



UNIVERSIDADE PAULISTA
CURSO DE CIÊNCIA DA COMPUTAÇÃO

APS – ATIVIDADES PRÁTICAS SUPERVISIONADAS

SANTOS – SP

2014



COLABORADORES

MATHEUS RODRIGUES MARTINS PEREIRA – RA: B73ABD-5

MATHEUS FELIPE DOS PASSOS E PAZ – RA: B57IAJ-0

MATHEUS GONÇALVES BRANDÃO – RA: B67712-0

JOHNNY AMANCIO SANTOS – RA: B633FF-8

APS – ATIVIDADES PRÁTICAS SUPERVISIONADAS

“DESENVOLVIMENTO DE SISTEMA PARA ANÁLISE DE PERFORMANCE DE ALGORITMOS DE ORDENAÇÃO DE DADOS”

Atividades Práticas Supervisionadas
trabalho apresentado como exigência
para avaliação do segundo bimestre,
em disciplina do 3º semestre, do curso
de Ciência da Computação da
Universidade Paulista, Sob orientação
do professor Takeshi.

SANTOS – SP

2014

SUMÁRIO

1 OBJETIVO DO TRABALHO.....	04
2 INTRODUÇÃO.....	04
3 REFERENCIAL TEORICO.....	06
4 DESENVOLVIMENTO.....	13
5 RESULTADOS E DISCUSSÃO.....	25
6 CONSIDERAÇÕES FINAIS.....	29
7 BIBLIOGRAFIA.....	29
8 CODIGO FONTE.....	30
9 FICHA DE ATIVIDADES PRÁTICAS SUPERVISIONADAS.....	38

OBJETIVO DO TRABALHO

O objetivo do trabalho tende a desenvolver dois programas estruturados com algoritmos de ordenação de dados onde a unidade de medida para efeito de comparação entre os dois algoritmos foi o tempo de ordenação entre eles.

O tempo de aquisição não foi contabilizado, e sim apenas o processo específico da ordenação.

INTRODUÇÃO

Algoritmo de ordenação, em ciência da computação, é um algoritmo que coloca os elementos de uma dada sequência em certa ordem. Em outras palavras efetua sua ordenação completa ou parcial. O objetivo da ordenação é facilitar a recuperação dos dados de uma lista.

Sabemos que as principais linguagens de programação atuais (para não afirmar todas) possuem alguma função para essa tarefa, ou seja, com apenas uma chamada de função (sort (), qsort (), qsort (), etc....) conseguimos ordenar determinada estrutura de dados. Mas muitas vezes precisamos fazer alguma coisa enquanto estamos ordenando esses dados, seja para adicionar mais coisas aos dados, seja para guardar determinadas informações desse processo. Para isso temos que programarmos nós mesmos essa função. Abaixo vou explicar um pouco sobre os principais algoritmos existentes:

Algoritmo de ordenação por inserção

Esse algoritmo, como muitos outros, é baseado em ações que nós (como pessoas) fazemos no dia-a-dia para resolver o problema. Lembra quando você está jogando

baralho, e suas cartas já estão na mesa e você precisa colocá-las na mão de uma forma ordenada? Essa ordenação deve ser feita de uma maneira que você esteja acostumado a escolher uma carta facilmente para jogar mais rápido e melhor... É exatamente isso que o algoritmo de ordenação por inserção faz por baixo dos panos. A idéia principal dele é: adiciona-se um item qualquer à estrutura de dados, depois, para cada item que ainda não esteja na estrutura, antes de adicioná-la, comparar com cada item que já está nela (consequentemente já ordenada) até encontrar a posição a ser encaixada. É exatamente o que fazemos com o baralho. Essa opção é boa quando temos uma entrada pequena de dados, para entradas grandes pode se consumir muito tempo de processamento.

Algoritmo Bubblesort

Talvez um dos mais populares dos algoritmos para ordenação seja o bubblesort, isso pela fácil memorização de como funciona e como é fácil a sua implementação. Ele consiste basicamente em intercalar elementos, por isso se enquadra na categoria de ordenação por intercalação, a implementação dele é simples. Com uma estrutura de dados desordenada inicia-se o algoritmo pelo primeiro elemento, depois faz-se a comparação dele com todos os que estão depois dele na estrutura desordenada, portanto com 4 linhas de código dá para se implementar.

Algoritmo Mergesort

Um algoritmo bem interessante por alguns motivos. Esse é um algoritmo que segue uma técnica de dividir para conquistar na base de sua idéia principal. O raciocínio básico para que ele funcione é o seguinte: ordenar uma estrutura significa ordenar várias subestruturas internas já ordenadas, caso essas estruturas não estejam ordenadas, basta ordená-las pelo mesmo método (ordenar suas subestruturas internas... até o infinito), é um algoritmo com uma base bem matemática e com um método bem interessante para estudo. Como a idéia pode não ter ficado bem clara vou mostrar um exemplo: ordenar 6, 3, 4, 8, 1, 2, 3, 5... o algoritmo irá dividir em pares ordenados: {3,6}, {4,8}, {1,2}, {3,5}, depois ir fazendo o merge desses dados, ou seja, juntando-os em dois pares ordenados: {3,4,6,8}, {1,2,3,5}, depois juntar

novamente: 1,2,3,3,4,5,6,8. No final do algoritmo a sequência inicial está ordenada a partir de divisões... Por sua base bem estruturada esse algoritmo tem tempo médio bem rápido.

Algoritmo Heapsort

Esse algoritmo tem tempo de execução de um algoritmo de ordenação por intercalação, e faz as suas operações localmente, sempre apenas um número constante de elementos é armazenados fora da estrutura de dados, como é feito o algoritmo por inserção. Mas o que ele tem de interessante é a forma que ele arranja os dados para depois ordená-los, ele utiliza uma estrutura chamada de heap ou monte, que é uma árvore binária que é extremamente importante para vários conceitos e problemas computacionais, pois depois de ordenados os dados podem, por exemplo, saber em que nível da árvore se encontra determinado item, operações sobre árvores binárias são conceitos muito importantes para outras estruturas como grafos.

Referencial teórico

Merge Sort

Introdução

O merge sort, ou ordenação por mistura, é um exemplo de algoritmo de ordenação do tipo dividir-para-conquistar.

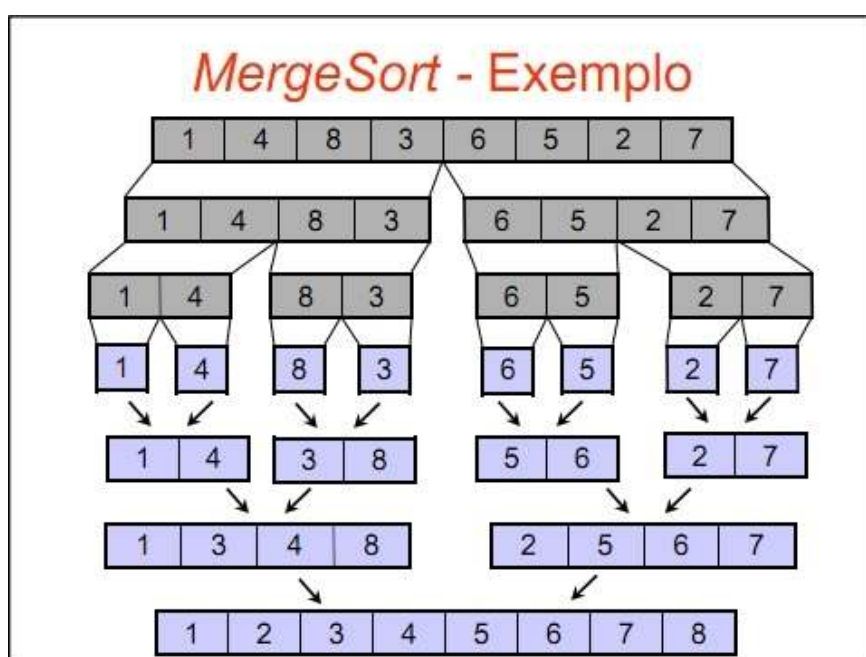
Sua ideia básica consiste em Dividir(o problema em vários sub-problemas e resolver esses sub-problemas através da recursividade) e Conquistar(após todos os sub-problemas terem sido resolvidos ocorre a conquista que é a união das resoluções dos sub-problemas). Como o algoritmo do Merge Sort usa a recursividade em alguns problemas esta técnica não é muito eficiente devido ao alto consumo de memória e tempo de execução.

Os três passos úteis dos algoritmos dividir-para-conquistar, ou *divide and conquer*, que se aplicam ao *merge sort* são:

1. Dividir: Dividir os dados em subsequências pequenas;
2. Conquistar: Classificar as duas metades recursivamente aplicando o *merge sort*;
3. Combinar: Juntar as duas metades em um único conjunto já classificado.

O Algoritmo e como funciona

Esse algoritmo de ordenação, consiste em quebrar todos os elementos pela metade, até com que eles fiquem todos separados, a partir daí, ordenando de par(es) em par(es)



Como vimos, ele pode parecer complicado no começo, mas na verdade ele é bem simples, apesar de ser bem mais complexo que o Bubble Sort.

Apesar desse algoritmo utilizar muitas funções recursivas, o que causa alto uso de memória, ele é considerado um algoritmo rápido.

Exemplo de código com método Marge Sort

```

inteiro [] sort(inteiro [] array)
inicio
  se array.tamanho <= 1, então
    retornar array
  fim se

  inteiro meio = array.tamanho/2
  inteiro [meio] esq
  inteiro [] dir

  se array.tamanho for divisível por dois, então
    dir = inteiro [meio];
  fim se
  senão
    dir = inteiro [meio+1]

  inteiro [array.tamanho] aux

  para i de 0 até meio
    faça
      esq[i] = array[i]
    fim para

  int auxIndex = 0;

  para i de meio até array.tamanho
    faça
      dir[auxIndex] = array[i]
      auxIndex = auxIndex + 1
    fim para

  esq = sort(esq)
  dir = sort(dir)

  aux = mergesort(esq, dir)

  retornar aux
fim

inteiro [] mergesort(inteiro [] esq, inteiro [] dir)
inicio
  inteiro [esq.tamanho+dir.tamanho] aux

```



```

inteiro indexDir = 0, indexEsq=0, indexAux = 0

enquanto indexEsq < esq.tamnhho ou indexDir < dir.tamanho
  faça
    se indexEsq < esq.tamnhho e indexDir < dir.tamanho, então
      se esq[indexEsq] <= dir[indexDir], então
        aux[indexAux] = esq[indexEsq]
        indexAux = indexAux + 1
        indexEsq = indexEsq + 1
      fim se
    senão
      aux[indexAux] = dir[indexDir]
      indexAux = indexAux + 1
      indexDir = indexDir + 1
    fim se
  senão se indexEsq < esq.tamanho, então
    aux[indexAux] = esq[indexEsq]
    indexAux = indexAux + 1
    indexEsq = indexEsq + 1
  fim se
  senão se indexDir < dir.tamanho, então
    aux[indexAux] = dir[indexDir]
    indexAux = indexAux + 1
    indexDir = indexDir + 1
  fim se
fim enquanto

retornar aux
fim

```

Vantagens

- Útil para ordenação externa;
- Pior caso: $O(n \log^2 n)$;
- Aplicações com restrição de tempo;
- Fácil implementação.

Desvantagens

- Utiliza memória auxiliar;
- Alto consumo de memória.

Quicksort

Introdução

O algoritmo Quicksort é um método de ordenação muito rápido e eficiente, inventado por C.A.R. Hoare em 1960 , quando visitou a Universidade de Moscovio como estudante. Naquela época, Hoare trabalhou em um projeto de tradução de máquina para o National Physical Laboratory. Ele criou o 'Quicksort ao tentar traduzir um dicionário de inglês para russo, ordenando as palavras, tendo como objetivo reduzir o problema original em subproblemas que possam ser resolvidos mais facil e rapidamente. Foi publicado em 1962 após uma série de refinamentos. O Quicksort é um algoritmo de ordenação por comparação não-estável.

O Algoritmo

Este é o algoritmo mais eficaz para ordenação. A ideia por trás do algoritmo é escolher um valor "médio" (o valor do meio, que pode não corresponder exatamente ao valor do meio do vetor, depois de este estar ordenado) do vetor, passar todos os valores maiores do que ele para a frente e todos os menores para trás.

Ficamos então com o vetor dividido em duas partes, uma tem todos os valores menores que o valor escolhido, a outra tem todos os valores maiores que o valor escolhido. Aplicamos agora o algoritmo a cada uma das partes.

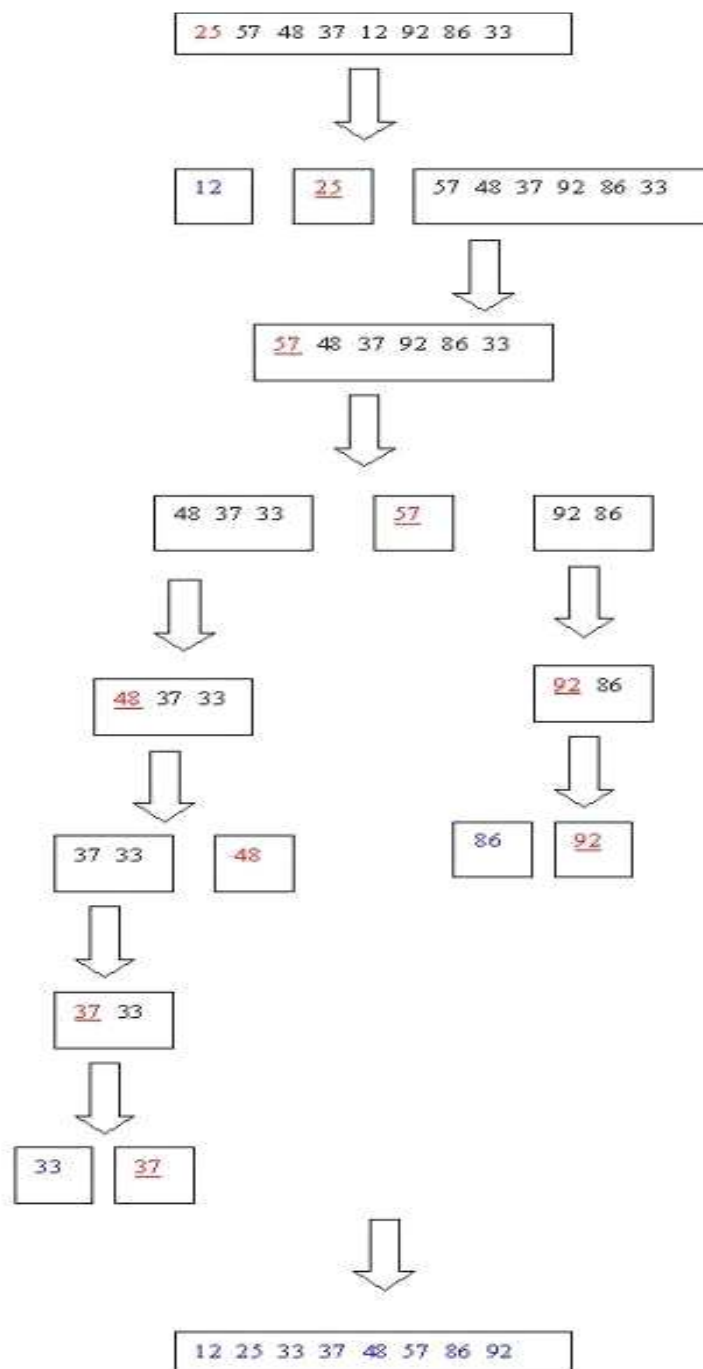
O processo é repetido até atingirmos partes de tamanho um, ou seja, Quicksort adota a estratégia de divisão e conquista. A estratégia consiste em rearranjar as chaves de modo que as chaves "menores" precedam as chaves "maiores". Em seguida o Quicksort ordena as duas sub-listas de chaves menores e maiores recursivamente até que a lista completa se encontre ordenada.

Como funciona

Este método divide a tabela em duas sub-tabelas, a partir de um elemento chamado pivô, normalmente o 1º elemento da tabela.

Uma das sub-tabelas contém os elementos menores que o pivô enquanto a outra contém os maiores. O pivô é colocado entre ambas ficando na posição correta

Exemplo de funcionamento do QuickSort



Legenda:

- É o pivô
- Elemento já ordenado
- Elementos por ordenar

Exemplo de código com método Quicksort

```

QuickSort (a, inicio, fim)
  If inicio < fim
    Meio = particiona (inicio, fim)
    QuickSort (inicio, meio-1)

```

```

    QuickSort (meio + 1, fim)
return

Particiona (inicio, fim)
    //troca o primeiro valor do vetor
    com o valor do meio para que o valor
    "médio" fique no início do vetor
    meio=(inicio+fim)/2
    aux=a[inicio];
    a[inicio]=a[meio];
    a[meio]=aux;

    pivot = a[inicio] //coloca como pivot o valor
        "médio" que se encontra no
        início do vetor
    ultbaixo = inicio //numero de elementos no
        vetor menores que o pivot
    for i = inicio to fim do
        if a[i] < pivot
            ultbaixo = ultbaixo + 1
            aux=a[i]
            a[i]=a[pivot]
            a[pivot]=a[i]

    //troca o pivot com o último elemento do
    vetor menor do que ele
    aux=a[inicio]
    a[inicio]=a[ultbaixo]
    a[ultbaixo]=aux
return ultbaixo

```

Conclusão

Este método de ordenação em comparação a outros algoritmos é um dos mais rápidos e eficientes para ser usado, fazendo com que este seja um dos melhores tipos de método de ordenação.

Desenvolvimento

Dados

Os dados utilizados para ordenação nessa aplicação foram do desmatamento da Amazônia Legal, fornecidos em quilômetros quadrados pelo Boletim Transparência Florestal da Amazônia Legal, através do site Imazon. Estes dados são detectados mensalmente pelo Sistema de Alerta de Desmatamento (SAD).

O Imazon é um instituto de pesquisa cuja missão é promover o desenvolvimento sustentável na Amazônia por meio de estudos, apoio à formulação de políticas públicas, disseminação ampla de informações e formação profissional.

Segue o link de onde os dados utilizados no programa foram importados:

http://www.imazon.org.br/publicacoes/transparencia-florestal/transparencia-florestal-amazonia-legal?b_start:int=0

Boletim do Desmatamento (SAD)

fevereiro e março 2014

Fonseca, A., Martins, H., Souza Jr., C., Sales, M., & Veríssimo, A. 2014. Boletim Transparência Florestal da Amazônia Legal (fevereiro e março de 2014) (p. 13). Belém: Imazon.

Recomendar

1

Tweetar

9

Enviar por e-mail

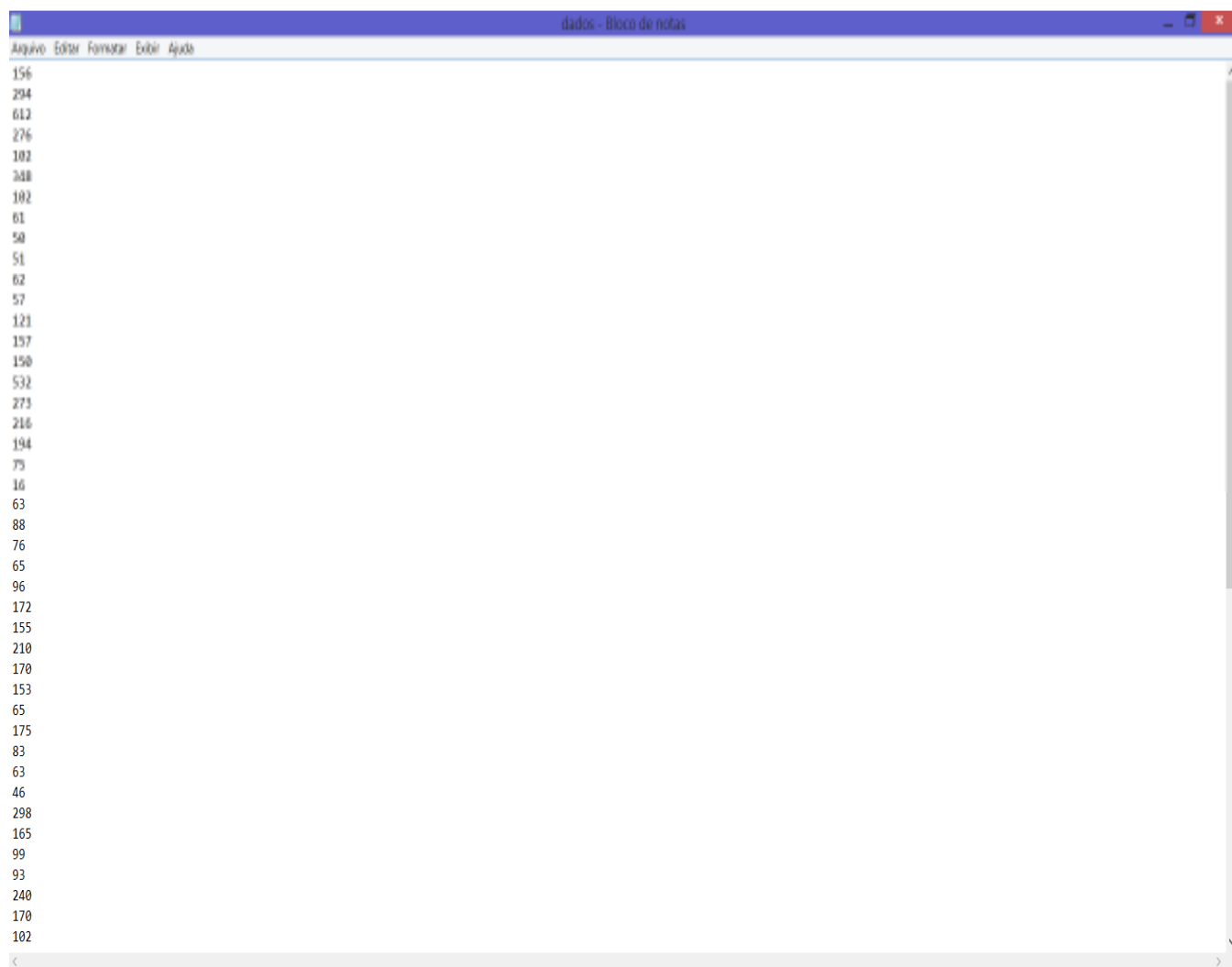
Em março de 2014, o Sistema de Alerta de Desmatamento (SAD) detectou 20 quilômetros quadrados de desmatamento na Amazônia Legal. Isso representou uma redução de 75% em relação a março de 2013 quando o desmatamento somou 79 quilômetros quadrados. Por sua vez, em fevereiro de 2014 foram registrados 11 quilômetros quadrados de desmatamento, o que representou uma redução de 77% em relação a fevereiro de 2013 quando o desmatamento atingiu 45 quilômetros quadrados.



O desmatamento acumulado no período de agosto de 2013 a março de 2014, correspondendo aos oito primeiros meses do calendário atual de desmatamento, totalizou 560 quilômetros quadrados. Isso significa uma redução do desmatamento acumulado de 61% em relação ao período anterior (agosto de 2012 a março de 2013) quando o desmatamento somou 1.430 quilômetros quadrados.

Em março de 2014, a maioria (52%) do desmatamento ocorreu no Mato

Conforme os dados eram puxados do site Imazon, eram colocados em um documento .txt criado na unidade C: do computador, linha por linha e em ordem mensal, no caso o primeiro valor colocado no documento corresponde ao valor de abril de 2008, e assim sucessivamente até o último valor que é de março de 2014. Conforme inserido valores uma linha abaixo no documento .txt, o software irá ordena-los e já mostrar para o usuário como valor correspondente ao mês seguinte. Isso será mostrado mais a frente neste tópico. Segue a imagem do documento dados.txt que foi usado para armazenamento dos dados:



Conforme foi dito, os dados organizados linha por linha, que é a forma que o programa irá identificar cada valor e coloca-los em um vetor, para assim começar a realizar a operação de ordenação em ambos os métodos MergeSort e QuickSort.

MergeSort

Começa agora a parte do desenvolvimento da aplicação, vamos começar pelo MergeSort. Após criado o .txt com os dados colocados linha por linha, é necessário que estes sejam transferidos para um vetor. Então é necessário que o programa localize o arquivo dados.txt que está no C:/. Para isso foi importada a classe System.IO e foi usado o comando StreamReader. Feito isso, inicialmente é necessário uma contagem de elementos contidos no .txt (para definir o índice do

vetor), e depois é criado o vetor e os valores são armazenados, cada linha é correspondente a um espaço no vetor numOrd:

```
static void Main(string[] args)
{
    int max = 0;
    string line;

    StreamReader file = new StreamReader(@"dados.txt"); //Importando o arquivo txt
    Console.WriteLine("\nOrdenação usando o algoritmo MergeSort\n\nSistema de Alerta de Desmatamento (SAD) Dados do desmatamento na Amazônia Legal (Em KM²): ");
    while ((line = file.ReadLine()) != null) //Fazendo a contagem de elementos que o arquivo txt possui, linha por linha
    {
        max++;
    }

    file.Close(); // Fecha o arquivo txt

    int[] numOrd = new int[max]; //Declara o vetor, com o índice igual ao número de elementos do arquivo txt
    int[] numOrdCopia = new int[max];
    StreamReader file2 = new StreamReader(@"dados.txt"); //Importa novamente o arquivo txt
    for (int i = 0; i < max; i++)
    {
        line = file2.ReadLine();
        numOrd[i] = Convert.ToInt32(line); //Colocando no vetor todos os elementos do arquivo txt, um índice por linha
    }

    for (int i = 0; i < max; i++)
    {
        numOrdCopia[i] = numOrd[i]; //Colocando no vetor todos os elementos do arquivo txt, um índice por linha
    }
}
```

O código acima apenas dá uma introdução ao programa, faz a contagem de dados do arquivo .txt, cria os vetores que serão usados no programa e passa os dados do arquivo para os mesmos vetores (O vetor numOrdCopia será usado para integração dos dados com os meses e anos correspondentes, isso será mostrado no final da apresentação do programa).

Agora que os dados já estão devidamente alocados no vetor, começará a ordenação dos mesmos. Um método específico é chamado para iniciar a ordenação, já enviando os dados do vetor numOrd e dos índices da ponta do vetor, no caso o zero e o valor de max (que contém o número de dados do .txt) menos 1, através dessa linha de código:

```
MergeSort(numOrd, 0, max - 1); //Chama o método MergeSort que fará a ordenação dos dados
```

Depois disso o método MergeSort é executado e segue as riscas do algoritmo. No método MergeSort ele “divide” os índices dos vetores e envia os valores para esquerda (que no caso seria o menor valor do vetor) e direita (o maior valor do vetor). Assim ele chama o método Merge, que esse sim fará de fato o processo de ordenação. A princípio veja o código do método MergeSort:

```
static public void MergeSort(int[] numOrd, int esquerda, int direita) //Este método vai "dividir" o vetor numOrd e enviar para o método Merge os valores das variáveis esquerda e direita
{
    int mid;

    if (direita > esquerda)
    {

        mid = (direita + esquerda) / 2;

        MergeSort(numOrd, esquerda, mid);
        MergeSort(numOrd, (mid + 1), direita);

        Merge(numOrd, esquerda, (mid + 1), direita);
    }
}
```

Perceba que no final do método ele está chamando o outro método (Merge) que será apresentado na próxima página.

Até agora, pouco se viu de ordenação de dados e realmente como o método MergeSort funciona na prática. Porém, a ordenação em si ocorre no método que será apresentado agora. O método Merge recebe os dados, tanto do vetor numOrd, quanto os valores dos índices que deverão ser ordenados. No MergeSort os dados são divididos, comparados e depois são mesclados novamente, já ordenados. Para auxiliar nessa operação foi criado o vetor temp, que terá um papel importante na ordenação. São criadas também três variáveis locais (eol, num, pos) para auxiliarem também na ordenação dos dados. Após criado todas essas estruturas, o programa entra em um laço de repetição while, que será mais usado perto do final da execução do programa, já que no começo ele compara apenas duas posições no vetor, tendo em vista que este está virtualmente “dividido” em várias duplas pelo método MergeSort. Portanto, primeiramente será comparado cada dupla do vetor, e depois de todas ordenadas, estas serão mescladas gradativamente. A comparação dos valores é feito através de uma estrutura IF/ELSE que verifica qual dos dois valores é menor, e este é colocado em uma posição inferior no vetor temp. Feito isso o outro valor em questão é colocado em uma posição maior do vetor temp e depois estes mesmos valores são transferidos do vetor temp novamente para o vetor numOrd. Este processo será repetido diversas vezes durante a execução do programa, quanto mais dados o .txt armazenar, mais vezes os métodos serão executados. Segue a tela do código mostrando o método Merge:

```

static public void Merge(int[] numOrd, int esquerda, int mid, int direita) // Este método irá receber as variáveis esquerda e direita, que serão usadas para representar os índices dos vetores. O.o
{
    int[] temp = new int[100000];
    int i, eol, num, pos;

    eol = (mid - 1);
    pos = esquerda;
    num = (direita - esquerda + 1);

    while ((esquerda <= eol) && (mid <= direita)) // Nas próximas três estruturas while, o programa irá verificar dois números e colocá-los em uma posição inferior no vetor temp
    {
        if (numOrd[esquerda] <= numOrd[mid])
        {
            temp[pos++] = numOrd[esquerda++];
        }
        else
        {
            temp[pos++] = numOrd[mid++];
        }
    }

    while (esquerda <= eol)
        temp[pos++] = numOrd[esquerda++];

    while (mid <= direita)
        temp[pos++] = numOrd[mid++];

    for (i = 0; i < num; i++) // passa os dados ordenados do vetor temp para o vetor numOrd
    {
        numOrd[direita] = temp[direita];
        direita--;
    }
}

static public void MergeSort(int[] numOrd, int esquerda, int direita) //Este método vai "dividir" o vetor numOrd e enviar para o método Merge os valores das variáveis esquerda e direita

```

Ao final do método Merge, o programa volta automaticamente para o método MergeSort, que irá indicar quais são os índices do vetor numOrd que deverão ser ordenados agora.

O vetor temp é zerado toda vez que o método Merge começa a ser executado. Isso possibilita que os valores se aloquem temporariamente em outros índices do vetor, enquanto o mesmo não está completamente ordenado.

Feita a ordenação do vetor temp e passado para o vetor numOrd, se encerra aqui o algoritmo MergeSort. Agora começa a etapa de saída de dados, ou seja, a exibição

dos dados para o usuário. Além de simplesmente mostrar o valor contido no arquivo .txt, é necessário também ligar o valor numérico em KM² com o mês e ano correspondente. Porém esta parte do desenvolvimento será apresentada depois do método QuickSort, pois as linhas de código usadas para exibição dos dados foram basicamente idênticas.

QuickSort

Baseado no mesmo arquivo .txt, será feita a ordenação de dados usando agora o método QuickSort. É importante ressaltar que, para o usuário, não será vista nenhuma diferença, pela forma de que os dados serão ordenados de qualquer maneira. A única diferença que pode ser vista entre os dois algoritmos é no tempo de execução, e mesmo assim uma diferença muito ligeira, ainda mais com a pouca quantidade de dados utilizados.

Os procedimentos iniciais, (criação do vetor, alocação dos dados do .txt no vetor) são realizados da mesma forma que no programa anterior. Após feito esses processos é chamado o método QuickSort,

```
q_Sort.QuickSort(); //Chama o método quicksort
```

Ao chamar esse método, ele obviamente sai do método principal (main) e inicia o processo de ordenação de dados com o algoritmo QuickSort.

O método que será apresentado agora é básico e simples, porém é essencial na ordenação dos dados. Ele tem uma função parecida com a do método MergeSort do programa anterior, apenas enviar o valor dos índices que deverão ser ordenados através de variáveis nomeadas como esquerda e direita.

Uma diferença no desenvolvimento dessa aplicação, é que não tem necessidade da utilização de um vetor auxiliar, ele vai substituindo os valores no próprio vetor numOrd. Isso pode ser uma grande vantagem no tempo de execução deste programa em comparação com o algoritmo MergeSort.

Segue a tela do método QuickSort:

```
private int len;
int[] numOrd = new int[100000];
public void QuickSort()
{
    sort(0, len - 1); // chama o método sort
}
```

Nota-se que o vetor numOrd não é criado dentro do método main, exatamente pelo fato de que todos os métodos main e sort irão utilizar o mesmo vetor. Uma vez que uma variável é criada dentro de um método, ela pode ser utilizada apenas nesse método.

Inicia-se agora a ordenação em si, a parte do código que faz a troca de posicionamento entre dos índices dos vetores. Como se sabe e já foi explicado, o método de ordenação QuickSort utiliza uma posição do vetor que é usada como pivô. No programa, foi criada uma variável para receber o valor deste. Todos os

valores menores que essa variável serão visto como valores pequenos, e os maiores como valores grandes. Entender isso é o princípio para começar a desenvolver um algoritmo QuickSort.

A variável pivot, onde será alocado os valores correspondentes ao pivô, inicia-se com o primeiro valor do .txt, correspondente ao numOrd[0].

Segue a tela do método sort:

```
public void sort(int esquerda, int direita)
{
    int pivo, e, d;

    e = esquerda;
    d = direita;
    pivo = numOrd[esquerda];

    while (esquerda < direita)
    {
        while ((numOrd[direita] >= pivo) && (esquerda < direita))
        {
            direita--;
        }

        if (esquerda != direita)
        {
            numOrd[esquerda] = numOrd[direita];
            esquerda++;
        }

        while ((numOrd[esquerda] <= pivo) && (esquerda < direita))
        {
            esquerda++;
        }

        if (esquerda != direita)
        {
            numOrd[direita] = numOrd[esquerda];
            direita--;
        }
    }

    numOrd[esquerda] = pivo;
    pivo = esquerda;
    esquerda = e;
    direita = d;

    if (esquerda < pivo)
    {
        sort(esquerda, pivo - 1);
    }

    if (direita > pivo)
    {
        sort(pivo + 1, direita);
    }
}
```

Percebe-se que no final do método é chamado novamente o mesmo método, aplicando a recursividade (chamada da própria função).

Ao executar esses procedimentos, os dados que estão armazenados no .txt estão devidamente ordenados no vetor. Agora está na hora de mostrar para o usuário que os dados estão ordenados e que o programa está funcionando corretamente. Primeiramente vamos executar uma estrutura de repetição *for*, e cada execução da estrutura representa a um valor que será mostrado para o usuário. Segue o print da tela mostrando apenas o número:

```
for (int i = 0; i < max; i++)
{
    int contador = 0;
    Console.Write(i + 1 + "º: " + numOrd[i]); //Apenas mostra o vetor, após ele ter sido ordenado
```

É criada uma variável nomeada como contador, essa terá uma participação essencial para ligar o número com o mês e o ano correspondente, seguindo os dados informados pelo Imazon.

Para auxiliar neste processo de ligação, foi criado também outro vetor que deverá ser inicialmente igual ao vetor numOrd. Este recebe o nome de numOrdCopia. Porém, a ordenação acontece apenas no vetor numOrd, e o vetor numOrdCopia fica com a mesma ordem do arquivo .txt. Assim é possível ver em qual posição o valor ordenado estava antes de ter sido ordenado. Portanto, com a ajuda de uma variável nomeada contador, o numOrdCopia “percorre” o vetor ordenado até encontrar o valor idêntico, assim encontrando a posição que ficará armazenada na variável contador.

Depois de feito isso, este valor que foi encontrado no vetor numOrdCopia será substituído por -1, para que não ocorra problemas de duplicidade. Vamos começar a explicar esse processo com um exemplo:

O valor 170 corresponde à posição 29 no vetor desordenado (numOrdCopia). Quando ele é ordenado ele vai para a posição 52 do vetor numOrd. Através da seguinte linha de código ele irá percorrer o vetor desordenado até encontrar o 170, e no final da execução, a variável contador irá receber +1. Portanto, neste exemplo a variável contador ficará com o valor de 29 (posição do 170 no vetor numOrdCopia).

```
int j = 0;
while (numOrd[i] != numOrdCopia[j])
{
    contador++;
    j++;
}
numOrdCopia[j] = -1;

int cont2 = contador;
```

Como os dados estão ordenados por mês, a contagem de mês começa por abril. Portanto 0 representa abril, 1 representa maio, 2 representa junho, e assim por diante... E foi essa forma que foi usada para mostrar ao usuário qual mês que é correspondente à aquele dado. A variável contador recebe o valor de até no máximo o número de dados do arquivo. Enquanto esse número não for menor que 11, ele vai diminuindo 12, ou seja, está mantendo o mês correspondente. É importante lembrar que o que vale é o vetor desordenado, pois este está organizado por meses e anos. Portanto se no vetor desordenado o valor está na posição 29, este cálculo é feito: $29 - 12 = 17$. $17 > 11 = V$. $17 - 12 = 5$. $5 > 11 = F$. Mês correspondente é: setembro. Veja a linha de código para entender melhor:

```

while (cont2 > 11)
{
    cont2 = cont2 - 12;
}
if (cont2 == 0)
{
    Console.WriteLine(" KM² em Abril");
}
else if (cont2 == 1)
{
    Console.WriteLine(" KM² em Maio");
}
else if (cont2 == 2)
{
    Console.WriteLine(" KM² em Junho");
}
else if (cont2 == 3)
{
    Console.WriteLine(" KM² em Julho");
}
else if (cont2 == 4)
{
    Console.WriteLine(" KM² em Agosto");
}
else if (cont2 == 5)
{
    Console.WriteLine(" KM² em Setembro");
}
else if (cont2 == 6)
{
    Console.WriteLine(" KM² em Outubro");
}
else if (cont2 == 7)
{
    Console.WriteLine(" KM² em Novembro");
}
else if (cont2 == 8)
{
    Console.WriteLine(" KM² em Dezembro");
}
else if (cont2 == 9)
{
    Console.WriteLine(" KM² em Janeiro");
}
else if (cont2 == 10)
{
    Console.WriteLine(" KM² em Fevereiro");
}
else if (cont2 == 11)
{
    Console.WriteLine(" KM² em Março");
}

```

Após descoberto o mês, agora precisamos descobrir qual é o ano. Sabemos que os dados informados são a partir de abril de 2008, portanto o menor ano será 2008. A variável contador ainda será usada. Sabemos também que os 8 primeiros índices do vetor correspondem ao ano de 2008, e a partir disso, a cada 12 índices é um ano. Com base nessas informações foi criada uma variável ano, com valor inicial = 2008. Enquanto ela contador for maior que 8, ele acrescenta 1 à variável ano, e decrescente 12 à variável contador, dessa forma vamos chegar ao ano correspondente. Vamos seguir o exemplo anterior:

Ano = 2008. Contador = 29. Contador > 8 = V. Contador = contador – 12. Contador = 17. Ano = ano + 1. Ano = 2009. Contador > 8 = V. Contador = contador – 12. Contador = 5. Ano = ano + 1. Ano = 2010. Contador > 8 = F.

Ano = 2010.

Portanto, o valor 170 corresponde ao mês de setembro e ao ano de 2010. Vamos conferir no site da Imazon:

Boletim Transparência Florestal

Amazônia Legal (Setembro de 2010)

Hayashi, S., Souza Jr., C., Sales, M., & Veríssimo, A. (2010). Boletim Transparência Florestal Amazônia Legal (Setembro de 2010) (p. 15). Belém: Imazon.

[Tweeter](#) 0
 [Enviar por e-mail](#)

Em setembro de 2010, o SAD detectou **170 quilômetros quadrados** de desmatamento na Amazônia Legal. Isso representou uma redução de 21% em relação a setembro de 2009 quando o desmatamento somou 216 quilômetros quadrados.



O desmatamento acumulado no período de agosto de 2010 a setembro de 2010 totalizou 380 quilômetros quadrados. Em comparação com o período anterior de agosto 2009 a setembro 2009 quando o desmatamento somou 489 quilômetros quadrados houve redução de 22%.

Foi comprovado que a lógica apresentada é correta. Agora vamos mostrar a linha de código que calcula o ano e em seguida o print da tela do programa em execução funcionando corretamente, tanto o MergeSort quanto o QuickSort. Pode ser verificado que todos os valores batem com os meses e anos informados no site Imazon. E um detalhe que o programa também oferece é que, quando necessário informar mais algum valor, basta adicioná-lo uma linha abaixo do último valor no dados.txt. Ao fazer isso o programa já irá ordená-lo na próxima execução, e também calculará o mês e o ano correspondente (Lembrando que os dados informados são de abril de 2008 até março de 2014, sendo assim se adicionado um valor em baixo, ele irá calcular como valor referente à abril de 2014 e assim por diante).

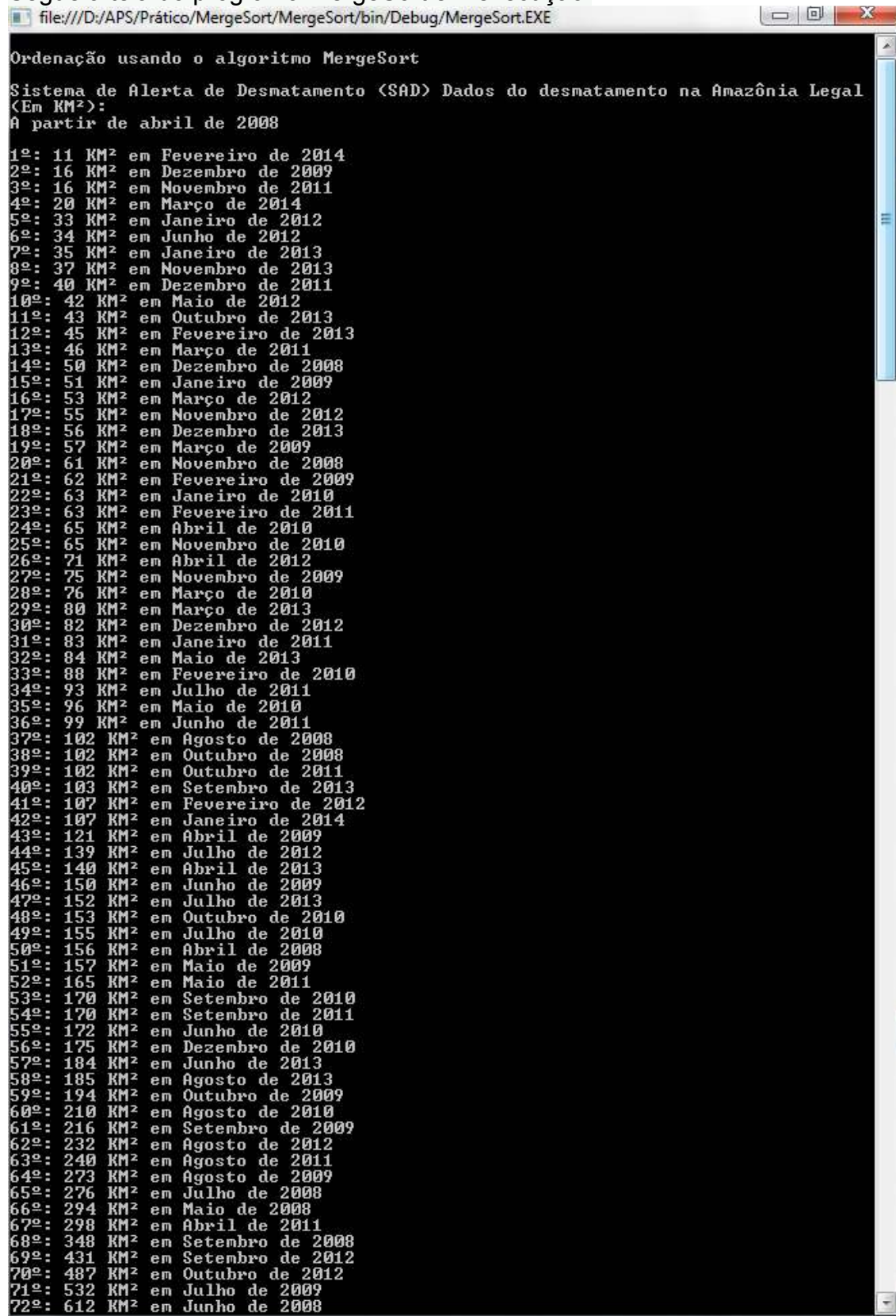
```
int ano = 2008;
while (contador > 8)
{
    ano = ano + 1;
    contador = contador - 12;
}

Console.WriteLine(" de " + ano);

Console.WriteLine("\n");
}
```


Os programas foram desenvolvidos na plataforma Visual Studio, utilizando a linguagem de programação Orientada a Objetos C#.

Segue a tela do programa MergeSort em execução:



```

file:///D:/APS/Prático/MergeSort/MergeSort/bin/Debug/MergeSort.EXE

Ordenação usando o algoritmo MergeSort

Sistema de Alerta de Desmatamento (SAD) Dados do desmatamento na Amazônia Legal
(Em KM²):
A partir de abril de 2008

1º: 11 KM² em Fevereiro de 2014
2º: 16 KM² em Dezembro de 2009
3º: 16 KM² em Novembro de 2011
4º: 20 KM² em Março de 2014
5º: 33 KM² em Janeiro de 2012
6º: 34 KM² em Junho de 2012
7º: 35 KM² em Janeiro de 2013
8º: 37 KM² em Novembro de 2013
9º: 40 KM² em Dezembro de 2011
10º: 42 KM² em Maio de 2012
11º: 43 KM² em Outubro de 2013
12º: 45 KM² em Fevereiro de 2013
13º: 46 KM² em Março de 2011
14º: 50 KM² em Dezembro de 2008
15º: 51 KM² em Janeiro de 2009
16º: 53 KM² em Março de 2012
17º: 55 KM² em Novembro de 2012
18º: 56 KM² em Dezembro de 2013
19º: 57 KM² em Março de 2009
20º: 61 KM² em Novembro de 2008
21º: 62 KM² em Fevereiro de 2009
22º: 63 KM² em Janeiro de 2010
23º: 63 KM² em Fevereiro de 2011
24º: 65 KM² em Abril de 2010
25º: 65 KM² em Novembro de 2010
26º: 71 KM² em Abril de 2012
27º: 75 KM² em Novembro de 2009
28º: 76 KM² em Março de 2010
29º: 80 KM² em Março de 2013
30º: 82 KM² em Dezembro de 2012
31º: 83 KM² em Janeiro de 2011
32º: 84 KM² em Maio de 2013
33º: 88 KM² em Fevereiro de 2010
34º: 93 KM² em Julho de 2011
35º: 96 KM² em Maio de 2010
36º: 99 KM² em Junho de 2011
37º: 102 KM² em Agosto de 2008
38º: 102 KM² em Outubro de 2008
39º: 102 KM² em Outubro de 2011
40º: 103 KM² em Setembro de 2013
41º: 107 KM² em Fevereiro de 2012
42º: 107 KM² em Janeiro de 2014
43º: 121 KM² em Abril de 2009
44º: 139 KM² em Julho de 2012
45º: 140 KM² em Abril de 2013
46º: 150 KM² em Junho de 2009
47º: 152 KM² em Julho de 2013
48º: 153 KM² em Outubro de 2010
49º: 155 KM² em Julho de 2010
50º: 156 KM² em Abril de 2008
51º: 157 KM² em Maio de 2009
52º: 165 KM² em Maio de 2011
53º: 170 KM² em Setembro de 2010
54º: 170 KM² em Setembro de 2011
55º: 172 KM² em Junho de 2010
56º: 175 KM² em Dezembro de 2010
57º: 184 KM² em Junho de 2013
58º: 185 KM² em Agosto de 2013
59º: 194 KM² em Outubro de 2009
60º: 210 KM² em Agosto de 2010
61º: 216 KM² em Setembro de 2009
62º: 232 KM² em Agosto de 2012
63º: 240 KM² em Agosto de 2011
64º: 273 KM² em Agosto de 2009
65º: 276 KM² em Julho de 2008
66º: 294 KM² em Maio de 2008
67º: 298 KM² em Abril de 2011
68º: 348 KM² em Setembro de 2008
69º: 431 KM² em Setembro de 2012
70º: 487 KM² em Outubro de 2012
71º: 532 KM² em Julho de 2009
72º: 612 KM² em Junho de 2008
  
```

Segue a tela do programa QuickSort em execução:

```
file:///D:/APS/Prático/QuickSort/QuickSort/bin/Debug/QuickSort.EXE

Ordenação usando o algoritmo QuickSort

Sistema de Alerta de Desmatamento (SAD) Dados do desmatamento na Amazônia Legal
(Em KM²):
A partir de 04/2008

1º: 11 KM² em Fevereiro de 2014
2º: 16 KM² em Dezembro de 2009
3º: 16 KM² em Novembro de 2011
4º: 20 KM² em Março de 2014
5º: 33 KM² em Janeiro de 2012
6º: 34 KM² em Junho de 2012
7º: 35 KM² em Janeiro de 2013
8º: 37 KM² em Novembro de 2013
9º: 40 KM² em Dezembro de 2011
10º: 42 KM² em Maio de 2012
11º: 43 KM² em Outubro de 2013
12º: 45 KM² em Fevereiro de 2013
13º: 46 KM² em Março de 2011
14º: 50 KM² em Dezembro de 2008
15º: 51 KM² em Janeiro de 2009
16º: 53 KM² em Março de 2012
17º: 55 KM² em Novembro de 2012
18º: 56 KM² em Dezembro de 2013
19º: 57 KM² em Março de 2009
20º: 61 KM² em Novembro de 2008
21º: 62 KM² em Fevereiro de 2009
22º: 63 KM² em Janeiro de 2010
23º: 63 KM² em Fevereiro de 2011
24º: 65 KM² em Abril de 2010
25º: 65 KM² em Novembro de 2010
26º: 71 KM² em Abril de 2012
27º: 75 KM² em Novembro de 2009
28º: 76 KM² em Março de 2010
29º: 80 KM² em Março de 2013
30º: 82 KM² em Dezembro de 2012
31º: 83 KM² em Janeiro de 2011
32º: 84 KM² em Maio de 2013
33º: 88 KM² em Fevereiro de 2010
34º: 93 KM² em Julho de 2011
35º: 96 KM² em Maio de 2010
36º: 99 KM² em Junho de 2011
37º: 102 KM² em Agosto de 2008
38º: 102 KM² em Outubro de 2008
39º: 102 KM² em Outubro de 2011
40º: 103 KM² em Setembro de 2013
41º: 107 KM² em Fevereiro de 2012
42º: 107 KM² em Janeiro de 2014
43º: 121 KM² em Abril de 2009
44º: 139 KM² em Julho de 2012
45º: 140 KM² em Abril de 2013
46º: 150 KM² em Junho de 2009
47º: 152 KM² em Julho de 2013
48º: 153 KM² em Outubro de 2010
49º: 155 KM² em Julho de 2010
50º: 156 KM² em Abril de 2008
51º: 157 KM² em Maio de 2009
52º: 165 KM² em Maio de 2011
53º: 170 KM² em Setembro de 2010
54º: 170 KM² em Setembro de 2011
55º: 172 KM² em Junho de 2010
56º: 175 KM² em Dezembro de 2010
57º: 184 KM² em Junho de 2013
58º: 185 KM² em Agosto de 2013
59º: 194 KM² em Outubro de 2009
60º: 210 KM² em Agosto de 2010
61º: 216 KM² em Setembro de 2009
62º: 232 KM² em Agosto de 2012
63º: 240 KM² em Agosto de 2011
64º: 273 KM² em Agosto de 2009
65º: 276 KM² em Julho de 2008
66º: 294 KM² em Maio de 2008
67º: 298 KM² em Abril de 2011
68º: 348 KM² em Setembro de 2008
69º: 431 KM² em Setembro de 2012
70º: 487 KM² em Outubro de 2012
71º: 532 KM² em Julho de 2009
72º: 612 KM² em Junho de 2008
```


Resultados e Discussão

Ordenar um conjunto de itens em uma lista é uma tarefa frequente na programação. Muitas vezes, um ser humano pode realizar essa tarefa intuitivamente. No entanto, um programa de computador precisa seguir uma sequência exata de instruções para completá-la, e essa sequência é chamada algoritmo. Um algoritmo de ordenação é um método utilizado para colocar uma lista de itens desorganizados em uma determinada ordem. A sequência da ordenação é determinada por uma chave. Existem vários algoritmos de ordenação que diferem em termos de eficiência e desempenho.

Quicksort é uma versão otimizada de uma árvore binária ordenada. Em vez de introduzir itens sequencialmente numa árvore explícita, o Quicksort organiza-os correntemente na árvore onde está implícito, fazendo-o com chamadas recursivas à mesma. O algoritmo faz exatamente as mesmas comparações, mas com uma ordem diferente.

O algoritmo que mais se familiariza com o Quicksort é o Heapsort. Mas, o Heapsort em média trata-se de um algoritmo mais lento que o Quicksort, embora tenha sido muito debatido essa afirmação. No Quicksort, à exceção quando se trata de usar a variante Intro sort, que muda para Heapsort quando um pior caso é detectado. Caso se saiba à partida que será necessário o uso do heapsort é aconselhável usá-lo diretamente, do que usar o introsort e depois chamar o heapsort, torna mais rápido o algoritmo.

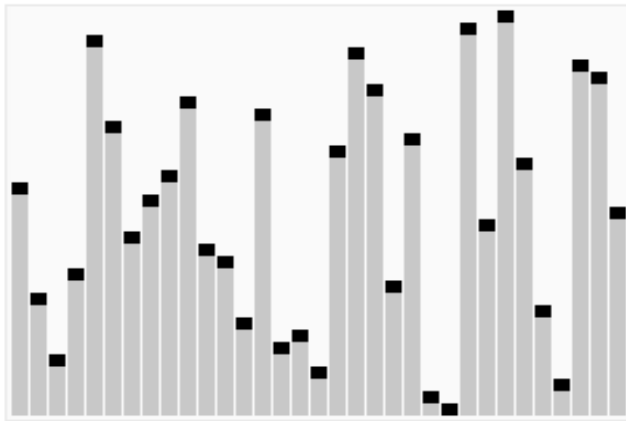


Gráfico de ordenação do quicksort

O Mergesort é classificado como ordenação por partição, que parte do princípio de "dividir para conquistar". Este princípio é uma técnica que foi utilizada pela primeira vez por Anatolii Karatsuba em 1960 e consiste em dividir um problema maior em problemas pequenos, e sucessivamente até que o mesmo seja resolvido diretamente.

Esta técnica realiza-se em três fases:

Divisão: o problema maior é dividido em problemas menores e os problemas menores obtidos são novamente divididos sucessivamente de maneira recursiva.

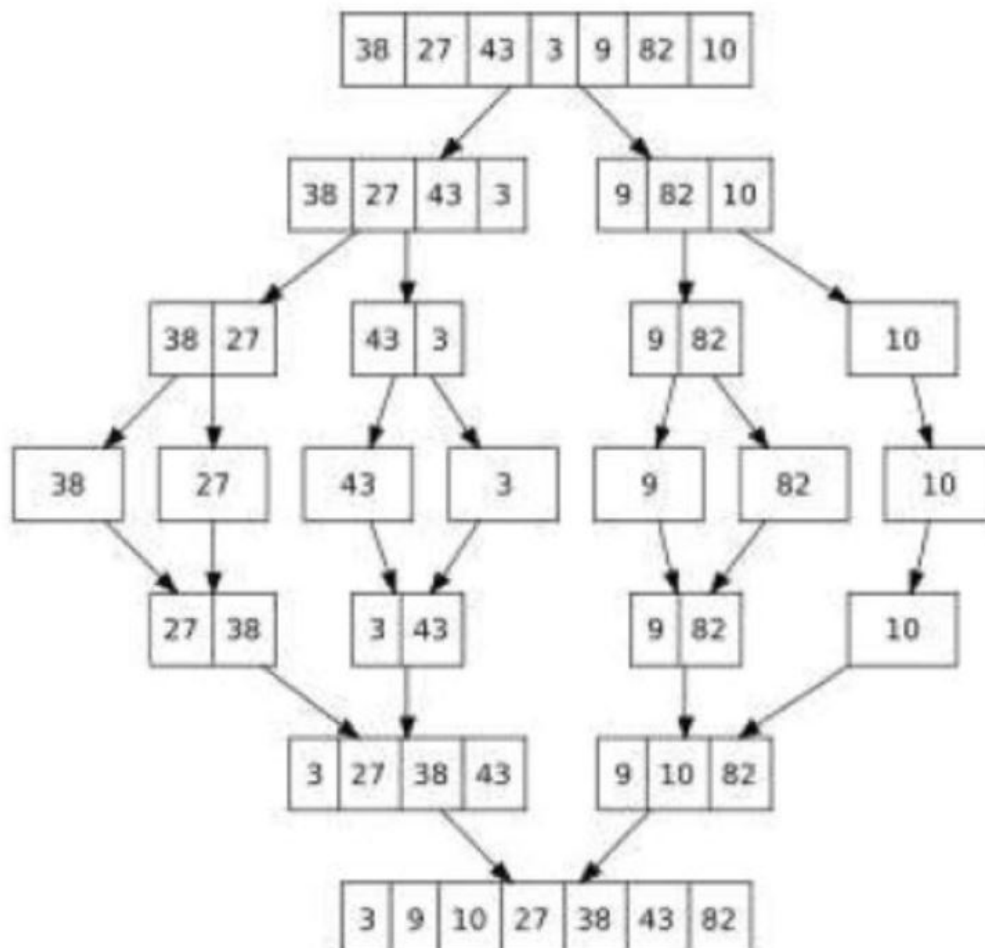
Conquista: o resultado do problema é calculado quando o problema é pequeno o suficiente.

Combinação: os resultados dos problemas menores são combinados até que seja obtida a solução do problema maior. Algoritmos que utilizam o método de partição são caracterizados por serem os mais rápidos dentre os outros algoritmos pelo fato de sua complexidade ser, na maioria das situações, $O(n \log n)$. Os dois representantes mais ilustres desta classe são o *quicksort* e o *mergesort*.

- Não é um método in-place

Em ciência da computação, um algoritmo in-place é um algoritmo que transforma a entrada de informação usando Estrutura de Dados com uma pequena e constante

quantidade de espaço de memória extra. A entrada de informação geralmente é sobrescrita por uma saída de dados como o algoritmo executa. Um algoritmo que não é in-place, no caso, do Merge Sort, é chamado de out-of-place.



Testes e comparações

Quantidade de Comparações				
	100	1.000	10.000	100.000
QuickSort	997	12.852	181.203	2.114.943
MergeSort	558	8.744	123.685	1.566.749

Quantidade de Movimentos				
	100	1.000	10.000	100.000
QuickSort	570	8.136	103.575	1.310.586
MergeSort	1376	19968	272640	3385984

Tempo de execução (s)				
	100	1.000	10.000	100.000
QuickSort	0,00003	0,0004	0,0049	0,0844
MergeSort	0,00015	0,0016	0,0194	0,2316

CONSIDERAÇÕES FINAIS

Ordenação de algoritmo consiste em pegar dados aleatórios ou semi-ordenados e os coloca em ordem crescente independente do tamanho do valor. Os métodos utilizados são um tanto complexos, porém eficazes em sua função como ordenar dados mesmo sendo quase idênticos no processo. Um deles utilizando o Pivô que serve como o ponto médio entre o vetor (QuickSort), e o outro dividindo o vetor pela metade até separar todos os elementos (MergeSort).

O método mais eficaz entre os dois, é o QuickSort onde em um vetor de 100 elementos ele faz mais comparações, (fazendo assim o programa ter mais certeza da ordenação apresentando menos falhas) 997 ,contra 558 do MergeSort, o mesmo também faz menos movimentos 570, contra 1376 do MergeSort, e por si só o tempo de execução varia muito em questão de programas onde executa a ordenação de um vetor de 100 elementos em apenas 0,00003 s, contra 0,00015 s do MergeSort.

Chegamos à conclusão que o método de ordenação QuickSort tem mais vantagens em relação a ordenar dados de um banco de dados por exemplo, e pode ser aplicado sem receio nenhum em um programa, lembrando que quanto maior o vetor mais tempo demora a ordenação, porém o tamanho não é condicional a velocidade de execução pois a velocidade de execução aumenta em algo consideravelmente insignificante em relação ao processo de ordenação.

Bibliografia

<http://www.devmedia.com.br/algoritmos-de-ordenacao-analise-e-comparacao/28261#ixzz32B3kKzHK>
<http://imasters.com.br/artigo/13015/desenvolvimento/principais-algoritmos-de-ordenacao-de-dados/>
http://pt.slideshare.net/line_vm/quick-sort-19987521
<http://www.dei.isep.ipp.pt/~i960231/ei/quicksort.htm>:
<http://www.academia.edu>
<http://www.devmedia.com.br/algoritmos-de-ordenacao-analise-e-comparacao/28261>
http://www.imazon.org.br/publicacoes/transparencia-florestal/transparencia-florestal-amazonia-legal?b_start:int=0

Código Fonte

MergeSort

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.IO;

namespace MergeSort
{
    class Program
    {
        static public void Merge(int[] numOrd, int esquerda, int mid, int direita)//
        Este método irá receber as variáveis esquerda e direita, que serão usadas para
        representar os índices dos vetores. Os valores que correspondem a estes índices, serão
        ordenados entre eles próprios. Ou seja, no começo ele irá dividir o vetor em várias
        partes de 2 índices cada, ordena essas partes, e depois junta com as outras partes,
        ordenando também e assim gradativamente
        {
            int[] temp = new int[100000];
            int i, eol, num, pos;

            eol = (mid - 1);
            pos = esquerda;
            num = (direita - esquerda + 1);

            while ((esquerda <= eol) && (mid <= direita)) //Nas próximas três
            estruturas While, o programa irá verificar dois números e coloca o menor deles em uma
            posição inferior no vetor temp
            {
                if (numOrd[esquerda] <= numOrd[mid])
                {
                    temp[pos++] = numOrd[esquerda++];
                }
                else
                {
                    temp[pos++] = numOrd[mid++];
                }
            }

            while (esquerda <= eol)
                temp[pos++] = numOrd[esquerda++];

            while (mid <= direita)
                temp[pos++] = numOrd[mid++];

            for (i = 0; i < num; i++) // passa os dados ordenados do vetor temp para o
            vetor numOrd
            {
                numOrd[direita] = temp[direita];
            }
        }
    }
}
```

```

        direita--;
    }
}

static public void MergeSort(int[] numOrd, int esquerda, int direita) //Este
método vai "dividir" o vetor numOrd e enviar para o método Merge os valores das
variáveis esquerda e direita
{
    int mid;

    if (direita > esquerda)
    {
        mid = (direita + esquerda) / 2;

        MergeSort(numOrd, esquerda, mid);
        MergeSort(numOrd, (mid + 1), direita);

        Merge(numOrd, esquerda, (mid + 1), direita);
    }
}

static void Main(string[] args)
{
    int max = 0;
    string line;

    StreamReader file = new StreamReader(@"dados.txt"); //Importando o
arquivo txt
    Console.WriteLine("\nOrdenação usando o algoritmo MergeSort\n\nSistema de
Alerta de Desmatamento (SAD) Dados do desmatamento na Amazônia Legal (Em KM²): ");
    while ((line = file.ReadLine()) != null) //Fazendo a contagem de elementos
que o arquivo txt possui, linha por linha
    {
        max++;
    }

    file.Close();// Fecha o arquivo txt

    int[] numOrd = new int[max]; //Declara o vetor, com o índice igual ao
número de elementos do arquivo txt
    int[] numOrdCopia = new int[max];
    StreamReader file2 = new StreamReader(@"dados.txt"); //Importa novamente
o arquivo txt
    for (int i = 0; i < max; i++)
    {
        line = file2.ReadLine();
        numOrd[i] = Convert.ToInt32(line); //Colocando no vetor todos os
elementos do arquivo txt, um índice por linha
    }

    for (int i = 0; i < max; i++)
    {
        numOrdCopia[i] = numOrd[i]; //Colocando no vetor todos os elementos do
arquivo txt, um índice por linha
    }

    Console.WriteLine("\nA partir de abril de 2008\n");
}

```

MergeSort(numOrd, 0, max - 1); //Chama o método MergeSort que fará a ordenação dos dados

```

for (int i = 0; i < max; i++)
{
    int contador = 0;
    Console.WriteLine(i + 1 + "º: " + numOrd[i]); //Apenas mostra o vetor, após
ele ter sido ordenado
    int j = 0;
    while (numOrd[i] != numOrdCopia[j])
    {
        contador++;
        j++;
    }
    numOrdCopia[j] = 0;
    int cont2 = contador;

    while (cont2 > 11)
    {
        cont2 = cont2 - 12;
    }
    if (cont2 == 0 )
    {
        Console.WriteLine(" KM² em Abril");
    }
    else if (cont2 == 1)
    {
        Console.WriteLine(" KM² em Maio");
    }
    else if (cont2 == 2)
    {
        Console.WriteLine(" KM² em Junho");
    }
    else if (cont2 == 3)
    {
        Console.WriteLine(" KM² em Julho");
    }
    else if (cont2 == 4)
    {
        Console.WriteLine(" KM² em Agosto");
    }
    else if (cont2 == 5)
    {
        Console.WriteLine(" KM² em Setembro");
    }
    else if (cont2 == 6)
    {
        Console.WriteLine(" KM² em Outubro");
    }
    else if (cont2 == 7)
    {
        Console.WriteLine(" KM² em Novembro");
    }
    else if (cont2 == 8)
    {
        Console.WriteLine(" KM² em Dezembro");
    }
    else if (cont2 == 9)
    {

```



```

        Console.WriteLine(" KM² em Janeiro");
    }
    else if (cont2 == 10)
    {
        Console.WriteLine(" KM² em Fevereiro");
    }
    else if (cont2 == 11)
    {
        Console.WriteLine(" KM² em Março");
    }

    int ano = 2008;
    while (contador > 8)
    {
        ano = ano + 1;
        contador = contador - 12;
    }

    Console.WriteLine(" de " + ano);

    Console.WriteLine("\n");
}

        Console.WriteLine("\n\nAperte qualquer tecla para encerrar o
programa...");
        Console.ReadKey();
    }
}

```

QuickSort

```

using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.IO;

namespace sortQuick
{
    class quickSort
    {

        private int len;
        int[] numOrd = new int[100000];
        public void QuickSort()
        {
            sort(0, len - 1); // chama o método sort
        }

        public void sort(int esquerda, int direita)
        {

            int pivo, e, d;

            e = esquerda;
            d = direita;
            pivo = numOrd[esquerda];

            while (esquerda < direita)
            {
                while ((numOrd[direita] >= pivo) && (esquerda < direita))
                {
                    direita--;
                }

                if (esquerda != direita)
                {
                    numOrd[esquerda] = numOrd[direita];
                    esquerda++;
                }

                while ((numOrd[esquerda] <= pivo) && (esquerda < direita))
                {
                    esquerda++;
                }

                if (esquerda != direita)
                {
                    numOrd[direita] = numOrd[esquerda];
                    direita--;
                }
            }

            numOrd[esquerda] = pivo;
            pivo = esquerda;
            esquerda = e;
            direita = d;

            if (esquerda < pivo)

```

```

        {
            sort(esquerda, pivo - 1);
        }

        if (direita > pivo)
        {
            sort(pivo + 1, direita);
        }
    }

    public static void Main()
    {
        quickSort q_Sort = new quickSort();//Método construtor

        int max = 0;
        string line;

        StreamReader file = new StreamReader(@"\dados.txt"); //Importando o
arquivo txt
        while ((line = file.ReadLine()) != null) //Fazendo a contagem de elementos
que o arquivo txt possui, linha por linha
        {
            max++;
        }

        file.Close();// Fecha o arquivo txt

        int[] numOrd = new int[max]; //Declara o vetor, com o índice igual ao
número de elementos do arquivo txt

        StreamReader file2 = new StreamReader(@"\dados.txt"); //Importa novamente
o arquivo txt
        for (int i = 0; i < max; i++)
        {
            line = file2.ReadLine();
            numOrd[i] = Convert.ToInt32(line); //Colocando no vetor todos os
elementos do arquivo txt, um índice por linha
        }

        int[] numOrdCopia = new int[max];

        for (int i = 0; i < max; i++)
        {
            numOrdCopia[i] = numOrd[i]; //Colocando no vetor todos os elementos do
arquivo txt, um índice por linha
        }

        q_Sort.numOrd = numOrd;
        q_Sort.len = max;

        q_Sort.QuickSort(); //Chama o método quicksort

        Console.WriteLine("\nOrdenação usando o algoritmo QuickSort\n\nSistema de
Alerta de Desmatamento (SAD) Dados do desmatamento na Amazônia Legal (Em KM²): ");
    }

```

```

Console.WriteLine("\nA partir de 04/2008\n");
for (int i = 0; i < max; i++)
{
    int contador = 0;
    Console.Write(i + 1 + "º: " + numOrd[i]); //Apenas mostra o vetor, após
ele ter sido ordenado
    int j = 0;
    while (numOrd[i] != numOrdCopia[j])
    {
        contador++;
        j++;
    }
    numOrdCopia[j] = -1;

    int cont2 = contador;

    while (cont2 > 11)
    {
        cont2 = cont2 - 12;
    }
    if (cont2 == 0)
    {
        Console.Write(" KM² em Abril");
    }
    else if (cont2 == 1)
    {
        Console.Write(" KM² em Maio");
    }
    else if (cont2 == 2)
    {
        Console.Write(" KM² em Junho");
    }
    else if (cont2 == 3)
    {
        Console.Write(" KM² em Julho");
    }
    else if (cont2 == 4)
    {
        Console.Write(" KM² em Agosto");
    }
    else if (cont2 == 5)
    {
        Console.Write(" KM² em Setembro");
    }
    else if (cont2 == 6)
    {
        Console.Write(" KM² em Outubro");
    }
    else if (cont2 == 7)
    {
        Console.Write(" KM² em Novembro");
    }
    else if (cont2 == 8)
    {
        Console.Write(" KM² em Dezembro");
    }
    else if (cont2 == 9)
    {
        Console.Write(" KM² em Janeiro");
    }
    else if (cont2 == 10)

```

```

    {
        Console.WriteLine(" KM² em Fevereiro");
    }
    else if (cont2 == 11)
    {
        Console.WriteLine(" KM² em Março");
    }

    int ano = 2008;
    while (contador > 8)
    {
        ano = ano + 1;
        contador = contador - 12;
    }

    Console.WriteLine(" de " + ano);


    Console.WriteLine("\n");
}

    Console.WriteLine("\n\nAperte qualquer tecla para encerrar o
programa...");

    Console.ReadKey();
}
}
}

```

FICHA DE ATIVIDADES PRÁTICAS SUPERVISIONADA



FICHA DAS ATIVIDADES PRÁTICAS SUPERVISIONADAS - APS

NOME: JOHNNY AMANCIO SANTOS

CURSO: 13701 - CIÊNCIA DA COMPUTAÇÃO

CÓDIGO DA ATIVIDADE: 504Z - ATIVIDADES PRÁTICAS SUPERVISIONADAS

TURMA: CC3P41

CAMPUS: RNG - SANTOS - RANGEL

SEMESTRE: 3º

RA: B633FF-8

TURNO: NOITE

ANO GRADE: 2014/1

DATA DA ATIVIDADE	DESCRIÇÃO DA ATIVIDADE	TOTAL DE HORAS	ASSINATURA DO ALUNO	HORAS ATRIBUÍDAS (1)	ASSINATURA DO PROFESSOR
15/04/2014	Reunião para discutir a forma que o trabalho será feito	02:00	Johnny Amancio Santos		
17/04/2014	Pesquisa em sites sobre os métodos de ordenação atribuídos	07:00			
19/04/2014	Divisão de tarefas entre os participantes do grupo	02:00			
19/04/2014	Estudo para definir qual o software que será usado para desenvolvimento da aplicação	02:00			
21/04/2014	Desenvolvimento da aplicação, usando o Visual Studio C# Express 2010	03:00			
22/04/2014	Desenvolvimento da aplicação, usando o Visual Studio C# Express 2010	03:00			
25/04/2014	Desenvolvimento da aplicação, usando o Visual Studio C# Express 2010	03:00			
26/04/2014	Desenvolvimento da aplicação, usando o Visual Studio C# Express 2010	03:00			
27/04/2014	Desenvolvimento da aplicação, usando o Visual Studio C# Express 2010	03:00			
28/04/2014	Desenvolvimento da aplicação, usando o Visual Studio C# Express 2010	03:00			
05/05/2014	Desenvolvimento da aplicação, usando o Visual Studio C# Express 2010	03:00			
06/05/2014	Desenvolvimento da aplicação, usando o Visual Studio C# Express 2010	03:00			
09/05/2014	Reunião para finalização e últimos ajustes sobre a aplicação desenvolvida	03:00			
13/05/2014	Reunião para mostrar, na parte teórica como funciona a lógica desses métodos de ordenação	04:00			
16/05/2014	Reunião para mostrar, na parte teórica como funciona a lógica desses métodos de ordenação	04:00			
17/05/2014	Conclusão da atividade	02:00			

(1) Horas atribuídas de acordo com o regulamento das Atividades Práticas Supervisionadas do curso.

TOTAL DE HORAS ATRIBUÍDAS: 50 Horas

AVALIAÇÃO: _____
Aprovado ou Reprovado

NOTA: _____

DATA: ____/____/____

CARIMBO E ASSINATURA DO COORDENADOR DO CURSO



FICHA DAS ATIVIDADES PRÁTICAS SUPERVISIONADAS - APS

NOME: MATHEUS GONÇALVES BRANDÃOTURMA: CC3Q41RA: B67712-0CURSO: 13701 - CIÊNCIA DA COMPUTAÇÃOCAMPUS: RNG - SANTOS - RANGELTURNO: NOITECÓDIGO DA
ATIVIDADE: 504Z - ATIVIDADES PRÁTICAS SUPERVISIONADASSEMESTRE: 3ºANO GRADE: 2014/1

DATA DA ATIVIDADE	DESCRIÇÃO DA ATIVIDADE	TOTAL DE HORAS	ASSINATURA DO ALUNO	HORAS ATRIBUÍDAS (1)	ASSINATURA DO PROFESSOR
15/04/2014	Reunião para discutir a forma que o trabalho será feito	02:00	Matheus Gonçalves Brandão		
17/04/2014	Pesquisa em sites sobre os métodos de ordenação atribuídos	07:00			
19/04/2014	Divisão de tarefas entre os participantes do grupo	02:00			
19/04/2014	Estudo para definir qual o software que será usado para desenvolvimento da aplicação	02:00			
21/04/2014	Desenvolvimento da aplicação, usando o Visual Studio C# Express 2010	03:00			
22/04/2014	Desenvolvimento da aplicação, usando o Visual Studio C# Express 2010	03:00			
25/04/2014	Desenvolvimento da aplicação, usando o Visual Studio C# Express 2010	03:00			
26/04/2014	Desenvolvimento da aplicação, usando o Visual Studio C# Express 2010	03:00			
27/04/2014	Desenvolvimento da aplicação, usando o Visual Studio C# Express 2010	03:00			
28/04/2014	Desenvolvimento da aplicação, usando o Visual Studio C# Express 2010	03:00			
05/05/2014	Desenvolvimento da aplicação, usando o Visual Studio C# Express 2010	03:00			
06/05/2014	Desenvolvimento da aplicação, usando o Visual Studio C# Express 2010	03:00			
09/05/2014	Reunião para finalização e últimos ajustes sobre a aplicação desenvolvida	03:00			
13/05/2014	Reunião para mostrar, na parte teórica como funciona a lógica desses métodos de ordenação	04:00			
16/05/2014	Reunião para mostrar, na parte teórica como funciona a lógica desses métodos de ordenação	04:00			
17/05/2014	Conclusão da atividade	02:00			

(1) Horas atribuídas de acordo com o regulamento das Atividades Práticas Supervisionadas do curso.

TOTAL DE HORAS ATRIBUÍDAS: 50 Horas

AVALIAÇÃO: _____

Aprovado ou Reprovado

NOTA: _____

DATA: ____/____/____

CARIMBO E ASSINATURA DO COORDENADOR DO CURSO



FICHA DAS ATIVIDADES PRÁTICAS SUPERVISIONADAS - APS

NOME: MATHEUS FELIPE DOS PASSOS E PAZTURMA: CC3Q41RA: B57IAJ-0CURSO: 13701 - CIÊNCIA DA COMPUTAÇÃOCAMPUS: RNG - SANTOS - RANGELTURNO: NOITECÓDIGO DA
ATIVIDADE: 504Z - ATIVIDADES PRÁTICAS SUPERVISIONADASSEMESTRE: 3ºANO GRADE: 2014/1

DATA DA ATIVIDADE	DESCRIÇÃO DA ATIVIDADE	TOTAL DE HORAS	ASSINATURA DO ALUNO	HORAS ATRIBUÍDAS (1)	ASSINATURA DO PROFESSOR
15/04/2014	Reunião para discutir a forma que o trabalho será feito	02:00	Matheus Felipe dos Passos e Paz		
17/04/2014	Pesquisa em sites sobre os métodos de ordenação atribuídos	07:00			
19/04/2014	Divisão de tarefas entre os participantes do grupo	02:00			
19/04/2014	Estudo para definir qual o software que será usado para desenvolvimento da aplicação	02:00			
21/04/2014	Desenvolvimento da aplicação, usando o Visual Studio C# Express 2010	03:00			
22/04/2014	Desenvolvimento da aplicação, usando o Visual Studio C# Express 2010	03:00			
25/04/2014	Desenvolvimento da aplicação, usando o Visual Studio C# Express 2010	03:00			
26/04/2014	Desenvolvimento da aplicação, usando o Visual Studio C# Express 2010	03:00			
27/04/2014	Desenvolvimento da aplicação, usando o Visual Studio C# Express 2010	03:00			
28/04/2014	Desenvolvimento da aplicação, usando o Visual Studio C# Express 2010	03:00			
05/05/2014	Desenvolvimento da aplicação, usando o Visual Studio C# Express 2010	03:00			
06/05/2014	Desenvolvimento da aplicação, usando o Visual Studio C# Express 2010	03:00			
09/05/2014	Reunião para finalização e últimos ajustes sobre a aplicação desenvolvida	03:00			
13/05/2014	Reunião para mostrar, na parte teórica como funciona a lógica desses métodos de ordenação	04:00			
16/05/2014	Reunião para mostrar, na parte teórica como funciona a lógica desses métodos de ordenação	04:00			
17/05/2014	Conclusão da atividade	02:00			

(1) Horas atribuídas de acordo com o regulamento das Atividades Práticas Supervisionadas do curso.

TOTAL DE HORAS ATRIBUÍDAS: 50 Horas

AVALIAÇÃO: _____

Aprovado ou Reprovado

NOTA: _____

DATA: ____/____/____

CARIMBO E ASSINATURA DO COORDENADOR DO CURSO



FICHA DAS ATIVIDADES PRÁTICAS SUPERVISIONADAS - APS

NOME: MATHEUS RODRIGUES MARTINS PEREIRATURMA: CC3Q41RA: B73ABD-5CURSO: 13701 - CIÊNCIA DA COMPUTAÇÃOCAMPUS: RNG - SANTOS - RANGELTURNO: NOITECÓDIGO DA
ATIVIDADE: 504Z - ATIVIDADES PRÁTICAS SUPERVISIONADASSEMESTRE: 3ºANO GRADE: 2014/1

DATA DA ATIVIDADE	DESCRIÇÃO DA ATIVIDADE	TOTAL DE HORAS	ASSINATURA DO ALUNO	HORAS ATRIBUÍDAS (1)	ASSINATURA DO PROFESSOR
15/04/2014	Reunião para discutir a forma que o trabalho será feito	02:00	Matheus Rodrigues Martins Pereira		
17/04/2014	Pesquisa em sites sobre os métodos de ordenação atribuídos	07:00			
19/04/2014	Divisão de tarefas entre os participantes do grupo	02:00			
19/04/2014	Estudo para definir qual o software que será usado para desenvolvimento da aplicação	02:00			
21/04/2014	Desenvolvimento da aplicação, usando o Visual Studio C# Express 2010	03:00			
22/04/2014	Desenvolvimento da aplicação, usando o Visual Studio C# Express 2010	03:00			
25/04/2014	Desenvolvimento da aplicação, usando o Visual Studio C# Express 2010	03:00			
26/04/2014	Desenvolvimento da aplicação, usando o Visual Studio C# Express 2010	03:00			
27/04/2014	Desenvolvimento da aplicação, usando o Visual Studio C# Express 2010	03:00			
28/04/2014	Desenvolvimento da aplicação, usando o Visual Studio C# Express 2010	03:00			
05/05/2014	Desenvolvimento da aplicação, usando o Visual Studio C# Express 2010	03:00			
06/05/2014	Desenvolvimento da aplicação, usando o Visual Studio C# Express 2010	03:00			
09/05/2014	Reunião para finalização e últimos ajustes sobre a aplicação desenvolvida	03:00			
13/05/2014	Reunião para mostrar, na parte teórica como funciona a lógica desses métodos de ordenação	04:00			
16/05/2014	Reunião para mostrar, na parte teórica como funciona a lógica desses métodos de ordenação	04:00			
17/05/2014	Conclusão da atividade	02:00			

(1) Horas atribuídas de acordo com o regulamento das Atividades Práticas Supervisionadas do curso.

TOTAL DE HORAS ATRIBUÍDAS: 50 Horas

AVALIAÇÃO: _____

Aprovado ou Reprovado

NOTA: _____

DATA: ____/____/____

CARIMBO E ASSINATURA DO COORDENADOR DO CURSO