MC-102 — Aula 17
Busca e ordenação Recursiva

Instituto de Computação – Unicamp

Segundo Semestre de 2007

Roteiro

1. Hanoi
2. Busca binária recursiva
3. Intercalação de seqüências
4. MergeSort
5. QuickSort

Torres de Hanói

```
void move_hanoi (int n, int origem, int destino,
                 int auxiliar) {
  if (n == 0) return;
  if (n == 1) {
    move_disco(origem, destino);
    return;
  }
  move_hanoi(n-1, origem, auxiliar, destino);
  move_disco(origem, destino);
  move_hanoi(n-1, auxiliar, destino, origem);
}
```

Torres de Hanói

Seja $T(n)$ o número de movimentos necessários para resolver o problema com n discos.
Pelo algoritmo anterior, $T(n)$ é dado pela seguinte recursão:

$$T(n) = \begin{cases} 1 & \text{se } n = 1 \\ 2T(n-1) + 1k & \text{se } n \geq 1 \end{cases}$$

Fazendo alguns cálculos, chegamos à solução seguinte:
$T(n) = 2^n - 1$, para $n \geq 1$.

Torres de Hanói

Quanto tempo para resolver o problema com 64 discos?
Suposições:

- Sacerdotes são semi-deuses que movem os discos na velocidade da luz (300.000.000 m/s) (lembre-se: só Chuck Norris consegue mover objetos mais rápido que a velocidade da luz.).
- 3 torres a 66.33 cm uma da outra. Distância média dos movimentos: $\frac{3*0.6633}{2} = 0.99495 \approx 0.995$.
- Discos com altura mínima, tendendo a zero $\Rightarrow$ tamanho da pilha tende a zero $\Rightarrow$ Tempo para tirar o disco da pilha tende a zero.

Busca binária

- Caso uma lista A esteja ordenada, sabemos que, para qualquer i e j, $i < j$ se, e somente se $A[i] \leq A[j]$.
- Portanto, comparando um elemento com o elemento procurado, saberemos:
  - se o elemento procurado é o elemento comparado, ou
  - se ele está antes do elemento comparado ou
  - se ele está depois.

Busca binária

- Se fizermos isso sempre com o elemento do meio da lista, a cada compara¸, dividiremos a lista em duas, reduzindo nosso espaço de busca.
- Se em um determinado momento o vetor, após sucessivas divisões, tiver tamanho zero, então o elemento não está no vetor.

Veja detalhes em `busca-bin.c`

Intercalando duas seqüências ordenadas

Problema

Dado duas seqüências ordenadas, como criar uma terceira seqüência, com todos os elementos das duas seqüências, tal que a terceira seqüência esteja também ordenada?

Intercalando duas seqüências ordenadas

Solução:

Copie as duas seqüências para um único vetor e ordene-as

Intercalando duas seqüências ordenadas

- Embora essa solução seja válida, ela não é a mais eficiente, pois não aproveita o fato que a seqüência está ordenada.
- Uma outra abordagem seria tentar encontrar um elemento que seja o menor possível entre todos os elementos existentes na seqüência. Então, movemos esse elemento de sua posição original para a nova seqüência.

Intercalando duas seqüências ordenadas

- Se fizermos isso para determinarmos cada uma das posições da nova seqüência, teríamos uma seqüência ordenada, uma vez que colocamos sempre o menor elemento possível para aquela posição
- Como as seqüências estão ordenadas, é muito simples encontrar o menor elemento dentre todos os elementos. Ele deve ser, obrigatoriamente, um dentre os menores elementos de cada seqüência (pois se isso não fosse verdade, uma das seqüências não estaria ordenada).

Intercalando duas seqüências ordenadas

- Basta então comparar esses dois elementos e escolhermos o menor. Ele é, garantidamente, o menor elemento dentre todos os elementos existentes nas duas seqüências

Intercalando duas seqüências ordenadas

Versão final da intercalação:

Como fazer isso se as seqüências não ordenadas estão no mesmo vetor, em posições consecutivas (ou seja, uma delas começa na posição *inicio* e vai até a posição *meio* e a outra, começa na posição $meio + 1$ e vai até a posição *fim*?

Veja detalhes em `intercala.c`

MergeSort

- Uma técnica para resolver um problema é dividi-lo em parte menores, para as quais saibamos resolver o problema, e a partir desses resultados, resolver o problema maior. Essa técnica é chamada de divisão e conquista.
- A idéia do MergeSort é dividir a ordenação em duas ordenações menores (divisão) e, a partir dessas listas ordenadas, utilizar a intercalação para reunir as duas listas

MergeSort

- Ora, ordenação é exatamente o que queremos fazer!!! Podemos utilizar uma chamada recursiva (divide-se a lista em duas até que consigamos um caso bem simples onde saibamos resolver e vamos conquistando para montar listas cada vez maiores até que chegaremos na lista total ordenada).
- Agora, precisamos de um caso simples. Trivialmente, ordenar uma lista de um elemento significa não fazer nada (uma lista de um elemento está sempre ordenada).

Veja detalhes em `mergesort.c`

MergeSort

- O algoritmo *mergesort* executa $n \log n$ comparações, que é o menor número possível de ordenações para algoritmos sem qualquer tipo de restrição e baseados em comparações. (nenhum dos dois fatos da sentença acima pode ser demonstrado de forma trivial).
- Entretanto, ele necessita de um espaço de memória extra proporcional ao tamanho do vetor para fazer a intercalação.

Quicksort

- Novamente dividiremos o problema de ordenação em problemas menores, até encontrarmos um caso que possa ser resolvido trivialmente.
- Escolhemos um elemento do vetor arbitrariamente (chamado de pivô) e dividimos a lista em duas listas menores: uma contendo os elementos menores que o pivô e outra com os elementos maiores que o pivô. Uma posição intermediária ficará disponível para o próprio pivô.

Quicksort

- Após a divisão da lista, ordenamos as duas sublistas geradas recursivamente.
- O caso base novamente é a ordenação de listas com um elemento ou vazias, que já estão ordenadas por definição.

Veja detalhes em `quicksort.c`

Quicksort

- O algoritmo *quicksort* executa $n \log n$ comparações no caso médio, mas pode executar n^2 operações no pior caso.
- Esse pior caso pode ser evitado facilmente, por exemplo, escolhendo os pivôs aleatoriamente. Existem outras técnicas que garantidamente evitam o caso n^2 (por exemplo, escolher sempre a mediana do vetor como pivô).

Quicksort

- O *quicksort* é o algoritmo de ordenação mais rápido pois, além de executar o mesmo número de comparações que o *mergesort*, utiliza menos operações durante a fase de conquista.
- Além disso, o quicksort não necessita de um espaço extra de armazenamento (as operações são realizadas no próprio vetor a ser ordenado).