Introduction to Algorithms (Chapter 8: Sorting in Linear Time)

Kyuseok Shim
Electrical and Computer Engineering
Seoul National University

Outline

- All the sorting algorithms introduced thus far are comparison sorts since *the sorted order they determine is based only on comparisons between the input elements*.
- We prove that any comparison sort must make $\Omega(n \log n)$ comparisons in the worst case to sort n elements.
- Thus, merge sort and heapsort are asymptotically optimal, and no comparison sort exists that is faster by more than a constant factor.
- We examine other sorting algorithms, such as counting sort and radix sort that run in linear time.
- Those algorithms use operations other than comparisons to determine the sorted order.
- Consequently, the $\Omega(n \log n)$ lower bound does not apply to them.

- All the sorting algorithms introduced thus far are comparison sorts since *the sorted order they determine is based only on comparisons between the input elements*.
- We prove that any comparison sort must make $\Omega(n \log n)$ comparisons in the worst case to sort n elements.
- Thus, merge sort and heapsort are asymptotically optimal, and no comparison sort exists that is faster by more than a constant factor.
- We examine other sorting algorithms, such as counting sort and radix sort that run in linear time.
- Those algorithms use operations other than comparisons to determine the sorted order.
- Consequently, the $\Omega(n \log n)$ lower bound does not apply to them.

Lower Bounds for Sorting

- Comparison sorting
  - Use only comparisons between elements to gain order information about an input sequence $<a_1, a_2,\ldots,a_n>$.
  - Given a two elements a_i and a_j, we perform only one of the tests $a_i < a_j$, $a_i \le a_j$, $a_i \ge a_j$, or $a_i > a_j$ to determine their relative order.
- Our assumption
  - All input elements are distinct (i.e., we do not check $a_i = a_{j)}$.
  - The comparisons $a_i < a_j$, $a_i \le a_j$, $a_i \ge a_j$, or $a_i > a_j$ are all equivalent in that they yield identical information about the relative order of a_i and a_j.
  - Thus, all comparisons have the form $a_i \le a_j$.

Decision Tree Model

- A Decision tree is a full binary tree representing the comparisons between elements performed by a particular sorting algorithm operating on an input of a given size.
- In a decision tree, we annotate each internal node by i:j for some i and j in the range $1 \leq i, j \leq n$, where n is the number of elements in the input sequence - each internal node indicates a comparison $a_i \leq a_j$.
- We also annotate each leaf by a permutation $\pi(1), \pi(2), \ldots, \pi(n)$.
- The execution of the sorting algorithm corresponds to tracing a simple path from the root of the decision tree down to a leaf.
- When we come to a leaf, the sorting algorithm has established the ordering $a_{\pi(1)} \leq a_{\pi(2)} \leq \ldots \leq a_{\pi}(n)$.

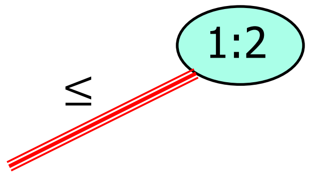
Decision Tree Model

- We consider only decision trees in which each permutation appears as a reachable leaf.
  - Because any correct sorting algorithm must be able to produce each permutation of its input, each of the n! permutations on n elements must appear as one of the leaves of the decision tree for a comparison sort to be correct.
  - Furthermore, each of these leaves must be reachable from the root by a downward path corresponding to an actual execution of the comparison sort.

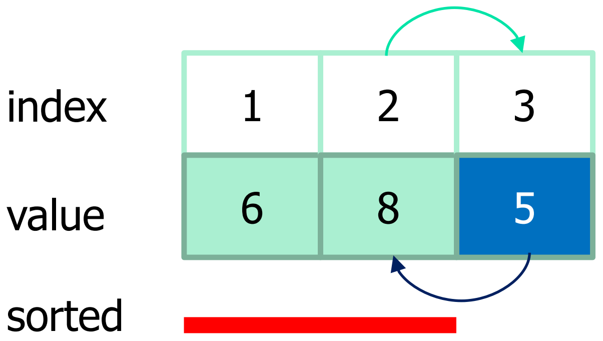
Insertion Sort

index	1	2	3
value	6	8	5
sorted	🟥		

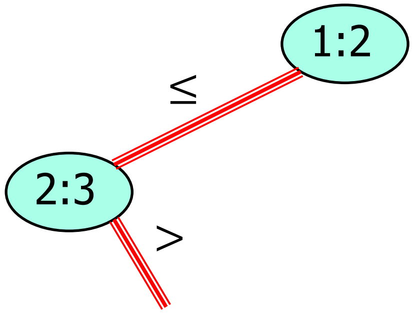
$a_1 = 6, a_2 = 8, a_3 = 5$



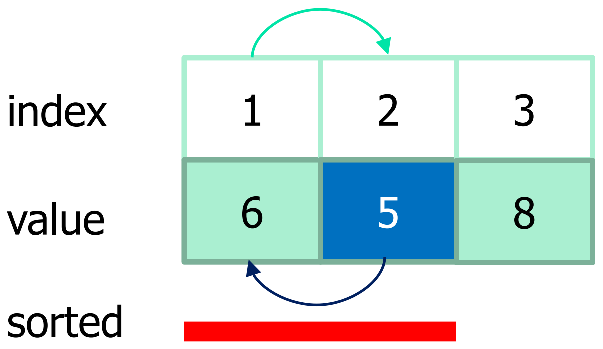
Insertion Sort



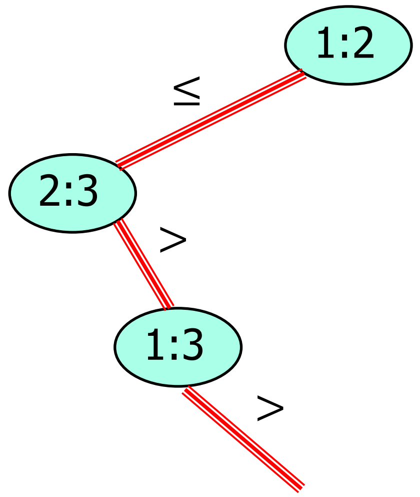
$a_1= 6, a_2=8, a_3=5$



Insertion Sort



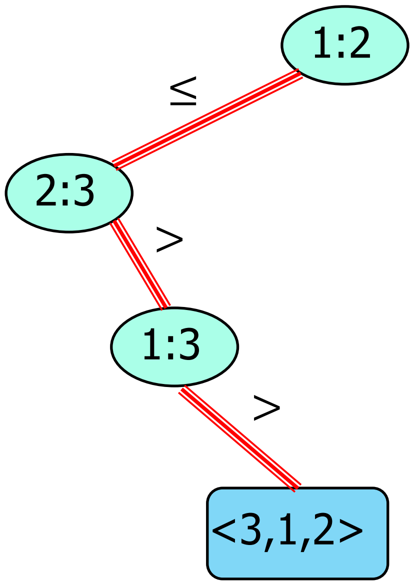
$a_1 = 6$, $a_2=8$, $a_3=5$



Insertion Sort

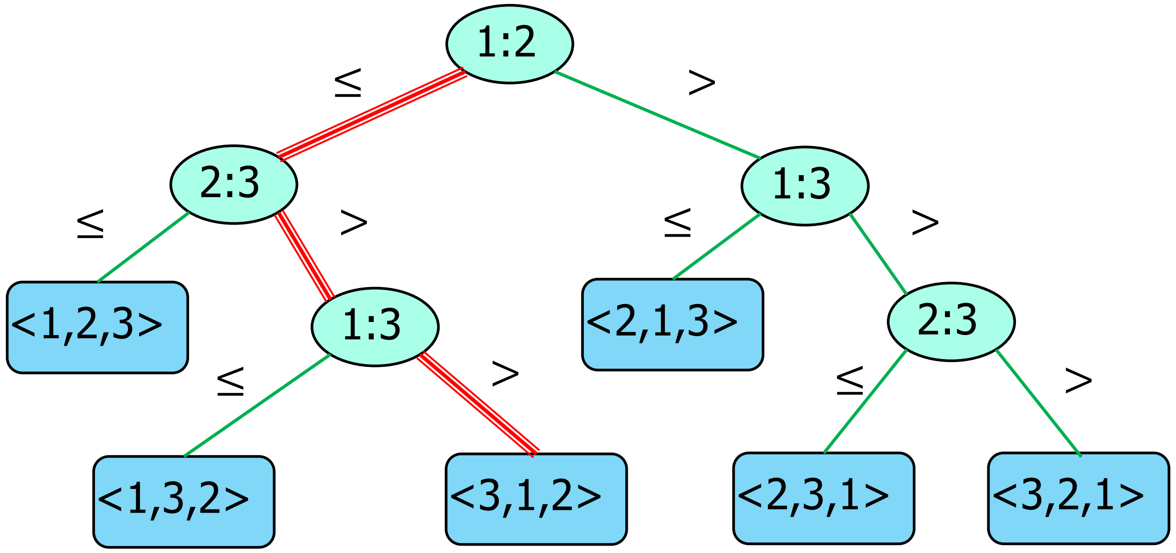
index	1	2	3
value	5	6	8
sorted			

$a_1 = 6$, $a_2 = 8$, $a_3 = 5$



The Decision tree for Insertion Sort

- The decision tree corresponding to the insertion sort algorithm operating on an input sequence of three elements.

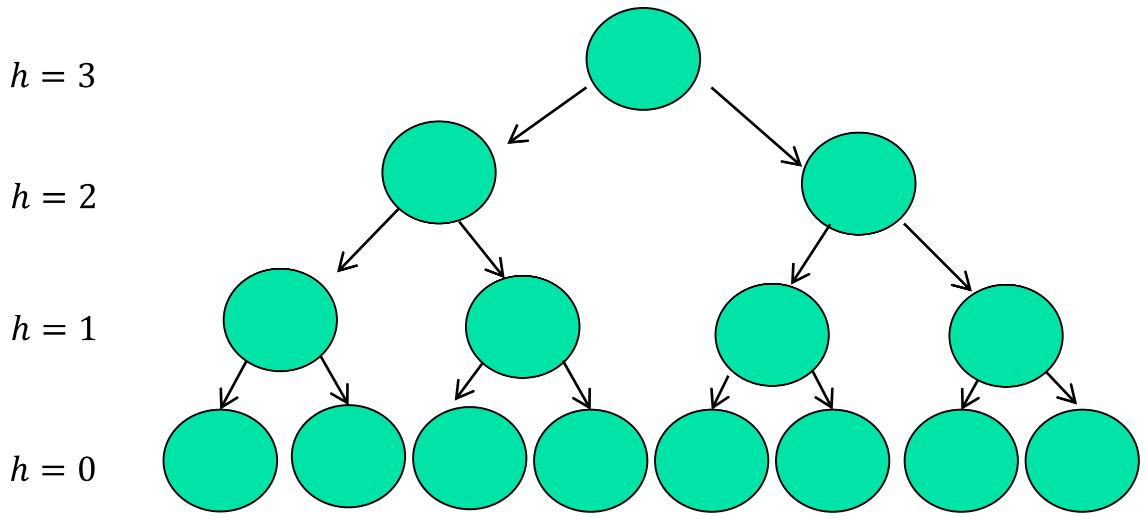


A Lower Bound for the Worst Case

- The length of the longest simple path from the root of a decision tree to any of its reachable leaves represents the worst-case number of comparisons that the corresponding sorting algorithm performs.
- Consequently, the worst-case number of comparisons for a given comparison sort algorithm equals the height of its decision tree.
- A lower bound on the heights of all decision trees in which each permutation appears as a reachable leaf is therefore a lower bound on the running time of any comparison sort algorithm.

A Binary Tree of Height h

- A binary tree of height h has no more than 2^h leaf nodes



A Lower Bound for the Worst Case

- Theorem 8.1
  - Any comparison sort algorithm requires $\Omega(n \log n)$ comparisons in the worst-case.
- Proof
  - It suffices to determine the height of a decision tree in which each permutation appears as a reachable leaf.
  - Consider a decision tree of height h with l reachable leaves corresponding to a comparison sort on n elements.
  - Because each of the $n!$ Permutations of the input appears as some leaf, $n! \leq l$.
  - Since a binary tree of height h has no more than 2^h, we have

$$n! \leq l \leq 2^h.$$

  - Thus,

$$\begin{aligned} h &\geq \log(n!) \\ &= \log(n(n-1)(n-2)\ldots 1) \end{aligned}$$

A Lower Bound for the Worst Case

- Theorem 8.1
  - Any comparison wort algorithm requires $\Omega(n \log n)$ comparisons in the worst-case.
- Proof
  - It suffices to determine the height of a decision tree in which each permutation appears as a reachable leaf.
  - Consider a decision tree of height h with l reachable leaves corresponding to a comparison sort on n elements.
  - Because each of the $n!$ Permutations of the input appears as some leaf, $n! \leq l$.
  - Since a binary tree of height h has no more than 2^h leaf nodes, we have

$$n! \leq l \leq 2^h.$$

  - Thus,

$$\begin{aligned} h &\geq \log(n!) \\ &= \log(n(n-1)(n-2)\dots 1) \\ &= \log n + \log(n-1) + \cdots + \log 1 \end{aligned}$$

A Lower Bound for the Worst Case

- Theorem 8.1
  - Any comparison wort algorithm requires $\Omega(n \log n)$ comparisons in the worst-case.
- Proof
  - It suffices to determine the height of a decision tree in which each permutation appears as a reachable leaf.
  - Consider a decision tree of height h with l reachable leaves corresponding to a comparison sort on n elements.
  - Because each of the $n!$ Permutations of the input appears as some leaf, $n! \leq l$.
  - Since a binary tree of height h has no more than 2^h leaf nodes, we have

$$n! \leq l \leq 2^h.$$

  - Thus,

$$\begin{aligned} h &\geq \log(n!) \\ &= \log(n(n-1)(n-2)\ldots 1) \\ &= \log n + \log(n-1) + \cdots + \log 1 \\ &\geq \log n + \log(n-1) + \cdots + \log\left(\frac{n}{2}\right) \end{aligned}$$

A Lower Bound for the Worst Case

- Theorem 8.1
  - Any comparison wort algorithm requires $\Omega(n \log n)$ comparisons in the worst-case.
- Proof
  - It suffices to determine the height of a decision tree in which each permutation appears as a reachable leaf.
  - Consider a decision tree of height h with l reachable leaves corresponding to a comparison sort on n elements.
  - Because each of the $n!$ Permutations of the input appears as some leaf, $n! \leq l$.
  - Since a binary tree of height h has no more than 2^h leaf nodes, we have

$$n! \leq l \leq 2^h.$$

  - Thus,

$$\begin{aligned} h &\geq \log(n!) \\ &= \log(n(n-1)(n-2)\ldots 1) \\ &= \log n + \log(n-1) + \cdots + \log 1 \\ &\geq \log n + \log(n-1) + \cdots + \log\left(\frac{n}{2}\right) \\ &\geq \frac{n}{2}\log\left(\frac{n}{2}\right) = \Omega(n \log n) \end{aligned}$$

A Lower Bound for the Worst Case

- Corollary 8.2
  - Heapsort and merge sort are asymptotically optimal comparison sorts.
- Proof
  - The O(n lg n) upper bounds on the running times for heapsort and merge sort match the $\Omega(n \log n)$ worst-case lower bound from Theorem 8.1.

Counting Sort

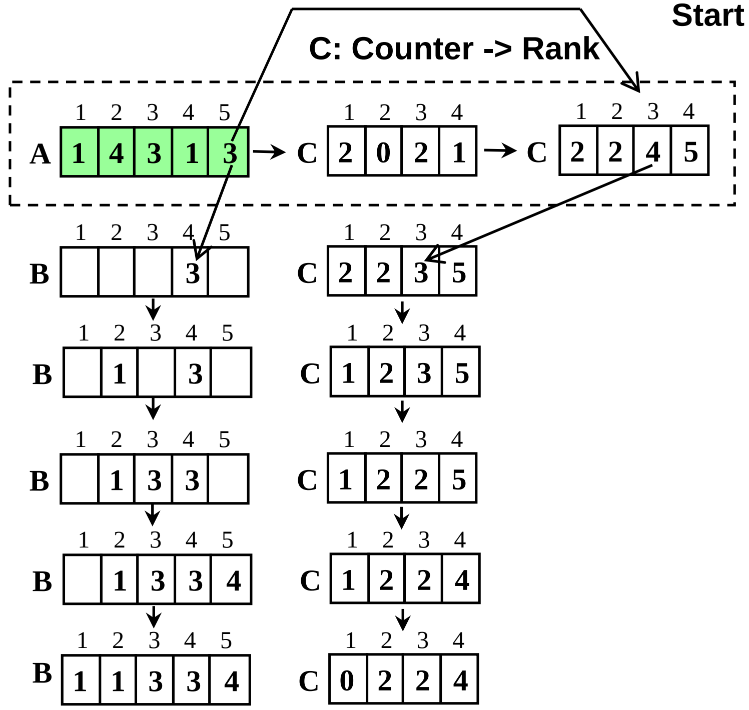
- Assumes that each of the n input elements is an integer in the range 1 to k, for some integer k.
- When k = O(n), the sort runs in Θ(n) time.
- Use three arrays
  - $A[1..n]$: the initial input
  - $B[1..n]$: the stored output
  - $C[1..k]$: $C[i]$ first store the number of input elements equal to i in $A[]$, and later to store the number of input elements less than or equal to i in $A[]$.
  - The number of occurences of $A[j]$ will be $C[A[j]]$.
  - We copy $A[j]$ to $B[C[A]]]$ - some care is needed for duplicate items

Counting Sort

```
COUNTING-SORT(A, B, k)
1.   let C[1..k] be a new array
2.   **for** i=1 **to** k
3.       C[i]=0
4.   **for** j=1 **to** A.length
5.       C[A[j]] = C[A[j]] + 1
6.   // C[i] has the number of elements of A equal to i.
7.   **for** i= 2 **to** k
8.       C[i] = C[i] + C[i-1]
9.   // C[i] has the number of elements of A that is at most i.
10.  **for** j=A.length **down** to 1
11.      B[C[A[j]]] = A[j]
12.      C[A[j]] = C[A[j]] – 1
```

- The overall time is $\Theta(k+n)$
  - The for loop of lines 2-3 takes $\Theta(k)$ time.
  - The for loop of lines 4-5 takes $\Theta(n)$ time.
  - The for loop of lines 10-12 takes $\Theta(n)$ time.

Illustration of Counting Sort



Start from the end of A
to the beginning:

Running Time: O(n)

Counting Sort

```
COUNTING-SORT(A, B, k)
1.    let C[1..k] be a new array
2.    for i=1 to k
3.        C[i]=0
4.    for j=1 to A.length
5.        C[A[j]] = C[A[j]] + 1
6.    // C[i] has the number of elements of A equal to i.
7.    for i= 2 to k
8.        C[i] = C[i] + C[i-1]
9.    // C[i] has the number of elements of A that is at most i.
10.   for j=A.length down to 1
11.       B[C[A[j]]] = A[j]
12.       C[A[j]] = C[A[j]] – 1
```

A	1	2	3	4	5
	1	4	3	1	3

B	1	2	3	4	5

Counting Sort

```
COUNTING-SORT(A, B, k)
1.    let C[1..k] be a new array
2.    for i=1 to k
3.        C[i]=0
4.    for j=1 to A.length
5.        C[A[j]] = C[A[j]] + 1
6.    // C[i] has the number of elements of A equal to i.
7.    for i= 2 to k
8.        C[i] = C[i] + C[i-1]
9.    // C[i] has the number of elements of A that is at most i.
10.   for j=A.length down to 1
11.       B[C[A[j]]] = A[j]
12.       C[A[j]] = C[A[j]] – 1
```

	1	2	3	4	5
A	1	4	3	1	3

	1	2	3	4	5
B					

	1	2	3	4
C				

Counting Sort

```
COUNTING-SORT(A, B, k)
1.    let C[1..k] be a new array
2.    for i=1 to k
3.        C[i]=0
4.    for j=1 to A.length
5.        C[A[j]] = C[A[j]] + 1
6.    // C[i] has the number of elements of A equal to i.
7.    for i= 2 to k
8.        C[i] = C[i] + C[i-1]
9.    // C[i] has the number of elements of A that is at most i.
10.   for j=A.length down to 1
11.       B[C[A[j]]] = A[j]
12.       C[A[j]] = C[A[j]] – 1
```

A	1	2	3	4	5
	1	4	3	1	3

B	1	2	3	4	5

C	1	2	3	4
	0	0	0	0

Counting Sort

```
COUNTING-SORT(A, B, k)
1.    let C[1..k] be a new array
2.    for i=1 to k
3.        C[i]=0
4.    for j=1 to A.length
5.        C[A[j]] = C[A[j]] + 1
6.    // C[i] has the number of elements of A equal to i.
7.    for i= 2 to k
8.        C[i] = C[i] + C[i-1]
9.    // C[i] has the number of elements of A that is at most i.
10.   for j=A.length down to 1
11.       B[C[A[j]]] = A[j]
12.       C[A[j]] = C[A[j]] – 1
```

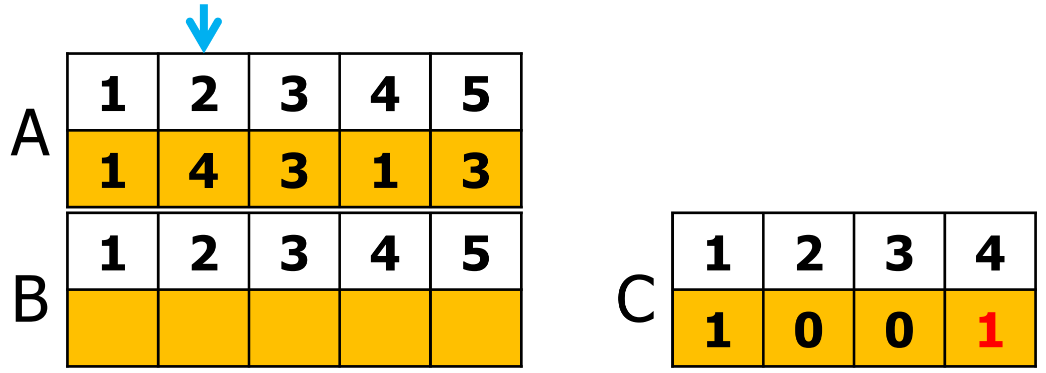
	↓				
A	1	2	3	4	5
	1	4	3	1	3

B	1	2	3	4	5

C	1	2	3	4
	1	0	0	0

Counting Sort

```
COUNTING-SORT(A, B, k)
1.   let C[1..k] be a new array
2.   for i=1 to k
3.       C[i]=0
4.   for j=1 to A.length
5.       C[A[j]] = C[A[j]] + 1
6.   // C[i] has the number of elements of A equal to i.
7.   for i= 2 to k
8.       C[i] = C[i] + C[i-1]
9.   // C[i] has the number of elements of A that is at most i.
10.  for j=A.length down to 1
11.      B[C[A[j]]] = A[j]
12.      C[A[j]] = C[A[j]] – 1
```



A

1	2	3	4	5
1	4	3	1	3

B

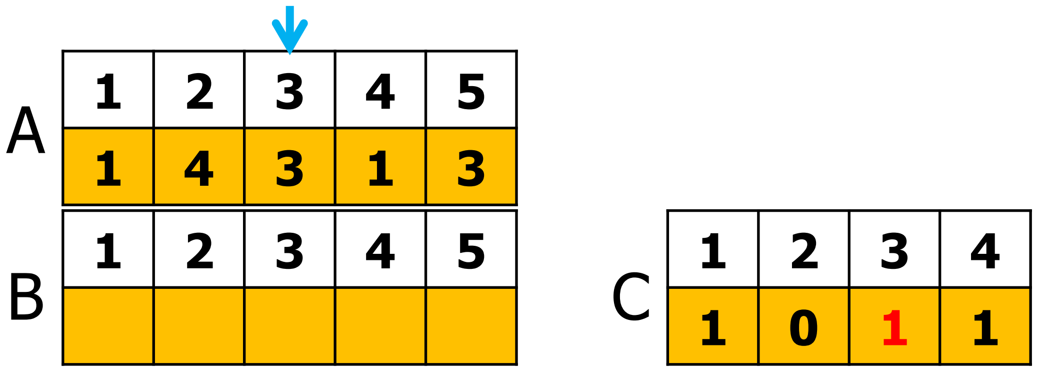
1	2	3	4	5

C

1	2	3	4
1	0	0	1

Counting Sort

```
COUNTING-SORT(A, B, k)
1.    let C[1..k] be a new array
2.    for i=1 to k
3.        C[i]=0
4.    for j=1 to A.length
5.        C[A[j]] = C[A[j]] + 1
6.    // C[i] has the number of elements of A equal to i.
7.    for i= 2 to k
8.        C[i] = C[i] + C[i-1]
9.    // C[i] has the number of elements of A that is at most i.
10.   for j=A.length down to 1
11.       B[C[A[j]]] = A[j]
12.       C[A[j]] = C[A[j]] – 1
```



A

1	2	3	4	5
1	4	3	1	3

B

1	2	3	4	5

C

1	2	3	4
1	0	1	1

Counting Sort

```
COUNTING-SORT(A, B, k)
1.  let C[1..k] be a new array
2.  for i=1 to k
3.      C[i]=0
4.  for j=1 to A.length
5.      C[A[j]] = C[A[j]] + 1
6.  // C[i] has the number of elements of A equal to i.
7.  for i= 2 to k
8.      C[i] = C[i] + C[i-1]
9.  // C[i] has the number of elements of A that is at most i.
10. for j=A.length down to 1
11.     B[C[A[j]]] = A[j]
12.     C[A[j]] = C[A[j]] – 1
```

	1	2	3	4	5
A	1	4	3	1	3

	1	2	3	4	5
B					

	1	2	3	4
C	2	0	1	1

Counting Sort

```
COUNTING-SORT(A, B, k)
1.     let C[1..k] be a new array
2.     for i=1 to k
3.         C[i]=0
4.     for j=1 to A.length
5.         C[A[j]] = C[A[j]] + 1
6.     // C[i] has the number of elements of A equal to i.
7.     for i= 2 to k
8.         C[i] = C[i] + C[i-1]
9.     // C[i] has the number of elements of A that is at most i.
10.    for j=A.length down to 1
11.        B[C[A[j]]] = A[j]
12.        C[A[j]] = C[A[j]] – 1
```

					↓
A	1	2	3	4	5
	1	4	3	1	3

B	1	2	3	4	5

C	1	2	3	4
	2	0	2	1

Counting Sort

```
COUNTING-SORT(A, B, k)
1.    let C[1..k] be a new array
2.    for i=1 to k
3.        C[i]=0
4.    for j=1 to A.length
5.        C[A[j]] = C[A[j]] + 1
6.    // C[i] has the number of elements of A equal to i.
7.    for i= 2 to k
8.        C[i] = C[i] + C[i-1]
9.    // C[i] has the number of elements of A that is at most i.
10.   for j=A.length down to 1
11.       B[C[A[j]]] = A[j]
12.       C[A[j]] = C[A[j]] – 1
```

A

1	2	3	4	5
1	4	3	1	3

B

1	2	3	4	5

C

1	2	3	4
2	0	2	1

Counting Sort

```
COUNTING-SORT(A, B, k)
1.    let C[1..k] be a new array
2.    for i=1 to k
3.        C[i]=0
4.    for j=1 to A.length
5.        C[A[j]] = C[A[j]] + 1
6.    // C[i] has the number of elements of A equal to i.
7.    for i= 2 to k
8.        C[i] = C[i] + C[i-1]
9.    // C[i] has the number of elements of A that is at most i.
10.   for j=A.length down to 1
11.       B[C[A[j]]] = A[j]
12.       C[A[j]] = C[A[j]] – 1
```

	1	2	3	4	5
A	1	4	3	1	3

	1	2	3	4	5
B					

	1	2	3	4
C	2	2	2	1

Counting Sort

```
COUNTING-SORT(A, B, k)
1.   let C[1..k] be a new array
2.   for i=1 to k
3.       C[i]=0
4.   for j=1 to A.length
5.       C[A[j]] = C[A[j]] + 1
6.   // C[i] has the number of elements of A equal to i.
7.   for i= 2 to k
8.       C[i] = C[i] + C[i-1]
9.   // C[i] has the number of elements of A that is at most i.
10.  for j=A.length down to 1
11.      B[C[A[j]]] = A[j]
12.      C[A[j]] = C[A[j]] – 1
```

	1	2	3	4	5
A	1	4	3	1	3

	1	2	3	4	5
B					

	1	2	↓ 3	4
C	2	2	4	1

Counting Sort

```
COUNTING-SORT(A, B, k)
1.    let C[1..k] be a new array
2.    for i=1 to k
3.        C[i]=0
4.    for j=1 to A.length
5.        C[A[j]] = C[A[j]] + 1
6.    // C[i] has the number of elements of A equal to i.
7.    for i= 2 to k
8.        C[i] = C[i] + C[i-1]
9.    // C[i] has the number of elements of A that is at most i.
10.   for j=A.length down to 1
11.       B[C[A[j]]] = A[j]
12.       C[A[j]] = C[A[j]] – 1
```

A

1	2	3	4	5
1	4	3	1	3

B

1	2	3	4	5

C

1	2	3	4
2	2	4	5

Counting Sort

```
COUNTING-SORT(A, B, k)
1.    let C[1..k] be a new array
2.    for i=1 to k
3.        C[i]=0
4.    for j=1 to A.length
5.        C[A[j]] = C[A[j]] + 1
6.    // C[i] has the number of elements of A equal to i.
7.    for i= 2 to k
8.        C[i] = C[i] + C[i-1]
9.    // C[i] has the number of elements of A that is at most i.
10.   for j=A.length down to 1
11.       B[C[A[j]]] = A[j]
12.       C[A[j]] = C[A[j]] – 1
```

A	1	2	3	4	5
	1	4	3	1	3

B	1	2	3	4	5

C	1	2	3	4
	2	2	4	5

Counting Sort

```
COUNTING-SORT(A, B, k)
1.    let C[1..k] be a new array
2.    for i=1 to k
3.        C[i]=0
4.    for j=1 to A.length
5.        C[A[j]] = C[A[j]] + 1
6.    // C[i] has the number of elements of A equal to i.
7.    for i= 2 to k
8.        C[i] = C[i] + C[i-1]
9.    // C[i] has the number of elements of A that is at most i.
10.   for j=A.length down to 1
11.       B[C[A[j]]] = A[j]
12.       C[A[j]] = C[A[j]] – 1
```

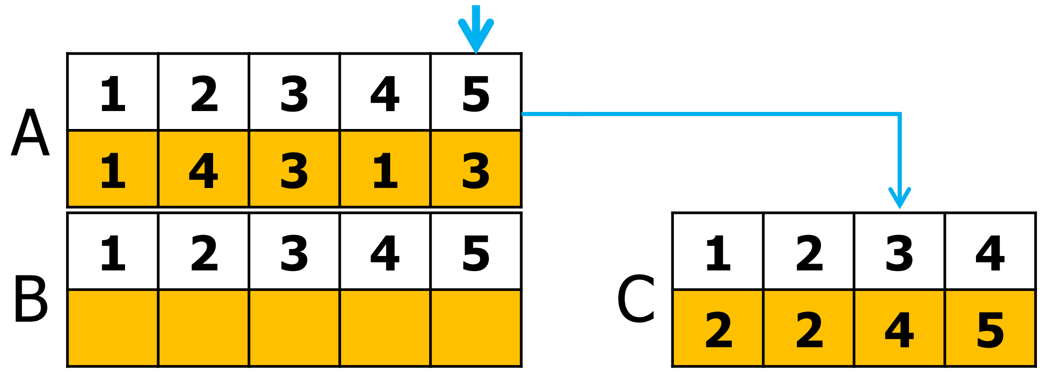
	1	2	3	4	5
A	1	4	3	1	3

	1	2	3	4	5
B					

	1	2	3	4
C	2	2	4	5

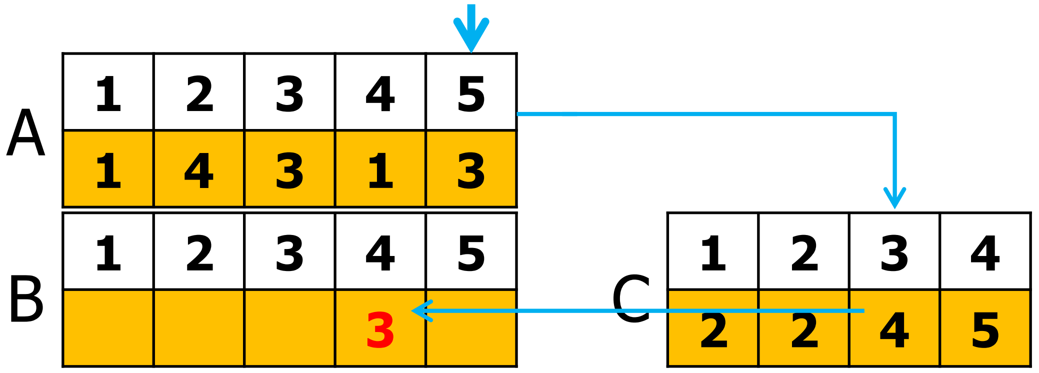
Counting Sort

```
COUNTING-SORT(A, B, k)
1.    let C[1..k] be a new array
2.    for i=1 to k
3.        C[i]=0
4.    for j=1 to A.length
5.        C[A[j]] = C[A[j]] + 1
6.    // C[i] has the number of elements of A equal to i.
7.    for i= 2 to k
8.        C[i] = C[i] + C[i-1]
9.    // C[i] has the number of elements of A that is at most i.
10.   for j=A.length down to 1
11.       B[C[A[j]]] = A[j]
12.       C[A[j]] = C[A[j]] – 1
```



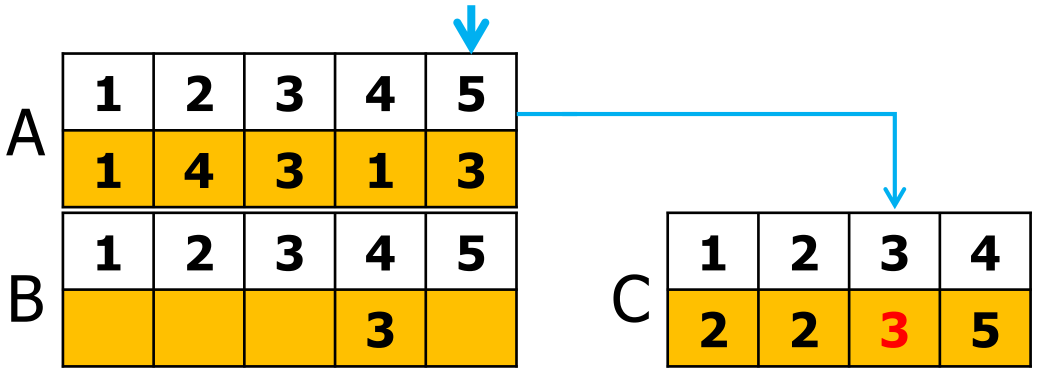
Counting Sort

```
COUNTING-SORT(A, B, k)
1.    let C[1..k] be a new array
2.    for i=1 to k
3.        C[i]=0
4.    for j=1 to A.length
5.        C[A[j]] = C[A[j]] + 1
6.    // C[i] has the number of elements of A equal to i.
7.    for i= 2 to k
8.        C[i] = C[i] + C[i-1]
9.    // C[i] has the number of elements of A that is at most i.
10.   for j=A.length down to 1
11.       B[C[A[j]]] = A[j]
12.       C[A[j]] = C[A[j]] – 1
```



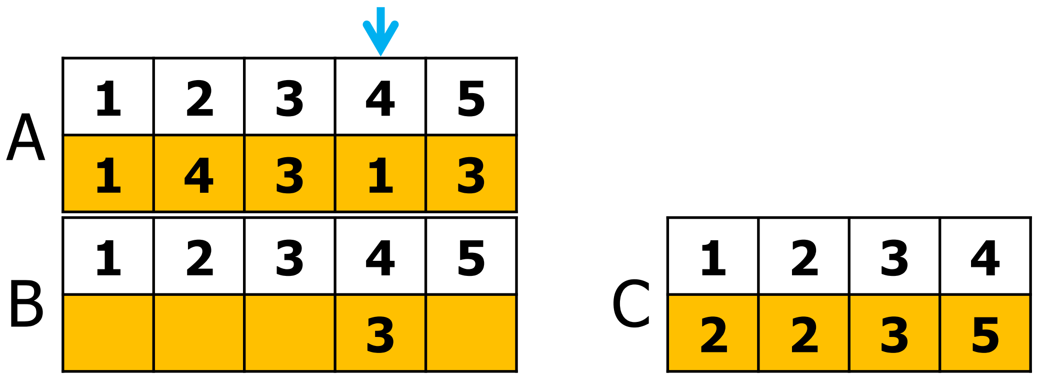
Counting Sort

```
COUNTING-SORT(A, B, k)
1.    let C[1..k] be a new array
2.    for i=1 to k
3.        C[i]=0
4.    for j=1 to A.length
5.        C[A[j]] = C[A[j]] + 1
6.    // C[i] has the number of elements of A equal to i.
7.    for i= 2 to k
8.        C[i] = C[i] + C[i-1]
9.    // C[i] has the number of elements of A that is at most i.
10.   for j=A.length down to 1
11.       B[C[A[j]]] = A[j]
12.       C[A[j]] = C[A[j]] – 1
```



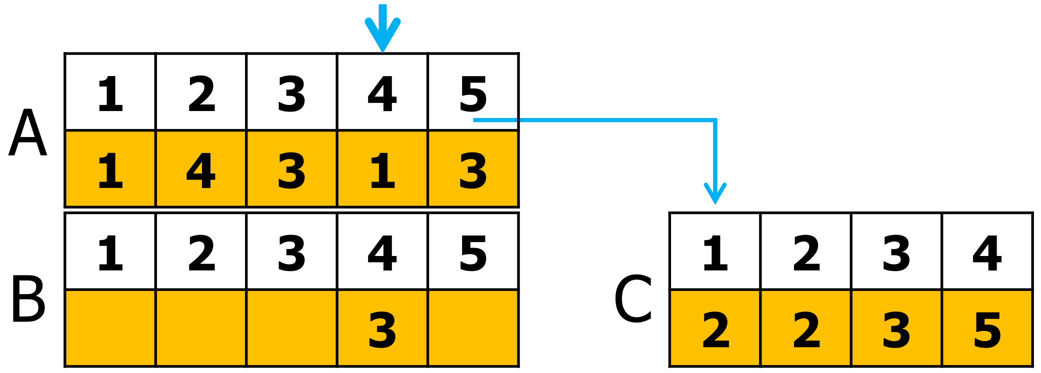
Counting Sort

```
COUNTING-SORT(A, B, k)
1.   let C[1..k] be a new array
2.   for i=1 to k
3.       C[i]=0
4.   for j=1 to A.length
5.       C[A[j]] = C[A[j]] + 1
6.   // C[i] has the number of elements of A equal to i.
7.   for i= 2 to k
8.       C[i] = C[i] + C[i-1]
9.   // C[i] has the number of elements of A that is at most i.
10.  for j=A.length down to 1
11.      B[C[A[j]]] = A[j]
12.      C[A[j]] = C[A[j]] – 1
```



Counting Sort

```
COUNTING-SORT(A, B, k)
1.    let C[1..k] be a new array
2.    for i=1 to k
3.        C[i]=0
4.    for j=1 to A.length
5.        C[A[j]] = C[A[j]] + 1
6.    // C[i] has the number of elements of A equal to i.
7.    for i= 2 to k
8.        C[i] = C[i] + C[i-1]
9.    // C[i] has the number of elements of A that is at most i.
10.   for j=A.length down to 1
11.       B[C[A[j]]] = A[j]
12.       C[A[j]] = C[A[j]] – 1
```



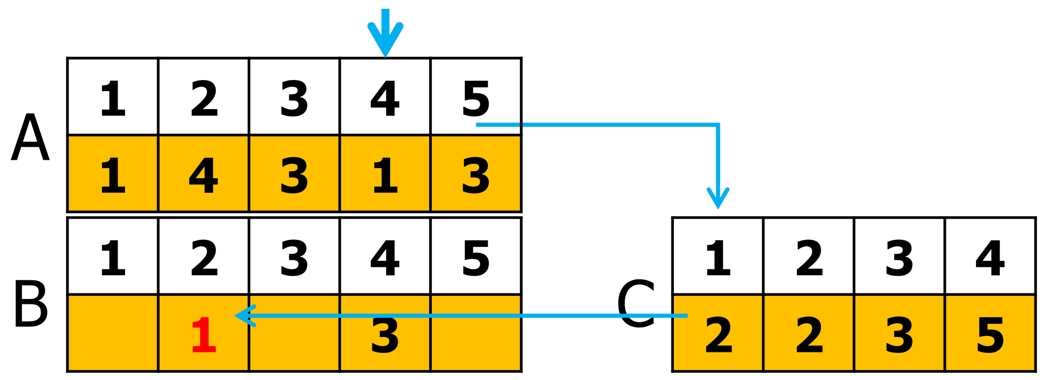
A	1	2	3	4	5
	1	4	3	1	3

B	1	2	3	4	5
				3	

C	1	2	3	4
	2	2	3	5

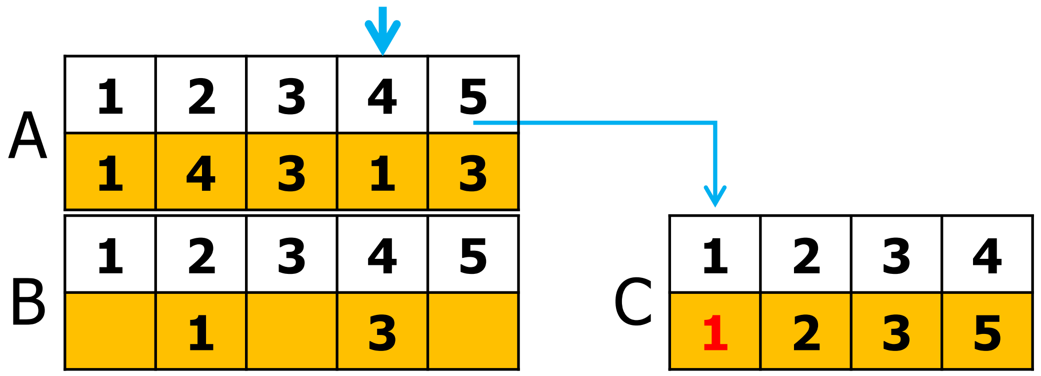
Counting Sort

```
COUNTING-SORT(A, B, k)
1.    let C[1..k] be a new array
2.    for i=1 to k
3.        C[i]=0
4.    for j=1 to A.length
5.        C[A[j]] = C[A[j]] + 1
6.    // C[i] has the number of elements of A equal to i.
7.    for i= 2 to k
8.        C[i] = C[i] + C[i-1]
9.    // C[i] has the number of elements of A that is at most i.
10.   for j=A.length down to 1
11.       B[C[A[j]]] = A[j]
12.       C[A[j]] = C[A[j]] – 1
```



Counting Sort

```
COUNTING-SORT(A, B, k)
1.     let C[1..k] be a new array
2.     for i=1 to k
3.         C[i]=0
4.     for j=1 to A.length
5.         C[A[j]] = C[A[j]] + 1
6.     // C[i] has the number of elements of A equal to i.
7.     for i= 2 to k
8.         C[i] = C[i] + C[i-1]
9.     // C[i] has the number of elements of A that is at most i.
10.    for j=A.length down to 1
11.        B[C[A[j]]] = A[j]
12.        C[A[j]] = C[A[j]] – 1
```



A

1	2	3	4	5
1	4	3	1	3

B

1	2	3	4	5
	1		3	

C

1	2	3	4
1	2	3	5

Counting Sort

```
COUNTING-SORT(A, B, k)
1.    let C[1..k] be a new array
2.    for i=1 to k
3.        C[i]=0
4.    for j=1 to A.length
5.        C[A[j]] = C[A[j]] + 1
6.    // C[i] has the number of elements of A equal to i.
7.    for i= 2 to k
8.        C[i] = C[i] + C[i-1]
9.    // C[i] has the number of elements of A that is at most i.
10.   for j=A.length down to 1
11.       B[C[A[j]]] = A[j]
12.       C[A[j]] = C[A[j]] – 1
```

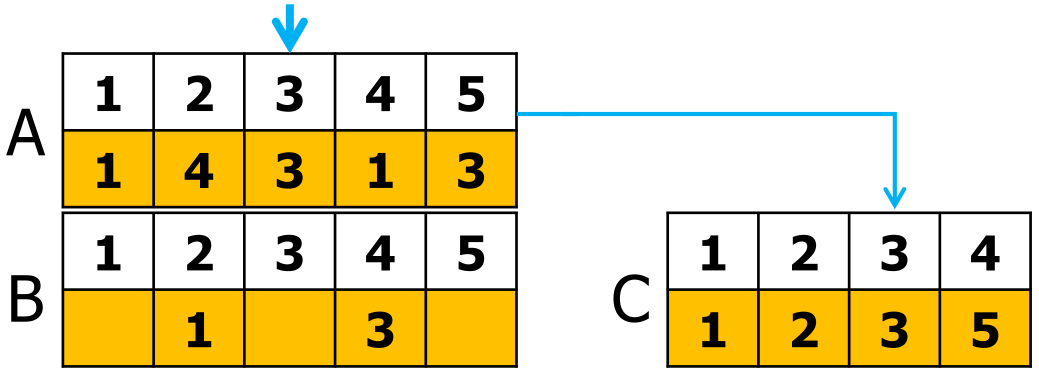
			↓		
A	1	2	3	4	5
	1	4	3	1	3

B	1	2	3	4	5
		1		3	

C	1	2	3	4
	1	2	3	5

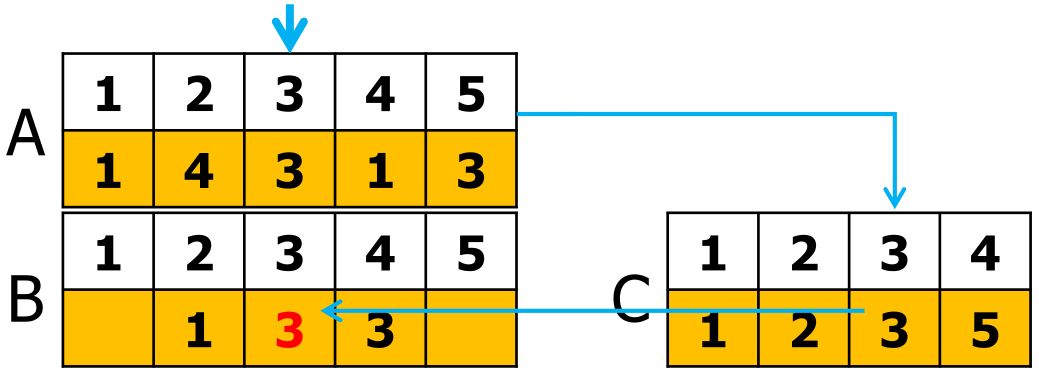
Counting Sort

```
COUNTING-SORT(A, B, k)
1.     let C[1..k] be a new array
2.     for i=1 to k
3.         C[i]=0
4.     for j=1 to A.length
5.         C[A[j]] = C[A[j]] + 1
6.     // C[i] has the number of elements of A equal to i.
7.     for i= 2 to k
8.         C[i] = C[i] + C[i-1]
9.     // C[i] has the number of elements of A that is at most i.
10.    for j=A.length down to 1
11.        B[C[A[j]]] = A[j]
12.        C[A[j]] = C[A[j]] – 1
```



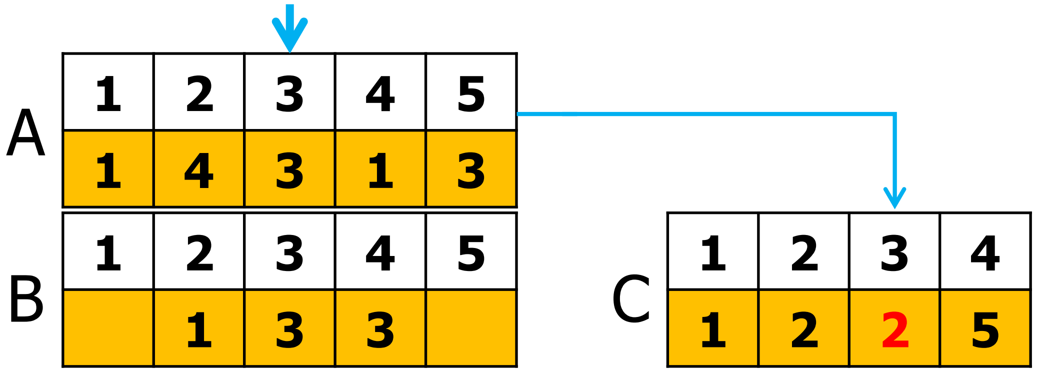
Counting Sort

```
COUNTING-SORT(A, B, k)
1.    let C[1..k] be a new array
2.    for i=1 to k
3.        C[i]=0
4.    for j=1 to A.length
5.        C[A[j]] = C[A[j]] + 1
6.    // C[i] has the number of elements of A equal to i.
7.    for i= 2 to k
8.        C[i] = C[i] + C[i-1]
9.    // C[i] has the number of elements of A that is at most i.
10.   for j=A.length down to 1
11.       B[C[A[j]]] = A[j]
12.       C[A[j]] = C[A[j]] – 1
```



Counting Sort

```
COUNTING-SORT(A, B, k)
1.    let C[1..k] be a new array
2.    for i=1 to k
3.        C[i]=0
4.    for j=1 to A.length
5.        C[A[j]] = C[A[j]] + 1
6.    // C[i] has the number of elements of A equal to i.
7.    for i= 2 to k
8.        C[i] = C[i] + C[i-1]
9.    // C[i] has the number of elements of A that is at most i.
10.   for j=A.length down to 1
11.       B[C[A[j]]] = A[j]
12.       C[A[j]] = C[A[j]] – 1
```



A

1	2	3	4	5
1	4	3	1	3

B

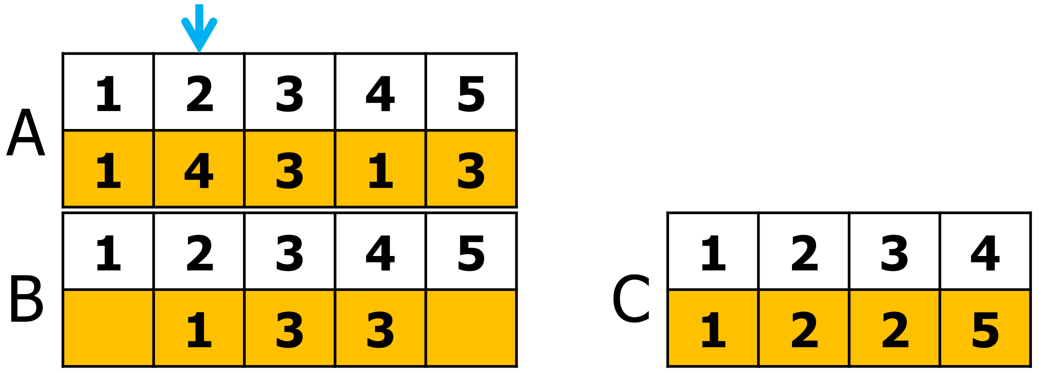
1	2	3	4	5
	1	3	3	

C

1	2	3	4
1	2	2	5

Counting Sort

```
COUNTING-SORT(A, B, k)
1.   let C[1..k] be a new array
2.   for i=1 to k
3.       C[i]=0
4.   for j=1 to A.length
5.       C[A[j]] = C[A[j]] + 1
6.   // C[i] has the number of elements of A equal to i.
7.   for i= 2 to k
8.       C[i] = C[i] + C[i-1]
9.   // C[i] has the number of elements of A that is at most i.
10.  for j=A.length down to 1
11.      B[C[A[j]]] = A[j]
12.      C[A[j]] = C[A[j]] – 1
```



A

1	2	3	4	5
1	4	3	1	3

B

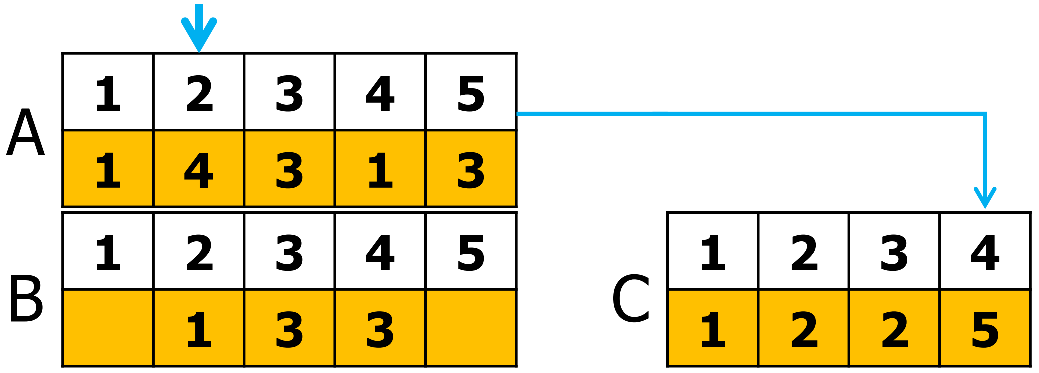
1	2	3	4	5
	1	3	3	

C

1	2	3	4
1	2	2	5

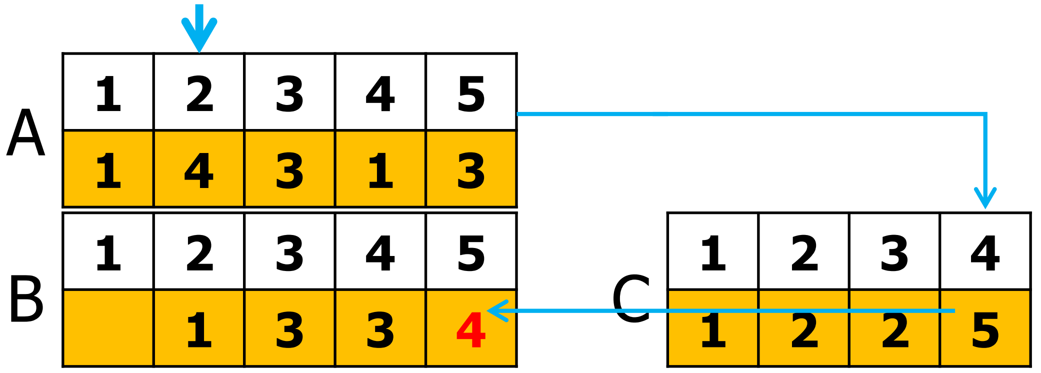
Counting Sort

```
COUNTING-SORT(A, B, k)
1.    let C[1..k] be a new array
2.    for i=1 to k
3.        C[i]=0
4.    for j=1 to A.length
5.        C[A[j]] = C[A[j]] + 1
6.    // C[i] has the number of elements of A equal to i.
7.    for i= 2 to k
8.        C[i] = C[i] + C[i-1]
9.    // C[i] has the number of elements of A that is at most i.
10.   for j=A.length down to 1
11.       B[C[A[j]]] = A[j]
12.       C[A[j]] = C[A[j]] – 1
```



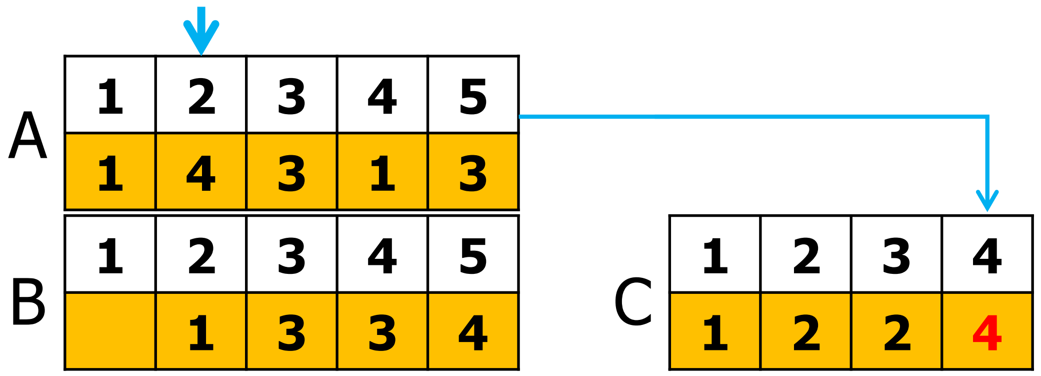
Counting Sort

```
COUNTING-SORT(A, B, k)
1.     let C[1..k] be a new array
2.     for i=1 to k
3.         C[i]=0
4.     for j=1 to A.length
5.         C[A[j]] = C[A[j]] + 1
6.     // C[i] has the number of elements of A equal to i.
7.     for i= 2 to k
8.         C[i] = C[i] + C[i-1]
9.     // C[i] has the number of elements of A that is at most i.
10.    for j=A.length down to 1
11.        B[C[A[j]]] = A[j]
12.        C[A[j]] = C[A[j]] – 1
```



Counting Sort

```
COUNTING-SORT(A, B, k)
1.    let C[1..k] be a new array
2.    for i=1 to k
3.        C[i]=0
4.    for j=1 to A.length
5.        C[A[j]] = C[A[j]] + 1
6.    // C[i] has the number of elements of A equal to i.
7.    for i= 2 to k
8.        C[i] = C[i] + C[i-1]
9.    // C[i] has the number of elements of A that is at most i.
10.   for j=A.length down to 1
11.       B[C[A[j]]] = A[j]
12.       C[A[j]] = C[A[j]] – 1
```



Counting Sort

```
COUNTING-SORT(A, B, k)
1.    let C[1..k] be a new array
2.    for i=1 to k
3.        C[i]=0
4.    for j=1 to A.length
5.        C[A[j]] = C[A[j]] + 1
6.    // C[i] has the number of elements of A equal to i.
7.    for i= 2 to k
8.        C[i] = C[i] + C[i-1]
9.    // C[i] has the number of elements of A that is at most i.
10.   for j=A.length down to 1
11.       B[C[A[j]]] = A[j]
12.       C[A[j]] = C[A[j]] – 1
```

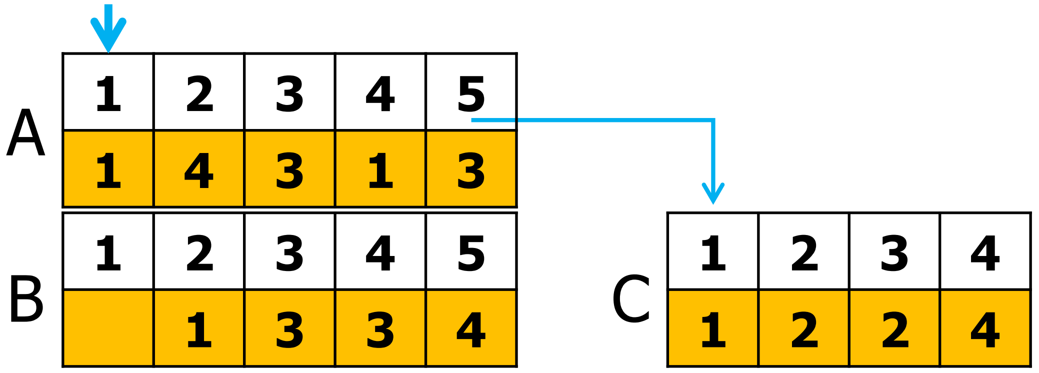
A	1	2	3	4	5
	1	4	3	1	3

B	1	2	3	4	5
		1	3	3	4

C	1	2	3	4
	1	2	2	4

Counting Sort

```
COUNTING-SORT(A, B, k)
1.    let C[1..k] be a new array
2.    for i=1 to k
3.        C[i]=0
4.    for j=1 to A.length
5.        C[A[j]] = C[A[j]] + 1
6.    // C[i] has the number of elements of A equal to i.
7.    for i= 2 to k
8.        C[i] = C[i] + C[i-1]
9.    // C[i] has the number of elements of A that is at most i.
10.   for j=A.length down to 1
11.       B[C[A[j]]] = A[j]
12.       C[A[j]] = C[A[j]] – 1
```



A

1	2	3	4	5
1	4	3	1	3

B

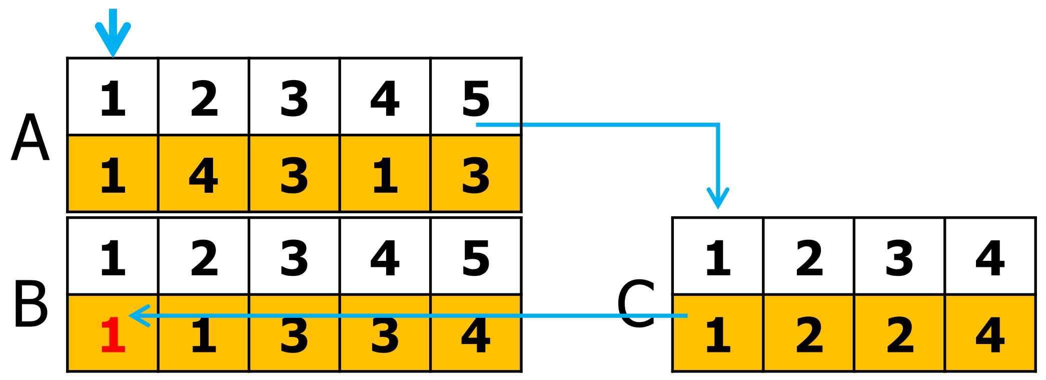
1	2	3	4	5
	1	3	3	4

C

1	2	3	4
1	2	2	4

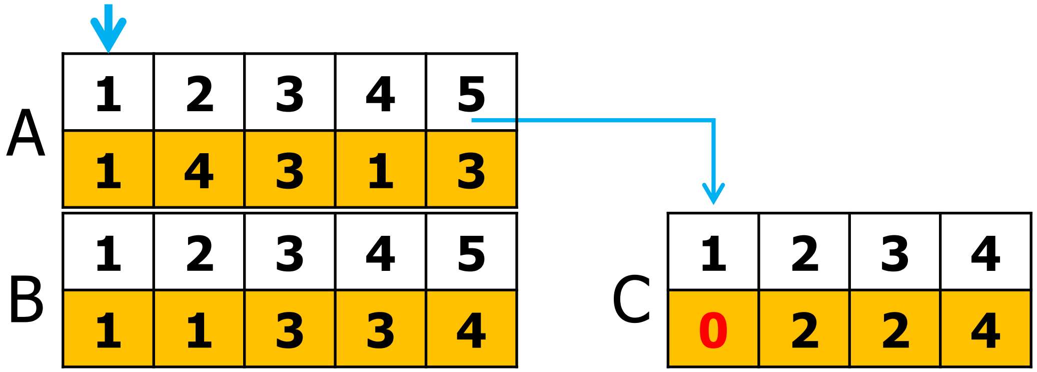
Counting Sort

```
COUNTING-SORT(A, B, k)
1.    let C[1..k] be a new array
2.    for i=1 to k
3.        C[i]=0
4.    for j=1 to A.length
5.        C[A[j]] = C[A[j]] + 1
6.    // C[i] has the number of elements of A equal to i.
7.    for i= 2 to k
8.        C[i] = C[i] + C[i-1]
9.    // C[i] has the number of elements of A that is at most i.
10.   for j=A.length down to 1
11.       B[C[A[j]]] = A[j]
12.       C[A[j]] = C[A[j]] – 1
```



Counting Sort

```
COUNTING-SORT(A, B, k)
1.   let C[1..k] be a new array
2.   for i=1 to k
3.       C[i]=0
4.   for j=1 to A.length
5.       C[A[j]] = C[A[j]] + 1
6.   // C[i] has the number of elements of A equal to i.
7.   for i= 2 to k
8.       C[i] = C[i] + C[i-1]
9.   // C[i] has the number of elements of A that is at most i.
10.  for j=A.length down to 1
11.      B[C[A[j]]] = A[j]
12.      C[A[j]] = C[A[j]] – 1
```



A	1	2	3	4	5
	1	4	3	1	3

B	1	2	3	4	5
	1	1	3	3	4

C	1	2	3	4
	0	2	2	4

Counting Sort

```
COUNTING-SORT(A, B, k)
1.    let C[1..k] be a new array
2.    for i=1 to k
3.        C[i]=0
4.    for j=1 to A.length
5.        C[A[j]] = C[A[j]] + 1
6.    // C[i] has the number of elements of A equal to i.
7.    for i= 2 to k
8.        C[i] = C[i] + C[i-1]
9.    // C[i] has the number of elements of A that is at most i.
10.   for j=A.length down to 1
11.       B[C[A[j]]] = A[j]
12.       C[A[j]] = C[A[j]] – 1
```

A

1	2	3	4	5
1	4	3	1	3

B

1	2	3	4	5
1	1	3	3	4

C

1	2	3	4
0	2	2	4

Property of Counting Sort

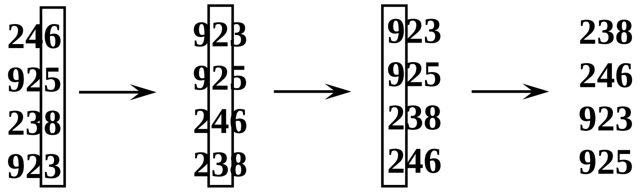
- It is not a comparison sort.
- No comparisons between input elements occur anywhere in the code.
- The $\Omega(n \lg n)$ lower bound for sorting does not apply when we depart from the comparison sort model.
- The property of stability is important when satellite data are carried around with the element being sorted.
  - The numbers with the same value appear in the output array in the same order as in the input array.

Radix Sort

- Counting Sort works only for small integers.
- Radix Sort sorts the numbers one digit at a time.
  - Each input has d decimal one digits (or digits in any base)
  - We use a stable sorting algorithm like Counting Sort
  - We sort repetitively, starting from the lowest order digit finishing at the highest digit.
  - Since the sorting algorithm is *stable*, if the numbers are sorted with respect low order digits and are later sorted with high order digits, numbers having the same high order digit will still remain sorted w.r.t their low order digit.

Radix Sort

```
RADIX-SORT(A,d)
1.  for i=1 to d
2.      use a stable sort to sort the array A on digit i
```



- Running Time : $O(d \times (n+k)) = O(n)$
  (k : number of values that a digit can have)

Radix Sort

- Lemma 8.3
  - Given n d-digit numbers in which each digit can take on up to k possible values, RADIX-SORT correctly sorts these numbers in $\Theta(d(n+k))$ time, if the stable sort takes $\Theta(n+k)$ time.
- Proof
  - The correctness of radix sort follows by induction on the column being sorted (see Exercise 8.3-3).
  - The analysis of the running time depends on the stable sort used as the intermediate sorting algorithm.
  - When each digit is in the range 0 to k-1 (so that it can take on k possible values), and k is not too large, counting sort is the obvious choice.
  - Each pass over n d-digit numbers then takes $\Theta(n+k)$ time.
  - There are d passes, and so radix sort is $\Theta(d(n+k))$ time.

Radix Sort

- Lemma 8.4
  - Given n b-bit numbers and any positive integer $r \leq b$, RADIX-SORT correctly sorts in $\Theta\big((b/r)(n + 2^r)\big)$ time, if the stable sort takes $\Theta(n + k)$ time for inputs in the range 0 to k.
- Proof
  - For a value r $\leq b,$ we view each key as having $d = \lceil b/r \rceil$ digits of r bits each.
  - Each digit is an integer 0 to $2^r - 1$, so that we can use counting sort with $k=2^r - 1$.
  - e.g.) A 32-bit word has four 8-bit digits - $b = 32, r = 8, k = 2^r - 1 = 255, d = b/r = 4.$
  - Each pass of counting sort $\Theta(n + k) = \Theta(n + 2^r)$, and there are d passes.
  - Total running time $\Theta\big(d(n + 2^r)\big) = \Theta\big((b/r)(n + 2^r)\big)$.

Bucket Sort

- Assumes that the n input numbers are drawn from a uniform distribution.
- Like counting sort, it is fast because it assumes that the input is generated by a random process that distributes elements uniformly and independently over the interval $[0,1)$.
- Average-case running time is $O(n)$.

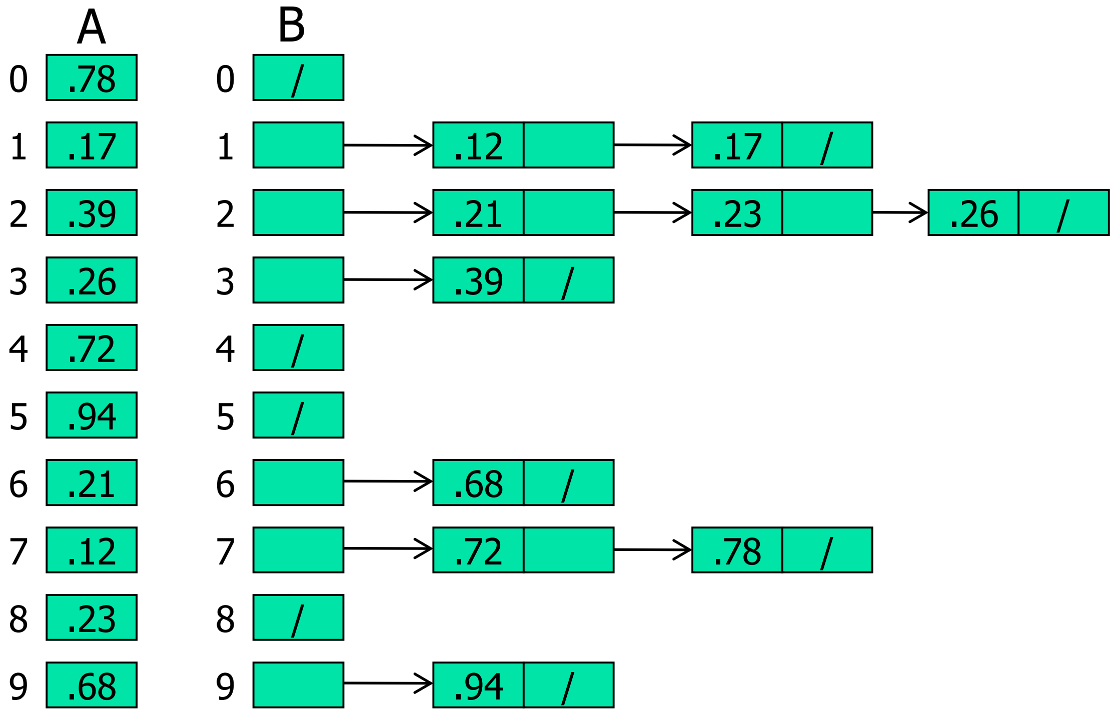
Bucket Sort

- Divide the interval $[0,1)$ into n equal-sized subintervals (buckets).
- Distributes the n input numbers into the buckets.
- Sort the numbers in each bucket and go through the buckets in order.

Bucket Sort

```
BUCKET-SORT(A)
1.  let B[0…n-1] be a new array
2.  n=A.length
3.  for i=0 to n-1
4.      make B[i] an empty list
5.  for i=1 to n
6.      insert A[i] into list B[$\lfloor nA[i] \rfloor$]
7.  for i=0 to n-1
8.      sort list B[i] with insertion sort
9.  concatenate the list B[0],B[1],…,B[n-1] together in order
```

Bucket Sort


A
B
0
1
2
3
4
5
6
7
8
9
.78
.17
.39
.26
.72
.94
.21
.12
.23
.68
/
.12
.17
.21
.23
.26
.39
.68
.72
.78
.94

Analysis of Bucket Sort

- Let n_i be the random variable denoting the number of elements placed in bucket B[i]
- Since the insertion sort runs in quadratic time, the running time of bucket sort is

$$T(n) = \Theta(n) + \sum_{i=0}^{n-1} O\left(n_i^2\right)$$

Analysis of Bucket Sort

- We now analyze the average-case running time of bucket sort, by computing the expected value of the running time, where we take the expectation over the input distribution.
- Taking expectations of both sides of

$$T(n) = \Theta(n) + \sum_{i=0}^{n-1} O(n_i^2)$$

and using linearity of expectation, we have

$$E[T(n)] = E\left[\Theta(n) + \sum_{i=0}^{n-1} O(n_i^2)\right]$$

$$= \Theta(n) + \sum_{i=0}^{n-1} E[O(n_i^2)] = \Theta(n) + \sum_{i=0}^{n-1} O(E[n_i^2])$$

Analysis of Bucket Sort

$$T(n) = \Theta(n) + \sum_{i=0}^{n-1} O\left(n_i^2\right)$$

$$E[T(n)] = E\left[\Theta(n) + \sum_{i=0}^{n-1} O\left(n_i^2\right)\right] = \Theta(n) + \sum_{i=0}^{n-1} E\left[O\left(n_i^2\right)\right] = \Theta(n) + \sum_{i=0}^{n-1} O\left(E\left[n_i^2\right]\right)$$

When $E\left[n_i^2\right] = 2 - 1/n \;\Rightarrow$

$E[T(n)] = \Theta(n) + n \cdot O(2 - 1/n) = \Theta(n)$

Analysis of Bucket Sort

- We claim that $E[n_i^2] = 2 - 1/n$ for i=0,...,n-1.
- It is no surprise that each bucket i has the same value of $E[n_i^2]$, since each value in the input array A is equally likely to fall in any bucket.
- To prove our claim, we define indicator random variables
  - $X_{ij} = \mathrm{I}\{A[j] \text{ falls in bucket } i\}$,
- Thus, $n_i = \sum_{j=1}^{n} X_{ij}$

Analysis of Bucket Sort

- To compute $E\left[n_i^2\right]$, we expand the square and regroup terms:

$$\mathrm{E}\left[n_i^2\right] = \mathrm{E}\left[\left(\sum_{j=1}^{n} X_{ij}\right)^2\right] = \mathrm{E}\left[\sum_{j=1}^{n}\sum_{k=1}^{n} X_{ij}X_{ik}\right]$$

$$= \mathrm{E}\left[\sum_{j=1}^{n} X_{ij}^2 + \sum_{j=1}^{n}\sum_{\substack{1\le k\le n \\ k\ne j}} X_{ij}X_{ik}\right]$$

$$= \sum_{j=1}^{n} \mathrm{E}\left[X_{ij}^2\right] + \sum_{j=1}^{n}\sum_{\substack{1\le k\le n \\ k\ne j}} \mathrm{E}\left[X_{ij}X_{ik}\right] \quad (8.3)$$

Analysis of Bucket Sort

- Indicator random variable X_{ij} is 1 with probability 1/n and 0 otherwise.
- Thus,

  $$\mathrm{E}[X_{ij}^2] = 1^2 \cdot 1/n + 0^2 \cdot (1 - 1/n) = 1/n$$
- When k ≠ j , the variables X_{ij} and X_{ik} are independent, and hence

$$\mathrm{E}[X_{ij}X_{ik}] = \mathrm{E}[\mathrm{X_{ij}}]\mathrm{E}[\mathrm{X_{ik}}] = 1/n \cdot 1/n = 1/n^2$$

Analysis of Bucket Sort

- Substituting these two expected values in equation (8.3), we obtain

$$\mathrm{E}\left[n_i^2\right] = \sum_{j=1}^{n} \mathrm{E}\left[X_{ij}^2\right] + \sum_{j=1}^{n} \sum_{\substack{1 \le k \le n \\ k \ne j}} \mathrm{E}\left[X_{ij} X_{ik}\right]$$

$$= \sum_{j=1}^{n} 1/n + \sum_{j=1}^{n} \sum_{\substack{1 \le k \le n \\ k \ne j}} 1/n^2$$

$$= \mathrm{n} \cdot 1/n + n(n-1) \cdot 1/n^2$$

$$= 1 + (n-1)/n = 2 - 1/n$$

Analysis of Bucket Sort

$X_{ij} = \mathrm{I}\{A[j] \text{ falls in bucket } i\}$ $\quad n_i = \sum_{j=1}^{n} X_{ij}$

$$\mathrm{E}[n_i^2] = \mathrm{E}\left[\left(\sum_{j=1}^{n} X_{ij}\right)^2\right] = \mathrm{E}\left[\sum_{j=1}^{n}\sum_{k=1}^{n} X_{ij}X_{ik}\right]$$

$$= \mathrm{E}\left[\sum_{j=1}^{n} X_{ij}^2 + \sum_{j=1}^{n}\sum_{\substack{1\le k\le n\\ k\ne j}} X_{ij}X_{ik}\right]$$

$$= \sum_{j=1}^{n} \mathrm{E}[X_{ij}^2] + \sum_{j=1}^{n}\sum_{\substack{1\le k\le n\\ k\ne j}} \mathrm{E}[X_{ij}X_{ik}]$$

Indicator random variable $X_{ij} = \begin{cases} 1 & p = 1/n \\ 0 & \text{otherwise} \end{cases}$

$\mathrm{E}[X_{ij}^2] = 1^2 \cdot 1/n + 0^2 \cdot (1 - 1/n) = 1/n$

X_{ij} *and* X_{ik} are independent

When $k \ne j$, $\mathrm{E}[X_{ij}X_{ik}] = \mathrm{E}[\mathrm{X_{ij}}]\mathrm{E}[\mathrm{X_{ik}}] = 1/n \cdot 1/n = 1/n^2$

Thus, $\quad E[n_i^2] = \sum_{j=1}^{n} 1/n + \sum_{j=1}^{n}\sum_{\substack{1\le k\le n\\ k\ne j}} 1/n^2$

$$= \mathrm{n} \cdot 1/n + n(n-1) \cdot 1/n^2 = 1 + (n-1)/n = 2 - 1/n$$

Any Question?

Introduction to Algorithms (Chapter 9: Median and Order Statistics)

Kyuseok Shim

Electrical and Computer Engineering
Seoul National University

Outline

- This chapter addresses the problem of selecting the i-th order statistic from a set of n distinct numbers.

K-th Smallest Number Selection Problem

- We assume for convenience that the set contains distinct numbers, although virtually everything that we do extends to the situation in which a set contains repeated values.
- We formally specify the selection problem as follows:
  - Input: A set of n (distinct) numbers and a number i with $1 \leq i \leq n$.
  - Output: The element $x \in A$ that is larger than exactly i-1 other elements of A.
- We can be solved in O(n log n) time by sorting and simply indexing the i-th element.

A Naïve Method

Using an Array with size i

$i = 8$

index	1	2	3	4	5	6	7	8	9	10	11	12
value	3	2	1	6	13	5	12	15	11	10	4	9

index	1	2	3	4	5	6	7	8
value								

Using an Array with size i

$i = 8$

index	1	2	3	4	5	6	7	8	9	10	11	12
value	3	2	1	6	13	5	12	15	11	10	4	9

$\uparrow$

index	1	2	3	4	5	6	7	8
value	3							

Using an Array with size i

$i = 8$

index	1	2	3	4	5	6	7	8	9	10	11	12
value	3	2	1	6	13	5	12	15	11	10	4	9

index	1	2	3	4	5	6	7	8
value	3	2						

Using an Array with size i

$i = 8$

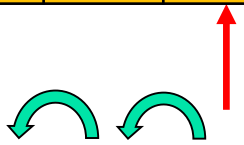
index	1	2	3	4	5	6	7	8	9	10	11	12
value	3	2	1	6	13	5	12	15	11	10	4	9

index	1	2	3	4	5	6	7	8
value	2	3						

Using an Array with size i

$i = 8$

index	1	2	3	4	5	6	7	8	9	10	11	12
value	3	2	1	6	13	5	12	15	11	10	4	9



index	1	2	3	4	5	6	7	8
value	1	2	3					

Using an Array with size i

$i = 8$

index	1	2	3	4	5	6	7	8	9	10	11	12
value	3	2	1	6	13	5	12	15	11	10	4	9

index	1	2	3	4	5	6	7	8
value	1	2	3	6				

Using an Array with size i

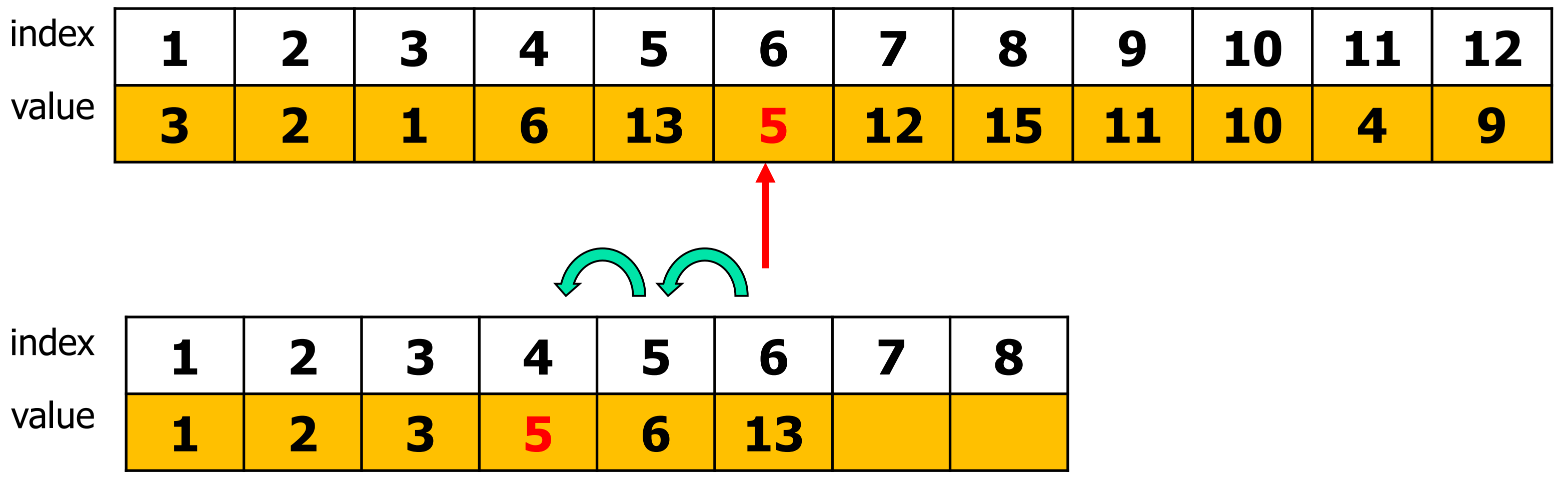
$i = 8$

index	1	2	3	4	5	6	7	8	9	10	11	12
value	3	2	1	6	13	5	12	15	11	10	4	9

index	1	2	3	4	5	6	7	8
value	1	2	3	6	13			

Using an Array with size i

$i = 8$



index	1	2	3	4	5	6	7	8	9	10	11	12
value	3	2	1	6	13	5	12	15	11	10	4	9

index	1	2	3	4	5	6	7	8
value	1	2	3	5	6	13		

Using an Array with size i

$i = 8$

index	1	2	3	4	5	6	7	8	9	10	11	12
value	3	2	1	6	13	5	12	15	11	10	4	9

index	1	2	3	4	5	6	7	8
value	1	2	3	5	6	12	13	

Using an Array with size i

$i = 8$

index	1	2	3	4	5	6	7	8	9	10	11	12
value	3	2	1	6	13	5	12	15	11	10	4	9

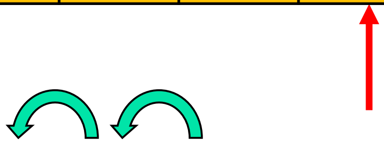
$\uparrow$

index	1	2	3	4	5	6	7	8
value	1	2	3	5	6	12	13	15

Using an Array with size i

$i = 8$

index	1	2	3	4	5	6	7	8	9	10	11	12
value	3	2	1	6	13	5	12	15	11	10	4	9



index	1	2	3	4	5	6	7	8
value	1	2	3	5	6	11	12	13

Using an Array with size i

$i = 8$

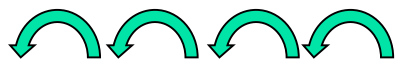
index	1	2	3	4	5	6	7	8	9	10	11	12
value	3	2	1	6	13	5	12	15	11	10	4	9

index	1	2	3	4	5	6	7	8
value	1	2	3	5	6	10	11	12

Using an Array with size i

$i = 8$

index	1	2	3	4	5	6	7	8	9	10	11	12
value	3	2	1	6	13	5	12	15	11	10	4	9



index	1	2	3	4	5	6	7	8
value	1	2	3	4	5	6	10	11

Using an Array with size i

$i = 8$

index	1	2	3	4	5	6	7	8	9	10	11	12
value	3	2	1	6	13	5	12	15	11	10	4	9

index	1	2	3	4	5	6	7	8
value	1	2	3	4	5	6	9	10

Using an Array with size i

$i = 8$

index	1	2	3	4	5	6	7	8	9	10	11	12
value	3	2	1	6	13	5	12	15	11	10	4	9

index	1	2	3	4	5	6	7	8
value	1	2	3	4	5	6	9	10

8-th smallest : 10

A Better Method Using Max-Heap

Using a Max-Heap

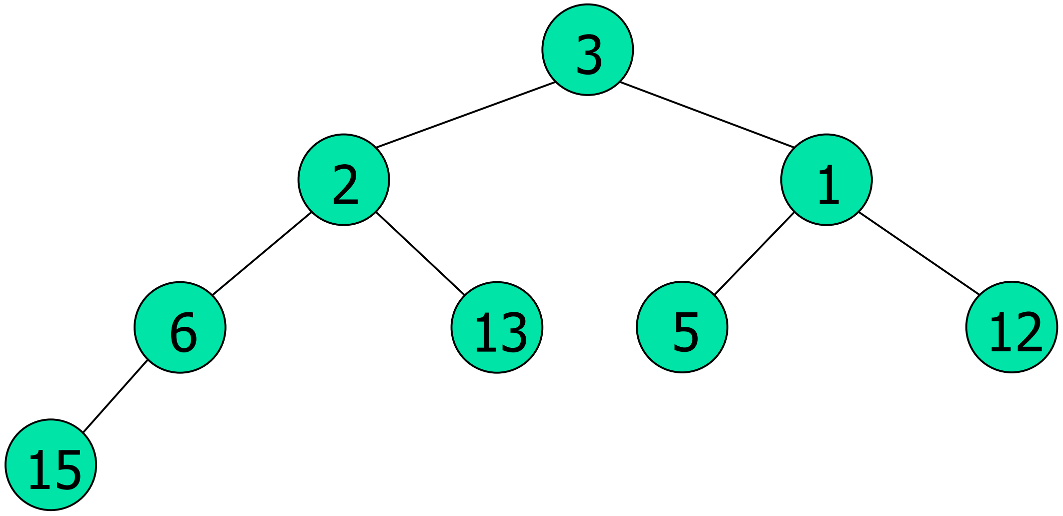
$i = 8$

index	1	2	3	4	5	6	7	8	9	10	11	12
value	3	2	1	6	13	5	12	15	11	10	4	9

Using a Max-Heap

$i = 8$

index	1	2	3	4	5	6	7	8	9	10	11	12
value	3	2	1	6	13	5	12	15	11	10	4	9

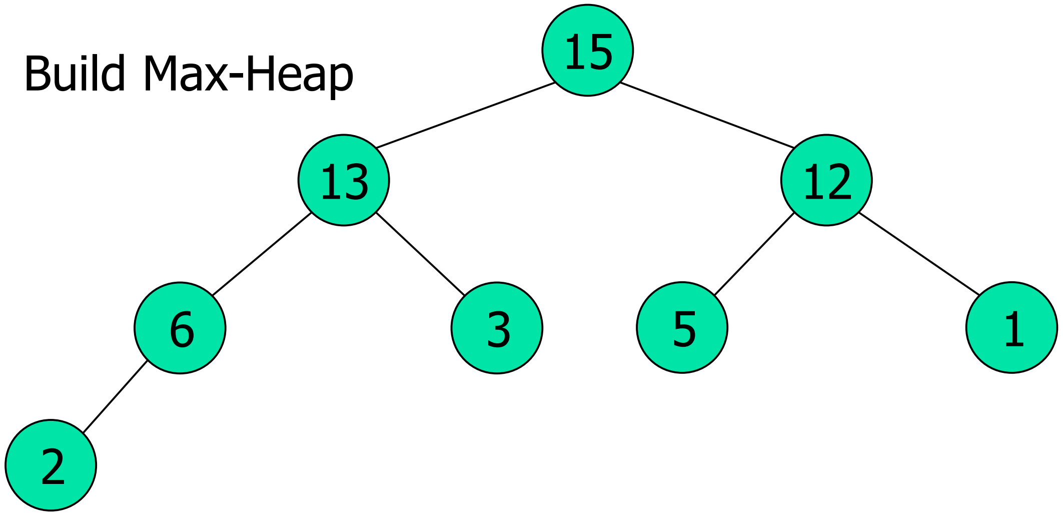


Using a Max-Heap

$i = 8$

index	1	2	3	4	5	6	7	8	9	10	11	12
value	3	2	1	6	13	5	12	15	11	10	4	9

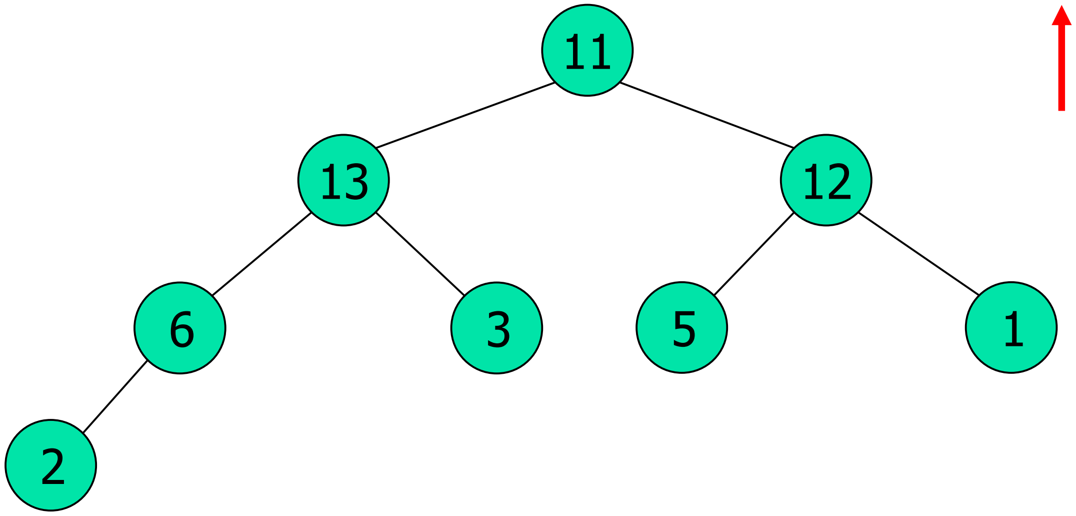
Build Max-Heap



Using a Max-Heap

$i = 8$

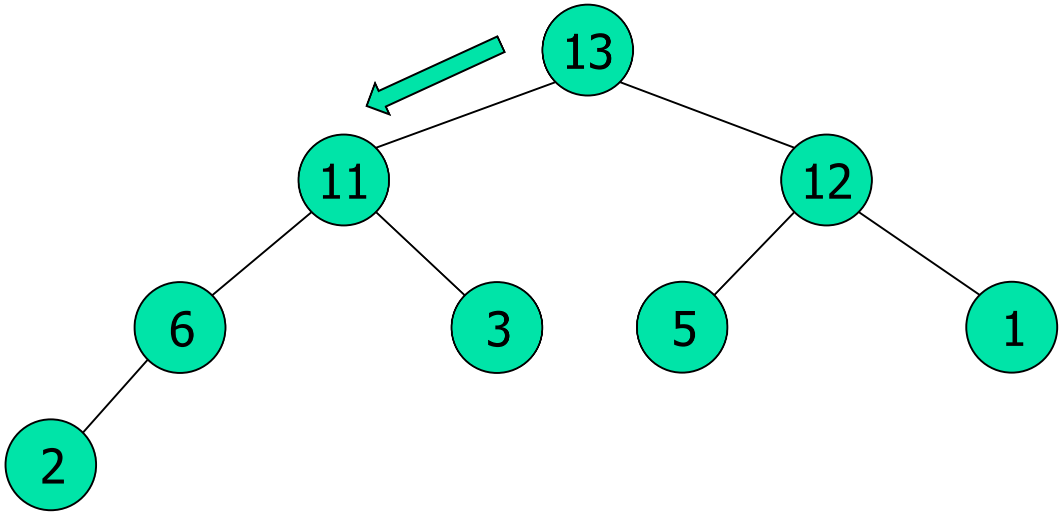
index	1	2	3	4	5	6	7	8	9	10	11	12
value	3	2	1	6	13	5	12	15	11	10	4	9



Using a Max-Heap

$i = 8$

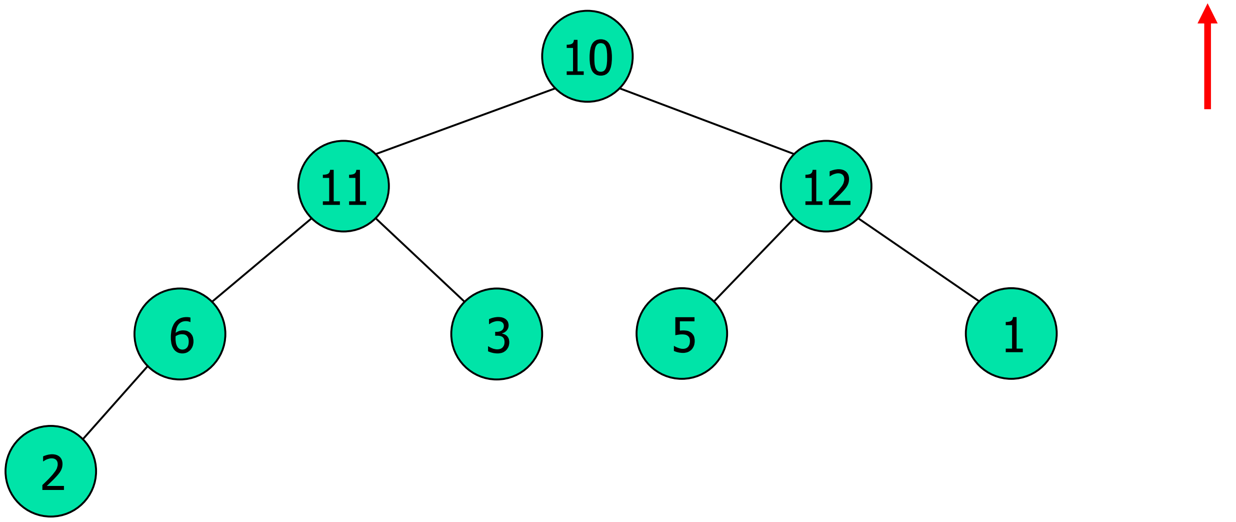
index	1	2	3	4	5	6	7	8	9	10	11	12
value	3	2	1	6	13	5	12	15	11	10	4	9



Using a Max-Heap

$i = 8$

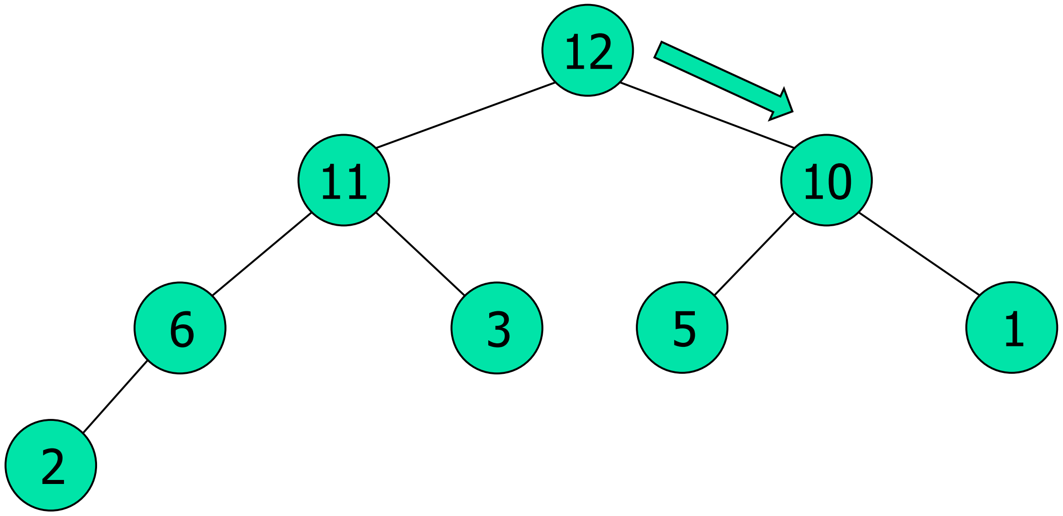
index	1	2	3	4	5	6	7	8	9	10	11	12
value	3	2	1	6	13	5	12	15	11	10	4	9



Using a Max-Heap

$i = 8$

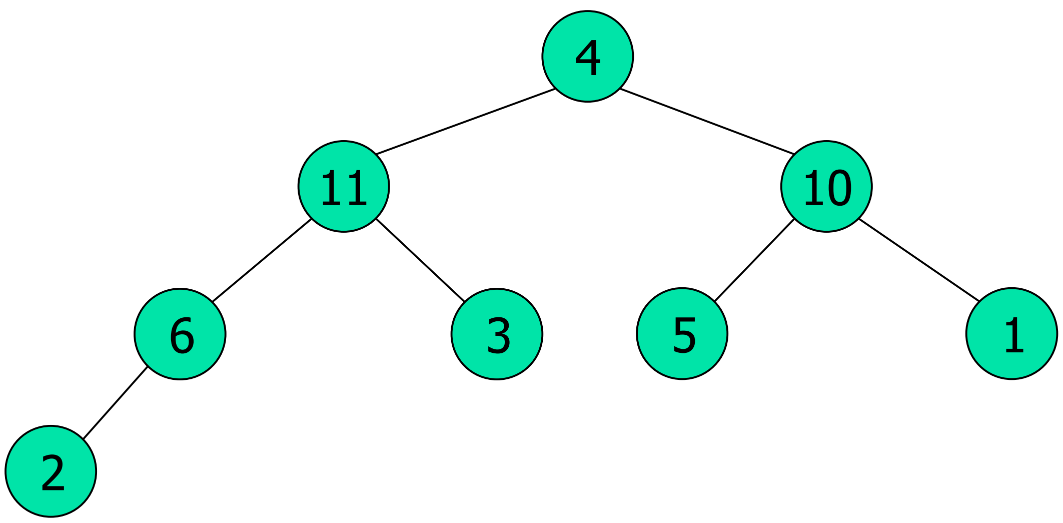
index	1	2	3	4	5	6	7	8	9	10	11	12
value	3	2	1	6	13	5	12	15	11	10	4	9



Using a Max-Heap

$i = 8$

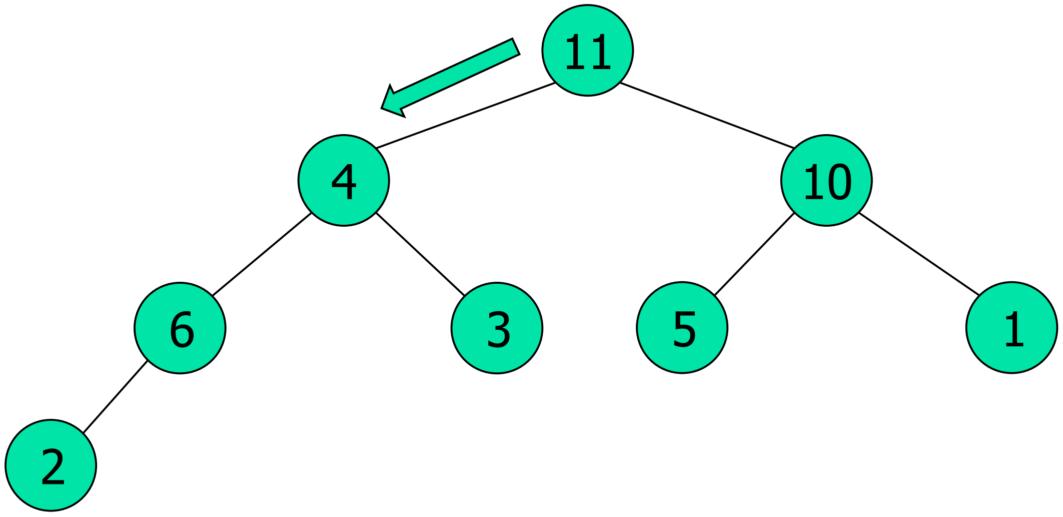
index	1	2	3	4	5	6	7	8	9	10	11	12
value	3	2	1	6	13	5	12	15	11	10	4	9



Using a Max-Heap

$i = 8$

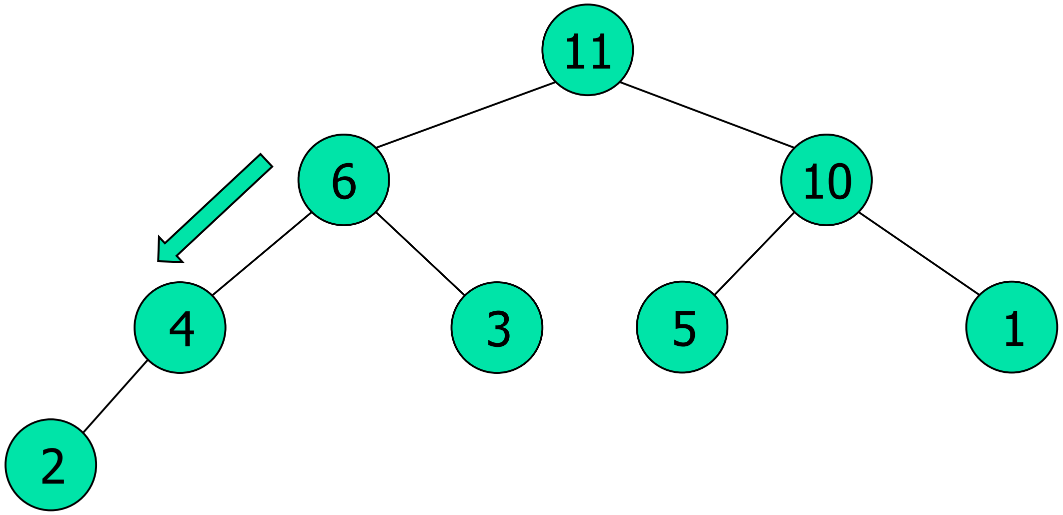
index	1	2	3	4	5	6	7	8	9	10	11	12
value	3	2	1	6	13	5	12	15	11	10	4	9



Using a Max-Heap

$i = 8$

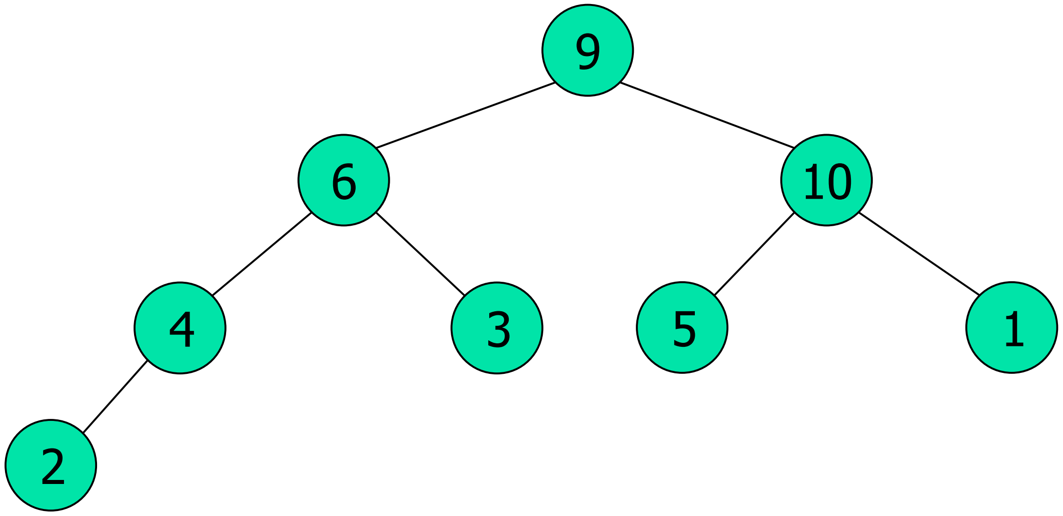
index	1	2	3	4	5	6	7	8	9	10	11	12
value	3	2	1	6	13	5	12	15	11	10	4	9



Using a Max-Heap

$i = 8$

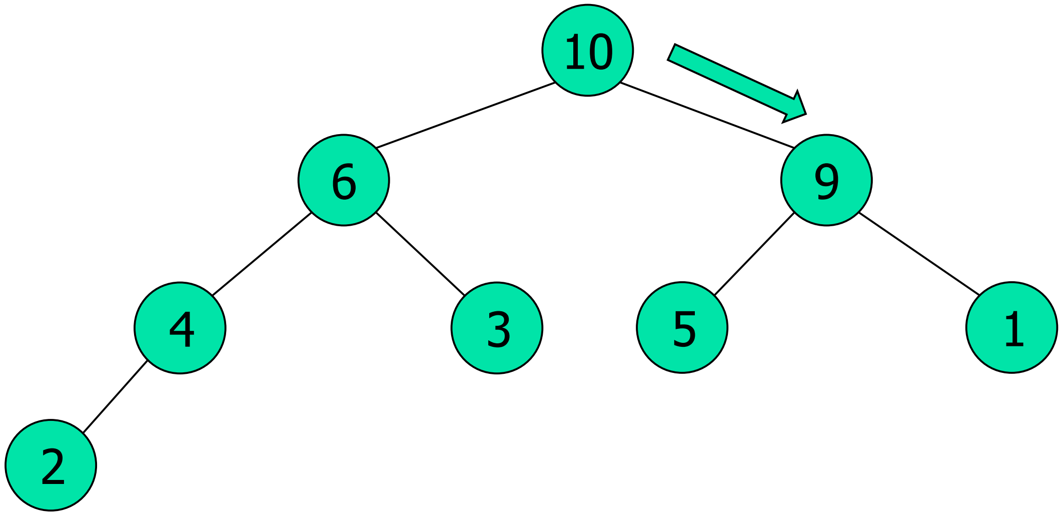
index	1	2	3	4	5	6	7	8	9	10	11	12
value	3	2	1	6	13	5	12	15	11	10	4	9



Using a Max-Heap

$i = 8$

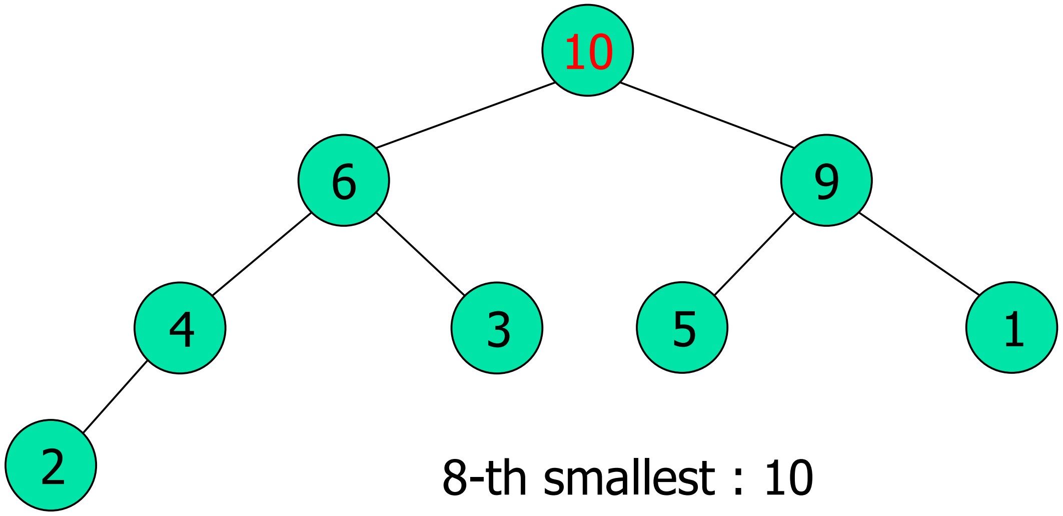
index	1	2	3	4	5	6	7	8	9	10	11	12
value	3	2	1	6	13	5	12	15	11	10	4	9



Using a Max-Heap

$i = 8$

index	1	2	3	4	5	6	7	8	9	10	11	12
value	3	2	1	6	13	5	12	15	11	10	4	9



8-th smallest : 10

A Divide and Conquer Method

Divide and Conquer

- RANDOMIZED-SELECT is modeled after quicksort algorithm.
  - It partitions the input array recursively.
  - It works on only one side of the partition.
  - While quicksort has an expected running time of $\Theta(n \lg n)$, its expected running time is $\Theta(n)$.

Divide and Conquer

- Quick Selection Algorithm
  - Pick a pivot v in S.
  - Partition S – {v} into S_1 and S_2.
  - If i = 1 + $|S_1|$, we got the answer.
  - If i < $|S_1|$, then k-th smallest element must be in S_1.
  - Otherwise, the i-th smallest element lies in S_2 and it is (i-$|S_1|$)-st smallest element in S_2.

Divide and Conquer

```
QUICK-SELECT(A,p,r,i)
1.  **if** p == r
2.      **return** A[p]
3.  q = PARTITON(A,p,r)
4.  k = q - p + 1
5.  **if** i == k
6.      **return** A[q]
7.  **else if** i < k
8.          **return** QUICK-SELECT(A,p,q-1,i)
9.  **else return** QUICK-SELECT(A,q+1,r,i-k)
```

Divide and Conquer

```
QUICK-SELECT(A,p,r,i)
    1. if p == r
    2.     return A[p]
    3. q = PARTITON(A,p,r)
    4. k = q - p + 1
    5. if i == k
    6.     return A[q]
    7. else if i < k
    8.         return QUICK-SELECT(A,p,q-1,i)
    9. else return QUICK-SELECT(A,q+1,r,i-k)
```

$i = 8$

index	1	2	3	4	5	6	7	8	9	10	11	12
value	3	2	1	6	13	5	12	15	11	10	4	9

Divide and Conquer

```
QUICK-SELECT(A,p,r,i)
1.  if p == r
2.      return A[p]
3.  q = PARTITON(A,p,r)
4.  k = q - p + 1
5.  if i == k
6.      return A[q]
7.  else if i < k
8.          return QUICK-SELECT(A,p,q-1,i)
9.  else return QUICK-SELECT(A,q+1,r,i-k)
```

$i = 8$

index	1	2	3	4	5	6	7	8	9	10	11	12
value	3	2	1	6	13	5	12	15	11	10	4	9

Divide and Conquer

```
QUICK-SELECT(A,p,r,i)
1.  if p == r
2.      return A[p]
3.  q = PARTITON(A,p,r)
4.  k = q - p + 1
5.  if i == k
6.      return A[q]
7.  else if i < k
8.          return QUICK-SELECT(A,p,q-1,i)
9.  else return QUICK-SELECT(A,q+1,r,i-k)
```

$i = 8$

index	1	2	3	4	5	6	7	8	9	10	11	12
value	3	2	1	6	13	5	12	15	11	10	4	9

↑ pivot

Divide and Conquer

```
QUICK-SELECT(A,p,r,i)
1.  if p == r
2.      return A[p]
3.  q = PARTITON(A,p,r)
4.  k = q - p + 1
5.  if i == k
6.      return A[q]
7.  else if i < k
8.         return QUICK-SELECT(A,p,q-1,i)
9.  else return QUICK-SELECT(A,q+1,r,i-k)
```

$i = 8$

index	1	2	3	4	5	6	7	8	9	10	11	12
value	3	2	1	6	4	5	9	15	11	10	13	12

Divide and Conquer

```
QUICK-SELECT(A,p,r,i)
    1.  if p == r
    2.      return A[p]
    3.  q = PARTITON(A,p,r)
    4.  k = q - p + 1
    5.  if i == k
    6.      return A[q]
    7.  else if i < k
    8.          return QUICK-SELECT(A,p,q-1,i)
    9.  else return QUICK-SELECT(A,q+1,r,i-k)
```

$i = 8$

index	1	2	3	4	5	6	7	8	9	10	11	12
value	3	2	1	6	4	5	9	15	11	10	13	12

Divide and Conquer

```
QUICK-SELECT(A,p,r,i)
  1.  if p == r
  2.      return A[p]
  3.  q = PARTITON(A,p,r)
  4.  k = q - p + 1
  5.  if i == k
  6.      return A[q]
  7.  else if i < k
  8.      return QUICK-SELECT(A,p,q-1,i)
  9.  else return QUICK-SELECT(A,q+1,r,i-k)
```

$i = 8$

index	1	2	3	4	5	6	7	8	9	10	11	12
value	3	2	1	6	4	5	9	15	11	10	13	12

$\underbrace{\qquad\qquad\qquad\qquad}_{q - p}$

q – p =6

i > q – p +1

Divide and Conquer

```
QUICK-SELECT(A,p,r,i)
  1. if p == r
  2.     return A[p]
  3. q = PARTITON(A,p,r)
  4. k = q - p + 1
  5. if i == k
  6.     return A[q]
  7. else if i < k
  8.     return QUICK-SELECT(A,p,q-1,i)
  9. else return QUICK-SELECT(A,q+1,r,i-k)
```

$i = 8$

index	1	2	3	4	5	6	7	8	9	10	11	12
value	3	2	1	6	4	5	9	15	11	10	13	12

$\underbrace{\qquad\qquad}_{q - p}$

q – p =6

i > q – p +1

Divide and Conquer

```
QUICK-SELECT(A,p,r,i)
    1.  if p == r
    2.      return A[p]
    3.  q = PARTITON(A,p,r)
    4.  k = q - p + 1
    5.  if i == k
    6.      return A[q]
    7.  else if i < k
    8.          return QUICK-SELECT(A,p,q-1,i)
    9.  else return QUICK-SELECT(A,q+1,r,i-k)
```

$i = 8$

index	1	2	3	4	5	6	7	8	9	10	11	12
value	3	2	1	6	4	5	9	15	11	10	13	12

$\underbrace{\qquad\qquad\qquad\qquad}_{q - p}$

q – p =6

i > q – p +1

Divide and Conquer

```
QUICK-SELECT(A,p,r,i)
1.  if p == r
2.      return A[p]
3.  q = PARTITON(A,p,r)
4.  k = q - p + 1
5.  if i == k
6.      return A[q]
7.  else if i < k
8.          return QUICK-SELECT(A,p,q-1,i)
9.  else return QUICK-SELECT(A,q+1,r,i-k)
```

$i = 8 - 6 - 1 = 1$

index	1	2	3	4	5	6	7	8	9	10	11	12
value	3	2	1	6	4	5	9	15	11	10	13	12

Divide and Conquer

```
QUICK-SELECT(A,p,r,i)
1. if p == r
2.     return A[p]
3. q = PARTITON(A,p,r)
4. k = q - p + 1
5. if i == k
6.     return A[q]
7. else if i < k
8.         return QUICK-SELECT(A,p,q-1,i)
9. else return QUICK-SELECT(A,q+1,r,i-k)
```

$i = 1$

index	1	2	3	4	5	6	7	8	9	10	11	12
value	3	2	1	6	4	5	9	15	11	10	13	12

Divide and Conquer

```
QUICK-SELECT(A,p,r,i)
    1.  if p == r
    2.      return A[p]
    3.  q = PARTITON(A,p,r)
    4.  k = q - p + 1
    5.  if i == k
    6.      return A[q]
    7.  else if i < k
    8.          return QUICK-SELECT(A,p,q-1,i)
    9.  else return QUICK-SELECT(A,q+1,r,i-k)
```

$i = 1$

index	1	2	3	4	5	6	7	8	9	10	11	12
value	3	2	1	6	4	5	9	15	11	10	13	12

pivot

Divide and Conquer

```
QUICK-SELECT(A,p,r,i)
1. if p == r
2.     return A[p]
3. q = PARTITON(A,p,r)
4. k = q - p + 1
5. if i == k
6.     return A[q]
7. else if i < k
8.         return QUICK-SELECT(A,p,q-1,i)
9. else return QUICK-SELECT(A,q+1,r,i-k)
```

$i = 1$

index	1	2	3	4	5	6	7	8	9	10	11	12
value	3	2	1	6	4	5	9	10	11	12	13	15

Divide and Conquer

```
QUICK-SELECT(A,p,r,i)
    1.  if p == r
    2.      return A[p]
    3.  q = PARTITON(A,p,r)
    4.  k = q - p + 1
    5.  if i == k
    6.      return A[q]
    7.  else if i < k
    8.          return QUICK-SELECT(A,p,q-1,i)
    9.  else return QUICK-SELECT(A,q+1,r,i-k)
```

$i = 1$

index	1	2	3	4	5	6	7	8	9	10	11	12
value	3	2	1	6	4	5	9	10	11	12	13	15

q - p

q - p=2

i<q – p + 1

Divide and Conquer

```
QUICK-SELECT(A,p,r,i)
1. if p == r
2.     return A[p]
3. q = PARTITON(A,p,r)
4. k = q - p + 1
5. if i == k
6.     return A[q]
7. else if i < k
8.         return QUICK-SELECT(A,p,q-1,i)
9. else return QUICK-SELECT(A,q+1,r,i-k)
```

$i = 1$

index	1	2	3	4	5	6	7	8	9	10	11	12
value	3	2	1	6	4	5	9	10	11	12	13	15

q - p

q - p=2
i<q – p + 1

Divide and Conquer

```
QUICK-SELECT(A,p,r,i)
1.  if p == r
2.      return A[p]
3.  q = PARTITON(A,p,r)
4.  k = q - p + 1
5.  if i == k
6.      return A[q]
7.  else if i < k
8.          return QUICK-SELECT(A,p,q-1,i)
9.  else return QUICK-SELECT(A,q+1,r,i-k)
```

$i = 1$

index	1	2	3	4	5	6	7	8	9	10	11	12
value	3	2	1	6	4	5	9	10	11	12	13	15

q - p=2
i<q – p + 1

q - p

Divide and Conquer

```
QUICK-SELECT(A,p,r,i)
  1. if p == r
  2.     return A[p]
  3. q = PARTITON(A,p,r)
  4. k = q - p + 1
  5. if i == k
  6.     return A[q]
  7. else if i < k
  8.         return QUICK-SELECT(A,p,q-1,i)
  9. else return QUICK-SELECT(A,q+1,r,i-k)
```

$i = 1$

index	1	2	3	4	5	6	7	8	9	10	11	12
value	3	2	1	6	4	5	9	10	11	12	13	15

Divide and Conquer

```
QUICK-SELECT(A,p,r,i)
1.  if p == r
2.      return A[p]
3.  q = PARTITON(A,p,r)
4.  k = q - p + 1
5.  if i == k
6.      return A[q]
7.  else if i < k
8.          return QUICK-SELECT(A,p,q-1,i)
9.  else return QUICK-SELECT(A,q+1,r,i-k)
```

$i = 1$

index	1	2	3	4	5	6	7	8	9	10	11	12
value	3	2	1	6	4	5	9	10	11	12	13	15

pivot

Divide and Conquer

```
QUICK-SELECT(A,p,r,i)
    1.  if p == r
    2.      return A[p]
    3.  q = PARTITON(A,p,r)
    4.  k = q - p + 1
    5.  if i == k
    6.      return A[q]
    7.  else if i < k
    8.          return QUICK-SELECT(A,p,q-1,i)
    9.  else return QUICK-SELECT(A,q+1,r,i-k)
```

$i = 1$

index	1	2	3	4	5	6	7	8	9	10	11	12
value	3	2	1	6	4	5	9	10	11	12	13	15

Divide and Conquer

```
QUICK-SELECT(A,p,r,i)
1.  if p == r
2.      return A[p]
3.  q = PARTITON(A,p,r)
4.  k = q - p + 1
5.  if i == k
6.      return A[q]
7.  else if i < k
8.          return QUICK-SELECT(A,p,q-1,i)
9.  else return QUICK-SELECT(A,q+1,r,i-k)
```

$i = 1$

index	1	2	3	4	5	6	7	8	9	10	11	12
value	3	2	1	6	4	5	9	10	11	12	13	15

$\underbrace{\quad}_{q - p}$

q – p = 1

i<q – p + 1

Divide and Conquer

```
QUICK-SELECT(A,p,r,i)
 1.  if p == r
 2.      return A[p]
 3.  q = PARTITON(A,p,r)
 4.  k = q - p + 1
 5.  if i == k
 6.      return A[q]
 7.  else if i < k
 8.          return QUICK-SELECT(A,p,q-1,i)
 9.  else return QUICK-SELECT(A,q+1,r,i-k)
```

$i = 1$

index	1	2	3	4	5	6	7	8	9	10	11	12
value	3	2	1	6	4	5	9	10	11	12	13	15

$\underbrace{\quad}_{q\ -\ p}$

$q - p = 1$

$i < q - p + 1$

Divide and Conquer

```
QUICK-SELECT(A,p,r,i)
  1.  if p == r
  2.      return A[p]
  3.  q = PARTITON(A,p,r)
  4.  k = q - p + 1
  5.  if i == k
  6.      return A[q]
  7.  else if i < k
  8.          return QUICK-SELECT(A,p,q-1,i)
  9.  else return QUICK-SELECT(A,q+1,r,i-k)
```

$i = 1$

index	1	2	3	4	5	6	7	8	9	10	11	12
value	3	2	1	6	4	5	9	10	11	12	13	15

q - p

$q - p = 1$
$i < q - p + 1$

Divide and Conquer

```
QUICK-SELECT(A,p,r,i)
1.  if p == r
2.      return A[p]
3.  q = PARTITON(A,p,r)
4.  k = q - p + 1
5.  if i == k
6.      return A[q]
7.  else if i < k
8.          return QUICK-SELECT(A,p,q-1,i)
9.  else return QUICK-SELECT(A,q+1,r,i-k)
```

$i = 1$

index	1	2	3	4	5	6	7	8	9	10	11	12
value	3	2	1	6	4	5	9	10	11	12	13	15

Divide and Conquer

```
QUICK-SELECT(A,p,r,i)
    1.  if p == r
    2.      return A[p]
    3.  q = PARTITON(A,p,r)
    4.  k = q - p + 1
    5.  if i == k
    6.      return A[q]
    7.  else if i < k
    8.          return QUICK-SELECT(A,p,q-1,i)
    9.  else return QUICK-SELECT(A,q+1,r,i-k)
```

$i = 1$

index	1	2	3	4	5	6	7	8	9	10	11	12
value	3	2	1	6	4	5	9	10	11	12	13	15

return 10

Divide and Conquer

```
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
    6.      return A[q]
    7.  else if i < k
    8.          return RANDOMIZED-SELECT(A,p,q-1,i)
    9.  else return RANDOMIZED-SELECT(A,q+1,r,i-k)
```

Analysis of RANDOMIZED-SELECT

- Worst-case running time is $\Theta(n^2)$.
- To analyze the expected running time of RANDOMIZED-SELECT, we let the running time on an input array A[p,r] of n elements be a random variable that we denote by T(n), and we obtain an upper bound on E[T(n)]
- RANDOMIZED-PARTITION is equally likely to return any element as the pivot.
- Therefore, for each k such that $1 \le k \le n$, the subarray A[p,q] has k elements (all less than or equal to the pivot) with probability 1/n.
- For k=1,2,…,n, we define indicator random variables X_k where

  X_k = I {subarray A[p..q] has exactly k elements}
- Assuming that the elements are distinct, we have $E[X_k] = 1/n$

Analysis of RANDOMIZED-SELECT

- When we call RANDOMIZED-SELECT and choose A[q] as the pivot element, we do not know, a priori, if we will terminate immediately with the correct answer, recurse on the subarray A[p..q-1], or recurse on the subarray A[q+1..r].
- To obtain an upper bound, we assume that the i-th element is always on the side of the partition with the greater number of elements.
- For a given call of RANDOMIZED-SELECT, the indicator random variable X_k has the value 1 for exactly one value of k, and it is 0 for all other k
- When X_k=1, the two subarrays on which we might recurse have sizes k-1 and n-k.
- Thus, $T(n) \leq \sum_{k=1}^{n} X_k(T\big(\max(k-1, n-k)\big) + O(n))$

  $= \sum_{k=1}^{n} X_k(T\big(\max(k-1, n-k)\big) + O(n)$

Analysis of RANDOMIZED-SELECT

- Taking expected value,

$$E[T(n)] \leq E[\sum_{k=1}^{n} X_k(T\big(\max(k-1, n-k)\big) + O(n))]$$

$$= \sum_{k=1}^{n} E[X_k(T\big(\max(k-1, n-k)\big)] + O(n)$$

$$= \sum_{k=1}^{n} E[X_k]E[T\big(\max(k-1, n-k)\big)] + O(n)$$

$$= \sum_{k=1}^{n} \frac{1}{n} \times E[T\big(\max(k-1, n-k)\big)] + O(n)$$

  - Note that X_k and T(max(k-1,n-k)) being independent random variables.

Analysis of RANDOMIZED-SELECT

- $\max(k-1, n-k) = \begin{cases} k-1 & \text{if } k \geq \left\lceil \frac{n}{2} \right\rceil \\ n-k & \text{if } k \leq \left\lceil \frac{n}{2} \right\rceil \end{cases}$
- If n is even, each term from T($\left\lceil \frac{n}{2} \right\rceil$) up to T(n-1) appears exactly twice in the summation, and if n is odd, all these terms appear twice and T($\left\lfloor \frac{n}{2} \right\rfloor$) appears once.
- Thus, we have $E[T(n)] \leq \frac{2}{n} \sum_{k=\left\lfloor \frac{n}{2} \right\rfloor}^{n-1} E[T(k)] + O(n)$

Analysis of RANDOMIZED-SELECT

- Assume that E[T(n)]≤cn for some constant c that satisfies the initial conditions of the recurrence.
- We assume that T[n]=O(1) for n less than some constant.
- Using this inductive hypothesis,

$$E[T(n)] \le \frac{2}{n}\sum_{k=\lfloor n/2 \rfloor}^{n-1} ck + an = \frac{2c}{n}\left(\sum_{k=1}^{n-1} k - \sum_{k=1}^{\lfloor n/2 \rfloor - 1} k\right) + an = \frac{2c}{n}\left(\frac{(n-1)n}{2} - \frac{(\lfloor n/2 \rfloor - 1)\lfloor n/2 \rfloor}{2}\right) + an$$

$$\le \frac{2c}{n}\left(\frac{(n-1)n}{2} - \frac{(n/2-2)(n/2-1)}{2}\right) + an = \frac{2c}{n}\left(\frac{n^2-1}{2} - \frac{n^2/4 - 3n/2 + 2}{2}\right) + an$$

$$= \frac{c}{n}\left(\frac{3n^2}{4} + \frac{n}{2} - 2\right) + an = c\left(\frac{3n}{4} + \frac{1}{2} - \frac{2}{n}\right) + an \le \frac{3cn}{4} + \frac{c}{2} + an$$

$$= cn - \left(\frac{cn}{4} - \frac{c}{2} - an\right)$$

Analysis of RANDOMIZED-SELECT

Assume $E[T(n)] \le cn$ for some constant c

$$\mathrm{E[T(n)]} \le \frac{2}{\mathrm{n}} \sum_{k=\lfloor n/2 \rfloor}^{n-1} ck + an = cn - \left(\frac{cn}{4} - \frac{c}{2} - an\right)$$

$$\text{if } T(n) = O(1) \text{ for } \frac{cn}{4} - \frac{c}{2} - an < 0, \text{ then } E[T(n)] = O(n)$$

- We need to show that for sufficient large n, cn/4-c/2-an ≥ 0.
- If we add c/2 to both sides and factor out n, we get n(c/4-a) ≥ c/2.
- As long as we choose the constant c so that (c/4-a)>0, i.e., c > 4a, we can divide both sides by c/4-a, and obtain

$$n \ge c/2/\left(\frac{c}{4} - a\right) = 2c/(c-4a).$$

- If we assume T(n) = O(1) for n < 2c/(c-4a), then E(T(n)] = O(n).

Selection in Worst-case Linear Time

- Want to develop a selection algorithm in O(n) time in worst case.
- Idea:
  - Guarantee a good split for partitioning
  - Use deterministic partitioning algorithm

Selection in Worst-case Linear Time

- The SELECT algorithm the desired element by recursively partitioning the input array.
- However, we want to guarantee a good split upon partition the array.
- SELECT uses the deterministic partitioning algorithm PARTITION from quicksort.
- But, we modify PARTITION to take the element to split around as an input parameter.

Selection in Worst-case Linear Time

- Divide the input elements into $\lceil n/5 \rceil$ groups (5 elements each)
- Find the median of each of the $\lceil n/5 \rceil$ groups by sorting.
- Use SELECT to find the find median x of the $\lceil n/5 \rceil$ medians.
- Partition the input array around the median-of-median x.
- Let k be one more than the # of elements on the left side. Thus, n-k elements in right side
- If i=k, then return x.
- Otherwise, call SELECT recursively to find
  - the i-th smallest one in the left side, if i<k,
  - the (i-k)-th smallest one in the right side, if i>k

Selection in Worst-case Linear Time

$i = 8$

index	1	2	3	4	5	6	7	8	9	10	11	12
value	3	2	1	6	13	5	12	15	11	10	4	9

Selection in Worst-case Linear Time

$i = 8$

index	1	2	3	4	5	6	7	8	9	10	11	12
value	3	2	1	6	13	5	12	15	11	10	4	9

3	2	1	6	13

5	12	15	11	10

4	9

Selection in Worst-case Linear Time

$i = 8$

index	1	2	3	4	5	6	7	8	9	10	11	12
value	3	2	1	6	13	5	12	15	11	10	4	9

3	2	1	6	13

5	12	15	11	10

4	9

3	11	4

Selection in Worst-case Linear Time

$i = 8$

index	1	2	3	4	5	6	7	8	9	10	11	12
value	3	2	1	6	13	5	12	15	11	10	4	9

3	2	1	6	13

5	12	15	11	10

4	9

3	11	4

4 : median
of medians

Selection in Worst-case Linear Time

$i = 8$

index	1	2	3	4	5	6	7	8	9	10	11	12
value	3	2	1	6	13	5	12	15	11	10	4	9

$\uparrow$ pivot

Selection in Worst-case Linear Time

$i = 8$

index	1	2	3	4	5	6	7	8	9	10	11	12
value	3	2	1	4	6	13	5	12	15	11	10	9

partition

Selection in Worst-case Linear Time

$i = 8$

index	1	2	3	4	5	6	7	8	9	10	11	12
value	3	2	1	4	6	13	5	12	15	11	10	9

$p = 1$ $\quad$ $q = 4$

$$q - p + 1 < i$$

Selection in Worst-case Linear Time

$i = 8 - (4 - 1 + 1) = 4$

index	1	2	3	4	5	6	7	8	9	10	11	12
value	3	2	1	4	6	13	5	12	15	11	10	9

Selection in Worst-case Linear Time

$i = 4$

index	1	2	3	4	5	6	7	8	9	10	11	12
value	3	2	1	4	6	13	5	12	15	11	10	9

6	13	5	12	15

11	10	9

Selection in Worst-case Linear Time

$i = 4$

index	1	2	3	4	5	6	7	8	9	10	11	12
value	3	2	1	4	6	13	5	12	15	11	10	9

6	13	5	12	15

11	10	9

Selection in Worst-case Linear Time

$i = 4$

index	1	2	3	4	5	6	7	8	9	10	11	12
value	3	2	1	4	6	13	5	12	15	11	10	9

6	13	5	12	15

11	10	9

median of medians : 10

Selection in Worst-case Linear Time

$i = 4$

index	1	2	3	4	5	6	7	8	9	10	11	12
value	3	2	1	4	6	13	5	12	15	11	10	9

pivot

Selection in Worst-case Linear Time

$i = 4$

index	1	2	3	4	5	6	7	8	9	10	11	12
value	3	2	1	4	6	5	9	10	15	11	13	12

partition

Selection in Worst-case Linear Time

$i = 4$

index	1	2	3	4	5	6	7	8	9	10	11	12
value	3	2	1	4	6	5	9	10	15	11	13	12

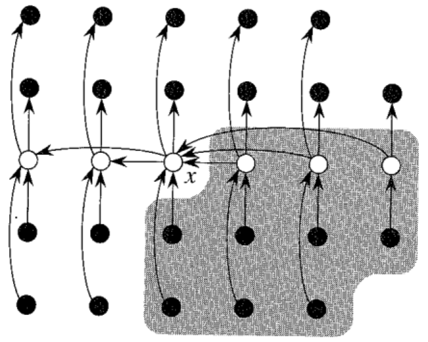
$p = 5$ $q = 8$

$q - p + 1 == i$

$\therefore$ return $A[q] = 10$

Analysis of SELECT

- The n elements are represented by small circles, and each group of 5 elements occupies a column.
- The medians of the groups are whitened, and the median-of-medians x is labeled.
- Arrows go from larger elements to smaller, from which we can see that 3 out of every full group of 5 elements to the right of x are greater than x, and 3 out of every group of 5 elements to the left of x.
- The elements known to be greater than x appear on a shaded background.



Selection in Worst-case Linear Time

- We assume that every numbers are distinct!
- We want to determine a lower bound on the number of elements that are greater than the partitioning element the median-of-median x.
- At least half of the medians found in the 2-nd step are greater than or equal to the median-of-median x.
- Thus, at least half of the $\lceil n/5 \rceil$ groups contribute at least 3 elements that are greater than x except for the last one group and the one group containing x itself
- Discounting these two groups, we obtain

$$3\left(\left\lceil \frac{1}{2}\left\lceil \frac{n}{5} \right\rceil \right\rceil - 2\right) \geq \frac{3n}{10} - 6$$

Selection in Worst-case Linear Time

- We assume that every numbers are distinct!
- The number of elements greater than x is at least

$$3\left(\left\lceil \frac{1}{2}\left\lceil \frac{n}{5} \right\rceil \right\rceil - 2\right) \geq \frac{3n}{10} - 6$$

- Similarly, the number of elements less than x is the same as above.
- Thus, in worst case, SELECT is called recursively on at most 7n/10+6 elements in the last step.

Linear Time Complexity

$$T(n) \le \begin{cases} O(1) & \text{if } n < 140 \\ T(\lceil n/5 \rceil) + T(7n/10+6) + O(n) & \text{if } n \ge 140 \end{cases}$$

Assume $T(n) \le cn$ for some suitably large constant c and all $n \ge 140$

Proof :

$$T(n) \le c\lceil n/5 \rceil + c(7n/10+6) + an$$
$$\le cn/5 + c + 7cn/10 + 6c + an$$
$$= 9cn/10 + 7c + an = cn + (-cn/10 + 7c + an)$$

which is at most if

$$-cn/10 + 7c + an \le 0 \;\rightarrow\; c \ge 10a(n/(n-70)) \text{ when } n > 70$$
$$n \ge 140 \;\rightarrow\; n/(n-70) \le 2 \;\rightarrow\; c \ge 20a$$

Any Question?