Ordenação

Prof. Rafael Alves Bonfim de Queiroz
rafael.queiroz@ufop.edu.br

Conteúdo

Introdução

- A ordenação é o problema algorítmico mais fundamental em ciência da computação
- A maioria dos paradigmas de design de algoritmos leva a algoritmos de ordenação interessantes, incluindo divisão e conquista, randomização, inserção incremental e estruturas de dados avançadas
- Muitos problemas de programação/matemáticos interessantes surgem das propriedades desses algoritmos
- Vamos rever as principais aplicações da ordenação, bem como a teoria por trás dos algoritmos mais importantes
- Descreveremos as rotinas da biblioteca de ordenação fornecidas por linguagens de programação modernas

Aplicações de Ordenação

A chave para entender a ordenação é ver como ela pode ser usada para resolver muitas importantes tarefas de programação:

- Teste de unicidade
  - Como podemos testar se os elementos de uma dada coleção de itens S são todos distintos?
  - Ordene-os em ordem crescente ou decrescente para que quaisquer itens repetidos caiam um ao lado do outro
  - Uma passagem pelos elementos testando se $S[i] == S[i+1]$ para qualquer $1 \leq i < n$ então terminará o trabalho
- Deletando duplicatas
  - Como podemos remover todas cópias de quaisquer elementos repetidos em S?
  - Ordenar e varrer novamente faz o trabalho
  - Note que a varredura é melhor feito mantendo dois índices - *back*, apontando para o último elemento no vetor de prefixo limpo, e i, apontando para o próximo elemento a ser considerado
  - Se $S[back] <> S[i]$, incremente *back* e copie $S[i]$ para $S[back]$

- Priorizando Eventos
  - Suponha que recebemos um conjunto de tarefas a fazer, cada uma com seu próprio prazo
  - Ordenar os itens de acordo com a data limite (ou algum critério relacionado) coloca os trabalhos na ordem certa para processá-los
  - Estruturas de dados de fila de prioridade são úteis para manter calendários ou agendas quando há inserções e exclusões, mas a ordenação faz o trabalho se o conjunto de eventos não mudar durante a execução
- Mediana/Seleção
  - Suponha que queremos encontrar o k-ésimo maior item no conjunto S
  - Após ordenar os itens em ordem crescente, este sujeito estará em $S[k]$
  - Essa abordagem pode ser usada para encontrar (de uma maneira um pouco ineficiente) o menor, maior, e elementos medianos como casos especiais
- Contagem de Frequência
  - Qual é o elemento que ocorre com mais frequência em S, ou seja, a moda?
  - Após a ordenação, uma varredura linear nos permite contar o número de vezes que cada elemento ocorre

- Reconstruindo o Pedido Original
  - Como podemos restaurar o arranjo original de um conjunto de itens depois de permutá-los para alguma aplicação?
  - Adiciona-se um campo extra para o registro de dados do item, de modo que o i-ésimo registro defina esse campo como i
  - Carrega-se este campo sempre que você mover o registro e depois classifique-o quando quiser o pedido inicial de volta
- Definir Intersecção/União
  - Como podemos cruzar ou unir os elementos de dois conjuntos?
  - Se ambos foram ordenados, podemos mesclá-los repetidamente pegando o menor dos dois elementos de cabeça, colocando-os no novo conjunto se desejado e, em seguida, excluindo a cabeça da lista apropriada

- Encontrando um par alvo
  - Como podemos testar se existem dois inteiros $x, y \in S$ tal que $x + y = z$ para algum alvo z?
  - Em vez de testar todos os pares possíveis, classifique os números em ordem crescente e varredura
  - À medida que $S[i]$ aumenta com i, seu possível parceiro j tal que $S[j] = z - S[i]$ diminui
  - Assim, diminuir j apropriadamente à medida que i aumenta dá uma boa solução
- Pesquisa Eficiente
  - Como podemos testar eficientemente se o elemento s está no conjunto S?
  - Ordenar um conjunto de modo a permitir buscas binárias eficientes talvez seja o aplicação mais comum de ordenação

Algoritmos de Ordenação

- Você provavelmente já viu uma dúzia ou mais de algoritmos diferentes para ordenar dados
- Bubblesort, insert sort, selection sort, heapsort, mergesort, quicksort, radix sort, Shell sort, percurso de árvore, redes de ordenação....
- A verdadeira razão para estudar algoritmos de ordenação é que as ideias por trás deles reaparecem como as ideias por trás dos algoritmos para muitos outros problemas
- Heapsort é realmente sobre estruturas de dados
- Quicksort está associado à randomização
- Mergesort é realmente sobre dividir e conquistar

Ordenação por Seleção

- Este algoritmo divide o vetor de entrada em partes ordenada e não ordenada, e com cada iteração encontra o menor elemento restante na região não ordenada e move ele para o final da região sorteada

```
selection_sort(int s[], int n)
{
        int i,j;                        /* counters */
        int min;                        /* index of minimum */

        for (i=0; i<n; i++) {
                min=i;
                for (j=i+1; j<n; j++)
                        if (s[j] < s[min]) min=j;
                swap(&s[i],&s[min]);
        }
}
```

Ordenação por Seleção

- A ordenação por seleção faz muitas comparações, mas é bastante eficiente se tudo o que contarmos são o número de movimentos de dados
- Apenas $n - 1$ trocas são executadas pelo algoritmo, o que é necessário no pior caso
- Ele também fornece um exemplo do poder das estruturas de dados avançadas
- Usando um fila de prioridade eficiente para manter a parte não ordenada do vetor transformará a ordenação por seleção $O(n^2)$ em heapsort $O(nlgn)$

Ordenação por Inserção

- Este algoritmo também mantém regiões ordenadas e não ordenadas do vetor
- Em cada iteração, o próximo elemento não ordenado move-se para a sua posição na região ordenada

```
insertion_sort(int s[], int n)
{
        int i,j;                        /* counters */

        for (i=1; i<n; i++) {
                j=i;
                while ((j>0) && (s[j] < s[j-1])) {
                        swap(&s[j],&s[j-1]);
                        j = j-1;
                }
        }
}
```

Ordenação por Inserção

- A ordenação por inserção é particularmente importante como o algoritmo que minimiza a quantidade de movimentação de dados
- Uma inversão em uma permutação p é um par de elementos que estão fora de ordem, ou seja, um i, j tal que $i < j$ ainda $p[i] > p[j]$
- Cada troca na ordenação por inserção apaga exatamente uma inversão, e nenhum elemento é movido de outra forma, então o número de trocas é igual ao número de inversões
- Como uma permutação quase ordenada tem poucas inversões, a ordenação por inserção pode ser muito eficaz nesses dados

Quicksort

- Este algoritmo reduz o trabalho de ordenar um grande vetor para o trabalho de ordenar dois vetores menores executando uma etapa de partição
- A partição separa o vetor naqueles elementos que são menores que o elemento pivô/divisor, e aqueles que são estritamente maiores que este elemento pivô/divisor
- Como nenhum elemento precisa sair de sua região após a partição, cada subvetor pode ser classificado independentemente
- Para facilitar a ordenação de subvetores, os argumentos para quicksort incluem os índices do primeiro (l) e último (h) elementos do subvetor

```
quicksort(int s[], int l, int h)
{
        int p;                  /* index of partition */

        if ((h-l)>0) {
                p = partition(s,l,h);
                quicksort(s,l,p-1);
                quicksort(s,p+1,h);
        }
}

int partition(int s[], int l, int h)
{
        int i;                  /* counter */
        int p;                  /* pivot element index */
        int firsthigh;          /* divider position for pivot */

        p = h;
        firsthigh = l;
        for (i=l; i<h; i++)
                if (s[i] < s[p]) {
                        swap(&s[i],&s[firsthigh]);
                        firsthigh ++;
                }
        swap(&s[p],&s[firsthigh]);

        return(firsthigh);
}
```

Quicksort

- Quicksort é interessante por vários motivos
- Quando implementado corretamente, é o algoritmo de ordenação na memória mais rápido
- É um belo exemplo do poder da recursão
- O algoritmo de partição é útil para muitas tarefas por si só
  - Por exemplo, como você pode separar um vetor contendo apenas 0's e 1's em um corrida de cada símbolo?

Estabilidade de algoritmo de ordenação

- O algoritmo de ordenação é estável quanda preserva a ordem relativa de chaves iguais
- As funções de ordenação por inserção e seleção são estáveis, enquanto o quicksort não é estável.

Ordenando e Pesquisando em C

- Sempre que possível, aproveite as bibliotecas de ordenação/pesquisa embutidas em sua linguagem de programação favorita
- O **stdlib.h** contém funções de biblioteca para ordenação e busca.
- Para ordenação, existe a função **qsort**:

```
#include <stdlib.h>

void qsort(void *base, size_t nel, size_t width,
           int (*compare) (const void *, const void *));
```

- A chave para usar o **qsort** é perceber o que seus argumentos fazem.
  - Ele classifica os primeiros elementos **nel** de um vetor (apontado por **base**), onde cada elemento tem **width**-bytes.
  - Assim, podemos ordenar vetores de caracteres de 1 byte, inteiros de 4 bytes ou registros de 100 bytes, tudo por mudar o valor de **width**.

- A ordem final desejada é determinada pela função **intcompare**
- Recebe como argumentos ponteiros para dois elementos de byte de largura e retorna um número negativo se o primeiro pertencer antes do segundo em ordem de ordenação, um número positivo se o segundo pertencer antes do primeiro, ou zero se forem iguais

```
int intcompare(int *i, int *j)
{
    if (*i > *j) return (1);
    if (*i < *j) return (-1);

    return (0);
}
```

- Esta função de comparação pode ser usada para ordenar um vetor *a*, do qual os primeiros *cnt* elementos estão ocupados

```
qsort((char *) a, cnt, sizeof(int), intcompare);
```

- O nome **qsort** sugere que quicksort é o algoritmo implementado nesta função de biblioteca
- Note que qsort destrói o conteúdo do vetor original, então se você precisar restaurar a ordem original, faça uma cópia ou adicione um campo extra ao registro

Pesquisa binária

- A biblioteca **stdlib.h** contém uma implementação chamado **bsearch**(), pesquisa binária.
- Exceto pela chave de pesquisa, os argumentos são os mesmos do **qsort**
- Para pesquisar no vetor ordenado anteriormente, tente

```
bsearch(key, (char *) a, cnt, sizeof(int), intcompare);
```

Classificando e Pesquisando em C++

- A biblioteca de Modelos Padrão C++ (STL) inclui métodos para classificação, pesquisa e muito mais
- Para classificar com STL, podemos usar a função de comparação padrão definida para a classe ou substitua-a por uma função de comparação de propósito especial op:

```
void sort(RandomAccessIterator bg, RandomAccessIterator end)
void sort(RandomAccessIterator bg, RandomAccessIterator end,
          BinaryPredicate op)
```

Ordenando e Pesquisando em C++

- STL também fornece uma rotina de classificação estável, onde as chaves de igual valor são garantidas permanecer na mesma ordem relativa
- Isso pode ser útil se estivermos ordenando por vários critério:

```
void stable_sort(RandomAccessIterator bg, RandomAccessIterator end)
void stable_sort(RandomAccessIterator bg, RandomAccessIterator end,
          BinaryPredicate op)
```

- Outras funções STL implementam algumas das aplicações de ordenação descritas
  - enésimo elemento: retorna o enésimo maior item do conjunto
  - conjuntos união, interseção e diferença
  - unique: remove todas as duplicatas consecutivas

Ordenando e Pesquisando em Java

- A classe **java.util.Arrays** contém vários métodos para ordenar e pesquisar.

```
static void sort(Object[] a)
static void sort(Object[] a, Comparator c)
```

- Ordena o vetor especificado de objetos em ordem crescente usando a ordenação natural de seus elementos ou um comparador específico *c*
- Ordenações estáveis também estão disponíveis
- Métodos para pesquisar um vetor ordenado por um objeto especificado usando a função de comparação natural ou um novo comparador *c* também são fornecidos:

```
binarySearch(Object[] a, Object key)
binarySearch(Object[] a, Object key, Comparator c)
```