Atividade 6 - Sem sort ("Bogosort")

Davi Gabriel Domingues - 15447497

25 de Outubro de 2025

1 Explicação do cenário

A situação solicitou o desenvolvimento de um algoritmo de ordenação peculiar e famoso, didaticamente, o "Bogosort", a fim de contrapor com a intuição da entrega anterior (Entrega05). O objetivo é ilustrar como o código foi desenvolvido, além de seu impacto prático no cenário de aplicação construído.

2 Código desenvolvido

A seguir, apresenta-se o algoritmo propriamente desenvolvido e comentado sucintamente para expor o que cada trecho possui como funcionalidade, para a correta execução no ambiente do runcodes:

```c
#include <stdio.h>
#include <stdlib.h>
#include <stdint.h>
#include <stdbool.h>
#include "util.h"

// Helper para imprimir long long sem printf %lld
static void print_ll(long long x) {
    // Trata nmeros negativos
    if (x < 0) {
        putchar('-');
        x = -x;
    }

    char buf[32]; // Buffer suficiente para armazenar o nmero

    // Preenche o buffer com os dgitos em ordem reversa
    int i = 0;
    do {
        buf[i++] = (char)('0' + (int)(x % 10));
        x /= 10;
    } while (x);

    // Imprime os dgitos na ordem correta
    while (i--)
        putchar(buf[i]);
}

// Verifica se o vetor est ordenado em ordem crescente
bool is_sorted(int *v, int n) {
    for (int i = 1; i < n; i++)
        if (v[i - 1] > v[i])
```

```
            return false;

    return true;
}

// Verso do PDF (necessria para passar no runcodes)
void shuffle(int *v, int n, int *seed) {
    for (int i = n - 1; i >= 1; i--) {
        int j = get_random(seed, i) - 1, temp = v[i];
        v[i] = v[j];
        v[j] = temp;
    }
}

int main() {
    int n, seed = 12345;
    scanf("%d", &n);
    int *v = (int *)malloc(n * sizeof(int));

    // Preenche o vetor com nmeros aleatrios entre 1 e n
    for (int i = 0; i < n; i++)
        v[i] = get_random(&seed, n);

    long long count = 0;

    // Aplica o bogosort (ordenao por permutao aleatria)
    while (!is_sorted(v, n)) {
        shuffle(v, n, &seed);
        count++;
    }

    // Imprime o nmero de permutaes realizadas
    print_ll(count);
    putchar('\n');

    // Imprime o vetor ordenado
    for (int i = 0; i < n; i++)
        printf("%d%s", v[i], (i == n - 1) ? "" : " ");
    printf("\n");

    free(v);
    return 0;
}
```

3 Disucssão sobre desempenho

O algoritmo gera permutações aleatórias do vetor (por exemplo, via "Fisher–Yates") e, após cada embaralhamento, verifica se está ordenado, sendo que o processo se repete até que a verificação retorne verdadeira. Trata-se de um procedimento puramente aleatório cuja lógica não explora propriedades estruturais dos dados.

Complexidade probabilística

Seja m o número de permutações distintas do multiconjunto de entrada (para elementos distintos $m = n!$; com multiplicidades $m = \dfrac{n!}{\prod_k c_k!}$). A probabilidade de obter, numa tentativa qualquer, a permutação ordenada é $1/m$, portanto, o número esperado de embaralhamentos até sucesso é:

$$E[\text{shuffles}] = m.$$

Cada tentativa custa $\Theta(n)$ (embaralhar $\Theta(n)$ + verificação $\Theta(n)$), logo o tempo esperado é:

$$T_{\text{esperado}} = \Theta(n \cdot m)$$

Para chaves distintas, resulta em $\Theta(n \cdot n!)$ — crescimento "superfatorial" e impraticável já para n modestos. O melhor caso é imediato (vetor já ordenado); não existe garantia determinística de término em tempo polinomial (apenas expectativa finita).

Memória e propriedades adicionais

O algoritmo é in-place (uso extra $\Theta(1)$), não é estável (embaralhamento altera a ordem relativa de iguais) e é puramente probabilístico. Em entradas com muitas repetições, m diminui e o desempenho melhora proporcionalmente; mesmo assim, em termos assintóticos, continua não competitivo.

Comparação sucinta com algoritmos clássicos

- **Mergesort / Heapsort / Quicksort (esperado)**: tempo $\Theta(n \log n)$ (Quicksort pior caso $O(n^2)$); práticos e escaláveis — contrastam drasticamente com $\Theta(n \cdot n!)$ do Bogosort.
- **Timsort / Introsort**: $\Theta(n \log n)$ com otimizações e robustez em dados reais; padrão em bibliotecas.
- **Counting / Radix**: para chaves inteiras pequenas, $O(n + k)$ ou $O(n \cdot w)$, muito mais eficientes que abordagens permutacionais.

Resumidamente, o Bogosort é um instrumento didático que ilustra limitações de estratégias puramente aleatórias: sua complexidade esperada cresce como $\Theta(n \cdot m)$ (tipicamente $\Theta(n \cdot n!)$), tornando-o inviável na prática, frente a algoritmos com garantia polinomial como mergesort, heapsort e timsort.