AED2 - Aula 13
Limitante inferior para ordenação baseada em comparações e bucket sort

"Encarando e transpondo barreiras"

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:

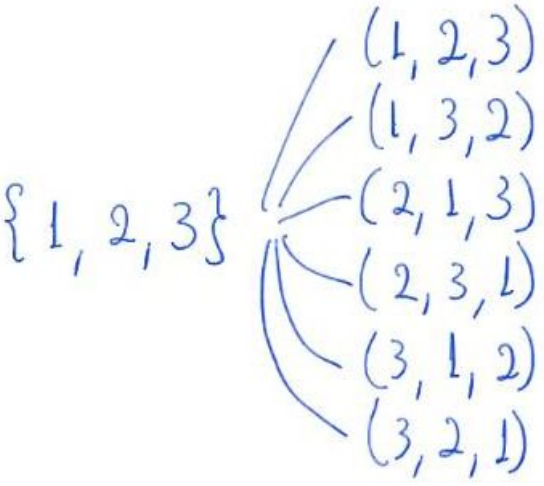
- mergeSort, quickSort, heapSort, selectionSort, insertionSort e bubbleSort.

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

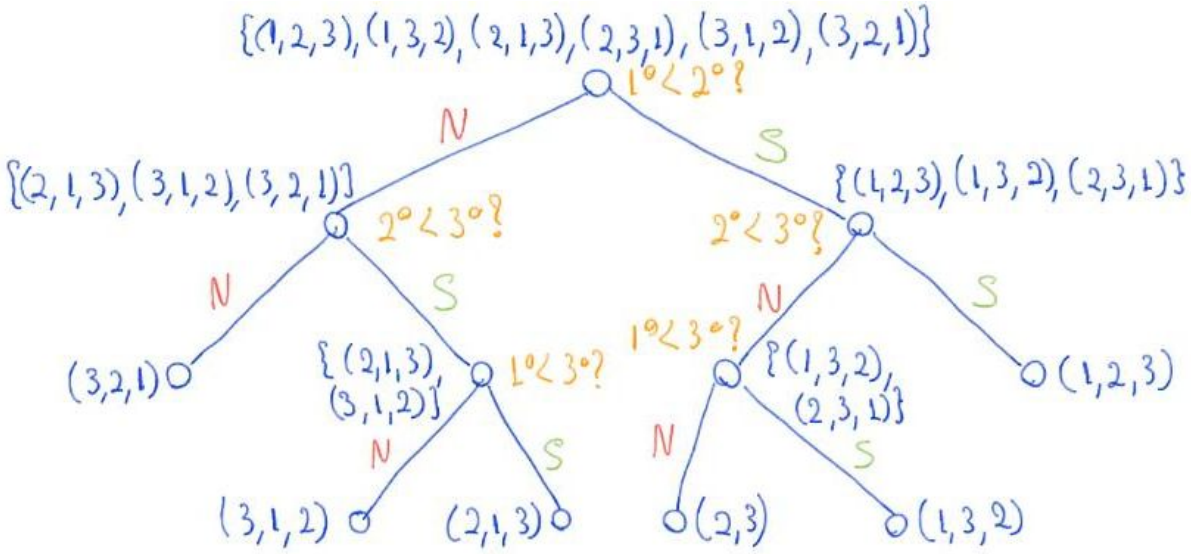
- que uma sequência de tamanho n pode apresentar.



- 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

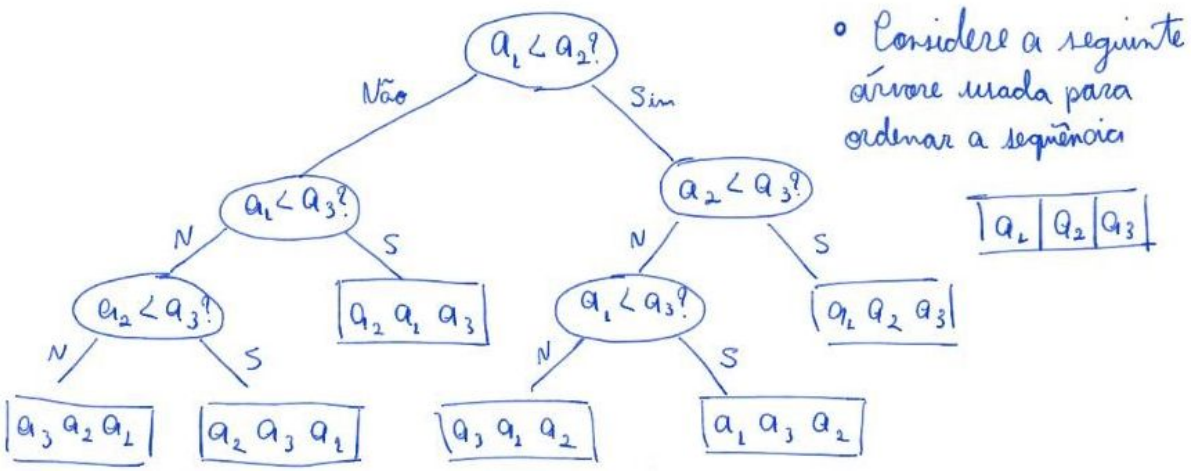
- que um algoritmo consegue distinguir após realizar k comparações.



- Esse número é importante pois,
    - se o algoritmo obtiver as mesmas respostas das comparações
        - para duas sequências distintas,
    - ele executará da mesma forma para ambas,
        - devolvendo uma resposta errada em pelo menos um dos casos.
- Note que, no início todas as sequências são iguais para o algoritmo,
    - e que cada comparação entre um par de elementos
        - pode resultar em duas possibilidades,
            - o primeiro é maior que o segundo,
            - ou o segundo é maior que o primeiro.
- Podemos pensar que, na melhor das hipóteses,
    - cada comparação divide o espaço de busca em duas metades ou,
    - de modo complementar, dobra o número de sequências identificadas.
- Assim, no melhor caso, após k comparações,
    - conseguimos identificar 2^k sequências diferentes.

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.



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

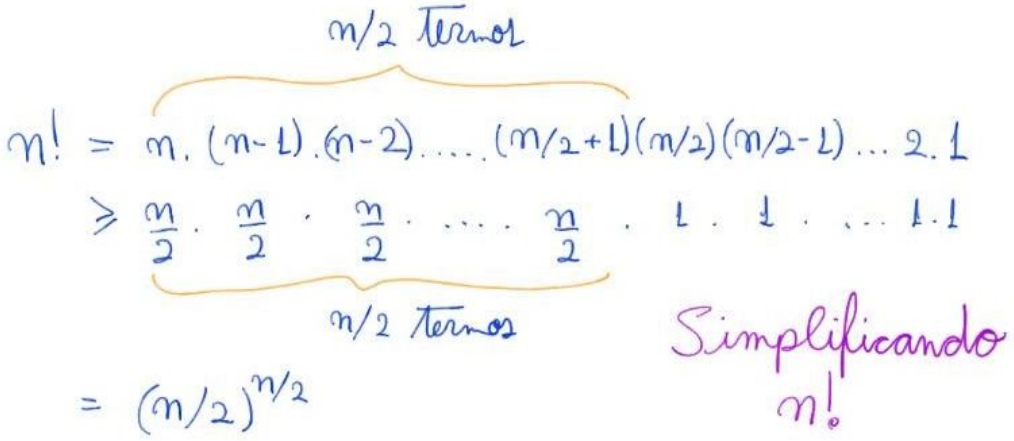
- se tivermos n + 1 pombos para serem colocados em n casas,
    - então pelo menos uma casa deverá conter dois ou mais pombos.

No nosso problema,

- os pombos são as n! permutações de uma sequência de tamanho n,
- as casas dos pombos são 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



Assim,

- 2^h >= n! >= (n/2)^(n/2)

Aplicando lg dos dois lados

- h >= (n/2) lg (n/2) = Omega(n log n),
    - sendo que = Omega(f(n)) deve ser interpretado como
        - pelo da ordem de f(n).

Assim, concluímos que

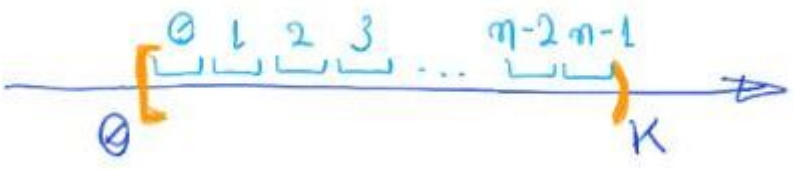
- qualquer algoritmo de ordenação baseado em comparações
    - precisa realizar pelo menos da ordem de n log n comparações
        - para ordenar uma sequência com n elementos.

Em vista dessa barreira, como podemos transpô-la?

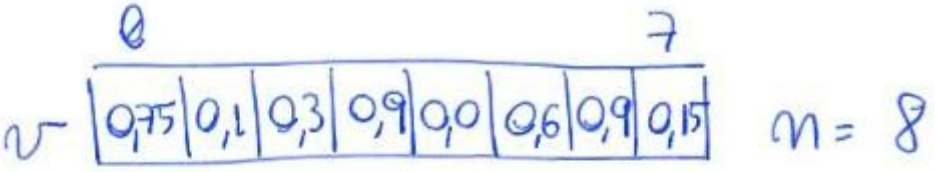
Bucket sort

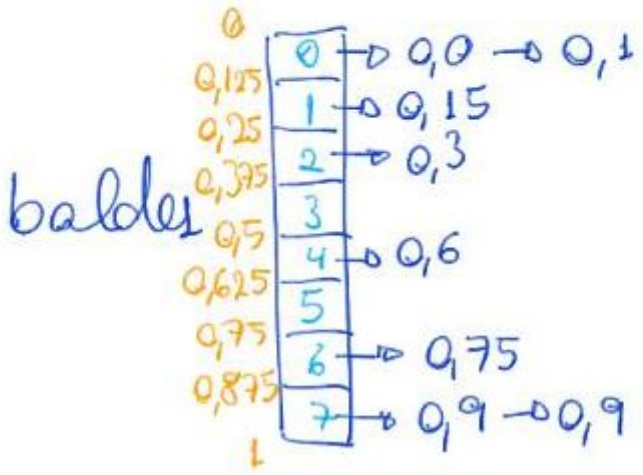
Este é um método eficiente para ordenar um conjunto com n elementos

- distribuídos uniformemente num intervalo de tamanho k.
- Primeiro, dividimos o intervalo de tamanho k em n baldes (buckets),
    - associando a cada balde uma fração de valor igual a k/n.



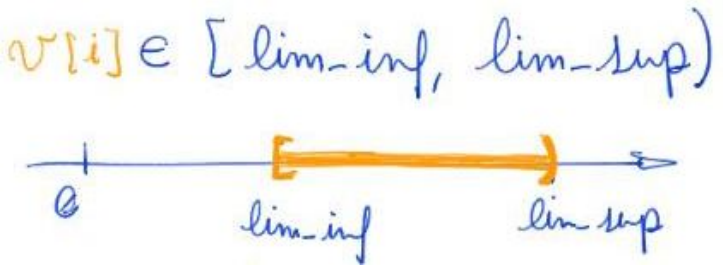
- Então colocamos cada elemento em seu respectivo balde.
- Em seguida ordenamos os elementos de cada balde,
    - o que leva tempo esperado constante,
        - já que cada balde deve ter um número pequeno de elementos,
            - uma vez que estes vieram de uma distribuição uniforme.

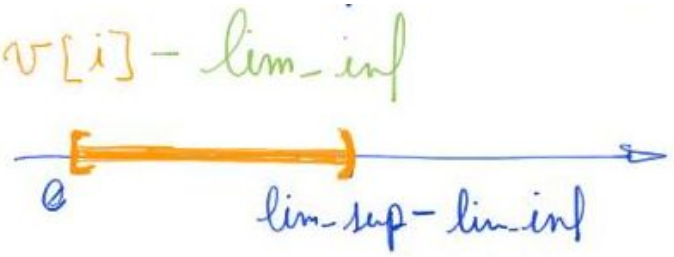


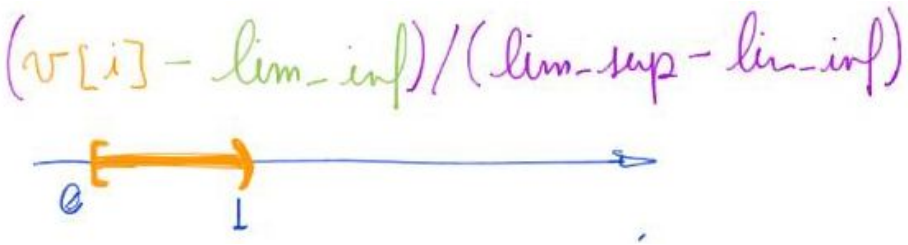


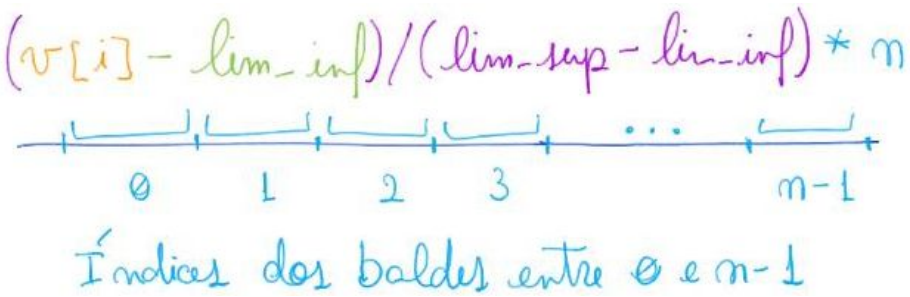
- Por fim, percorremos os baldes em ordem
    - e os elementos de cada balde, também em ordem,
        - copiando eles de volta para o vetor original.

Vamos ver como calcular os índices dos baldes:









Códigos:

```c
#define lim_inf 0
#define lim_sup 1

typedef struct celula
{
    double chave;
    struct celula *prox;
} Celula;

void bucketSort(double v[], int n)
{
    int i, j;
    Celula *p, *nova, *morta;
    Celula **baldes = (Celula **)malloc(n * sizeof(Celula *));
    // inicializando cada balde com uma lista com um nó cabeça
    for (j = 0; j < n; j++)
    {
        baldes[j] = (Celula *)malloc(sizeof(Celula));
        baldes[j]->prox = NULL;
    }
    // coloca cada elemento no balde correspondente
    for (i = 0; i < n; i++)
    {
        // j = (int)(v[i] * n);
        j = (int)((double)(v[i] - lim_inf) / (lim_sup - lim_inf) *
n);
        p = baldes[j];
        // já insere o elemento na ordem correta dentro do balde
```

```
        while (p->prox != NULL && p->prox->chave < v[i])
            p = p->prox;
        nova = (Celula *)malloc(sizeof(Celula));
        nova->chave = v[i];
        nova->prox = p->prox;
        p->prox = nova;
    }
    // põe os elementos dos baldes de volta no vetor
    i = 0;
    for (j = 0; j < n; j++)
    {
        p = baldes[j]->prox;
        while (p != NULL)
        {
            v[i] = p->chave;
            i++;
            p = p->prox;
        }
    }
    // libera os baldes
    for (j = 0; j < n; j++)
    {
        p = baldes[j];
        while (p != NULL)
        {
            morta = p;
            p = p->prox;
            free(morta);
        }
    }
    free(baldes);
}
```

Eficiência de tempo esperado é O(n),

- apenas se as chaves forem uniformemente distribuídas.
- Sem essa hipótese o tempo de pior caso do algoritmo é O(n^2),

    - pois muitos elementos podem se acumular num mesmo balde,
    - e a ordenação deste pode levar tempo quadrático,
        - dependendo do método utilizado.
- Diante disso, vale a pena usar um método de ordenação O(n lg n),
    - para ordenar cada balde?
        - Em geral não, pois são esperados poucos elementos por balde,
        - e métodos como insertionSort são melhores para n pequeno.

Eficiência de espaço:

- Memória adicional da ordem de n,
    - pois são utilizados n baldes e
    - n células, uma para cada elemento.

Estabilidade:

- A estabilidade depende da implementação da ordenação intrabalde.
- O código que estudamos não é estável, mas uma pequena modificação
    - na inserção/ordenação intra balde pode corrigir isso.
        - Que modificação é essa?

Bônus:

- O método de inserção em ordem intrabalde lembra
    - um algoritmo de ordenação que já estudamos.
        - Qual é esse algoritmo?