INF1010 - ESTRUTURAS DE DADOS AVANCADAS - 2022.1 - 3WB

Lab1 - Complexidade de Algoritmos

Nome: Eric Leão Matrícula: 2110694
Nome: Marina Schuler Martins Matrícula: 2110075

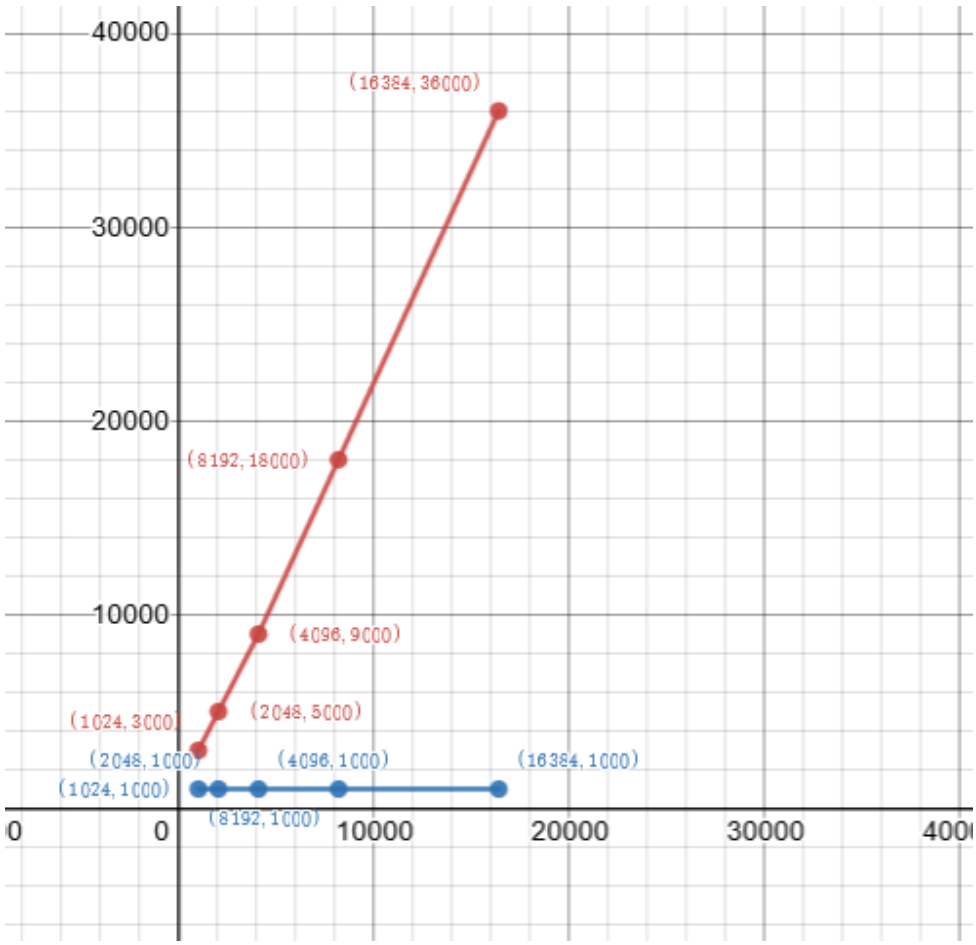
Código fonte:

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <unistd.h>
#define TAM 1024
#define MAX 18000
#define TESTE 18001

int buscaVetor(int *v, int i, int tam){
  int c;
  for (c=0;c<tam;c++)
    if (v[c]==i)
      return 1;
  return 0;
}

int buscaBin(int*v, int b, int tam){
  int in = 0, fim = tam -1;
  int meio;
  while(in<=fim){
    meio = (in + fim)/2;
    if(b<v[meio])fim=meio-1;
    if(b>v[meio]) in=meio+1;
    else return 1;
  }return 0;
}

void criaVetor(int*v, int tam){
  int i;
  for(int k=0; k<tam; k++){
    i = rand()%MAX;
    if(buscaVetor(v, i, k+1))
      k--;
    else
      v[k] = i;
  }
}

void ordenaVetor(int *v, int tam){
```

```
  int i,j,aux;
  for (i=0;i<tam;i++)
    for (j=0;j<tam;j++)
      if(v[j+1]<v[j]){
        aux = v[j];
        v[j]= v[j+1];
        v[j+1]=aux;
        }
}

int main(void) {
  int vet1[TAM];
  int vet2[2*TAM];
  int vet3[4*TAM];
  int vet4[8*TAM];
  int vet5[16*TAM];
  clock_t inicio, fim;
  double total;

  criaVetor(vet1, TAM);
  ordenaVetor(vet1, TAM);
  criaVetor(vet2, 2*TAM);
  ordenaVetor(vet2, 2*TAM);
  criaVetor(vet3, 4*TAM);
  ordenaVetor(vet3, 4*TAM);
  criaVetor(vet4, 8*TAM);
  ordenaVetor(vet4, 8*TAM);
  criaVetor(vet5, 16*TAM);
  ordenaVetor(vet5, 16*TAM);

  printf("Busca Sequencial: \n");
  /*busca sequencial*/
  inicio = clock();
  buscaVetor(vet1, TESTE, TAM);
  fim = clock();
  total = (double) (fim - inicio) / CLOCKS_PER_SEC;
  printf("Tempo total 1K = %f\n",total);

  inicio = clock();
  buscaVetor(vet2, TESTE, 2*TAM);
  fim = clock();
  total = (double) (fim - inicio) / CLOCKS_PER_SEC;
  printf("Tempo total 2K = %f\n",total);

  inicio = clock();
  buscaVetor(vet3, TESTE, 4*TAM);
  fim = clock();
  total = (double) (fim - inicio) / CLOCKS_PER_SEC;
```

```
  printf("Tempo total 4K = %f\n",total);

  inicio = clock();
  buscaVetor(vet4, TESTE, 8*TAM);
  fim = clock();
  total = (double) (fim - inicio) / CLOCKS_PER_SEC;
  printf("Tempo total 8K = %f\n",total);

  inicio = clock();
  buscaVetor(vet5, TESTE, 16*TAM);
  fim = clock();
  total = (double) (fim - inicio) / CLOCKS_PER_SEC;
  printf("Tempo total 16K = %f\n",total);

  printf("Busca Binária: \n");
  /*busca binária*/
  inicio = clock();
  buscaBin(vet1, TESTE, TAM);
  fim = clock();
  total = (double) (fim - inicio) / CLOCKS_PER_SEC;
  printf("Tempo total 1K = %f\n",total);

  inicio = clock();
  buscaBin(vet2, TESTE, 2*TAM);
  fim = clock();
  total = (double) (fim - inicio) / CLOCKS_PER_SEC;
  printf("Tempo total 2K = %f\n",total);

  inicio = clock();
  buscaBin(vet3, TESTE, 4*TAM);
  fim = clock();
  total = (double) (fim - inicio) / CLOCKS_PER_SEC;
  printf("Tempo total 4K = %f\n",total);

  inicio = clock();
  buscaBin(vet4, TESTE, 8*TAM);
  fim = clock();
  total = (double) (fim - inicio) / CLOCKS_PER_SEC;
  printf("Tempo total 8K = %f\n",total);

  inicio = clock();
  buscaBin(vet5, TESTE, 16*TAM);
  fim = clock();
  total = (double) (fim - inicio) / CLOCKS_PER_SEC;
  printf("Tempo total 16K = %f\n",total);

  return 0;
}
```

Gráfico gerado:



Legenda:

Em vermelho está a função de busca sequencial
Em azul está a função de busca binária
O eixo y representa o tamanho do vetor
O eixo x representa o tempo de execução em segundos vezes 10^9

Como pode ser observado, o gráfico da busca sequencial aumenta dobrando o tempo a cada vez que dobramos o número de inteiros dentro do vetor, isso ocorre, pois, o pior caso da busca sequencial é passando por todo vetor n, tendo uma complexidade O(n). Por conta disso, se dobrarmos o tamanho do vetor, dobramos o tempo que demora para percorrê-lo. Em contrapartida, o tempo da busca binária aumenta de forma quase imperceptível, de forma que todos os valores do tempo foram "iguais". O tempo decorrido não é, de fato, igual, contudo por ser uma diferença tão pequena, todos os valores acabaram sendo arredondados para o mesmo, o 0.00001 segundos. Isso ocorre, pois na busca binária o pior caso é quando precisamos dividir n por 2 até chegarmos no número 1. A quantidade de vezes que precisamos fazer tal divisão pode ser representada por log2(n). Isso faz com que o código tenha complexidade O(log(n)), o que significa que sempre que dobrarmos o tamanho do vetor, precisaremos fazer apenas mais 1 teste para descobrir onde o número está.