AED2 - Aula 14
Problema da contagem de inversões e limitante inferior para ordenação baseada em comparações

"Podemos fazer melhor?"
- mote do projetista de algoritmos

Nesta aula vamos estudar outro problema básico e descobrir como reaproveitar ideias centrais de outros algoritmos de ordenação para desenvolver diversas soluções interessantes para ele.

Também vamos verificar que existe uma limitação teórica para a eficiência de qualquer algoritmo de ordenação baseado em comparações. Note que este conjunto contém todos os algoritmos de ordenação que estudamos até o momento.

Problema da Contagem de Inversões

Definição:

- Uma inversão corresponde a um par de elementos v[i] e v[j]
    - tal que i < j e v[i] > v[j]
- Dado um vetor v de tamanho n,
    - queremos saber quantas inversões existem em v

Exemplos:

- 3 2 5 4 1
    - 3 está invertido com 2 e 1
    - 2 está invertido com 1
    - 5 está invertido com 4 e 1
    - 4 está invertido com 1
    - Total de inversões = 2 + 1 + 2 + 1 = 6
- 1 2 3 4 5
    - Total de inversões = 0
- 5 4 3 2 1
    - 5 está invertido com 4, 3, 2 e 1
    - 4 está invertido com 3, 2 e 1
    - 3 está invertido com 2 e 1
    - 2 está invertido com 1
    - Total de inversões = 4 + 3 + 2 + 1

Curiosidades:

- Número mínimo de inversões = 0
    - ocorre quando o vetor está em ordem crescente
- Número máximo de inversões = (n escolhe 2) = n(n - 1)/2
    - o valor (n escolhe 2) corresponde a todo par ser uma inversão
    - ocorre quando o vetor está em ordem decrescente

Algoritmos:

```
unsigned long long contarInversoes1(int v[], int n)
{
   int i, j, aux;
   unsigned long long num_inv = 0;
   for (j = 1; j < n; j++)
   {
       aux = v[j];
       for (i = j - 1; i >= 0 && aux < v[i]; i--)
       {
           v[i + 1] = v[i];
           // num_inv++;
       }
       num_inv += j - 1 - i;
       v[i + 1] = aux; /* por que i+1? */
   }
   return num_inv;
}
```

- Invariante e corretude:
    - v[0 .. j - 1] está ordenado
    - num_inv = número de inversões envolvendo apenas elementos do subvetor v[0 .. j - 1] original
- Eficiência de tempo:
    - O(n^2) no pior caso
    - O(n) no melhor caso
- Eficiência de espaço: O(1) espaço adicional

```
unsigned long long contarInversoes2(int v[], int n)
{
   int j, i, aux, ut, l;
   unsigned long long num_inv = 0;
   l = n;
   for (j = 0; j < n; j++)
   {
       ut = 0;
       for (i = 1; i < l; i++)
           if (v[i - 1] > v[i])
           {
               aux = v[i - 1];
               v[i - 1] = v[i];
```

```
                v[i] = aux;
                ut = i;
                num_inv++;
            }
        l = ut;
    }
    return num_inv;
}
```

- Invariante e corretude:
    - v[l .. n - 1] está ordenado e é >= v[0 .. l - 1]
    - num_inv = número de inversões desfeitas até o momento
- Eficiência de tempo:
    - O(n^2) no pior caso
    - O(n) no melhor caso
- Eficiência de espaço: O(1) espaço adicional

```
// primeiro subvetor entre p e q-1, segundo subvetor entre q e r-1
unsigned long long intercala(int v[], int p, int q, int r)
{
    int i, j, k, tam;
    unsigned long long num_inv = 0;
    i = p;
    j = q;
    k = 0;
    tam = r - p;
    int *w = malloc(tam * sizeof(int));
    while (i < q && j < r)
    {
        if (v[i] <= v[j])
            w[k++] = v[i++];
        else // v[i] > v[j]
        {
            w[k++] = v[j++];
            num_inv += q - i;
        }
    }
    while (i < q)
        w[k++] = v[i++];
    while (j < r)
        w[k++] = v[j++];
    for (k = 0; k < tam; k++)
        v[p + k] = w[k];

    free(w);
    return num_inv;
}
```

- Invariante e corretude:

        - w[0 .. k - 1] ordenado e possui os elementos de v[0 .. i - 1] e v[q .. j - 1]
        - num_inv = número de inversões envolvendo elemento de v[q .. j - 1] e elementos de v[0 .. q - 1]
- Eficiência de tempo: O(r - p)
- Eficiência de espaço: O(r - p) espaço adicional

```
// p indica a primeira posicao e r-1 a ultima
unsigned long long contarInversoesR(int v[], int p, int r)
{
   int m;
   unsigned long long num_inv = 0;
   if (r - p > 1)
   {
       m = (p + r) / 2;
       // m = p + (r - p) / 2;
       num_inv += contarInversoesR(v, p, m);
       num_inv += contarInversoesR(v, m, r);
       num_inv += intercala(v, p, m, r);
   }
   return num_inv;
}
```

- Eficiência de tempo: O((r - p) lg (r - p))
- Eficiência de espaço: O(r - p) espaço adicional

```
unsigned long long contarInversoes3(int v[], int n)
{
   return contarInversoesR(v, 0, n);
}
```

- Eficiência de tempo: O(n lg n)
- Eficiência de espaço: O(n) espaço adicional

Quizz:

- Todas as nossas adaptações de algoritmos para contagem de inversões
    - são de algoritmos de ordenação estável. Será coincidência?

Limitante inferior para ordenação baseada em comparações

Algoritmos de ordenação baseados em comparação são aqueles que

- só obtém informação sobre a sequência sendo ordenada
- através de uma função de comparação que
    - recebe dois elementos
    - e diz qual deles é maior.

Exemplos destes algoritmos são:

- mergeSort, quickSort, heapSort, selectionSort, insertionSort e bubbleSort.

Para obter um limitante inferior para o número de operações que
- qualquer algoritmo de ordenação precisa realizar,
    - vamos comparar duas grandezas.

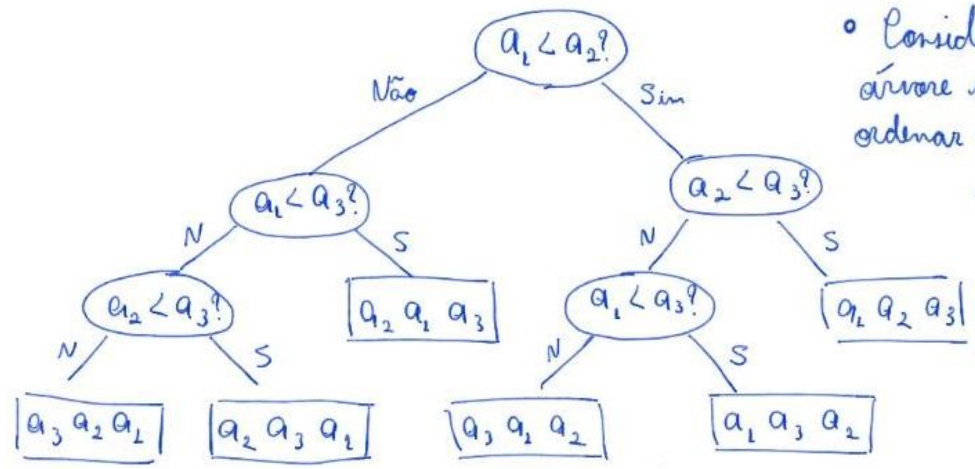
A primeira é o número de permutações distintas
- que uma sequência de tamanho n pode apresentar.
- Este número é n!,
    - pois qualquer um dos n elementos pode ser o primeiro,
    - qualquer dos n-1 restantes pode ser o segundo,
    - e assim por diante.

A segunda grandeza é o número de sequências
- que um algoritmo consegue distinguir após realizar k comparações.
- Este número é 2^k,
    - pois cada uma das k comparações entre um par de elementos
    - pode resultar em duas possibilidade,
        - o primeiro é maior que o segundo,
        - ou o segundo é maior que o primeiro.

Para facilitar a visualização,
- podemos representar as possíveis comparações de um algoritmo
    - usando uma árvore binária de decisão,
        - na qual cada nó interno corresponde a uma comparação
        - e cada folha corresponde a uma sequência.

- Considere a seguinte árvore usada para ordenar a sequência

| a_1 | a_2 | a_3 |



A altura desta árvore, ou seja,
- o caminho mais longo da raíz até uma folha corresponde a
    - um limitante inferior para a complexidade de tempo de pior caso do algoritmo.

Pelo princípio da casa dos pombos temos que

- se tivermos n + 1 pombos para serem colocados em n casas,
    - então pelo menos uma casa deverá conter dois ou mais pombos.

No nosso problema,

- os pombos são as n! permutações de uma sequência de tamanho n
- as casas dos pombos são cada as 2^k sequências
    - que um algoritmo consegue identificar depois de k comparações.
- se houverem mais pombos do que casas,
    - i.e., 2^k < n!
    - então o algoritmo não conseguirá distinguir
        - entre duas sequências diferentes.
    - Portanto, ele não ordenará corretamente ao menos uma delas.
- Assim, para que o algoritmo tenha chance de ordenar corretamente
    - 2^k >= n!

Para resolver a inequação vamos simplificar n!,

- substituindo-o por um limitante inferior

$$n! = \overbrace{n.(n-1).(n-2)\ldots.(n/2+1)}^{n/2 \text{ termos}}(n/2)(n/2-1)\ldots 2.1$$

$$\geq \underbrace{\frac{n}{2}.\frac{n}{2}.\frac{n}{2}.\ldots.\frac{n}{2}}_{n/2 \text{ termos}}.1.1.\ldots 1.1$$

$$= (n/2)^{n/2}$$

Simplificando n!

Assim,

2^h >= n!

>= (n/2)^(n/2)

Aplicando lg dos dois lados

h >= (n/2) lg (n/2) = Omega(n log n)

Assim, concluímos que

- qualquer algoritmo de ordenação baseado em comparação
    - precisa realizar pelo menos da ordem de n log n comparações
        - para ordenar uma sequência com n elementos.