SCC0220 - Laboratório Introdução à Ciência da Computação II

Relatório de execução da aula prática 5

Alunos	**NUSP**
Juan Henriques Passos	15464826
Alec Campos Aoki	15436800

Trabalho 5 - Bubble and Quick

Bubblesort

Comentário

O algoritmo *bubblesort* foi utilizado para ordenar os itens de um cardápio em ordem crescente de prioridade (se a prioridade entre dois itens forem a mesma, o de menor tempo de preparo é priorizado). Esse algoritmo funciona percorrendo o vetor *n* vezes e trocando valores de posição se eles estiverem na ordem errada. Analisando seu código, temos dois laços *for* alinhados, gerando uma complexidade de $O(n^2)$ no pior caso (quando o vetor está inversamente ordenado). No melhor caso, temos $O(n)$, pois guardamos a quantidade de trocas realizadas em cada passagem completada do vetor. Se nenhuma troca foi feita, sabemos que o vetor já está ordenado, então finalizamos o algoritmo. Note também que, no *bubblesort*, a cada iteração temos que o último elemento do vetor já está em sua posição correta. Na próxima iteração, portanto, checamos somente *n-1* elementos, e na próxima *n-2*, e assim por diante.

Código

```
//Retorna true se prato1 > prato2 (prioridade, tempo de preparo)
bool comparar_pratos(PRATO prato1, PRATO prato2){
   if(prato1.prioridade > prato2.prioridade) return true;
   else if(prato1.prioridade == prato2.prioridade){
       if(prato1.tempoPreparo < prato2.tempoPreparo) return true;
   }

   return false;
}

void bubblesort(PRATO *pontVetPratos, int tam){
   for(int i=0; i<tam; i++){
       int trocas = 0;
```

```
        for(int j=1; j<tam-i; j++){
            if(comparar_pratos(pontVetPratos[j-1], pontVetPratos[j])){
                PRATO aux = pontVetPratos[j-1];
                pontVetPratos[j-1] = pontVetPratos[j];
                pontVetPratos[j] = aux;

                trocas++;
            }
        }

        if(trocas == 0) return;
    }

    return;
}
```

Saída

```
→  Aula05 git:(main) ✗ make run < Casos\ de\ Teste\ -\ Trabalho\ 5\ -\ Bubble\ and\ Quick/7.in
./main

Tempo de execucao: 0.728013ms
→  Aula05 git:(main) ✗ make run < Casos\ de\ Teste\ -\ Trabalho\ 5\ -\ Bubble\ and\ Quick/7.in
./main

Tempo de execucao: 0.731071ms
→  Aula05 git:(main) ✗ make run < Casos\ de\ Teste\ -\ Trabalho\ 5\ -\ Bubble\ and\ Quick/7.in
./main

Tempo de execucao: 0.738778ms
→  Aula05 git:(main) ✗
```

Quicksort

Comentário

Utilizamos o algoritmo *quicksort* para resolver o mesmo problema. Ele funciona selecionando um pivô no vetor (arbitrariamente) e manipulando seus elementos de forma que os elementos à esquerda do pivô sejam todos menores que ele (não necessariamente ordenados), e os elementos à direita sejam todos maiores que ele (idem). Fazemos isso colocando o pivô na última posição do vetor, e então utilizando dois índice *i* e *j* para percorrer o vetor. Vamos avançar o índice *j* (que começa no início do vetor) até que o valor indicado por ele no vetor seja menor que o pivô; avançamos então o índice *i* (que começa na posição *início do vetor -1)* e trocamos os valores indicados por eles Repetimos esse processo até que *j* chegue no fim do pivô; no fim, avançamos *i* mais uma posição e o trocamos de lugar com o pivô. Agora temos que todos os elementos à esquerda do pivô são menores que ele, os elementos à direita são todos maiores que ele e o pivô está em sua posição correta. Dividimos então o vetor em dois, do início à posição do pivô *-1* e da posição do pivô *+1* até o fim. Caso o vetor esteja ordenado inversamente, e selecionarmos o pivô como a última posição (pior caso), teremos que percorrer o vetor n^2 vezes e o dividiremos em partes de tamanhos diferentes. Temos, portanto, que a complexidade do *quicksort* é $O(n^2)$. Podemos evitar esse pior caso selecionando o pivô aleatoriamente, ou então selecionando três valores e escolhendo como pivô a mediana entre eles (esses três valores podem ser aleatórios ou o início, meio e fim do vetor). Fazendo isso, dividimos o vetor em duas metade de tamanhos iguais e temos uma complexidade de $O(nlog(n))$ (caso médio do algoritmo). Outra alternativa é selecionar o pivô como o meio do vetor. Note que isso não necessariamente evita o pior caso, pois o elemento no meio do vetor pode ser o maior valor do vetor. De qualquer forma, temos que o caso médio do *quicksort* tem menor taxa de crescimento que o melhor caso do *bubblesort*, de forma que o primeiro será mais eficiente que o último para um *n* muito grande e em seus casos médios. Logo, o *quicksort* é mais eficiente na ordenação dos pratos, como mostrados pelos tempos de saída dos códigos.

Código

```
void quicksort(PRATO *pontVetPratos, int inicio, int fim){
   if(fim <= inicio) return; //caso base

   int pivot = (inicio + fim)/2;
   PRATO aux = pontVetPratos[fim];
   pontVetPratos[fim] = pontVetPratos[pivot];
   pontVetPratos[pivot] = aux;
   pivot = fim;

   int i = inicio-1;
   int j = inicio;
```

```
    while(j < fim){
        if(comparar_pratos(pontVetPratos[fim], pontVetPratos[j])){
            i++;
            aux = pontVetPratos[i];
            pontVetPratos[i] = pontVetPratos[j];
            pontVetPratos[j] = aux;
        }

        j++;
    }

    i++;
    aux = pontVetPratos[pivot];
    pontVetPratos[pivot] = pontVetPratos[i];
    pontVetPratos[i] = aux;

    quicksort(pontVetPratos, inicio, i-1);
    quicksort(pontVetPratos, i+1, fim);

    return;
}
```

Saída

```
→  Aula05 git:(main) ✗ make run < Casos\ de\ Teste\ -\ Trabalho\ 5\ -\ Bubble\ and\ Quick/7.in
./main

Tempo de execucao: 0.002681ms
→  Aula05 git:(main) ✗ make run < Casos\ de\ Teste\ -\ Trabalho\ 5\ -\ Bubble\ and\ Quick/7.in
./main

Tempo de execucao: 0.002520ms
→  Aula05 git:(main) ✗ make run < Casos\ de\ Teste\ -\ Trabalho\ 5\ -\ Bubble\ and\ Quick/7.in
./main

Tempo de execucao: 0.002730ms
→  Aula05 git:(main) ✗
```