Análise de Algoritmos

Nem todos os **problemas algorítmicos** que podem ser resolvidos em princípio podem ser resolvidos na prática: os recursos computacionais requeridos (tempo ou espaço) podem ser proibitivos.

- Suponha que duas soluções corretas diferentes sejam apresentadas para um mesmo problema matemático.

- Qual delas é a melhor?
  - Esta pergunta é praticamente impossível de ser respondida dada a dificuldade de encontrar critérios que as diferenciem de forma convincente.

- Considere agora dois algoritmos para a mesma tarefa.

- Qual deles é o melhor?

- Em geral, dizemos
  - que o mais rápido e o que consome menos memória.

- Tempo de execução é uma medida natural,
  - embora o processador, memória e compilador possam influenciar nos resultados da medição...
  - por exemplo, quando grandes quantidades de memória são usadas, as medidas de tempo são influenciadas por esse aspecto

Introdução

- Objetivos:

  Mostrar como medir o tempo de execução de um algoritmo através de ferramentas matemáticas

  que são independentes da máquina, da linguagem de implementação e do compilador.

- Tópicos:

- Medindo tempo em computador real versus modelo genérico de processador
- Custo de execução de um algoritmo
- Análise de um algoritmo versus análise da classe de problemas
- Análise de pior caso, melhor caso e caso médio
- Limite inferior para uma classe de problemas

Por que analisar um algoritmo?

- Descobrir suas características para avaliar sua adequação para várias aplicações
  - Tempo de execução – 2 formas
    - Custo real e custo não aparente como alocação de memória, carga, etc.
    - Número de vezes que uma operação relevante é executada
  - Espaço ocupado – memória necessária
- Compará-lo com outros algoritmos para a mesma aplicação

Formas de medir o custo de execução

- Execução em computador real (Empírica)
  - Importante quando existem vários algoritmos de uma mesma classe de complexidade

- Uso de um modelo genérico de processador, com execução seqüencial das operações (Teórica)

Independência da Análise com a Implementação

- Os resultados da análise dependem do compilador, da arquitetura de hardware e do ambiente operacional
  - Porém, na prática, ao invés de avaliar um algoritmo em um computador real usa-se um modelo genérico de um processador em que as instruções são executadas umas após as outras sem nenhuma concorrência, como uma MT.
  - Ainda, ignoramos o custo de algumas operações e consideramos apenas as mais significativas (ex: na ordenação consideramos o número de comparações de chaves)
  - Todo comando de atribuição, leitura, escrita, avaliação de comando condicional e avaliação em laço tem tempo constante.
  - Tempo não é tempo!! na realidade a complexidade de tempo representa o número de vezes que determinada operação relevante é executada

As abordagens são complementares

- As duas são necessárias no projeto e análise de algoritmos eficientes

  - Quando projetando um novo algoritmo utilizamos estudos de complexidade computacional em um modelo genérico para ter uma rápida idéia de como ele vai se comportar.
    - Este é o primeiro passo.

  - Uma análise de tempo real mais refinada é necessária para predizer o desempenho numa implementação real ou para compará-lo com outros algoritmos para a mesma tarefa.
    - Nesse segundo passo consideramos custos reais de operações e custos não aparentes como alocação de memória, indexação, carga, etc.

Como medir o custo de execução de um algoritmo

- Função de Custo ou Função de Complexidade

f(n) = medida de custo necessário para executar um algoritmo para um problema de tamanho de entrada n

  - Se f(n) é uma medida da quantidade de tempo necessária para executar o algoritmo então f é chamada de função de função de complexidade de tempo
  - Se f(n) é uma medida da quantidade de memória necessária para executar o algoritmo então f é chamada função de complexidade de espaço

Dois Tipos de Análises Predominam

- Análise de um particular algoritmo
  - Análise do número de vezes que cada parte do algoritmo deve ser executada
  - Estudo da quantidade de memória necessária

- Análise de uma classe de algoritmos
  - Estabelece limites para a complexidade computacional dos algoritmos pertencente à classe
    - Quanto determinamos o menor custo possível (limite inferior) para resolver problemas de uma classe, temos uma medida da dificuldade inerente da classe
  - Investiga-se toda uma classe para identificar um que seja o melhor possível
    - Algoritmo ótimo – custo é igual ao limite inferior
  - Veremos 3 exemplos.

Exemplo 1: Problema da Ordenação

Teorema 1 (Complexidade da Ordenação) – Qualquer algoritmo de ordenação baseado na abordagem de comparação deve usar pelo menos log N! >= (n/2) log n – n/2 comparações para uma entrada. Requer n log n no pior caso.

Prova inteira em Aho, Hopcroft e Ullman – Data Structures and Algorithms

Intuitivamente o resultado segue da observação que:

- cada comparação pode reduzir o número de possíveis permutações de elementos a serem considerados, no máximo por um fator de 2.

Desde que existem N! possíveis permutações antes da ordenação e o objetivo é ter somente uma no final, o número de comparações deve ser pelo menos o número de vezes que N! pode ser dividido por 2 antes de alcançar um número menor que 1.

Aproximação de Stirling

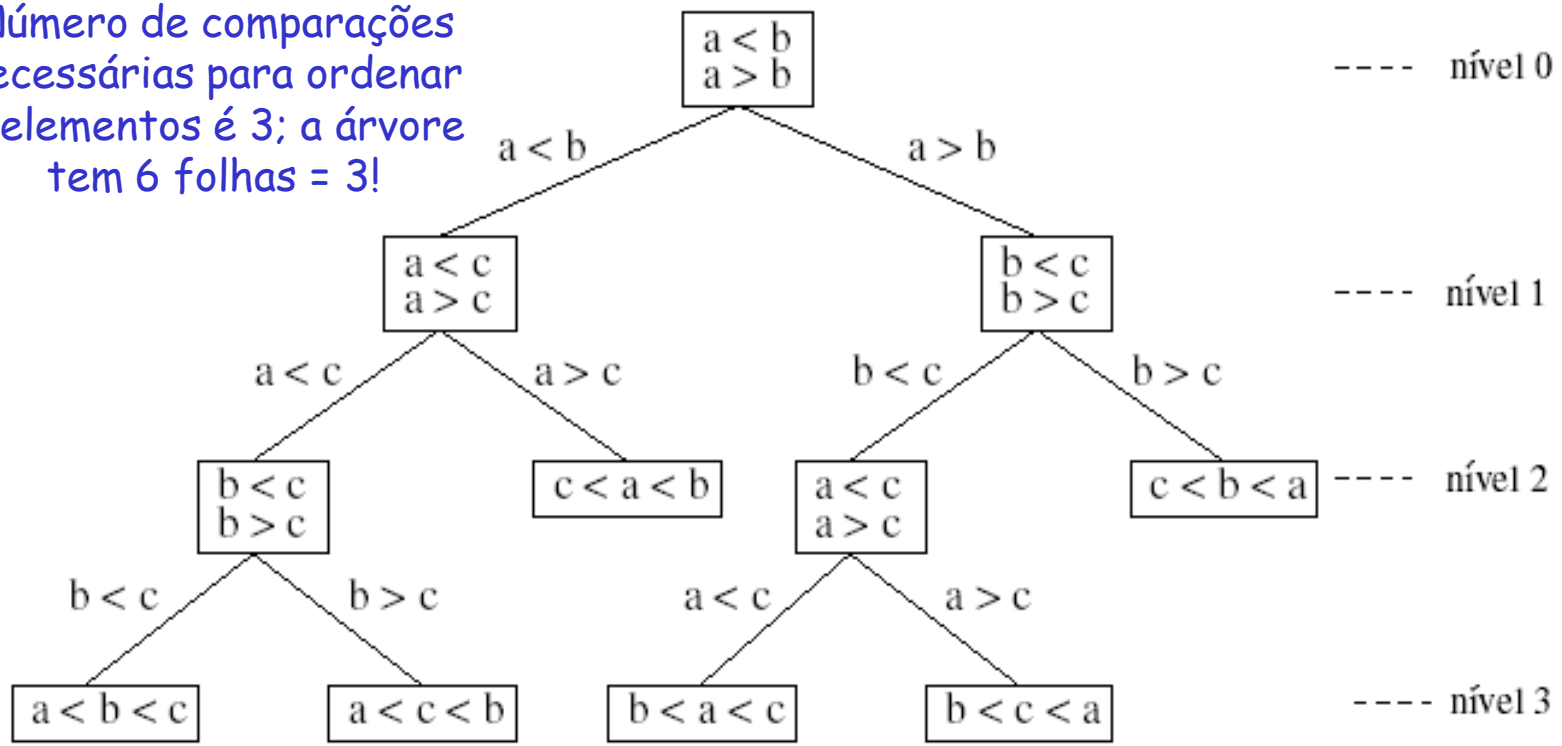
- Qualquer algoritmo de ordenação de n itens por comparação deve ter uma árvore de decisão com n! folhas, de altura pelo menos igual a log(n!).

- n log(n) é um limite inferior para log(n!),
  - assim nenhum algoritmo de ordenação por comparação pode ser mais rápido do que O(n log(n)).

Árvore de decisão mostrando comparações na ordenação de 3 elementos

abc
acb
bac
bca
cba
cab

Número de comparações necessárias para ordenar 3 elementos é 3; a árvore tem 6 folhas = 3!



N! ~$(n/e)^n$ e, sendo log $(n/e)^n$ = n log n – n log e, ele é da ordem de n log n.

Mais precisamente: N! é o produto da metade de fatores que são cada um pelo menos n/2.

Assim, n! >= $(n/2)^{n/2}$

log(n!) >= (n/2) log (n/2) = (n/2) log n - n/2

Do ponto de vista da Complexidade, o resultado do teorema demonstra que N log N é um limite inferior para a dificuldade do problema de ordenação por comparação de chaves.

Ordenação por Distribuição

- Existem métodos de ordenação que utilizam o princípio da **distribuição**.

- E utilizam informação a respeito da chave
  - Por exemplo, o fato dela poder ser quebrada em partes A e B e que A seja mais significativo que B
  - Exemplos de tipos de dados que permitem a aplicação:
    - inteiros (começa do dígito menos significativo - unidade) e
    - strings (O(n*L) com n strings e L o tamanho da maior string = O(n))

- Exemplo de ordenação por distribuição:
  - considere o problema de ordenar um baralho com 52 cartas
  - Cada carta é representada como U = U1 x U2 e usando ordem lexicográfica:

A < 2 < 3 < < 10 < J < Q < K (U2)

♣ < ♦ < ♥ < ♠ (U1)

Algoritmo

1. Distribuir as cartas abertas em treze montes: ases, dois, três, : : :, reis.
2. Colete os montes na ordem especificada.
3. Distribua novamente as cartas abertas em quatro montes: paus, ouros, copas e espadas.
4. Colete os montes na ordem especificada.

OBS1: ordenamos primeiro U2 depois U1

Radixsort, bucketsort (ou binsort)

- Métodos como o ilustrado são também conhecidos como **ordenação digital**, **radixsort (least-significant-digit-first radix sorting)** ou **bucketsort/binsort**.

- Binsort que trabalha somente com a permutação de valores entre 0..n-1

```
for (i=0; i<n; i++)
    B[A[i]] = A[i];
```

- O radix usa o binsort em cada parte da chave

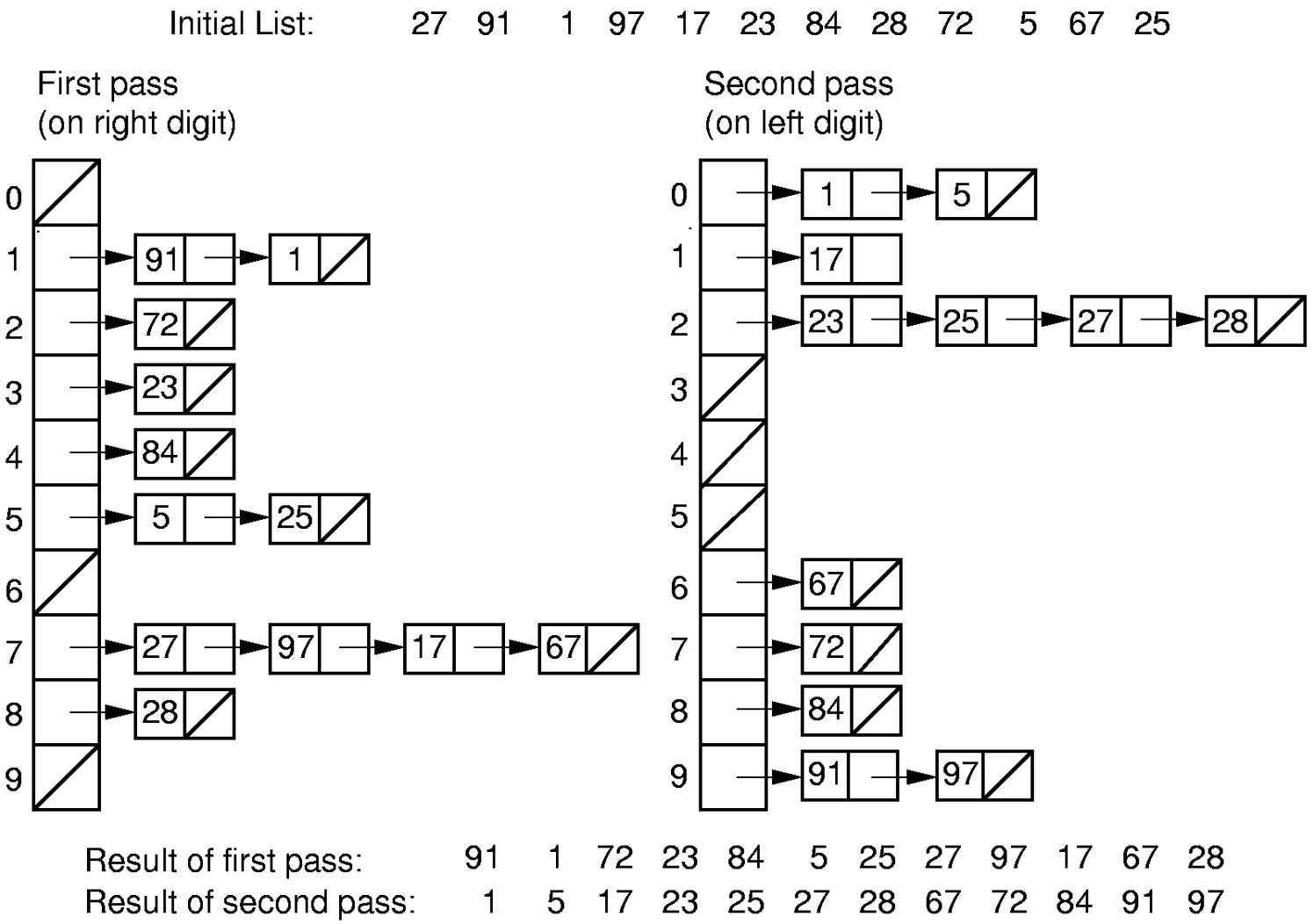
- Os métodos não utilizam comparação entre chaves.

- Uma das dificuldades de implementar estes métodos está relacionada com o problema de lidar com cada monte.

- Se para cada monte nós reservarmos uma área, então a demanda por memória extra pode tornar-se proibitiva.

- O custo para ordenar um arquivo com n elementos é da ordem de O(n), isto é, linear.

RadixSort


Initial List: 27 91 1 97 17 23 84 28 72 5 67 25
First pass
(on right digit)
Second pass
(on left digit)
0
1
2
3
4
5
6
7
8
9
91
1
72
23
84
5
25
27
97
17
67
28
1
5
17
23
25
27
28
67
72
84
91
97
Result of first pass: 91 1 72 23 84 5 25 27 97 17 67 28
Result of second pass: 1 5 17 23 25 27 28 67 72 84 91 97

Exemplo 2: Máximo de um conjunto A[1..n], n>= 1

```
Function Max (var A: vetor): integer;
Var i, temp: integer;
Begin
    temp:=A[1];
    for i:= 2 to n do
        if A[i] > temp then temp:= A[i];
    Max:= temp
End.
```

Seja f a função de complexidade tal que f(n) é o número de comparações entre os elementos de A se A tiver n elementos:

f(1) = 0

f(n) = n-1, n > 1

MAX é ótimo?

Teorema 2: Qualquer algoritmo para encontrar o maior elemento de um conjunto com n elementos n >= 1, faz pelo menos n-1 comparações

Prova: Cada um dos n-1 elementos tem que ser mostrado, através de comparações, que é menor do que algum outro elemento. Logo n-1 comparações são necessárias.

Algoritmos Ótimos

- Se dispusermos de um algoritmo cujo custo é igual ao limite inferior da classe, então teremos um algoritmo ótimo para a tarefa,
  - no sentido de que a quantidade de recursos utilizada por qualquer outro algoritmo para a tarefa será maior, ou no melhor caso igual à do algoritmo que temos.

- Vejam que o limite inferior da classe é provado teoricamente, assim, em alguns casos MUITO trabalho ainda precisa ser feito antes que se consiga tal algoritmo.
  - Exemplo: multiplicação de matrizes. Método convencional $O(n^3)$; existe o método de Strassen com $O(n^{2.81})$, um ainda mais rápido com $O(n^{2.5})$ e teoricamente o limite inferior é $\Omega(n^{2+\xi})$.

- Muitas vezes, na prática, algoritmos não-ótimos são mais rápidos, mais claros e até consomem menos memória!

Exemplo 3: Pesquisa Seqüencial

Definição do Problema: Cada registro contém uma chave única que é utilizada para recuperar registros do arquivo. Dada uma chave qualquer o problema consiste em localizar o registro que contenha a chave.

Solução do problema: O algoritmo de pesquisa mais simples que existe é o que faz uma pesquisa seqüencial.

Seja f uma função de complexidade tal que f(n) é o número de vezes que a chave de consulta é comparada com a chave de cada registro.

Melhor Caso:

o registro procurado é o primeiro consultado.

f(n) = 1

Pior Caso:

o registro procurado é o último consultado, ou então não está presente no arquivo.

f(n) = n

Caso Médio: considerando que toda pesquisa recupera um registro e sendo pi a probabilidade de que o i-ésimo registro seja procurado e para recuperá-lo são necessárias i comparações: $f(n) = 1.p1 + 2.p2 + ...+ n.pn$

Se cada registro tiver a mesma probabilidade de ser acessado, então

$pi = 1/n$, $1 <= i <= n$. $f(n) = 1/n(1 + 2 + 3 + ... + n) = 1/n\ (1/2\ n.(n+1)) = (n+1)/2$

$\sum_{i=1}^{n} i$ (série aritmética)

Melhor Caso, Pior Caso,Caso Médio

- Melhor Caso

  Corresponde ao menor tempo de execução sobre todas as possíveis entradas de tamanho n.

  (Se n = 3, 123, 132, ..., 321)

- Pior Caso:

  Corresponde ao maior tempo de execução sobre todas as entradas de tamanho n.

- Caso Médio (ou caso esperado):

  Corresponde à média dos tempos de execução de todas as entradas de tamanho n.

  - Nesta análise, uma distribuição de probabilidades sobre o conjunto de entradas de tamanho n é suposta, e o custo médio é obtido com base nesta distribuição.
  - É mais difícil de se obter do que as outras duas análises.

Diferenças entre Cálculo do Máximo e a Pesquisa Seqüencial

- O Algoritmo para encontrar o maior elemento de um conjunto possui custo uniforme sobre todos os tamanhos de entrada n.

- O Algoritmo da pesquisa seqüencial depende da ordem em que os registros aparecem.
  - Assim, fazemos o estudo do pior caso (custo máximo) e o do caso médio (custo médio), assumindo uma distribuição da entrada.

Link importante- Curso do Prof. Nivio Ziviani da UFMG

- http://www.dcc.ufmg.br/~nivio/cursos/pa02/pa02apr.html

- **Projeto de Algoritmos com Implementações em Pascal e C Nivio Ziviani:** http://www2.dcc.ufmg.br/livros/algoritmos/