

Análise e Projeto de Algoritmos

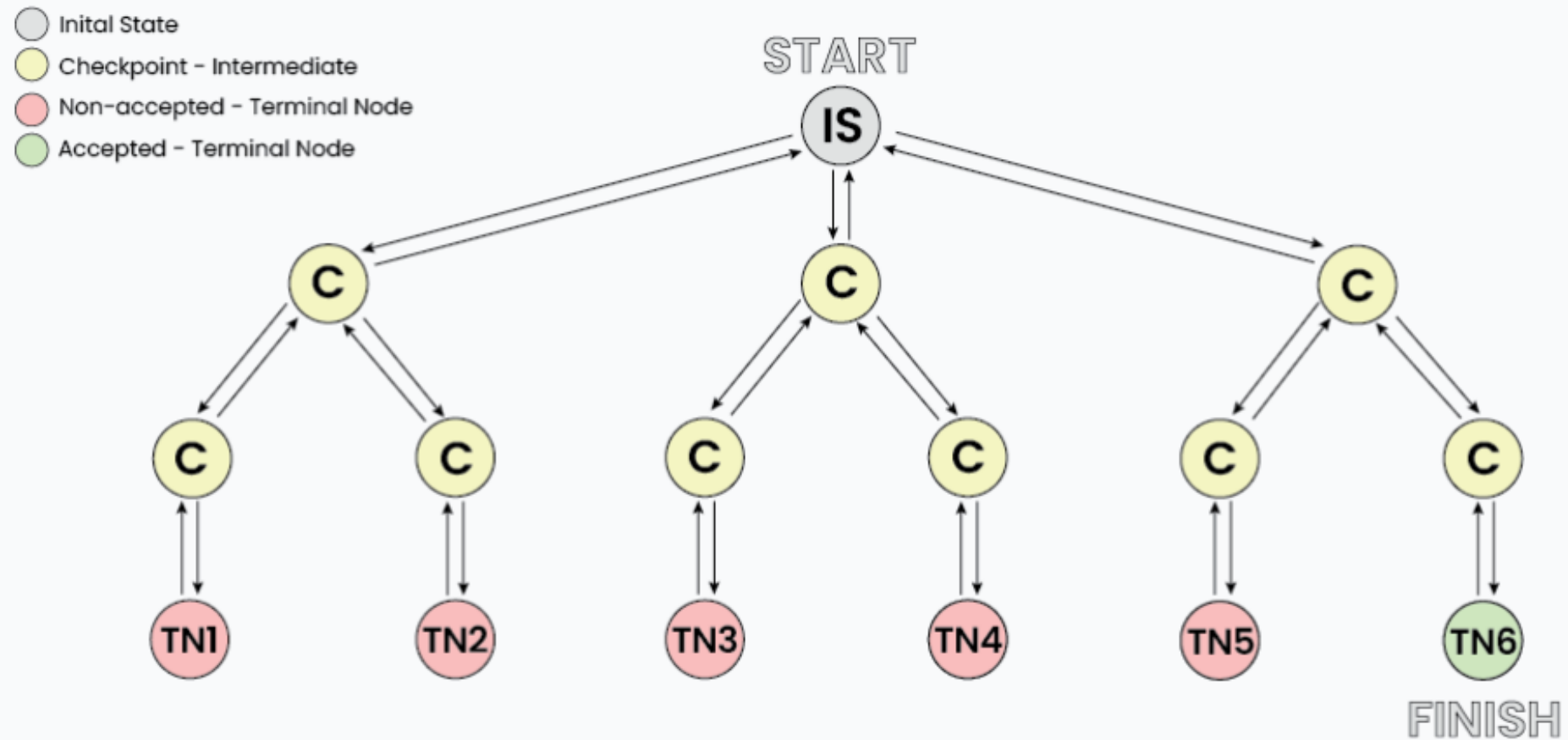
Backtracking e Algoritmos de classificação Interna

Backtracking

É um tipo de algoritmo que representa um refinamento da busca por **Força Bruta**, em que múltiplas soluções podem ser eliminadas sem serem explicitamente organizadas;

Força Bruta - É um algoritmo simples, mas de uso muito geral que consiste em enumerar todos os possíveis candidatos de uma solução e verificar se cada um satisfaz o problema;

Como funciona o Backtracking



Backtracking



Quando usar Backtracking

- Quando existe várias soluções para o problema;
- O problema pode ser dividido em subproblemas menores;
- Os subproblemas podem ser resolvidos de forma independente;

Aplicações do Backtracking

- Resolver quebra cabeças;
- Encontrar o caminho mais curto através de um labirinto;
- Agendamento de problemas;
- Problemas de alocação de recurso;
- Problemas de otimização de redes;

Algoritmos de classificação interna

São algoritmos usados para ordenar dados que cabem inteiramente na memória principal (RAM) do computador, sem a necessidade de uso de memória secundária. Ou seja, são algoritmos de ordenação;

Exemplos:

- BubbleSort;
- SelectionSort;
- InsertionSort;
- MergeSort;
- QuickSort;
- HeapSort;

Bubble Sort

Funcionamento: Compara elementos adjacentes e os trata se estiverem fora de ordem;

Complexidade: $O(n^2)$ em todos os casos;

Aplicações: Didática e pequenas listas

```
c Copiar código  
  
void bubbleSort(int arr[], int n) {  
    for (int i = 0; i < n - 1; i++) {  
        for (int j = 0; j < n - i - 1; j++) {  
            if (arr[j] > arr[j + 1]) {  
                int temp = arr[j];  
                arr[j] = arr[j + 1];  
                arr[j + 1] = temp;  
            }  
        }  
    }  
}
```


Selection Sort

Funcionamento: Em cada iteração, seleciona o maior ou menor elemento e coloca na posição correta da parte ordenada;

Complexidade: $O(n^2)$ em todos os casos;

Aplicações: Quando o número de trocas é importante

c

 Copiar código

```
void selectionSort(int arr[], int n) {  
    for (int i = 0; i < n - 1; i++) {  
        int minIndex = i;  
        for (int j = i + 1; j < n; j++) {  
            if (arr[j] < arr[minIndex]) {  
                minIndex = j;  
            }  
        }  
        int temp = arr[minIndex];  
        arr[minIndex] = arr[i];  
        arr[i] = temp;  
    }  
}
```


Insertion Sort

Funcionamento: Constrói a lista ordenada elemento por elemento, inserindo cada novo elemento na posição correta;

Complexidade: $O(n^2)$ pior caso e $O(n)$ melhor caso;

Aplicações: Listas já quase ordenadas;

```
void insertionSort(int arr[], int n) {  
    for (int i = 1; i < n; i++) {  
        int key = arr[i];  
        int j = i - 1;  
        while (j >= 0 && arr[j] > key) {  
            arr[j + 1] = arr[j];  
            j--;  
        }  
        arr[j + 1] = key;  
    }  
}
```

Merge Sort

Funcionamento: Divide o array em duas partes, ordena cada metade recursivamente, e depois intercala (merge) as duas metades ordenadas

Complexidade: $O(n \log n)$ em todos os casos;

Aplicações: grandes conjuntos de dados;

Merge Sort

```
void merge(int arr[], int l, int m, int r) {
    int n1 = m - l + 1;
    int n2 = r - m;

    int L[n1], R[n2];

    for (int i = 0; i < n1; i++)
        L[i] = arr[l + i];
    for (int j = 0; j < n2; j++)
        R[j] = arr[m + 1 + j];

    int i = 0, j = 0, k = l;
    while (i < n1 && j < n2) {
        if (L[i] <= R[j]) {
            arr[k] = L[i];
            i++;
        } else {
            arr[k] = R[j];
            j++;
        }
        k++;
    }

    while (i < n1) {
        arr[k] = L[i];
        i++;
        k++;
    }

    while (j < n2) {
        arr[k] = R[j];
        j++;
        k++;
    }
}
```

```
void mergeSort(int arr[], int l, int r) {
    if (l < r) {
        int m = l + (r - l) / 2;
        mergeSort(arr, l, m);
        mergeSort(arr, m + 1, r);
        merge(arr, l, m, r);
    }
}
```

Quick Sort

Funcionamento: Escolhe um pivô, particiona o array em duas subpartes, e depois ordena recursivamente

Complexidade: $O(n \log n)$ no caso médio e $O(n^2)$ pior caso;

Aplicações: grande conjunto de dados;

```
int partition(int arr[], int low, int high) {  
    int pivot = arr[high];  
    int i = (low - 1);  
  
    for (int j = low; j <= high - 1; j++) {  
        if (arr[j] < pivot) {  
            i++;  
            int temp = arr[i];  
            arr[i] = arr[j];  
            arr[j] = temp;  
        }  
    }  
  
    int temp = arr[i + 1];  
    arr[i + 1] = arr[high];  
    arr[high] = temp;  
    return (i + 1);  
}
```

```
void quickSort(int arr[], int low, int high) {  
    if (low < high) {  
        int pi = partition(arr, low, high);  
        quickSort(arr, low, pi - 1);  
        quickSort(arr, pi + 1, high);  
    }  
}
```

Heap Sort

Funcionamento: Primeiro, o array é transformado em um heap máximo. Depois, o maior elemento na raiz do heap é trocado com o último elemento, e o heap é ajustado;

Complexidade: $O(n \log n)$ em todos os casos;

Aplicações: Quando se necessita de uma ordenação “in-place”, ou seja, sem uso de memória externa
;

```
void heapify(int arr[], int n, int i) {  
    int largest = i;  
    int left = 2 * i + 1;  
    int right = 2 * i + 2;  
  
    if (left < n && arr[left] > arr[largest])  
        largest = left;  
  
    if (right < n && arr[right] > arr[largest])  
        largest = right;  
  
    if (largest != i) {  
        int temp = arr[i];  
        arr[i] = arr[largest];  
        arr[largest] = temp;  
  
        heapify(arr, n, largest);  
    }  
}
```

```
void heapSort(int arr[], int n) {  
    for (int i = n / 2 - 1; i >= 0; i--)  
        heapify(arr, n, i);  
  
    for (int i = n - 1; i >= 0; i--) {  
        int temp = arr[0];  
        arr[0] = arr[i];  
        arr[i] = temp;  
  
        heapify(arr, i, 0);  
    }  
}
```

Resumo das complexidades

Algoritmo	Complexidade Melhor Caso	Complexidade Médio Caso	Complexidade Pior Caso
Bubble Sort	$O(n)$	$O(n^2)$	$O(n^2)$
Selection Sort	$O(n^2)$	$O(n^2)$	$O(n^2)$
Insertion Sort	$O(n)$	$O(n^2)$	$O(n^2)$
Merge Sort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$
Quick Sort	$O(n \log n)$	$O(n \log n)$	$O(n^2)$
Heap Sort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$

Referências

[https://chatgpt.com/;](https://chatgpt.com/)

<https://www.geeksforgeeks.org/introduction-to-backtracking-2/>

<https://pt.wikipedia.org/wiki/Backtracking>