UNIVERSIDAD NACIONAL AUTÓNOMA DE MÉXICO

FACULTAD DE CIENCIAS

Comparación de Algoritmos Recusivos e Iterativos

Profesora

Verónica Esther Arriola Ríos

Autor

Arroyo Martínez Erick Daniel

ASIGNATURA

Introducción a Ciencias de la Computación

10 de mayo de 2024

Índice

1. Introducción

1.1. Definición de Algoritmos Recursivos

Los algoritmos recursivos son aquellos que se definen en términos de sí mismos, es decir, la solución de un problema se obtiene resolviendo una versión más pequeña del mismo problema. Se componen de una condición base que detiene la recursión y una regla de recursión que define cómo el problema actual se descompone en subproblemas más pequeños.

1.2. Definición de Algoritmos Iterativos

Los algoritmos iterativos resuelven problemas repitiendo una secuencia de instrucciones de forma iterativa, utilizando bucles como `for`, `while` o estructuras similares. Cada iteración se aproxima a la solución final, hasta que se alcance una condición de parada específica.

1.3. Importancia de la Comparación

Comparar algoritmos recursivos e iterativos es esencial para elegir el enfoque más adecuado en la solución de problemas. Ambos métodos tienen ventajas y desventajas en términos de complejidad temporal y espacial, lo que afecta el rendimiento general. Entender estas diferencias permite a los desarrolladores seleccionar el método que mejor se adapte a las necesidades de un problema en particular.

2. Algoritmos Recursivos

2.1. Características

Los algoritmos recursivos se caracterizan por:

- **Descomposición:** Se dividen los problemas en subproblemas más pequeños.
- **Condición Base:** Un caso que detiene la recursión para evitar una llamada infinita.
- **Llamada Recursiva:** Una función que se llama a sí misma en cada iteración.

2.2. Ventajas

- Fácil de entender y de implementar cuando el problema es inherentemente recursivo, como el cálculo de factoriales.
- Permite implementar algoritmos complejos con menos líneas de código.

2.3. Desventajas

- Consume más memoria, ya que cada llamada recursiva añade un nuevo marco al stack.
- Es propenso a errores si la condición base no está correctamente definida, lo que puede llevar a una recursión infinita.

2.4. Ejemplos Comunes

2.4.1. Factorial de un Número

El cálculo del factorial se logra fácilmente utilizando recursión. Aquí tienes un ejemplo en Java:

Listing 1: Factorial en Java

```
public static int factorial(int n) {
    // Condicion base
    if (n == 0) {
        return 1;
    }
    // Llamada recursiva
    return n * factorial(n - 1);
}
```

2.4.2. Búsqueda Binaria

La búsqueda binaria encuentra un valor en un array ordenado dividiéndolo recursivamente:

Listing 2: Búsqueda Binaria

```
public static int busquedaBinaria(int[] arr, int valor, int inicio, int fin) {
    if (inicio > fin) {
        return -1; // Valor no encontrado
    }
    int medio = inicio + (fin - inicio) / 2;

    if (arr[medio] == valor) {
        return medio;
    } else if (arr[medio] > valor) {
        return busquedaBinaria(arr, valor, inicio, medio - 1);
    } else {
        return busquedaBinaria(arr, valor, medio + 1, fin);
    }
}
```

3. Algoritmos Iterativos

3.1. Características

Los algoritmos iterativos presentan las siguientes características:

- **Estructura de Bucle:** Emplean bucles como `for` o `while` para repetir un conjunto de instrucciones.
- **Condición de Salida:** Una condición que detiene el bucle para evitar una iteración infinita.
- **Estado Controlado:** Las variables de control se actualizan en cada iteración para avanzar hacia la condición de salida.

3.2. Ventajas

- Generalmente consumen menos memoria que los algoritmos recursivos, ya que no se crea una pila de llamadas.
- Tienen un mayor control sobre el flujo de ejecución y manejo de errores.

3.3. Desventajas

- Algunos problemas no son tan intuitivos de implementar con iteración, especialmente los que se resuelven de forma natural mediante recursión.
- Pueden resultar más complejos en términos de líneas de código para implementar la misma lógica.

3.4. Ejemplos Comunes

3.4.1. Factorial de un Número

El cálculo del factorial de un número mediante iteración es directo y controlado:

Listing 3: Factorial iterativo

```
public static int factorialIterativo(int n) {
    int resultado = 1;
    for (int i = 1; i <= n; i++) {
        resultado *= i;
    }
    return resultado;
}
```

3.4.2. Búsqueda Binaria

La implementación iterativa de la búsqueda binaria evita el uso de recursión y se basa en un bucle:

Listing 4: Búsqueda binaria iterativa

```
public static int busquedaBinariaIterativa(int[] arr, int valor) {
    int inicio = 0, fin = arr.length - 1;

    while (inicio <= fin) {
        int medio = inicio + (fin - inicio) / 2;

        if (arr[medio] == valor) {
            return medio;
        } else if (arr[medio] > valor) {
            fin = medio - 1;
        } else {
            inicio = medio + 1;
        }
    }
    return -1; // Valor no encontrado
}
```

4. Comparación de Algoritmos Recursivos e Iterativos

4.1. Análisis de Complejidad

4.1.1. Complejidad Temporal

La complejidad temporal de un algoritmo mide cuánto tiempo toma ejecutarse en función del tamaño de la entrada.

Factorial

- **Algoritmo Recursivo:** La complejidad temporal del algoritmo recursivo para calcular el factorial es $O(n)$, ya que realiza n llamadas recursivas hasta alcanzar la condición base.
- **Algoritmo Iterativo:** La complejidad temporal del algoritmo iterativo para calcular el factorial también es $O(n)$, ya que ejecuta un bucle que itera n veces.

Búsqueda Binaria

- **Algoritmo Recursivo:** La complejidad temporal de la búsqueda binaria recursiva es $O(\log n)$, ya que en cada llamada recursiva el tamaño del array se reduce a la mitad.
- **Algoritmo Iterativo:** La complejidad temporal de la búsqueda binaria iterativa es también $O(\log n)$ por la misma razón: el array se divide en dos en cada iteración del bucle.

4.1.2. Complejidad Espacial

La complejidad espacial de un algoritmo mide la cantidad de memoria adicional que requiere su ejecución.

Factorial

- **Algoritmo Recursivo:** La complejidad espacial del algoritmo recursivo para calcular el factorial es $O(n)$, ya que cada llamada recursiva se agrega a la pila de llamadas, consumiendo memoria adicional.
- **Algoritmo Iterativo:** La complejidad espacial del algoritmo iterativo para calcular el factorial es $O(1)$, ya que utiliza un número constante de variables y no requiere memoria adicional proporcional al tamaño de la entrada.

Búsqueda Binaria

- **Algoritmo Recursivo:** La complejidad espacial de la búsqueda binaria recursiva es $O(\log n)$, debido a la profundidad de la recursión, que agrega marcos a la pila de llamadas.
- **Algoritmo Iterativo:** La complejidad espacial de la búsqueda binaria iterativa es $O(1)$, ya que solo utiliza un número constante de variables y no depende del tamaño de la entrada.

5. Conclusión

5.1. Cuándo Usar Algoritmos Recursivos

Los algoritmos recursivos son ideales para problemas que se pueden dividir de forma natural en subproblemas más pequeños, especialmente cuando la relación entre ellos es clara. Algunos casos en los que puede ser preferible usar recursión incluyen:

- **Estructuras de Datos Recursivas:** Problemas que trabajan con estructuras como árboles, listas enlazadas, y grafos.
- **Dividir y Vencerás:** Algoritmos como Quicksort y Mergesort que dividen el problema en partes más pequeñas.
- **Problemas de Retroceso:** Problemas como el N-Queens o la generación de combinaciones.
- **Programación Dinámica:** Algoritmos que resuelven subproblemas repetidos, como el cálculo de Fibonacci.

A pesar de sus ventajas, es importante tener en cuenta la profundidad de las llamadas recursivas, ya que la pila de llamadas puede llenarse rápidamente.

5.2. Cuándo Usar Algoritmos Iterativos

Los algoritmos iterativos pueden ser la mejor opción cuando el problema no requiere una estructura recursiva o cuando es crítico minimizar el uso de memoria. Algunos casos en los que se recomienda el uso de la iteración incluyen:

- **Estructuras Lineales:** Procesar datos en estructuras como arrays o listas.
- **Repetición Controlada:** Bucles que necesitan un control preciso del flujo de ejecución.
- **Optimización de Memoria:** Algoritmos donde la eficiencia espacial es crucial.
- **Problemas Sencillos:** Ejemplos como el cálculo del factorial o el recorrido de un array.

En estos casos, la iteración permite un control más directo sobre la secuencia de ejecución y a menudo es más eficiente en términos de memoria.

6. Referencias

- Singh, V. (2024, 9 febrero). Difference Between Recursion and Iteration - Shiksha Online. Shiksha.com. `https://www.shiksha.com/online-courses/articles/difference-between-recursion-and-iteration/#:~:text=Recursion%20is%20a%20technique%20in,until%20the%20condition%20is%20unmet.`
- GeeksforGeeks. (2023, 22 mayo). Difference between Recursion and Iteration. GeeksforGeeks. `https://www.geeksforgeeks.org/difference-between-recursion-and-iteration/`
- Singh, H., Singh, H. (2023, 12 julio). Practical Examples of the Big O Notation — Baeldung on Computer Science. Baeldung On Computer Science. `https://www.baeldung.com/cs/big-oh-asymptotic-complex`
- Olawanle, J. (2023, 10 abril). Big o cheat sheet – Time Complexity chart. freeCodeCamp.org. `https://www.freecodecamp.org/news/big-o-cheat-sheet-time-complexity-chart/`