Ordenação em vetores

1. Introdução

Vimos dois algoritmos para buscas em vetores, a busca sequencial, que pode ser usada em qualquer vetor, independende da ordem dos elementos, e a busca binária, que é muito mais rápida, mas exige que os elementos do vetor estejam em ordem.

Mas nem sempre os elementos são inseridos em ordem em um vetor. Assim, pode ser necessário implementar algum algoritmo de ordenação.

1.1. Classificação

Existem vários algoritmos de ordenação, que podem ser classificados:

Quanto à eficiência

- **Métodos simples.** O algoritmo é mais fácil de entender mas com maior custo computacional. Exemplos: Bubble Sort, Insertion Sort, Selection Sort.
- **Métodos eficientes.** Em geral o algoritmo é mais complexos, mas com execução mais rápida. Exemplos: Quick Sort, Merge Sort, Heap Sort, Shell Sort.

Quanto à alocação de memória

- **Ordenação interna.** Usa somente as posições do vetor original para armazenamento de dados. Exemplos: Bubble Sort, Insertion Sort, Quick Sort.
- **Ordenação externa.** Usa um segundo vetor para armazenar os dados temporariamente. Exemplo: Merge Sort.

1.2. Custo computacional

Para avaliar o custo computacional dos algoritmos de ordenação leva-se em consideração o número de comparações realizadas, o número de movimentações de dados e o espaço extra de memória necessário. Dependendo da ordem em que os elementos se encontram no início da ordenação, alguns métodos podem variar a quantidade de comparações e movimentações, entre o melhor caso, o caso médio e o pior caso. Segue uma tabela comparativa entre alguns dos métodos de ordenação mais conhecidos.

Algoritmo	**Comparações**			**Movimentações**			**Memória extra**
	Melhor	**Médio**	**Pior**	**Melhor**	**Médio**	**Pior**	
Bubble	O(n)	$O(n^2)$		O(0)	$O(n^2)$		O(1)
Selection	$O(n^2)$			O(n)			O(1)
Insertion	O(n)	$O(n^2)$		O(n)	$O(n^2)$		O(1)
Merge	O(n log n)			O(n log n)			O(n)
Quick	O(n log n)		$O(n^2)$	O(n log n)		$O(n^2)$	O(1)

Para o objetivo dessa disciplina, daremos ênfase a um algoritmo simples (Bubble Sort) e um algoritmo eficiente (Merge Sort). Para quem tem interesse em conhecer outros algoritmos, são indicados os seguintes sites:

- Daniel Viana: "*Conheça os principais algoritmos de ordenação*" (https://www.treinaweb.com.br/blog/conheca-os-principais-algoritmos-de-

ordenacao). Esse site tem uma descrição interessante dos principais algoritmos, incluindo animações e vídeos ilustrativos.

- Wikipedia: *"Algoritmos de ordenação"* (https://pt.wikipedia.org/wiki/Algoritmo_de_ordena%C3%A7%C3%A3o). Tem uma relação mais completa de algoritmos, alguns deles com animações.

2. Bubble Sort

O método de ordenação ***bubble sort***, ou, em uma tradução livre, "ordenação pela bolha" é um dos algoritmos de ordenação dos mais simples. A ideia é percorrer o vetor várias vezes, comparando cada elemento com o próximo, e se o anterior for maior do que o próximo trocar os elementos de posição. Desta forma, a cada passagem pelo vetor, o maior elemento é levado até o final do vetor, ficando já na sua posição definitiva. Na próxima passagem pelo vetor, só é necessário ir até o penúltimo elemento, pois o último já está ordenado.

Para um vetor com `n` posições, o vetor será varrido `n` vezes. Na primeira vez, serão `n-1` comparações. Na segunda vez, `n-2` comparações, e assim por diante. Assim, o número de comparações é da ordem de $(n^2)/2$, ou seja, ordem de grandeza quadrática $O(n^2)$.

O algoritmo precisa continuar enquanto houver alguma troca (ou seja, há algum elemento fora de ordem). Se foi realizada uma passagem por todo o vetor, e não foi realizada nenhuma troca, significa que o vetor inteiro está em ordem, e o algoritmo pode ser encerrado.

Uma ilustração do funcionamento do algoritmo Bubble Sort segue abaixo. Observe que cada elemento é comparado com o próximo elemento, e se for maior que o próximo (comparações em vermelho), ocorre a troca. Caso contrário (comparações em verde), não há a troca. Para o exemplo, com um vetor de 5 elementos, são necessárias 4 passagens completas no vetor. A cada passagem, o último elemento é considerado ordenado, e não necessita mais ser comparado.

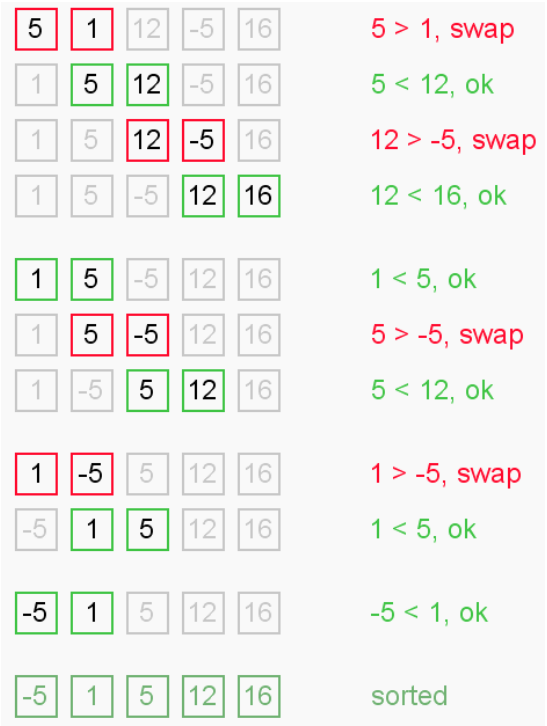

Figura 1. Ilustração da ordenação Bubble Sort
Fonte: (https://howtodoinjava.com/algorithm/bubble-sort-java-example/)

Por ser de ordem quadrática, não é indicado para processamento de grandes volumes de dados.

O código do bubble sort pode ser implementado como abaixo:

```
void bubbleSort(int *vet, int n)
{
   int i, j, trocou, aux;
   j=1;
   trocou=1;   // Sinalizador. Se houve trocas é porque há algo fora de ordem, e precisa continuar
   while ((j<n) && trocou)   // Continua enquanto houve trocas e não esgotou o vetor
   {
      trocou=0;   // Nessa passada não houve trocas
      for(i=0; i<(n-j); i++)
      {
         if(vet[i]>vet[i+1])   // Se o elemento é maior que o próximo, troca
         {
            aux= vet[i];
            vet[i]=vet[i+1];
            vet[i+1]=aux;
            trocou=1;     // Sinaliza que houve uma troca
         }
      }
      j++;
   }
}
```

Observe que esse algoritmo possui dois laços aninhados, ambos podendo ir até `n` repetições. Para cada repetição do laço externo, o laço interno faz `n` comparações, o que gera a complexidade $O(n^2)$ para o caso médio e para o pior caso. No melhor caso, que é quando passamos um vetor já ordenado, não há trocas, então o flag `trocou` não é passado para `1`, e o laço externo é encerrado.

Exemplos de animação do algoritmo bubbleSort podem ser visto em (https://www.youtube.com/watch?v=cB0oY1oZzng), (https://www.youtube.com/watch?v=lUqFelc84XE) e (https://www.youtube.com/watch?v=lyZQPjUT5B4)

3. MergeSort

O algoritmo Merge sort, ou, em tradução livre, ordenação por mistura é um exemplo de algoritmo que se baseia no princípio dividir para conquistar.

A ideia básica é dividir o vetor `vet` a ser ordenado em dois vetores menores, `a` e `b`, ordenar esses dois vetores menores e depois fazer o merge (a intercalação ou mistura) dos dois vetores menores `a` e `b` já ordenados. Isso pode ser feito considerando-se que os dois subvetores são adjacentes, ou seja, *b* começa onde *a* termina. E isso é feito recursivamente, até que cada vetor contenha apenas um elemento, que por definição já está ordenado.

O coração do algoritmo é a função `merge`, que faz justamente a intercalação dos dois subvetores, usando um vetor auxiliar. Esse vetor auxiliar implica em um consumo de memória extra. O vetor auxiliar pode ser alocado dinamicamente ou ter alocação estática. Neste exemplo estamos usando alocação estática.

A função `merge` faz a intercalação de dois vetores ordenados `a` e `b` formando um vetor `vet` comparando os primeiros elementos de cada vetor, e movendo o menor deles para o auxiliar, e assim por diante, enquanto houver elementos dos dois vetores. Quando um dos vetores chegar ao fim, copia-se os elementos restantes do outro vetor para o auxiliar. Ao final, move todos os valores do vetor auxiliar novamente para o vetor original `vet`.

```
// Intercala dois subvetores do vetor vet no vetor aux.
// O primeiro subvetor inicia na posição esq, e vai até meio
// O segundo subvetor inicia na posição meio+1 e vai até dir
void merge(int *vet, int *aux, int esq, int meio, int dir) {
   int l1, l2, i;

   // Percorre os dois subvetores, comparando os elementos
   // Escolhe o menor elemento para mover para o vetor auxiliar
   // Até acabar um dos vetores. De
   for(l1 = esq, l2 = meio + 1, i = esq; l1 <= meio && l2 <= dir; i++) {
      if(vet[l1] <= vet[l2])
         aux[i] = vet[l1++];
      else
         aux[i] = vet[l2++];
   }
   // Quando encerrou o for, é porque um dos subvetores a ou b chegou ao fim
   while(l1 <= meio)  // Se houver dados restantes no primeiro subvetor, copia
      aux[i++] = vet[l1++];

   while(l2 <= dir)   // Se houver dados restantes no segundo subvetor, copia
      aux[i++] = vet[l2++];

   // Move os dados do vetor auxiliar novamente para o vetor original vet.
   for(i = esq; i <= dir; i++)
      vet[i] = aux[i];
}
```

O algoritmo MergeSort, que faz a divisão do vetor em vetores menores, pode ser codificado como abaixo:

```
void mergeSort(int *vet, int *aux, int esq, int dir) {
   int meio;

   if(esq < dir) { // Só continua se o vetor tem pelo menos 2 elementos
      meio = (esq + dir) / 2;              // Divide o vetor ao meio
      mergeSort(vet, aux, esq, meio);      // Ordena por mergeSort a primeira metade
      mergeSort(vet, aux, meio+1, dir);    // Ordena por mergeSort a segunda metade
      merge(vet, aux, esq, meio, dir);     // Intercala os dois subvetores
   } else {
      return;
   }
}
```

Essa é uma implementação recursiva, que simplifica bastante a codificação do mergeSort.

Observe a semelhança da estrutura do código com os códigos de percurso em árvore binária, notadamente o pós-ordem. Há duas chamadas recursivas, para a primeira metade do vetor, e depois para a parte restante. Depois de ambas as chamadas, é chamada a função merge, que faz a intercalação de dois subvetores pré ordenados em um vetor vet.

A figura 2 ilustra o funcionamento do merge sort, para um vetor de 7 elementos.

Observe na parte em vermelho as chamadas recursivas da função `mergeSort`, dividindo o vetor em vetores menores, até chegar em vetores com somente um elemento. A partir desse ponto, na fase sinalizada com a cor verde, os vetores são intercalados em vetores maiores, já ordenados, até recompor o vetor original, de acordo com a função `merge`.

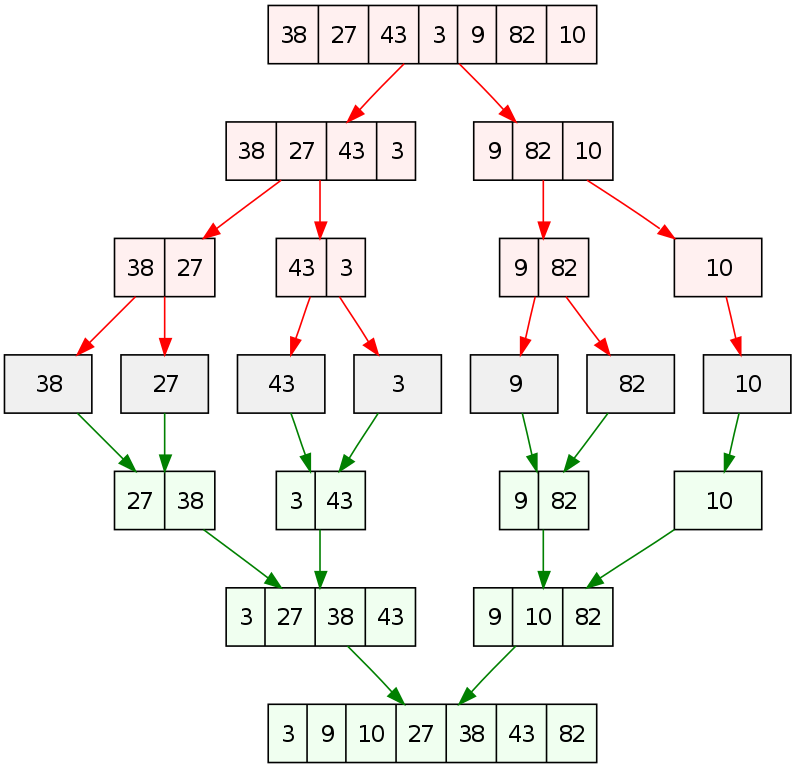

Figura 2. Ilustração do funcionamento do merge sort
Fonte: Wikipedia (https://en.wikipedia.org/wiki/Merge_sort)