Implementación y análisis del algoritmo de búsqueda galopante

Benjamin Alejandro Mosso Miller
Diego Alexis Salazar Jara

29 de mayo de 2025

1. Respuestas del Problema 1

1. **Implementación Búsqueda Binaria y Galopante:** Se implementó el algoritmo de búsqueda binaria de dos formas distintas para comparar su rendimiento al buscar las claves indicadas en el enunciado (1.000, 5.000, 10.000, 50.000, 100.000).

```cpp
int binarySearchIterative(const vector<int> &arr, size_t l, size_t r, int x)
{
    while (r >= l)
    {
        size_t mid = l + (r - l) / 2;
        if (arr[mid] == x)
            return mid;
        if (arr[mid] > x)
            r = mid - 1;
        else
            l = mid + 1;
    }
    return -1;
}
```

Código 1: Algoritmo de Búsqueda Binaria Iterativa.

```cpp
int binarySearchRecursive(const vector<int> &arr, int l, int r, int x)
{
    if (l > r)
        return -1;

    int mid = l + (r - l) / 2;
    if (arr[mid] == x)
        return mid;
    else if (x < arr[mid])
        return binarySearchRecursive(arr, l, mid - 1, x);
    else
        return binarySearchRecursive(arr, mid + 1, r, x);
}
```

Código 2: Algoritmo de Búsqueda Binaria Recursiva.

```cpp
int gallopingSearch(const std::vector<int> &arr, size_t size, int x)
{
    if (size == 0)
        return -1;
    if (arr[0] == x)
        return 0;

    size_t linf = 1;
    while (linf * 2 < size && arr[2 * linf] < x)
    {
        linf *= 2;
    }

    return binarySearchIterative(arr, linf, std::min(2 * linf, size - 1), x);
}
```

Código 3: Algoritmo de Búsqueda Galopante o Exponencial.

2. Se generó al azar un arreglo de tamaño 5.000.000.000 con enteros entre [1, 10.000.000], el cual fue ordenado utilizando un algoritmo de ordenamiento (QuickSort). Luego, se generaron aleatoriamente las claves 1.000, 5.000, 10.000, 50.000 y 100.000. Finalmente, se completó la tabla con los tiempos de búsqueda registrados para cada algoritmo.

```cpp
const int MIN_VALUE = 1;
const int MAX_VALUE = 10'000'000;

std::vector<int> generate_array_parallel(size_t size)
{
    std::vector<int> arr(size);

#pragma omp parallel
    {
        std::mt19937 gen(1234 + omp_get_thread_num());
        std::uniform_int_distribution<> dis(MIN_VALUE, MAX_VALUE);

#pragma omp for schedule(static)
        for (size_t i = 0; i < size; ++i)
        {
            arr[i] = dis(gen);
        }
    }

    return arr;
}
```

Código 4: Generar arreglo con números enteros al azar.

```cpp
int partition(std::vector<int> &arr, int low, int high)
{
   int i = low - 1;
   int pivot = arr[high];
   for (int j = low; j < high; j++)
   {
      if (arr[j] <= pivot)
      {
         i++;
         std::swap(arr[i], arr[j]);
      }
   }
   std::swap(arr[i + 1], arr[high]);
   return i + 1;
}

void quickSort(std::vector<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);
   }
}
```

Código 5: Implementación algoritmo de ordenamiento (QuickSort).

```cpp
std::vector<int> generate_keys(size_t count)
{
    std::vector<int> keys(count);
    std::mt19937 gen(std::random_device{}());
    std::uniform_int_distribution<> dis(MIN_VALUE, MAX_VALUE);
    for (size_t i = 0; i < count; ++i)
    {
        keys[i] = dis(gen);
    }

    return keys;
}
```

Código 6: Generar claves aleatorias.

Número de Claves	Búsqueda Binaria I.	Búsqueda Binaria R.	Búsqueda Galopante
1.000	1.94 ms	0.59 ms	2.37 ms
5.000	7.76 ms	2.77 ms	11.75 ms
10.000	14.84 ms	5.48 ms	23.98 ms
50.000	72.64 ms	26.26 ms	119.91 ms
100.000	144.74 ms	51.12 ms	239.10 ms

Tabla 1: Comparación de rendimiento de algoritmos de búsqueda.

3. De acuerdo a los datos de la tabla anterior, ¿Cuál algoritmo es mejor?

   Según los datos de la Tabla 1, el algoritmo de *búsqueda binaria recursiva* presenta mejores tiempos de ejecución en comparación con la *búsqueda binaria iterativa*. Sin embargo, se observa que la *búsqueda galopante/exponencial* tiene un mayor tiempo de ejecución conforme aumenta el tamaño del arreglo. Esto se debe a que el algoritmo de búsqueda galopante requiere más tiempo para encontrar el intervalo adecuado antes de aplicar la búsqueda binaria, lo que afecta su rendimiento frente a las implementaciones de búsqueda binaria, tanto recursiva como iterativa.

4. Justifique haciendo un estudio teórico de ambos algoritmos.

   - **Búsqueda Binaria Recursiva:**

   $$T(n) = T(n/2) + C \tag{1}$$

   Por Teorema Maestro Simplificado:

   $$T(N) = \begin{cases} \Theta\left(n^d\right), & \text{si } a < b^d \\ \Theta\left(n^d \log n\right), & \text{si } a = b^d \\ \Theta\left(n^{\log_b a}\right), & \text{si } a > b^d \end{cases}$$

   Identificar constantes:

   $$a = 1, \quad b = 2, \quad d = 0 \tag{2}$$

   Dado que $a = b^d$, es decir $1 = 2^0$, se cumple el segundo caso del teorema maestro simplificado, por lo que:

   $$T(N) = \Theta\left(n^0 \log n\right) = \Theta(\log n) \tag{3}$$

   - **Búsqueda Galopante/Exponencial:**

   Esta búsqueda consta de dos etapas:

   a) Primero, se realizan saltos exponenciales para encontrar un límite superior para el índice donde se encuentra la llave buscada. Este proceso tarda aproximadamente $\log(i)$ pasos, siendo i el índice real donde está el elemento.

   b) Luego, se aplica una búsqueda binaria dentro del intervalo acotado, que tiene un costo de $\Theta(\log(n))$, donde n es el tamaño del intervalo o la lista.

   Por tanto, el tiempo de ejecución se puede expresar como:

   $$T(n) = \log(i) + \Theta(\log(n)) \tag{1}$$

   Considerando que ambas etapas tienen complejidades del mismo orden, se puede simplificar a:

   $$T(n) = 2\Theta(\log(i)) \tag{2}$$

   Finalmente, quedando:

   $$T(n) = \Theta(\log(i)) \tag{3}$$

2. Anexos

El servidor está montado en la red universitaria de la Universidad del Bío-Bío. A continuación, se presentan las características principales del hardware y la memoria del sistema:

Procesador (CPU)	**Memoria RAM**
Intel(R) Xeon(R) CPU X5560 @ 2.80GHz	76 GiB

Tabla 2: Características principales del hardware del servidor