

DATA STRUCTURES AND ALGORITHMS

Searching algorithms of the sorted lists:

Jump search, Binary search, Interpolation search

Summary of the previous lecture

- External sorting
 - Merge sort
- What is difference between external and internal sorting?
- Introduction to Searching Algorithms

Search is a process to determine if an element with a particular value is a member of a particular set.
- Sequential search

Running time: $\frac{n+1}{2} \leq \mathbf{T}(n) \leq n.$

Searching in sorted arrays

For large collections of records that are searched repeatedly, sequential search is unacceptably slow.

One way to reduce search time is to preprocess the records by sorting them.

If the current element in array **L** is greater than value **K**, then we know that **K** cannot appear later in the array, and we can quit the search early.

This is an example of a divide and conquer algorithm.

But this still does not improve the worst-case cost of the searching algorithm.

Jump search

If we look first at position 1 in sorted array **L** and find that **K** is bigger, then we rule out positions 0 and 1, because more is often better.

What if we start searching from position **2** in array **L** and find that **K** is still bigger? This rules out positions 0, 1, and 2 with one comparison! 😊

What if we carry this to the extreme and look first at **the last** position in **L** and find that **K** is bigger? Then we know in only one comparison that **K** is not in **L**. 😊😊😊

But what is wrong with this approach?

Jump search

What is the right amount to jump?

Jump search algorithm

1. For some value j , we check every j 'th element in array \mathbf{L} , that is, we check elements $\mathbf{L}[j]$, $\mathbf{L}[2*j]$, and so on till $\mathbf{L}[m*j] > \mathbf{K}$.
2. When value $\mathbf{L}[m*j] > \mathbf{K}$, we do a linear search on the interval $[(m-1)*j; m*j]$ of length $j - 1$ that we know that \mathbf{K} exists.

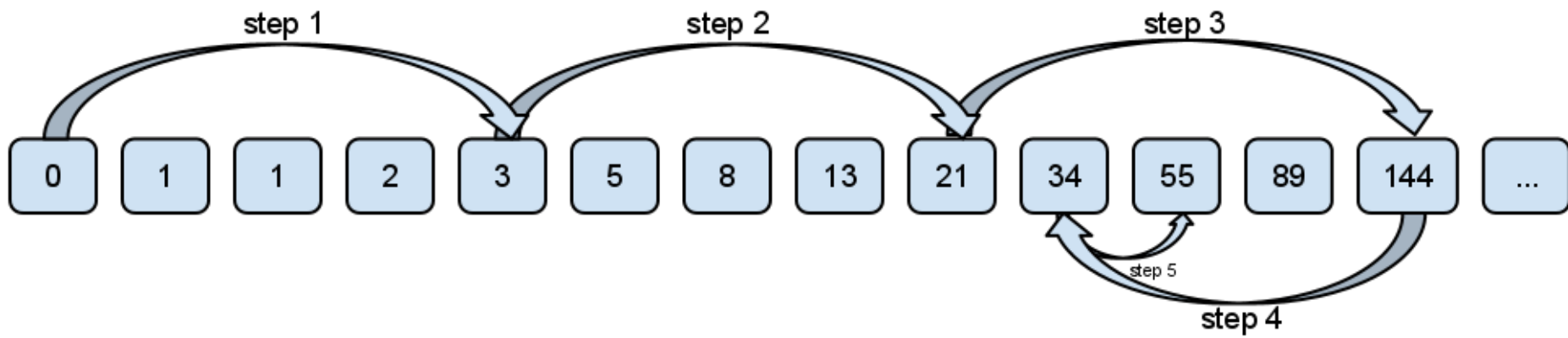
If $m*j < n < (m+1)*j$, then

the total running time of this algorithm is at most $m+j-1$

$$\mathbf{T}(n, j) = m + j - 1 = \left\lfloor \frac{n}{j} \right\rfloor + j - 1.$$

Example

Jump is = 4



Jump search

What is the best value that we can pick for j ?

Answer: We can find optimal j by minimizing the $T(n)$:

$$\min_{1 \leq j \leq n} \left\{ \left\lfloor \frac{n}{j} \right\rfloor + j - 1 \right\}$$

Solution:

Take the derivative and solve equation:

$$f'(j) = 0$$

to find the minimum, which is $j = \sqrt{n}$.

Best running time is $T(n) = O(\sqrt{n})$

Jump search

Lets extend jump search to three levels search.

We would first make jumps of some size j to find a sublist of size $j - 1$ whose end values bracket value K .

Then work through this sublist by making jumps of some smaller size, say j_1 .

Finally, once we find a bracketed sublist of size $j_1 - 1$, we would do sequential search to complete the process.

Recursion can be used for three levels search.

For an k -level jump search the optimum jump size j for the l th level (counting from 1) is $n^{(k-l)/k}$.

Example

```
#define min(x, y) ( (x) < (y) ? (x) : (y) )
int JumpSearch (int a[ ], int n, int K )
{
    int t = 0;
    int b = (int)sqrt(n);
    while (a [min(b,n) - 1] < K) {
        t = b;
        b = b + (int)sqrt(n);
        if ( t >= n) return -1 ;
    }
    while (a[t] < K ) {
        t = t+1;
        if ( t == min(b,n) )
            return -1 ;
        if ( a[t] == K ) {
            return t;
        }
    }
}
```

```
int main() {
    int a[7] = {3,7,9,12,14,15,16};
    int K =14;
    printf("Element found at the
position %d/n", JumpSearch(a, 7,K) );
    return 0;
}
```

Binary search

Binary search begins by examining the value in the **middle position** of the array **L**.

If $L[\text{mid}] = K$, then processing stops immediately.

If $L[\text{mid}] > K$, then search continues at the lower half of the array **L**.

If $L[\text{mid}] < K$, then search continues at the upper half of the array **L**.

Binary search

Next cycle

1. Next looks at the middle position in that part of the array where value **K** may exist.
2. The value at this position again allows to eliminate half of the remaining positions from consideration.
3. This process repeats until either the desired value is found, or there are no positions remaining in the array that might contain the value **K**.

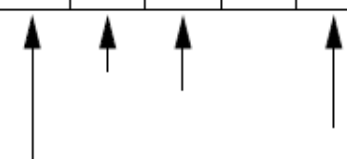
Running time:

worst case $T(n) = T(n/2) + 1$ for $n > 1$; $T(1) = 1$

Average case $T(n) = \log n$

Binary search

Position	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Key	11	13	21	26	29	36	40	41	45	51	54	56	65	72	77	83



Consider a search for the position with value $K = 45$. Binary search first checks the value at position 7. Because $41 < K$, the desired value cannot appear in any position below 7 in the array. Next, binary search checks the value at position 11.

Because $56 > K$, the desired value (*if it exists*) must be between positions 7 and 11. Position 9 is checked next. Again, its value is too great. The final search is at position 8, which contains the desired value. Thus, function returns position 8.

Alternatively, if K was 44, then the same series of record accesses would be made. After checking position 8, function would return a value of -1, indicating that the search is unsuccessful.

Example

```
int BinarySearch (int A[ ], int n, int K) {
    int low = -1;
    int high = n;                // low and high are beyond array bounds
    while ( low +1 != high) {    // Stop when low and high meet
        int i = (low + high)/2;  // Check middle of remaining subarray
        if ( K < A[i] )
            high = i;           // In left half
        if ( K == A[i] )
            return i;          // Found it
        if ( K > A[i] )
            low = i;           // In right half
    }
    return -1;                  // K value not found
}
```

Interpolation search

If we know nothing about the distribution of key values, **then binary search is the best algorithm** available for searching a sorted array.

However, sometimes we do know something about the expected key distribution.

Consider the typical behavior of a person **looking up a word** in a large dictionary. **Most people certainly do not use sequential search!**

A person looking for a word starting with 'S' generally assumes that entries beginning with 'S' start about three quarters of the way through the dictionary. Thus, he will first open the dictionary about three quarters of the way through and then make a decision based on what they find as to where to look next. In other words, people typically use some knowledge about the expected distribution of key values to “compute” where to look next.

Interpolation search

In an interpolation search, we search $\mathbf{L}[i]$ at a position \mathbf{p} that is appropriate to the value of \mathbf{K} as follows:

$$p = \frac{K - \mathbf{L}[1]}{\mathbf{L}[n] - \mathbf{L}[1]} * n$$

At each stage it computes a probe position then as with the binary search, moves either the upper or lower bound in to define a smaller interval containing the \mathbf{K} value. Unlike the binary search which guarantees a halving of the interval's size with each stage, interpolation *reduce/increase* the middle index by particular value.

Running time: worst case $\mathbf{T}(n) = \mathbf{O}(n)$

average case $\mathbf{T}(n) = \mathbf{O}(\log \log n)$

Example

```
int InterpolationSearch (int a[ ], int n, int K) {
    int low = 0;
    int high = n;
    int mid;
    while ( a[low] <= K && a[high] >= K) {
        mid = low + ( ( K - a[low] ) * (high - low)) / ( a[high] - a[low]);
        if (a[mid] < K)
            low = mid + 1;
        else if ( a[mid] > K)
            high = mid - 1;
        else
            return mid;
    }
    if (a[low] == K)
        return low;
    else
        return -1;           // K value not found
}
```