// **Código da main.c**

```
random.h veem da biblioteca de
// Eric Roberts, a#include <stdio.h>
#include <stdlib.h>
#include "simpio.h"
#include "random.h"

#define TAM_INI 1000
#define TAM_MAX 10000
#define INC 200

// simpio.h e cessível por exemplo em
// https://www.ime.usp.br/~pf/Roberts/C-library/standard/

typedef int *veti; // Apelida de "veti" ("vetor de inteiros") o tipo "int *".

veti criaVeti(int);
int busca (int, int, veti);
int maximo (int, veti);
int twosum (int, int, veti);

void insertionSort(int, veti);
int buscaBinaria(int, int, veti);

int NOP;

int main() {

   int tamanho, chave, i, r;
   veti vetor;

   Randomize();

   FILE *saida;

   // Mude o nome do arquivo de saída
   saida = fopen ("dadosBuscaLinear.dat", "w");

   for(tamanho = TAM_INI; tamanho <= TAM_MAX; tamanho += INC) {
      vetor = criaVeti(tamanho);

      for(i = 0; i < tamanho; i++)
         vetor[i] = RandomInteger(0, TAM_MAX);

      // Sorteia uma chave existente no vetor
      chave = vetor[RandomInteger(0, tamanho)];
```

```
        // r = maximo(tamanho, vetor);
        // r = twosum(chave, tamanho, vetor);

        // Busca linear
        //r = busca(chave, tamanho, vetor);

        // Habilite as duas linhas para busca binária:
        insertionSort(tamanho, vetor);
        r = buscaBinaria(chave, tamanho, vetor);

        fprintf(saida, "%d %d\n", tamanho, NOP);

        free(vetor);
    }

    fclose(saida);

    return 0;
}

int busca (int chave, int tam, int vetor[]) {
    int i;
    i = tam - 1;
    NOP = 0;

    while (i >= 0 && vetor[i] != chave) {
        i -= 1;
        NOP++;
    }

    return i;
}

int maximo (int tam, int vetor[]) {
    int i, m = vetor[0];
    NOP = 0;
    for(i = 1; i < tam; i++) {
        NOP++;
        if(vetor[i] > m)
            m = vetor[i];
    }
    return m;
}

int twosum (int chave, int tam, int vetor[]) {

    int i, j, r;
```

```
    NOP = 0;
    r = 0;

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

            if(vetor[i] + vetor[j] == chave)
                r = 1;
        }

    return r;
}

void insertionSort(int tam, veti vetor){
        int i, j, chave;

        NOP = 0;
        for(i = 1; i < tam; ++i){
                chave = vetor[i];

                for(j = i -1; j >= 0 && vetor[j] > chave; --j){
                        vetor[j+1] = vetor[j];
                        NOP++;
                }

                vetor[j+1] = chave;
        }
}

int buscaBinaria(int chave, int tam, veti vetor){
    int ini = 0, fim = tam - 1, meio;

    NOP = 0;

    while(ini <= fim){
        meio = (ini + fim)/2;

        NOP++;

        if(chave == vetor[meio])
            return meio;
        if(chave < vetor[meio])
            fim = meio - 1;
        else
            ini = meio + 1;
    }
```

```
    return -1;
}

// Aloca um vetor de inteiros de tamanho n.
// Devolve o ponteiro para esse vetor.
veti criaVeti(int tam) {
   return (int *) malloc(tam * sizeof(int));
}
```

Feedback da análise

Na **busca linear** a chave buscada pode estar em qualquer lugar do vetor, então, probabilisticamente, pode-se dizer que a posição esperada de se encontrar a chave é no meio do vetor, logo, o NOP médio para a busca linear é de metade do tamanho do vetor, o que faz com que a complexidade de tempo da busca linear seja da ordem de n , onde n é o tamanho do vetor.

Isto se reflete nos dados obtidos da execução do programa, onde observamos alguns NOP pequenos, alguns NOP bem altos (próximos ao tamanho do vetor), mas a maior parte se encontra em um intervalo em torno da média do vetor.

Por outro lado, na **busca binária**, em cada etapa dividimos o vetor em 2, logo, em m operações da busca, cobrimos um vetor de tamanho 2^m, logo, para um vetor de tamanho n, teremos um número próximo de $ln_2\ n$ operações.

Novamente, este fato se refletiu nos dados observados da execução do programa.