Divide and conquer

Dr. Priyadarshan Dhabe,
Ph.D (IIT Bombay)
Professor in Information Technology,
VIT, Pune

Divide and conquer- general strategy

In **divide-and-conquer,** we solve a problem recursively, applying **three steps** at each level of the recursion:

1. **Divide-** the problem into a number of sub-problems that are smaller instances of the same problem.
2. **Conquer-** the sub-problems by solving them recursively. If the sub-problem sizes are **small** enough, just solve the sub-problems in a straightforward manner.
3. **Combine** the **solutions** of the sub-problems into the solution for the original problem.

Divide and conquer- general strategy

- When the subproblems are **large enough** to solve recursively, we call that the ***recursive case***.
- Once the subproblems become small enough that we no longer recurse, we say that the recursion "*bottoms out*" and that we have gotten down to the ***base case***.
- **Recurrences relations** go hand in hand with the divide-and-conquer paradigm, because they give us a natural way to characterize the **running times** of divide-and-conquer algorithms.

Divide and conquer- binary search

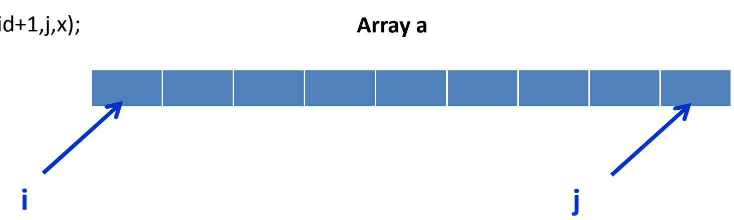
Binary search

```
BS(a, i, j, x)
{  mid=(i+j)/2;
   if (a[mid]==x)
      return mid;
Else
   if (a[mid]>x)
      BS(a,i,mid-1,x);
   else
      BS(a,mid+1,j,x);
}
```

$$T(n) = \begin{cases} 1 & \text{if } n = 1 \\ T(n/2) + C & \text{if } n > 1 \end{cases}$$

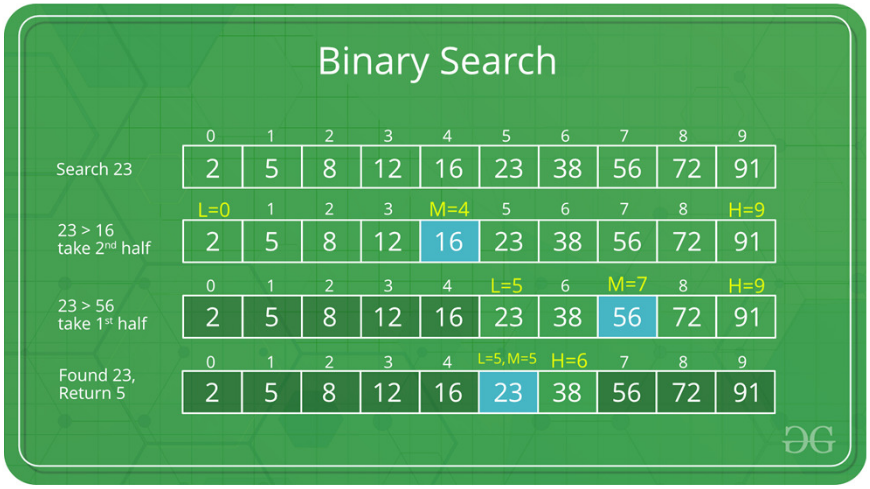
C – constant time needed for comparison and computing mid can be taken as 1

$$T(n) = \begin{cases} 1 & \text{if } n = 1 \\ T(n/2) + 1 & \text{if } n > 1 \end{cases}$$



Divide and conquer- binary search working

-In each step problem size is reduced to half



Recurrence relations- Substitution method

$$
\begin{aligned}
T(n) &= T(n/2)+1 \\
&= (T(n/4)+1)+1 \\
&= 2+T(n/4) \\
&= 2+(T(n/8)+1) \\
&= 3+T(n/8) \\
&\ldots\ldots\ldots \\
&= k+T(n/2^k) \qquad \text{max } \textit{vaue of k can be } \log n \\
&= \log n + T(n/2^{\log n}) \\
&= \log n + T(1) \\
&= \log n \\
&\Rightarrow O(\log n)
\end{aligned}
$$

Recurrence relations- Substitution method

To prove that our guess is correct $T(n) = O(\log n)$

we have to prove that $T(n) \le c * \log\ n$ using induction

$T(n) = T(n/2) + 1$ known, since $T(n/2) \le c * \log\ n/2$

$$\le c * \log\ n/2 + 1$$

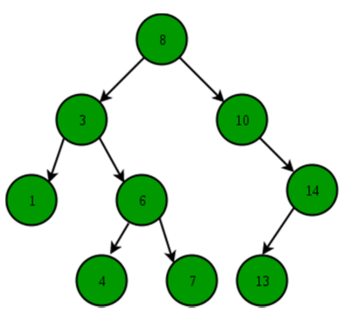
$$= c * (\log\ n - \log\ 2) + 1$$

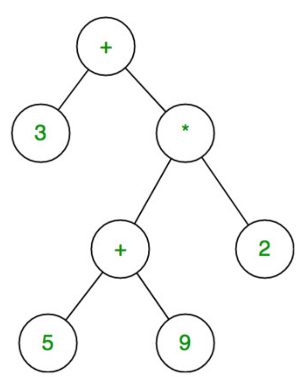
$$= c * \log n - c * \log 2 + 1$$

$$\boxed{T(n) \le c * \log\ n}$$

Applications of binary search

- Finding **duplicates** in the list of numbers
- **Binary search tree** can represent **arithmetic expression** including operands (leaf) and operators (**non-leaf nodes**). Traversal of the tree will lead to **infix**, **prefix** and **postfix form** of expressions
- Taking 2 way decisions





Quick sort- Overview

- Quicksort uses this basic process in order to sort:
  1. Pick a *pivot (central point)*
  2. Partition the array into 3 subarrays:
    A. items <= pivot,
    B. the pivot,
    C. items > pivot
  3. Recursively quicksort *A* and *C*
- **Quick sort uses *in-place* sort, i.e the original array itself is modified and thus do not require additional space.**
- **There is no need of combining also.**
- **Left sub array contains all those elements which are less than or equal to pivot and RHS sub array contains elements greater than the pivot.**

Quick sort- Algorithm

initial call is QUICKSORT(A, 1, $A.length$).

```
QUICKSORT(A, p, r)
if p < r
    q = PARTITION(A, p, r)
    QUICKSORT(A, p, q − 1)
    QUICKSORT(A, q + 1, r)
```

See the trace of Partition algorithm

```
PARTITION(A, p, r)
x = A[r]        ← Pivot last element
i = p − 1
for j = p to r − 1
    if A[j] ≤ x
        i = i + 1
        exchange A[i] with A[j]
exchange A[i + 1] with A[r]
return i + 1
```

Quick sort- Performance

- The running time of quicksort depends on whether the partitioning is **balanced** or **unbalanced**, which in turn depends on which elements are used for partitioning.
  - If the partitioning is **balanced (around n/2 elements in each partition)**, the algorithm runs asymptotically as fast as merge sort i.e **O(n log n).**
- If the partitioning is unbalanced, however, it can run asymptotically as **slowly as insertion sort O(n^2)**.
-

Quick sort- Performance

- **Worst-case partitioning** :-The worst-case behavior for quicksort occurs when the partitioning routine produces one subproblem with (n -1) elements and one with 0 elements.
- This will happen when the array is completely sorted.
- Let us assume that this unbalanced partitioning arises in each recursive call.

The partitioning costs $\Theta(\mathrm{n})$ time. Since the recursive call on an array of size 0 just returns, $\mathrm{T}(0) = \Theta(1)$, and the recurrence for the running time is

$$\mathrm{T}(\mathrm{n}) = \mathrm{T}(\mathrm{n}-1) + T(0) + \Theta(\mathrm{n})$$

$$\mathrm{T}(\mathrm{n}) = \mathrm{T}(\mathrm{n}-1) + \Theta(\mathrm{n})$$

Quick sort- Performance worst case

$$
\begin{aligned}
T(n) &= T(n-1) + \Theta(n) \\
&= T(n\text{-}1) + cn \qquad \text{for } c \geq 1 \\
&= cn + T(n\text{-}1) \\
&= cn + cn + T(n\text{-}2) \\
&= 2cn + T(n\text{-}2) \\
&= 3cn + T(n\text{-}3) \\
&\ldots\ldots\ldots\ldots \\
&= (n\text{-}1)cn + T(n\text{-}(n\text{-}1)) \\
&= (n\text{-}1)cn + T(1) \\
&= cn^2 - cn + 1 \Rightarrow O(n^2)
\end{aligned}
$$

Substitution method

Quick sort- Performance best case

- In the **most even possible split**, PARTITION produces two subproblems, each of size no more than n/2, since one is of size **floor(n/2)** and one of size **ceil(n/2)-1**.
- In this case, quicksort runs much faster. The recurrence for the running time is then

$$T(n) = 2T(n/2) + \Theta(n)$$

Quick sort- Performance best case

Substitution method

$$
\begin{aligned}
T(n) &= 2T(n/2) + \Theta(n) \\
&= 2T(n/2) + cn \quad \text{for } c \geq 1 \\
&= 2(2T(n/4) + cn) + cn \\
&= 4T(n/4) + 2cn \\
&= 4(2T(n/8) + cn) + 2cn \\
&= 8T(n/8) + 3cn \\
&\ldots\ldots\ldots \\
&= kT(n/2^k) + kcn \quad 2^k = n \text{ thus } k = \log_2 n \\
&= \log n\, T(1) + cn \log n \\
&= \log n + cn \log n = \log n(cn+1) = cn \log n \\
&\Rightarrow O(n \log n)
\end{aligned}
$$

Quick sort- Performance average case

- Quick sort, produces around 80% balanced partitions and 20% unbalanced partitions on a random input and thus average case time complexity is ***O(nlogn),*** eventhogh the constants involved are higher than the best case

Divide and conquer- Merge sort

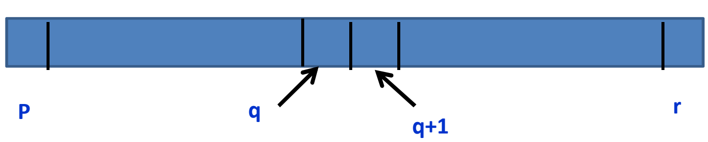
Proposed by John von Neumann, 1945

- The ***merge sort*** algorithm follows the divide-and-conquer paradigm. Intuitively, it operates as follows.
- **Divide:** Divide the n-element sequence to be sorted into two subsequences of n/2 elements each.
  Conquer: Sort the two subsequences recursively using merge sort.
  Combine: Merge the two sorted subsequences to produce the sorted answer.
- The recursion "**bottoms out**" when the sequence to be sorted has **length 1**, in which case there is no work to be done, since every sequence of length 1 is already in **sorted order**.

Divide and conquer- Merge sort

initial call MERGE-SORT(A, 1, A:*length*), where A:*length* = n.

```
MERGE-SORT(A, p, r)
if p < r
    q = ⌊(p + r)/2⌋
    MERGE-SORT(A, p, q)
    MERGE-SORT(A, q + 1, r)
    MERGE(A, p, q, r)
```



Divide and conquer- Merge sort

- The key operation of the merge sort algorithm is the **merging** of two sorted sequences in the "combine" step. We merge by calling the procedure **MERGE(A, p, q, r),** where A is an array and p, q, and r are indices into the array such that **p <=q < r.**
- The **procedure merge** assumes that the subarrays **A[p,… q]** and **A[q+1,.., r]** are in **sorted order**. It ***merges*** them to form a single sorted subarray that replaces the current subarray **A[p,…,r].**

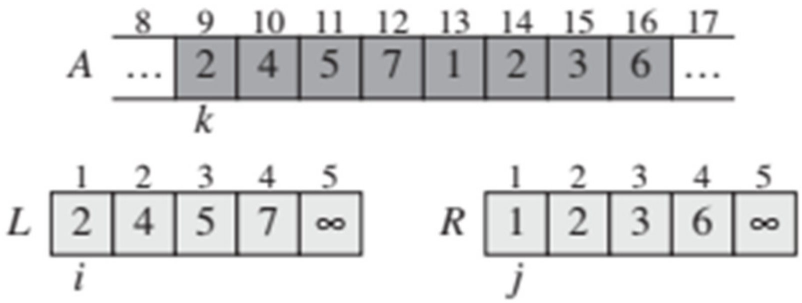
The procedure merge takes time of $\Theta(n)$, where $n = r\text{-}p+1$, the total number of elements to be merged.

Divide and conquer- Merge sort

```
MERGE(A, p, q, r)
n1 = q - p + 1                          No of elements in left half of A
n2 = r - q                              No of elements in right half of A
let L[1..n1 + 1] and R[1..n2 + 1] be new arrays
for i = 1 to n1
    L[i] = A[p + i - 1]                 Copy left half of A
for j = 1 to n2
    R[j] = A[q + j]                     Copy right half of A
L[n1 + 1] = ∞
R[n2 + 1] = ∞                           Add sentinels-guards
i = 1
j = 1
for k = p to r
    if L[i] ≤ R[j]
        A[k] = L[i]                     Merging of L
        i = i + 1                       and R in A
    else A[k] = R[j]
        j = j + 1
```

Working of merge Procedure

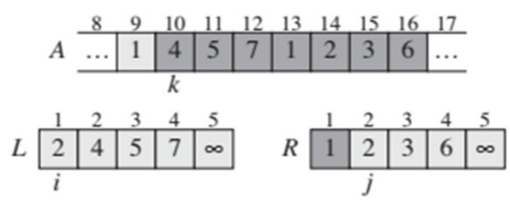
call MERGE(A, 9, 12, 16),




```
for k = p to r
    if L[i] ≤ R[j]
        A[k] = L[i]
        i = i + 1
    else A[k] = R[j]
        j = j + 1
```

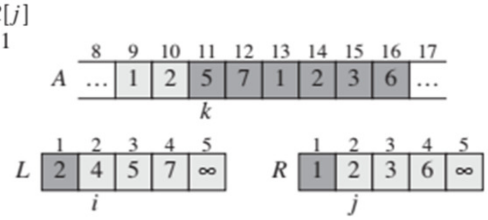
(a)

Working of merge procedure



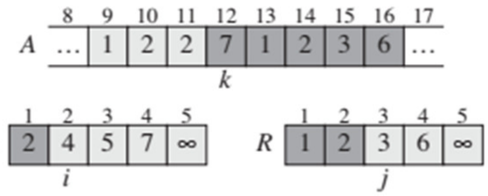
(b)

```
for k = p to r
    if L[i] ≤ R[j]
        A[k] = L[i]
        i = i + 1
    else A[k] = R[j]
        j = j + 1
```



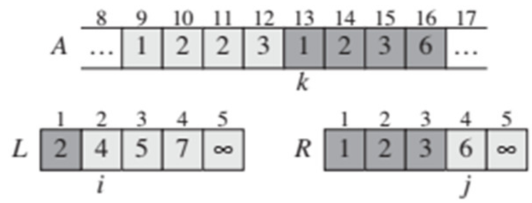
(c)

Working of merge procedure



(d)

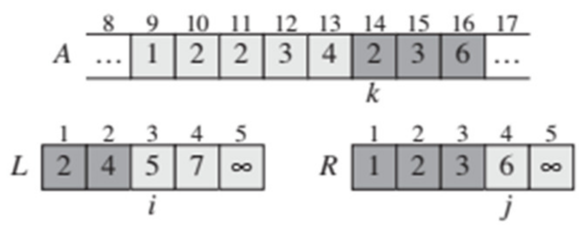
```
for k = p to r
    if L[i] ≤ R[j]
        A[k] = L[i]
        i = i + 1
    else A[k] = R[j]
        j = j + 1
```



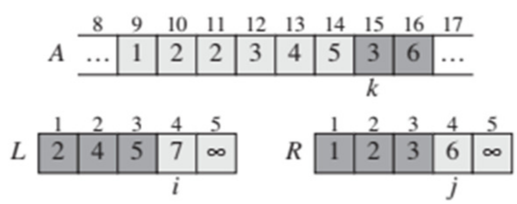
(e)

Working of merge procedure

```
for k = p to r
    if L[i] ≤ R[j]
        A[k] = L[i]
        i = i + 1
    else A[k] = R[j]
        j = j + 1
```



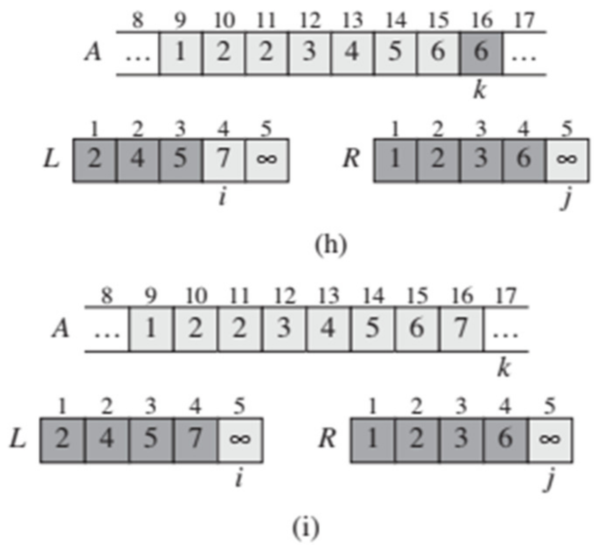
(f)



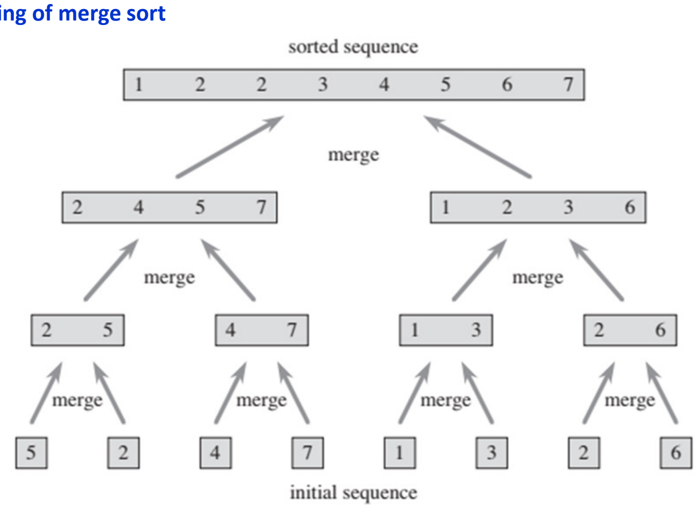
(g)

Working of merge procedure

```
for k = p to r
    if L[i] ≤ R[j]
        A[k] = L[i]
        i = i + 1
    else A[k] = R[j]
        j = j + 1
```



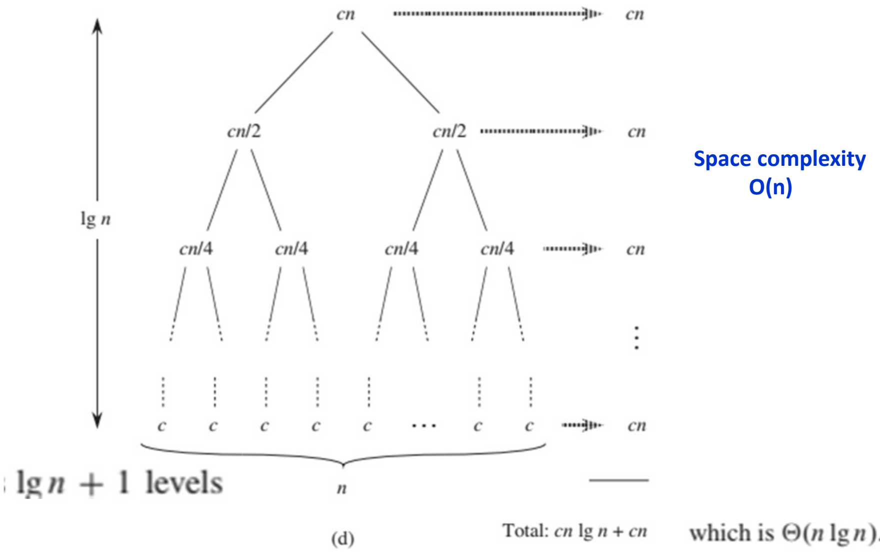
Working of merge sort



The operation of merge sort on the array A =[5; 2; 4; 7; 1; 3; 2; 6]. The lengths of the sorted sequences being merged increase as the algorithm progresses from bottom to top.

Analysis of merge sort

$$T(n) = 2T(n/2) + cn.$$



Space complexity
O(n)

Analysis of merge sort

Time complexity of Merge sort is $\Theta(n \log n)$ for all the best, worst and average cases since it divides the array into half always.

Finding majority element from an array

- An element x is a majority element in an array A of size n, if it appears more than n/2 times in A

A={2,2,3,4,2,2,2,6}

Majority element in A is 2, since it appears in A 5 times which is more than 8/2=4

B={1,2,3,4,5,7,2,6}

Array B don't have majority element

Algorithm Finding majority element-Divide and Conquer

Algorithm **Majority**(**A**[1..*n*]):

If |**A**| = 0 then output *null*, else if |**A**| = 1 then output **A**[1] ; else:

- If *n* = |**A**| is odd then
  - check whether **A**[*n*] is a majority in **A** by counting the number of occurrences of value **A**[*n*]; //O(n) work

    if yes then output **A**[*n*], otherwise decrease *n* by one
- Initialize additional array **B** of size |**A**|/2
- Set *j* to 0
- For *i* = 1,2,…,*n*/2 do:
  - if **A**[2*i*-1] = **A**[2*i*] then
    - increase *j* by one
    - set **B**[*j*] to **A**[2*i*]
- Find if there is a majority in **B**[1..*j*] by executing **Majority**(**B**[1..*j*])
- If a majority value **x** in **B**[1..*j*] is returned then check whether **x** is a majority in **A**, by going through array **A** and counting the number of occurrences of value **x** in **A**; if successful output **x**; otherwise *null*

Algorithm Finding majority element-Divide and Conquer

Majority element algorithm:

- Phase 1: Use divide-and-conquer to find candidate value M
- Phase 2: Check if M really is a majority element, $\theta(n)$ time, simple loop

Phase 1 details:

- Divide:
  - Group the elements of A into n/2 pairs
  - If n is odd, there is one unpaired element, x
    - Check if this x is majority element of A
    - If so, then return x, but otherwise discard x
  - Compare each pair (y, z)
    - If (y==z) keep y and discard z
    - If (y!=z) discard both y and z
  - So we keep ≤ n/2 elements
- Conquer: One recursive call on subarray of size ≤ n/2
- Combine: Nothing remains to be done, so omit this step

Algorithm Finding majority element-Divide and Conquer

Example:

A = [7, 7, 5, 2, 5, 5, 4, 5, 5, 5, 7]
(7, 7) (5, 2) (5, 5) (4, 5) (5, 5) (7)
A = [7, 5, 5]
(7, 5) (5) ⇒ return 5 (candidate, also majority)

Example:

A = [1, 2, 3, 1, 2, 3, 1, 2, 9, 9]
(1, 2) (3, 1) (2, 3) (1, 2) (9, 9)
A = [9] ⇒ return 9 (candidate, but not majority)

Algorithm Finding majority element-Divide and Conquer

Let T(n) = running time of Phase 1 on array of size n

$T(n) = T(n/2) + \theta(n)$

- Number of recursive subproblems = 1
- Size of each subproblem = n/2 [worst-case]
- Time for all the non-recursive steps = $\theta(n)$

Running time is T(n)= O(n)

Space complexity is n- for A + (n/2) for B =O(n)

Order statistics

- The ith ***order statistic*** of a set of n elements is the ith smallest element.
- For example, the ***minimum*** of a set of elements is the first order statistic (i= 1), and the ***maximum*** is the nth order statistic (i = n).

A median, informally, is the "*halfway point*" of the set.
When n is odd, the median is unique, occurring at $i = (n+1)/2$.
When n is even, there are two medians, occurring at $i = n/2$ and $i = n/2+1$. Thus, regardless of the parity of n, medians occur at $i = \lfloor (n+1)/2 \rfloor$ (the lower median) and $i = \lceil (n+1)/2 \rceil$ (the upper median). We use the term median to refer to the lower median.

Order statistics

n=7=odd

Median=(n+1)/2=8/2=4

A=[1,2,3,**4**,5,6,7]

n=6=even

Lower Median=floor((n+1)/2)=floor(7/2)=3

B=[1,2,3,4,5,6]

upper Median=ceil((n+1)/2)=ceil(7/2)=4

Order statistics-Selection Problem

- Problem of selecting the ith order statistic from a set of n distinct numbers is crucial for many applications.
- We assume for convenience that the set contains distinct numbers, but the same approach can be extended to a set contains repeated values.
- The problem is called as "**Selection Problem**"

A=[1,2,3,4,5,6,7]

Finding ith ordered element

Order statistics-Selection Problem

Selection Problem can be formally specified as follows

Input : - A set A of n (distinct) numbers and
an integer i with $1 \leq i \leq n$

Output : - The element $x \in A$ that is larger than
exactly $(i\text{-}1)$ other elements of A

We can solve the selection problem in O(n logn) time, since we can sort the numbers using **merge sort** and then simply index the *ith element* in the output array.

Order statistics- maximum and minimum

- How many comparisons are necessary to determine the **minimum** of a set of **n** elements?
- We can easily obtain an upper bound of **(n- 1)** comparisons: examine each element of the set in turn and keep track of the smallest element seen so far.
- In the following **procedure**, we assume that the set resides in array **A**, where **A.*length*=n**.

(n-1) comparison is optimal

```
MINIMUM(A)
min = A[1]
for i = 2 to A.length
    if min > A[i]
        min = A[i]
return min
```

Simultaneous minimum and maximum

- In some applications, we must find both the minimum and the maximum of a set of n elements.
- how to determine both the minimum and the maximum of n elements using O(n) comparisons, which is asymptotically optimal: simply find the minimum and maximum independently, using **(n – 1)** comparisons for each, for a total of (n-1)+(n-1)= **(2n – 2)** comparisons.

Simultaneous minimum and maximum

- In fact, we can find both the minimum and the maximum using at most 3 floor(n/2) comparisons by maintaining both the minimum and maximum elements seen thus far.
- Rather than processing each element of the input by comparing it against the current minimum and maximum, at a cost of 2 comparisons per element,
- We can process elements in pairs. We compare pairs of elements from the input first *with each other*, and then we compare the smaller with the current minimum and the larger to the current maximum, at a cost of 3 comparisons for every 2 elements.

Simultaneous minimum and maximum

Current min

Current max

A=[……………………(2,   3)……………………….]

compare

How to set current min and max depends upon value of n.?

Simultaneous minimum and maximum

- **If n is odd**, we set both the min and max to the value of the first element, and then we process the rest of the elements in pairs.
- **If n is even**, we perform 1 comparison on the first 2 elements to determine the initial values of the min and max, and then process the rest of the elements in pairs as in the case for odd n.

Simultaneous minimum and maximum

Let us find total number of comparisons.

If n is odd, then we need $3\lfloor n/2 \rfloor$ comparisons.

If n is even, we perform 1 initial comparison followed by

$3(n-2)/2$ comparisons, for a total of $(3(n-2)/2)+1 = 3n/2-2$.

Thus, in either case, the total number of comparisons is at most $3\lfloor n/2 \rfloor$

$\Rightarrow O(n)$.

divide-and-conquer algorithm for the -selection problem

- Selecting **i th smallest element** from an unsorted array.
- The algorithm **RANDOMIZED-SELECT is used to find i th smallest element in A[p,..,r].**
- The algorithm **RANDOMIZED-SELECT** is modeled after the **quicksort algorithm**.
- As in quicksort, we partition the input array recursively. But unlike quicksort, which recursively processes both sides of the partition, RANDOMIZED-SELECT works **on only one side** of the partition.
- In quick sort pivot is always the last element but in RANDOMIZED-SELECT, **Pivot is randomly chosen.**
- This difference shows up in the analysis: whereas quicksort has an expected running time of , **O(nlogn),** the expected running time of RANDOMIZED-SELECT is, **O(n)** assuming that the elements are distinct .

RANDOMIZED-PARTITION Algorithm

- Quick sort always uses the last element of the array A[r] as a pivot in partitioning, but RANDOMIZED-PARTITION uses a random number for choosing the pivot.
- Random choosing of pivot is expected to provide balanced partitioning on an **average**.

```
RANDOMIZED-PARTITION(A, p, r)
1  i = RANDOM(p, r)
2  exchange A[r] with A[i]     A[i] is pivot now
3  return PARTITION(A, p, r)
```

- **i** is the random number generated between **p** an **r**, using function RANDOM. The elements **A[i]** and **A[r]** are exchanged and thus, the call to the **PARTITION** algorithm uses **A[i]** as a **pivot** to partition the array A.

divide-and-conquer algorithm for the selection problem

-RANDOMIZED-SELECT uses the procedure RANDOMIZED-PARTITION. It is a randomized algorithm, since its behavior is determined in part by the output of a random-number generator.

-The following code for RANDOMIZED-SELECT returns the ith smallest element of the array **A[p,…,r]**.

-The RANDOMIZED-SELECT procedure works as follows. Line 1 checks for the **base case** of the recursion, in which the subarray **A[p,…,r]** consists of **just one element**.

- In this case, **i must equal 1**, and we simply return **A[p]** in line 2 as the **ith smallest element**.

divide-and-conquer algorithm for the selection problem

```
RANDOMIZED-SELECT(A, p, r, i)
if p == r
    return A[p]
q = RANDOMIZED-PARTITION(A, p, r)
k = q - p + 1
if i == k          // the pivot value is the answer
    return A[q]
elseif i < k
    return RANDOMIZED-SELECT(A, p, q - 1, i)
else return RANDOMIZED-SELECT(A, q + 1, r, i - k)
```

Return ith smallest element in A[p,…,r]

A[q] is pivot

No of elements <=pivot

Use left half

p		q=3					r

Pivot

RANDOMIZED-SELECT

assume

i=2

p		q=3					r

K=q-p+1=3-1+1=3
if i<k
yes i=2

RANDOMIZED-SELECT$(A, p, q-1, i)$

i=7

p		q=3	4	5	6	7	r

K=q-p+1=3-1+1=3
if i>k (7>3)
Now i=i-k=(7-3)=4
i=4

RANDOMIZED-SELECT$(A, q+1, r, i-k)$

RANDOMIZED-SELECT

RANDOMIZED-SELECT(A, p, r, i)

1 **if** $p == r$
2   **return** $A[p]$
3 $q =$ RANDOMIZED-PARTITION(A, p, r)
4 $k = q - p + 1$
5 **if** $i == k$ // the pivot value is the answer
6   **return** $A[q]$
7 **elseif** $i < k$
8   **return** RANDOMIZED-SELECT$(A, p, q - 1, i)$
9 **else return** RANDOMIZED-SELECT$(A, q + 1, r, i - k)$

The call to **RANDOMIZED-PARTITION** in **line 3** partitions the array **A[p,..,r]** into two (possibly empty) subarrays **A[p,…, (q-1)]** and **A[(q+1),…,r]** such that each element of **A[p,…, (q-1)]** is less than or equal **A[q],** which in turn is less than each element of **A[(q+1),…,r].**

divide-and-conquer algorithm for the selection problem

RANDOMIZED-SELECT(A, p, r, i)

1 **if** $p == r$
2   **return** $A[p]$
3 $q =$ RANDOMIZED-PARTITION(A, p, r)
4 $k = q - p + 1$
5 **if** $i == k$ // the pivot value is the answer
6   **return** $A[q]$
7 **elseif** $i < k$
8   **return** RANDOMIZED-SELECT$(A, p, q - 1, i)$
9 **else return** RANDOMIZED-SELECT$(A, q + 1, r, i - k)$

Line 4 computes the number k of elements in the subarray **A[p,…,q]** that is, the number of elements in the low side of the partition, **plus one** for the pivot element.

divide-and-conquer algo.-- selection problem

```
RANDOMIZED-SELECT(A, p, r, i)
1  if p == r
2      return A[p]
3  q = RANDOMIZED-PARTITION(A, p, r)
4  k = q − p + 1
5  if i == k          // the pivot value is the a
6      return A[q]
7  elseif i < k
8      return RANDOMIZED-SELECT(A, p, q − 1, i)
9  else return RANDOMIZED-SELECT(A, q + 1, r, i − k)
```

Line 5 then checks whether **A[q] is the ith smallest element**. If it is, then line 6 **returns A[q].**
Otherwise, the algorithm determines in which of the two subarrays **A[p,…,(q-1)]** and **A[(q+1),…,r]** the ith smallest element lies.

```
RANDOMIZED-SELECT(A, p, r, i)
1  if p == r
2      return A[p]
3  q = RANDOMIZED-PARTITION(A, p, r)
4  k = q − p + 1
5  if i == k          // the pivot value is the answer
6      return A[q]
7  elseif i < k
8      return RANDOMIZED-SELECT(A, p, q − 1, i)
9  else return RANDOMIZED-SELECT(A, q + 1, r, i − k)
```

- **If i < k**, then the desired element lies on the **low side** of the partition, and line 8 recursively selects it from the subarray. (k number elements in low subarray)
- **If i > k**, however, then the desired element lies on the high side of the partition. Thus, the desired element is the **(i-k)th** smallest element of **A[(q+1),…, r],** which line 9 finds recursively.

Performance of algorithm for the selection problem- RANDOMIZED-SELECT

1. In worst case all the unbalanced partitions will be genrated and thus it is $O(n^2)$. (same as worst case of quick sort one partition is (n - 1) and other is of size zero)

2. For the best case (assume exactly) balanced partitions are generated but only one partition is processed in Radomized - selection

Thus, the total workdone is

$n + n/2 + n/4 + \ldots\ldots + 1 = (2n - 1)$, thus is $O(n)$ (linear) time

Recurrence relation is

$$T(n) = \begin{cases} 1 & \text{if } n = 1 \\ T(n/2) + cn \end{cases}$$

Time Complexity of RANDOMIZED-SELECT- balanced partitioning

	tree	Time needed for partioning
	n	n
	|	|
	n/2	n/2
	|	|
Log n	n/4	n/4
	|	|
	n/8	n/8
	|	|
	.	.
	.	.
	1	1

$$\text{Log n} \left\{ \begin{array}{c} n \\ | \\ n/2 \\ | \\ n/4 \\ | \\ n/8 \\ | \\ . \\ . \\ 1 \end{array} \right.$$

Total workdone is sum of workdone at all the levels

Time Complexity of RANDOMIZED-SELECT-balanced partitioning

$n + n/2 + n/4 + \ldots\ldots. + 1 = (2n - 1)$,
thus is $O(n)$ (linear) time