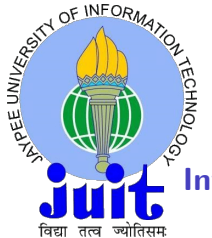


Data Structure

(Course Code: 25B11CI318)

Interpolation and Median Search

Mr. Sandeep Kumar Patel

Assistant Professor

Jaypee University of Information Technology

Waknaghat, Solan, HP-173234

Interpolation Search

Definition

Interpolation Search is an efficient searching algorithm for a **sorted array** in which the elements are approximately uniformly distributed.

Unlike Binary Search, which always checks the middle element, Interpolation Search estimates the probable position of the search key.

Binary Search $\rightarrow$ Middle Position

Interpolation Search $\rightarrow$ Estimated Position

Prerequisite

Interpolation Search requires:

- The array must be sorted.
- Elements should preferably be uniformly distributed.
- Random access should be available.

Example of approximately uniform data:

$$[10,\ 20,\ 30,\ 40,\ 50,\ 60,\ 70,\ 80]$$

Non-uniform data:

$$[1,\ 2,\ 3,\ 4,\ 100,\ 500,\ 1000]$$

Important

Interpolation Search performs best when the values are approximately uniformly distributed.

Idea of Interpolation Search

Suppose:

$$A = [10,\ 20,\ 30,\ 40,\ 50,\ 60,\ 70,\ 80]$$

Search for:

$$key = 70$$

Binary Search checks approximately the middle:

$$40$$

But interpolation search asks:

Where should 70 approximately occur?

Since 70 is close to the upper end of the range, the algorithm checks a position close to the upper end.

Interpolation Formula

The estimated position is calculated using:

$$pos = low + \frac{(key - A[low])(high - low)}{A[high] - A[low]}$$

where:

- low = lower index
- $high$ = upper index
- key = element being searched
- $A[low]$ = minimum value in current range
- $A[high]$ = maximum value in current range

The formula estimates where the key is likely to be located.

Interpolation Formula: Intuition

The formula is based on the idea of a straight line.
Suppose:

$$A[low] \leq key \leq A[high]$$

The fraction:

$$\frac{key - A[low]}{A[high] - A[low]}$$

tells us approximately how far the key is between the minimum and maximum values.
Therefore:

$$pos = low + \text{fraction} \times (high - low)$$

Estimate position instead of always taking the middle

Interpolation Search: Example

Consider:

$$A = [10,\ 20,\ 30,\ 40,\ 50,\ 60,\ 70,\ 80,\ 90]$$

Search for:

$$\boxed{70}$$

Initially:

$$low = 0, \qquad high = 8$$

$$A[low] = 10, \qquad A[high] = 90$$

Using:

$$pos = 0 + \frac{(70 - 10)(8 - 0)}{90 - 10}$$

$$pos = \frac{60 \times 8}{80}$$

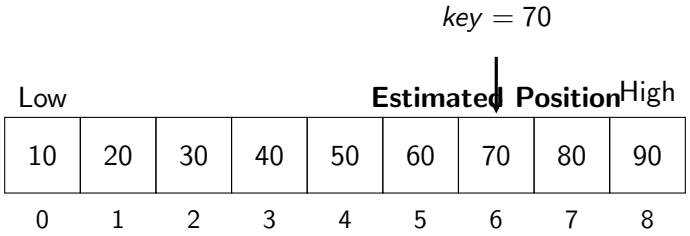
$$\boxed{pos = 6}$$

Since:

$$A[6] = 70$$

The element is found immediately.

Interpolation Search: Visual Example



$A[6] = 70 \Rightarrow$ Found

Interpolation Search: Steps

1. Set:
$$low = 0, \qquad high = n - 1$$
2. Check whether the key lies within:
$$A[low] \leq key \leq A[high]$$
3. Calculate the estimated position:
$$pos = low + \frac{(key - A[low])(high - low)}{A[high] - A[low]}$$
4. Compare $A[pos]$ with key.
5. If equal, return pos.
6. If $A[pos] < key$:
$$low = pos + 1$$
7. Otherwise:
$$high = pos - 1$$

Interpolation Search: Pseudocode

```
InterpolationSearch(A, n, key)
low = 0; high = n - 1
while low <= high  and key >= A[low] and key <= A[high]
    if A[high] == A[low]
        if A[low] == key
            return low
        else
            return -1
    pos = low +((key-A[low])*(high-low)) /(A[high]-A[low])
    if A[pos] == key
        return pos
    else if A[pos] < key
        low = pos + 1
    else
        high = pos - 1
return -1
```

Example: Key Near the Beginning

Consider:

$$A = [10,\ 20,\ 30,\ 40,\ 50,\ 60,\ 70,\ 80,\ 90]$$

Search for:

$$key = 20$$

Interpolation estimates:

$$pos = 0 + \frac{(20 - 10)(8)}{90 - 10}$$

$$pos = \frac{80}{80}$$

$$\boxed{pos = 1}$$

Since:

Interpolation Search: Complexity

Case	Time Complexity
Best Case	$O(1)$
Average Case	$O(\log \log n)$
Worst Case	$O(n)$

For approximately uniformly distributed data:

$$\boxed{O(\log \log n)}$$

In the worst case, if the distribution is highly non-uniform:

$$\boxed{O(n)}$$

Why $O(\log \log n)$?

Binary Search approximately halves the search space:

$$n \to \frac{n}{2} \to \frac{n}{4} \to \cdots$$

Therefore:

$$\boxed{O(\log n)}$$

Interpolation Search can make much more accurate position estimates when the data is uniformly distributed.

The search space can shrink much faster. Average complexity:

$$\boxed{O(\log \log n)}$$

Important

The $O(\log \log n)$ bound depends on assumptions about the distribution of the data.

Space Complexity

The iterative implementation uses only a few variables:

- *low*
- *high*
- *pos*
- *key*

No additional data structure is required.
Therefore:

$$\boxed{S(n) = O(1)}$$

Interpolation Search vs Binary Search

Feature	Binary Search	Interpolation Search
Sorted Data	Required	Required
Position	Middle	Estimated
Best Case	$O(1)$	$O(1)$
Average Case	$O(\log n)$	$O(\log \log n)$
Worst Case	$O(\log n)$	$O(n)$
Space	$O(1)$	$O(1)$

Key Difference

Binary Search depends mainly on the **position**, whereas Interpolation Search uses the **values** to estimate the position.

When Should We Use Interpolation Search?

Interpolation Search is suitable when:

- Data is sorted.
- Data is approximately uniformly distributed.
- Random access is available.
- The dataset is large.

Example:

$$[10,\ 20,\ 30,\ 40,\ 50,\ 60, \ldots]$$

is a good candidate.

Highly skewed data such as:

$$[1,\ 2,\ 3,\ 4,\ 5,\ 1000,\ 10000]$$

may result in poor performance.

Applications of Interpolation Search

Interpolation Search can be useful for:

- Searching large sorted numerical datasets.
- Searching uniformly distributed keys.
- Numerical databases.
- Index structures.
- Searching records with approximately uniform keys.

Practical Observation

For general-purpose searching, Binary Search is often preferred because its $O(\log n)$ worst-case guarantee does not depend on uniform data distribution.

Interpolation Search: Key Takeaways

1. Interpolation Search works on a **sorted array**.

Interpolation Search: Key Takeaways

1. Interpolation Search works on a **sorted array**.
2. It estimates the position of the key.

Interpolation Search: Key Takeaways

1. Interpolation Search works on a **sorted array**.
2. It estimates the position of the key.
3. The position is calculated using:

$$pos = low + \frac{(key - A[low])(high - low)}{A[high] - A[low]}$$

Interpolation Search: Key Takeaways

1. Interpolation Search works on a **sorted array**.
2. It estimates the position of the key.
3. The position is calculated using:

$$pos = low + \frac{(key - A[low])(high - low)}{A[high] - A[low]}$$

4. It performs very well for uniformly distributed data.

Interpolation Search: Key Takeaways

1. Interpolation Search works on a **sorted array**.
2. It estimates the position of the key.
3. The position is calculated using:

$$pos = low + \frac{(key - A[low])(high - low)}{A[high] - A[low]}$$

4. It performs very well for uniformly distributed data.
5. Average complexity:

$$\boxed{O(\log \log n)}$$

Interpolation Search: Key Takeaways

1. Interpolation Search works on a **sorted array**.
2. It estimates the position of the key.
3. The position is calculated using:

$$pos = low + \frac{(key - A[low])(high - low)}{A[high] - A[low]}$$

4. It performs very well for uniformly distributed data.
5. Average complexity:

$$\boxed{O(\log \log n)}$$

6. Worst-case complexity:

$$\boxed{O(n)}$$

Median and Selection Problem

Median

The **median** is the middle element when the elements are arranged in sorted order.

For n elements:

$$\begin{cases} \text{Median position} = \dfrac{n+1}{2}, & n \text{ odd} \\ \text{Median} = \text{average of two middle elements}, & n \text{ even} \end{cases}$$

Example:

$$A = [12, 5, 8, 20, 3, 15, 10]$$

Sorted array:

$$[3, 5, 8, 10, 12, 15, 20]$$

Selection Problem

Selection Problem

The **selection problem** is to find the k-th smallest element in an unsorted array without necessarily sorting the entire array.

Examples:

$$k = 1 \Rightarrow \text{Minimum}$$

$$k = n \Rightarrow \text{Maximum}$$

$$k = \left\lceil \frac{n}{2} \right\rceil \Rightarrow \text{Median}$$

Key Idea

Finding the median can be treated as a special case of the general k-**th smallest element** problem.

Partition-Based Selection

The idea is similar to the partition operation used in Quick Sort.

Choose a pivot and partition the array into:

Elements smaller than pivot

Pivot

Elements greater than pivot

After partitioning, suppose the pivot reaches position p.

- If $p = k$, the pivot is the required element.
- If $k < p$, search only the left part.
- If $k > p$, search only the right part.

Only one partition is explored

Partition: Example

Consider:

$$A = [12, 5, 8, 20, 3, 15, 10]$$

Choose:

$$\boxed{\text{Pivot} = 10}$$

After partitioning:

$$[5, 8, 3] \quad \boxed{10} \quad [12, 20, 15]$$

There are three elements smaller than 10. Therefore, 10 is the:

$$(3+1)^{th} = 4^{th}$$

smallest element. If we need:

$$k = 4$$

then:

$$\boxed{\text{Answer} = 10}$$

Partition-Based Selection: Algorithm

1. Choose a pivot.
2. Partition the array around the pivot.
3. Let the final position of the pivot be p.
4. Compare p with the required rank k.
   - If $p = k$, return the pivot.
   - If $k < p$, continue in the left subarray.
   - If $k > p$, continue in the right subarray.
5. Repeat until the required element is found.

Selection avoids sorting the entire array

Quickselect Algorithm

```
QuickSelect(A, low, high, k)

if low == high
    return A[low]

p = Partition(A, low, high)

if p == k
    return A[p]

else if k < p
    return QuickSelect(A, low, p-1, k)

else
    return QuickSelect(A, p+1, high, k)
```

Complexity of Partition-Based Selection

Partitioning an array of size n requires:

$$O(n)$$

time.

If the pivot is reasonably balanced:

$$T(n) = T\left(\frac{n}{2}\right) + O(n)$$

Therefore:

$$\boxed{T(n) = O(n)}$$

However, if the pivot is always the smallest or largest element:

$$T(n) = T(n-1) + O(n)$$

which gives:

$$\boxed{T(n) = O(n^2)}$$

Randomized Selection

The performance of Quickselect depends heavily on the choice of pivot.

Instead of always choosing a fixed position, choose the pivot **randomly**.

This gives:

Randomized Quickselect

Advantages:

- Avoids consistently poor pivot choices.
- Expected linear running time.
- Simple to implement.
- Does not require the array to be sorted.

Randomized Partition

```
RandomizedPartition(A, low, high)

r = Random(low, high)

swap(A[r], A[high])

return Partition(A, low, high)
```

The randomly selected element is moved to the partition position.

Random Pivot $\rightarrow$ Partition

Randomized Select

```
RandomizedSelect(A, low, high, k)

if low == high
    return A[low]

p = RandomizedPartition(A, low, high)

if p == k
    return A[p]

else if k < p
    return RandomizedSelect(A, low, p-1, k)

else
    return RandomizedSelect(A, p+1, high, k)
```

Randomized Selection: Complexity

Each partition requires:

$$O(n)$$

time.

Because the pivot is selected randomly, the expected size of the remaining problem decreases sufficiently fast.

Expected recurrence:

$$T(n) = T(\text{smaller expected subproblem}) + O(n)$$

Therefore:

$$\boxed{E[T(n)] = O(n)}$$

Worst case is still possible:

Why Do We Need Median-of-Medians?

Randomized Selection has:

$$\boxed{\text{Expected } O(n)}$$

but its worst case can still be:

$$\boxed{O(n^2)}$$

Can we choose a pivot that guarantees a **good partition**?

Why Do We Need Median-of-Medians?

Randomized Selection has:

$$\boxed{\text{Expected } O(n)}$$

but its worst case can still be:

$$\boxed{O(n^2)}$$

Can we choose a pivot that guarantees a **good partition**?
Yes.

$$\boxed{\text{Median-of-Medians}}$$

The algorithm selects a pivot deterministically such that a constant fraction of elements is guaranteed to be discarded at every iteration.

Median-of-Medians: Basic Idea

Suppose we have:

$$A = [\text{unsorted elements}]$$

Divide the elements into groups of:

$$\boxed{5}$$

For example:

12	5	8	20	3
15	10	7	18	6
25	11	14	2	9

Sort each group and find its median.

Then recursively find the median of these medians.

Median-of-Medians: Example

Consider:

$$A = [12, 5, 8, 20, 3,\ 15, 10, 7, 18, 6,\ 25, 11, 14, 2, 9]$$

Divide into groups of five:

$$\begin{array}{ccccc} 12 & 5 & 8 & 20 & 3 \\ 15 & 10 & 7 & 18 & 6 \\ 25 & 11 & 14 & 2 & 9 \end{array}$$

Sort each group:

$$[3, 5, 8, 12, 20]$$

$$[6, 7, 10, 15, 18]$$

$$[2, 9, 11, 14, 25]$$

Median-of-Medians: Steps

1. Divide the array into groups of at most five elements.

Median-of-Medians: Steps

1. Divide the array into groups of at most five elements.
2. Sort each group.

Median-of-Medians: Steps

1. Divide the array into groups of at most five elements.
2. Sort each group.
3. Find the median of each group.

Median-of-Medians: Steps

1. Divide the array into groups of at most five elements.
2. Sort each group.
3. Find the median of each group.
4. Recursively find the median of these medians.

Median-of-Medians: Steps

1. Divide the array into groups of at most five elements.
2. Sort each group.
3. Find the median of each group.
4. Recursively find the median of these medians.
5. Use this value as the pivot.

Median-of-Medians: Steps

1. Divide the array into groups of at most five elements.
2. Sort each group.
3. Find the median of each group.
4. Recursively find the median of these medians.
5. Use this value as the pivot.
6. Partition the original array around this pivot.

Median-of-Medians: Steps

1. Divide the array into groups of at most five elements.
2. Sort each group.
3. Find the median of each group.
4. Recursively find the median of these medians.
5. Use this value as the pivot.
6. Partition the original array around this pivot.
7. Recursively search only the required partition.

Median-of-Medians: Pseudocode

```
Select(A, k)
if size(A) <= 5
    sort(A)
    return A[k]
Divide A into groups of 5
Find the median of each group
pivot = Select(medians, middle)
Partition A around pivot
if pivot has rank k
    return pivot
else if k < pivot rank
    Select(left part, k)
else
    Select(right part, k - pivot rank)
```

Why Median-of-Medians is Linear

The median-of-medians pivot guarantees that a constant fraction of elements lies on both sides of the pivot.

For groups of five, at least approximately:

$$\frac{3n}{10}$$

elements are guaranteed to be on each side (up to small boundary effects).

Therefore, the remaining recursive problem has size at most approximately:

$$\frac{7n}{10}$$

The recurrence is:

$$T(n) \leq T\left(\frac{n}{5}\right) + T\left(\frac{7n}{10}\right) + O(n)$$

Comparison of Selection Methods

Method	Average/Expected	Worst Case
Sorting + Middle	$O(n \log n)$	$O(n \log n)$
Quickselect	$O(n)$	$O(n^2)$
Randomized Quickselect	$O(n)$ expected	$O(n^2)$
Median-of-Medians	$O(n)$	$O(n)$

Key Observation

Median-of-Medians provides a deterministic **worst-case linear-time** selection algorithm.

Selection Algorithms: Intuition

Method	Pivot Strategy
Quickselect	Fixed/chosen pivot
Randomized Select	Random pivot
Median-of-Medians	Carefully selected pivot

Quickselect

↓

Simple but worst case $O(n^2)$

Randomized Select

↓

Expected $O(n)$

Median-of-Medians

↓

Guaranteed $O(n)$

Key Takeaways

1. Finding the median is a special case of the **selection problem**.

Key Takeaways

1. Finding the median is a special case of the **selection problem**.
2. Partition-based selection is called **Quickselect**.

Key Takeaways

1. Finding the median is a special case of the **selection problem**.
2. Partition-based selection is called **Quickselect**.
3. Quickselect has average $O(n)$ but worst-case $O(n^2)$ complexity.

Key Takeaways

1. Finding the median is a special case of the **selection problem**.
2. Partition-based selection is called **Quickselect**.
3. Quickselect has average $O(n)$ but worst-case $O(n^2)$ complexity.
4. Randomized Quickselect chooses the pivot randomly and has expected $O(n)$ complexity.

Key Takeaways

1. Finding the median is a special case of the **selection problem**.
2. Partition-based selection is called **Quickselect**.
3. Quickselect has average $O(n)$ but worst-case $O(n^2)$ complexity.
4. Randomized Quickselect chooses the pivot randomly and has expected $O(n)$ complexity.
5. Median-of-Medians chooses a deterministic high-quality pivot.

Key Takeaways

1. Finding the median is a special case of the **selection problem**.
2. Partition-based selection is called **Quickselect**.
3. Quickselect has average $O(n)$ but worst-case $O(n^2)$ complexity.
4. Randomized Quickselect chooses the pivot randomly and has expected $O(n)$ complexity.
5. Median-of-Medians chooses a deterministic high-quality pivot.
6. Median-of-Medians guarantees:

$$\boxed{O(n)}$$

worst-case time.

Remember

Selection $\neq$ Sorting

We do not need to sort the entire array to find its median.

Thank You

Questions?