os valores de $f_i(j)$ na ordem de números de estacão j crescentes - ou seja, da esquerda para a direita na Figura 7 - podemos calcular o caminho mais rápido pela fábrica e o tempo que ele demora com tempo $\Theta(n)$.

Este procedimento é mostrado no trecho de código abaixo:

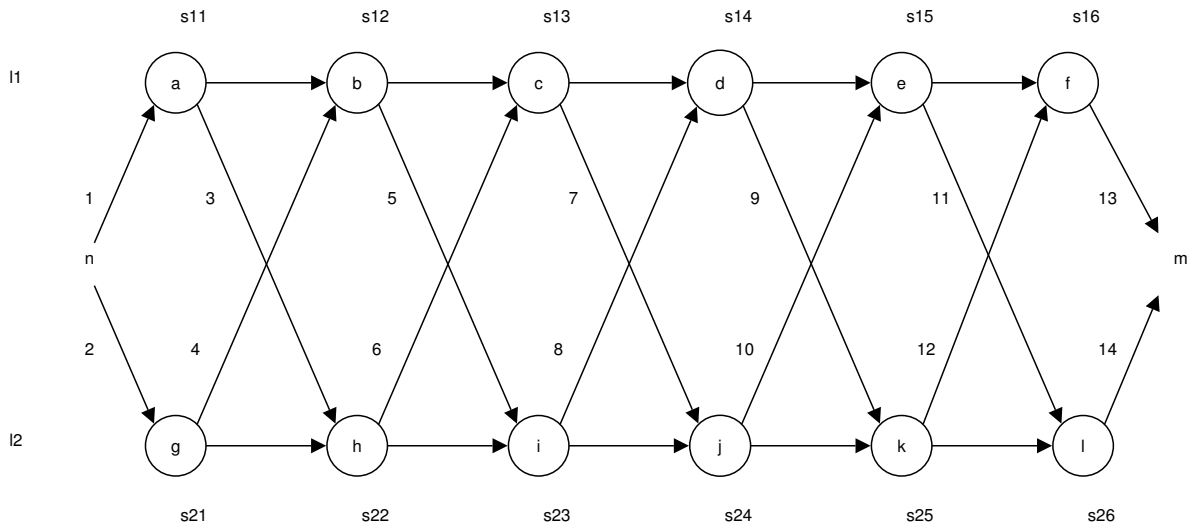
```
// Inicialização
for (i=0;i<nLinhas;i++)
{
  f[i][0] = m[i][0].t[i]+m[i][1].a;
  l[i][0] = 0;
}

// Indução
for (k=1;k<nMaquinas;k++)
  for (i=0;i<nLinhas;i++)
  {
    for (j=0;j<nLinhas;j++)
      custo[j] = f[j][k-1]+m[j][k].t[i]+m[i][k+1].a;
    minimo(custo,nLinhas,valor,linha);
    f[i][k] = valor;
    l[i][k] = linha;
  }

// Ultima maquina
for (i=0;i<nLinhas;i++)
{
  f[i][nMaquinas] = f[i][nMaquinas-1]+m[i][nMaquinas].t[i];
  l[i][nMaquinas] = i;
}
```

Etapa 4: caminho mais rápido pela fábrica

O procedimento para descobrir o caminho mais rápido pela fábrica já foi mostrado na Figura 7. O código abaixo impementa esta idéia:

```
int estacao = nMaquinas;
float minimo = f[0][nMaquinas];
int linha = 0;
for (int i=1;i<nLinhas;i++)
{
  if (f[i][nMaquinas] < minimo)
  {
    minimo = f[i][nMaquinas];
    linha = i;
  }
}
cout << "Linha " << linha << ", estação " << estacao-- << endl;
while (estacao > 0)
{
  linha = l[linha][estacao];
  cout << "Linha " << l[linha][estacao] << ", estação " << estacao-- << endl;
}
```

Exemplo 7. Seja a linha de montagem mostrada na figura abaixo:



Neste caso, as matrizes f e l são dadas por

jh	0	1	2	3	4	5	6
f1	a	b	c	d	e	f	g
f2	h	i	j	k	l	m	n

jh	0	1	2	3	4	5	6
f1	a	b	c	d	e	f	g
f2	h	i	j	k	l	m	n

4.4 Multiplicação de uma cadeia de matrizes

Suponhamos que precisamos multiplicar uma sequência longa de matrizes $A \times B \times C \times D \ldots$

Antes de detalhar o problema, vamos recordar algumas propriedades da multiplicação de matrizes:

- a multiplicação de uma matriz $(i \times j)$ por uma matriz $(j \times k)$ (usando o algoritmo convencional) demanda $i \times j \times k$ multiplicações.
- a multiplicação de matrizes é associativa. Assim, $(A \times B) \times C = A \times (B \times C)$.
- a multiplicação de matrizes não é comutativa, ou seja, $A \times B \neq B \times A$.

Exemplo 8. Suponha que queremos calcular

$$A \times B \times C \times D$$

onde as dimensões das matrizes são:

- A: 30×1
- B: 1×40
- C: 40×10
- D: 10×25

Existem três formas de calcular este produto:

1. ((AB)C)D=30 * 1 * 40+30 * 40 * 10+30 * 10 * 25 = 20700 multiplicações
2. (AB)(CD)=30 * 1 * 40+40 * 10 * 25+30 * 40 * 25 = 41200 multiplicações
3. A((BC)D)=1 * 40 * 10+1 * 10 * 25+30 * 1 * 25 = 1400 multiplicações

---

Desta forma, podemos ver que a ordem na qual realizamos as multiplicações tem uma grande influência no esforço total (e consequentemente no tempo de processamento). O problema consiste então em determinar qual a ordem em que as matrizes devem ser multiplicadas, de modo a minimizar o esforço total (medido aqui como o número de multiplicações realizadas).

Colocando de outra forma, desejamos calcular

$$A_{1..n} = A_1 \times A_2 \times \cdots \times A_n$$

com o menor número possível de multiplicações escalares.

Para facilitar a nossa vida, vamos criar um vetor p com as dimensões das matrizes, onde as dimensões de A_i são armazenadas como $p[i-1]$ linhas e e $p[i]$ colunas. É facil ver que este vetor tem comprimento $n+1$. No caso das matrizes do exemplo anterior, temos $p = \{30, 1, 40, 10, 25\}$.

Vamos ver agora como aplicar os quatro passos da programação dinâmica para resolver este problema.

Etapa 1: a estrutura de uma colocação ótima de parênteses.

Suponha que temos uma sequência de matrizes para multiplicar:

$$A_{i..j} = A_i \times A_{i+1} \times \cdots \times A_j$$

Desejamos dividir este produto da seguinte forma:

$$(A_i \times A_{i+1} \times \cdots \times A_k) \times (A_{k+1} \times A_{k+2} \times \cdots \times A_j)$$

de modo a minimizar o número de multiplicações escalares necessárias.

Podemos mostrar que o custo desta decisão é igual à soma de 3 custos:

- o custo de calcular $A_i \times A_{i+1} \times \cdots \times A_k$
- o custo de calcular $A_{k+1} \times A_{k+2} \times \cdots \times A_j$
- o custo de calcular o produto destas duas matrizes entre si

Etapa 2: uma solução recursiva

Seja $m[i][j]$ o número mínimo de multiplicações para calcular $A_{i..j}$. Assim, para o problema completo, o custo do caminho mais econômico para calcular $A_{1..n}$ é $m[1][n]$.

Podemos definir $m[i][j]$ recursivamente, da seguinte maneira:

- $i = j$: neste caso a cadeia só tem uma matriz, e portanto não temos nada para multiplicar. Assim: $m[i][i] = 0, i = 1, 2, \ldots, n$.

- $i < j$: suponha que o ponto ótimo de dividir esta sequência em duas seja no ponto k, como vimos na primeira etapa. O custo desta decisão é então:

$$m[i][j] = m[i][k] + m[k+1][j] + p[i-1]p[k][j]$$

Esta solução recursiva pressupões que sabemos o valor de k. Como não sabemos, temos que testar todos os valores pra ver qual dá o menor custo. A sorte é que os valores possíveis para k vão de i até $j-1$. Com tudo isto, a solução recursiva para o problema fica:

$$m[i][j] = \begin{cases} 0, & i = j \\ \min\limits_{i \leq k < j} \{m[i][k] + m[k+1][j] + p_{i-1}p_k p_j\}, & i < j \end{cases}$$

Etapa 3: cálculo dos custo ótimos

Vamos agora estabelecer uma estratégia *bottom-up* para resolver este problema. Mas que raios quer dizer *bottom-up*? Basicamente é o seguinte:

- o algoritmo calcula primeiro $m[i][i] = 0$ para $i = 1, 2, \cdots, n$ (custos mínimos para cadeias de comprimento 1);
- depois, na primeira passagem pelo loop em l, calcula o custo mínimo para cadeias de $l = 2$ matrizes $m[i][i+1]$, $i = 1, 2, \cdots, n-1$. Veja que a solução para este problema também é trivial;
- na segunda passagem pelo loop ($l = 3$), calcula o custo de sequências de três matrizes: $m[i][i+2]$, $i = 1, 2, \cdots, n-2$. Isto já é mais difícil de fazer. Vamos a um exemplo pra entender melhor: suponha que queremos determinar $m[1][3]$, isto é, o número mínimo de multiplicações para calcular $A_1 \times A_2 \times A_3$. Neste caso, temos duas alternativas possíveis:

  $(A_1 \times A_2) \times A_3$, com custo $m[1][2] + p[0]p[2]p[3]$

  $A_1 \times (A_2 \times A_3)$, com custo $p[0]p[1]p[3] + m[2][3]$

  $m[1][3]$ será dada então pelo menor delas. Aproveitamos também para anotar aonde quebramos a sequência, em uma matriz s, para podermos saber também em que ordem devemos resolver o problema total. No exemplo dado, se a primeira opção gerar o menor custo, fazemos $s[1][3] = 2$, caso contrário, fazemos $s[1][3] = 1$.
- vamos nesta toada, calculando o custo de sequências de 4 matrizes, e assim por diante, até chegarmos a uma sequência de n matrizes, que é o nosso objetivo.

O código abaixo implementa esta estratégia:

```
void matrixChainOrder
(
  int *p,
  int lengthp
)
{
  int i,j,k,l; // contadores
  int n;       // número de matrizes a serem multiplicadas
  int **m;     // armazena o número de multiplicações necessárias
  int **s;     // backpointer para recuperar a sequência ótima
  int q; // var aux
  n=lengthp-1;
  m = new int *[n];
  s = new int *[n];
  for (i=0;i<n;i++)
  {
    m[i] = new int[n];
    s[i] = new int[n];
  }
  for (i=0;i<n;i++)
    for (j=0;j<n;j++)
    {
      m[i][j]=0;
      s[i][j]=0;
    }
  for (l=2;l<=n;l++)
    for (i=1;i<=n-l+1;i++)
    {
      j=i+l-1;
      m[i-1][j-1] = INF;
      for (k=i;k<=j-1;k++)
      {
        q = m[i-1][k-1] + m[k+1-1][j-1]+p[i-1]*p[k]*p[j];
        if (q < m[i-1][j-1])
        {
          m[i-1][j-1] = q;
          s[i-1][j-1] = k;
        }
      }
    }
    // Desalocando memória
    for (i=0;i<n;i++)
    {
      delete [] m[i];
      delete [] s[i];
    }
    delete [] m;
    delete [] s;
}
```

Exemplo 9. Se dermos como entrada para este programa o vetor $p = \{30, 35, 15, 5, 10, 10, 25\}$, correspondente às matrizes

matriz	dimensão
A_0	30×35
A_1	35×15
A_2	15×5
A_3	5×10
A_4	10×20
A_5	20×25

teremos como saída, as seguintes matrizes:

x

y \ x	0	1	2	3	4	5
0	0	a	b	c	d	e
1		0	f	g	h	i
2			0	j	k	l
3				0	m	n
4					0	o
5						0

t

x

y \ x	0	1	2	3	4	5
0		0	0	2	2	2
1			1	2	2	2
2				2	2	2
3					3	4
4						4
5						

t

Etapa 4: construção de uma solução ótima

Falta agora determinar em que ordem devemos multiplicar as matrizes, baseados na informação que armazenamos na matriz s.

Vamos lembrar o que significa esta matriz: cada elemento de $s[i][j]$ registra o valor de k tal que a divisão ótima de $A_{i..j}$ divide o produto entre A_k e A_{k+1}. Com base nesta colocação, temos o seguinte:

- a última multiplicação a ser feita é entre A_k e A_{k+1}, ou escrevendo de outra forma, a multiplicação final no cálculo de $A_{1..n}$ é $A_{1..s[1][n]} \times A_{s[1][n]+1,n}$.
- com isso, precisamos achar como calcular as matrizes $A_{1..k}$ e $A_{k+1..n}$, ou seja, as matrizes $A_{1..s[1][n]}$ e $A_{s[1][n]+1,n}$. Estas informações estão nas posições $s[1][s[1][n]]$ e $s[s[1][n] + 1][n]$, respectivamente;
- este procedimento é repetido até termos todas as informações. Este procedimento leva ao algoritmo revursivo mostrado abaixo:

```
void printOptimalParens(int **s,int i,int j)
{
  int k=s[i][j];
  cout << k << "\t";
  if (i<k)
    printOptimalParens(s,i,k);
  if (k+1<j)
    printOptimalParens(s,k+1,j);
}
```

Esta função irá mostrar a ordem em que as matrizes devem ser multiplicadas, mas de trás para frente. Para obter a ordem correta, é só invertê-la.

Exemplo 10. No caso do exemplo anterior, a saída desta função será:

```
2 0 1 4 3
```

o que equivale ao seguinte:

$((A_0 \times (A_1 \times A_2)) \times ((A_3 \times A_4) \times A_5)$

---

5 Exercícios

1. Para o problema de programação de linha de montagem, mostre como modificar o procedimento de mostrar a ordem em que as linhas são escolhidas para imprimir as estações em ordem crescente de número da estação.

2. Encontre a sequência ótima para multiplicar um produto de cadeias de matrizes cujas sequência de dimensões é: $\{5, 10, 3, 12, 5, 50, 6\}$.

3. Forneça um algoritmo recursivo `matrix_chain_multiply(A,s,i,j)` que realmente execute a multiplicação ótima de cadeia de matrizes, dada a sequência de matrizes $A_1, A_2, \cdots, A_n$, matriz s calculada por `matrixchainorder` e os índices i e j. (A chamada inicial seria `matrix_chain_multiply(A,s,1,n)`)