UNIVERSIDADE FEDERAL DE RORAIMA

CENTRO DE CIÊNCIA E TECNOLOGIA

DEPARTAMENTO DE CIÊNCIA DA COMPUTAÇÃO

ESTRUTURA DE DADOS I

Boa Vista - RR

UNIVERSIDADE FEDERAL DE RORAIMA

CENTRO DE CIÊNCIA E TECNOLOGIA

DEPARTAMENTO DE CIÊNCIA DA COMPUTAÇÃO

ESTRUTURA DE DADOS I

JOÃO PAULO PARREIRA PEIXOTO

LARISSA SANTOS SILVA

ORDENAÇÃO

Boa Vista – RR

Sumário

1. Introdução

Este trabalho tem como finalidade, apresentar e implementar todos os métodos de ordenação, já vista na disciplina de estrutura de dados I.

Ordenação é o ato de alocar elementos sequencias de tal informações, ou dados, em uma relação predefinida. Esta ordenação poderá ser implementada de várias formas possíveis, dependendo apenas da quantidade de informação que deseja ser ordenada e o desempenho que deseja alcançar, já que isto varia de algoritmo para algoritmo.

2. Selection Sort

É um algoritmo de ordenação baseado em passar o menor valor do vetor para sua primeira posição (ou o maior assim dependendo da ordem desejada), sendo assim feita de forma sucessiva com a expressão (n-1) até os últimos dois elementos. (KONISHI, 2011).

Vantagens: selection sort tem suas vantagens em aplicações com lista pequena, já que sua aplicação é bem mais rápida pelo fato de não precisar de um armazenamento temporário, só de uma auxiliar. (WANDY).

Desvantagem: selection sort quando se trata de lista muito grande o seu desempenho torna-se ruim já que um algoritmo de n^2 possibilidades para cada n elementos, visto ainda que o seu desempenho é influenciado pela ordem inicial dos números. (WANDY).

Código em C:

```
void selection_sort(int *vet, int tam, double *selection_time, int *comparar, int *troca){
 clock_t start_time;
 start_time = clock();
 *comparar = 0;
 *troca = 0;
 int i, j, menor, aux;
 for (i = 0; i < tam-1; i++){
    menor = i;
    for (j = i+1; j < tam; j++){
       if(vet[j] < vet[menor]){
          menor = j;
       }
       *comparar+=1;
```

```
        }
        if (i != menor){
            aux = vet[i];
            vet[i] = vet[menor];
            vet[menor] = aux;
            *troca+=1;
        }
    }

    *selection_time = ((double)clock() - start_time) / (double)CLOCKS_PER_SEC;
}
```

3. Insertion Sort

Insertion sort, ou ordenação por inserção como também é conhecido, é um algoritmo simples, pois ele percorre um vetor de elementos da esquerda para a direita e à medida que avança vai deixando os elementos mais à esquerda ordenados. (KONISHI, 2011). Similar a ordenação de cartas de baralho com as mãos, em que pega-se uma carta de cada vez e a coloca em seu devido lugar, sempre deixando as cartas da mão em ordem. (BACKES, 2014).

Vantagem: insertion sort tem como principal vantagem a simplicidade e um bom desempenho em listas pequenas, tendo algoritmo de classificação estável, ou seja não altera a ordem de dados iguais, portanto o requisito de espaço e mínimo. (WANDY).

Desvantagem: Não se lidar muito bem com listas grandes, tendo assim um desempenho ruim e o tipo de inserção é particularmente útil somente em uma lista com poucos elementos. (WANDY).

Código em C:

```
void insertionSort(int *vet, int tam, double *insertion_time,unsigned long long int
*compara, int *troca){
```

```
    clock_t start_time;
    start_time = clock();
    *compara = 0;
    *troca = 0;
    int i, j, aux;
    for(i = 1; i < tam; i++){
        aux = vet[i];
        for(j = i; (j > 0) && (aux < vet[j - 1]); j--){
            *compara+=1;
            vet[j] = vet[j - 1];
        }
        vet[j] = aux;
        *troca+=1;
    }
    *insertion_time = ((double)clock() - start_time) / (double)CLOCKS_PER_SEC;
}
```

4. Bubble Sort

O bubble sort tem uma movimentação que lembra muito uma bolha de sabão (bolhas flutuantes), já que percorrer o vetor diversas vezes e a cada passagem fazem o elemento menor ou maior, dependendo de como deseja a ordenação, flutuar para o topo. (KONISHI, 2011).

Vantagem: O algoritmo tem várias vantagens e uma delas e a simplicidade de se escrito, fácil compreensão e leva apenas algumas linhas de códigos. (COMPUTER).

Desvantagem: Ao percorrer o vetor diversas vezes o algoritmo se torna extremamente lento, assim não sendo recomendado para grandes conjuntos de dados. (COMPUTER).

Código em C:

```
void bubble_sort(int vet[], int tam, double *bubble_time, int *comparar, unsigned long long int *troca){

    clock_t start_time;

    start_time = clock();

    int i, j, aux;

    for (i = 0 ; i < ( tam - 1 ); i++){

        for (j = 0 ; j < tam - i - 1; j++){

            if (vet[j] > vet[j+1]){

                /* Swapping */

                aux         = vet[j];

                vet[j]   = vet[j+1];

                vet[j+1] = aux;

                *troca+=1;

            }

            *comparar+=1;

        }

    }

    *bubble_time = ((double)clock() - start_time) / (double)CLOCKS_PER_SEC;

}
```

5. Radix Sort

O radix sort foi originalmente usado para ordenar cartões perfurados, sendo assim um algoritmo de ordenação rápida e estável que pode ser usada para ordenar itens que estão identificados por chaves únicas. (KONISHI, 2011)

Vantagens: radix sort é um algoritmo estável, preservam a ordem de chaves iguais, apresentam tempo linear e bastante eficiente quando o número de registros é grande mas o tamanho da chave é pequeno. (MELO, 2010)

Desvantagens: não funciona bem quando as chaves são longas, pois o custo é proporcional ao tamanho da chave e o número de elementos a serem ordenados, além disto precisam de memória auxiliar para ordenar o conjunto. (MELO, 2010)

Código em C:

```
void radixsort(int vetor[], int tamanho, double *radix_time) {
   clock_t start_time;
   start_time = clock();
   int *b;
   int maior = vetor[0];
   int i, exp = 1;
   b = (int *)calloc(tamanho, sizeof(int));
   for (i = 0; i < tamanho; i++) {
      if (vetor[i] > maior)
         maior = vetor[i];
   }
   while (maior/exp > 0) {
      int bucket[10] = { 0 };
      for (i = 0; i < tamanho; i++)
```

```
        bucket[(vetor[i] / exp) % 10]++;
        for (i = 1; i < 10; i++)
            bucket[i] += bucket[i - 1];
            for (i = tamanho - 1; i >= 0; i--)
                b[--bucket[(vetor[i] / exp) % 10]] = vetor[i];
                for (i = 0; i < tamanho; i++)
                    vetor[i] = b[i];
                    exp *= 10;
    }
    free(b);
    *radix_time = ((double)clock() - start_time) / (double)CLOCKS_PER_SEC;
}
```

6. Quick Sort

O quick sort e um algoritmo de ordenação não estável, porém com algumas modificações pode se tornar estável, rápido e eficiente. O mesmo adota a estratégia de dividir para conquistar, que funciona da seguinte maneira, escolhe um elemento da lista que denomina pivô em seguida reorganiza a lista de forma que todos os elementos anteriores ao pivô sejam menores, e os posteriores ao pivô sejam maiores. Essa operação é chamada de partições. (KONISHI, 2011)

Vantagem: melhor opção para ordenar vetores grandes, muito rápido por que o laço interno é simples, memoria auxiliar para pilha de recursão é pequena, complexidade no melhor caso é O(n lg(n)) e no caso médio é O(n lg(n)). (CHUNHA)

Desvantagem: A desvantagem do quick sort é não ser estável e seu pior caso é quadrático $O(N^2)$. (CHUNHA)

Código em C:

```
void quick(int *V, int inicio, int fim, double *quick_time, int *comparar, int *troca){
```

```
clock_t start_time;
start_time = clock();
int particiona(int *V, int inicio, int fim){
    int esq, dir, pivo, aux;
    esq = inicio;
    dir = fim;
    pivo = V[inicio];

    while(esq < dir){
            *comparar+=1;
        while (V[esq] <= pivo){
            *comparar+=1;
            esq++;
        }
        while (V[dir] > pivo){
            *comparar+=1;
            dir--;
        }
        if (esq < dir){
            *troca+=1;
            aux = V[esq];
            V[esq] = V[dir];
            V[dir] = aux;
```

```
            }
        }
        V[inicio] = V[dir];
        V[dir] = pivo;
        return dir;
    }
    int pivo;
        if(fim > inicio){
            *comparar+=1;
            pivo = particiona(V, inicio, fim);
            quick(V, inicio, pivo-1, quick_time, comparar, troca);
            quick(V, pivo+1, fim, quick_time, comparar, troca);
        }
    *quick_time = ((double)clock() - start_time) / (double)CLOCKS_PER_SEC;
}
```

7. Merge Sort

O merge sort é classificado como ordenação por mistura, que parte no princípio de “dividir para conquistar”, onde divide recursivamente, o conjunto de dados até que cada subconjunto possua 1 elemento, combina 2 subconjuntos de forma a obter 1 conjunto maior e ordenado, esse processo se repete até que exista apenas 1 conjunto.(BACKES, 2014).

Vantagem: É um algoritmo de ordenação simples de implementar e fácil entendimento. (FARIAS, 2014).

Desvantagem: O merge sort gasta mais memória que os outros métodos, devido as chamadas recursivas, precisará de um vetor auxiliar e cada vez que é chamado recursivamente cria novos vetores. (BACKES, 2014).

Código em C:

```
void void mergeSort (int *vet, int inicio, int fim, double *merge_time, int *comparar, int *troca){
   clock_t start_time;
   start_time = clock();
   void merge(int *vet, int inicio, int meio, int fim){
      int *aux, p1, p2, tam, i, j, k;
      int fim1 = 1, fim2 = 1;
      tam = fim-inicio+1;
      p1 = inicio;
      p2 = meio +1;
      aux = (int *) malloc(tam*sizeof(int));
      for (i = 0; i < tam; i++){
         if(fim1 && fim2){
               *comparar+=1;
               //*troca+=1;
            if(vet[p1] < vet[p2]){
               aux[i] = vet[p1++];
            }
            else{
               aux[i] = vet[p2++];
```

```
            }
            if(p1 > meio)
                fim1 = 0;
            if(p2 > fim)
                fim2 = 0;
        }
        else{
                *troca+=1;
            if(fim1){
                aux[i] = vet[p1++];
            }
            else{

                aux[i] = vet[p2++];
            }
        }
    }
    for(j = 0, k = inicio; j < tam; j++, k++){
        vet[k] = aux[j];
    }

    free(aux);
}
```

```
    int meio;
    if (inicio < fim){
        meio = (inicio + fim)/2;
        mergeSort(vet, inicio, meio, merge_time, comparar, troca);
        mergeSort(vet, meio+1, fim, merge_time, comparar, troca);
        merge(vet, inicio, meio, fim);
    }

    *merge_time = ((double)clock() - start_time) / (double)CLOCKS_PER_SEC;
}
```

8. Busca sequencial

A busca sequencial é uma forma bem simples de buscar elementos, podendo ser aplicada apenas em tabelas já ordenadas. Percorrendo registros por registros em busca do elemento desejado. (GARCIA, 2009)

Vantagem: Possui um melhor resultado, quando se trata de buscas pequenas e é capaz de realizar uma busca em um vetor tanto desordenado quanto ordenado. (GARCIA, 2009)

Desvantagem: Uma única recuperação não indica que o registro será frequentemente recuperado, além disto o método e mais “caro” em vetores do que em listas. (GARCIA, 2009). Não recomendado para vetores ou listas de grande tamanho, pois faz a busca elemento a elemento.

Código em C:

```
void buscaLinear(int vet[], int tam, int elemento){
    int i;
```

```
int *aux ;
aux = (int *) malloc(tam*sizeof(int));
if (tam == 100){
    FILE *arquivo100;
    arquivo100 = fopen("arquivo100.txt", "r");
    for (i=0; i<tam; i++){
        fscanf(arquivo100, "%d", &aux[i]);
    }
    fclose(arquivo100);
}
if (tam == 1000){
    FILE *arquivo1000;
    arquivo1000 = fopen("arquivo1000.txt", "r");
    for (i=0; i<tam; i++){
        fscanf(arquivo1000, "%d", &aux[i]);
    }
    fclose(arquivo1000);
}
if (tam == 10000){
    FILE *arquivo10000;
    arquivo10000 = fopen("arquivo10000.txt", "r");
    for (i=0; i<tam; i++){
        fscanf(arquivo10000, "%d", &aux[i]);
```

```
    }
    fclose(arquivo10000);
}
if (tam == 100000){
    FILE *arquivo100000;
    arquivo100000 = fopen("arquivo100000.txt", "r");
    for (i=0; i<tam; i++){
        fscanf(arquivo100000, "%d", &aux[i]);
    }
    fclose(arquivo100000);
}

for (i = 0; i < tam; i++){
    if (aux[i] == elemento){
        printf("\nValor %d encontrado na posição %d.\n", aux[i], i);
    }
}

for (i = 0; i < tam; i++){
    if (aux[i] == elemento){
        break;
    }
}
```

```
    if (aux[i] != elemento)
        printf("\nValor não encontrado\n");
    free(aux);
}
```

9. Busca Binária

A busca binaria compara a chave com o registrador que está no meio da tabela, se a chave é menor que o registrador que está na primeira metade da tabela, ou se a chave e maior que o registrador da segunda metade da tabela. Repete-se até que a chave seja encontrada. (TOFFOLO, 2013).

Vantagem: A vantagem da busca binaria e a eficiência, e simplicidade de implementar. (TOFFOLO, 2013). Recomendada para vetores grandes e mais rápida.

Desvantagem: para ser realizada a busca os vetores precisam necessariamente estar ordenados antes.

Código em C:

```
int buscaBinaria(int *vet, int tam, int elemento){
    int inicio, meio, fim;
    inicio = 0;
    fim = tam-1;
    while(inicio <= fim){
        meio = (inicio + fim)/2;
        if (elemento < vet[meio])
            fim = meio - 1;
        else
            if (elemento > vet[meio])
            inicio = meio + 1;
```

```
        else{
        printf("\nElemento %d encontrado na posição vet[%d].\n", elemento, meio);
        continue;
        return meio;
        }
    }
    printf("\nElemento não encontrado.\n");
    return -1;
}
```

10. Heap Sort

O heap sort possui o princípio da ordenação por seleção o mesmo utilizado pelo selection sort, onde seleciona o menor elemento do vetor e troca pelo elemento da primeira posição, repetindo sucessivamente até que estejam ordenados. (MENOTTI, 2015)

Vantagem: O comportamento do heap sort é sempre o mesmo O(n log n), independente da sua entrada. (ZIVIANI)

Desvantagem: O anel interno do algoritmo é bastante complexo se for comparado com quick sort, o heap sort não é estável. (ZIVIANI)

Código em C:

```
void heapSort(int *vet, int inicio, int tam, double *heap_time, int *comparar, int *troca){
    clock_t start_time;
    start_time = clock();


    void criaHeap(int *vet, int i, int f){
        int aux = vet[i];
```

```
    int j = i*2+1;
    while (j <= f){
        if (j < f){
            *comparar+=1;
            if (vet[j] < vet[j + 1]){
                j++;
            }
        }
        *comparar+=1;
        if(aux < vet[j]){
            *troca+=1;
            vet[i] = vet[j];
            i = j;
            j = 2 * i + 1;
        }
        else{
            j = f + 1;
        }
    }
    vet[i] = aux;
}


    int i, aux;
```

```
    for (i = (tam-1)/2; i >= 0; i--){
        criaHeap(vet, i, tam-1);
    }

    for (i = tam-1; i >= 1; i--){
        *troca+=1;
        aux = vet[0];
        vet[0] = vet[i];
        vet[i] = aux;
        criaHeap(vet, 0 , i-1);
    }

    *heap_time = ((double)clock() - start_time) / (double)CLOCKS_PER_SEC;
}
```

11. Conclusão

Através da realização deste trabalho podemos concluir que independentemente do método de ordenação que esteja sendo usado, sempre levará em conta as quantidade de elementos que está trabalhando, já que seu desempenho dependerá do mesmo. Tentando assim, atender todas as necessidades da melhor forma possível.

Então podemos concluir que não tem um algoritmo de ordenação melhor ou pior que o outro, apenas tem um algoritmo que melhor analisa um caso especifico melhor que outro.

Sendo assim, o primeiro passo para trabalhar com algoritmo de ordenação e escolher qual ira supri suas necessidade em tal tarefa.

12. Referencias

WANDY, Joe. **The Advantages & Disadvantages of Sorting Algorithms**. Disponível em <http://www.ehow.com/info_8446142_advantages-disadvantages-sorting-algorithms.html> acessado em 25 de fevereiro de 2017.

KONISHI, Mari. **Métodos de ordenação.** (2011). Disponível em <https://marikonishi.wordpress.com/category/metodos-de-ordenacao/> acessado em 25 de fevereiro de 2017.

COMPUTER. **Vantagens e Desvantagens de Bubble Sort**. Disponível em <http://ptcomputador.com/P/computer-programming-languages/87814.html> acessado em 25 de fevereiro de 2017.

MELO, carlos. **Radix e Bucket Sort.** (2010) Disponível em <https://cavmelo.files.wordpress.com/2010/11/radixebucketsort.pdf> acessado em 25 de fevereiro de 2017.

CHUNHA. **Algoritmos e Estrutura de Dados.** Disponível em <http://homepages.dcc.ufmg.br/~cunha/teaching/20121/aeds2/quicksort.pdf> acessado em 25 de fevereiro de 2017.

FARIAS, Ricardo. **Estrutura de Dados e Algoritmos.** (2014). Disponível em < http://www.cos.ufrj.br/~rfarias/cos121/aula_07.html > acessado em 25 de fevereiro de 2017.

TOFFOLO, Túlio. **Pesquisa Sequencial e Binária.** (20113). Disponível em < http://www.decom.ufop.br/toffolo/site_media/uploads/2013-1/bcc202/slides/20._pesquisa_(parte_1).pdf> acessado em 25 de fevereiro de 2017.

GARCIA, João Luís. **Métodos de Busca.** (2009). Disponível em <http://wiki.icmc.usp.br/images/d/df/ICC2_12.Busca.pdf> acessado em 25 de fevereiro de 2017.

MENOTTI, David. **Filas de Prioridade – Heap.** (2015). Disponível em < http://www2.dcc.ufmg.br/disciplinas/aeds2_turmaA1/cap4.pdf > acessado em 25 de fevereiro de 2017.

ZIVIANI, Nívio. **Projeto de Algoritmo.** Disponível em < http://www2.dcc.ufmg.br/disciplinas/aeds2_turmaA1/cap4.pdf > acessado em 25 de fevereiro de 2017.