AED2 - Aula ~~13~~14.1
Limitante inferior para ordenação baseada em comparações e bucketSort

"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 desses algoritmos são:

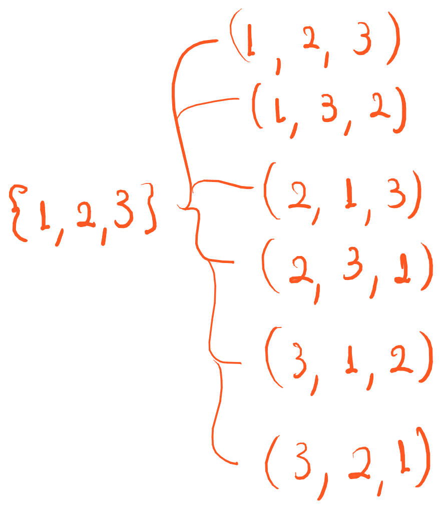
- mergeSort, quickSort, heapSort, selectionSort, insertionSort e bubbleSort.

Para obter um limitante inferior para o número de operações

- um algoritmo de ordenação baseado em comparações precisa realizar,
    - vamos comparar duas grandezas.

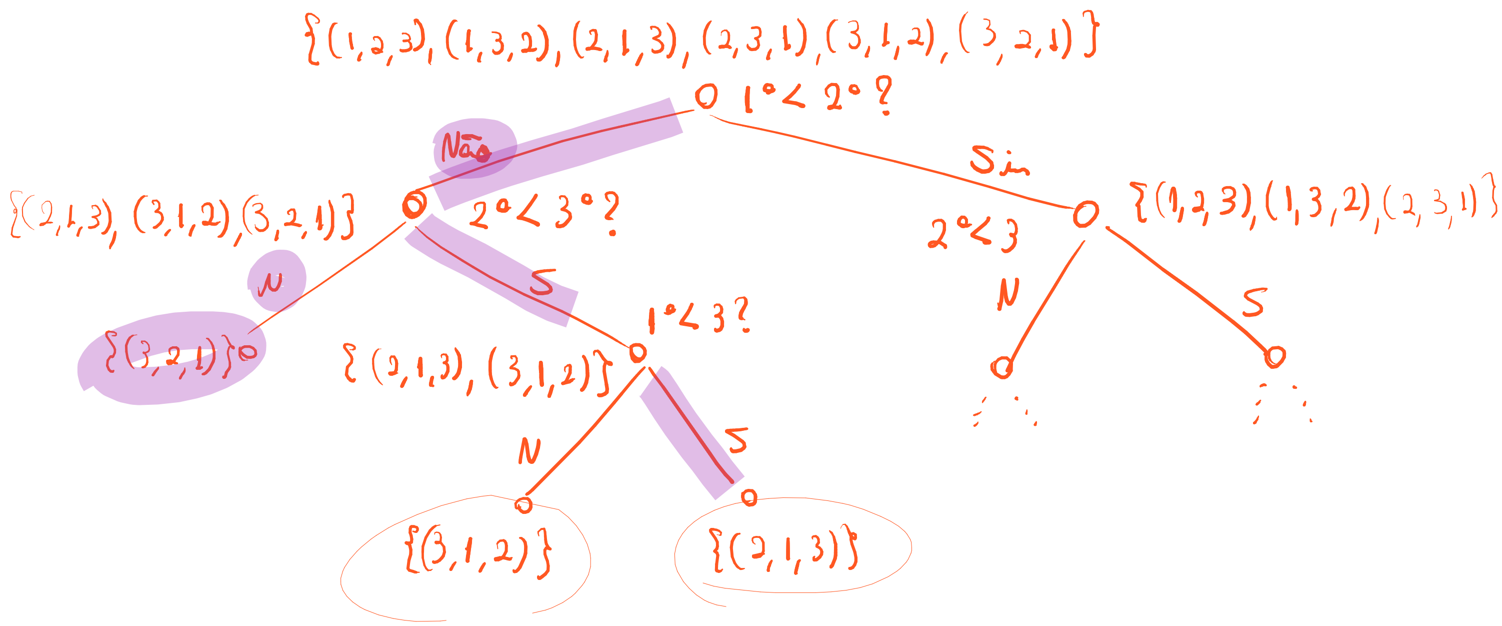
A primeira é o número de permutações,

- 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.



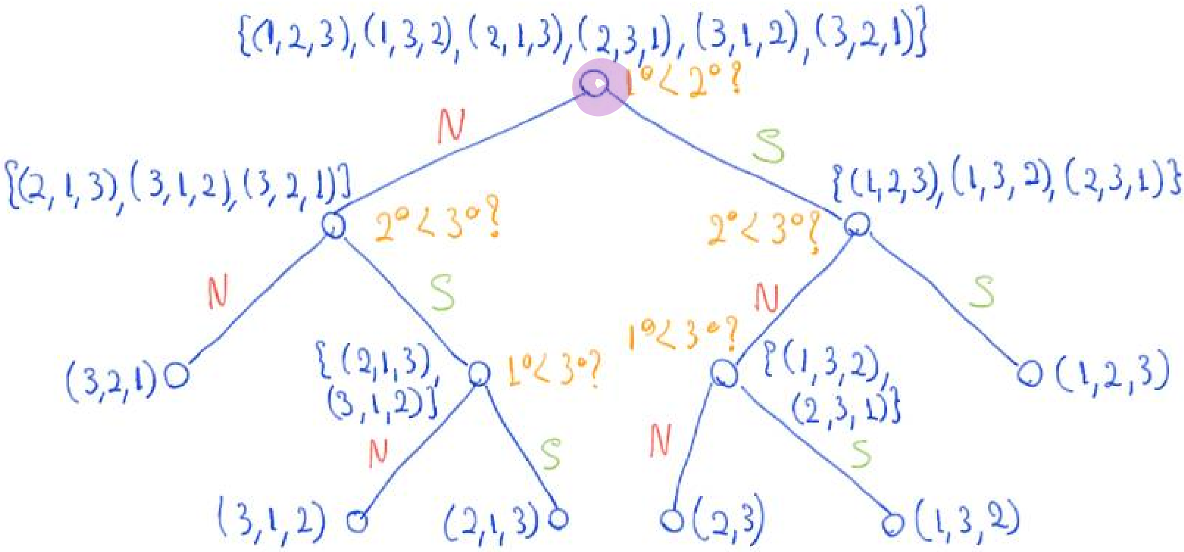
A segunda é o número de sequências que um algoritmo consegue distinguir

- após realizar k comparações. Vejamos um exemplo em que tentamos
    - classificar as permutações anteriores.



O número de sequências que um algoritmo consegue distinguir após k comparações

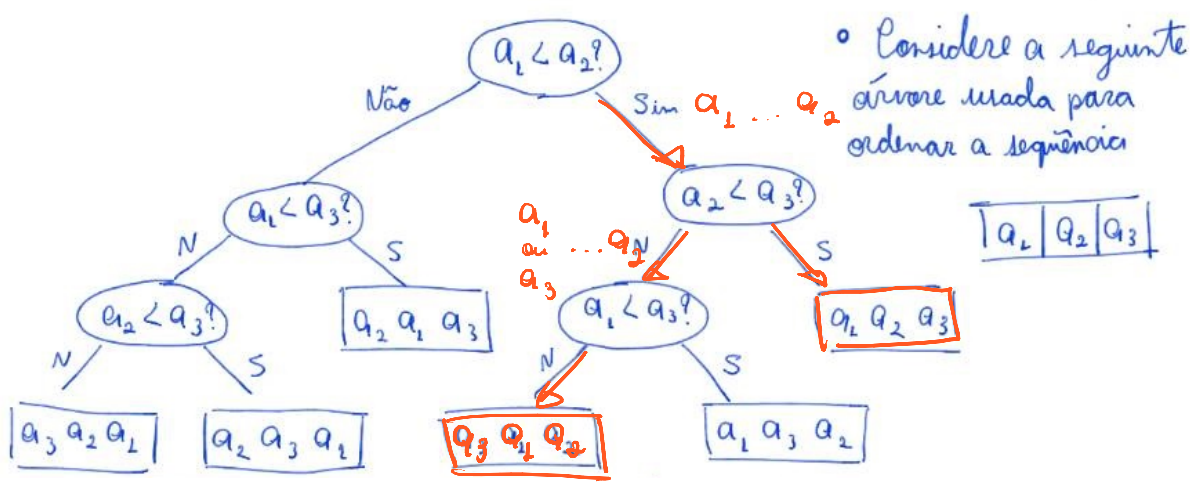
- é importante pois, se o algoritmo obtiver as mesmas respostas
    - ao comparar duas sequências distintas,
- ele executará da mesma forma para ambas,
    - devolvendo uma resposta errada em pelo menos um dos casos.



- 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 no máximo 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 comprimento do caminho mais longo da raiz 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,
- e 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., $n! > 2^k$
  - 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
  - é necessário que $2^k \geq n!$

Para resolver a inequação vamos simplificar n!,

- substituindo-o por um limitante inferior.

$$n! = \underbrace{n(n-1)(n-2)\dots\left(\tfrac{n}{2}+1\right)}_{n/2 \text{ termos}}\underbrace{\left(\tfrac{n}{2}\right)\left(\tfrac{n}{2}-1\right)\dots 3.2.1}_{n/2 \text{ termos}}$$

$$\geq \left[\frac{n}{2}\cdot\frac{n}{2}\cdot\frac{n}{2}\dots\frac{n}{2}\right].1.1.\dots1.1.1$$

$$= \left(\frac{n}{2}\right)^{n/2}$$

Assim, $2^k \geq n! \geq \left(\frac{n}{2}\right)^{n/2}$

Aplicando lg dos dois lados temos $lg(2^k) \geq lg\left(\left(\frac{n}{2}\right)^{\frac{n}{2}}\right) \Rightarrow k \geq \frac{n}{2}\cdot lg\left(\frac{n}{2}\right)$

- sendo que $\Omega(f(n))$ significa pelo menos da ordem de $\rightarrow f(n)$

$k = \Omega(n\, lg\, n)$

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.

Agora que encaramos essa barreira, como podemos transpô-la?

- Será que conseguimos ordenar sem fazer comparações?

bucketSort

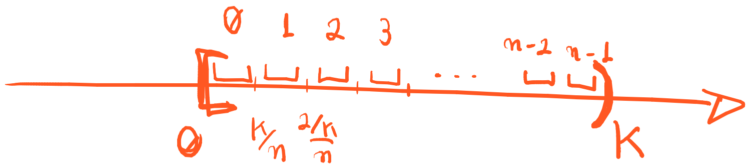
$\Omega(n \log n)$

É é um método eficiente para ordenar um conjunto com n números

- distribuídos uniformemente num intervalo de tamanho k.

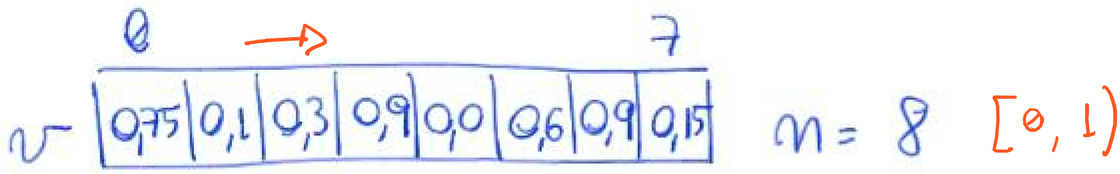
Ideia:

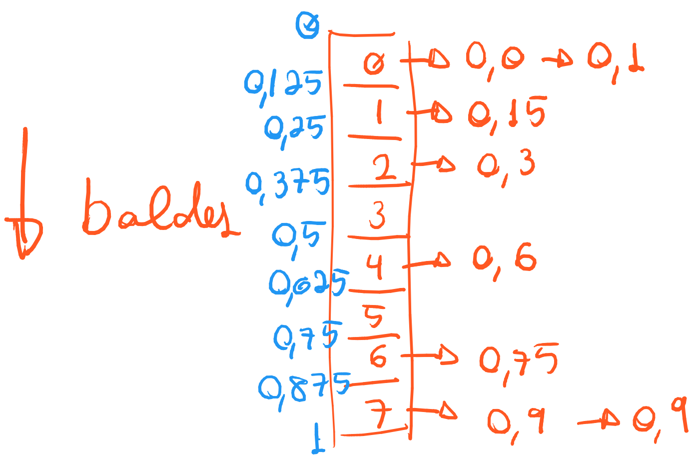
- Primeiro, dividimos o intervalo de tamanho k em n baldes (buckets),
    - associando a cada balde uma fração do intervalo, de tamanho $\frac{k}{n}$

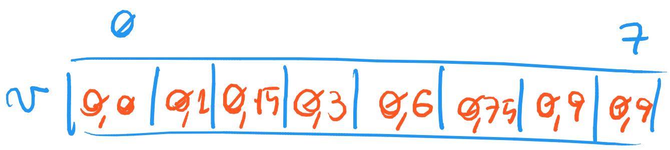


- Então, colocamos cada número em seu respectivo balde.
- Em seguida, ordenamos os elementos de cada balde.
- Por fim, percorremos os baldes em ordem
    - e os elementos de cada balde, também em ordem,
        - copiando eles de volta para o vetor original.

Exemplo:



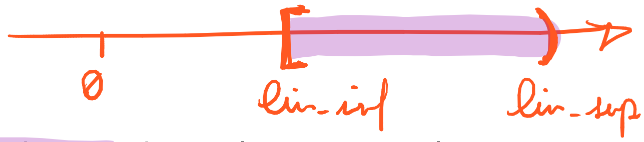




Vamos ver como calcular o índice do balde em que deve entrar cada elemento:

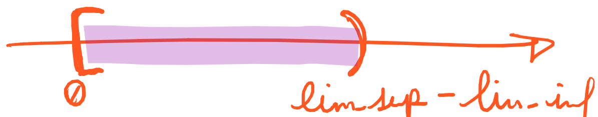
- Considere um elemento v[i] qualquer com valor no intervalo [lim_inf, lim_sup).

$$v[i] \in [\text{lim\_inf}, \text{lim\_sup})$$



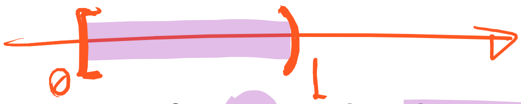
- Primeiro trasladamos o intervalo, para que ele comece em 0.

$$(v[i] - \text{lim\_inf}) \in [0, \text{lim\_sup} - \text{lim\_inf})$$



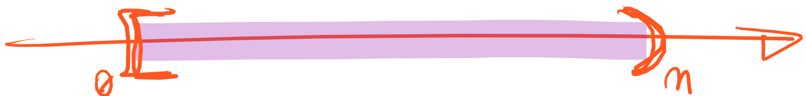
- Então, redimensionamos para um intervalo unitário.

$$((v[i] - \text{lim\_inf})/(\text{lim\_sup} - \text{lim\_inf})) \in [0, 1)$$



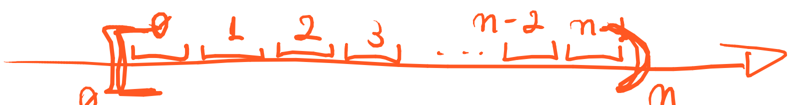
- Agora, reescalamos pelo fator n, que é o número total de elementos.

$$((v[i] - \text{lim\_inf})/(\text{lim\_sup} - \text{lim\_inf}) * n) \in [0, n)$$



- Por fim, truncamos para o inteiro menor mais próximo,

$$(\text{int})((v[i] - \text{lim\_inf})/(\text{lim\_sup} - \text{lim\_inf}) * n) \in [0, n-1]$$



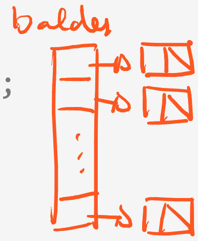
  - obtendo um índice de balde entre 0 e n - 1.

Códigos:

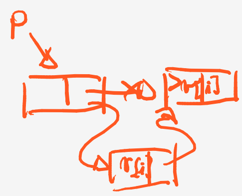
```c
#define lim_inf 0
#define lim_sup 1
```


```c
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 nó cabeça
    for (j = 0; j < n; j++) {
        baldes[j] = (Celula *)malloc(sizeof(Celula));
        baldes[j]->prox = NULL;
    }
```




```c
    // 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;
    }
```




```c
    // 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),

- desde que as chaves sejam uniformemente distribuídas no intervalo.
- Para verificar isso, note que copiar os números
  - do vetor de entrada para os baldes leva tempo O(n).
- De modo semelhante, copiá-los dos baldes de volta ao vetor leva tempo O(n).
- Além dessas operações, temos que ordenar os n baldes,
  - o que leva tempo esperado constante, i.e., O(1).
- Isso porque, cada balde deve ter um número pequeno de elementos,
  - uma vez que estes vieram de uma distribuição uniforme.

- 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.
- Assim, vale a pena usar um método de ordenação O(n lg n) para 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.

Estabilidade:

- A estabilidade depende da implementação da ordenação intrabalde.
- O código que estudamos não é estável, mas uma pequena modificação
  - na inserção/ordenação intrabalde pode corrigir isso.
- Quiz1: Que modificação é essa?

Quiz2: O método de inserção em ordem intrabalde lembra

- um algoritmo de ordenação que já estudamos. Qual é esse algoritmo?

Curiosidade: a alocação de memória das listas pode piorar a constante do algoritmo.

- Uma alternativa é usar um vetor, com tamanho 2 lg n, por balde.
  - Isso porque, é muito improvável que uma distribuição uniforme
    - produza tal acúmulo em um balde.