CI1056: Algoritmos e Estruturas de Dados II

Prof. Dr. Marcos Castilho

Departamento de Informática/UFPR

8 de dezembro de 2020

Resumo

Um panorama sobre algoritmos de ordenação (parte 3)

Objetivos da aula

- Discutir algoritmos de ordenação de forma geral (parte 3)

Variante do *quicksort*

- Mais conhecido como `3-way-quicksort`
- Tenta melhorar o caso em que tem muitos elementos repetidos
- Divide em três partes:
  - Os menores que o pivô
  - Os iguais ao pivô
  - Os maiores que o pivô
- Depois recursivamente ignora os que são iguais

Outras variantes do *quicksort*

- O pior caso do `quicksort` é quando os elementos chegam ordenados
- É um caso raro, mas existem diversos estudos sobre como minimizar o problema:
  - Escolha do pivô: média de três, média
    - Existem estudos que indicam que escolhendo a média dos elementos o pior caso é $n.log_2(n)$, confiram!!!
  - Quando a recursão chega em um subvetor com tamanho "pequeno", use o `insertionsort`
  - Várias outras, todas documentadas na literatura
  - Uma delas inclui eliminar a recursão usando pilhas, veja...
  - Vejam a literatura!

Heapsort

- O `heapsort` pode ser considerado uma variante do `selectionsort`, pois a cada iteração tem que se saber o maior
- Em um *heap* de máximo, o maior está na primeira posição
- Na troca, deve-se garantir a invariante de laço e para isto se chama o `max_heapify`

Enfim

- Existem realmente muitos algoritmos de ordenação
- Saber escolher o melhor pode depender de alguns outros critérios

Estabilidade

- Um algoritmo de ordenação *estável* é aquele que mantém a ordem relativa entre dois elementos iguais
- Se o vetor de entrada tiver duas ou mais chaves iguais, então a saída deve manter estes elementos na mesma posição relativa

Exemplo:

	1	2	3	4	5	6	7	8	9	10
v	5	10	15	20	25	35	15	45	70	90

- Observem os dois números 15 repetidos, o primero vermelho e o segundo azul
- Se o algoritmo for estável, o 15 vermelho deve estar antes do 15 azul na ordeanção final, preservando a ordem relativa entre eles
- Isto pode ser extremamente importante em algumas aplicações
- Vamos ver um exemplo?

Impacto da não estabilidade

- O problema é melhor percebido quando se ordenam registros
- Por exemplo, considere registros contendo nomes de cidades e *timestamps*, ordenados por *timestamp* (adaptado de Sedwick)

Curitiba	10:00
Recife	10:01
Maceió	10:02
Curitiba	10:03
Maceió	10:04

Impacto da não estabilidade

- Quando se ordena por nome de cidade em um algoritmo não estável, pode-se perder a ordenação por *timestamp*

Curitiba	10:03
Curitiba	10:00
Maceió	10:04
Maceió	10:02
Recife	10:01

Impacto da estabilidade

- Quando se ordena por nome de cidade em um algoritmo estável, ainda temos a ordenação por *timestamp* relativa às cidades

Curitiba	10:00
Curitiba	10:03
Maceió	10:02
Maceió	10:04
Recife	10:01

Algoritmos estáveis e não estáveis

- Estáveis
  - insertion, mergesort
- Não estáveis
  - selection, shellsort, quicksort, heapsort
- Alguns estudos procuram tornar um método não estável em um estável
- Porém com grande custo
- O melhor, se precisar de estabilidade, use um método naturalmente estável

Uso de memória

- O `mergesort` usa um vetor auxiliar
- Se o vetor for "grande", existe memória suficiente?
- Dos algoritmos estudados o `mergesort` é o único que usa um vetor auxiliar!
- Isto é relevante na sua aplicação?

Melhor caso, caso médio e pior caso

- O melhor caso ocorre sempre? Creio que não.
- O pior caso ocorre sempre? Há que se pensar sobre isso
- Aliás, isso dá origem aos algoritmos derivados
- E o caso médio?
- O estudo teórico disponível para estudo serve para você?
- Por exemplo, o `quicksort` tem um pior caso quadrático, este caso é frequente?
- Existem estudos sobre isso!!!

Custo da troca

- O custo da troca de 2 elementos não foi estudado em sala de aula, mas existe
- Por exemplo, o `selectionsort` é, provavelmente o que faz menos trocas ($n - 1$ trocas exatamente)
- O custo das trocas pode ser eliminado usando-se ponteiros para os conteúdos, estude!
- A maior parte dos estudos não considera o custo das trocas por este motivo
- Porém, para conjuntos de tipos básicos, não vale o custo!

Hipótese

- Existe uma hipótese não provada cientificamente
  - O `quicksort` é o mais eficiente de todos!
- Não existe prova disso
- Mas, na prática, é verdade (cuidado com a estabilidade)

Qual o motivo?

- O `quicksort` tem um algoritmo dos mais eficientes pois:
  - o laço interno tem poucas instruções
  - O *cache* ajuda, pois ele opera sobre índices consecutivos sequencialmente
  - O `3-way-quicksort` pode até se tornar linear com algumas distribuições de "chaves" comuns na prática

Qual é o melhor?

- Depende! (a melhor resposta da ciência da computação!)
- Depende dos dados, do hardware, do sistema operacional, da linguagem, do compilador e da maneira como foi implementado
- Mas, francamente, na prática, o `quicksort` é na maior parte dos casos o melhor de todos até hoje
- A menos que você dependa de estabilidade...
- Neste caso, estude, mas talvez você queira usar o `mergesort`

Você prefere n^2 ou $n.log_2(n)$

- Existem alguns estudos que apontam que, se o conjunto for "pequeno" então o `insertionsort` tem melhor desempenho do que qualquer outro
- Existe inclusive, uma implementação do `quicksort` que, nas chamadas recursivas, quando chega neste tal tamanho "pequeno", ao invéz de chamar a recursão, chamar o `insertionsort`

Alguns resultados conhecidos empiricamente

- `heapsort` computa o dobro que o `mergesort`
- `heapsort` e `mergesort` acessam o vetor muito mais do que o `quicksort`
- Mas, como dito, depende do hardware, SO, etc. . .

Enfim, qual é o melhor?

- Provavelmente, você vai usar o `quicksort`
- Caso precise de estabilidade, talvez vá preferir o `mergesort`
- Mas talvez você queira fazer um estudo *empírico* para o teu caso particular e talvez possa concluir que:
  - A quantidade de dados que eu tenho é pequena e o `insertionsort` ou `shellsort` são melhores
  - Eu ordeno durante a noite, não faz diferença, então eu vou implementar o mais fácil de dar manutenção...
  - Outras justificativas...

Conclusões

- Para conjuntos quaisquer, veremos que um algoritmo de ordenação não pode ser melhor do que $n.log(n)$
- Para situações absolutamente particulares, veremos que a ordenação pode ser feita em tempo linear
- Na dúvida? Seu problema é tempo de execução?
  - Use alguma variante (ou o clássico) `quicksort`
- Precisa de algo especial?
  - Talvez o `mergesort`, mas *ESTUDE* antes!

Próximas aulas

- Veremos alguns algoritmos que lidam bem com conjuntos de dados bem especiais, aqueles que têm alguma propriedade adicional além de "um vetor qualquer"

Fim do tópico

- O conteúdo desta aula está basicamente no livro do Sedgewick, capítulo 2 (base) e em vários outros. . . . Também está espalhado por diversos outros livros, tais quais o do Cormem, na "biblia" do Knuth, ou até mesmo em alguns sites confiáveis da Internet (quando se tratar de implementações). . .

Licença

- Slides feitos em LaTeX usando beamer
- Licença

  Creative Commons Atribuição-Uso Não-Comercial-Vedada a Criação de Obras Derivadas 2.5 Brasil License.`http://creativecommons.org/licenses/by-nc-nd/2.5/br/`

  Creative Commons Atribuição-Uso Não-Comercial-Vedada a Criação de Obras Derivadas 2.5 Brasil License.`http://creativecommons.org/licenses/by-nc-nd/2.5/br/`