Universidade Federal de Roraima
Centro de Ciência e Tecnologia
Departamento de Ciência da Computação

CURSO DE BACHAREL EM CIÊNCIAS DA COMPUTAÇÃO

ALGORITMOS DE BUSCA

Boa Vista/RR
2025

ACADÊMICO:

ABRAHÃO PICANÇO NERES DE OLIVEIRA

ALGORITMOS DE BUSCA

Trabalho Avaliativo para matéria de Estrutura de Dados I em Ciências da Computação da Universidade de Roraima - UFRR.

Orientador: Filipe Dwan Pereira

Boa Vista/RR
2025

SUMÁRIO

1. INTRODUÇÃO

A ordenação e a busca são desafios essenciais na ciência da computação, impactando diretamente o desempenho de aplicações como bancos de dados e algoritmos de inteligência artificial, sendo abordados por diversos algoritmos com diferentes níveis de eficiência, complexidade e usos práticos; este trabalho busca explorar em profundidade os principais deles, detalhando definições, funcionamento, benefícios, limitações e exemplos de aplicação.

Este trabalho analisará os principais algoritmos de ordenação e busca sob uma perspectiva teórica e prática, destacando seu funcionamento e aplicações, com o suporte de referências bibliográficas que aprofundam o estudo, contribuindo assim para o desenvolvimento acadêmico e profissional por meio de um material que facilita a compreensão e implementação desses algoritmos essenciais à computação.

2. SELECTION SORT

2.1 DEFINIÇÃO

O Selection Sort (Ordenação por Seleção) é um algoritmo de ordenação comparativo que organiza uma lista dividindo-a em duas partes: uma parte ordenada (inicialmente vazia) e uma parte não ordenada (inicialmente o array inteiro). Ele funciona selecionando repetidamente o elemento de menor valor (ou maior, dependendo da ordem desejada) na parte não ordenada e movendo-o para a parte ordenada, através de uma troca com o primeiro elemento da parte não ordenada. Esse processo continua até que todos os elementos estejam ordenados.

2.2 ALGORITMO

O algoritmo pode ser descrito da seguinte forma:

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

2.3 VANTAGENS

- Simplicidade: É fácil de entender e implementar, tornando-o ideal para fins educacionais ou pequenos conjuntos de dados.
- Eficiência em Trocas: Realiza no máximo n−1 trocas, o que é vantajoso em situações onde a operação de troca é custosa (por exemplo, em memória flash).
- In-Place: Não requer memória extra além do array original, tendo complexidade de espaço O(1).
- Performance Independente da Entrada: O tempo de execução é sempre $O(n^2)$, independentemente de o array estar pré-ordenado ou não, o que pode ser previsível em alguns contextos.

2.4 DESVANTAGENS

- Ineficiência em Grandes Conjuntos: Com complexidade de tempo $O(n^2)$ no pior, médio e melhor caso, é muito lento para arrays grandes, sendo superado por algoritmos como Quick Sort ou Merge Sort.
- Não Estável: Não preserva a ordem relativa de elementos iguais, o que pode ser um problema em certas aplicações.

- Ineficiente em Comparações: Realiza muitas comparações (aproximadamente n2/2) mesmo quando o array já está parcialmente ordenado.
- Não Adaptativo: Não aproveita a estrutura existente do array para melhorar o desempenho.

3. INSERTION SORT

3.1 DEFINIÇÃO

O Insertion Sort (Ordenação por Inserção) é um algoritmo de ordenação comparativo que constroi uma lista ordenada elemento por elemento, inserindo cada item em sua posição correta dentro da porção já ordenada do array. Ele simula o processo de ordenar cartas em um jogo, onde cada nova carta é inserida no lugar apropriado entre as cartas já organizadas. É particularmente eficiente para pequenos conjuntos de dados ou arrays quase ordenados.

3.2 ALGORITMO

O algoritmo pode ser descrito da seguinte forma:

```
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;
    }
}
```

3.3 VANTAGENS

- Simplicidade: Fácil de implementar e entender, ideal para fins educacionais.
- Eficiência em Pequenos Conjuntos: Tem bom desempenho em arrays pequenos (n < 50) ou quase ordenados, com complexidade próxima de O(n) no melhor caso.

- Estável: Preserva a ordem relativa de elementos iguais, útil em aplicações específicas.
- Adaptativo: Ajusta-se à ordenação pré-existente, reduzindo comparações e deslocamentos em dados parcialmente ordenados.
- In-Place: Requer apenas O(1) de espaço adicional.

3.4 DESVANTAGENS

- Ineficiência em Grandes Conjuntos: Complexidade de tempo $O(n^2)$ no pior e médio caso, tornando-o lento para grandes arrays.
- Deslocamentos Frequentes: Realiza muitos deslocamentos de elementos, o que pode ser custoso em certas implementações.
- Não Ideal para Dados Aleatórios: Seu desempenho é ruim em arrays completamente desordenados, comparado a algoritmos como Quick Sort.

4. BUBBLE SORT

4.1 DEFINIÇÃO

O Bubble Sort é um algoritmo de ordenação comparativo simples que organiza uma lista comparando pares de elementos adjacentes e trocando-os se estiverem na ordem errada. Esse processo é repetido até que nenhuma troca seja necessária, assemelhando-se a "bolhas" subindo à superfície, daí o nome. É um dos algoritmos mais básicos e intuitivos, frequentemente usado para fins educacionais.

4.2 ALGORITMO

O algoritmo pode ser descrito da seguinte forma:

```java
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;
            }
        }
    }
}
```

4.3 VANTAGENS

- Simplicidade: Extremamente fácil de entender e implementar, ideal para ensino.
- Estabilidade: Preserva a ordem relativa de elementos iguais.
- In-Place: Requer apenas O(1) de espaço adicional.
- Detecta Ordenação Precoce: Na versão otimizada, para imediatamente se o array já estiver ordenado, reduzindo o tempo no melhor caso para O(n).

4.4 DESVANTAGENS

- Ineficiência: Complexidade de tempo $O(n^2)$ no pior e médio caso, tornando-o muito lento para grandes conjuntos de dados.
- Muitas Trocas: Realiza um número elevado de trocas desnecessárias, mesmo em arrays parcialmente ordenados (sem otimização).
- Não Adaptativo (sem otimização): Não aproveita a ordenação pré-existente, a menos que seja modificado com uma flag.
- Obsoleto em Prática: Superado por algoritmos mais rápidos como Quick Sort ou Merge Sort em quase todos os cenários reais.

5. HEAP SORT

5.1 DEFINIÇÃO

O Heap Sort (Ordenação por Heap) é um algoritmo de ordenação comparativo baseado na estrutura de dados conhecida como heap, geralmente um

max-heap (árvore binária onde o valor de cada nó é maior ou igual ao de seus filhos).

Ele funciona em duas etapas principais: primeiro, transforma o array em um heap; depois, extrai repetidamente o maior elemento do heap e o posiciona no final do array, reduzindo o tamanho do heap até que todos os elementos estejam ordenados. É eficiente e garante uma ordenação estável em termos de complexidade.

5.2 ALGORITMO

O algoritmo pode ser descrito da seguinte forma:

```
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);
    }
}
```

5.3 VANTAGENS

- Complexidade Garantida: Tem complexidade de tempo O(n log n) no pior, médio e melhor caso, o que o torna eficiente e previsível.
- In-Place: Apesar de usar a estrutura de heap, não requer espaço adicional além do array original (O(1) de espaço auxiliar).
- Não Requer Estrutura Extra: O heap é construído diretamente no array, sem necessidade de alocação dinâmica.
- Útil em Aplicações Específicas: Serve como base para filas de prioridade e outras estruturas baseadas em heap.

5.4 DESVANTAGENS

- Não Estável: Não preserva a ordem relativa de elementos iguais.
- Mais Lento na Prática: Apesar da complexidade O(n log n), constantes ocultas e acesso não sequencial à memória o tornam menos eficiente que Quick Sort em muitos casos.
- Complexidade de Implementação: Mais difícil de implementar e entender do que algoritmos simples como Bubble Sort ou Insertion Sort.
- Não Adaptativo: Não aproveita ordenação pré-existente no array.

6. RADIX SORT

6.1 DEFINIÇÃO

O Radix Sort constitui um algoritmo de ordenação eficiente e estável, apropriado para organizar elementos identificados por chaves distintas. Tais chaves, representadas por sequências de caracteres ou valores numéricos, são ordenadas pelo Radix Sort conforme uma disposição associada à ordem lexicográfica.

No âmbito da ciência da computação, trata-se de um método que estrutura inteiros ao processar seus dígitos de forma individual. Dado que inteiros podem simbolizar cadeias de caracteres (como nomes ou datas) e números de ponto flutuante devidamente ajustados, o Radix Sort demonstra aplicabilidade que transcende a mera ordenação de inteiros.

6.2 ALGORITMO

O algoritmo pode ser descrito da seguinte forma:

```c
void radixSort(int vetor[], int tamanho) {
    int i, exp;
    int maior = vetor[0];
    for (i = 1; i < tamanho; i++) {
        if (vetor[i] > maior)
            maior = vetor[i];
    }
    for (exp = 1; maior / exp > 0; exp *= 10) {
        int bucket[10] = {0};
        int *b = (int *)calloc(tamanho, sizeof(int));
        if (b == NULL) {
            printf("Erro: Falha na alocação de memória!\n");
            return;
        }
        for (i = 0; i < tamanho; i++)
            bucket[(vetor[i] / exp) % 10]++;
        for (i = 1; i < 10; i++)
            bucket[i] += bucket[i - 1];
        for (i = tamanho - 1; i >= 0; i--) {
            b[bucket[(vetor[i] / exp) % 10] - 1] = vetor[i];
            bucket[(vetor[i] / exp) % 10]--;
        }
        for (i = 0; i < tamanho; i++)
            vetor[i] = b[i];
        free(b);
    }
}
```

6.3 VANTAGENS

- Eficiência em casos específicos: Tem complexidade de tempo linear O(nk), onde n é o número de elementos e k é o número médio de dígitos, tornando-o mais rápido que algoritmos baseados em comparação (como Quick Sort, O(n log n)) para grandes conjuntos de dados com k pequeno.
- Não comparativo: Não depende de comparações entre elementos, o que o torna eficiente para dados com estrutura fixa (como inteiros ou strings).

- Estabilidade: Preserva a ordem relativa de elementos iguais, útil em aplicações como ordenação de registros com múltiplos campos.

6.4 DESVANTAGENS

- Uso de memória: Requer espaço adicional para os baldes, o que pode ser um problema em sistemas com memória limitada.
- Limitação de tipos de dados: Funciona bem apenas com dados que podem ser divididos em dígitos ou unidades (como inteiros ou strings), sendo menos versátil que algoritmos como Merge Sort.
- Dependência de k: Se o número de dígitos (k) for muito grande, a eficiência diminui, especialmente se k se aproxima de log n.
- Complexidade de implementação: Comparado a algoritmos mais simples como Bubble Sort, exige mais cuidado para lidar com os baldes e a estabilidade.

7. QUICK SORT

7.1 DEFINIÇÃO

O Quick Sort é um algoritmo de ordenação baseado em divisão e conquista (divide-and-conquer). Ele seleciona um elemento pivô, particiona o array ao redor desse pivô (colocando elementos menores à esquerda e maiores à direita) e, recursivamente, ordena as subpartes.

7.2 ALGORITMO

O algoritmo pode ser descrito da seguinte forma:

```c
void swap(int *a, int *b) {
    int temp = *a;
    *a = *b;
    *b = temp;
}

int partition(int vet[], int low, int high) {
    srand(time(NULL));
    int pivotIndex = low + rand() % (high - low + 1);
    swap(&vet[pivotIndex], &vet[high]);
    int pivot = vet[high];
    int i = low - 1;
    for (int j = low; j < high; j++) {
        if (vet[j] <= pivot) {
            i++;
            swap(&vet[i], &vet[j]);
        }
    }
    swap(&vet[i + 1], &vet[high]);
    return i + 1;
}

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

7.3 VANTAGENS

- Eficiência média: Complexidade média de O(n log n), tornando-o muito rápido para a maioria dos casos.
- Uso de memória in-place: Requer pouco espaço extra (O(log n) para a pilha de recursão), sendo eficiente em termos de memória.
- Flexibilidade: Pode ser adaptado com diferentes estratégias de escolha de pivô (ex.: mediana de três, pivô aleatório) para melhorar o desempenho.

7.4 DESVANTAGENS

- Pior caso: Complexidade de $O(n^2)$ ocorre quando o pivô é sempre o menor ou maior elemento (ex.: array já ordenado ou em ordem reversa, com pivô fixo no início ou fim).
- Não estável: Não preserva a ordem relativa de elementos iguais, o que pode ser um problema em algumas aplicações.
- Recursão profunda: Pode causar estouro de pilha em arrays muito grandes ou mal distribuídos, embora isso seja raro.

8. MERGE SORT

8.1 DEFINIÇÃO

O Merge Sort é um algoritmo de ordenação baseado no paradigma de divisão e conquista (divide-and-conquer). Ele divide o array de entrada em duas metades, ordena recursivamente cada metade e, em seguida, mescla (merge) as subpartes ordenadas para criar um array totalmente ordenado.

8.2 ALGORITMO

O algoritmo pode ser descrito da seguinte forma:

```c
void merge(int vet[], 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] = vet[l + i];
    for (int j = 0; j < n2; j++) R[j] = vet[m + 1 + j];
    int i = 0, j = 0, k = l;
    while (i < n1 && j < n2) {
        if (L[i] <= R[j]) {
            vet[k] = L[i];
            i++;
        } else {
            vet[k] = R[j];
            j++;
        }
        k++;
    }
    while (i < n1) {
        vet[k] = L[i];
        i++;
        k++;
    }
    while (j < n2) {
        vet[k] = R[j];
        j++;
        k++;
    }
}

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

8.3 VANTAGENS

- Eficiência garantida: Complexidade de tempo O(n log n) em todos os casos (melhor, médio e pior), independentemente da entrada.
- Estabilidade: Preserva a ordem relativa de elementos iguais, o que é útil em aplicações como ordenação de registros com múltiplos campos.
- Adequado para grandes datasets: Excelente para arrays grandes e para listas vinculadas (linked lists), onde não requer acesso aleatório.

- Previsibilidade: Desempenho consistente, sem variações drásticas como no Quick Sort.

8.4 DESVANTAGENS

- Uso de memória: Requer espaço extra O(n) para os arrays temporários durante a mesclagem, o que pode ser um problema em sistemas com memória limitada.
- Overhead para pequenos arrays: Menos eficiente para arrays pequenos devido ao overhead da recursão e da mesclagem, sendo melhor combiná-lo com algoritmos como Insertion Sort para tamanhos menores.
- Não in-place: Não ordena o array no local, exigindo memória adicional, o que o torna menos eficiente em termos de espaço em comparação com algoritmos como Quick Sort.

9. BUSCA SEQUENCIAL

9.1. DEFINIÇÃO

A Busca Sequencial, também conhecida como Busca Linear, é um algoritmo simples utilizado para encontrar um valor dentro de uma lista (ou array). Ele percorre a lista de elementos, verificando cada item sequencialmente até encontrar o valor desejado ou até o final da lista.

9.2. ALGORITMO

O algoritmo de Busca Sequencial pode ser descrito da seguinte maneira:

1. Inicialização: Inicie a busca a partir do primeiro elemento do array.
2. Comparação: Compare o valor do elemento atual com o valor que está sendo procurado.
3. Condição de Encontro: Se os valores forem iguais, retorne o índice correspondente ao elemento encontrado.
4. Iteração: Caso contrário, avance para o próximo elemento do array e repita o processo de comparação.

5. Condição de Término: Se o valor não for encontrado após a verificação de todos os elementos, retorne um valor que indique a ausência do item (comumente representado por -1).

9.3 IMPLEMENTAÇÃO EM C

O algoritmo pode ser descrito da seguinte forma:

```c
int buscaSequencial(int vet[], int tamanho, int chave) {
    for (int i = 0; i < tamanho; i++) {
        if (vet[i] == chave) {
            return i;
        }
    }
    return -1;
}

void imprimirVetor(int vet[], int tamanho) {
    for (int i = 0; i < tamanho; i++)
        printf("%d ", vet[i]);
    printf("\n");
}

int main() {
    int vet[] = {64, 34, 25, 12, 22, 11, 90};
    int tamanho = sizeof(vet) / sizeof(vet[0]);
    int chave = 25;
    imprimirVetor(vet, tamanho);
    int resultado = buscaSequencial(vet, tamanho, chave);
    if (resultado != -1) {
        printf("Elemento %d encontrado na posição %d\n", chave, resultado);
    } else {
        printf("Elemento %d não encontrado\n", chave);
    }
    return 0;
}
```

9.4 VANTAGENS

- O algoritmo de busca sequencial é de fácil compreensão e implementação, sendo uma solução direta para o problema de localização de elementos em uma lista.
- Diferente de outros métodos de busca, como a busca binária, a busca sequencial não requer que a lista esteja previamente ordenada, o que a torna aplicável a uma ampla variedade de cenários.
- Em casos onde o volume de dados é reduzido, a busca sequencial pode ser uma opção eficiente, uma vez que o tempo de execução é aceitável para listas de tamanho limitado.

9.5 DESVANTAGENS

- O algoritmo possui complexidade temporal O(n), o que significa que, no pior caso, ele precisa percorrer todos os elementos da lista para encontrar o valor desejado. Isso o torna ineficiente em termos de desempenho para listas grandes.
- Em cenários com listas de tamanho considerável, a busca sequencial tende a ser lenta, uma vez que o tempo de execução aumenta linearmente com o crescimento do número de elementos. Isso a torna inadequada para aplicações que exigem alta eficiência em grandes conjuntos de dados.

10. BUSCA BINÁRIA

10.1 DEFINIÇÃO

A Busca Binária é um algoritmo de busca eficiente que funciona em listas ordenadas. Ao invés de verificar cada elemento, como na busca sequencial, a busca binária divide a lista ao meio, verificando se o valor procurado está na metade inferior ou superior. Esse processo é repetido até encontrar o valor ou determinar que ele não está presente.

10.2 ALGORITMO

O algoritmo da Busca Binária pode ser descrito de forma estruturada e formal da seguinte maneira:

1. Defina os índices de início (baixo) e fim (alto) da lista.
2. Calcule o índice médio entre os índices baixo e alto.
3. Se o valor procurado for igual ao elemento no índice médio, retorne o índice.
4. Se o valor for menor que o valor no índice médio, ajuste o índice alto para o meio - 1.
5. Se o valor for maior, ajuste o índice baixo para o meio + 1.
6. Repita o processo até que o valor seja encontrado ou até que o índice baixo seja maior que o índice alto.
7. Se o valor não for encontrado, retorna -1

10.3 IMPLEMENTAÇÃO EM C

O algoritmo pode ser descrito da seguinte forma:

```c
int buscaBinaria(int vet[], int tamanho, int chave) {
    int esquerda = 0, direita = tamanho - 1;
    while (esquerda <= direita) {
        int meio = esquerda + (direita - esquerda) / 2;
        if (vet[meio] == chave) {
            return meio;
        }
        if (vet[meio] < chave) {
            esquerda = meio + 1;
        } else {
            direita = meio - 1;
        }
    }
    return -1;
}

void imprimirVetor(int vet[], int tamanho) {
    for (int i = 0; i < tamanho; i++)
        printf("%d ", vet[i]);
    printf("\n");
}

int main() {
    int vet[] = {11, 12, 22, 25, 34, 64, 90};
    int tamanho = sizeof(vet) / sizeof(vet[0]);
    int chave = 25;
    imprimirVetor(vet, tamanho);
    int resultado = buscaBinaria(vet, tamanho, chave);
    if (resultado != -1) {
        printf("Elemento %d encontrado na posição %d\n", chave, resultado);
    } else {
        printf("Elemento %d não encontrado\n", chave);
    }
    return 0;
}
```

10.4 VANTAGENS

- A sua complexidade de algoritmo, tornando-o muito mais rápido em listas grandes quando comparado à busca sequencial.
- A cada iteração, a lista é dividida pela metade, o que reduz o número de elementos a serem verificados.

10.5 DESVANTAGENS

- Para que a busca binária funcione, a lista precisa estar ordenada.
- Em comparação com a busca sequencial, a implementação da busca binária é mais difícil de entender e implementar corretamente.

11. CONCLUSÃO

Este estudo procedeu à análise dos algoritmos de ordenação (Selection Sort, Insertion Sort, Bubble Sort, Heap Sort, Radix Sort, Quick Sort e Merge Sort) e de busca (Sequencial e Binária), enfatizando suas definições, modos de funcionamento, vantagens, desvantagens e implementações. Ressalta-se que algoritmos como o Quick Sort e o Merge Sort destacam-se por sua eficiência de O(n log n) em grandes conjuntos de dados, enquanto o Selection Sort, Insertion Sort e o Bubble Sort apresentam limitações decorrentes de sua complexidade de $O(n^2)$.

Ademais, a Busca Sequencial, embora simples, revela-se ineficiente para volumes expressivos de dados, enquanto a Busca Binária demonstra elevado desempenho em listas previamente ordenadas, sendo imprescindível a ordenação prévia dos dados.

12. REFERÊNCIAS

https://www.passeidireto.com/arquivo/130825478/algoritmos-de-busca-sequencial-e-binaria
https://www.mundojs.com.br/2018/02/05/algoritmos-de-busca-sequencial-e-binaria/
https://giovanidacruz.com.br/merge-sort-um-metodo-eficaz-de-ordenacao/
https://blog.pantuza.com/artigos/o-algoritmo-de-ordenacao-quicksort
https://www.techiedelight.com/pt/heap-sort-place-place-implementation-c-c/
https://terminaldeinformacao.com/2013/05/10/ordenando-vetores-usando-linguagem-c/
https://www.italoinfo.com.br/algoritmos/selectionsort/index.php
https://www.blogcyberini.com/2018/06/insertion-sort.html
https://embarcados.com.br/algoritmos-de-ordenacao-bubble-sort/
https://www.guru99.com/pt/radix-sort.html