Course: Analysis of Algorithms
Code: CS33104
Branch: MCA -3rd Semester

Lecture – 4 : Decrease and Conquer Strategy

Faculty & Coordinator : Dr. J Sathish Kumar (JSK)

Department of Computer Science and Engineering

Motilal Nehru National Institute of Technology Allahabad,
Prayagraj-211004

Searching an element

- We are given a list of records.
- Each record has an associated key.
- Give efficient algorithm for searching for a record containing a particular key.
- Efficiency is quantified in terms of average time analysis (number of comparisons) to retrieve an item.
- Each record in list has an associated key.

- In this example, the keys are ID numbers.

- Given a particular key,
  how can we efficiently retrieve the
  record from the list?

Class Exercise

- 10, 20, 80, 30, 60, 50, 110, 100, 130, 170
  - Search Key = 110?
  - Ans: Returned Position: 7
- 10, 20, 80, 30, 60, 50, 110, 100, 130, 170
  - Search Key = 175?
  - Ans; Returned Position: -1 or Not available implies, element is not present.

Linear Search

- Step through array of records, one at a time.
- Look for record with matching key.
- Search stops when
  - record with matching key is found
  - or when search has examined all records without success

Pseudo Code

- // Search for a desired item in the n array elements
- // starting at a[first].
- // Returns pointer to desired record if found.
- // Otherwise, return NULL
- …
- for(i = first; i < n; ++i)
- if(a[first+i] is desired item)
- return &a[first+i];
- // if we drop through loop, then desired item was not found
- return NULL;

Time Complexity Analysis

- Number of operations depends on n, the number of entries in the list.
- For an array of n elements, the worst case time for serial search requires n array accesses: $O(n)$.
- Consider cases where we must loop over all n records:
  - desired record appears in the last position of the array
  - desired record does not appear in the array at all
- Can we do better than O(n)?

Example

Example: sorted array of integer keys. Target=7.

[0]	[1]	[2]	[3]	[4]	[5]	[6]
3	6	7	11	32	33	53

3	6	7	11	32	33	53

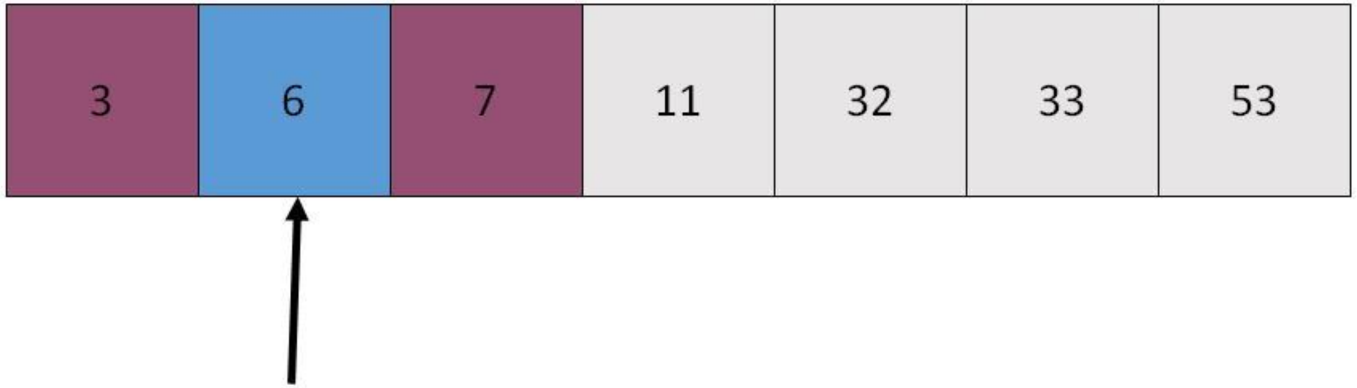
Find approximate midpoint

Is 7 = midpoint key? NO.

Is 7 < midpoint key? YES.

3	6	7	11	32	33	53

Search for the target in the area before midpoint.

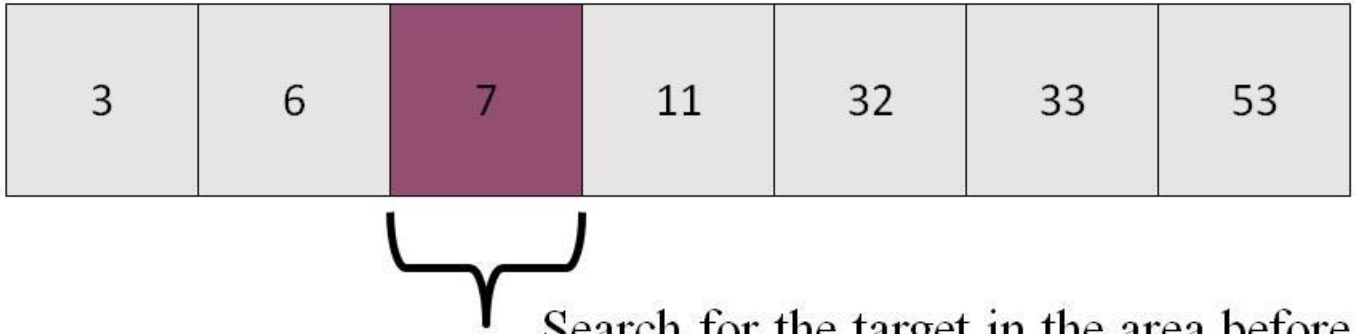


Find approximate midpoint

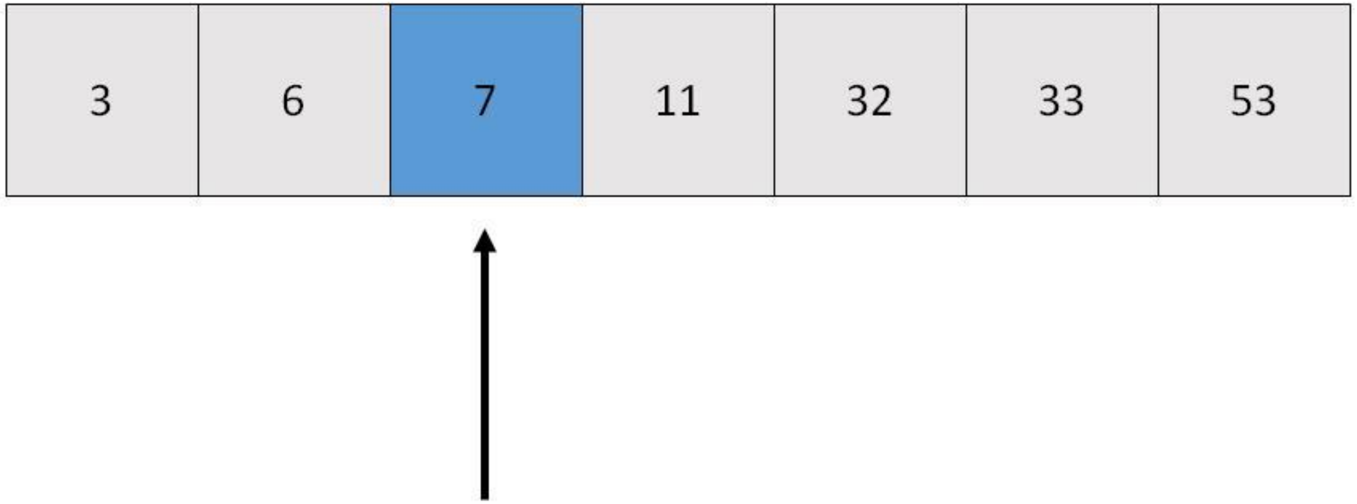
Is 7 = midpoint key? NO.

Target < key of midpoint? NO.

Target > key of midpoint? YES.



Search for the target in the area before midpoint.



Find approximate midpoint.
Is target = midpoint key? YES.

Class Exercise

- 10, 20, 80, 30, 60, 50, 110, 100, 130, 170
  - Search Key = 110?
  - 10,20,30,50,60,80,100,110,130,170
  - Ans: Returned Position: 8
- 10, 20, 80, 30, 60, 50, 110, 100, 130, 170
  - Search Key = 175?
  - Ans; Returned Position: -1 or Not available implies, element is not present.

Class Exercise

  - Search Key = 110?
  - Ans: Returned Position: 7
- 10, 20, 80, 30, 60, 50, 110, 100, 130, 170
  - Search Key = 175?
  - Ans; Returned Position: -1 or Not available implies, element is not present.

Class Exercise

- 11 23 31 33 65 68 71 89 100
  - Search Key = 31.
    - At first hi=8 low=0 so mid=4 and x[mid]= 65 is the center element but 65 > 31.
    - So now hi = 4-1=3. Now mid= (0 + 3)/2 = 1, so x[mid]= 23 < 31.
    - So again low= 1 + 1 = 2. Now mid = (3 + 2)/2 = 2 & x[mid]= 31 = key.
    - So the search is successful.
  - Search Key =75
    - Unsuccessful search.

Binary Search

- Perhaps we can do better than O(n) in the average case?
- Assume that we are give an array of records that is sorted. For instance:
  - an array of records with integer keys sorted from smallest to largest (e.g., ID numbers), or
  - an array of records with string keys sorted in alphabetical order (e.g., names).

Pseudo Code

```
…
if(size == 0)
   found = false;
else {
   middle = index of approximate midpoint of
   array segment;
   if(target == a[middle])
         target has been found!
   else if(target < a[middle])
         search for target in area before
   midpoint;
   else
         search for target in area after midpoint;
}
…
```

Time Complexity Analysis

- Each level in the recursion, we split the array in half (divide by two).
- **T(n) = T(n/2) + 1** => Masters Theorem O(logn).
- Therefore maximum recursion depth is floor($\log_2 n$) and worst case = O($\log_2 n$).
- Average case is also = O($\log_2 n$).
- Average and worst case of serial search = O(n)
- Average and worst case of binary search = O($\log_2$n)
- Can we do better than this?

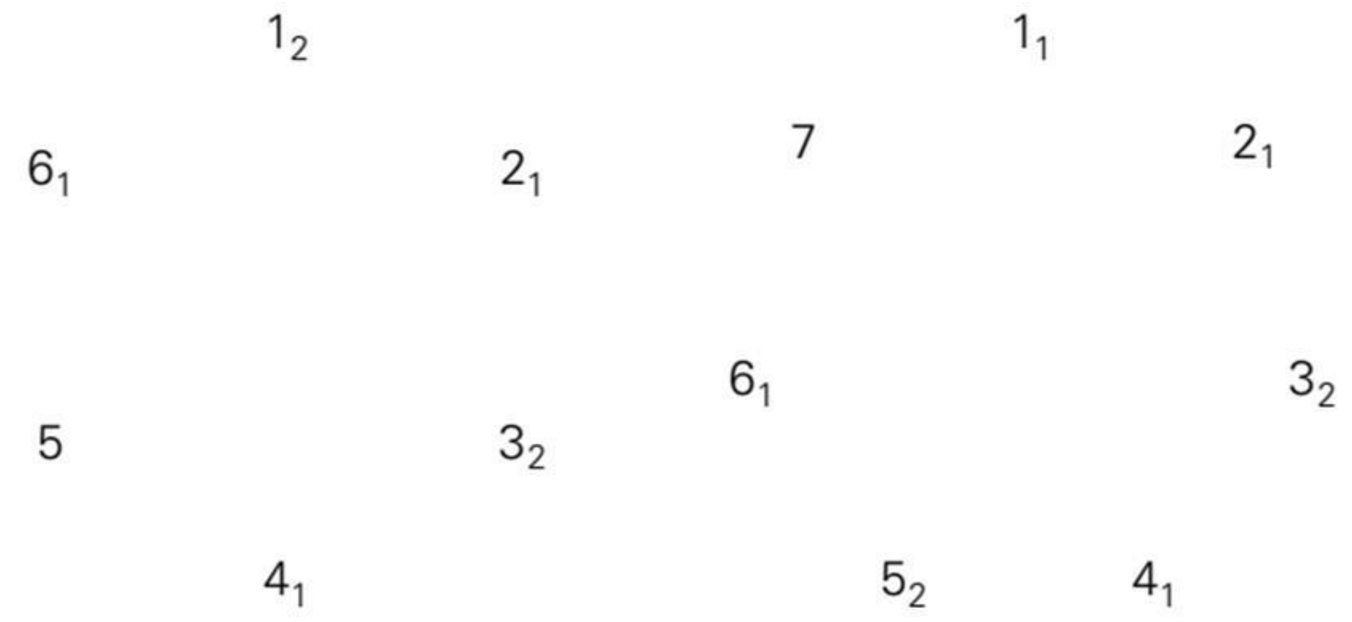
YES. Use a hash table!

Josephus Problem

- So let n people numbered 1 to n stand in a circle.
- Starting the grim count with person number 1, we eliminate every second person until only one survivor is left.
- The problem is to determine the survivor's number $J(n)$.
- For example, if n is 6, people in positions 2, 4, and 6 will be eliminated on the first pass through the circle, and people in initial positions 3 and 1 will be eliminated on the second pass, leaving a sole survivor in initial position 5—thus, $J(6) = 5$.
- To give another example, if n is 7, people in positions 2, 4, 6, and 1 will be eliminated on the first pass (it is more convenient to include 1 in the first pass) and people in positions 5 and, for convenience, 3 on the second—thus, $J(7) = 7$.

Josephus Problem

- **Josephus Problem**



(a) (b)

Josephus Problem

- **Josephus Problem (Decrease and Conquer Strategy)**
  - It is convenient to consider the cases of even and odd *n*'s separately. If *n* is even, i.e., $n = 2k \Rightarrow J(2k) = 2J(k) - 1$.
  - Let us now consider the case of an odd *n* ($n > 1$), i.e., $n = 2k + 1 \Rightarrow J(2k + 1) = 2J(k) + 1$
  - Interestingly, the most elegant form of the closed-form answer involves the binary representation of size *n*: $J(n)$ can be obtained by a 1-bit cyclic shift left of *n* itself!
  - For example, $J(6) = J(110_2) = 101_2 = 5$ and $J(7) = J(111_2) = 111_2 = 7$.