Quiz

- Insert 14, 6, 16, 22, and 27 to a hash table of size **m** = 7 using $h(k) = (k\%m + i*h2(k)) \% m$, where $h2(k) = k\%(m - 1)$

a[0]	
a[1]	
a[2]	
a[3]	
a[4]	
a[5]	
a[6]	

Quiz

- Insert 14, 6, 16, 22, and 27 to a hash table of size **m** = 7 using h(k) = (k%m + i*h2(k)) % m, where h2(k) = k%(m – 1)

a[0]	14
a[1]	22
a[2]	16
a[3]	
a[4]	
a[5]	
a[6]	6

Quiz

- Insert 14, 6, 16, 22, and 27 to a hash table of size **m** = 7 using h(k) = (k%m + i*h2(k)) % m, where h2(k) = k%(m – 1)

a[0]	14
a[1]	22
a[2]	16
a[3]	
a[4]	
a[5]	27
a[6]	6

h(k) = (27%7 + 0*27%(7-1)) % 7 = 6

h(k) = (6 + 1*3) % 7 = 2

h(k) = (6 + 2*3) % 7 = 5

8. Sorting

Bubble Sort

- the simplest sorting algorithm devised
- objects to be sorted are proportional to their weights and are kept in a tube(with water in it) and held vertically upward.
- several passes are made and for each pass the lightest is **bubbled up** to the top

Bubble Sort

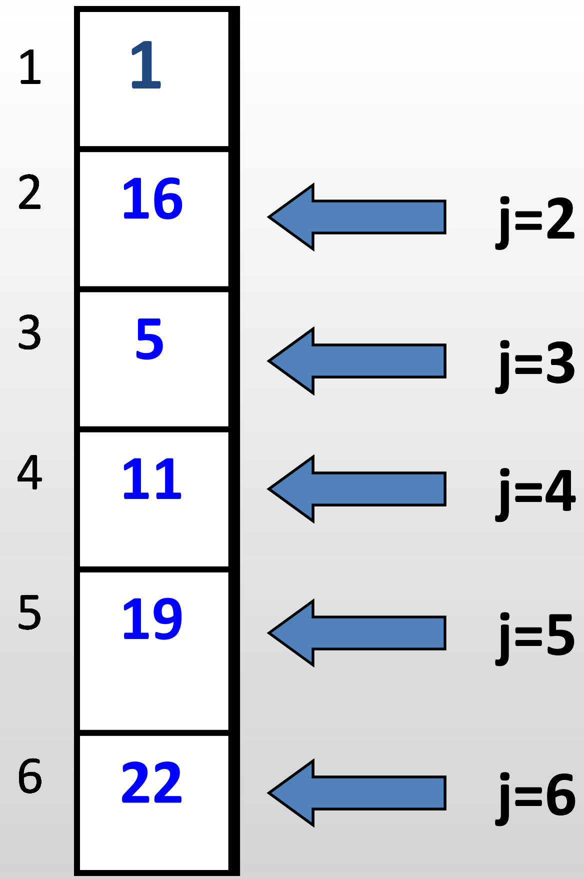
```
BubbleSort(A,n)
 begin
   for i = 1 to n-1 do
         for j = n downto i+1 do
                  if A[j]<A[j-1]
                           swap(A[j],A[j-1])
end
```

Sort :16, 5, 11, 1, 22, 19

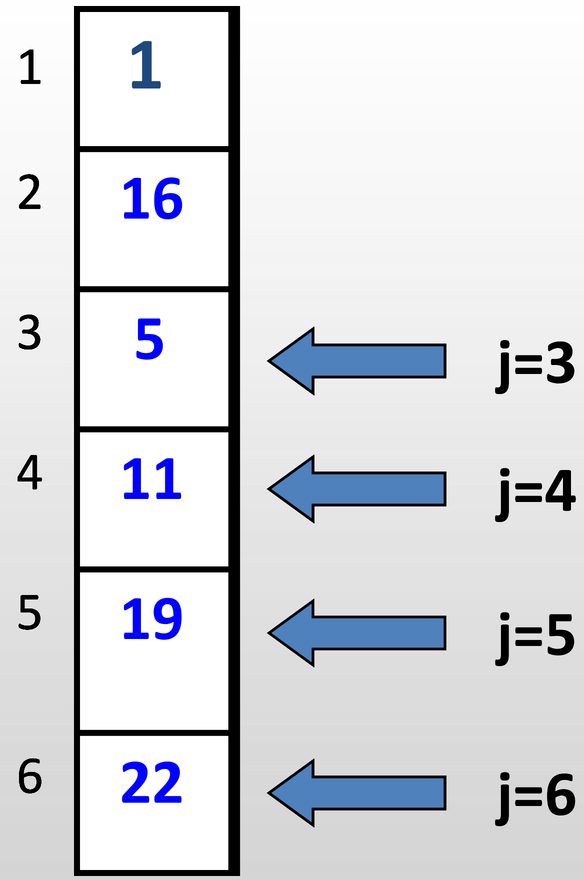
initial configuration

1	16
2	5
3	11
4	1
5	22
6	19

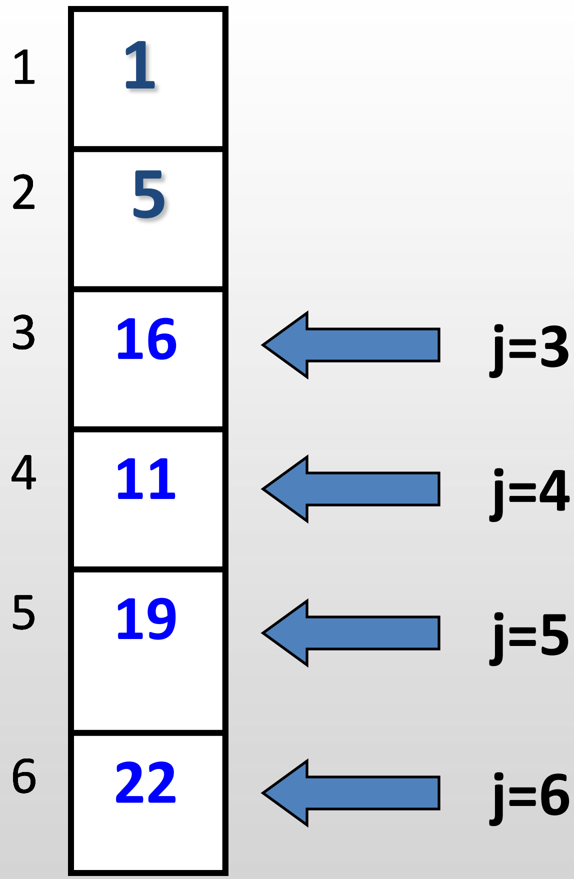
1st pass: i = 1, j={6,5,4,3,2}



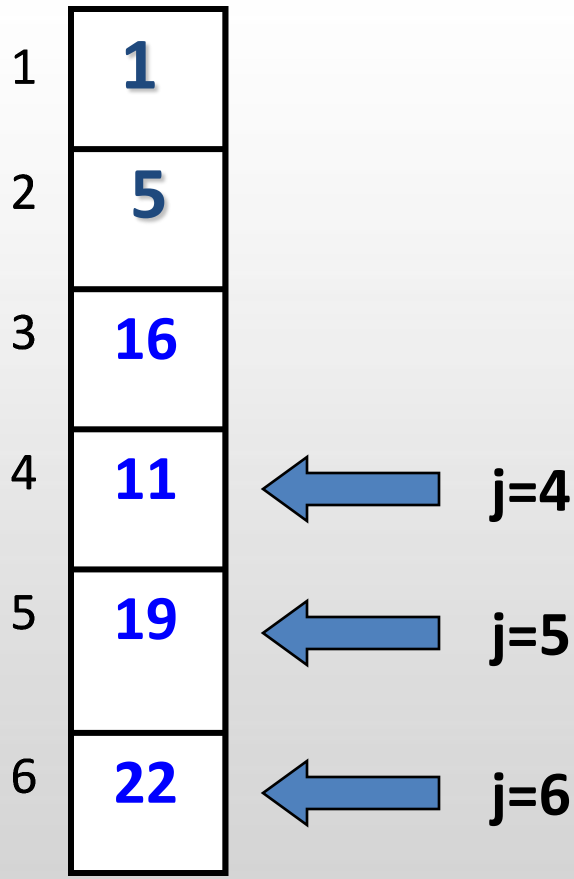
2nd pass: i=2, j={6,5,4,3}



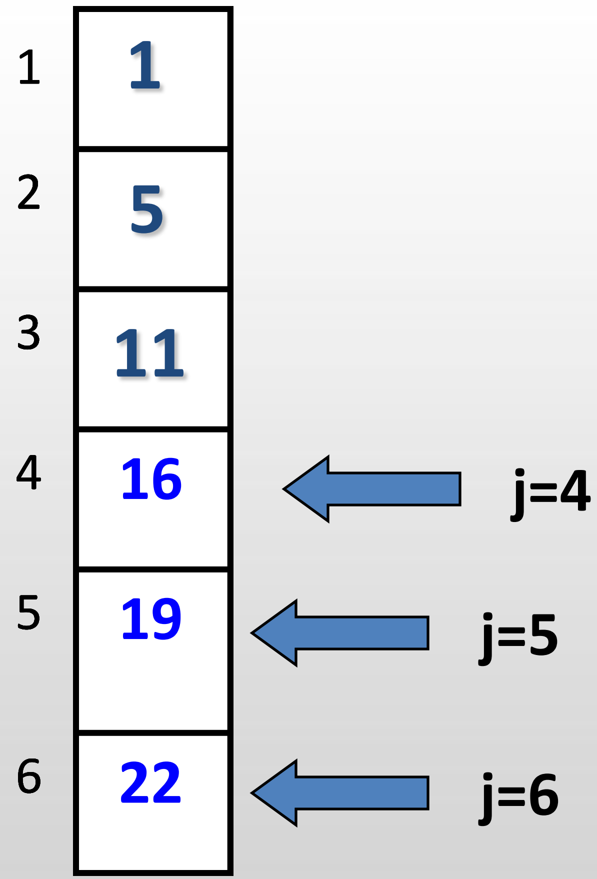
2nd pass: i=2, j={6,5,4,3}



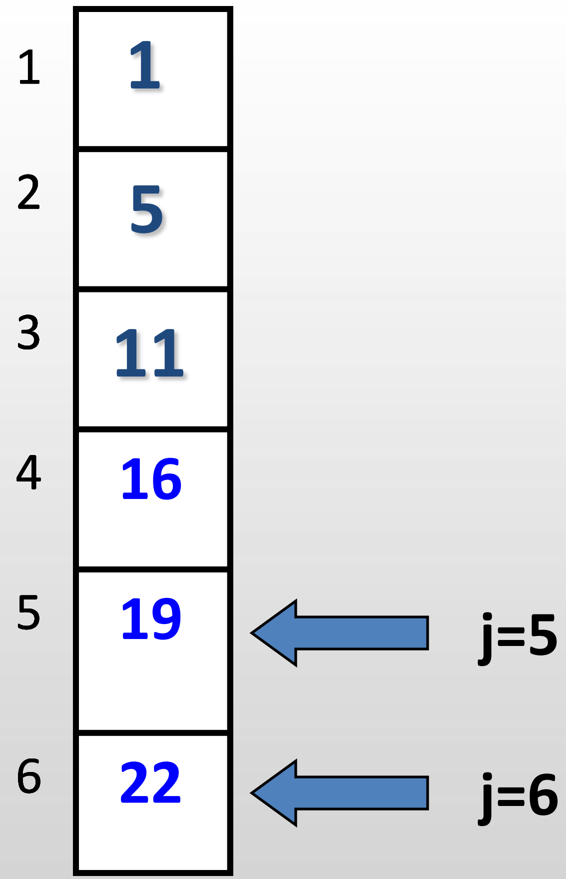
3rd pass: i=3, j={6,5,4}



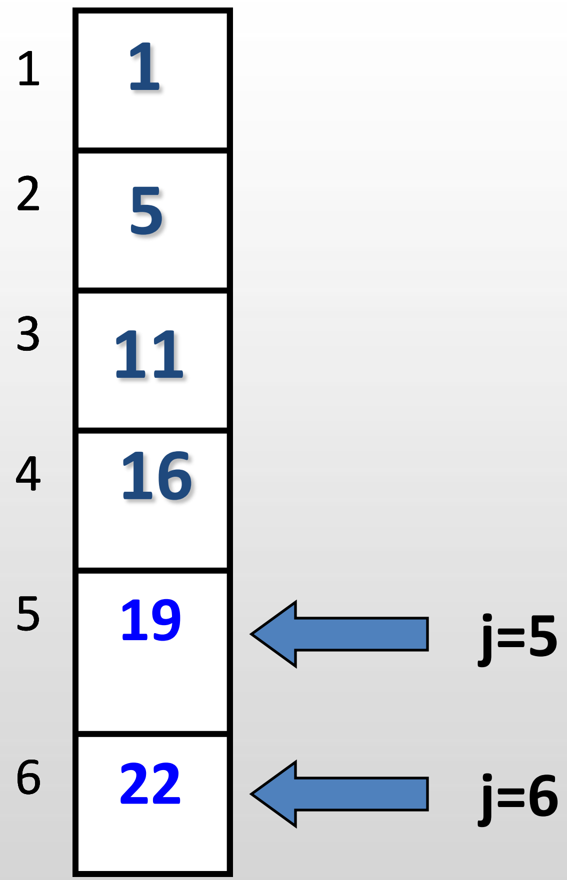
3rd pass: i=3, j={6,5,4}



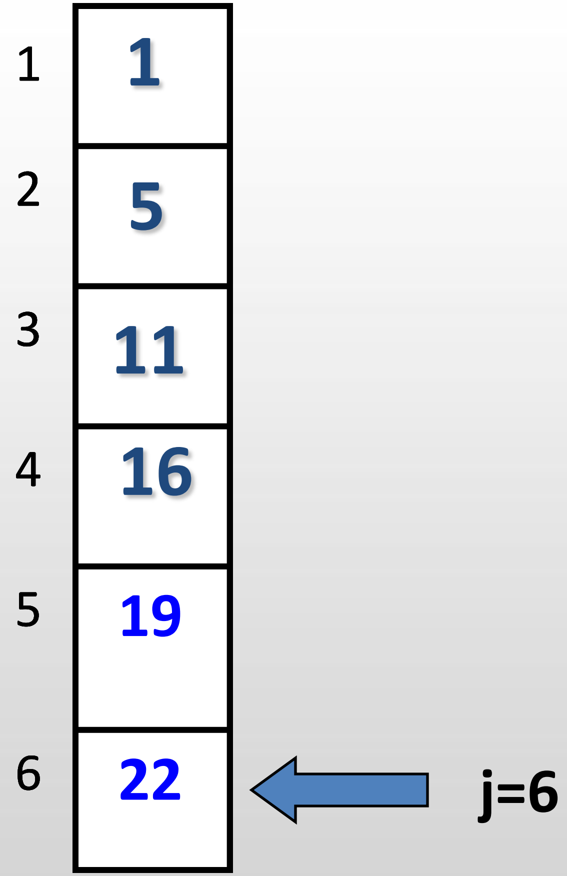
4th pass: i=4, j={6,5}



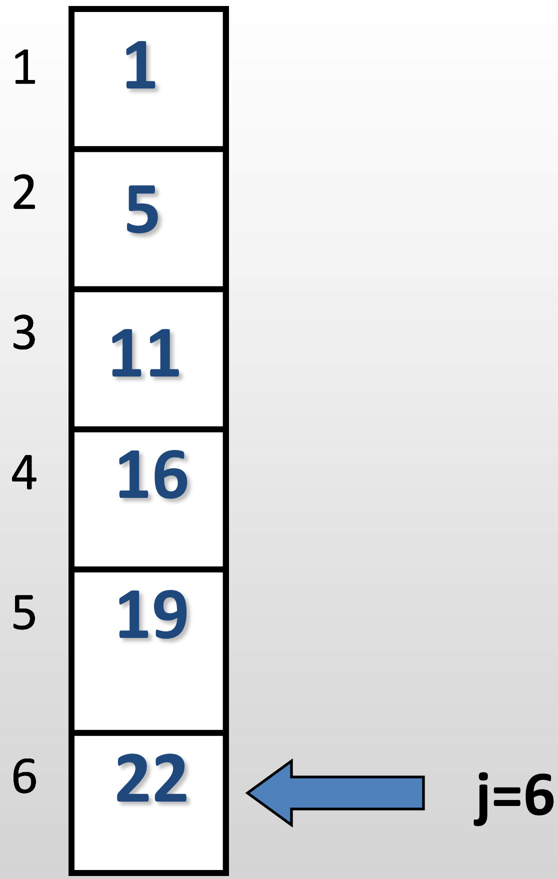
4th pass: i=4, j={6,5}



5th pass: i=5, j={6}



5th pass: i=5, j={6}



Insertion Sort

- this is how most of us logically sort objects
- the fundamental method is to initially sort two observations of the data set, take another observation and insert it in its correct position, repeat the process until the last element has been inserted

Insertion Sort

for i = 2 to n do

insert the ith observation in its correct position between the first and the ith observation

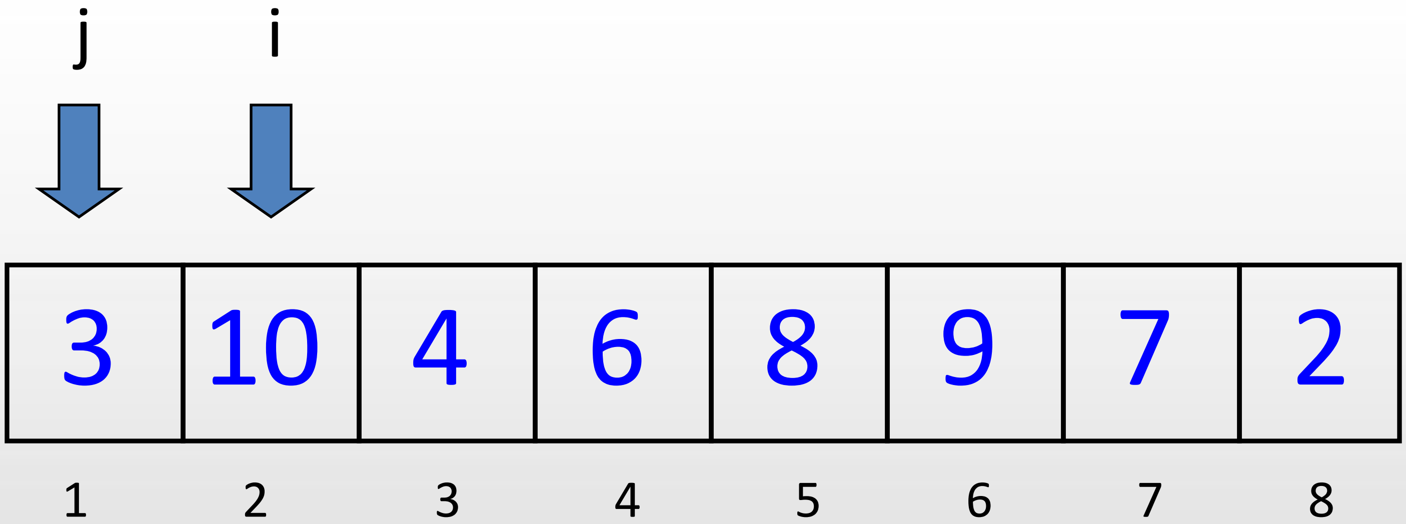
Insertion Sort

```
InsertionSort(n)
 begin
   for i = 2 to n do
   begin
        obs = A[i]
        j = i-1
        while(obs < A[j]) and (j>0) do
        begin
              A[j+1] = A[j]
              j = j – 1
        end
        A[j+1] = obs
   end
 end
```

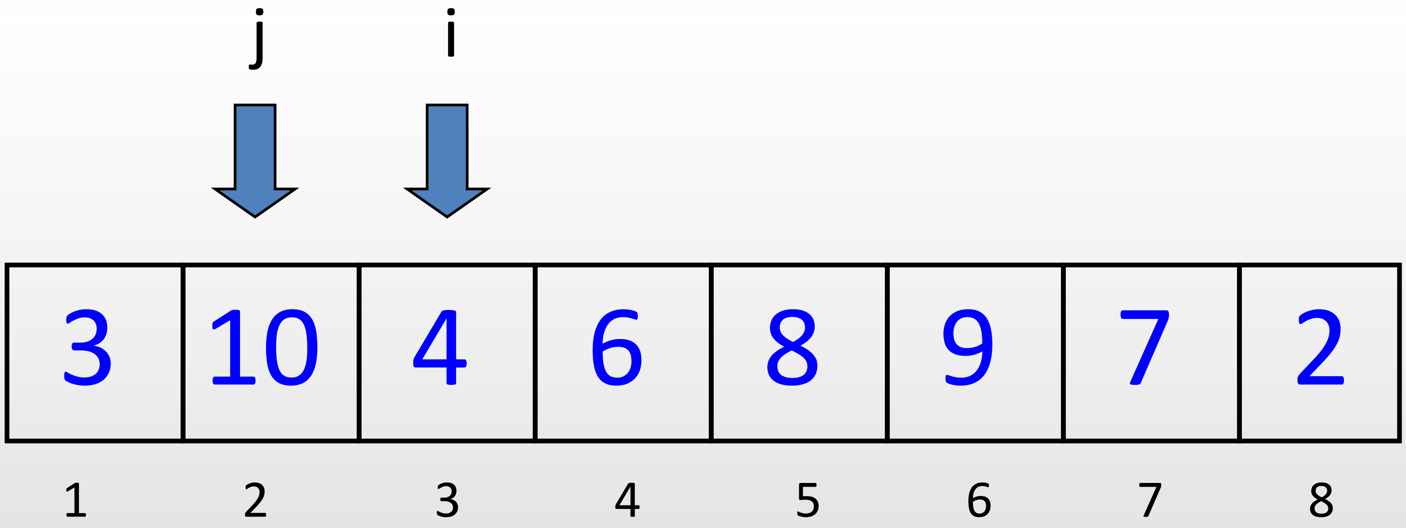
Sort 3, 10, 4, 6, 8, 9, 7, 2 using insertion sort

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

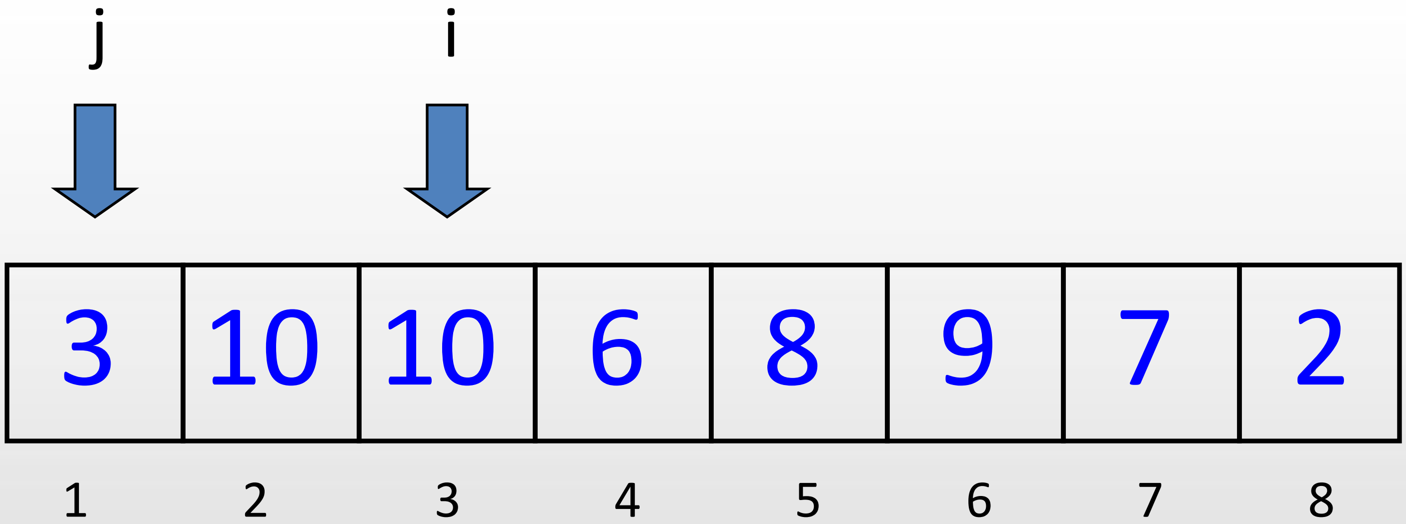
1st pass: i=2, obs=10
j = {1}



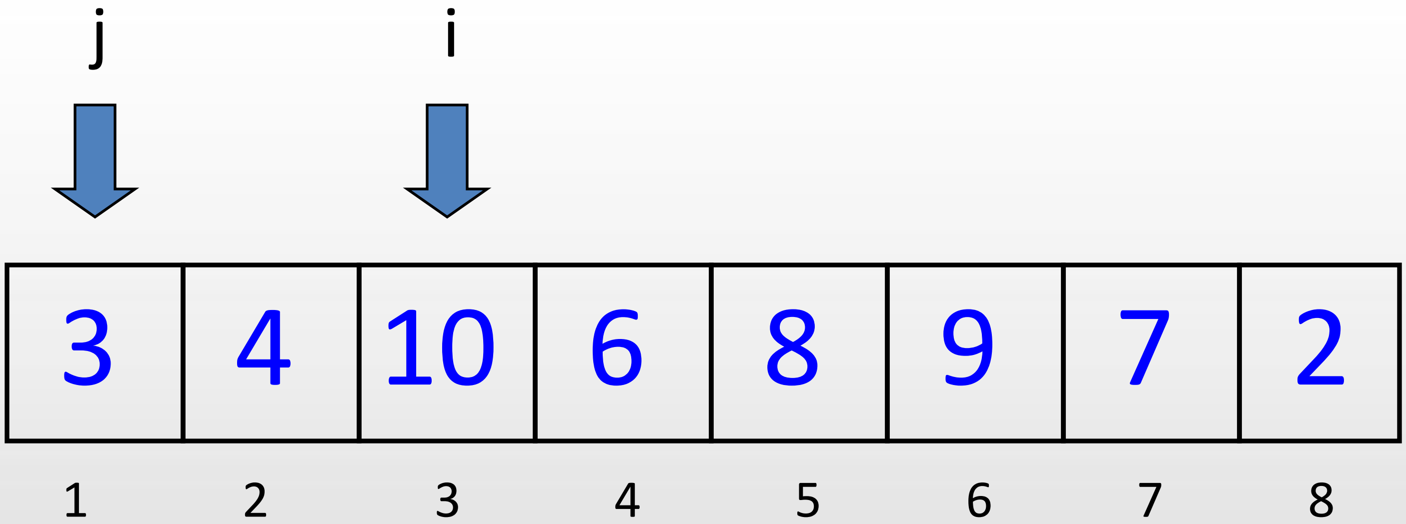
1st pass: i=3, obs=4
j = {1,2}



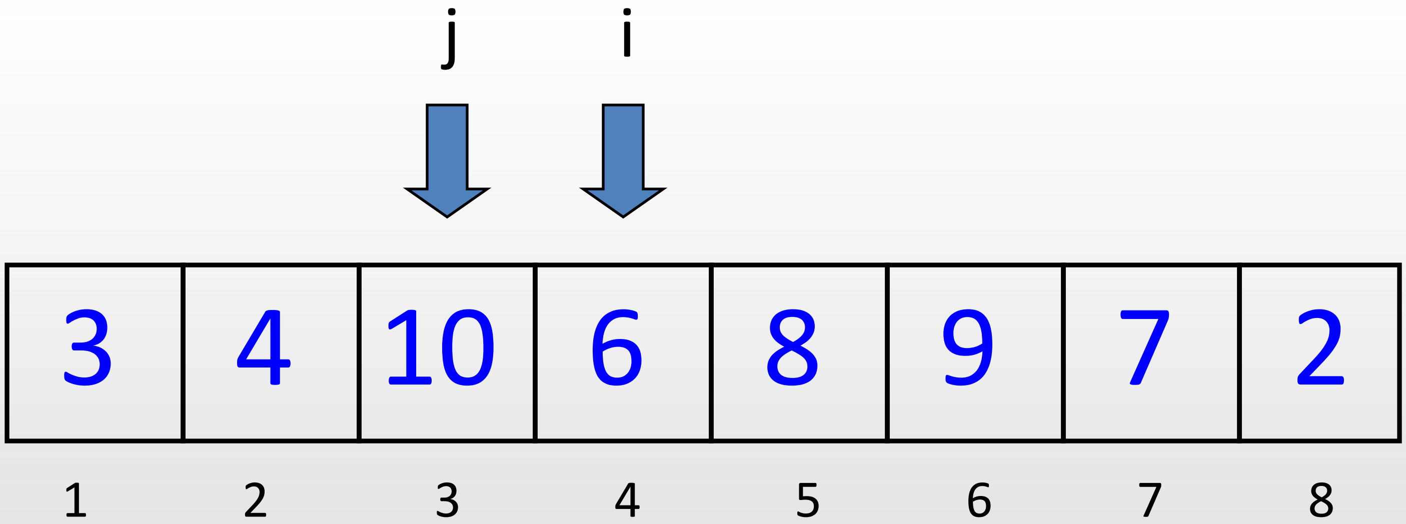
1st pass: i=3, obs=4
j = {1,2}



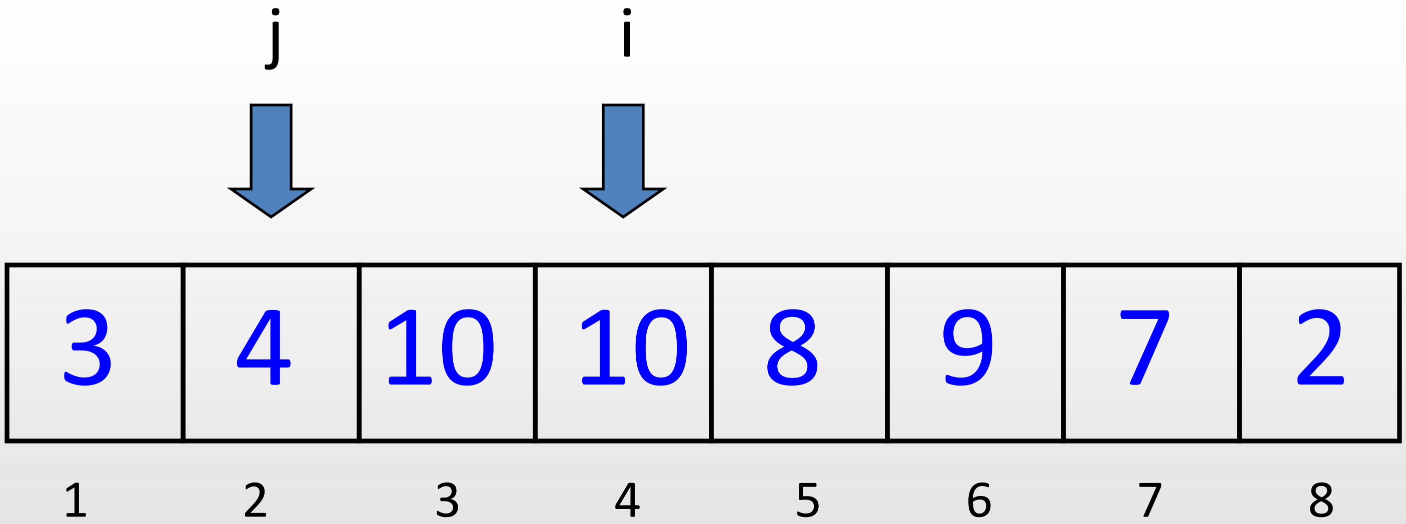
1st pass: i=3, obs=4
j = {1,2}



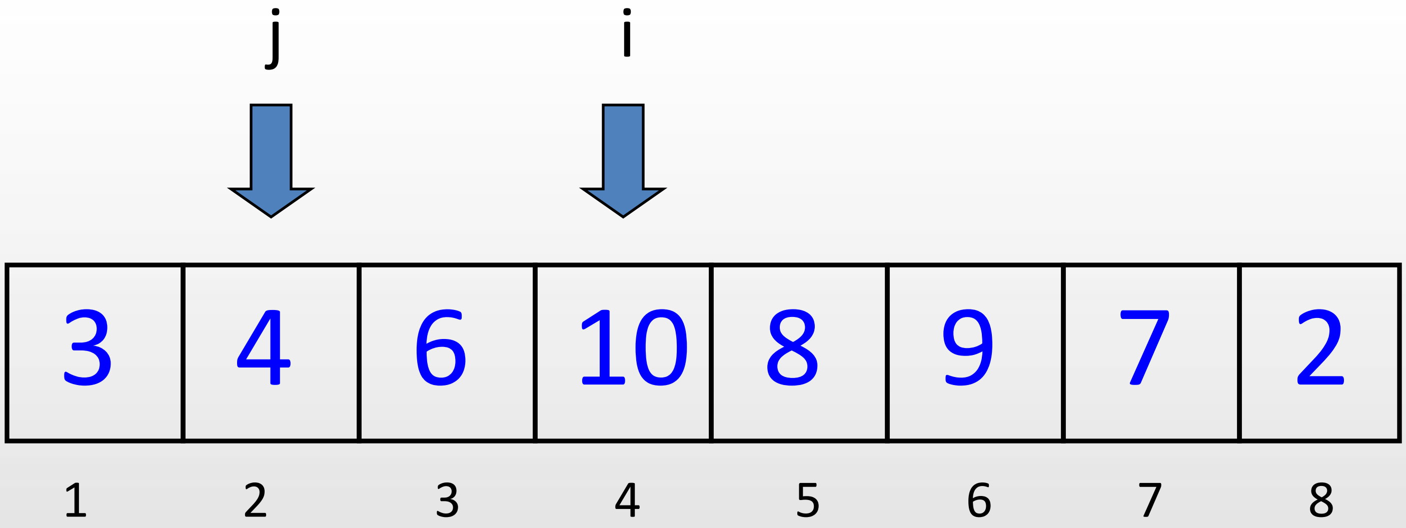
1st pass: i=4, obs=6
j = {1,2,3}



1st pass: i=4, obs=6
j = {1,2,3}



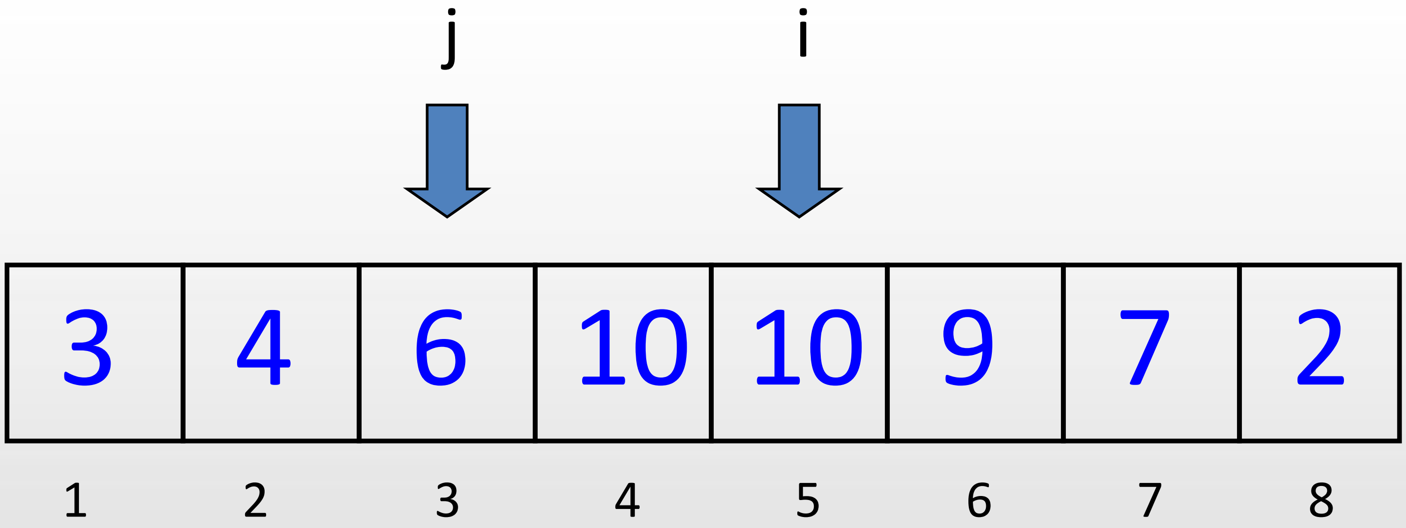
1st pass: i=4, obs=6
j = {1,2,3}



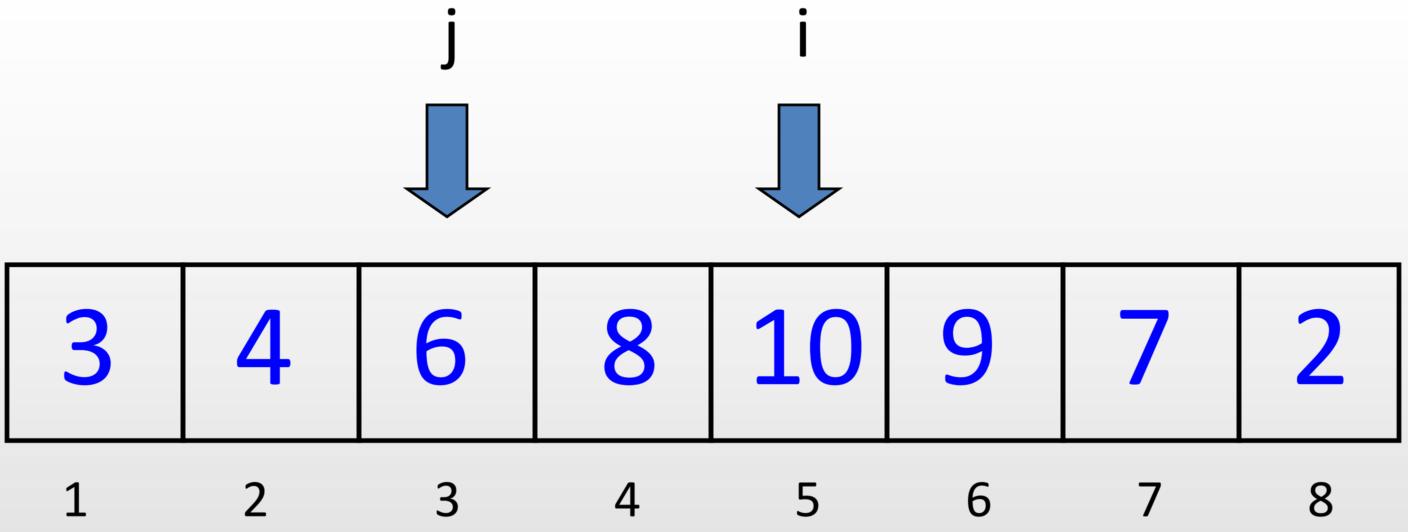
1st pass: i=5, obs=8
j = {1,2,3,4}

			j	i			
3	4	6	10	8	9	7	2
1	2	3	4	5	6	7	8

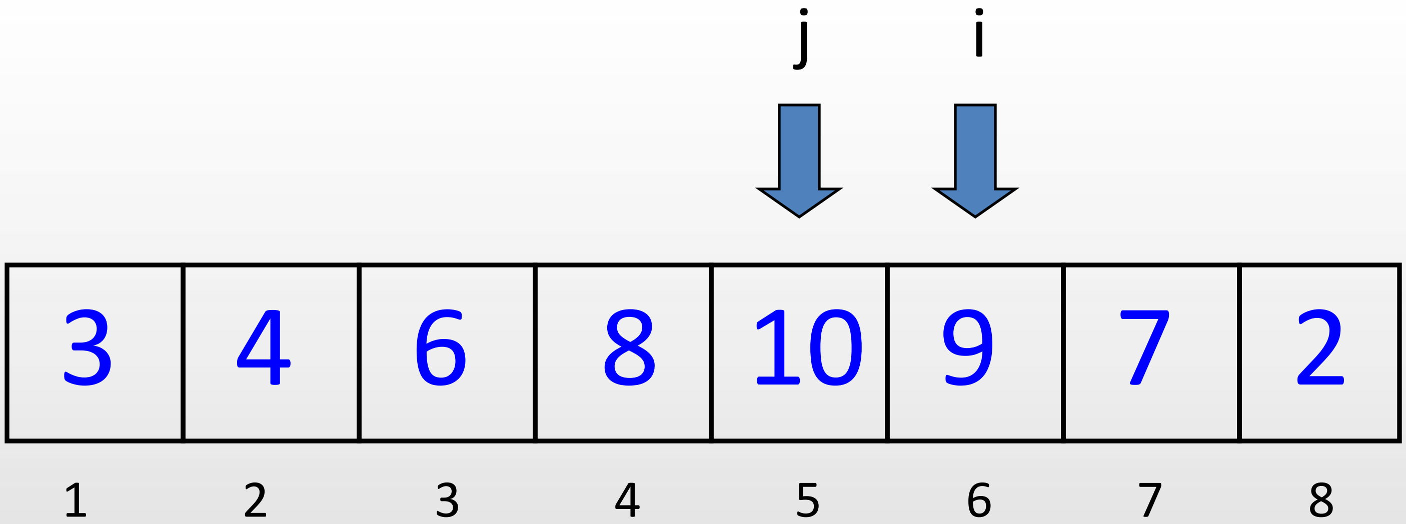
1st pass: i=5, obs=8
j = {1,2,3,4}



1st pass: i=5, obs=8
j = {1,2,3,4}



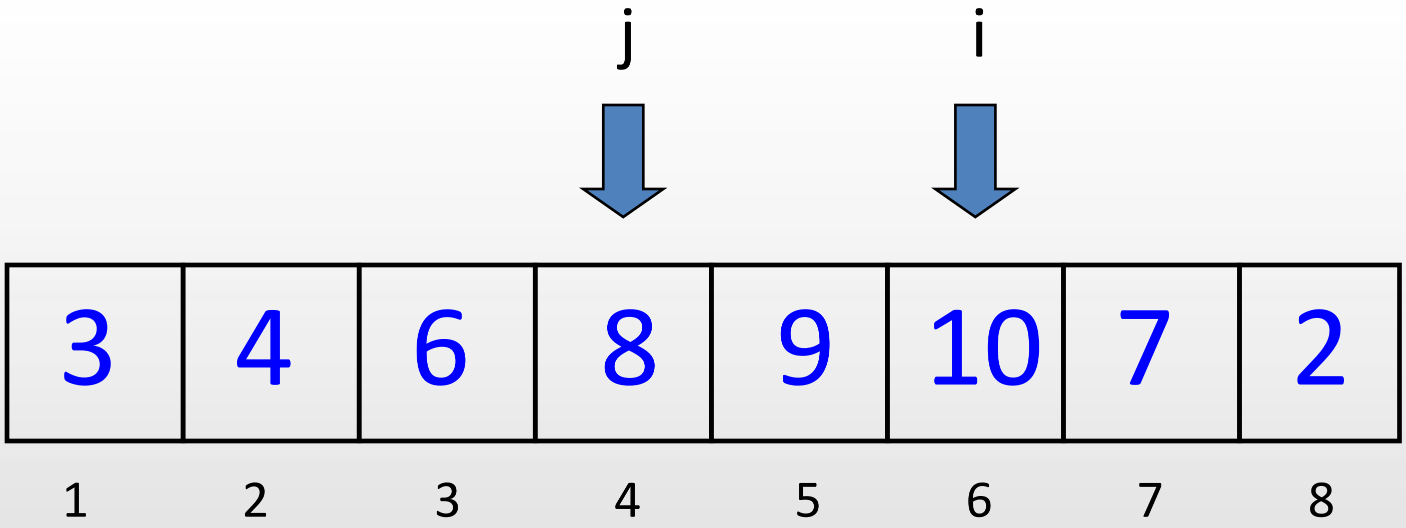
1st pass: i=6, obs=9
j = {1,2,3,4,5}



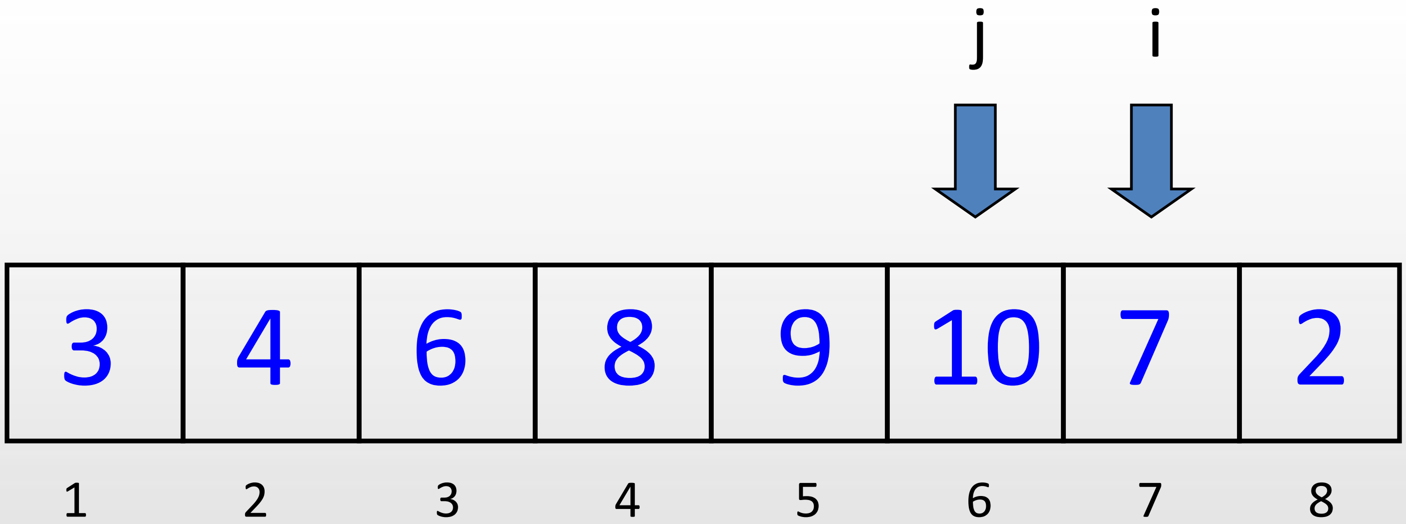
1st pass: i=6, obs=9
j = {1,2,3,4,5}

			j		i		
3	4	6	8	10	10	7	2
1	2	3	4	5	6	7	8

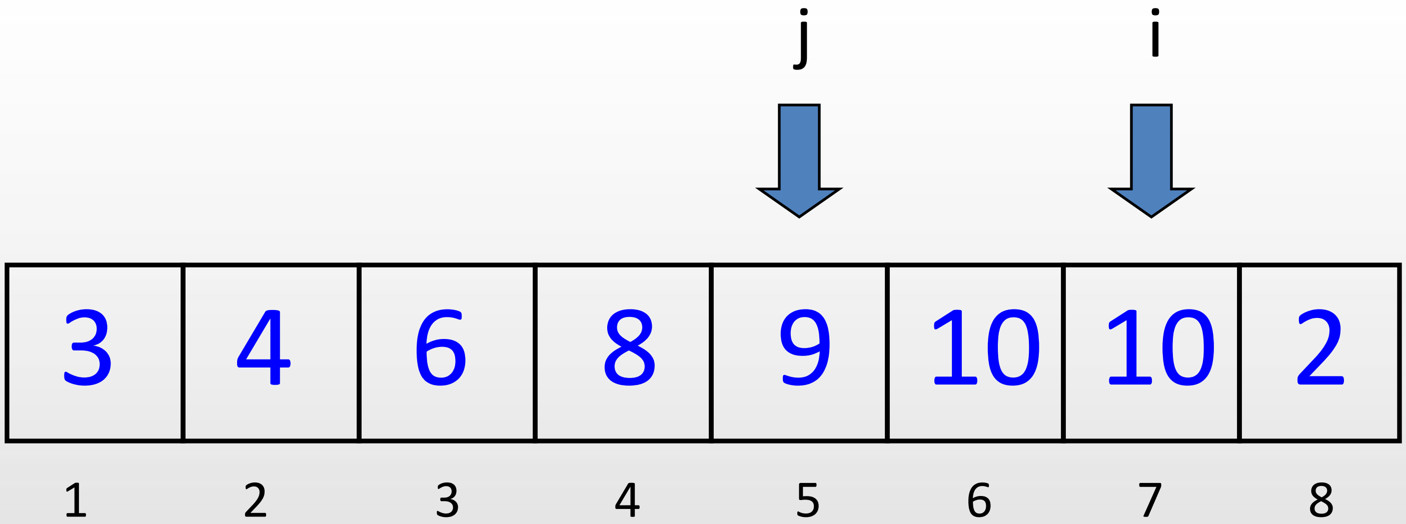
1st pass: i=6, obs=9
j = {1,2,3,4,5}



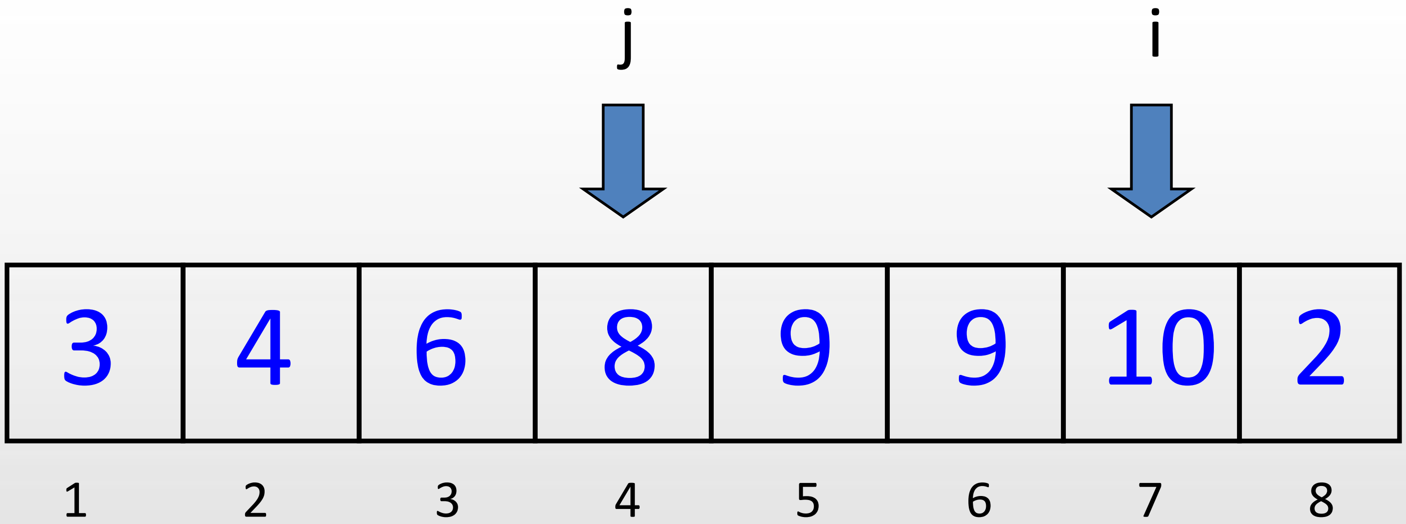
1st pass: i=7, obs=7
j = {1,2,3,4,5,6}



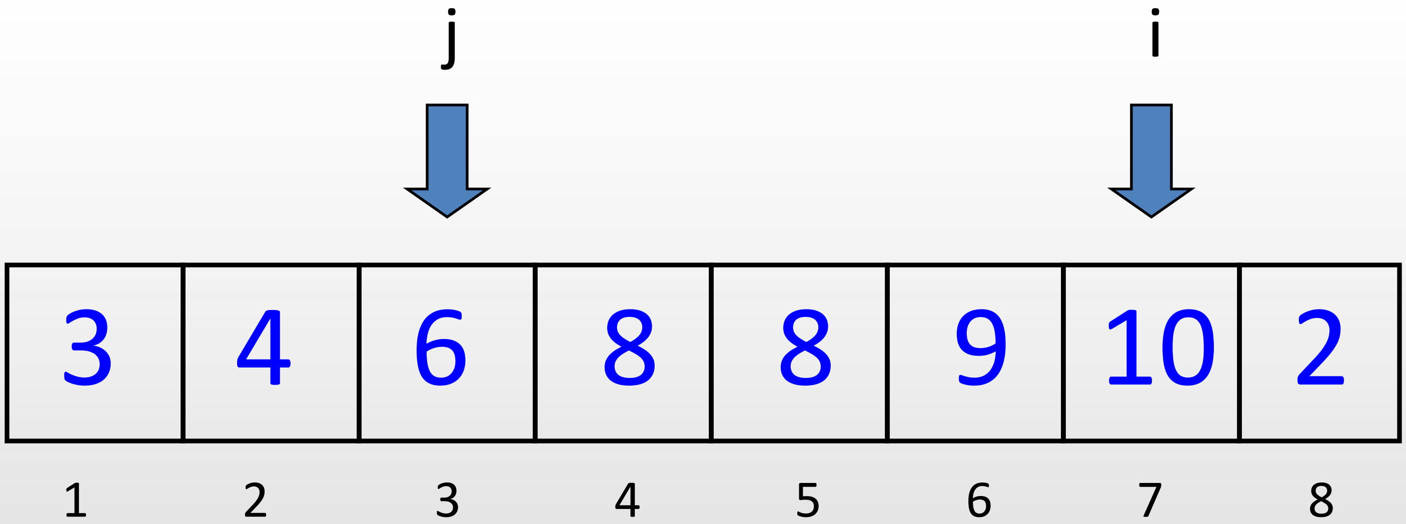
1st pass: i=7, obs=7
j = {1,2,3,4,5,6}



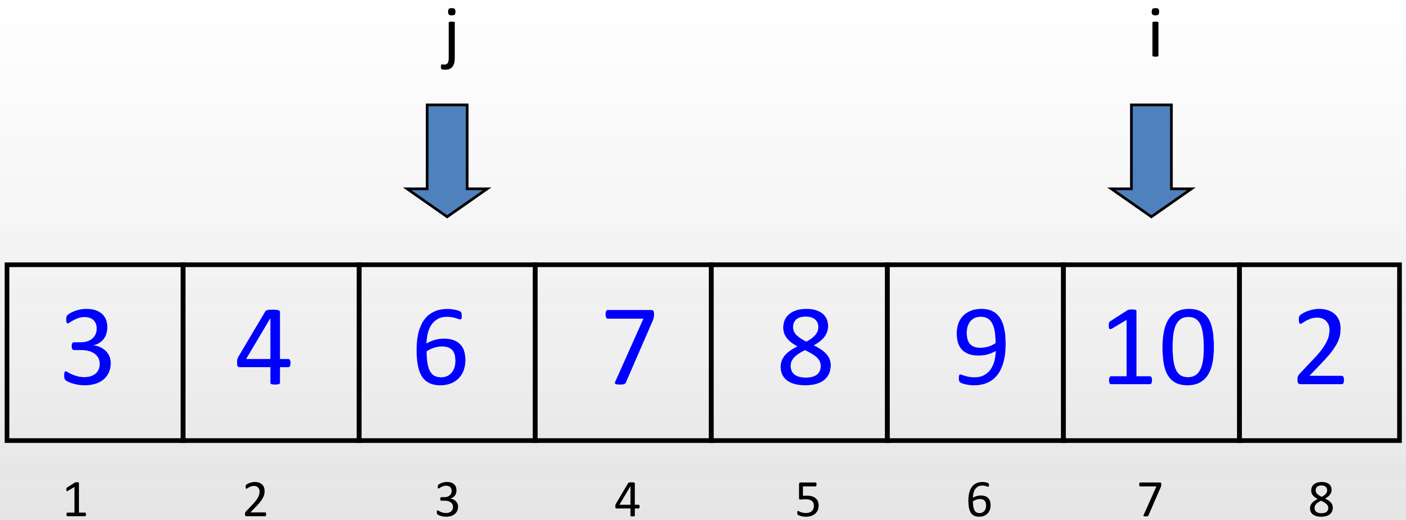
1st pass: i=7, obs=7
j = {1,2,3,4,5,6}



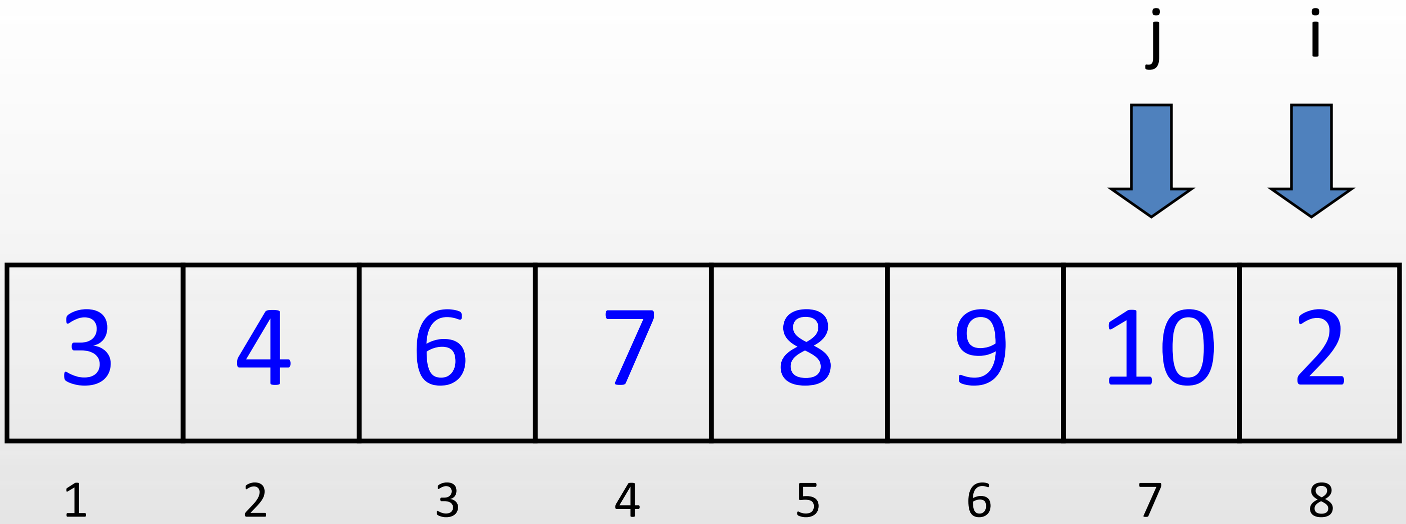
1st pass: i=7, obs=7
j = {1,2,3,4,5,6}



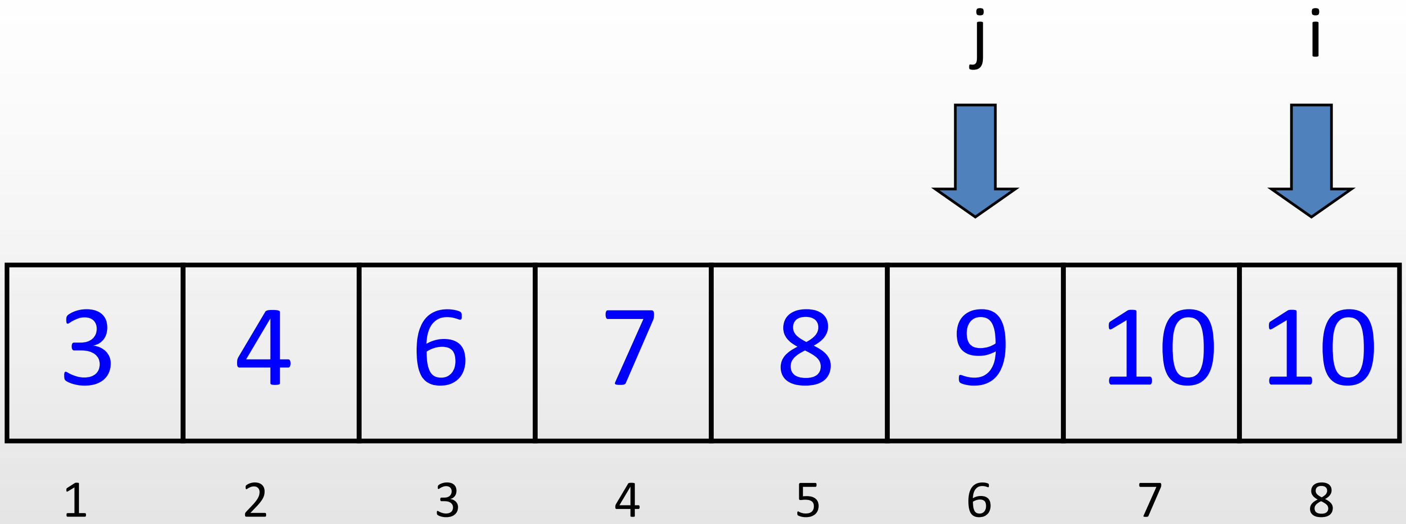
1st pass: i=7, obs=7
j = {1,2,3,4,5,6}



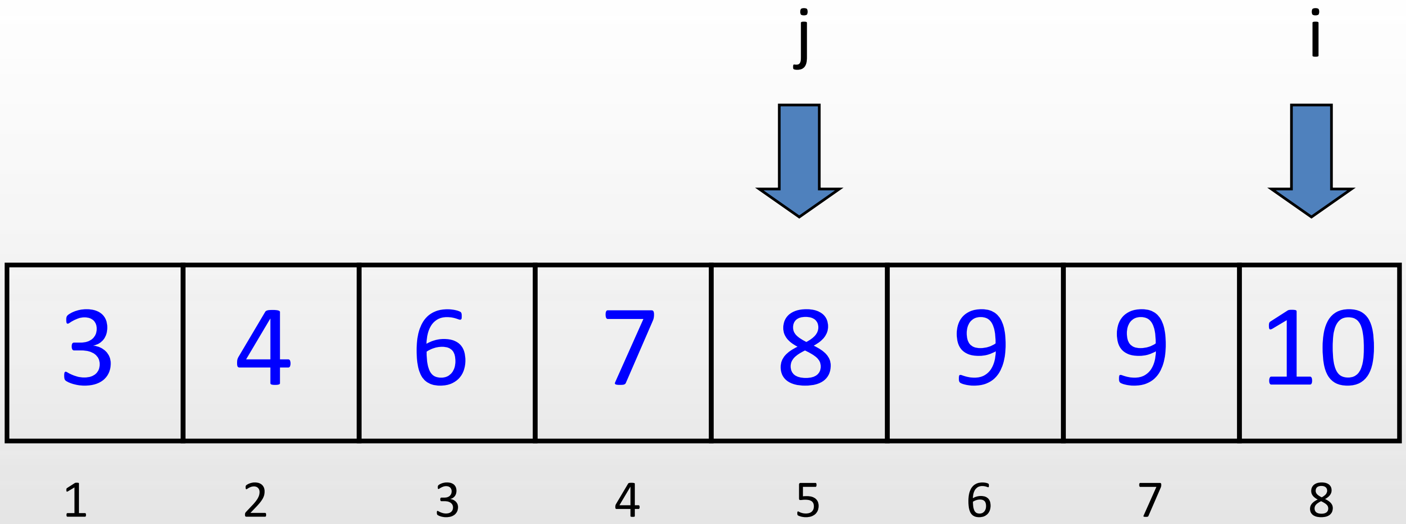
1st pass: i=8, obs=2
j = {1,2,3,4,5,6,7}



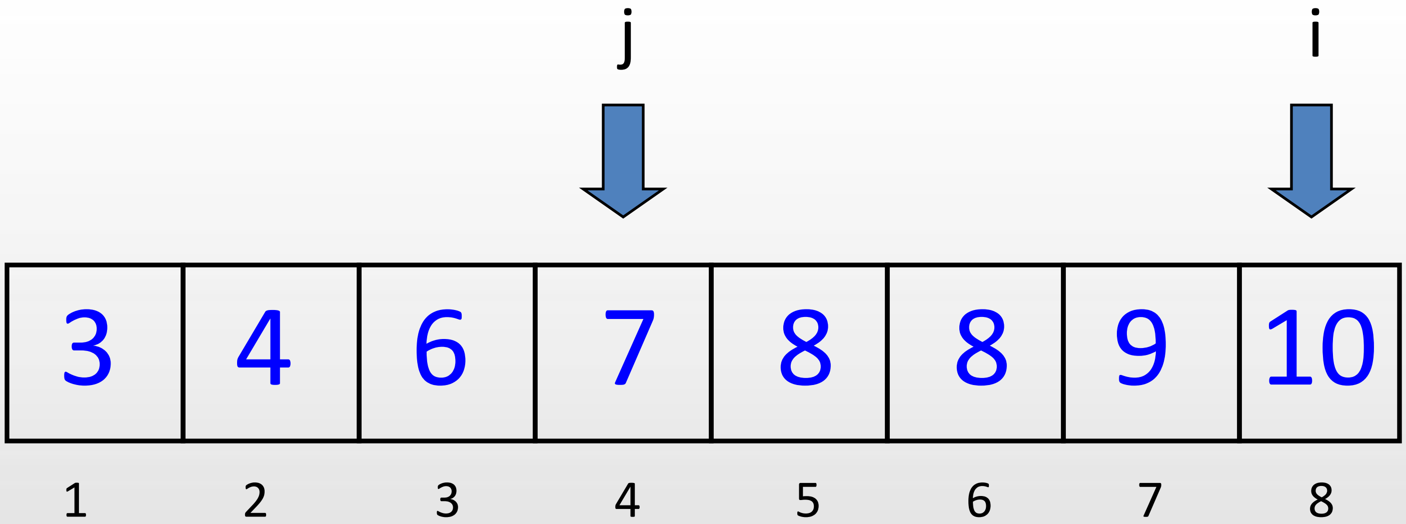
1st pass: i=8, obs=2
j = {1,2,3,4,5,6,7}



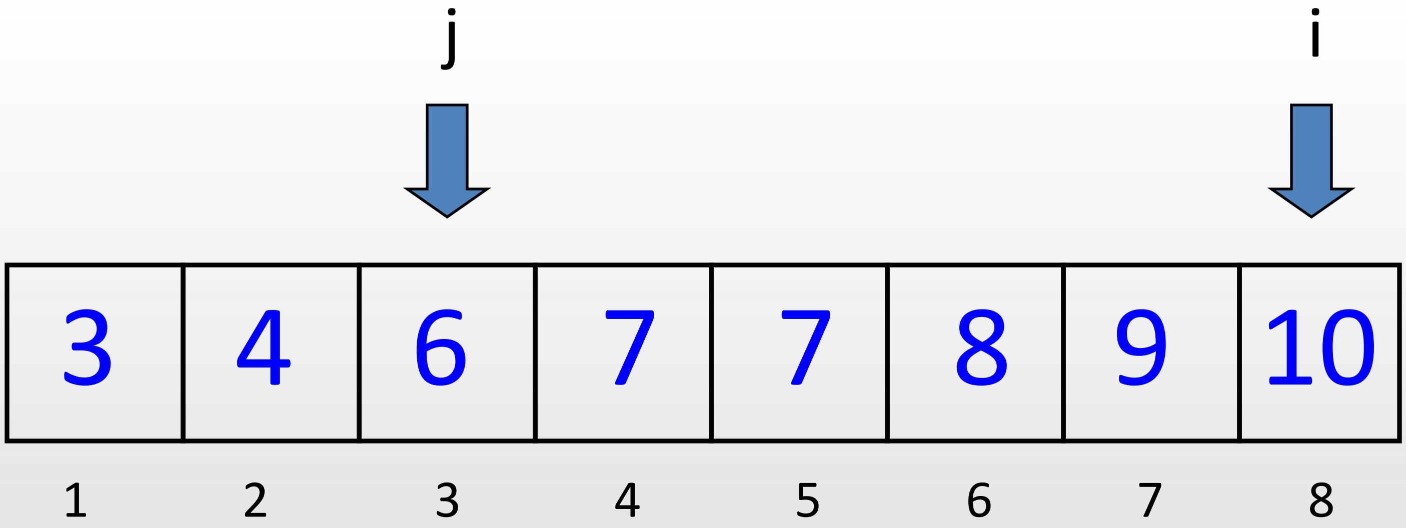
1st pass: i=8, obs=2
j = {1,2,3,4,5,6,7}



1st pass: i=8, obs=2
j = {1,2,3,4,5,6,7}



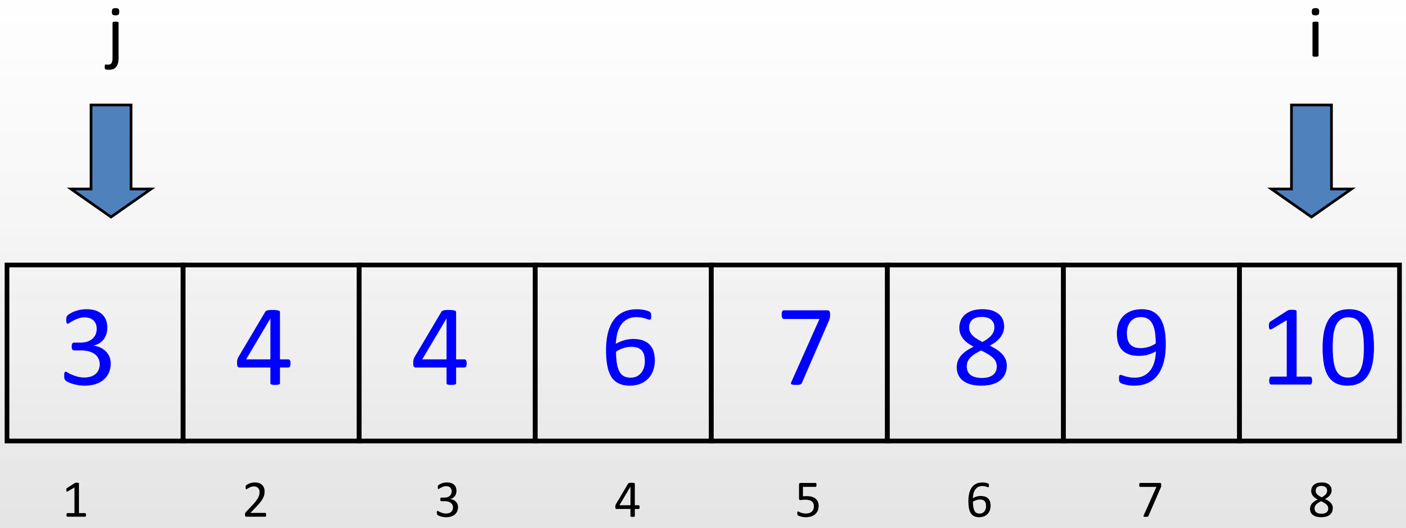
1st pass: i=8, obs=2
j = {1,2,3,4,5,6,7}



1st pass: i=8, obs=2
j = {1,2,3,4,5,6,7}

	j						i
3	4	6	6	7	8	9	10
1	2	3	4	5	6	7	8

1st pass: i=8, obs=2
j = {1,2,3,4,5,6,7}



1st pass: i=8, obs=2

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

j

i

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

1st pass: i=8, obs=2
j = {1,2,3,4,5,6,7}

Selection Sort

- Selection sort performs sorting by repeatedly putting the largest element in the unprocessed portion of the array to the end of the unprocessed portion until the whole array is sorted.

Selection Sort

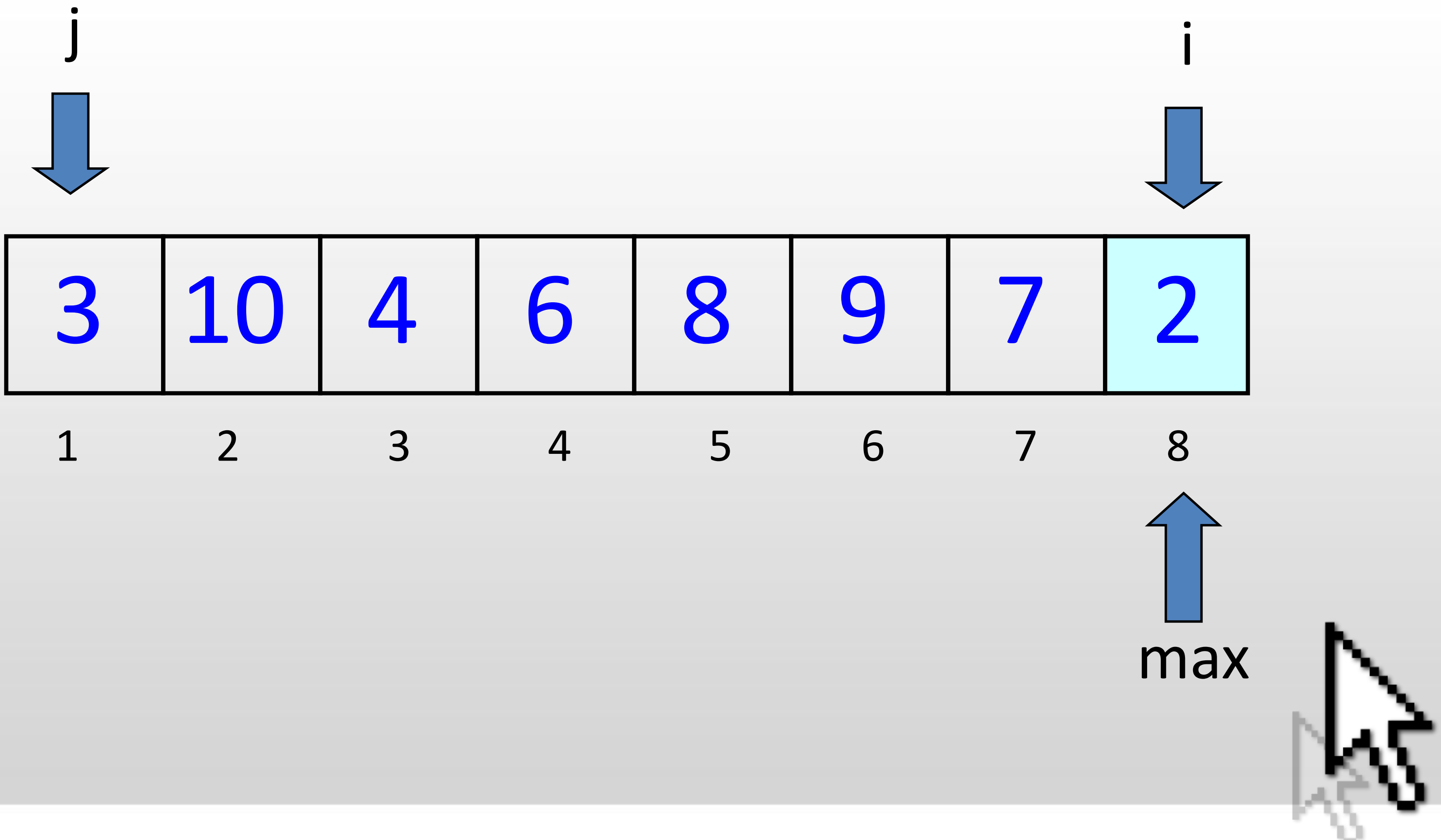
```
SelectionSort(A,n)
 begin
   for i = n downto 2 do
        max = i
        for j = 1 to (i-1) do
         if A[j] >A[max]
              max = j
        swap(A[i], A[max])
  end
```

Sort 3, 10, 4, 6, 8, 9, 7, 2 using selection sort

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

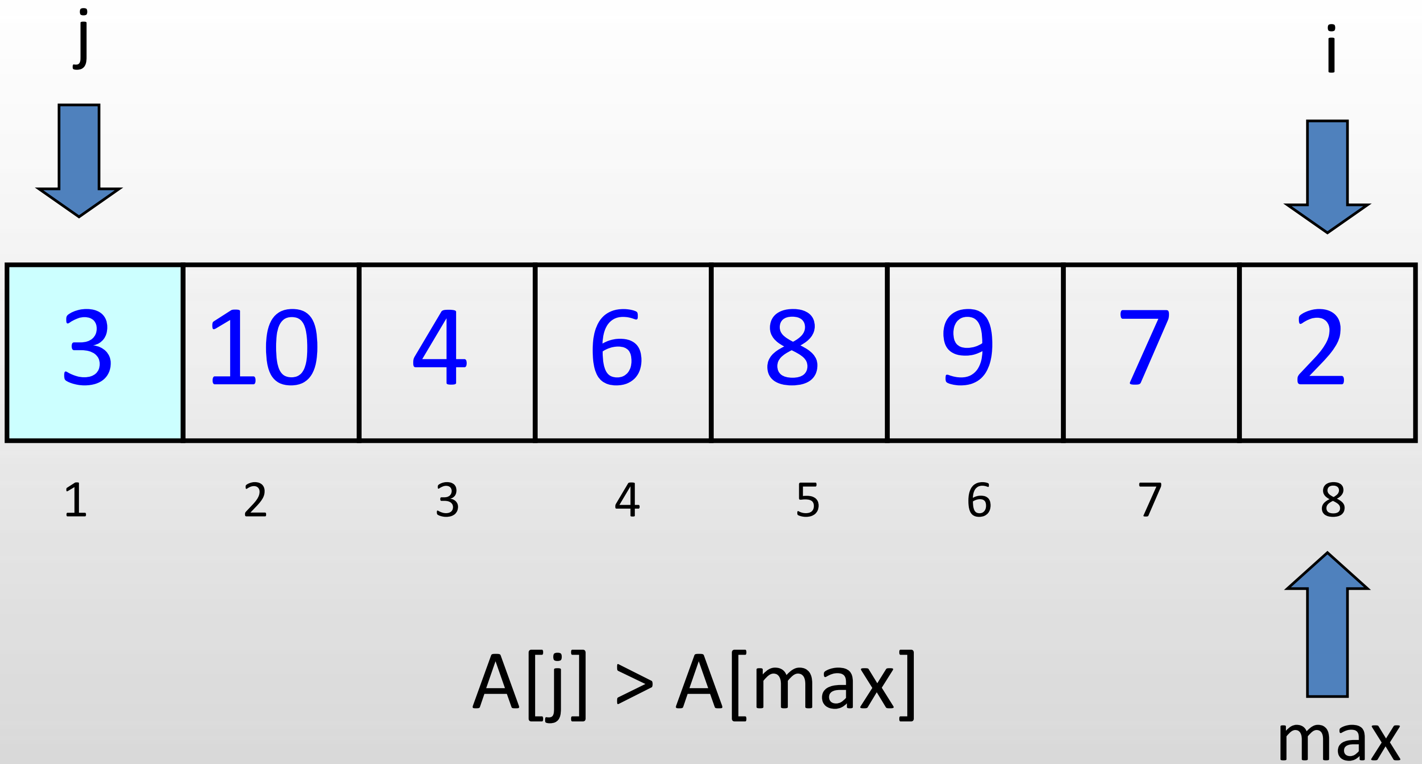
1st pass: i=8

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



1st pass: i=8

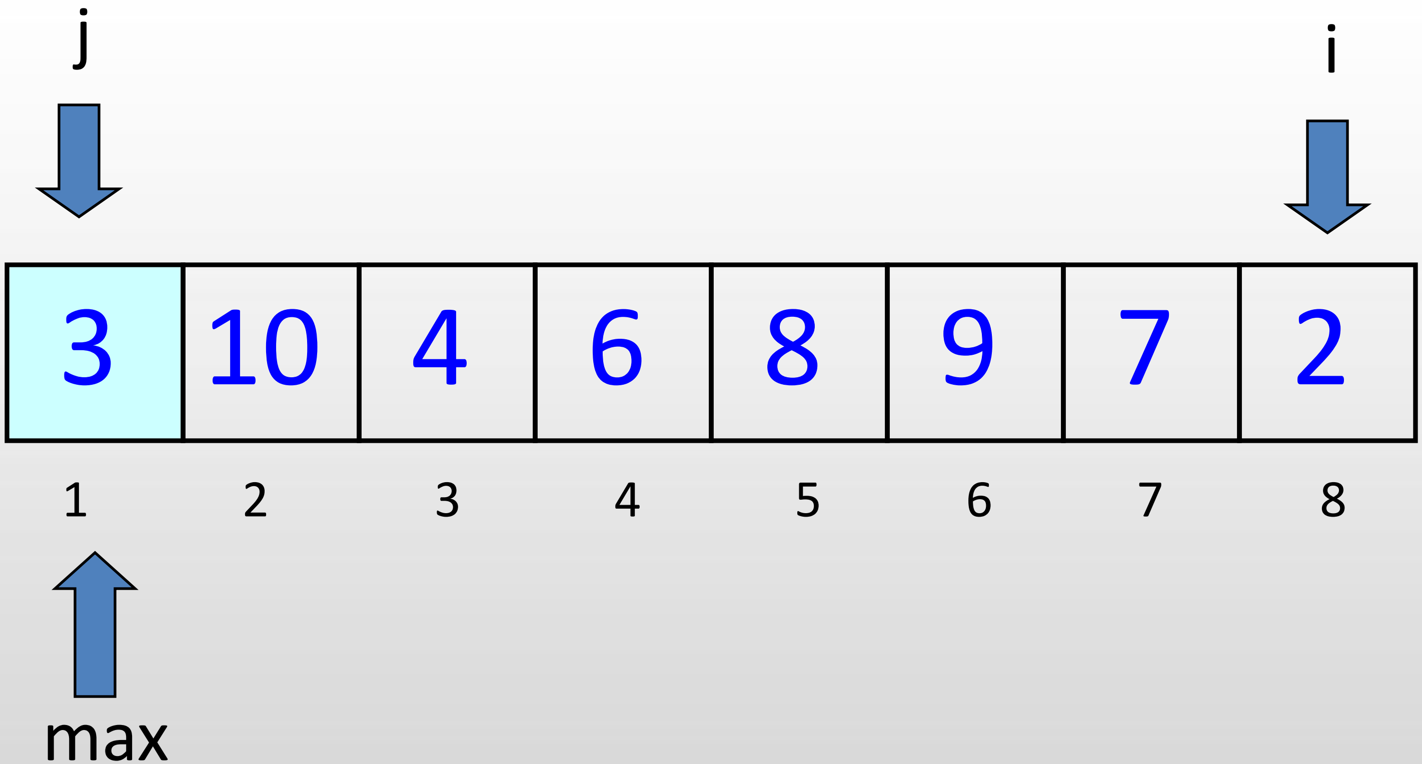
j = {1,2,3,4,5,6,7}



A[j] > A[max]

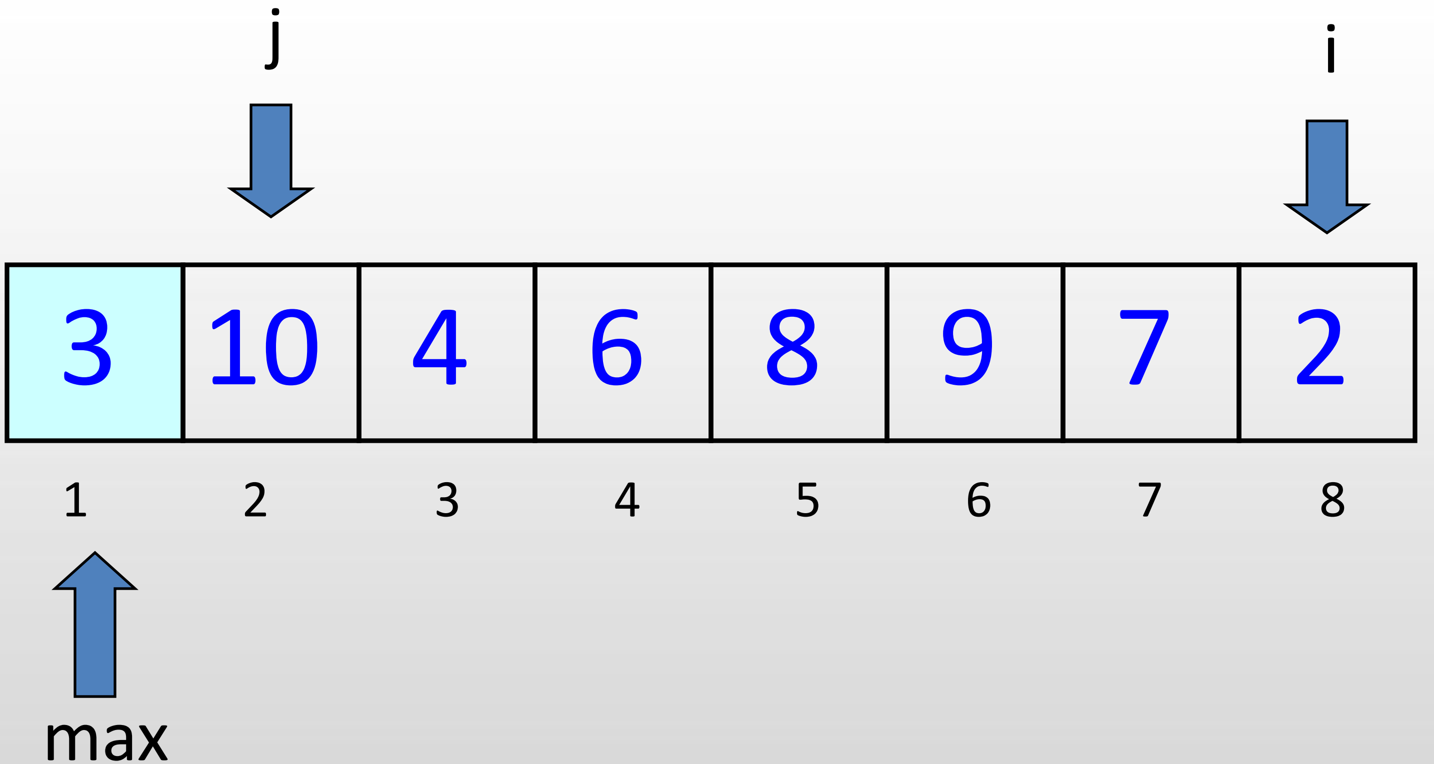
1st pass: i=8

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



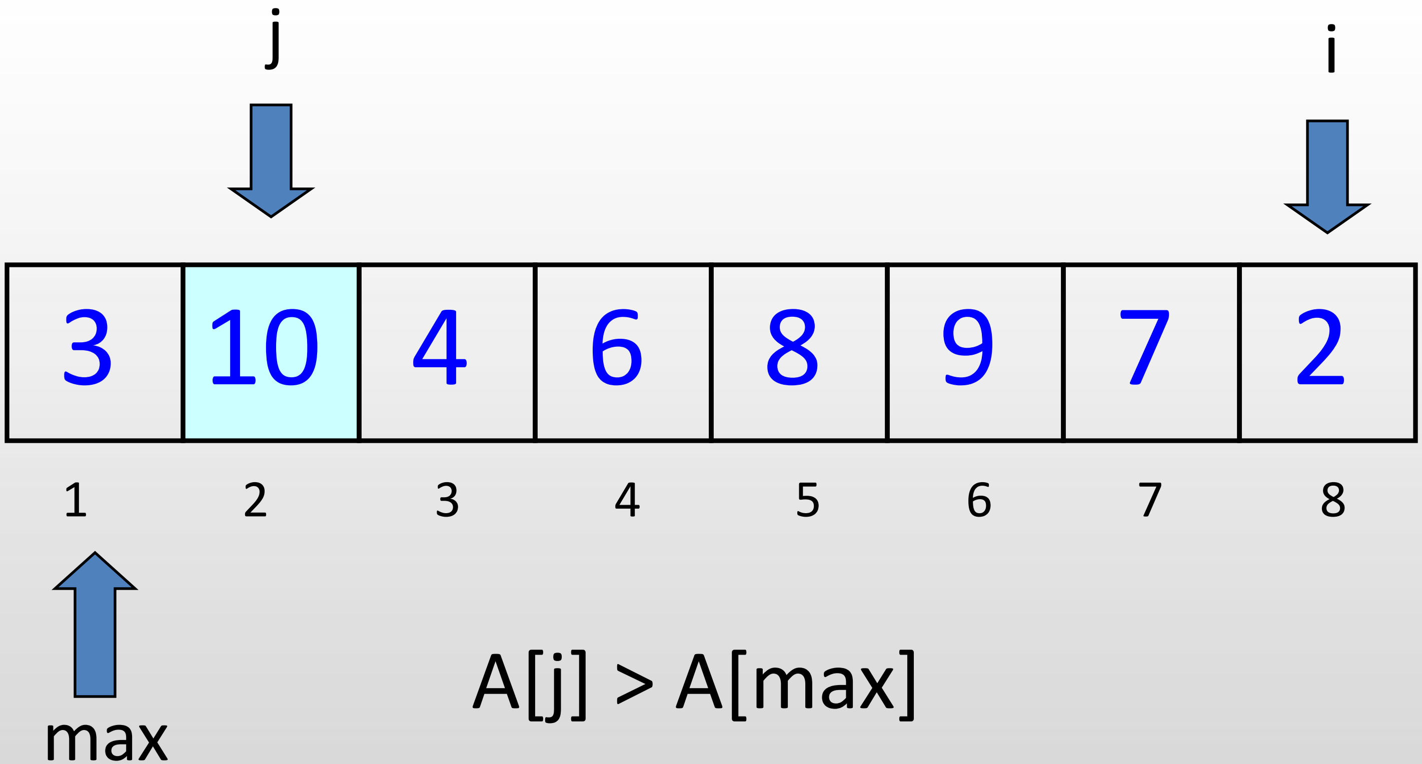
1st pass: i=8

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



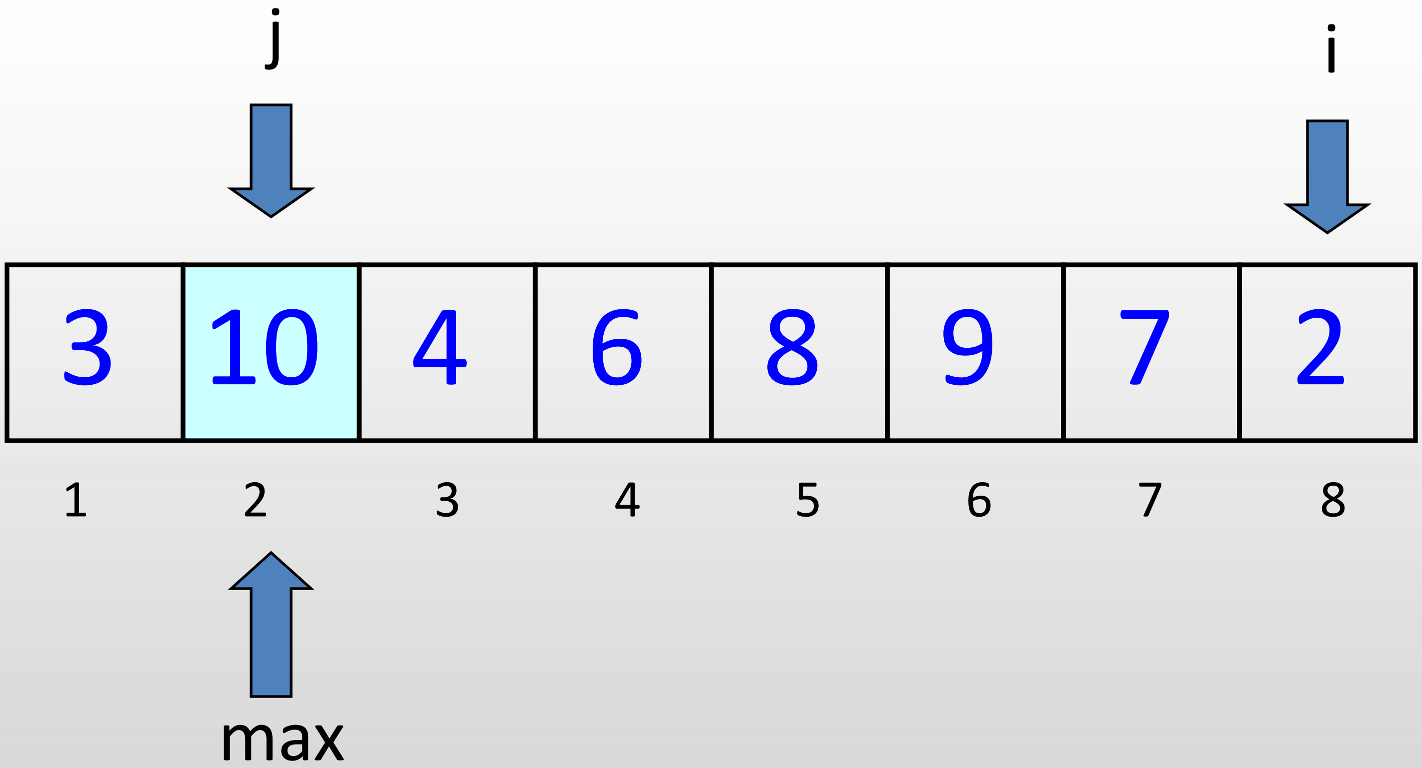
1st pass: i=8

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



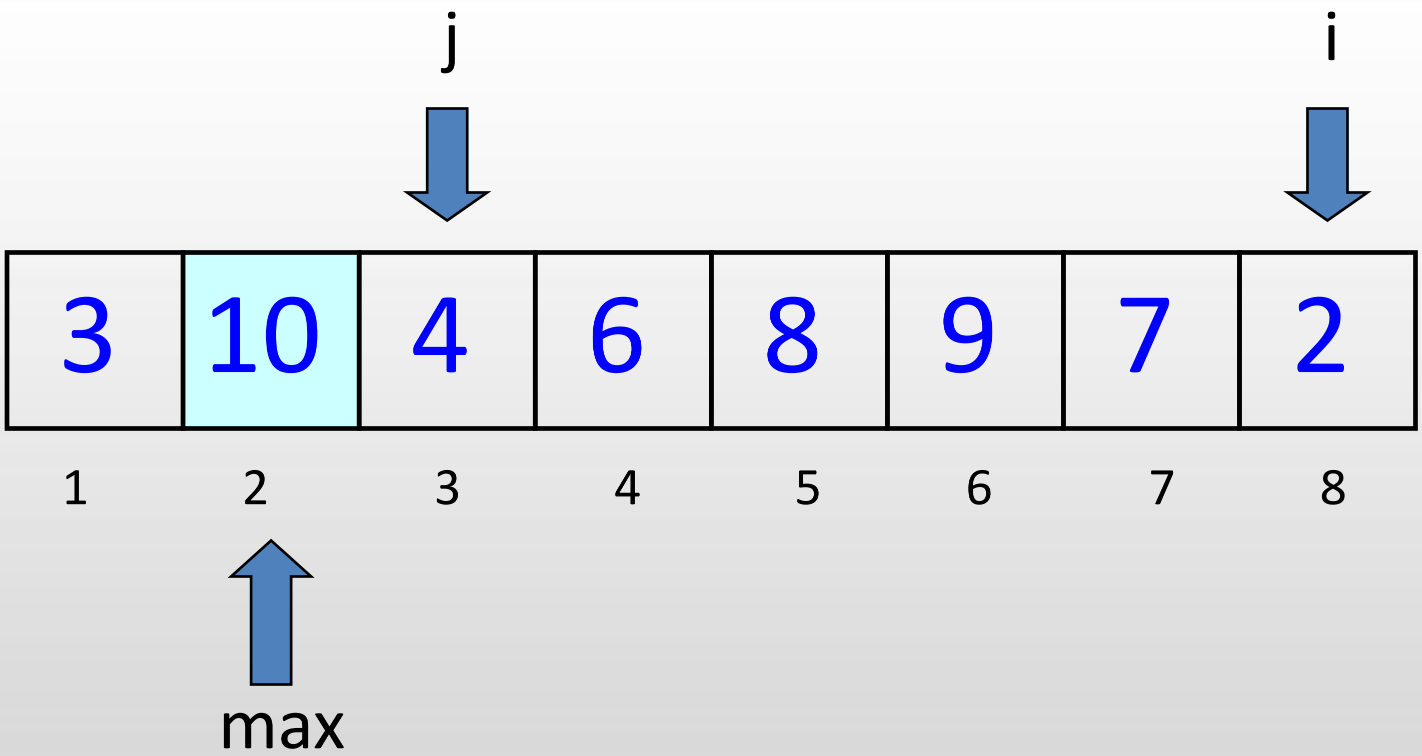
1st pass: i=8

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



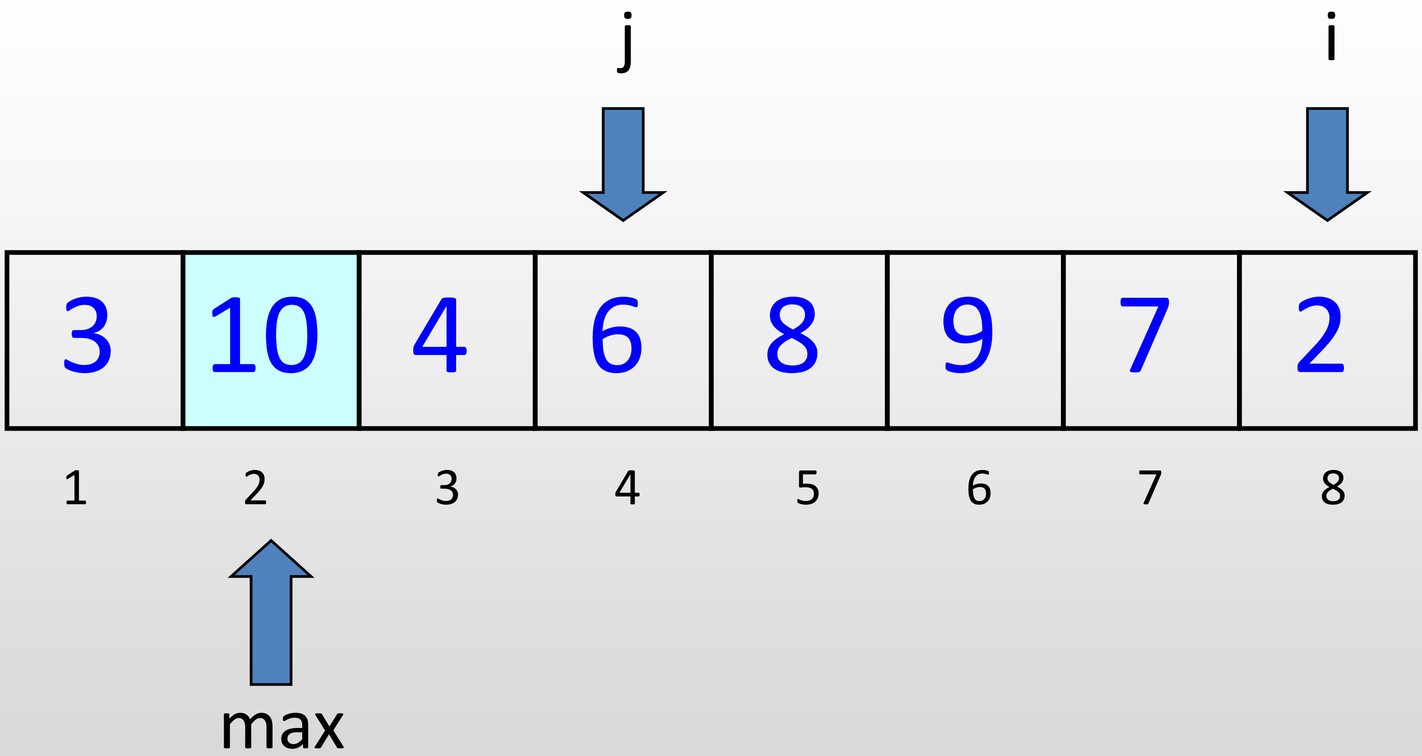
1st pass: i=8

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



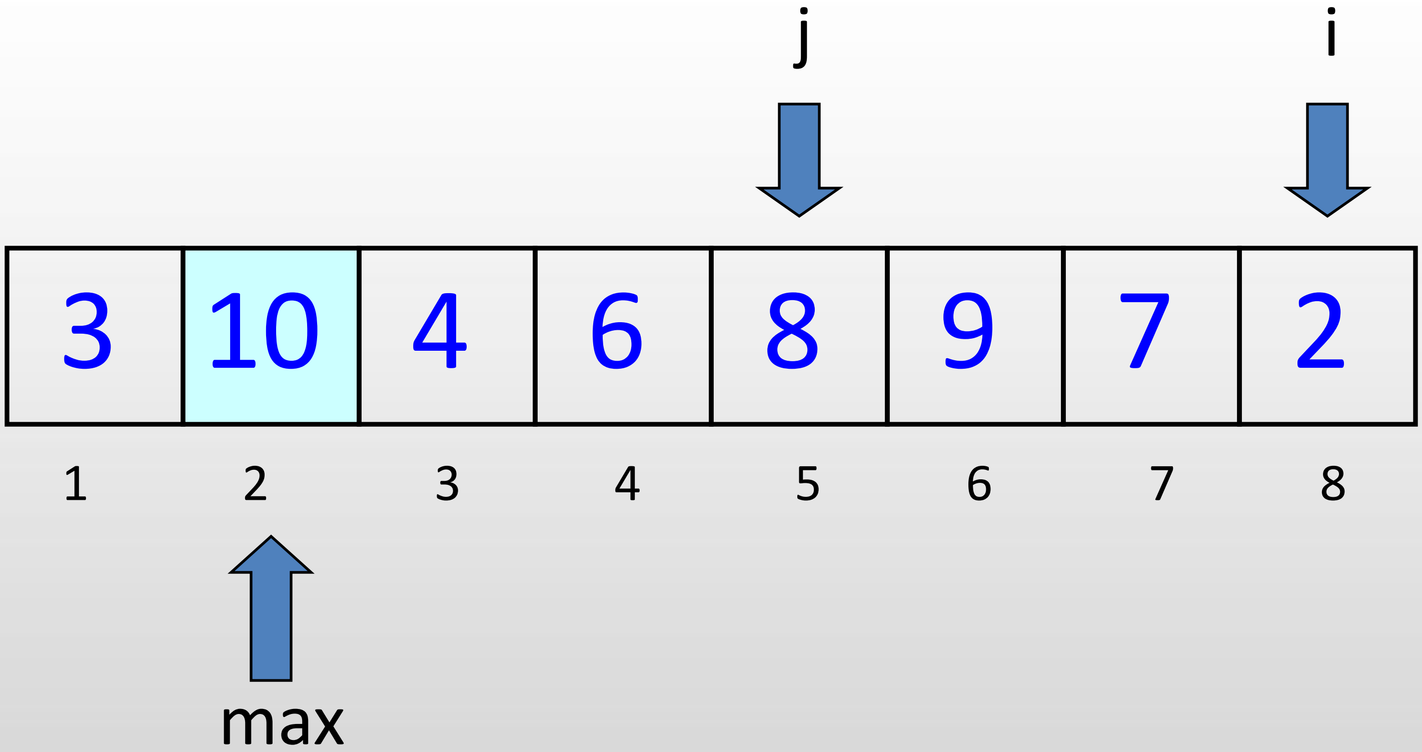
1st pass: i=8

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



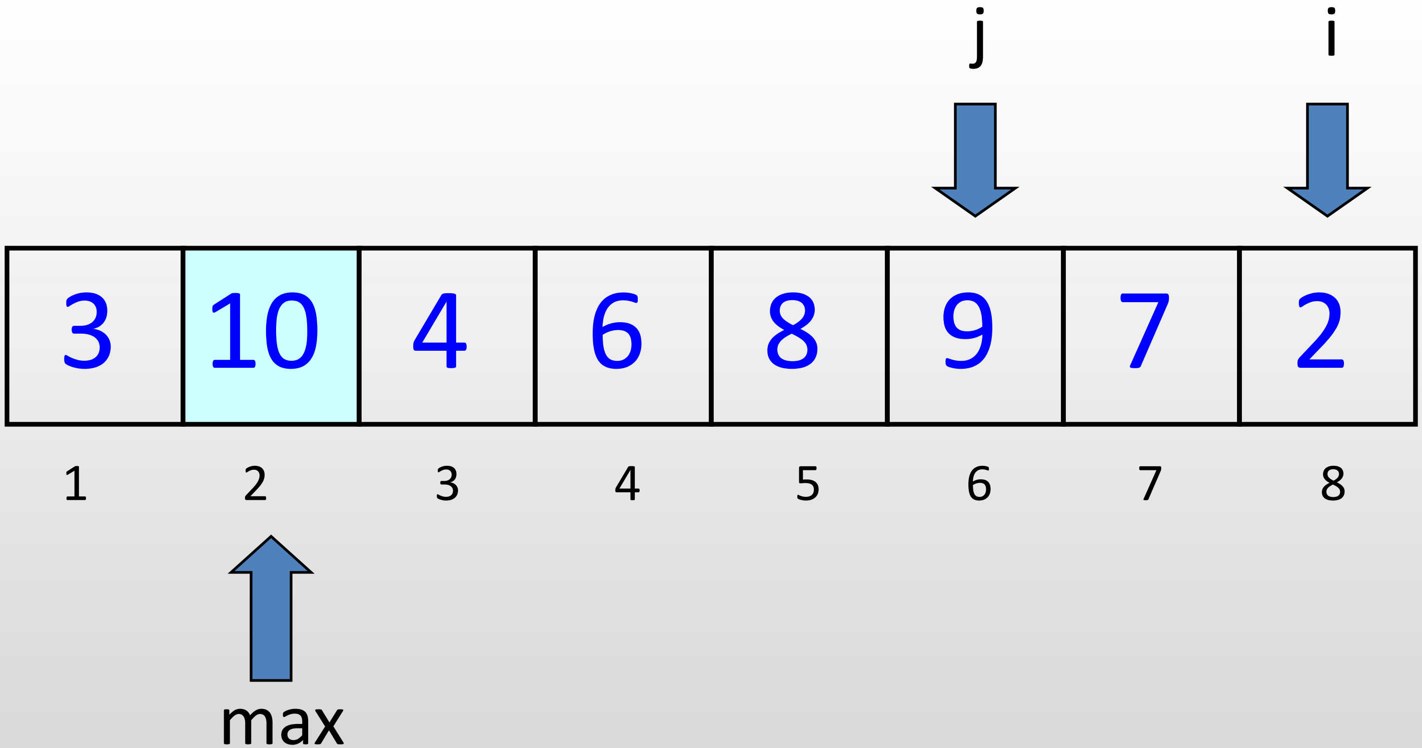
1st pass: i=8

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



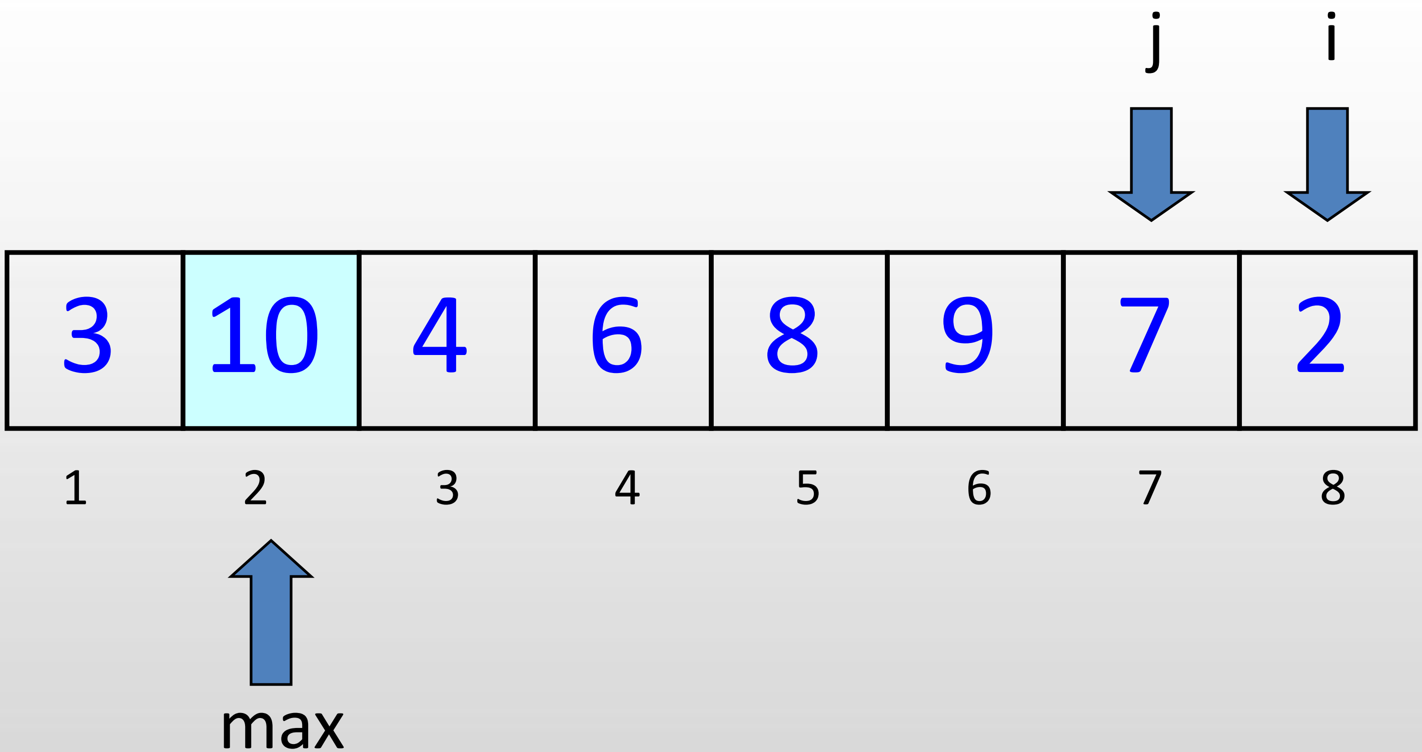
1st pass: i=8

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



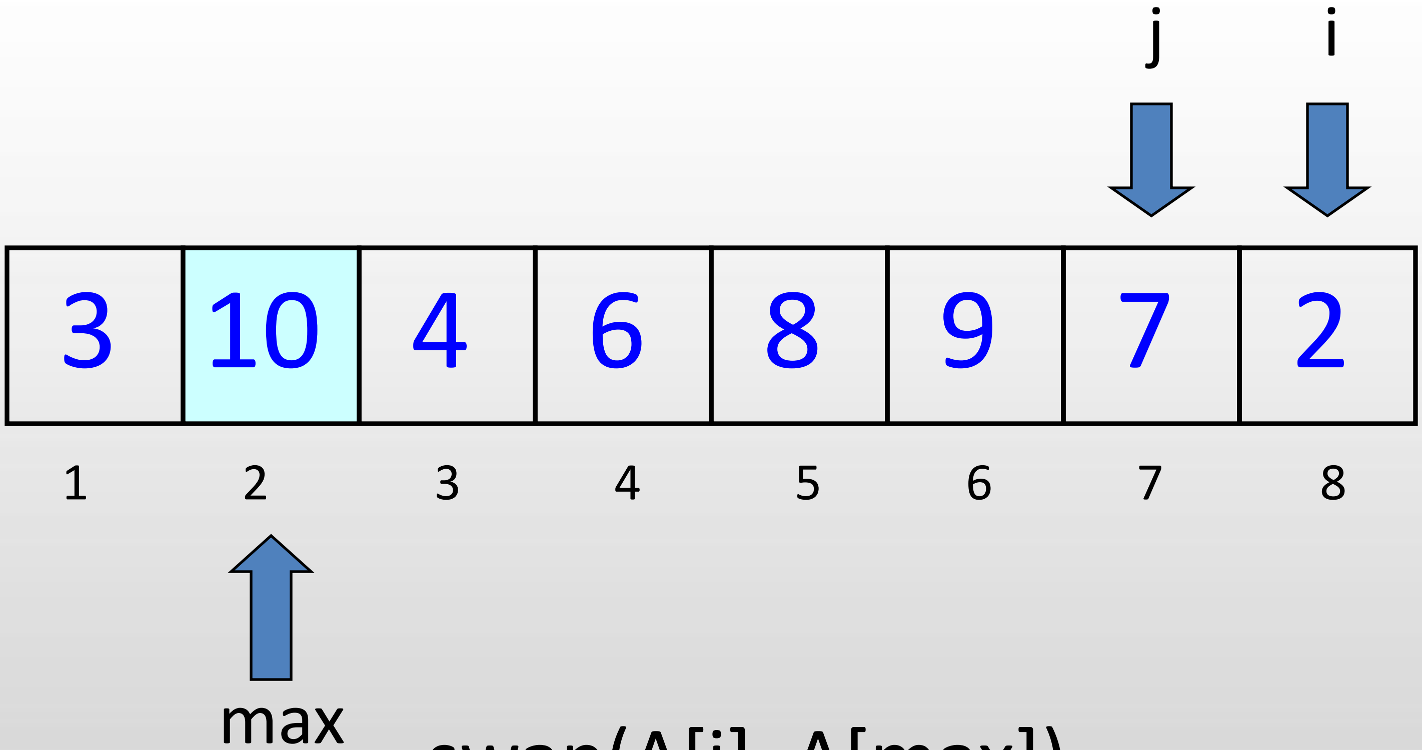
1st pass: i=8

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



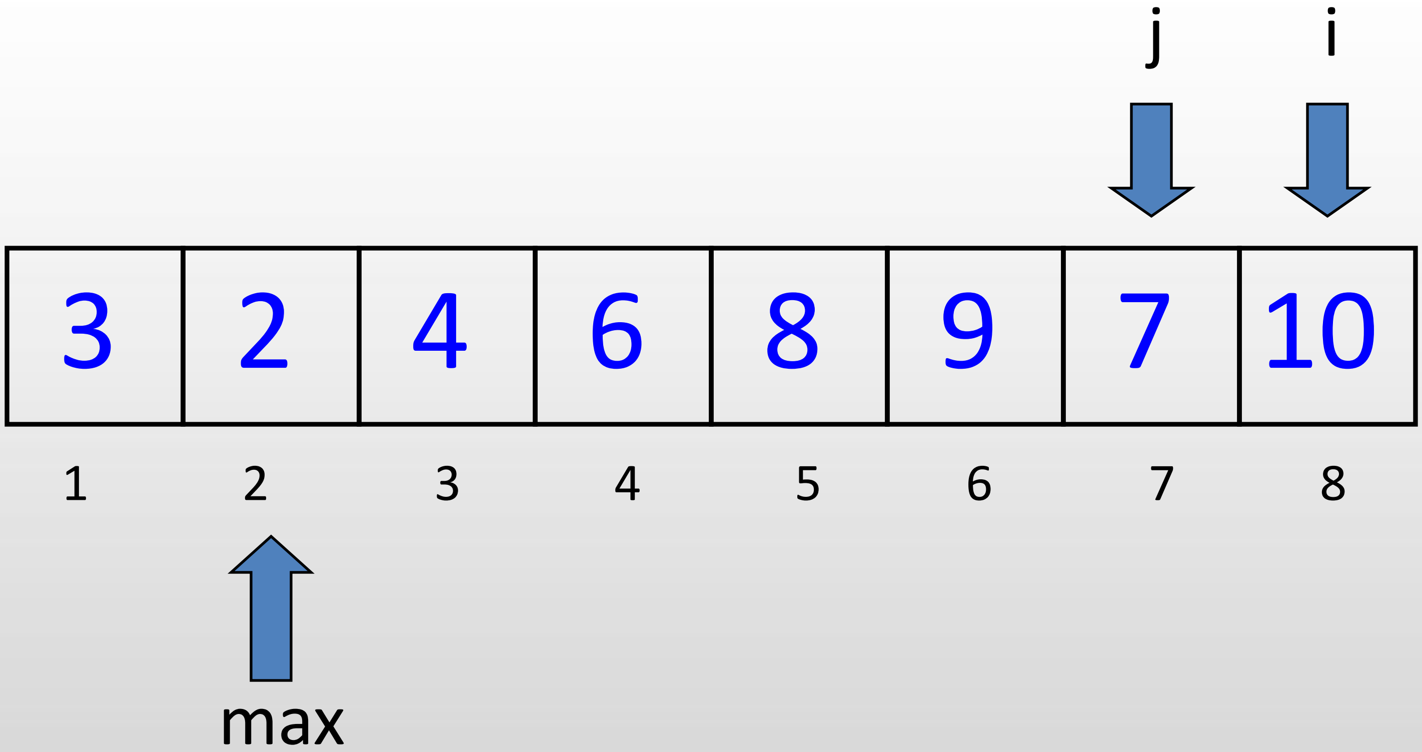
1st pass: i=8

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



swap(A[i], A[max])

1st pass: i=8
j = {1,2,3,4,5,6,7}



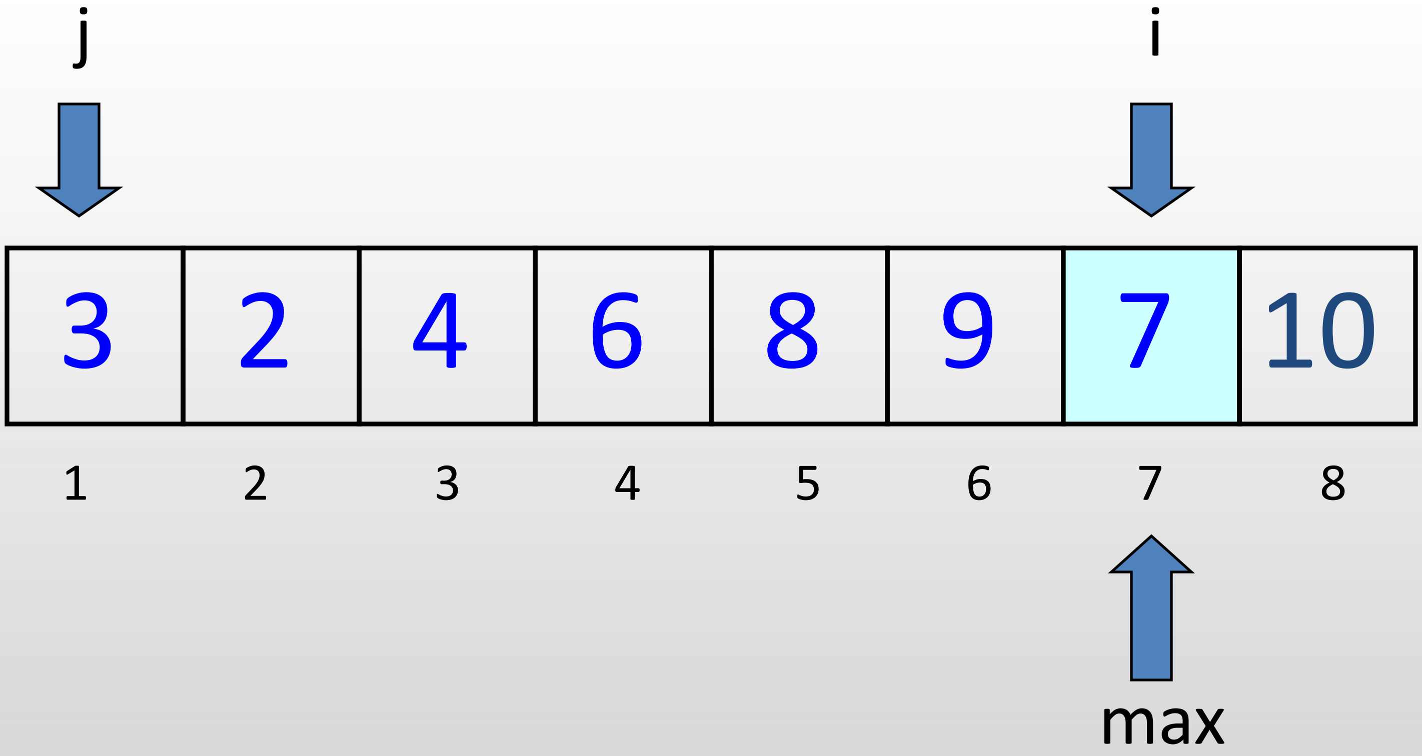
2nd pass: i=7
j = {1,2,3,4,5,6}

i

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

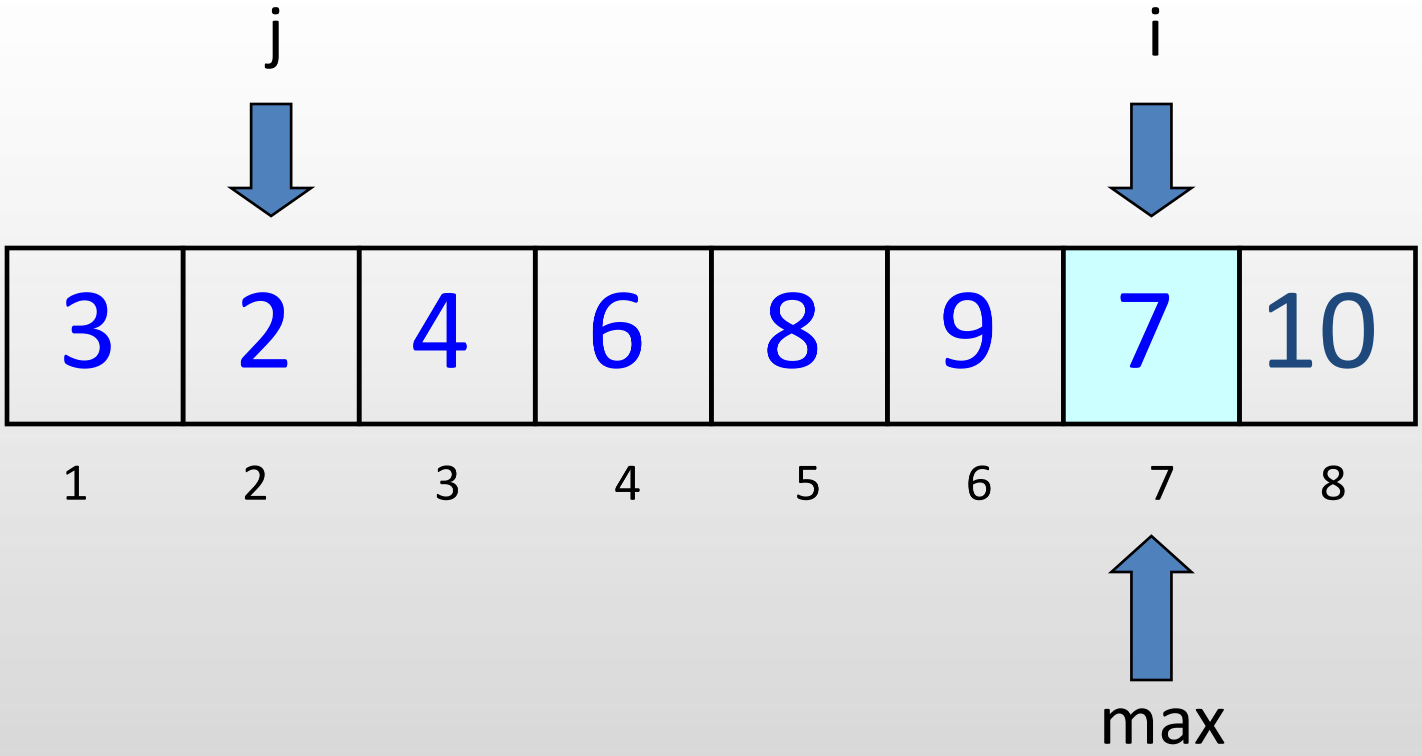
2nd pass: i=7

j = {1,2,3,4,5,6}



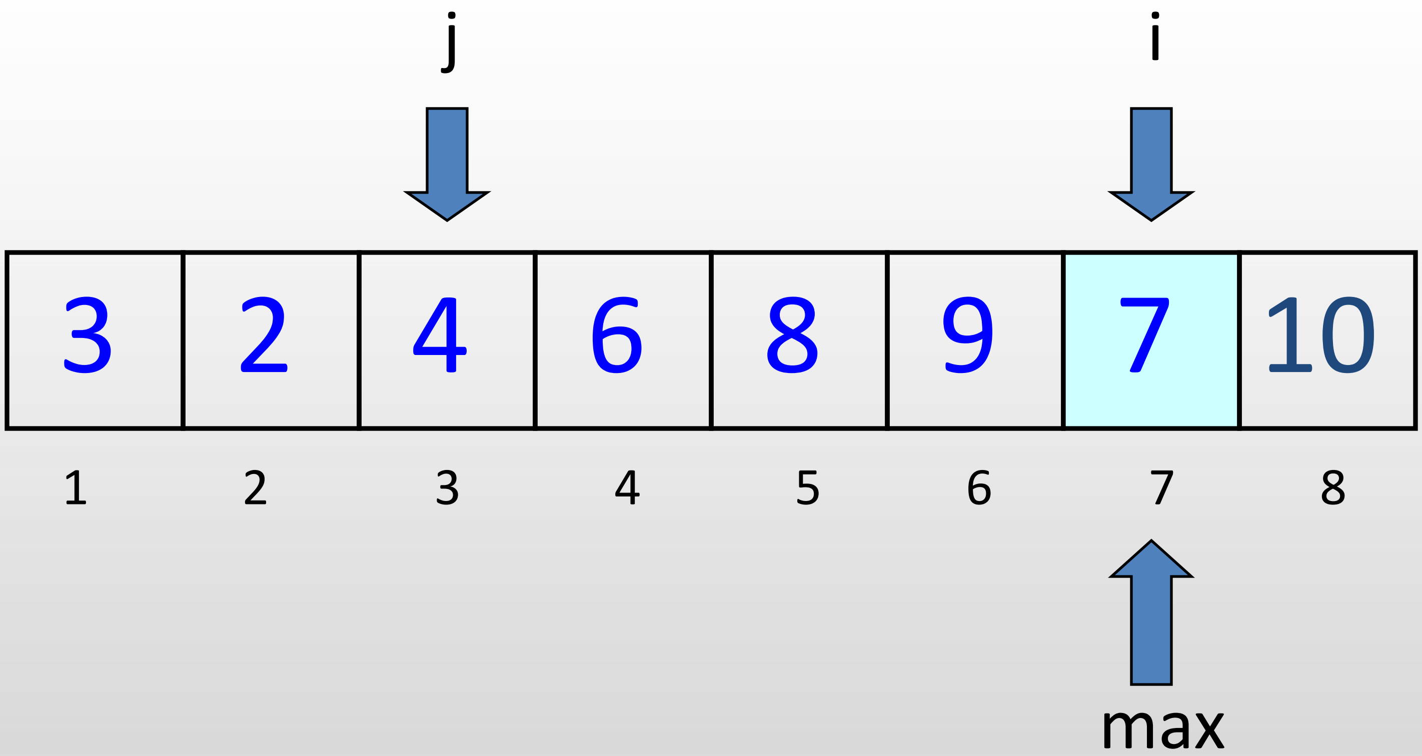
2nd pass: i=7

j = {1,2,3,4,5,6}



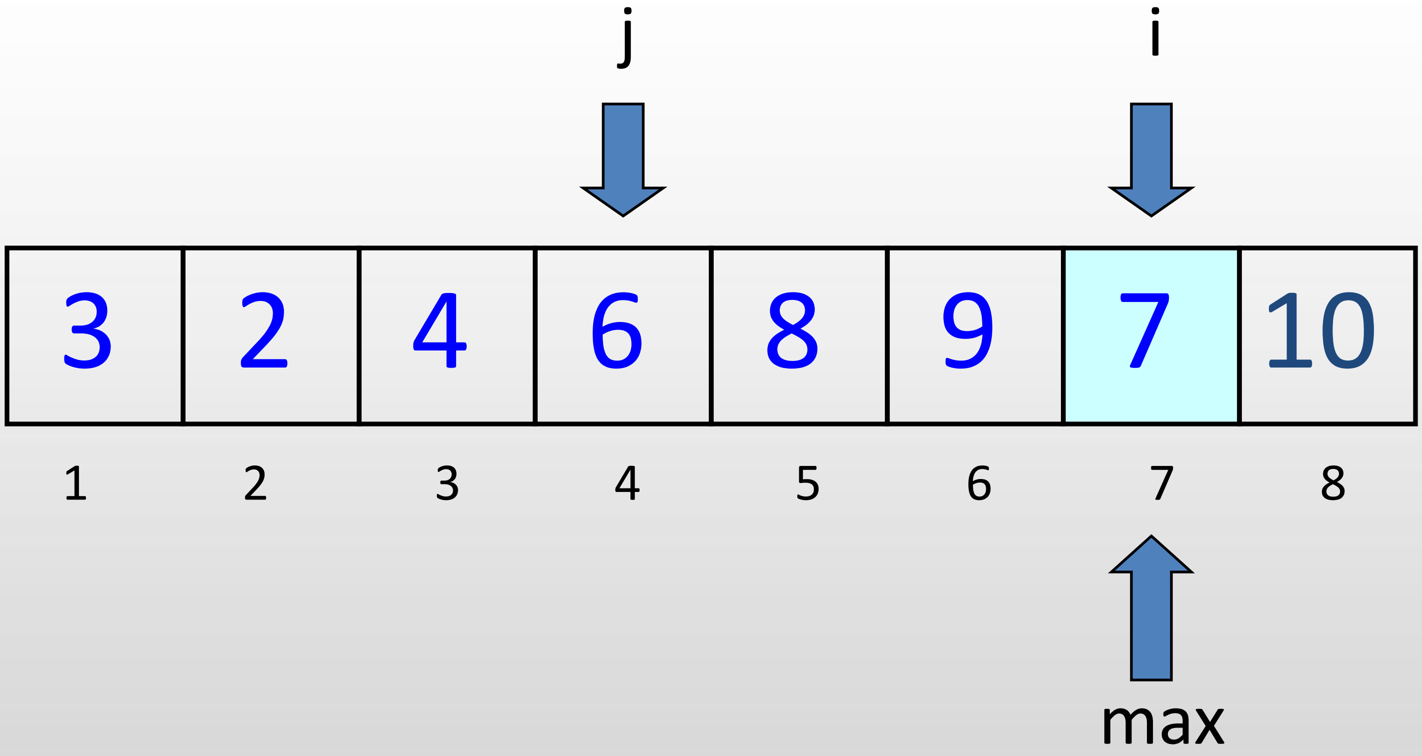
2nd pass: i=7

j = {1,2,3,4,5,6}



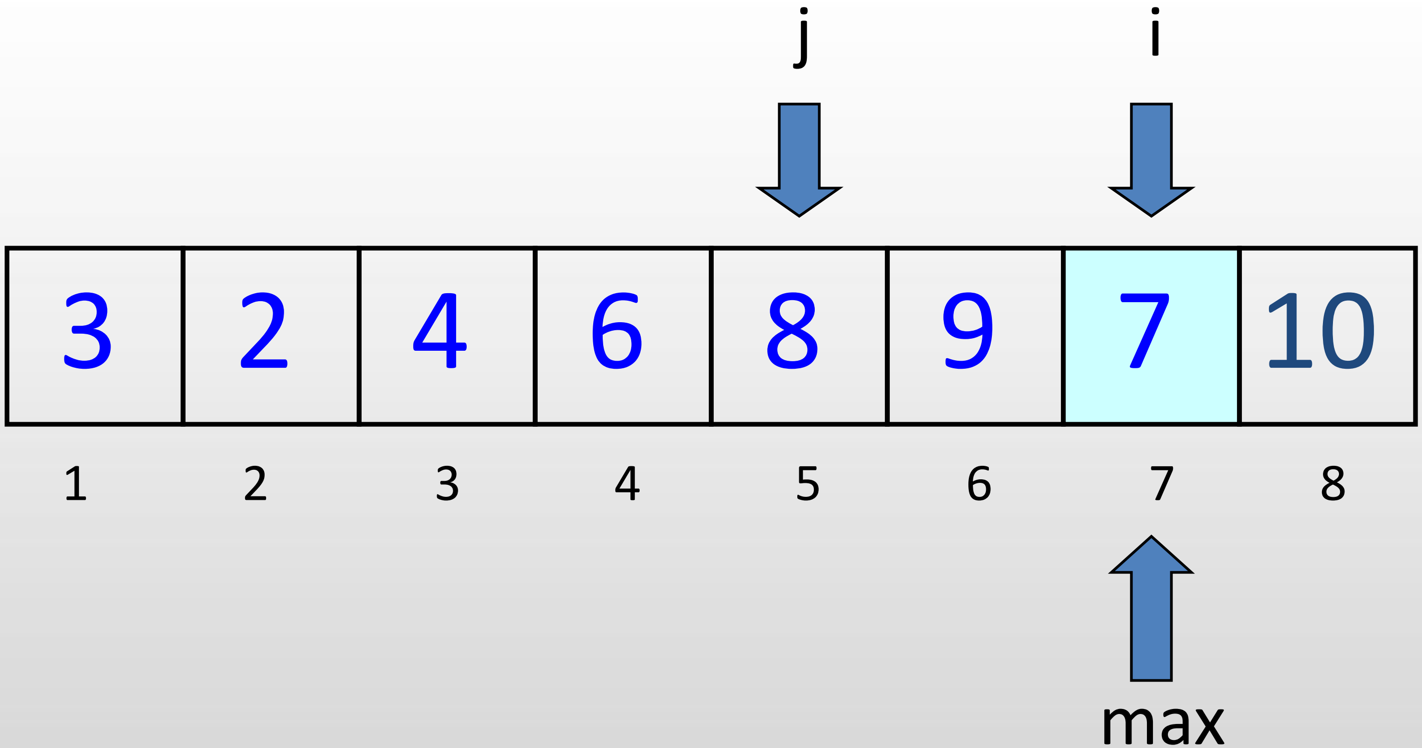
2nd pass: i=7

j = {1,2,3,4,5,6}



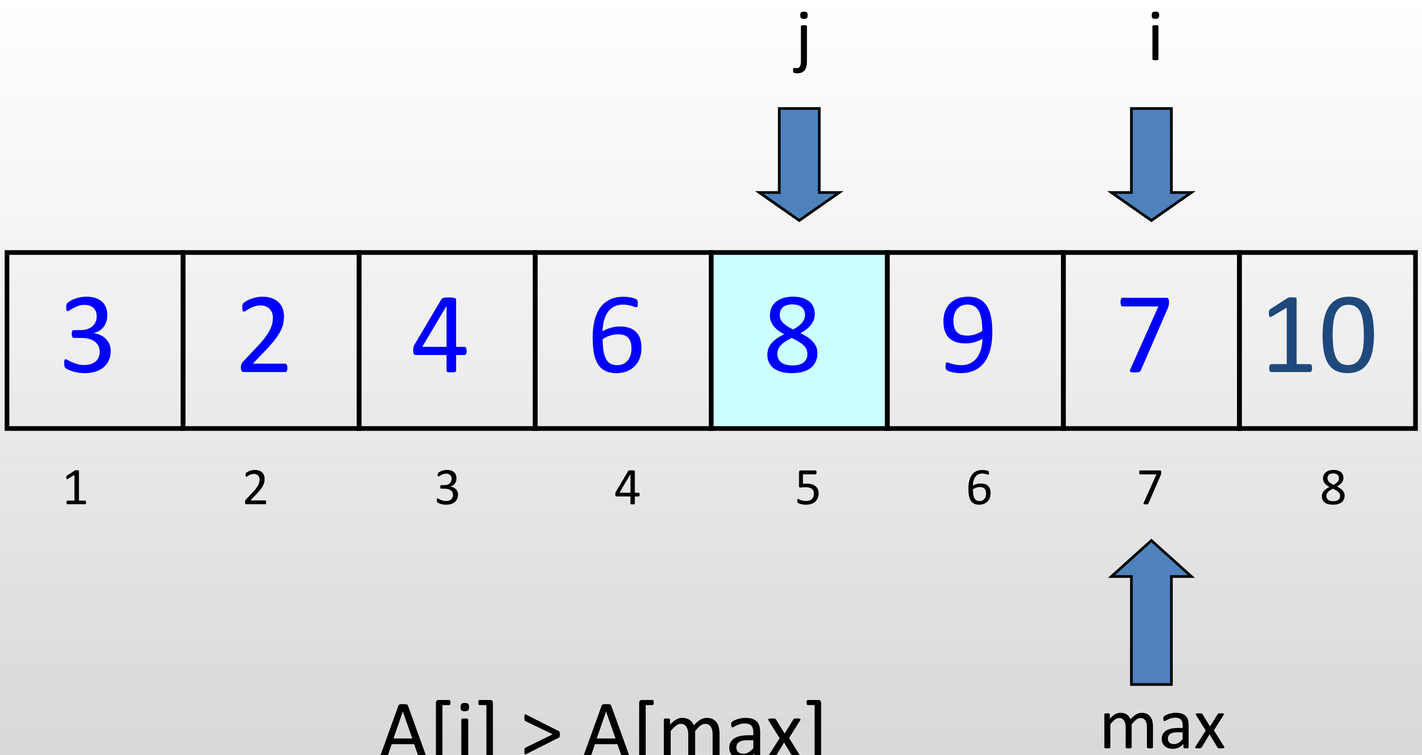
2nd pass: i=7

j = {1,2,3,4,5,6}



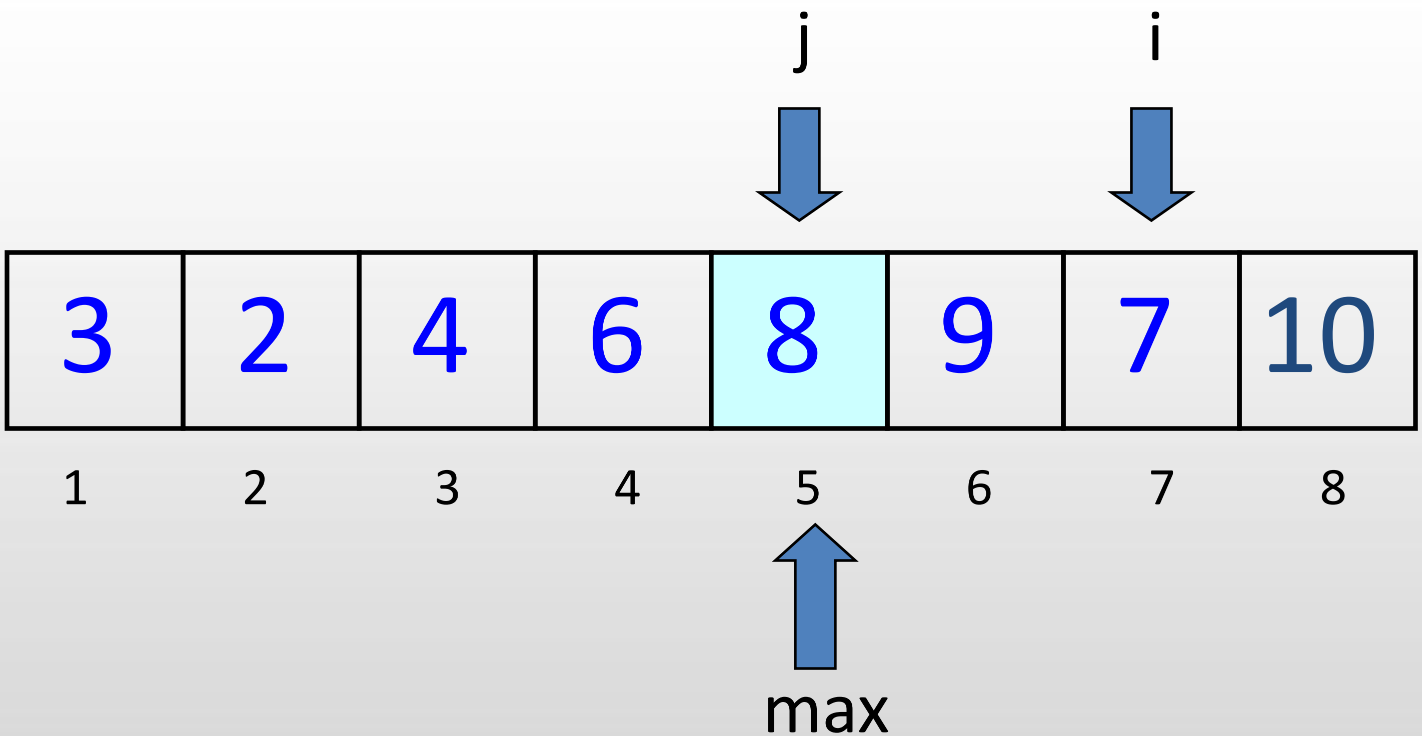
2nd pass: i=7

j = {1,2,3,4,5,6}

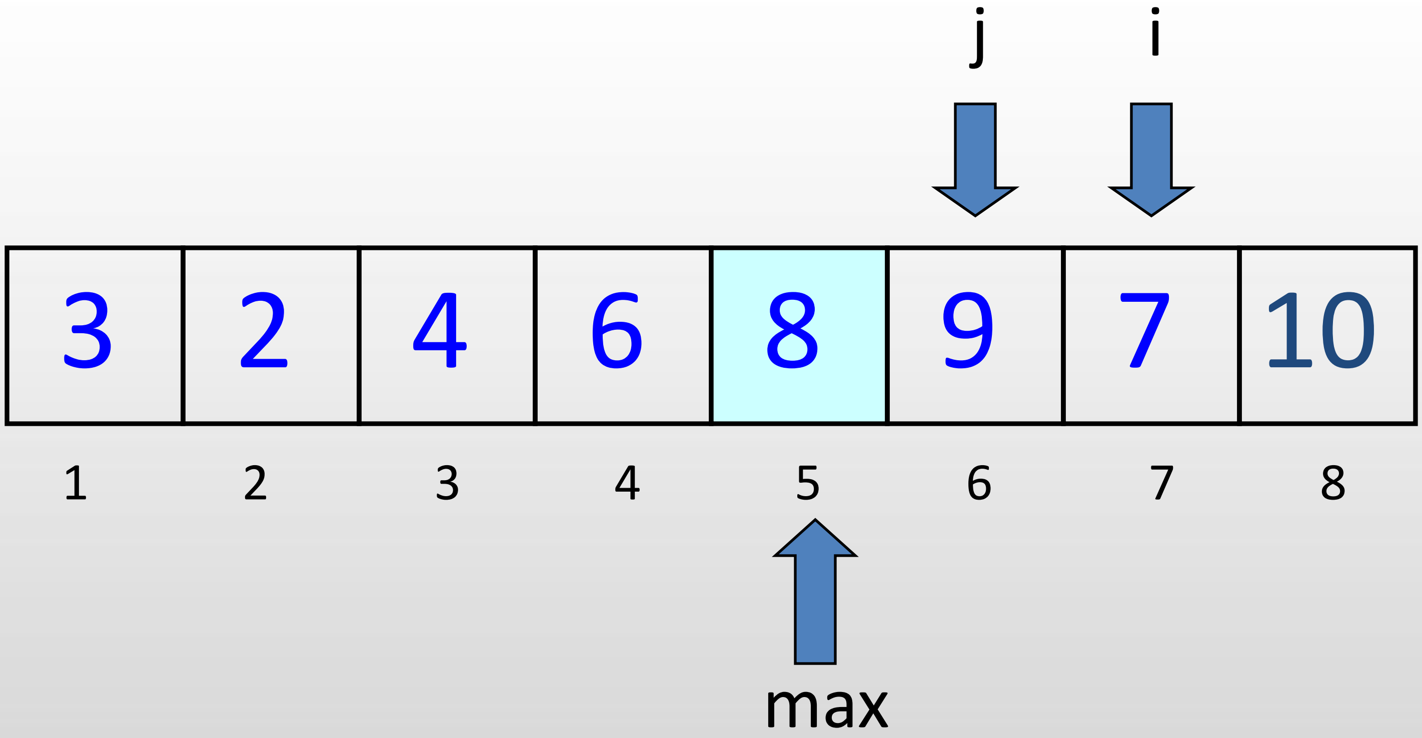


A[j] > A[max]

2nd pass: i=7
j = {1,2,3,4,5,6}

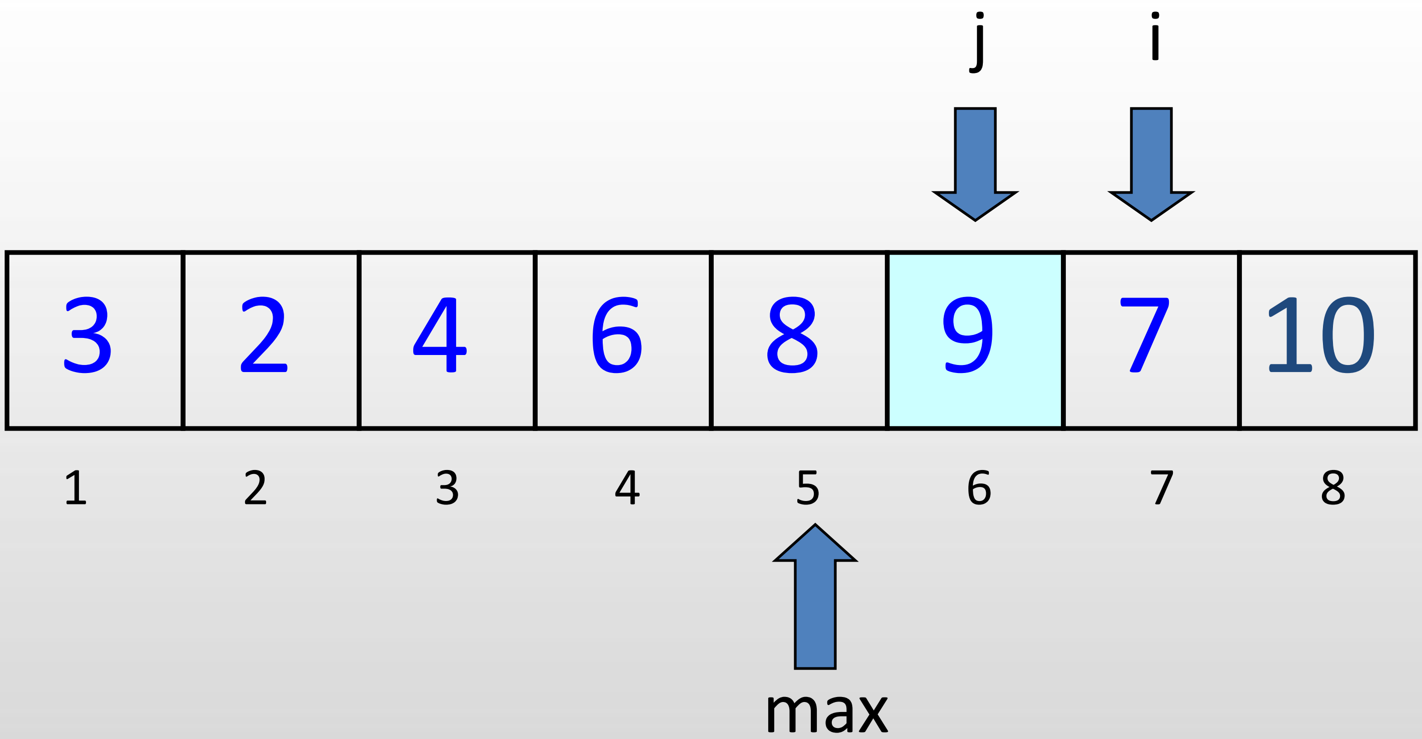


2nd pass: i=7
j = {1,2,3,4,5,6}



2nd pass: i=7

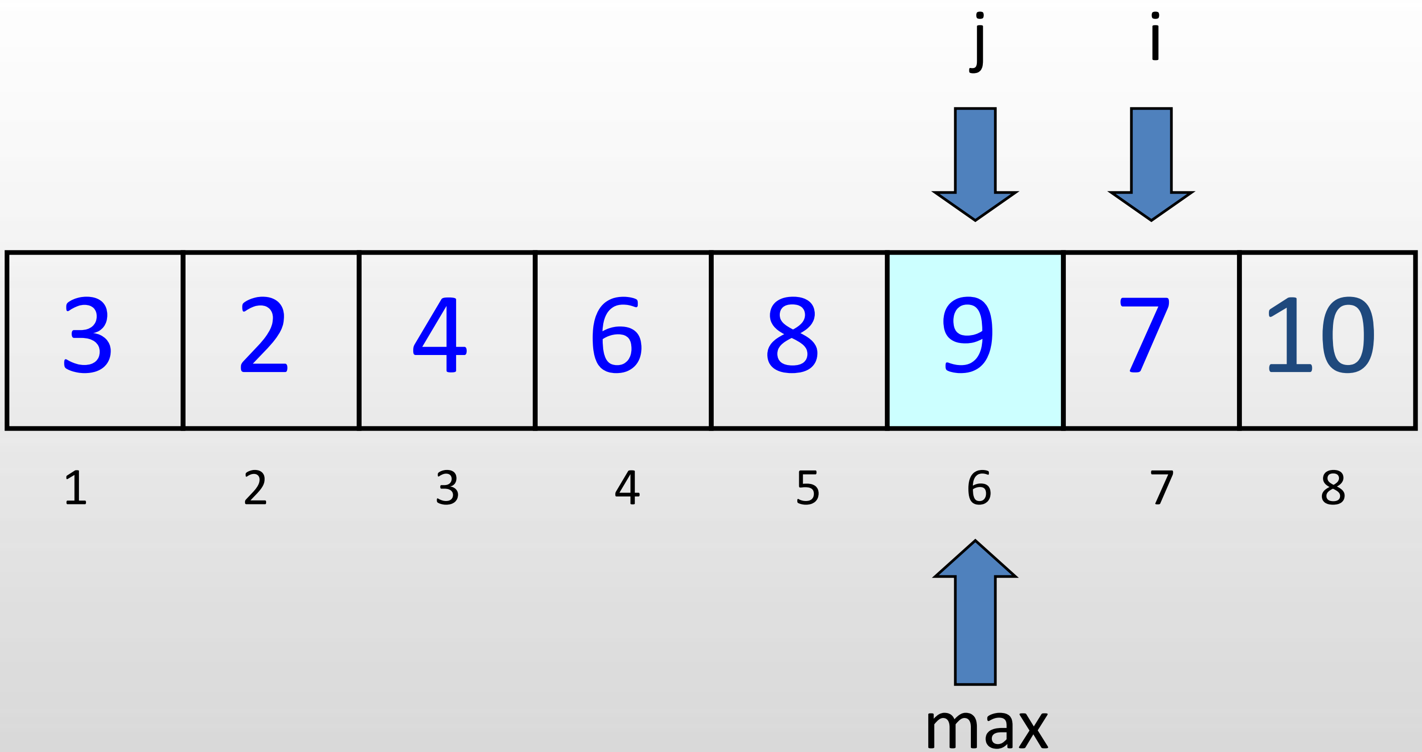
j = {1,2,3,4,5,6}



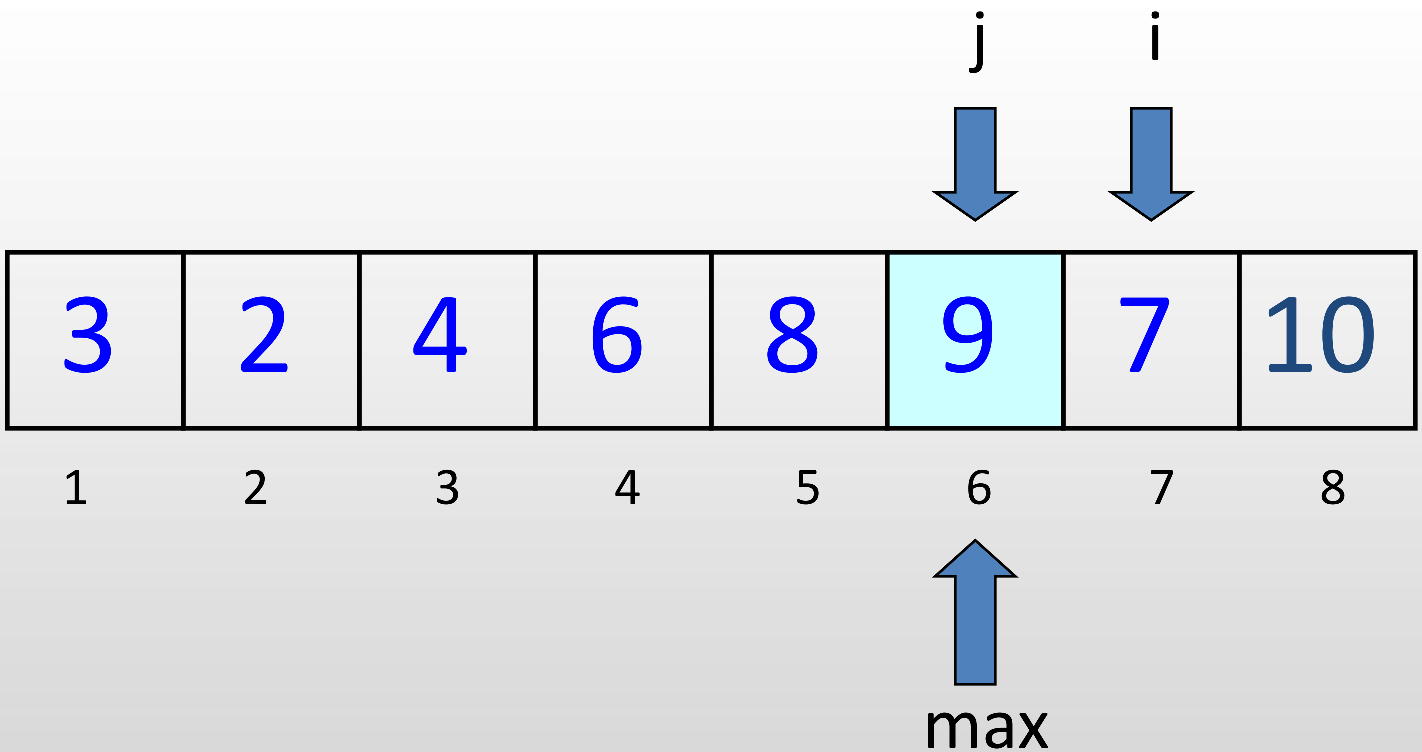
A[j] > A[max]

2nd pass: i=7

j = {1,2,3,4,5,6}

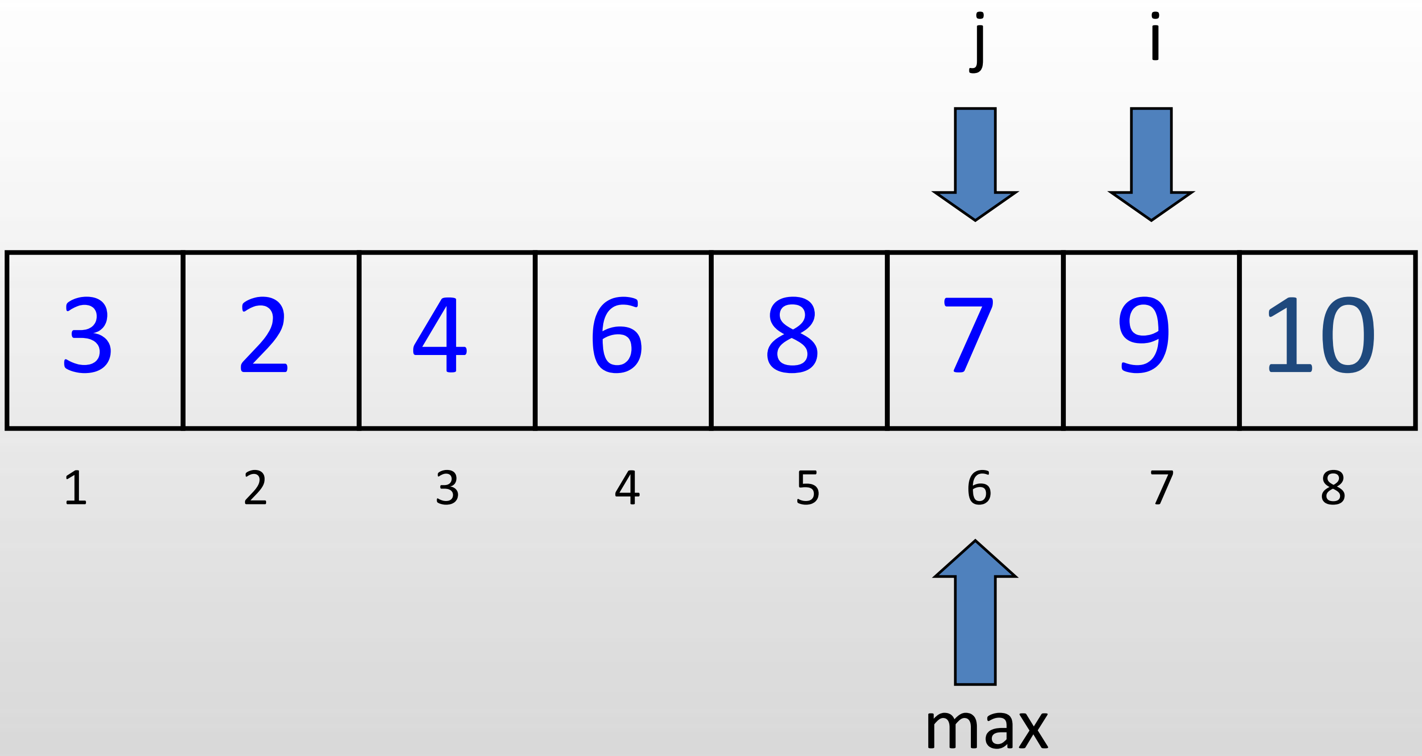


2nd pass: i=7
j = {1,2,3,4,5,6}



swap(A[i], A[max])

2nd pass: i=7
j = {1,2,3,4,5,6}



3rd pass: i=6

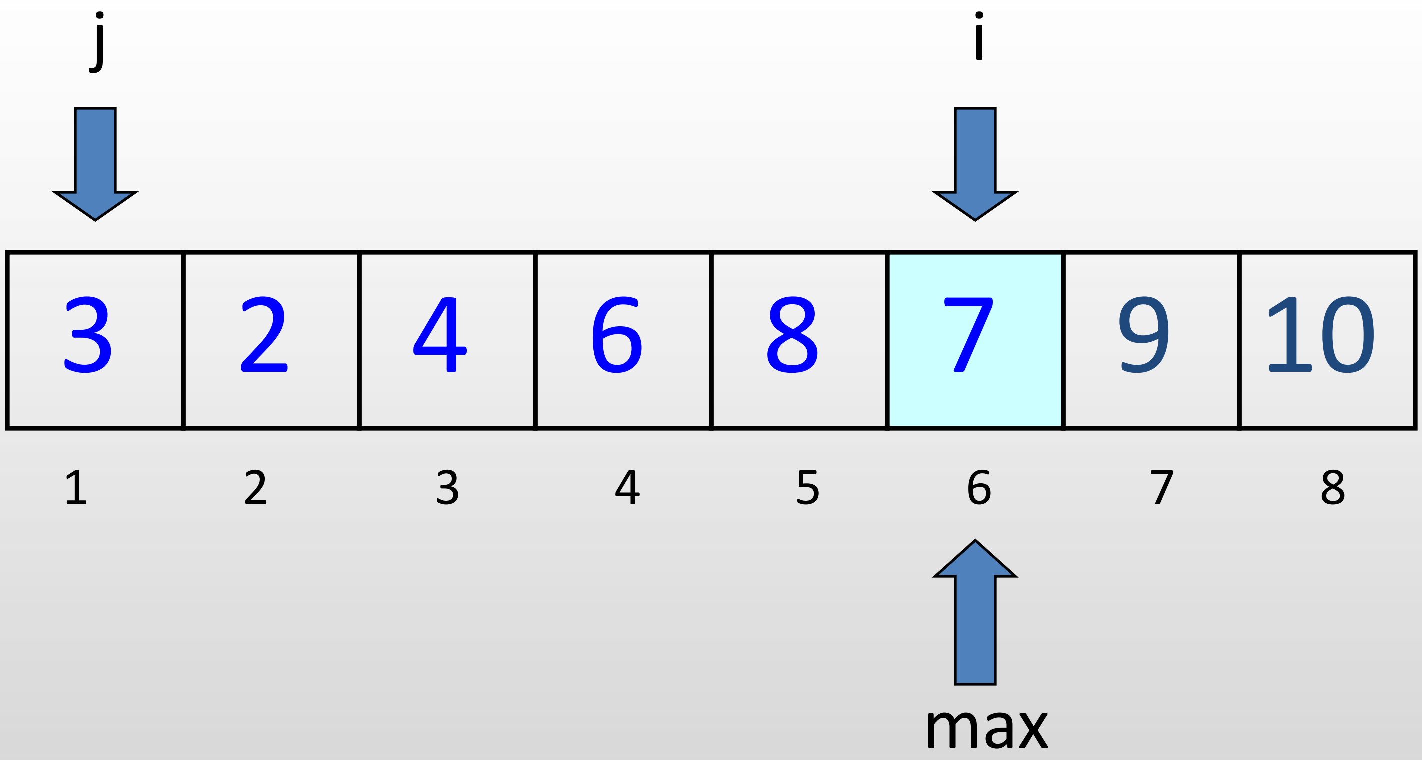
j = {1,2,3,4,5}

i

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

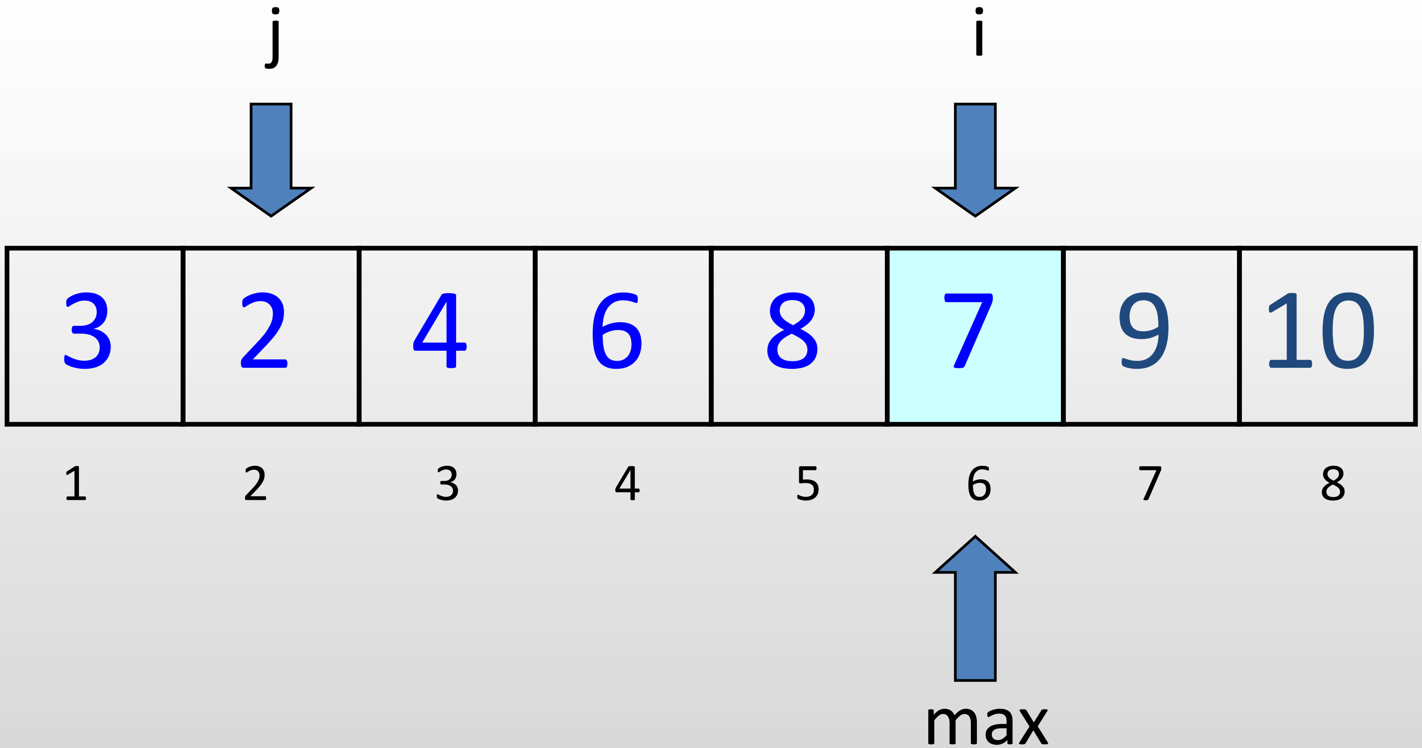
3rd pass: i=6

j = {1,2,3,4,5}



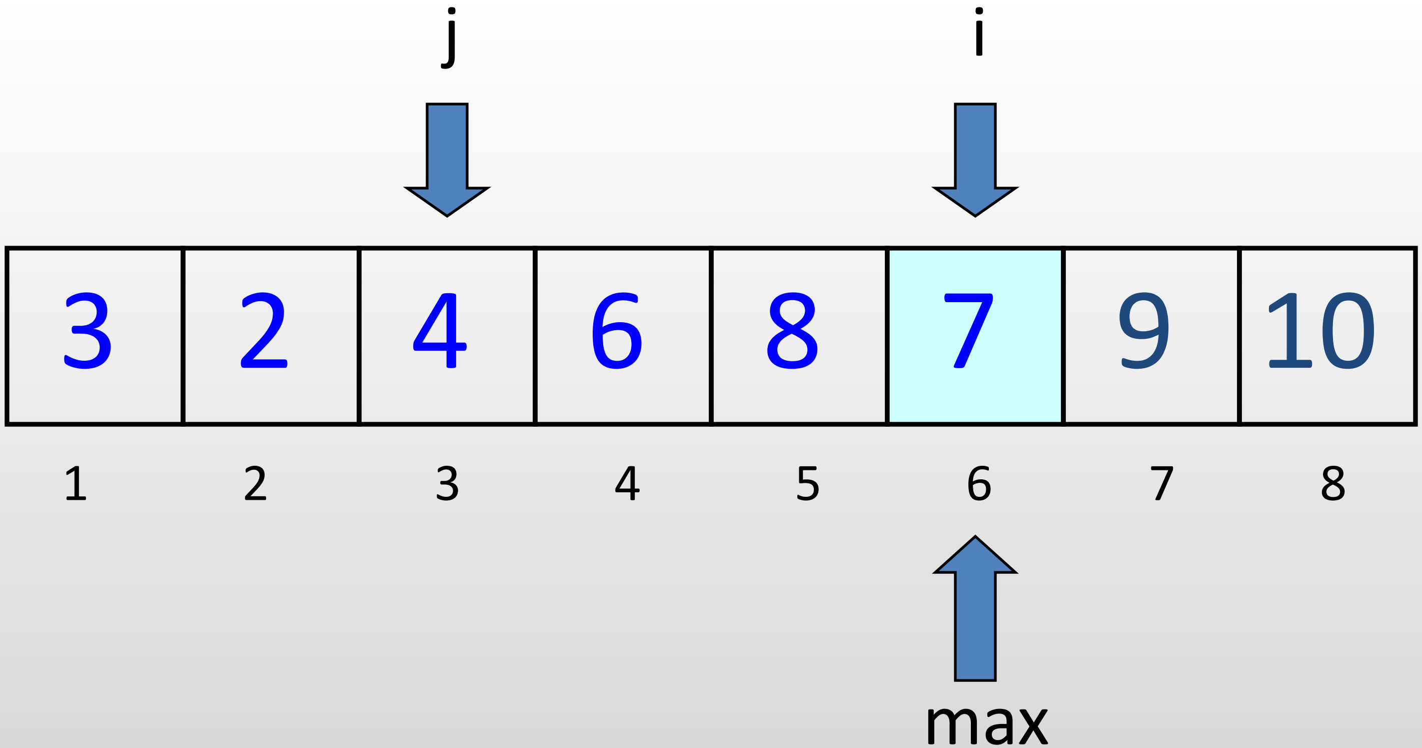
3rd pass: i=6

j = {1,2,3,4,5}



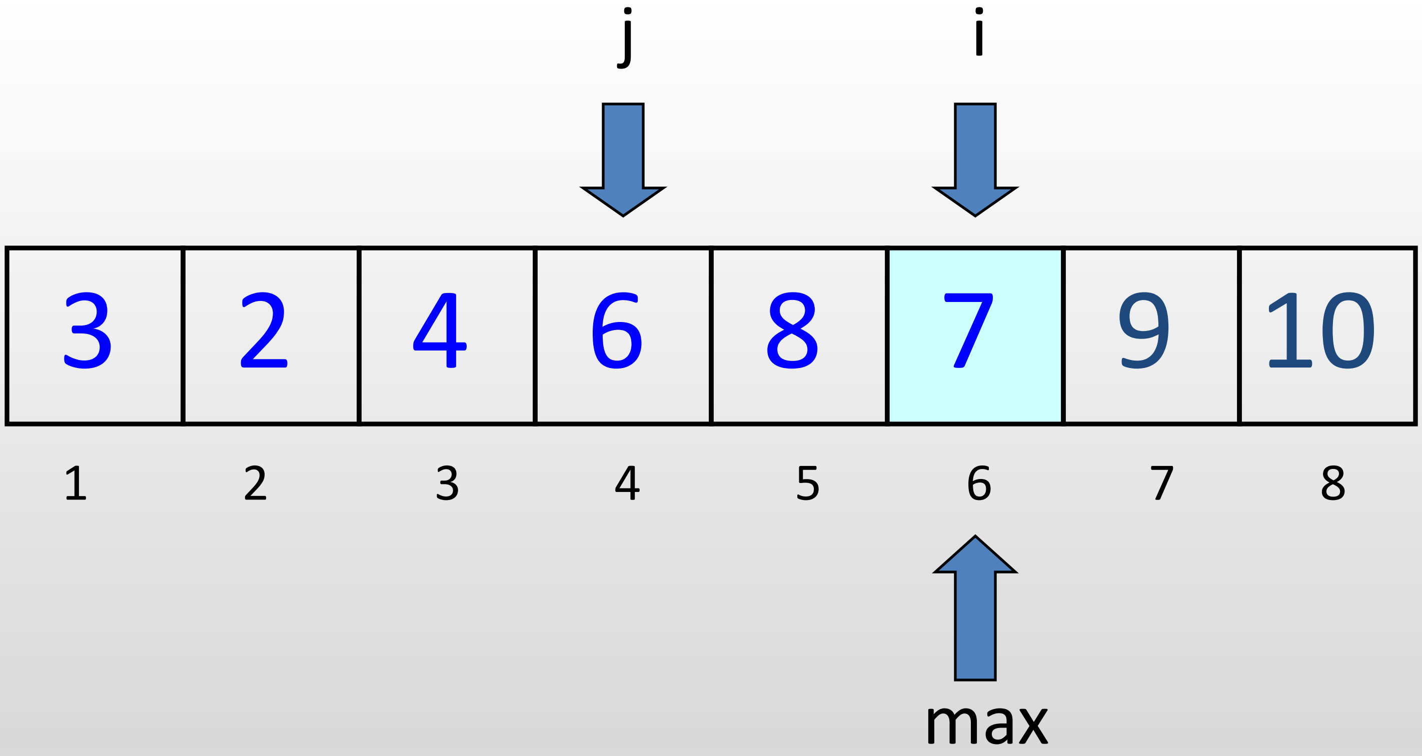
3rd pass: i=6

j = {1,2,3,4,5}



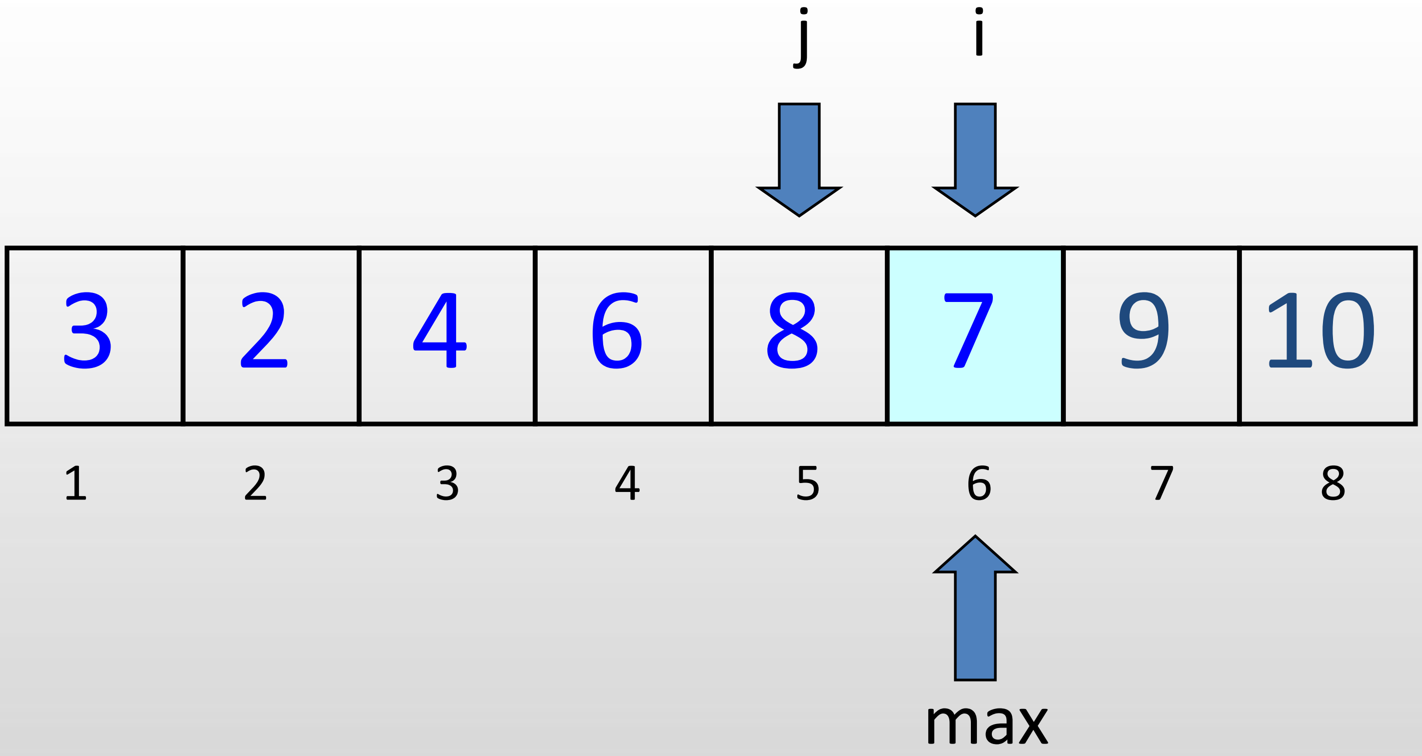
3rd pass: i=6

j = {1,2,3,4,5}



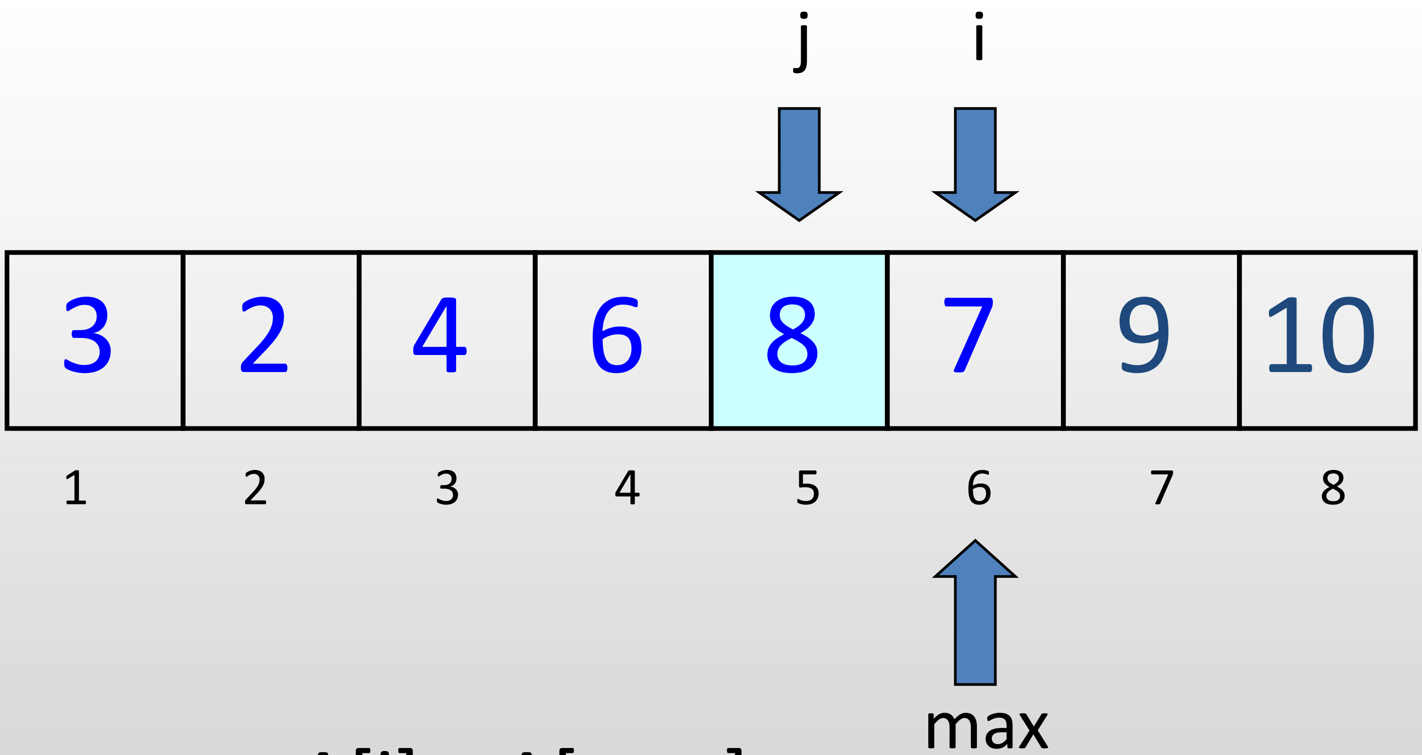
3rd pass: i=6

j = {1,2,3,4,5}



3rd pass: i=6

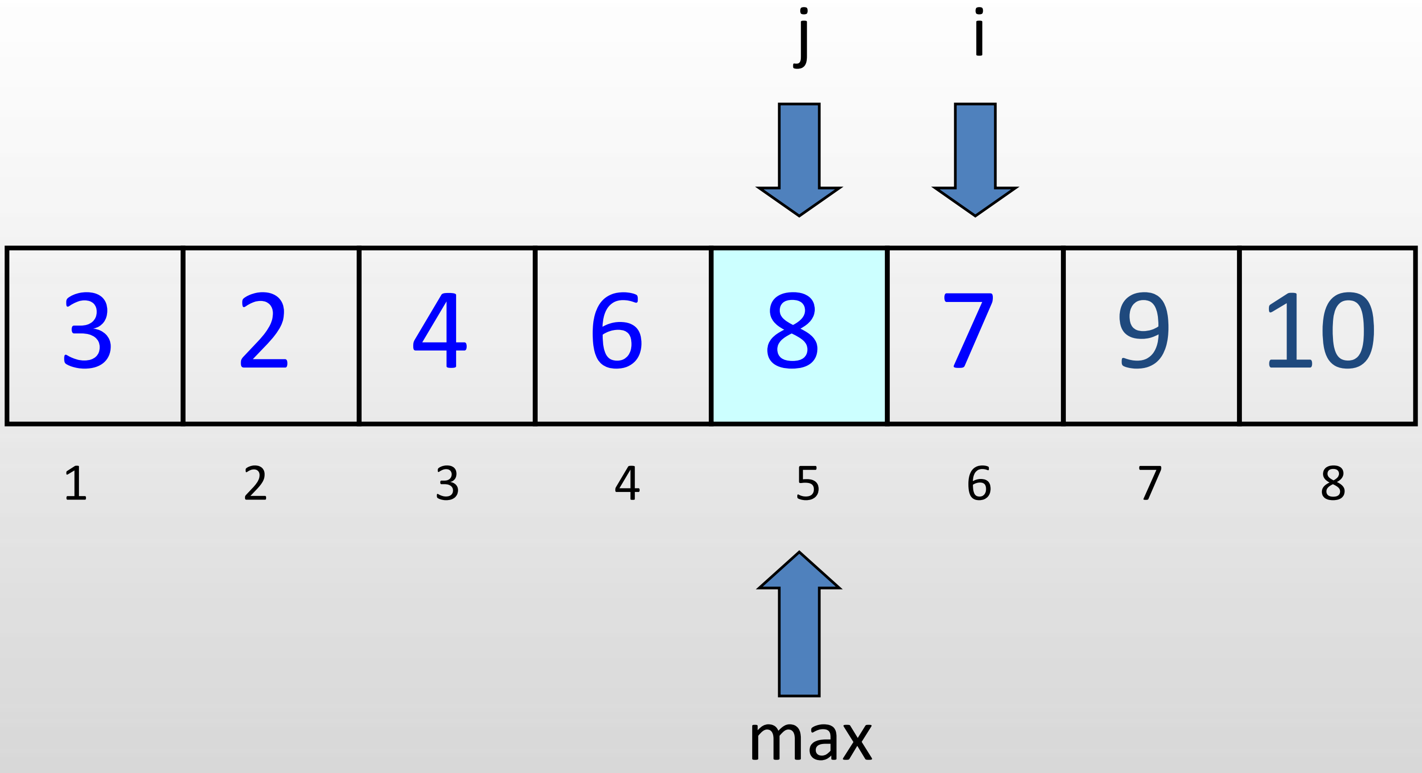
j = {1,2,3,4,5}



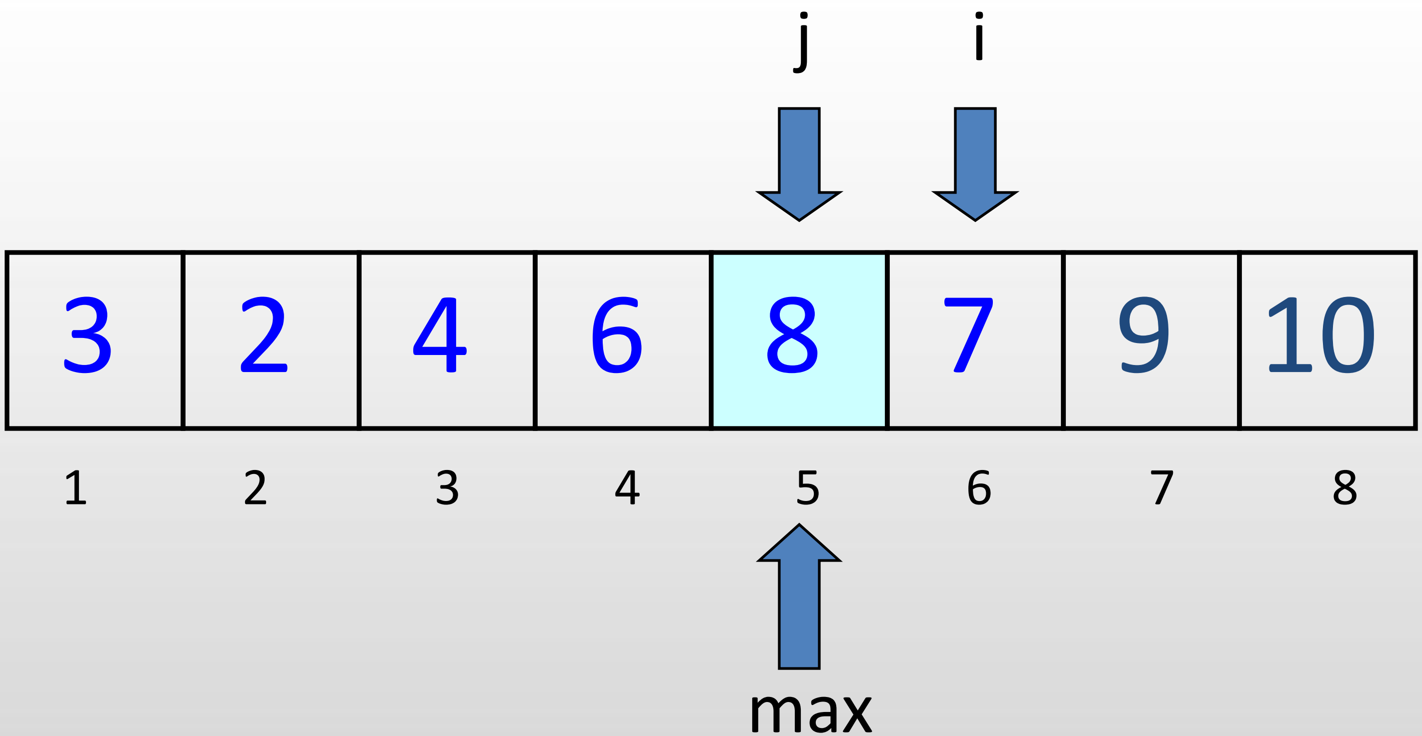
A[j] > A[max]

3rd pass: i=6

j = {1,2,3,4,5}



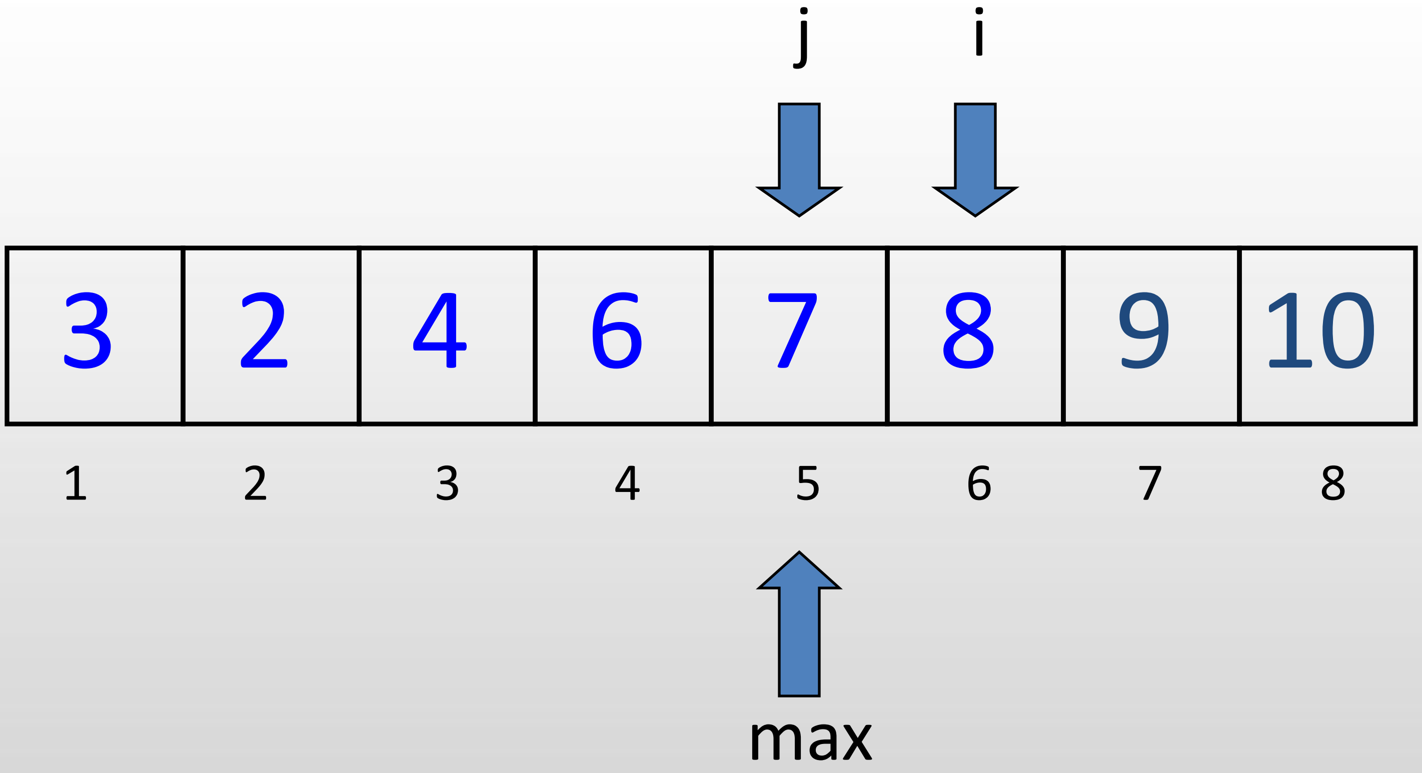
3rd pass: i=6
j = {1,2,3,4,5}



swap(A[i],A[max])

3rd pass: i=6

j = {1,2,3,4,5}



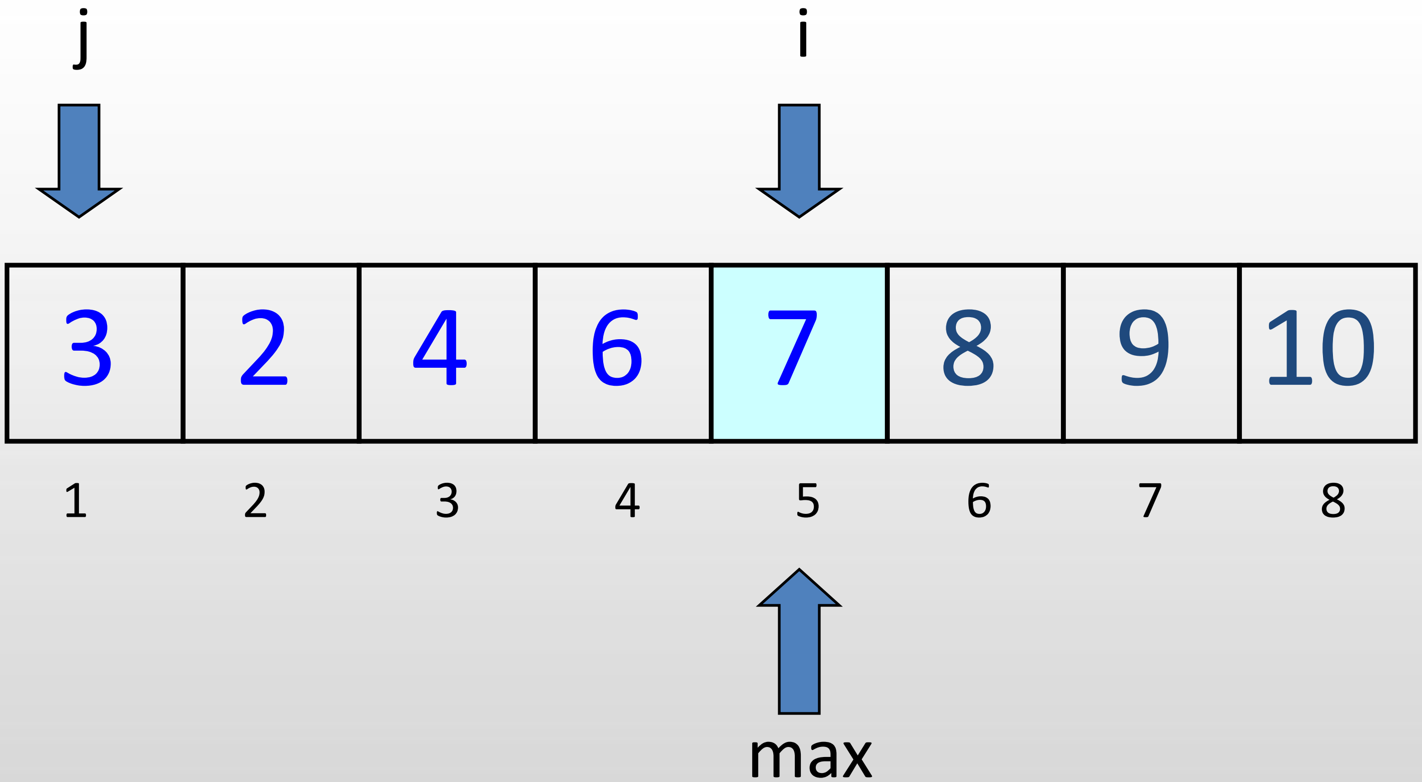
4th pass: i=5
j = {1,2,3,4}

i

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

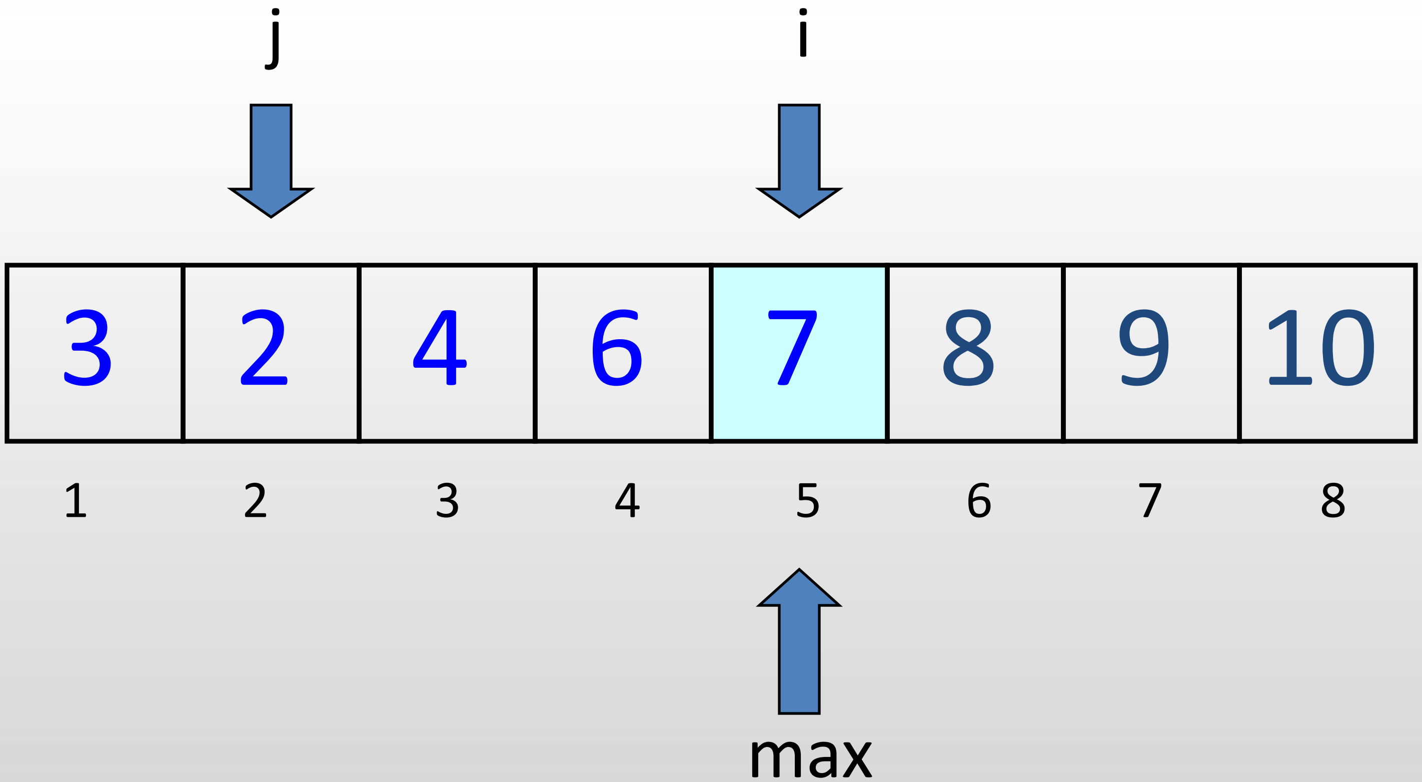
4th pass: i=5

j = {1,2,3,4}

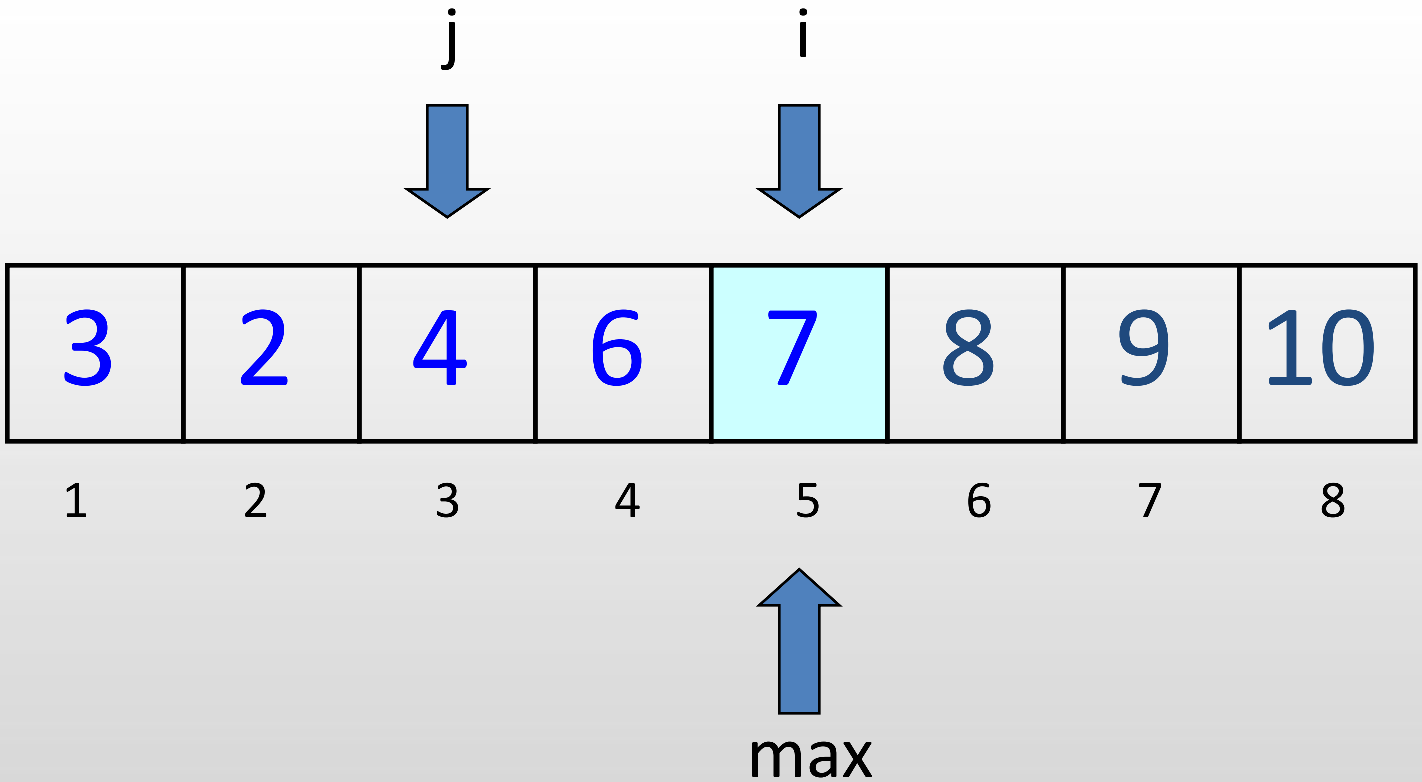


4th pass: i=5

j = {1,2,3,4}

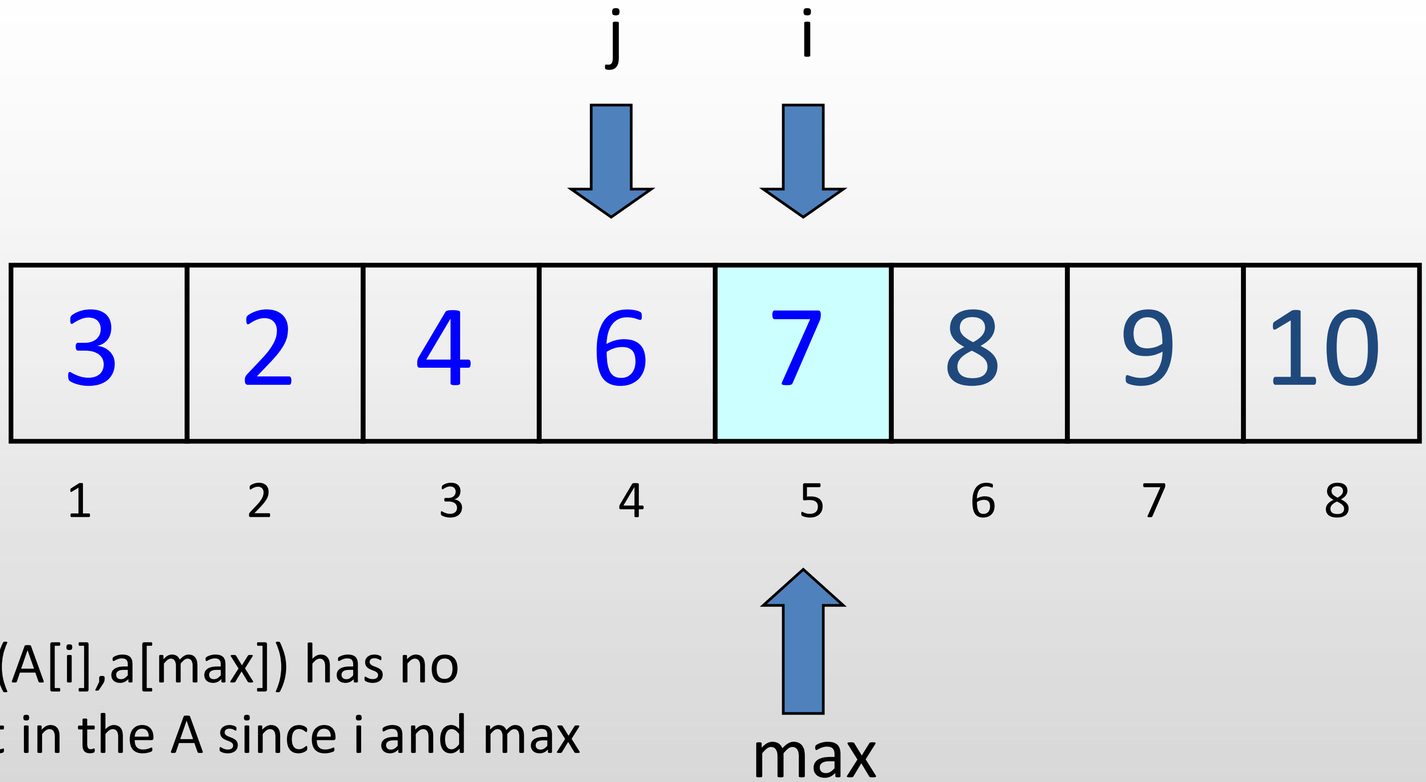


4th pass: i=5
j = {1,2,3,4}



4th pass: i=5

j = {1,2,3,4}



swap(A[i],a[max]) has no effect in the A since i and max have the same value.

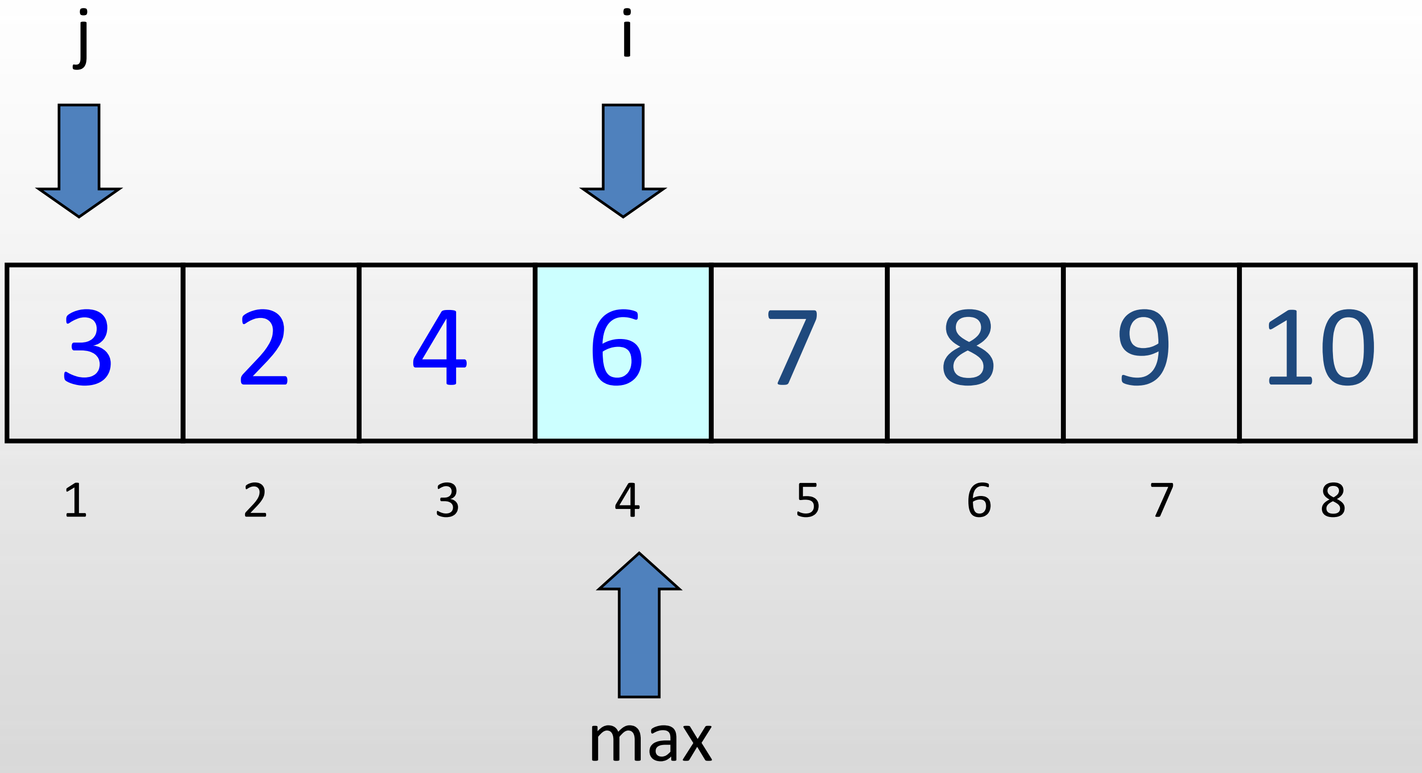
5th pass: i=4
j = {1,2,3}

i

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

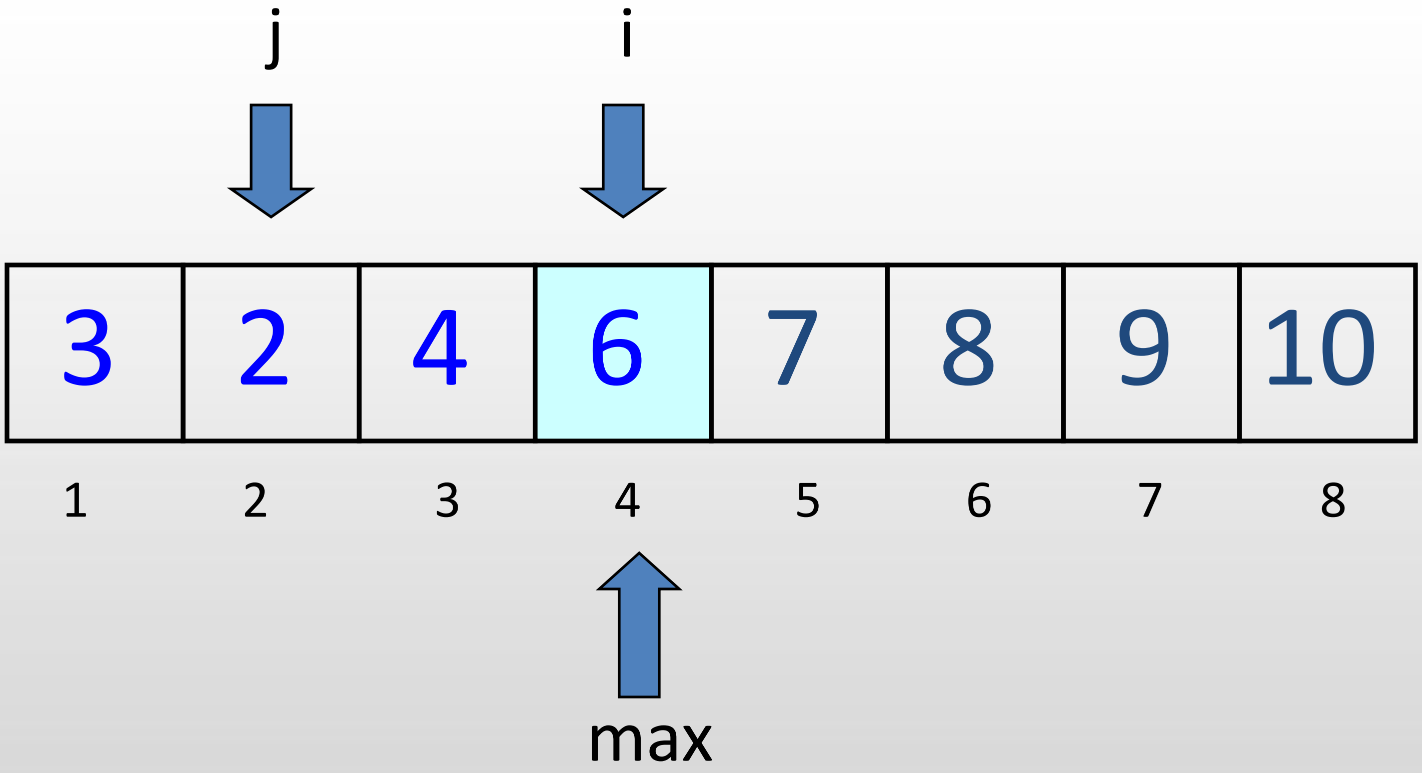
5th pass: i=4

j = {1,2,3}



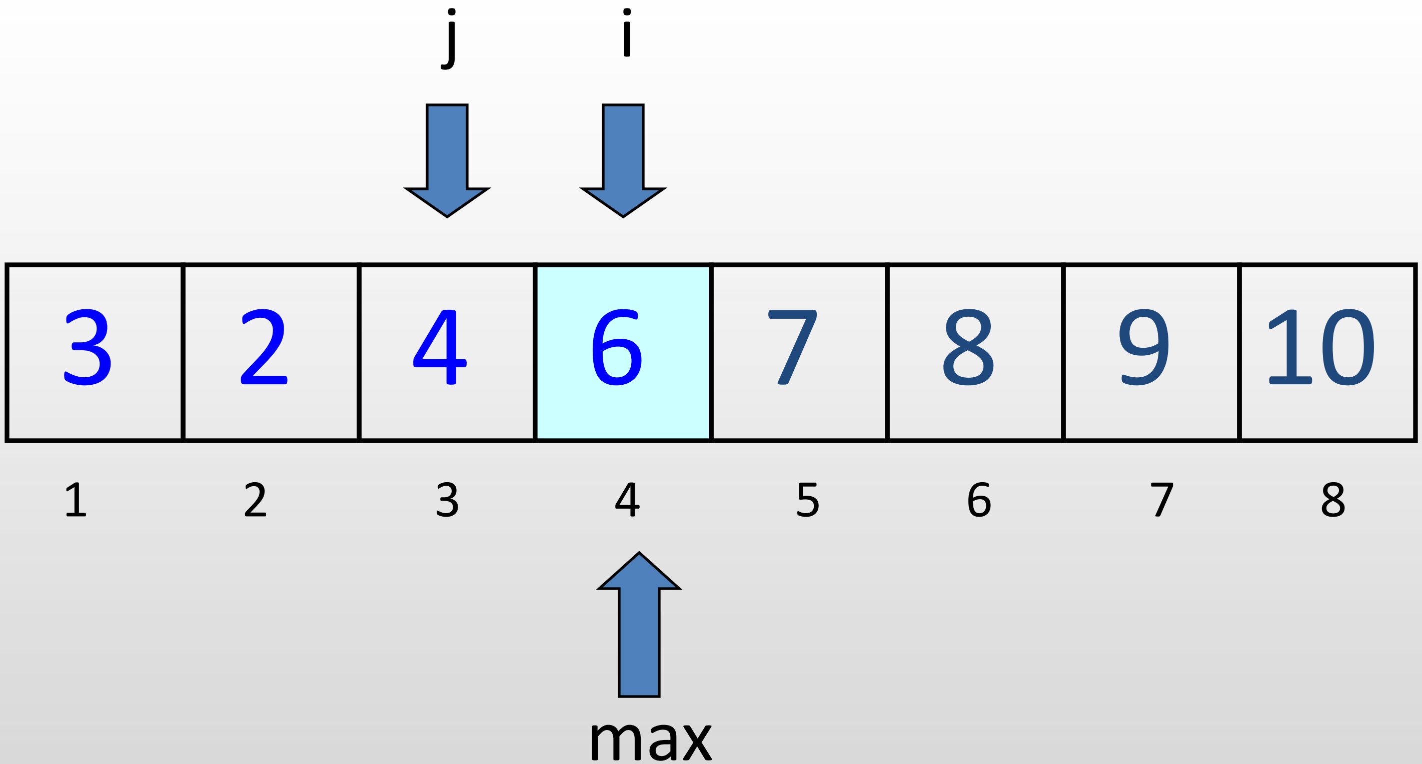
5th pass: i=4

$j = \{1,2,3\}$



5th pass: i=4

j = {1,2,3}



6th pass: i=3

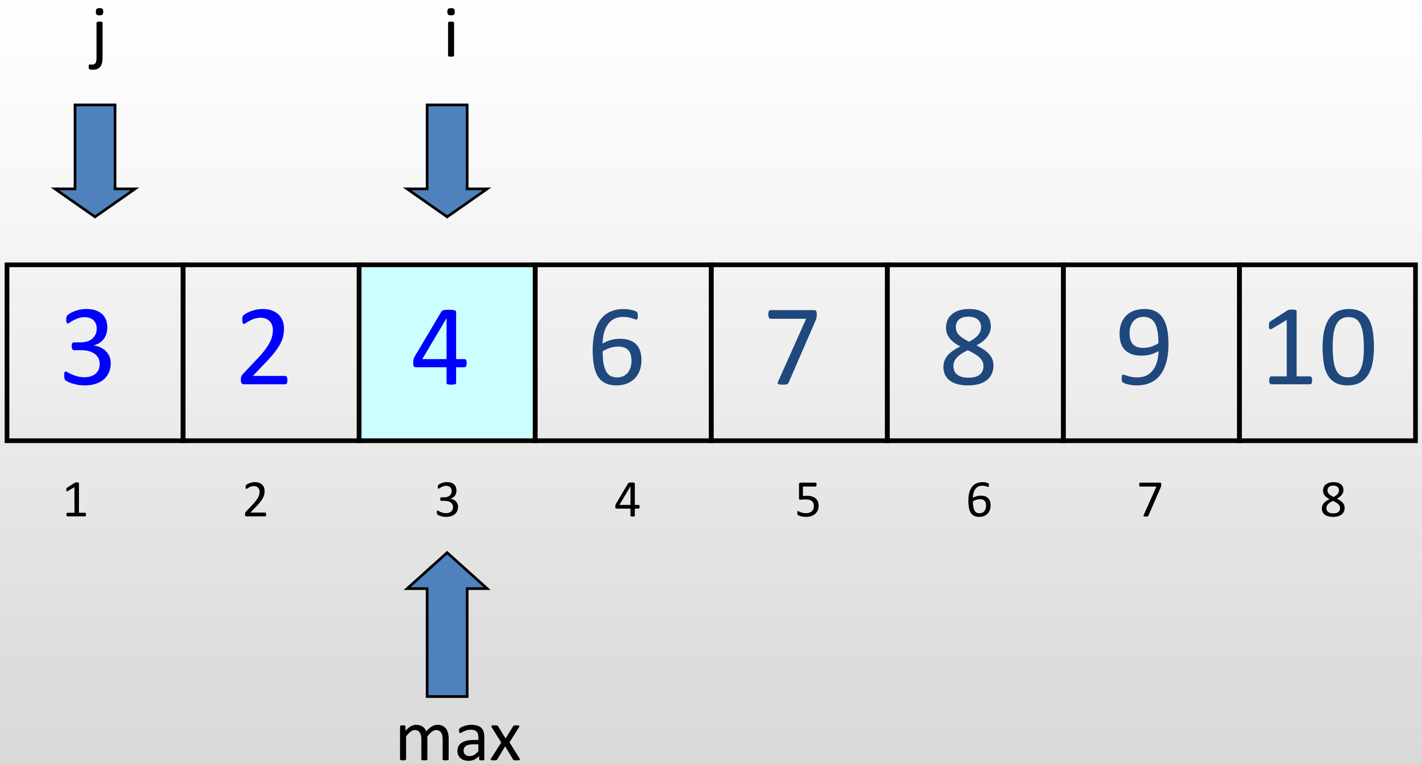
j = {1,2}

i

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

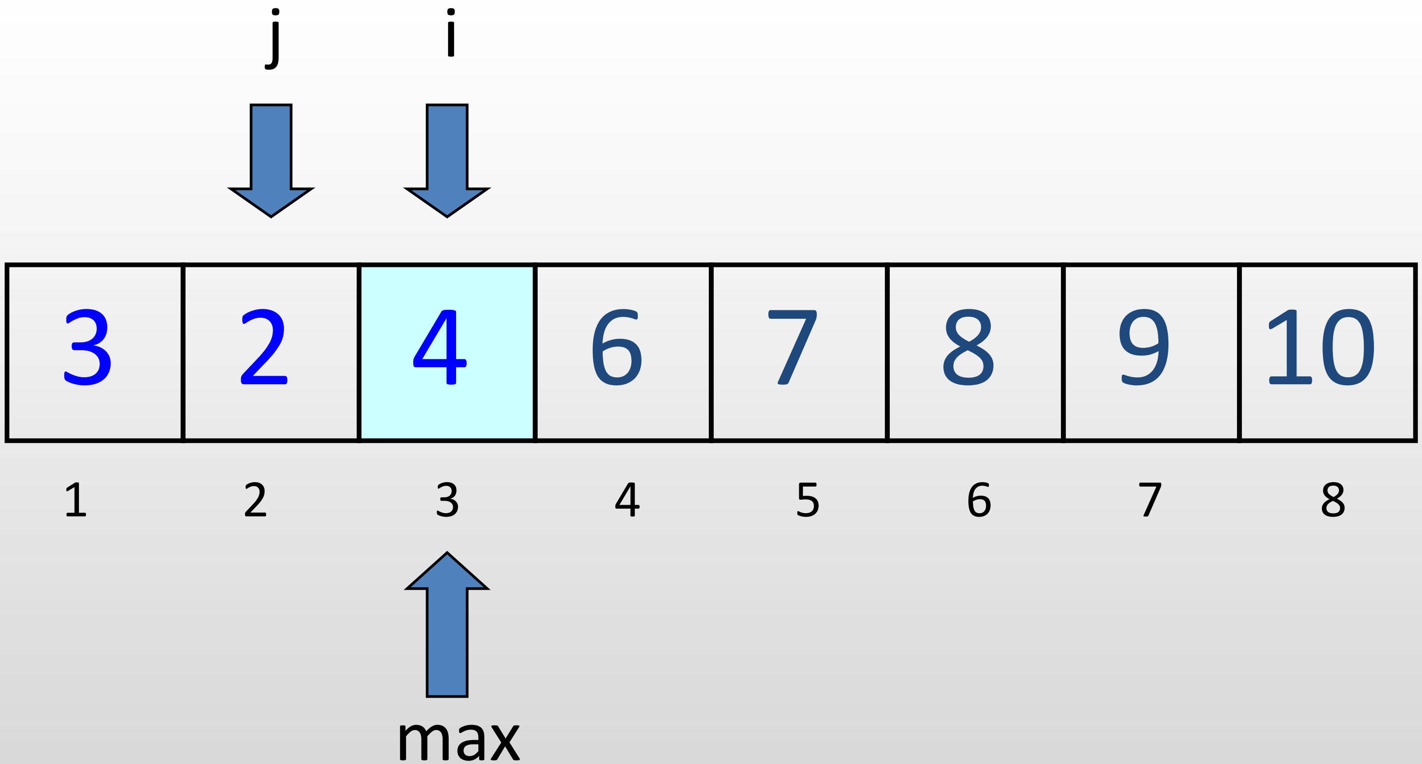
6th pass: i=3

j = {1,2}



6th pass: i=3

j = {1,2}

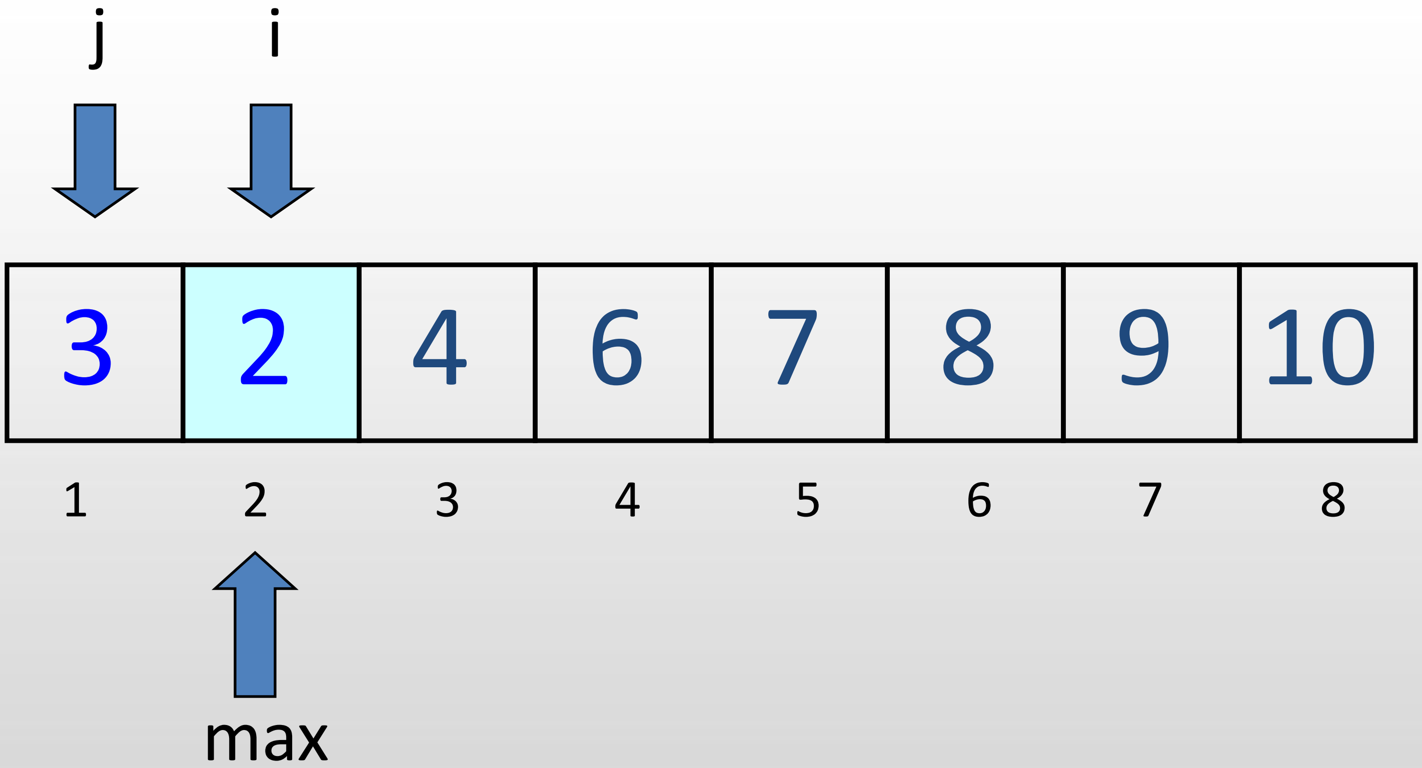


final pass: i=2
j = {1}

i

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

final pass: i=2
j = {1}



final pass: i=2

j = {1}

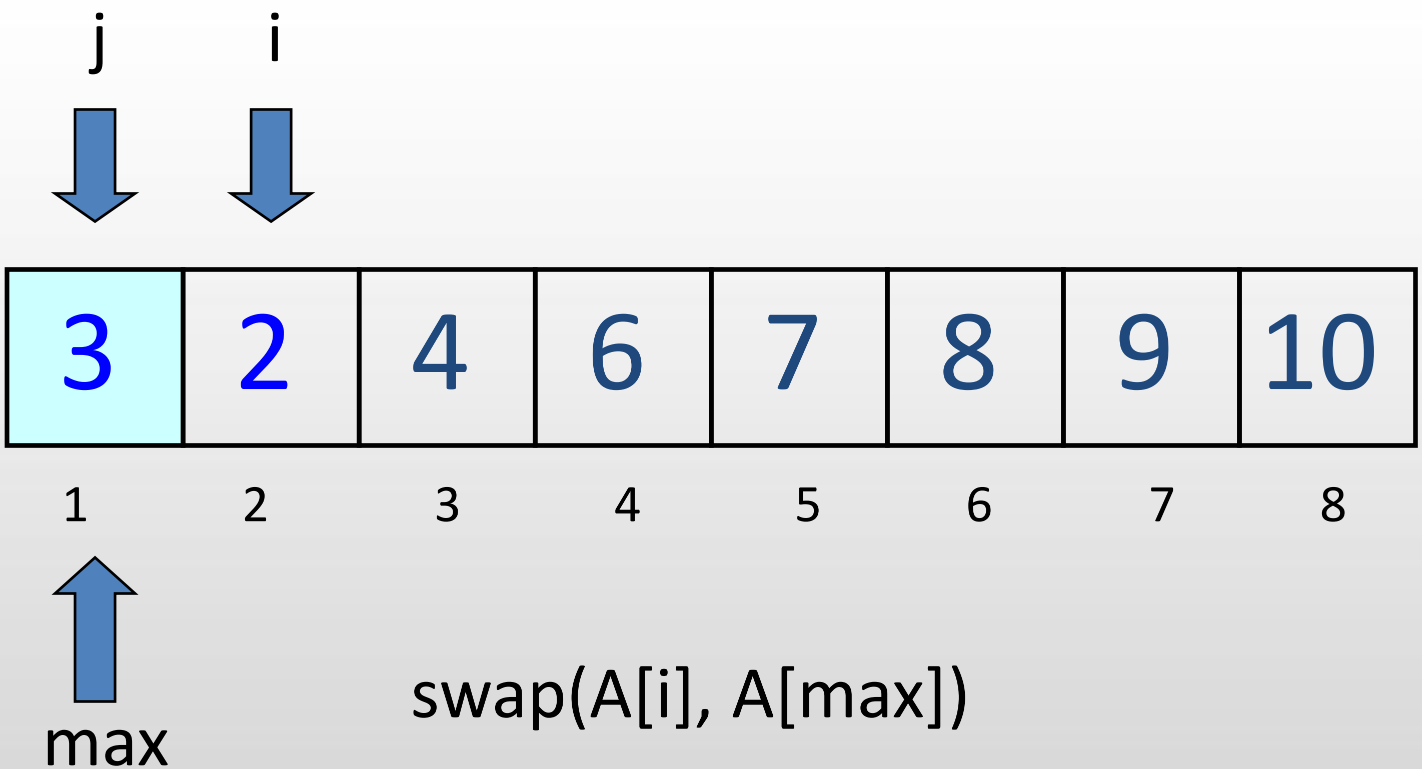
j i

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

max

A[j] > A[max]

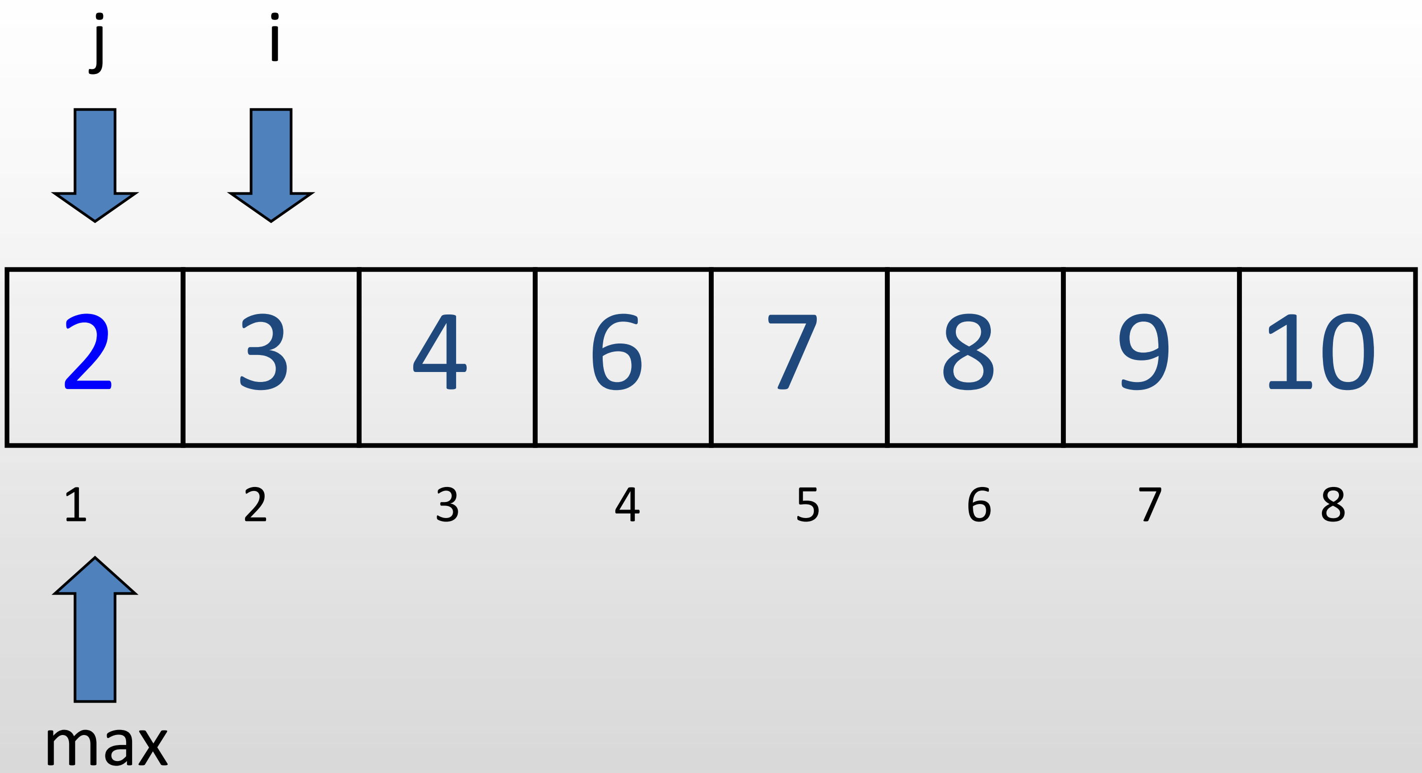
final pass: i=2
j = {1}



swap(A[i], A[max])

final pass: i=2

j = {1}



Final array

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

Merge Sort

```
void merge_sort(int a[], int lower, int upper){
    int mid;

    if (upper-lower>0) {
        mid=(lower+upper)/2
        merge_sort(a, lower, mid);
        merge_sort(a, mid+1, upper);
        merge(a, lower, mid, upper);
    }
}
```

Merge Sort

a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]
8	1	5	3	7	2	6	4

Merge Sort

a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]
8	1	5	3	7	2	6	4

Merge Sort

a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]
8	1	5	3	7	2	6	4

8	1	5	3	7	2	6	4

Merge Sort

a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]
8	1	5	3	7	2	6	4
8	1	5	3	7	2	6	4
1	8	5	3	7	2	6	4

→merge←

Merge Sort

a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]
8	1	5	3	7	2	6	4

8	1	5	3	7	2	6	4

1	8	5	3	7	2	6	4

→merge←

1	8	3	5	7	2	6	4

→merge←

Merge Sort

a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]
8	1	5	3	7	2	6	4
8	1	5	3	7	2	6	4
1	8	5	3	7	2	6	4

→merge←

1	8	3	5	7	2	6	4

→merge←

1	3	5	8	7	2	6	4

→ merge ←

Merge Sort

a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]
1	3	5	8	7	2	6	4
1	3	5	8	2	7	6	4

→merge←

1	3	5	8	2	7	4	6

→merge←

1	3	5	8	2	4	6	7

→ merge ←

1	2	3	4	5	6	7	8

→ merge ←

Merge

```
void merge (int a[], int lower, int mid, int upper){
    int *temp,i,j,k;

    temp=(int *)malloc((upper-lower+1)*sizeof(int));

    for(i=0,j=lower,k=mid+1; j<=mid || k<=upper; i++)
        temp[i]=(j<=mid && (k>upper || a[j]<a[k]))?
                         a[j++]:a[k++];
    for(i=0,j=lower;j<=upper; i++, j++)
        a[j]=temp[i];

    free(temp);
}
```

Merge

	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	
a	1	3	5	8	2	4	6	7	j++
temp	1								i=0
a	1	3	5	8	2	4	6	7	k++
temp	1	2							i=1
a	1	3	5	8	2	4	6	7	j++
temp	1	2	3						i=2
a	1	3	5	8	2	4	6	7	k++
temp	1	2	3	4					i=3

Merge

	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	
a	1	3	5	8	2	4	6	7	j++
temp	1	2	3	4	5				i=4

a	1	3	5	8	2	4	6	7	k++
temp	1	2	3	4	5	6			i=5

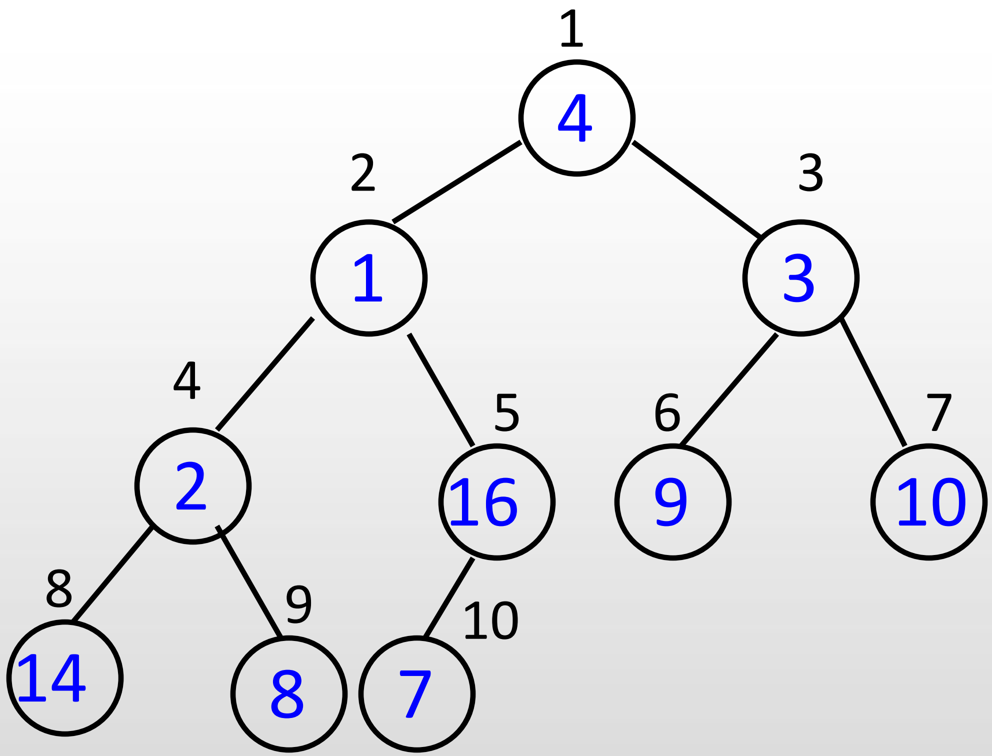
a	1	3	5	8	2	4	6	7	k++
temp	1	2	3	4	5	6	7		i=6

a	1	3	5	8	2	4	6	7	j++
temp	1	2	3	4	5	6	7	8	i=7

Heapsort

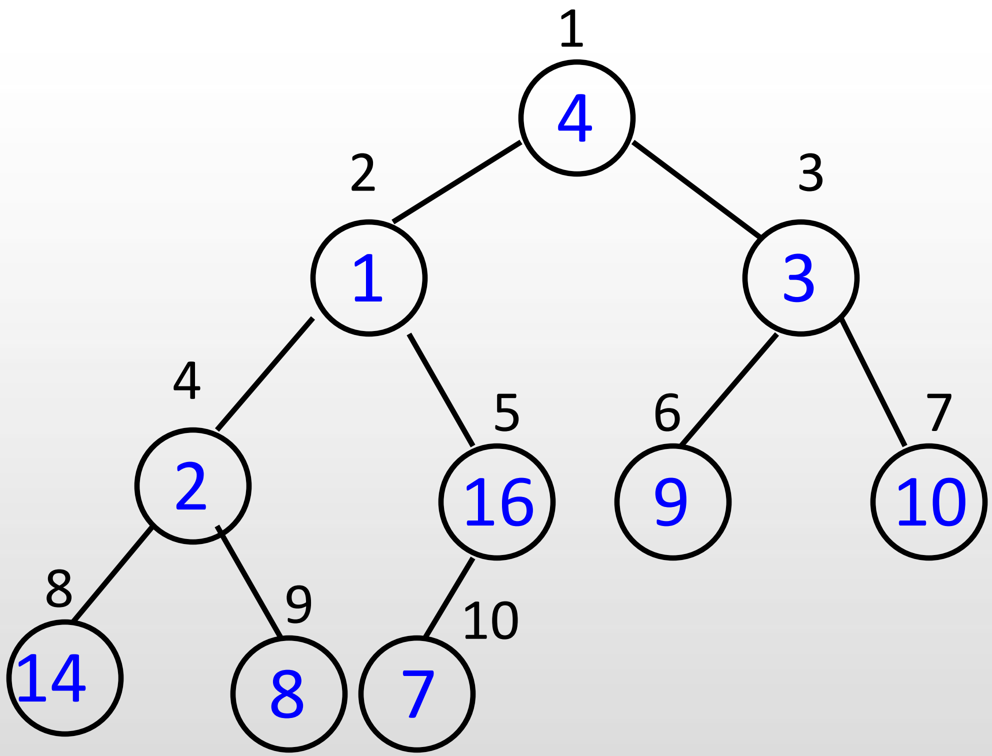
```
HeapSort(A,n)
 begin
   BuildHeap(A)
   for i=n downto 2 do
        swap(A[1],A[i])
        Heapify(A,1,(i-1))
end
```

Sort 4,1,3,2,16,9,10,14,8,7



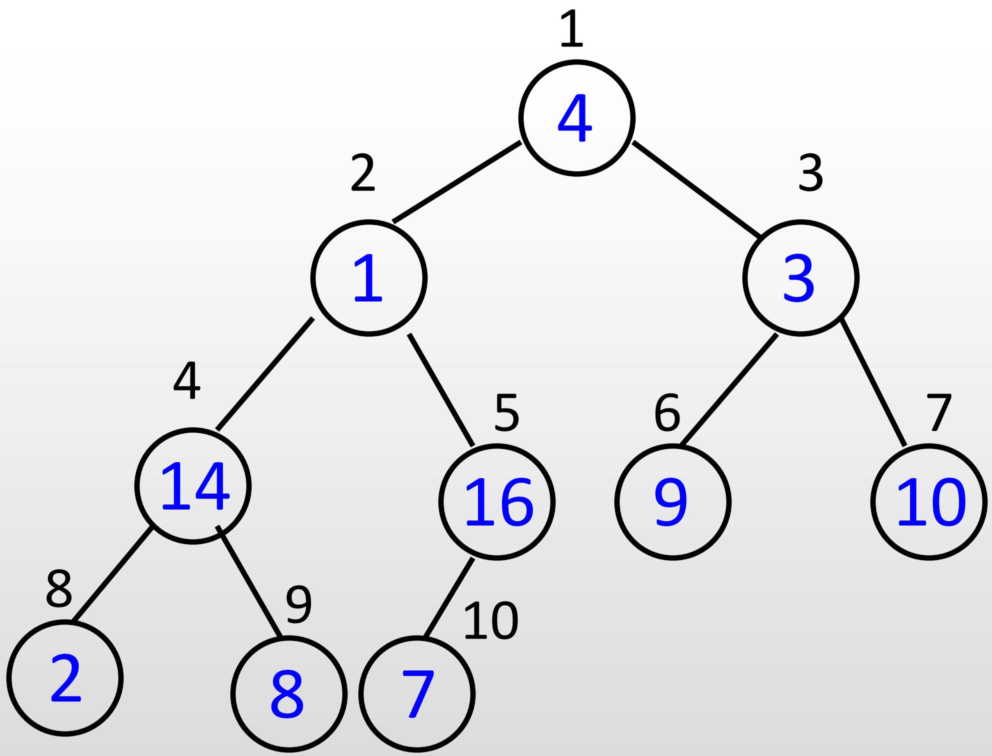
BUILDHEAP(A)

Sort 4,1,3,2,16,9,10,14,8,7



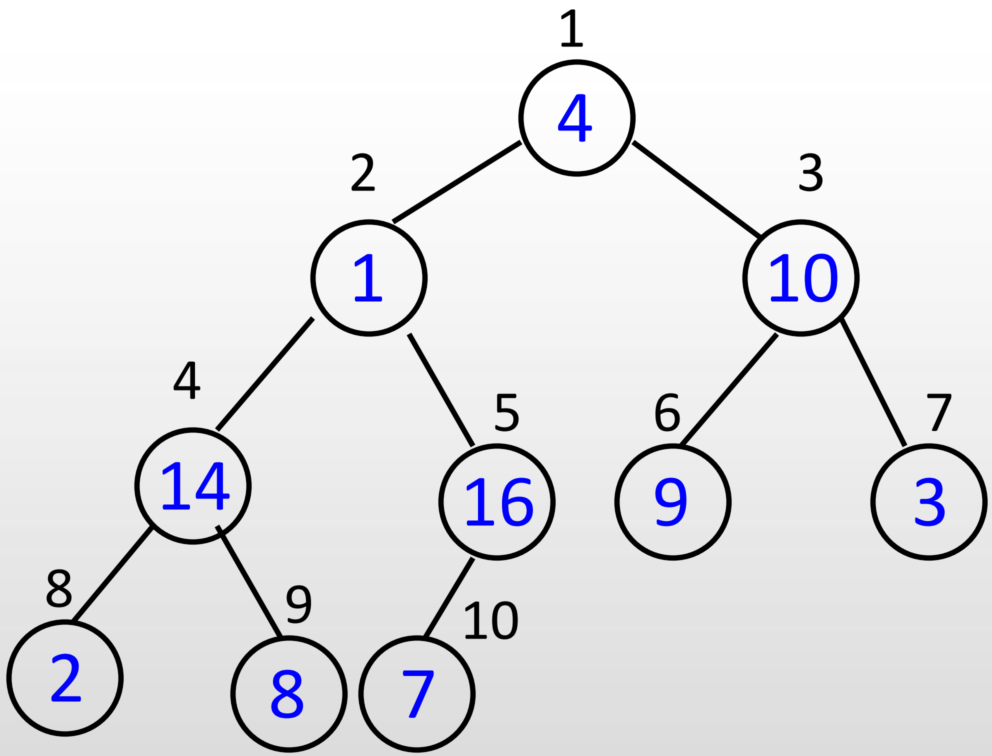
BUILDHEAP(A)

Sort 4,1,3,2,16,9,10,14,8,7



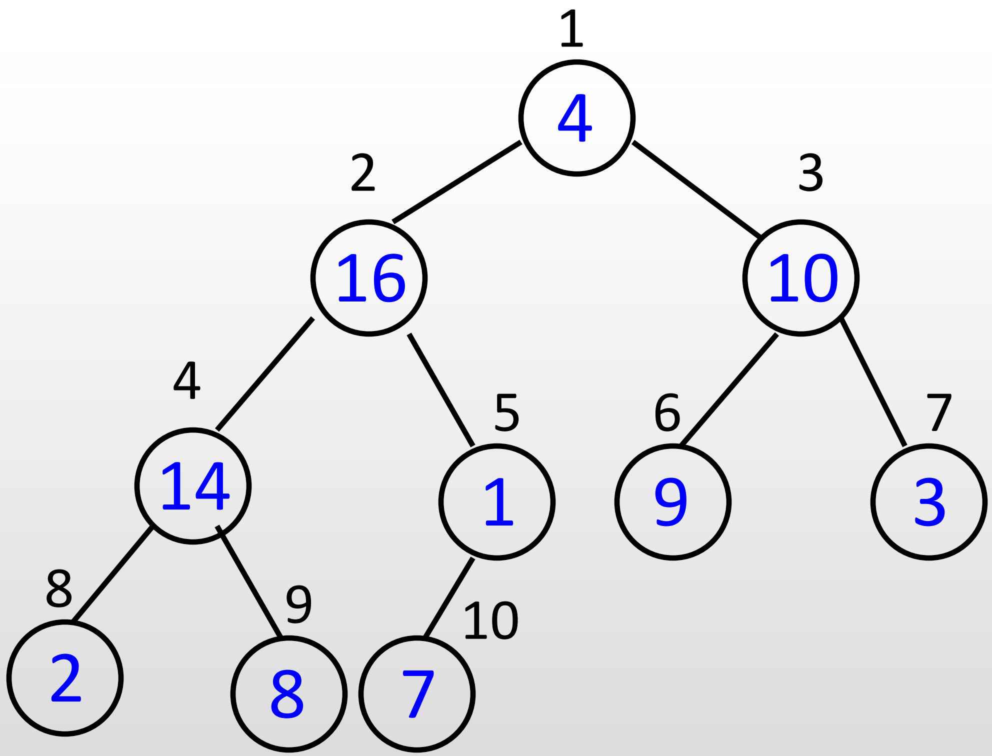
BUILDHEAP(A)

Sort 4,1,3,2,16,9,10,14,8,7



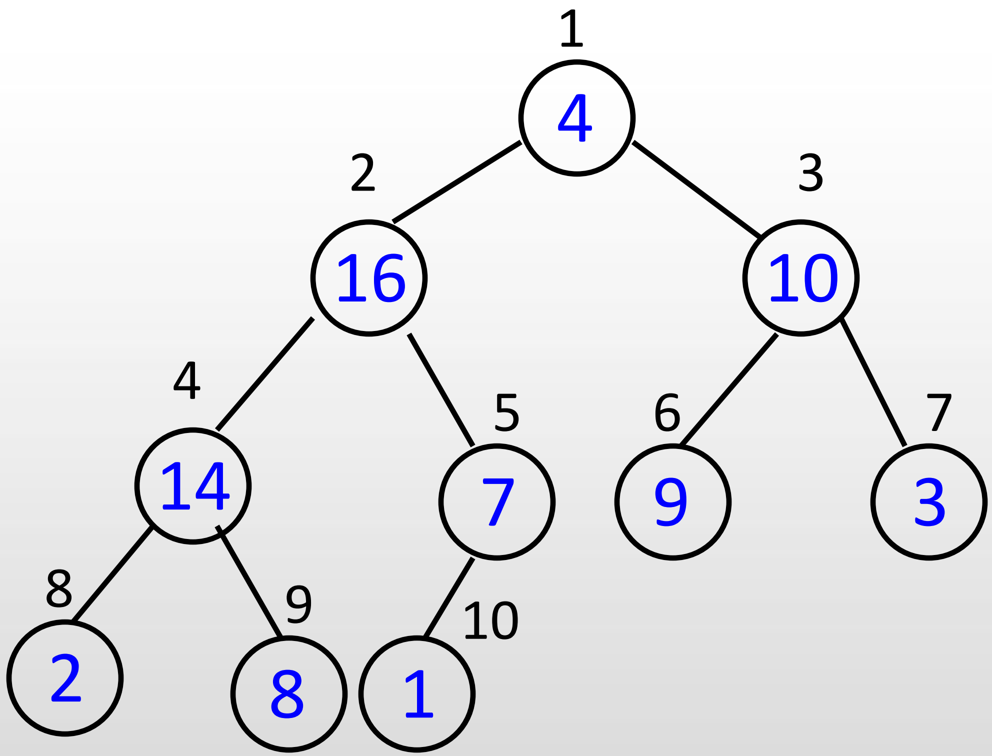
BUILDHEAP(A)

Sort 4,1,3,2,16,9,10,14,8,7



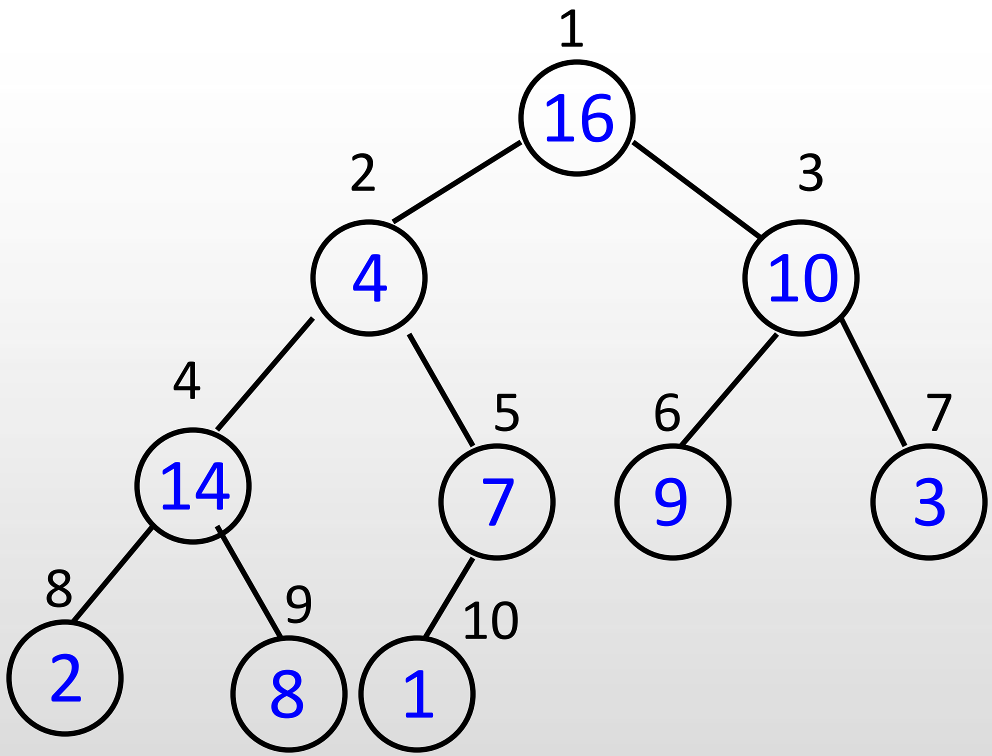
BUILDHEAP(A)

Sort 4,1,3,2,16,9,10,14,8,7



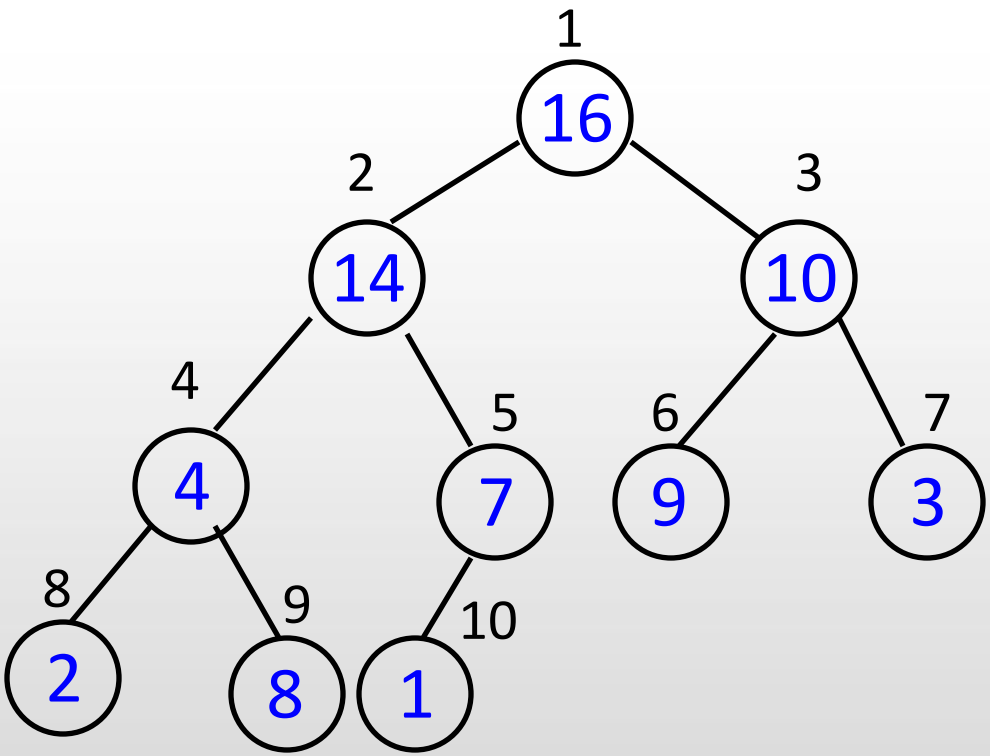
BUILDHEAP(A)

Sort 4,1,3,2,16,9,10,14,8,7



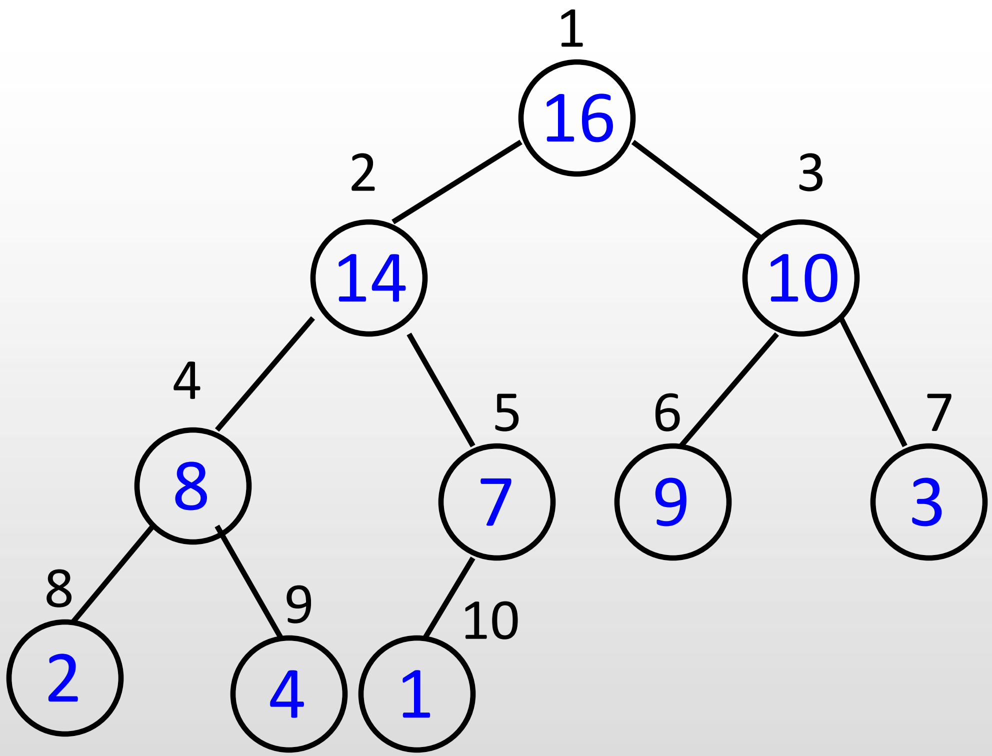
BUILDHEAP(A)

Sort 4,1,3,2,16,9,10,14,8,7

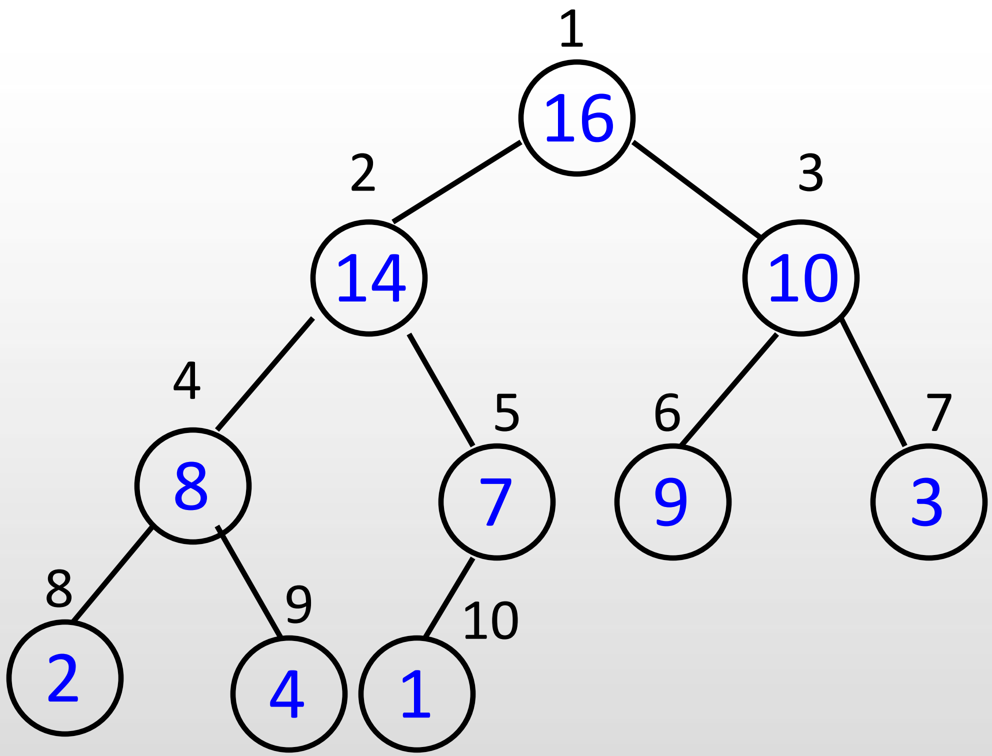


BUILDHEAP(A)

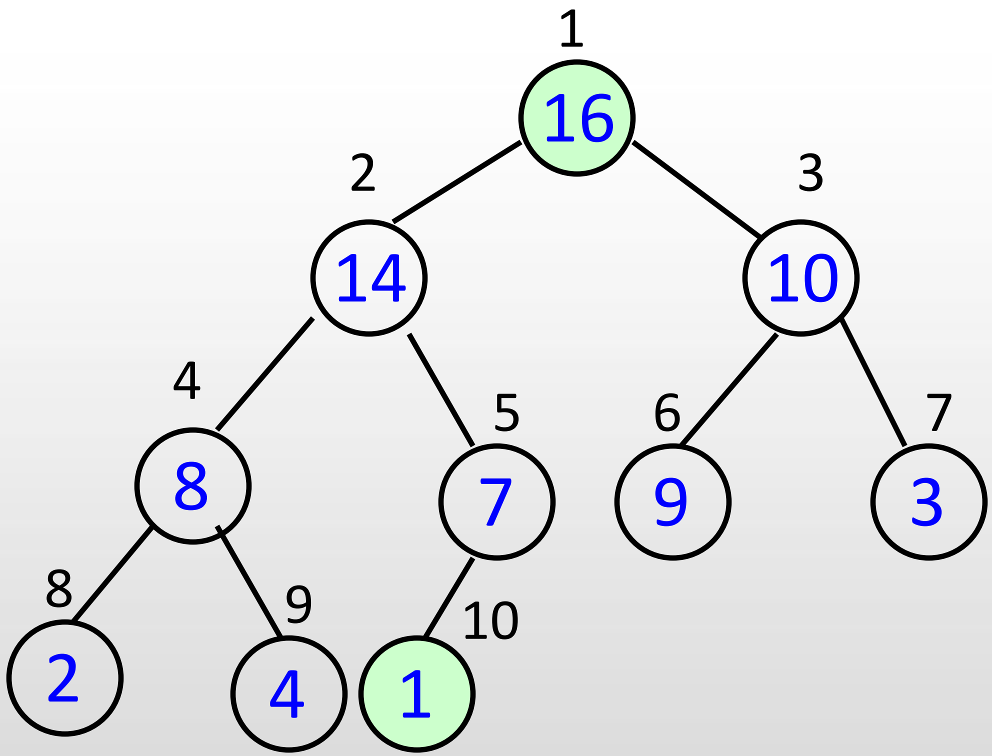
Resulting Max-Heap



1st pass: i=10

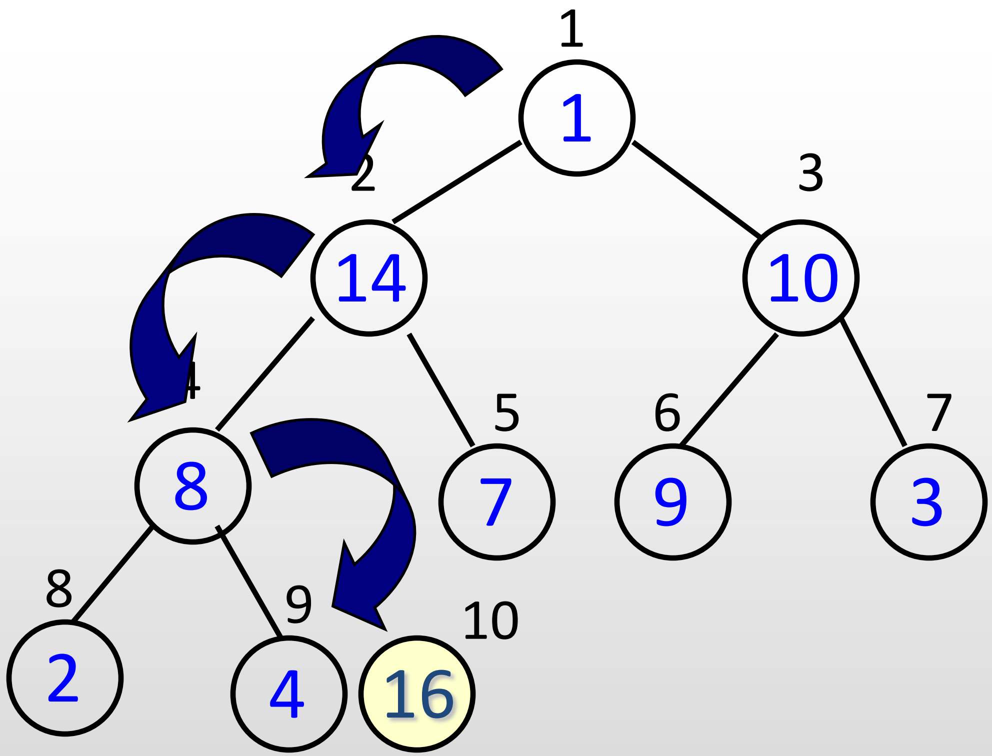


1st pass: i=10



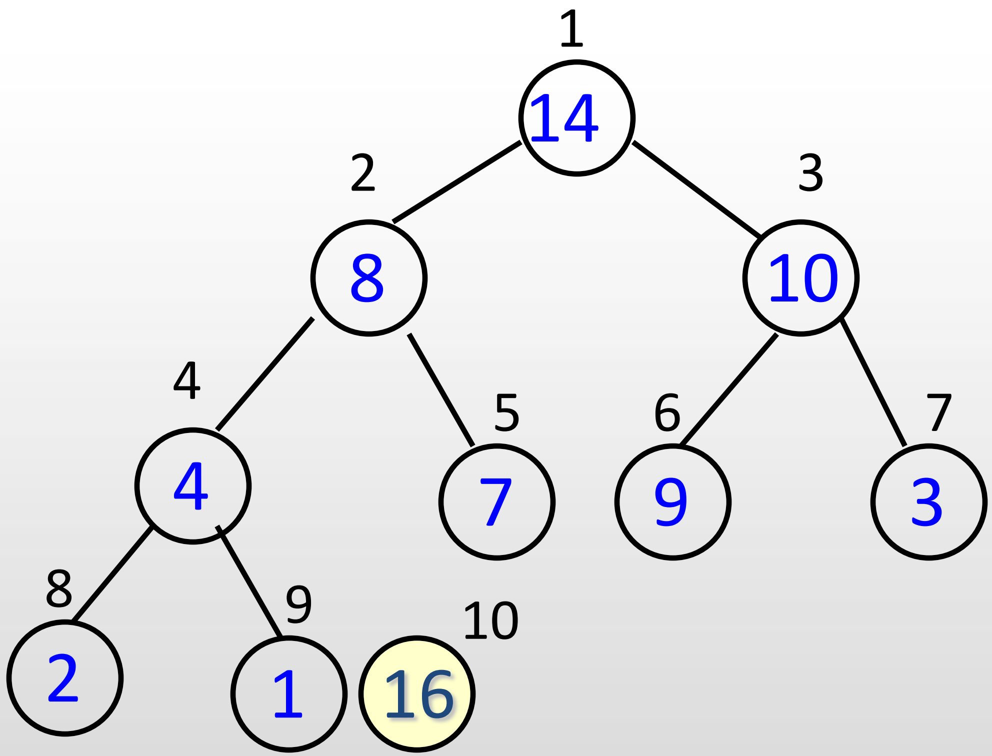
swap(A[1],A[i])

1st pass: i=10

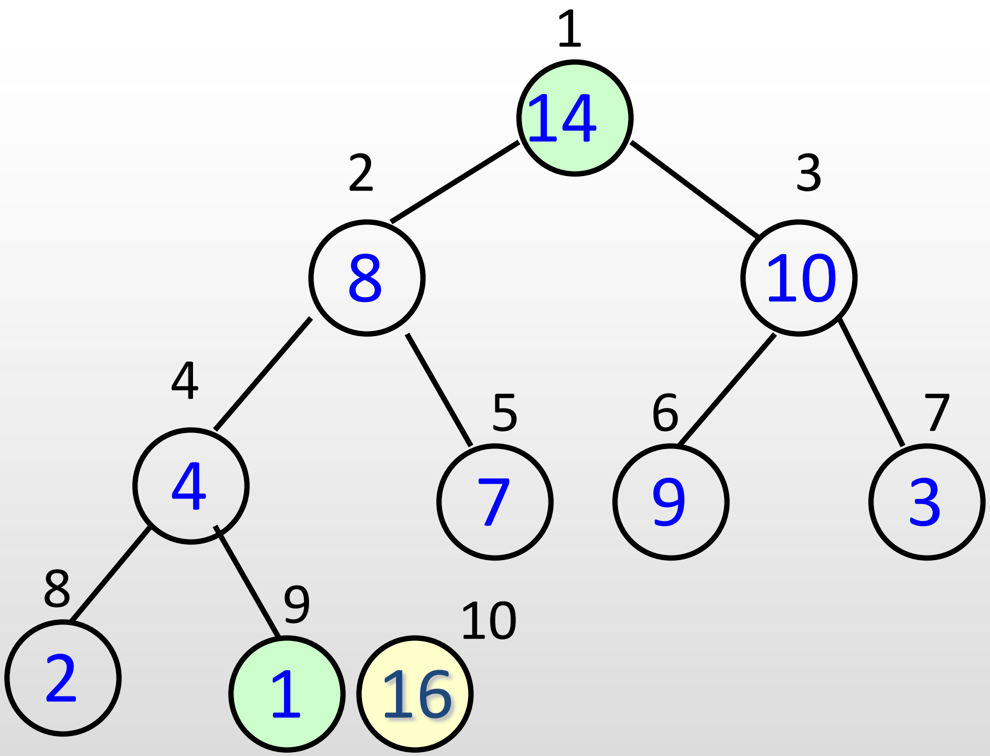


Heapify(A,1,9)

2nd pass: i=9

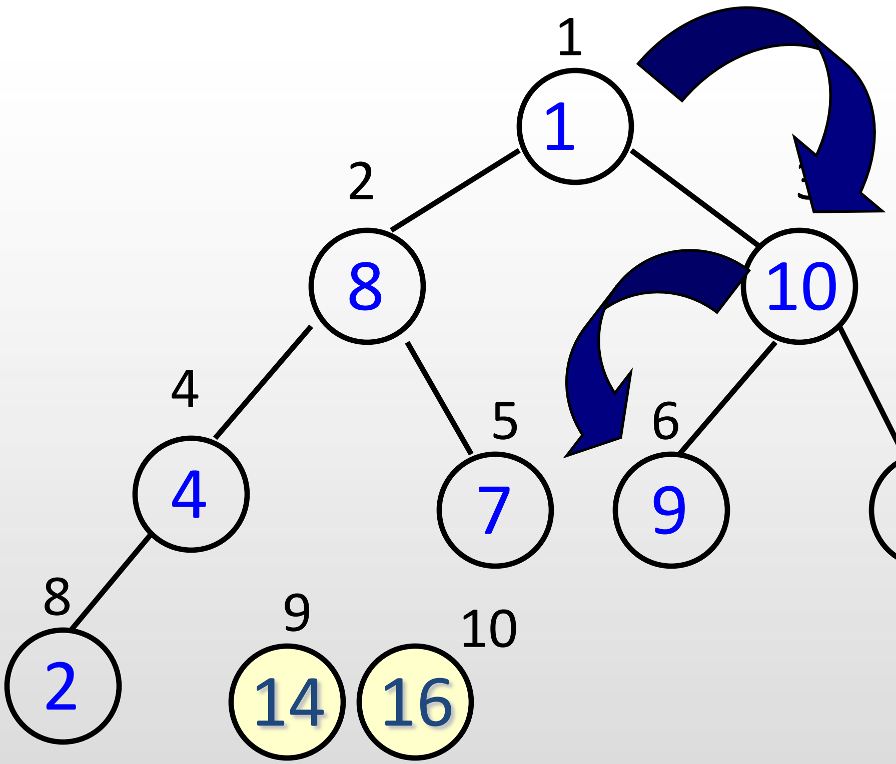


2nd pass: i=9



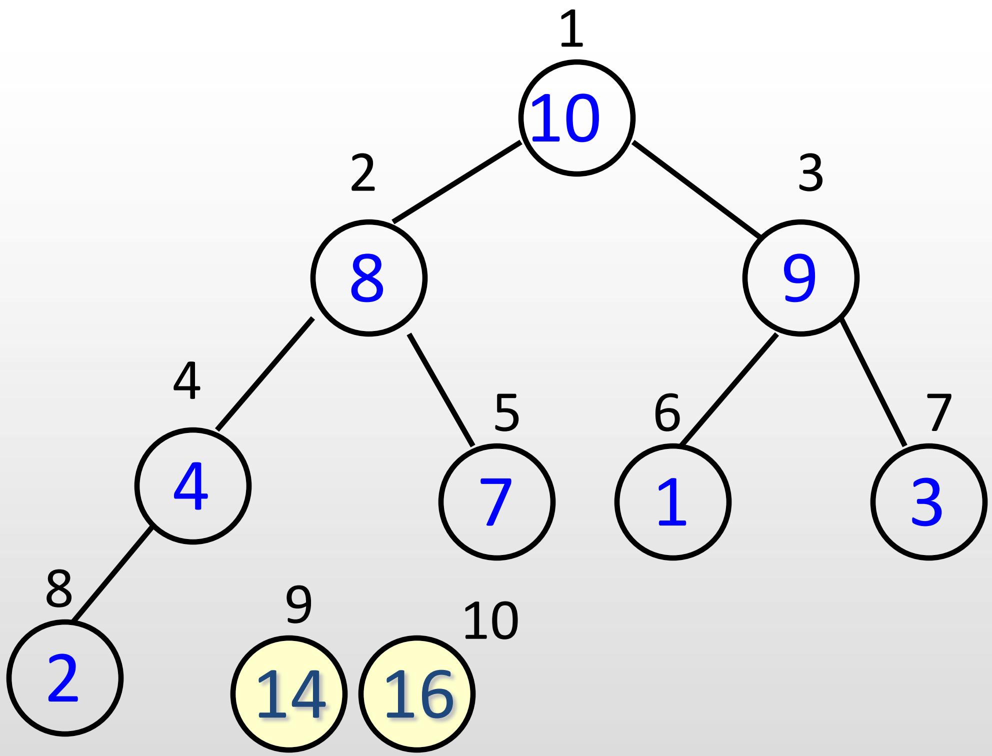
swap(A[1],A[9])

2nd pass: i=9

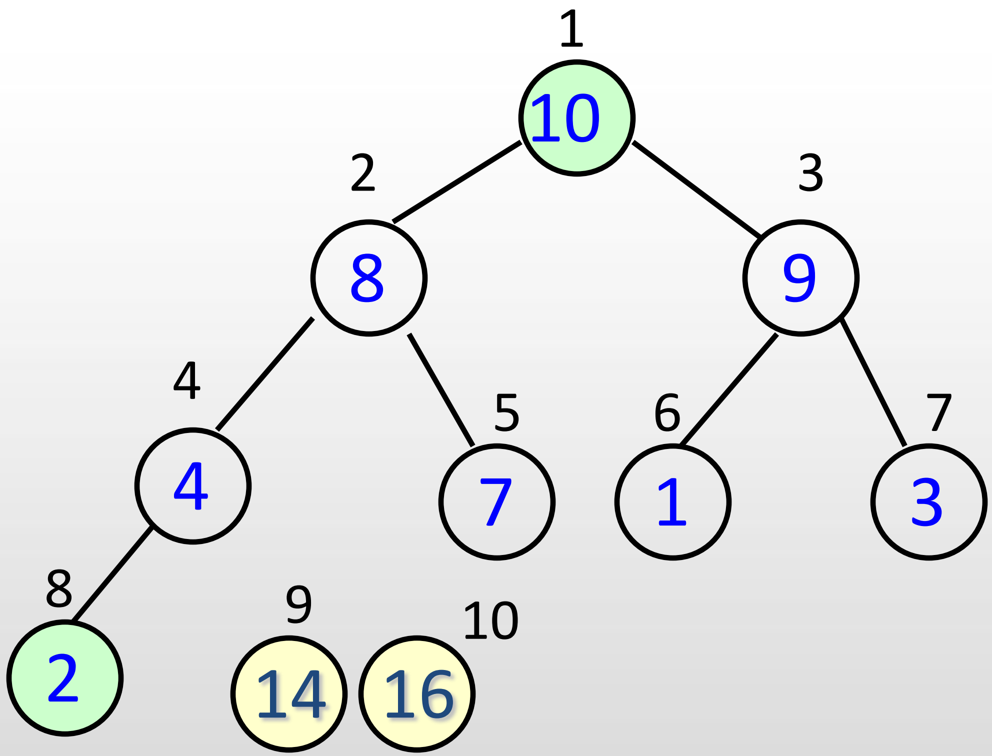


Heapify(A,1,8)

3rd pass: i=8

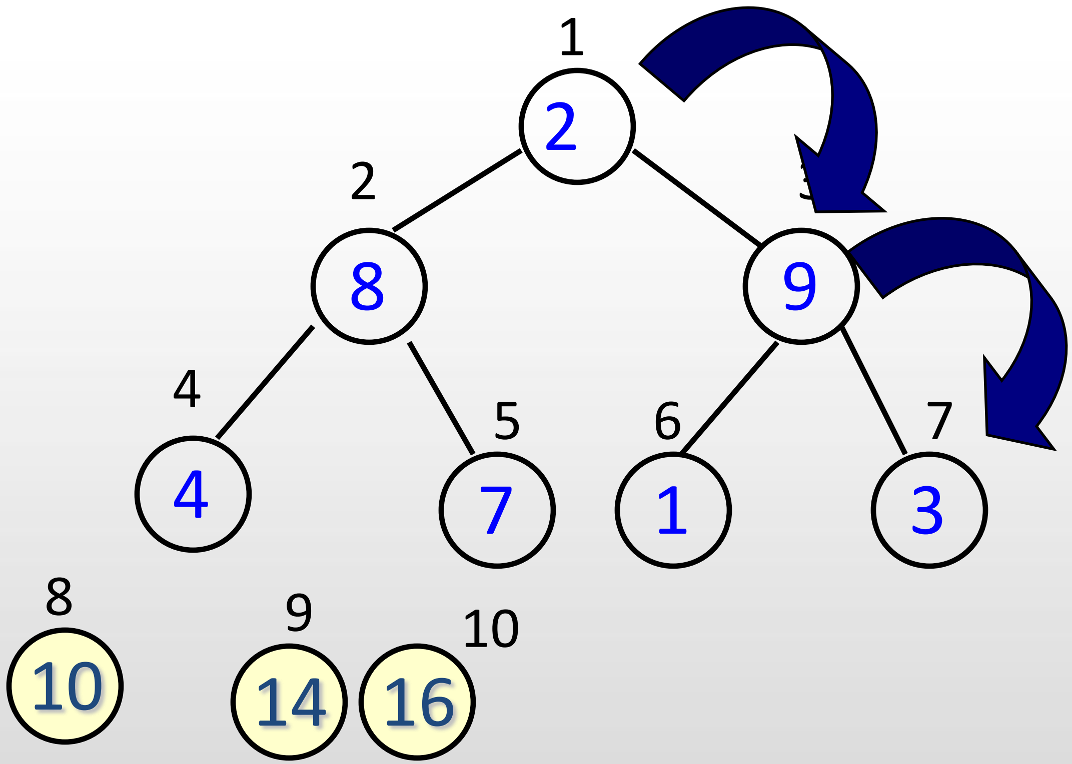


3rd pass: i=8



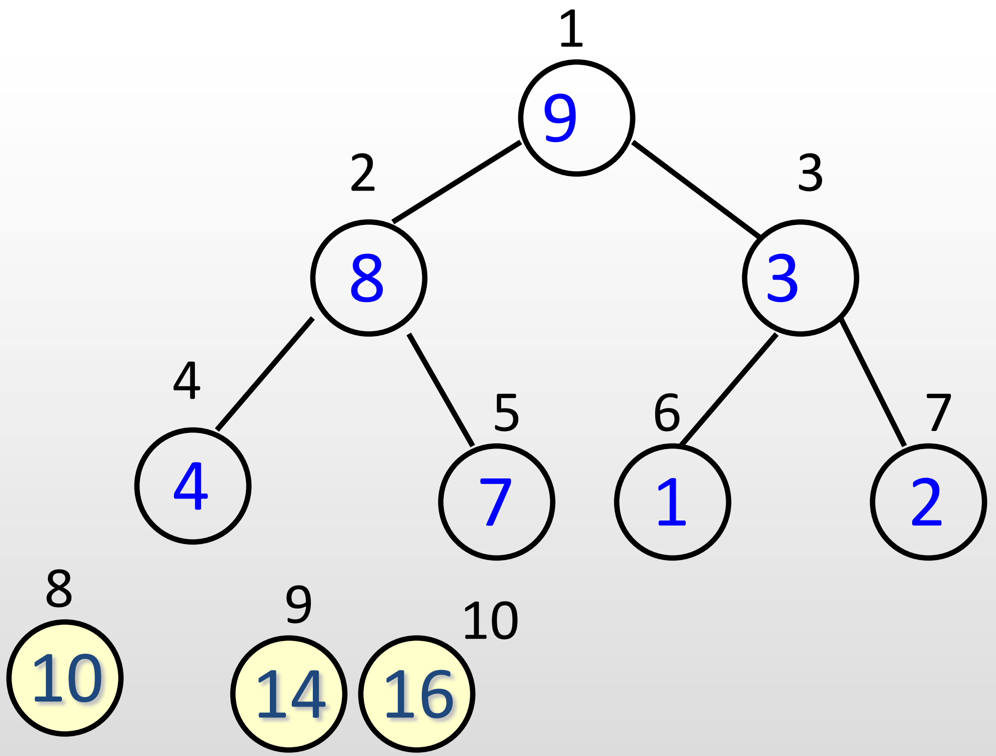
swap(A[1],A[8])

3rd pass: i=8

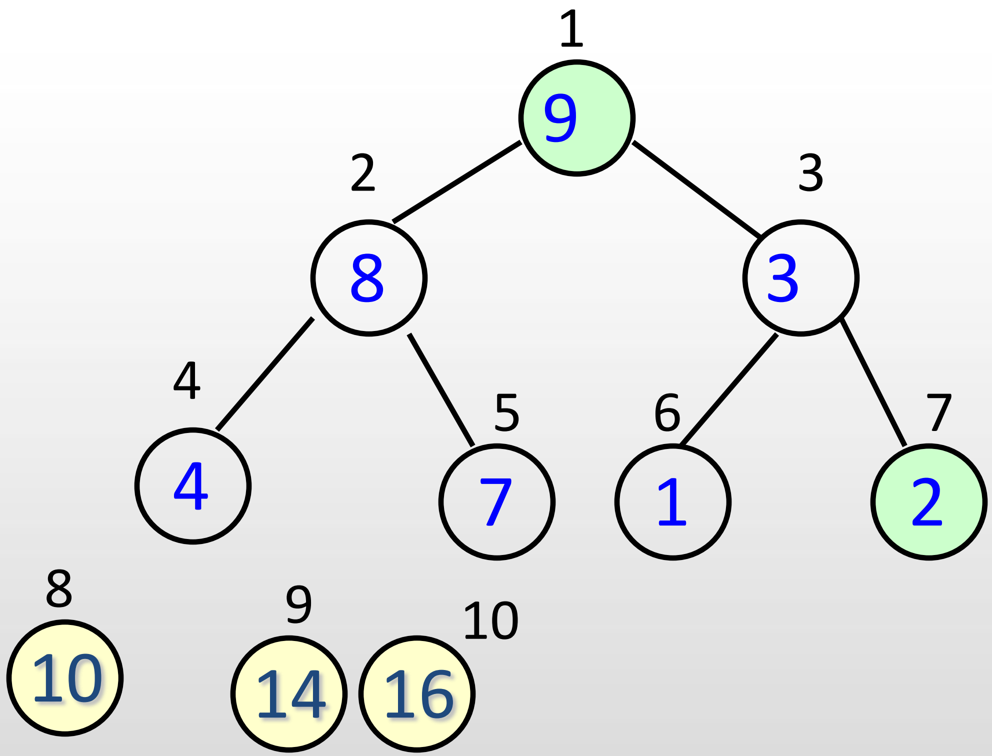


Heapify(A,1,7)

4th pass: i=7

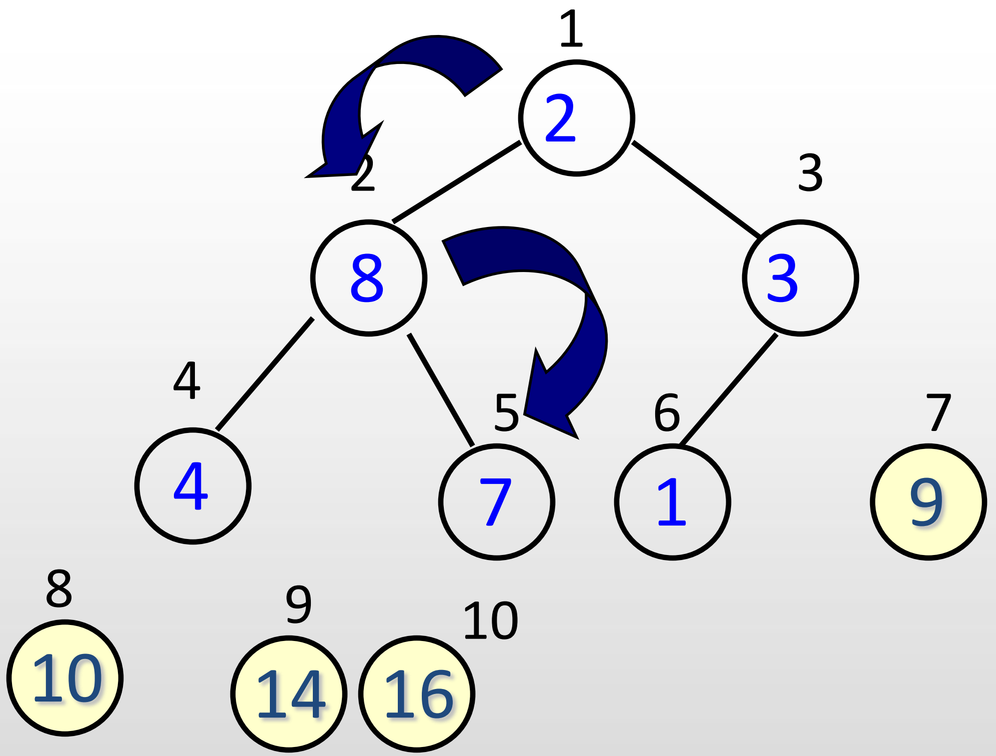


4th pass: i=7



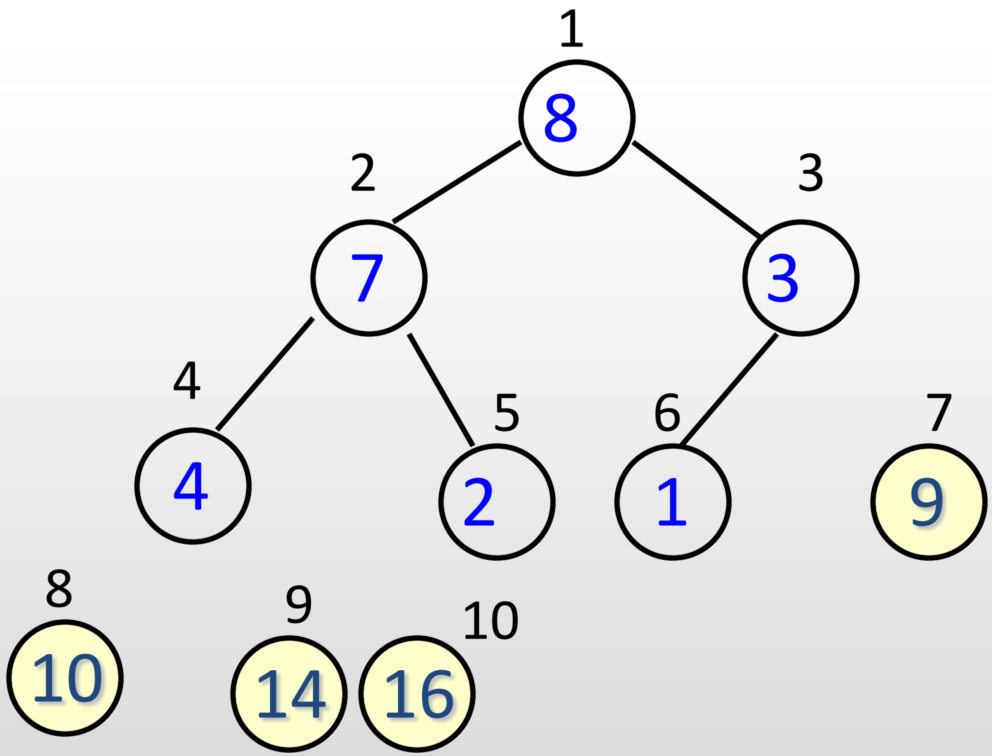
swap(A[1],A[7])

4th pass: i=7

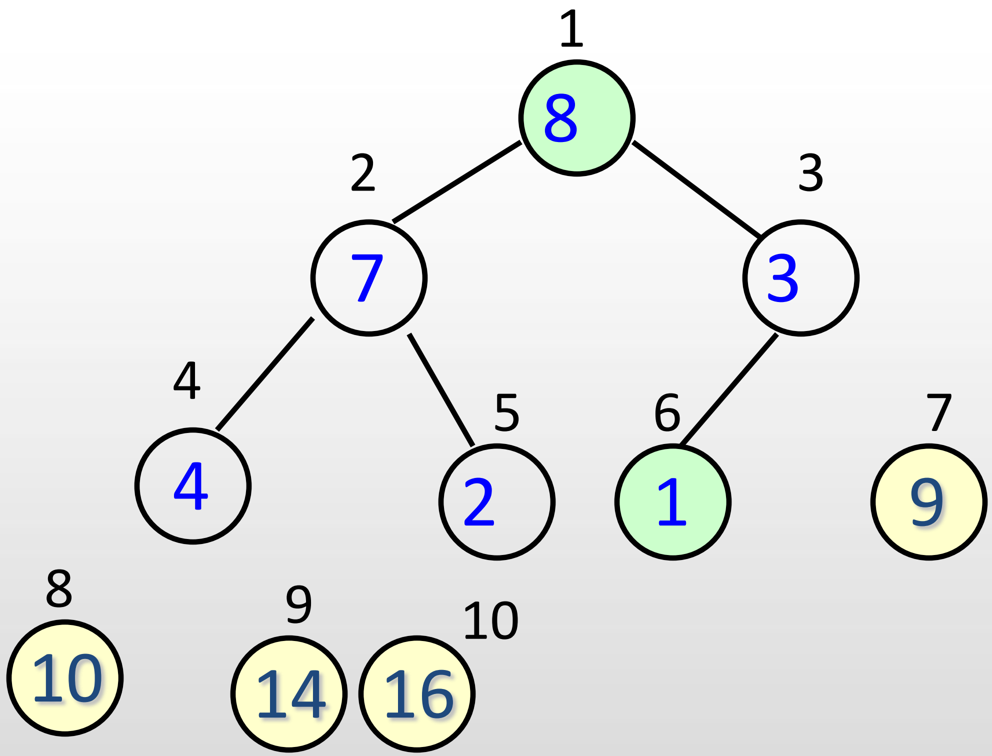


Heapify(A,1,6)

5th pass: i=6

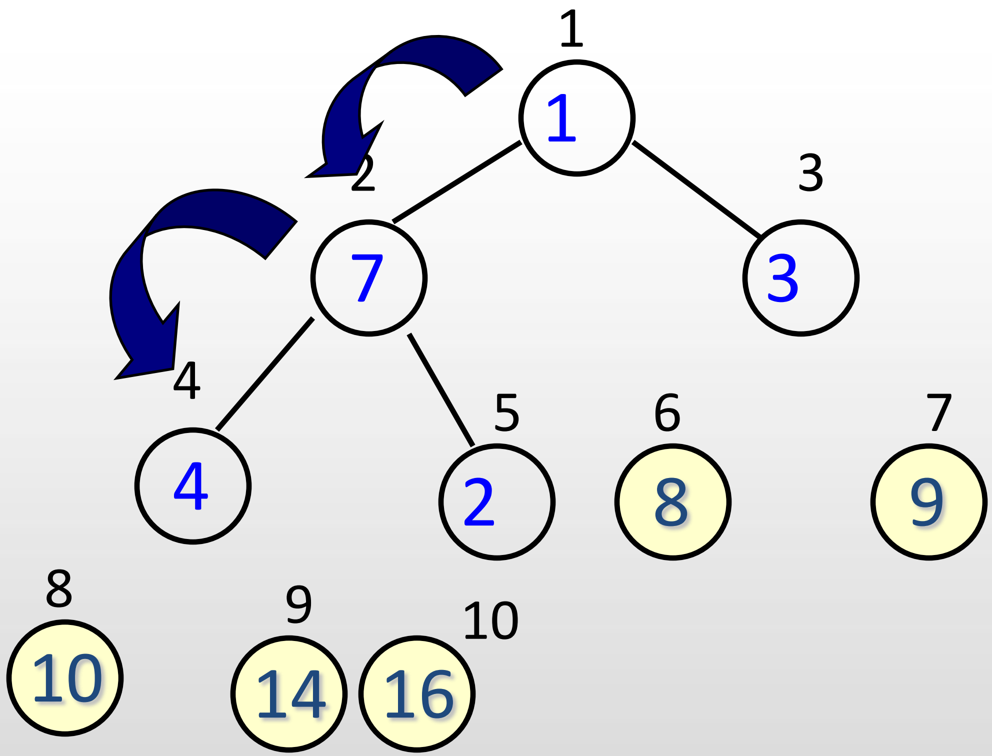


5th pass: i=6



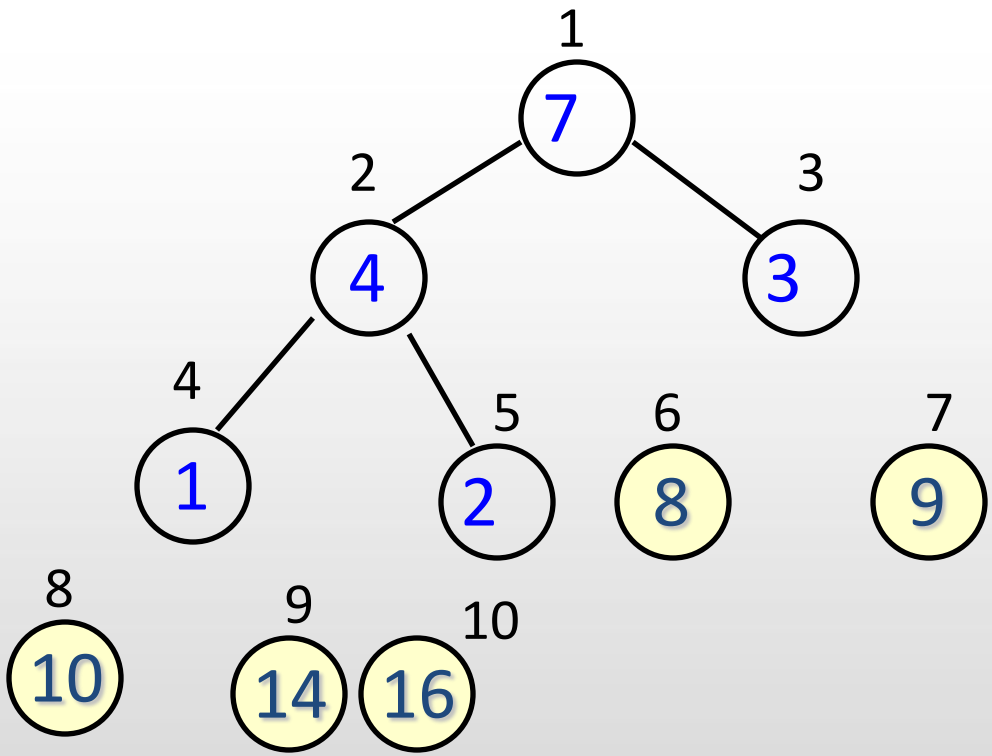
swap(A[1],A[6])

5th pass: i=6

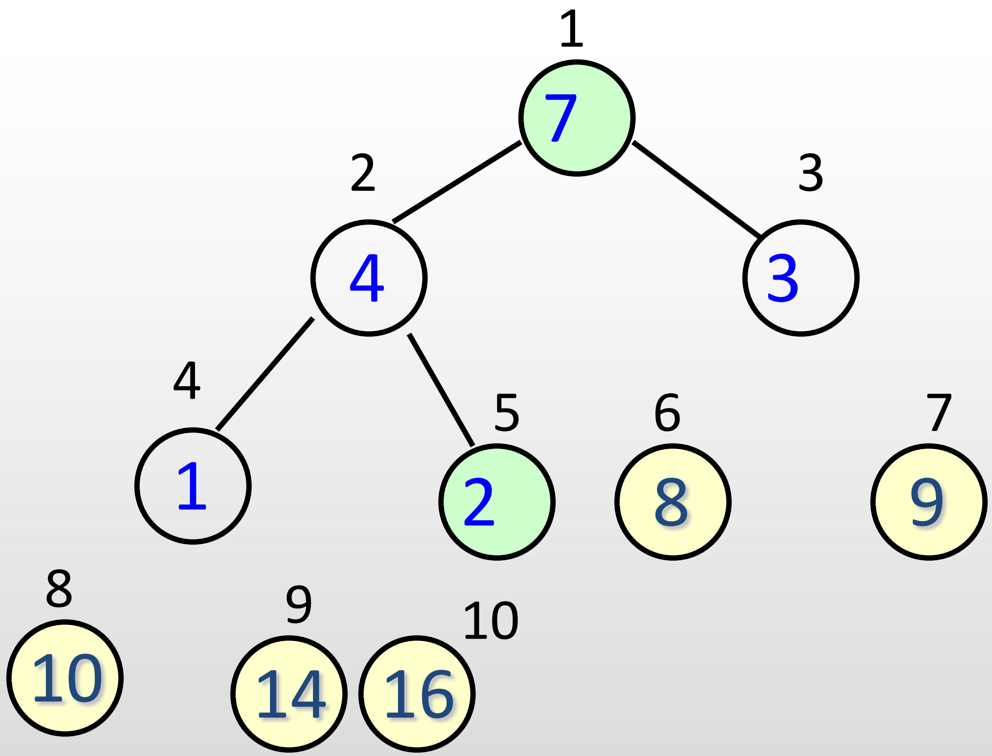


Heapify(A,1,5)

6th pass: i=5

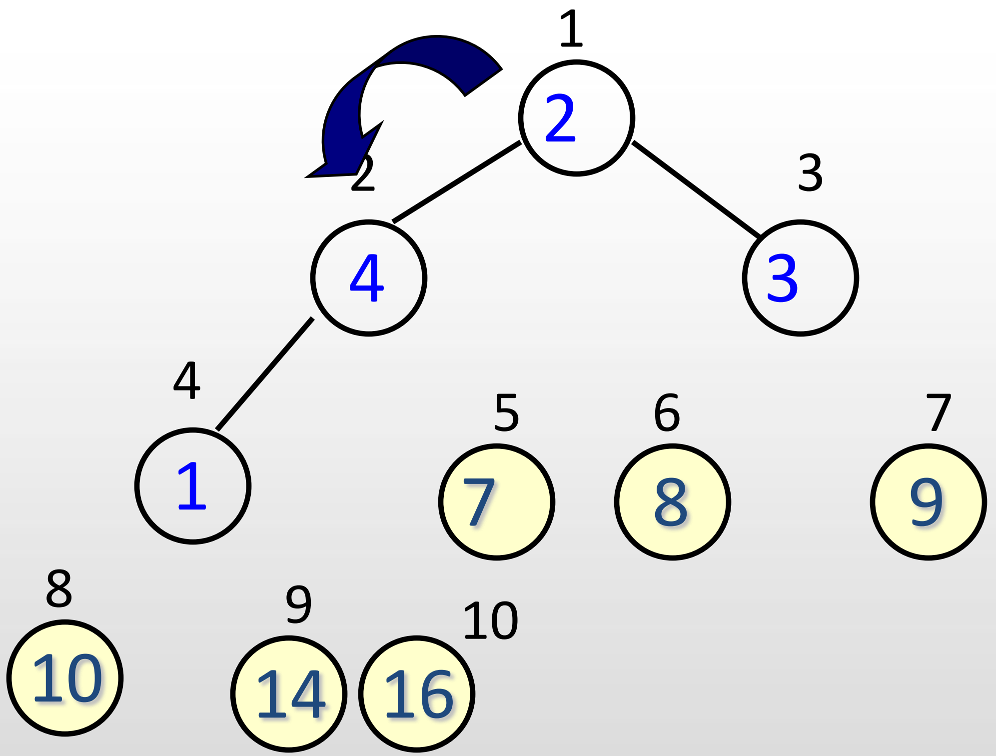


6th pass: i=5



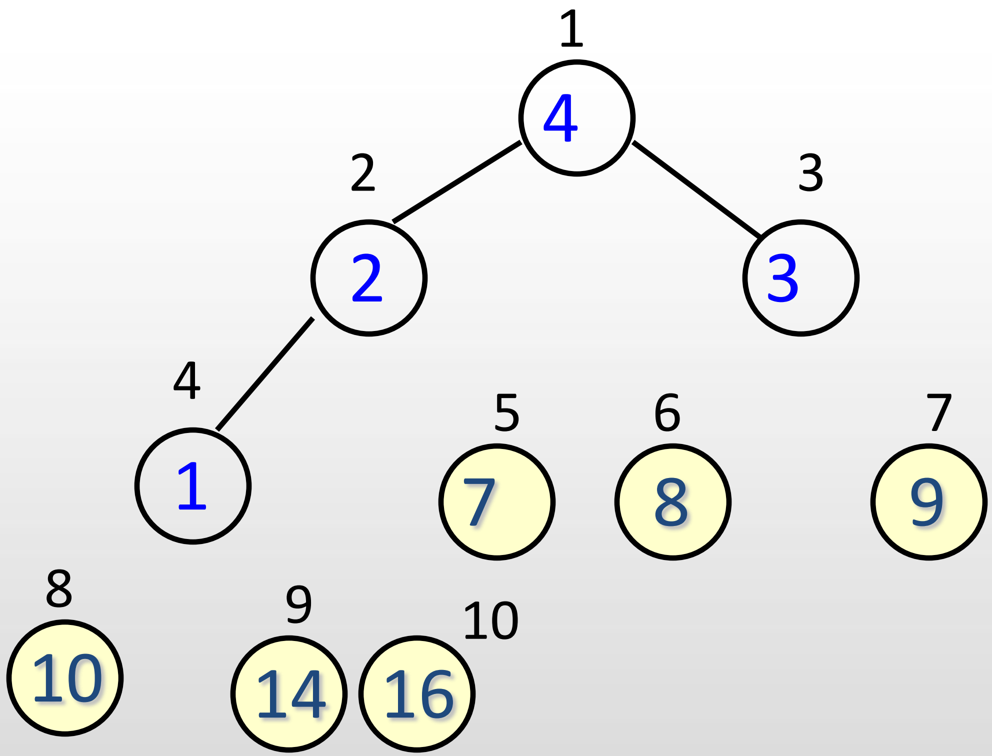
swap(A[1],A[5])

6th pass: i=5

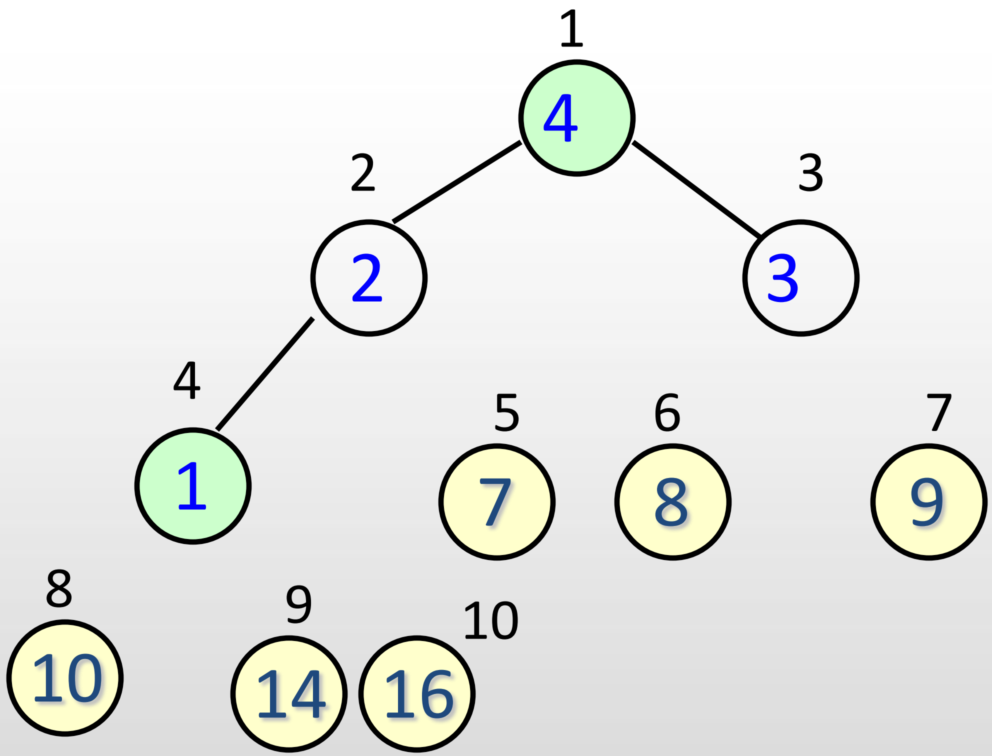


Heapify(A,1,4)

7th pass: i=4

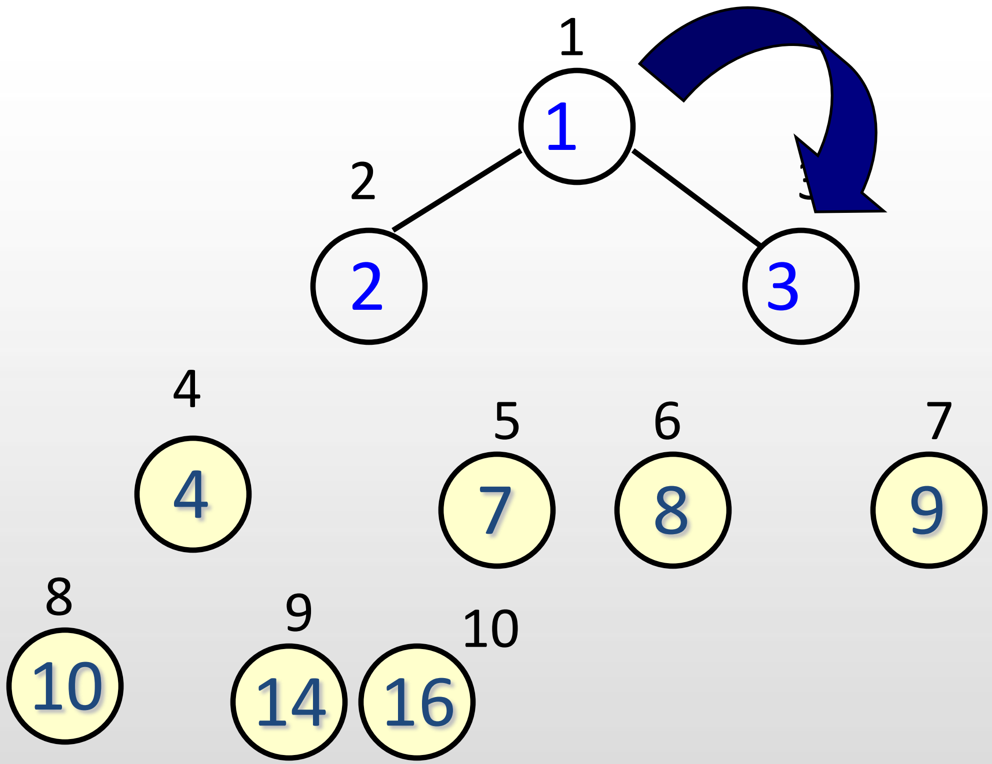


7th pass: i=4



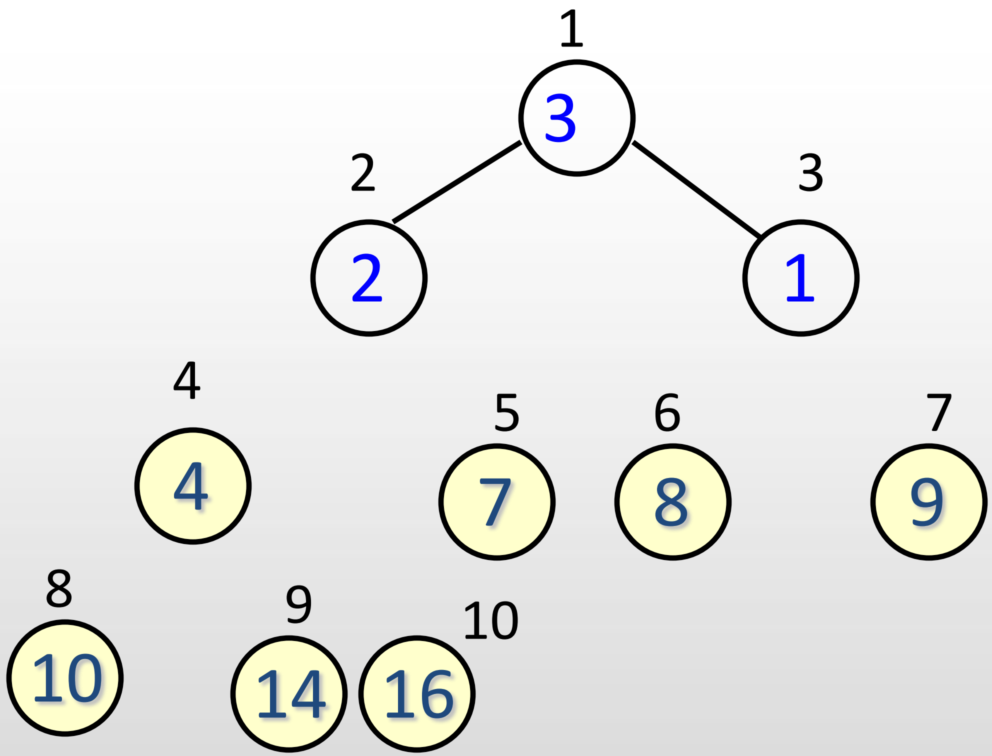
swap(A[1],A[4])

7th pass: i=4

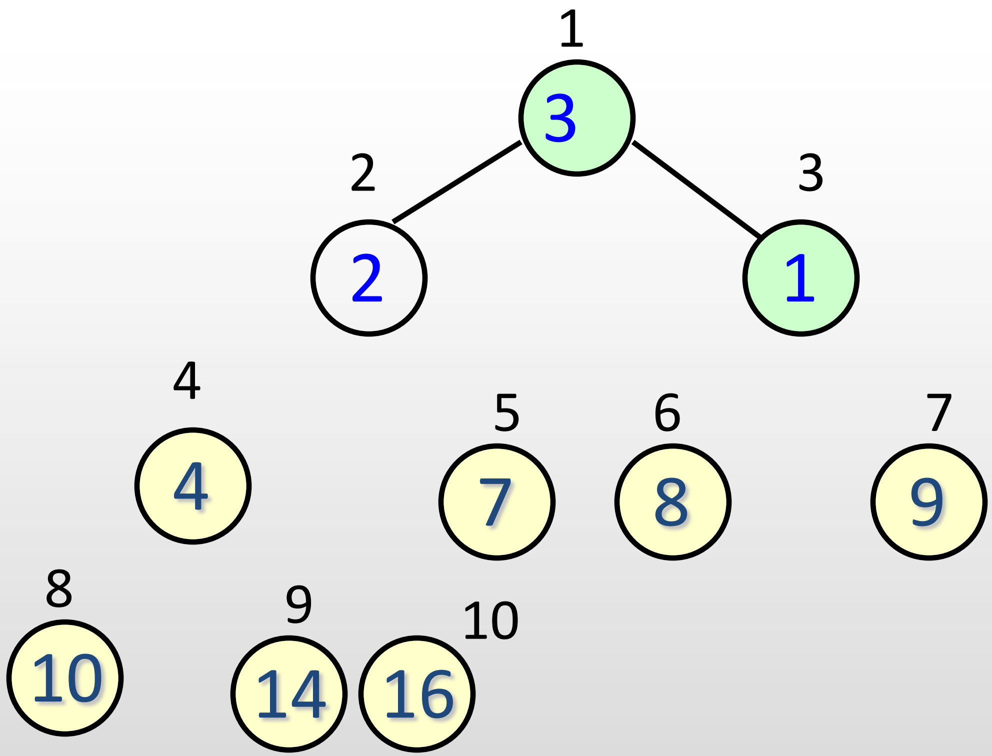


Heapify(A,1,3)

8th pass: i=3

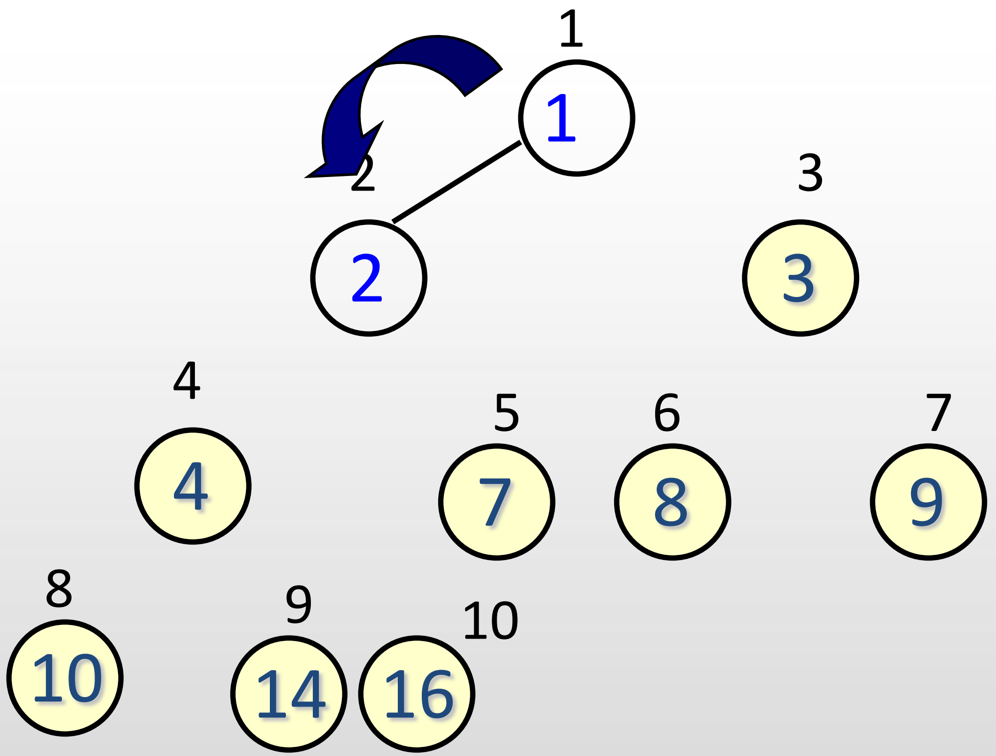


8th pass: i=3



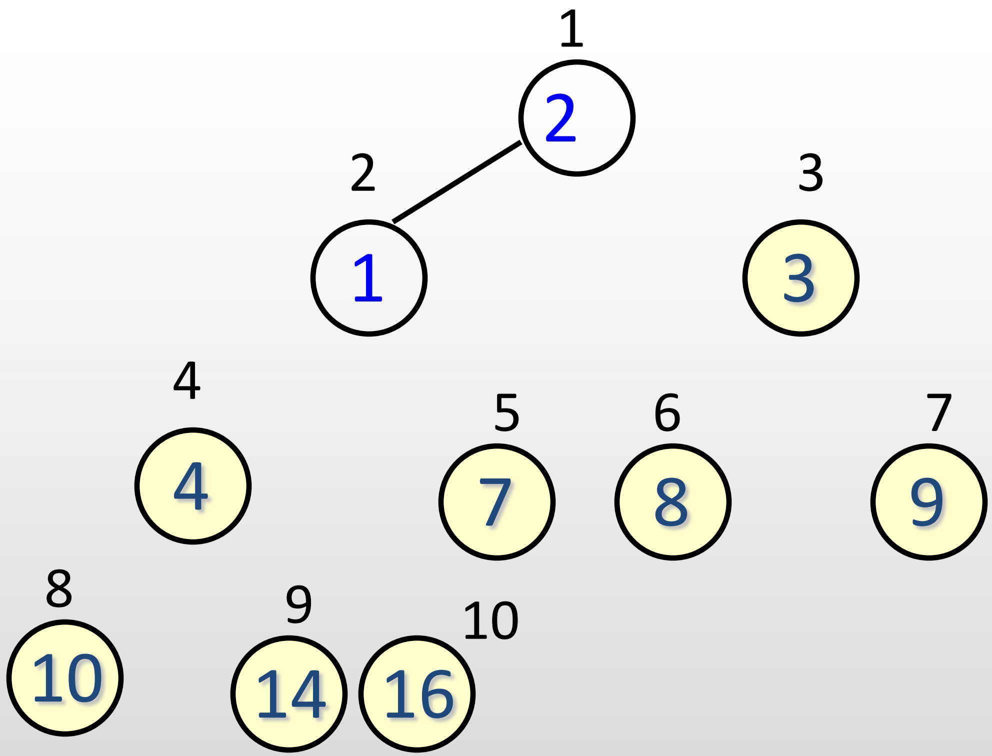
swap(A[1],A[3])

8th pass: i=3

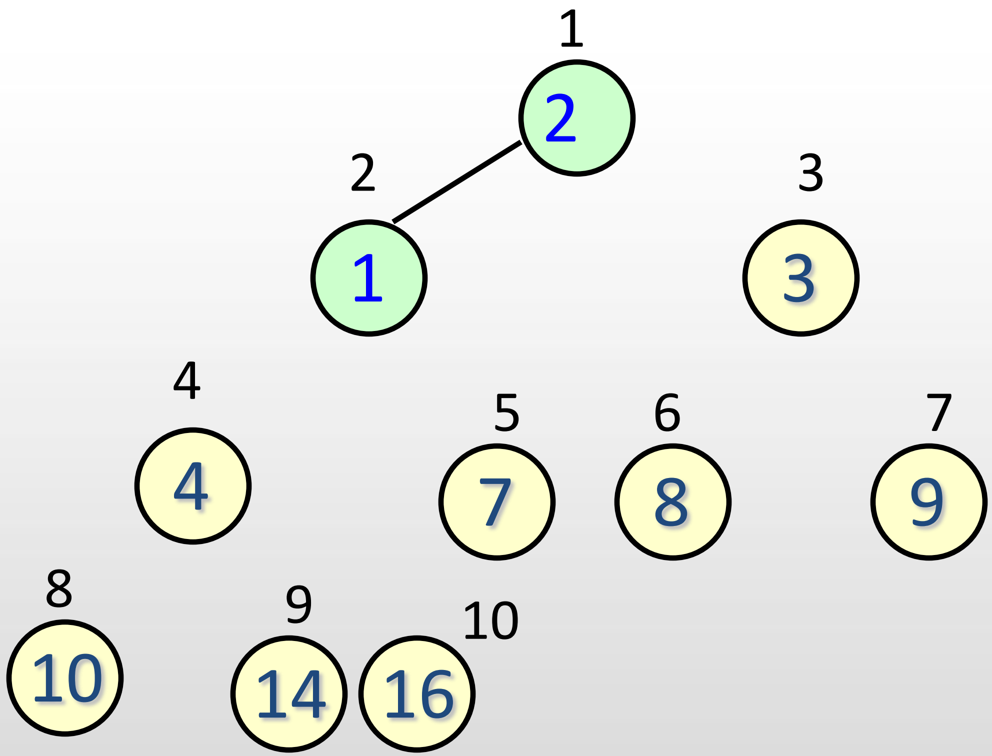


Heapify(A,1,2)

9th pass: i=2

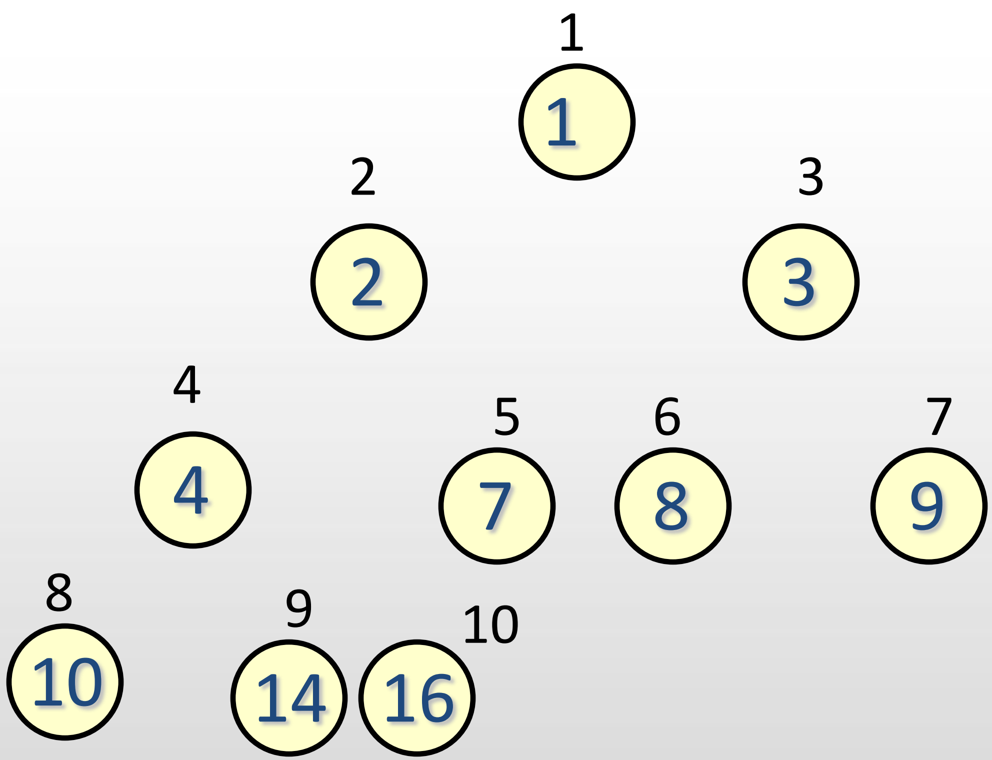


9th pass: i=2



swap(A[1],A[2])

9th pass: i=2



A									
1	2	3	4	7	8	9	10	14	16

Quicksort

Basic algorithm to sort an array S:

1. If no. of elements in S is 0 or 1, return.
2. Pick any element v in S. This is called the pivot.
3. Partition S-{v} into two disjoint groups.
   $S_1=\{x \in S-\{v\} \mid x \le v\}$, and $S_2=\{x \in S-\{v\} \mid x \ge v\}$
4. Return {quicksort(S_1), v, quicksort(S_2)}.

Quicksort

```
quicksort(int start ,int end) {
    int pivot;
    pivot=select_pivot();
    partition;
    quicksort(partition1);
    quicksort(partition2);
}
```

Picking the pivot

- Choice of pivot is a factor on the performance of the algorithm
- Wrong way: first element provides poor partition if input is presorted or in reverse order
- Safe course: random
- Median-of-three Partitioning
  - Pick three elements randomly and use the median as pivot
  - OR median of the left, right and center elements

Partitioning Strategy

1. Get the pivot element out of the way by swapping it with the last element.
2. i = first element, j = next-to-last element
3. Move all small elements (relative to the pivot) to the left part of the array and all large elements to the right.
4. While i < j, move i to the right, skipping over elements smaller than the pivot. Move j to the left, skipping over elements larger than the pivot.

Partitioning Strategy

5. When i and j have stopped, i is pointing at a large element and j is pointing at a small element. If i is to the left of j, swap the elements.
6. Repeat the process until i and j cross.
7. Swap the pivot element with the element pointed to by i.

Sort 3, 10, 4, 6, 8, 9, 7, 2 using quicksort (random pivot)

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

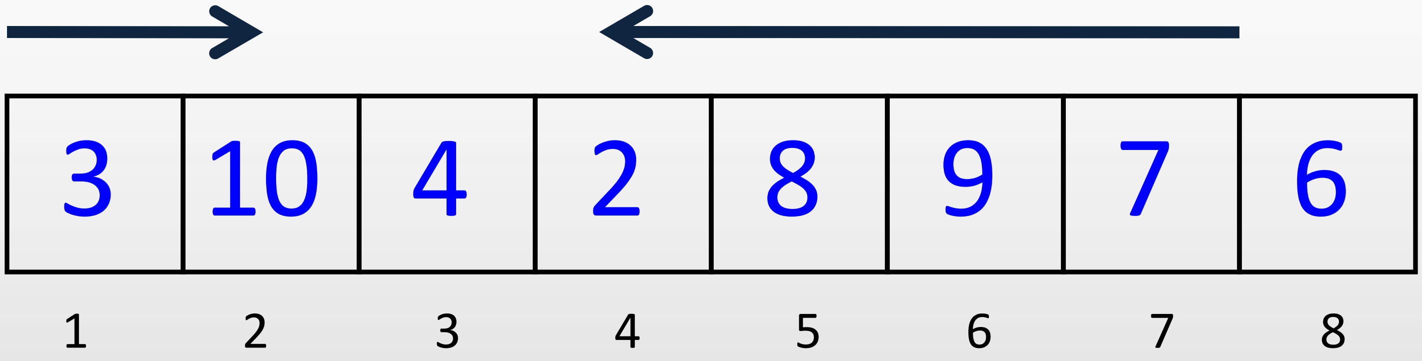
pivot = 6

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

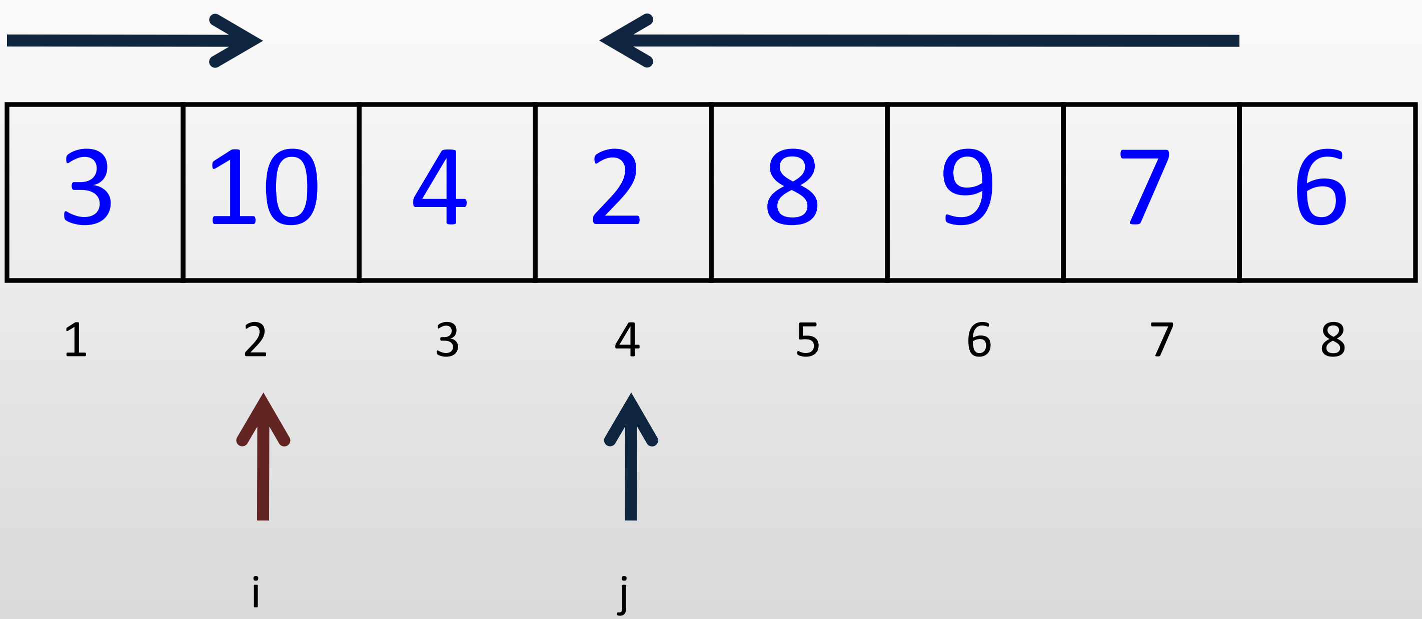
pivot = 6

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

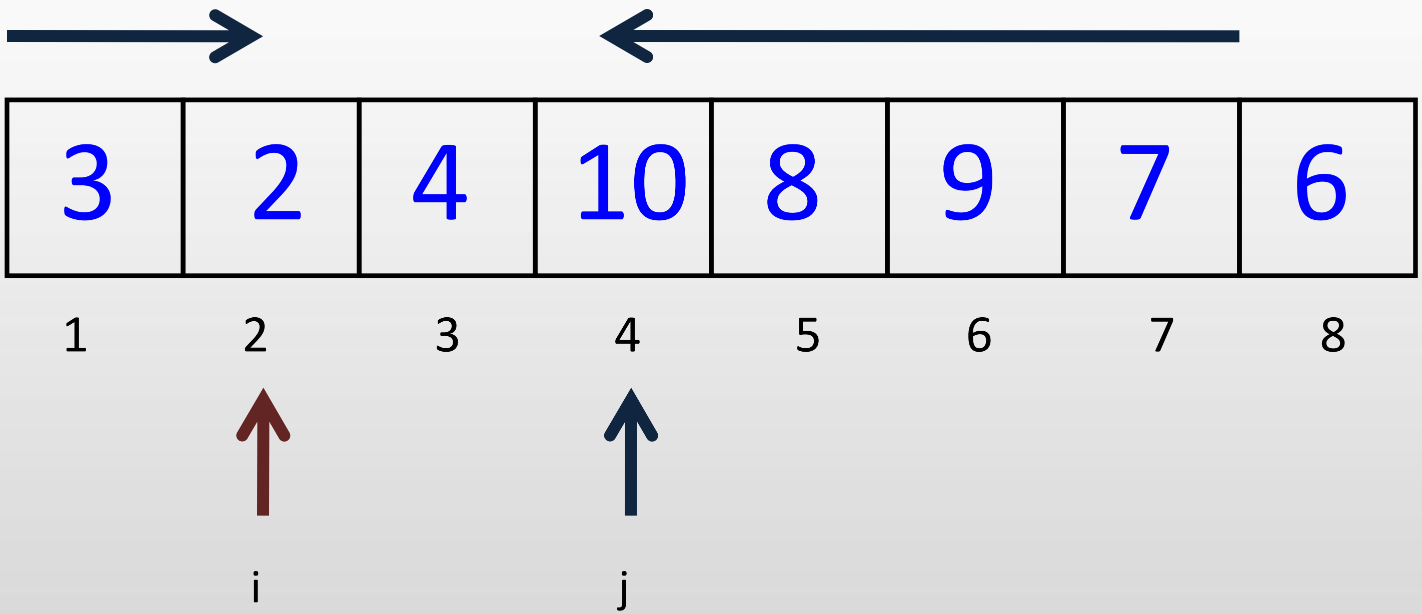
pivot = 6



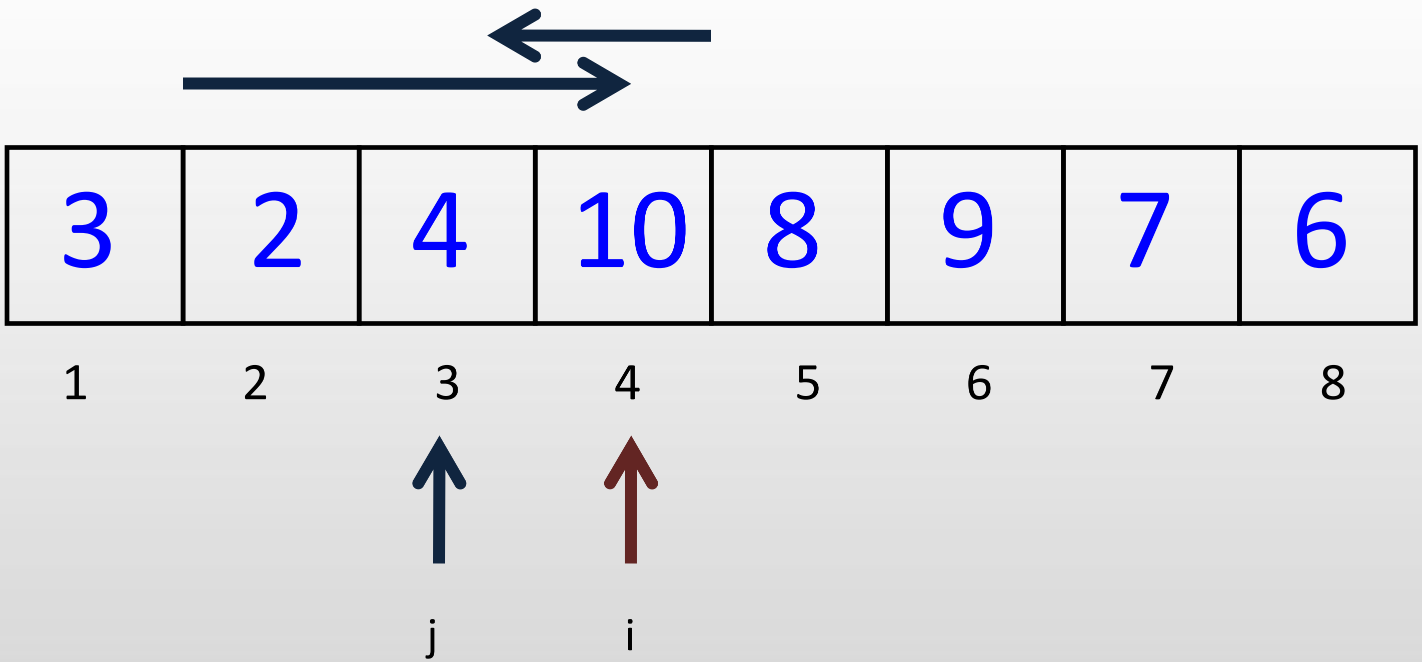
pivot = 6



pivot = 6



pivot = 6



pivot = 6

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

i

pivot = 6

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

pivot = 6

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

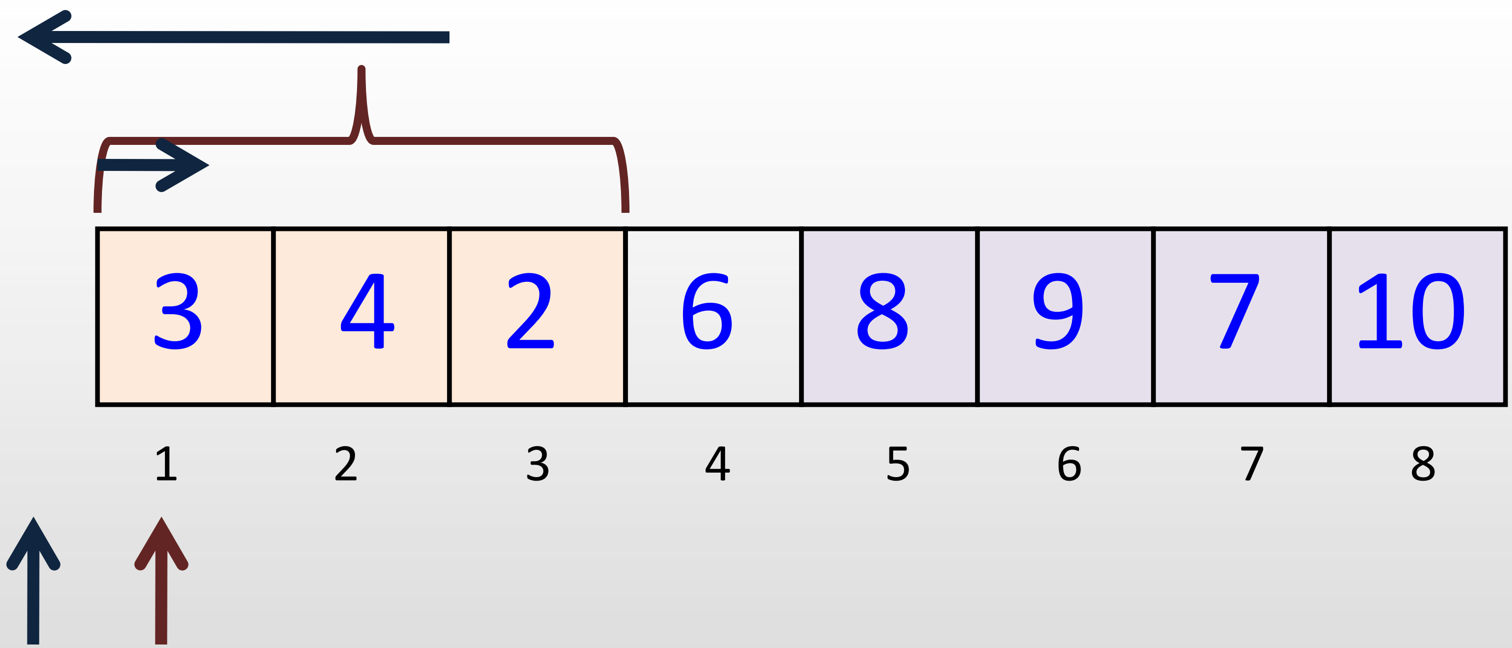
S_1: pivot = 2

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

S_1: pivot = 2

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

S_1: pivot = 2



S_1: pivot = 2

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

S_1: pivot = 2

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

S_2: pivot = 4

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

S_2: pivot = 4

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

S_2: pivot = 4

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

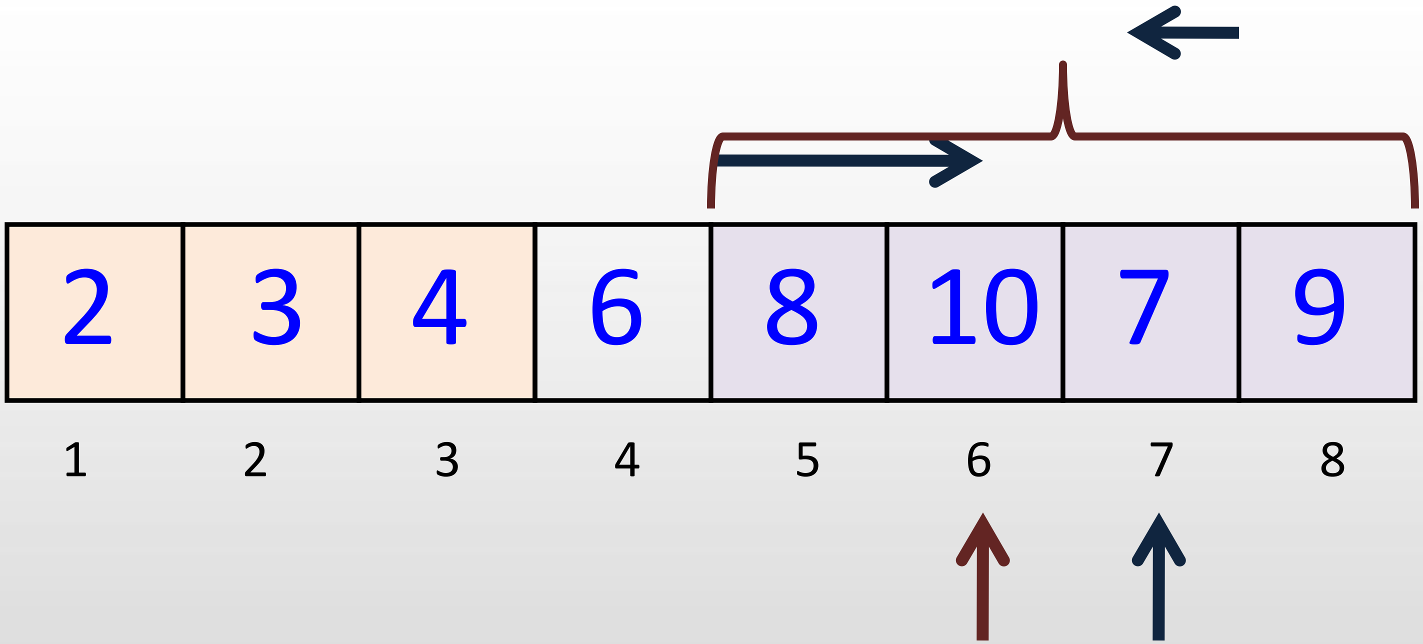
S_1: pivot = 9

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

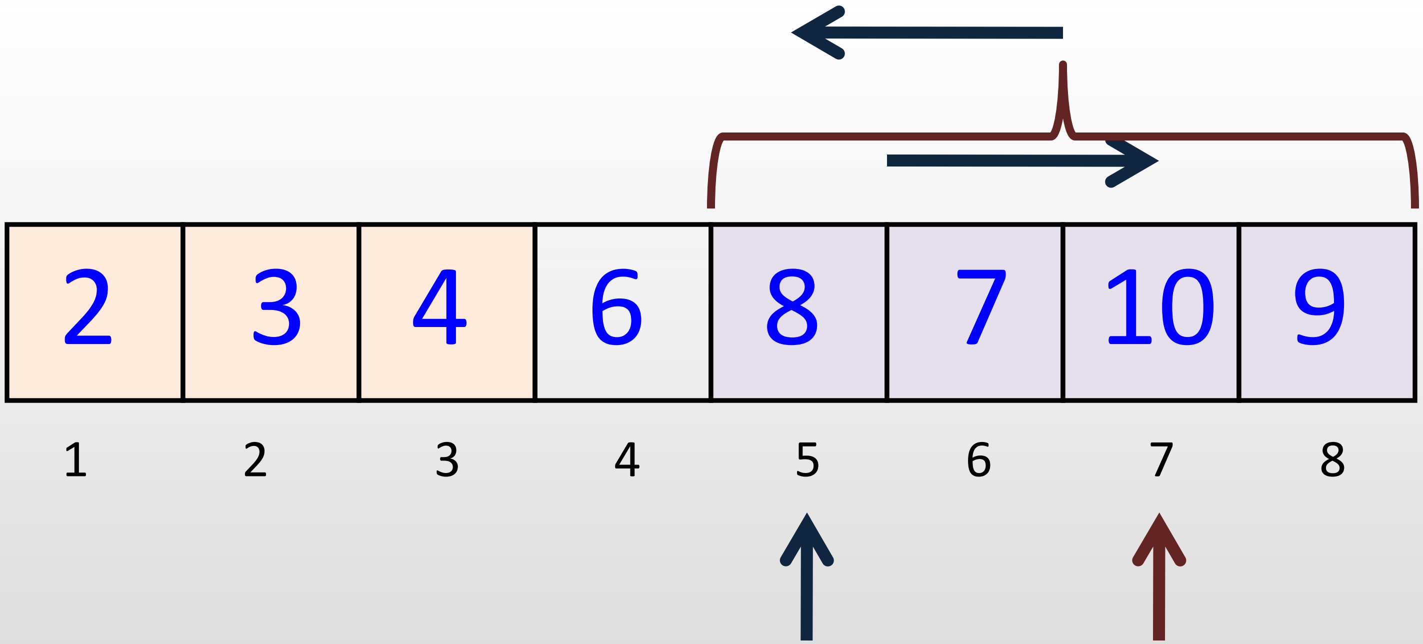
S_1: pivot = 9

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

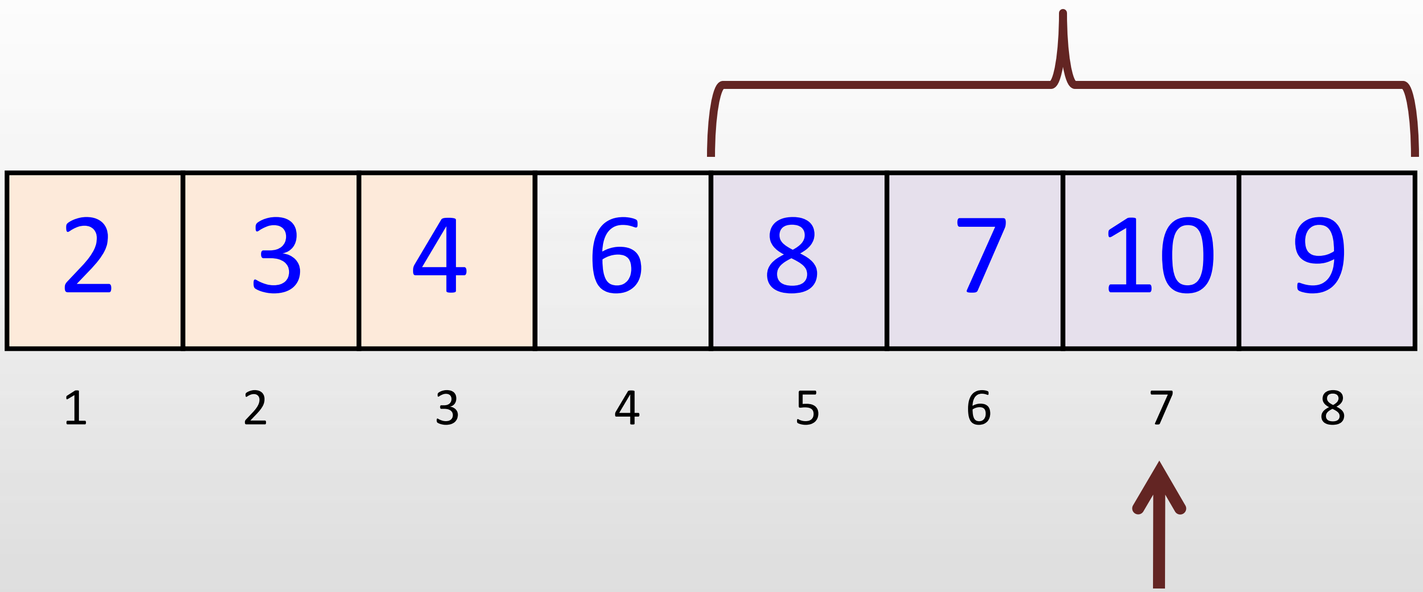
S_1: pivot = 9



S_1: pivot = 9



S_1: pivot = 9



S_1: pivot = 9

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

S_1: pivot = 9

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

S_1: pivot = 8

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

S_1: pivot = 8

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

Shell Sort

- works by comparing elements that are distant
- distance between comparisons decreases as the algorithm runs until the last phase, in which adjacent elements are compared

- uses a sequence $h_1, h_2, \ldots, h_t$, called the *increment sequence*
- any increment sequence will do as long as $h_1 = 1$
- after a phase, using some increment h_k, for every i, $a[i] \leq a[i+h_k]$

Shell Sort

- A popular choice for increment sequence is to use the sequence suggested by Shell:

  $h_t = \lfloor n/2 \rfloor$ and $h_k = \lfloor h_{k+1}/2 \rfloor$

Shell Sort

```
void shellsort(int a[],int n) {
    int hk, tmp, i, j;
    for (hk=n/2; hk>0; hk/=2)
        for(i=hk+1; i<=n; i++){
            tmp = a[i];
            for(j=i; j>hk; j-=hk)
                if (tmp<a[j-hk])
                    a[j] = a[j-hk];
                else
                    break;
            a[j] = tmp;
        }
}
```

Sort the ff. using shellsort

81	94	11	96	12	35	17	95	28	58	41	75	15
1	2	3	4	5	6	7	8	9	10	11	12	13

Sort the ff. using shellsort

81	94	11	96	12	35	17	95	28	58	41	75	15
1	2	3	4	5	6	7	8	9	10	11	12	13

- hk = 6
- i = 7
- tmp = 17

Sort the ff. using shellsort

81	94	11	96	12	35	17	95	28	58	41	75	15
1	2	3	4	5	6	7	8	9	10	11	12	13

- hk = 6
- i = 7
- tmp = 17

Sort the ff. using shellsort

17	94	11	96	12	35	81	95	28	58	41	75	15
1	2	3	4	5	6	7	8	9	10	11	12	13

- hk = 6
- i = 7
- tmp = 17

Sort the ff. using shellsort

17	94	11	96	12	35	81	95	28	58	41	75	15
1	2	3	4	5	6	7	8	9	10	11	12	13

- hk = 6
- i = 8
- tmp = 95

Sort the ff. using shellsort

17	94	11	96	12	35	81	95	28	58	41	75	15
1	2	3	4	5	6	7	8	9	10	11	12	13

- hk = 6
- i = 9
- tmp = 28

Sort the ff. using shellsort

17	94	11	96	12	35	81	95	28	58	41	75	15
1	2	3	4	5	6	7	8	9	10	11	12	13

- hk = 6
- i = 10
- tmp = 58

Sort the ff. using shellsort

17	94	11	58	12	35	81	95	28	96	41	75	15
1	2	3	4	5	6	7	8	9	10	11	12	13

- hk = 6
- i = 10
- tmp = 58

Sort the ff. using shellsort

17	94	11	58	12	35	81	95	28	96	41	75	15
1	2	3	4	5	6	7	8	9	10	11	12	13

- hk = 6
- i = 11
- tmp = 41

Sort the ff. using shellsort

17	94	11	58	12	35	81	95	28	96	41	75	15
1	2	3	4	5	6	7	8	9	10	11	12	13

- hk = 6
- i = 12
- tmp = 75

Sort the ff. using shellsort

17	94	11	58	12	35	81	95	28	96	41	75	15
1	2	3	4	5	6	7	8	9	10	11	12	13

- hk = 6
- i = 13
- tmp = 15

Sort the ff. using shellsort

17	94	11	58	12	35	15	95	28	96	41	75	81
1	2	3	4	5	6	7	8	9	10	11	12	13

- hk = 6
- i = 13
- tmp = 15

Sort the ff. using shellsort

15	94	11	58	12	35	17	95	28	96	41	75	81
1	2	3	4	5	6	7	8	9	10	11	12	13

- hk = 6
- i = 13
- tmp = 15

Sort the ff. using shellsort

15	94	11	58	12	35	17	95	28	96	41	75	81
1	2	3	4	5	6	7	8	9	10	11	12	13

Sort the ff. using shellsort

15	94	11	58	12	35	17	95	28	96	41	75	81
1	2	3	4	5	6	7	8	9	10	11	12	13

- hk = 6/2 = 3
- i = 4
- tmp = 58

Sort the ff. using shellsort

15	94	11	58	12	35	17	95	28	96	41	75	81
1	2	3	4	5	6	7	8	9	10	11	12	13

- hk = 3
- i = 5
- tmp = 12

Sort the ff. using shellsort

15	12	11	58	94	35	17	95	28	96	41	75	81
1	2	3	4	5	6	7	8	9	10	11	12	13

- hk = 3
- i = 5
- tmp = 12

Sort the ff. using shellsort

15	12	11	58	94	35	17	95	28	96	41	75	81
1	2	3	4	5	6	7	8	9	10	11	12	13

- hk = 3
- i = 6
- tmp = 35

Sort the ff. using shellsort

15	12	11	58	94	35	17	95	28	96	41	75	81
1	2	3	4	5	6	7	8	9	10	11	12	13

- hk = 3
- i = 7
- tmp = 17

Sort the ff. using shellsort

15	12	11	17	94	35	58	95	28	96	41	75	81
1	2	3	4	5	6	7	8	9	10	11	12	13

- hk = 3
- i = 7
- tmp = 17

Sort the ff. using shellsort

15	12	11	17	94	35	58	95	28	96	41	75	81
1	2	3	4	5	6	7	8	9	10	11	12	13

- hk = 3
- i = 8
- tmp = 95

Sort the ff. using shellsort

15	12	11	17	94	35	58	95	28	96	41	75	81
1	2	3	4	5	6	7	8	9	10	11	12	13

- hk = 3
- i = 9
- tmp = 28

Sort the ff. using shellsort

15	12	11	17	94	28	58	95	35	96	41	75	81
1	2	3	4	5	6	7	8	9	10	11	12	13

- hk = 3
- i = 9
- tmp = 28

Sort the ff. using shellsort

15	12	11	17	94	28	58	95	35	96	41	75	81
1	2	3	4	5	6	7	8	9	10	11	12	13

- hk = 3
- i = 10
- tmp = 96

Sort the ff. using shellsort

15	12	11	17	94	28	58	95	35	96	41	75	81
1	2	3	4	5	6	7	8	9	10	11	12	13

- hk = 3
- i = 11
- tmp = 41

Sort the ff. using shellsort

15	12	11	17	94	28	58	41	35	96	95	75	81
1	2	3	4	5	6	7	8	9	10	11	12	13

- hk = 3
- i = 11
- tmp = 41

Sort the ff. using shellsort

15	12	11	17	41	28	58	94	35	96	95	75	81
1	2	3	4	5	6	7	8	9	10	11	12	13

- hk = 3
- i = 11
- tmp = 41

Sort the ff. using shellsort

15	12	11	17	41	28	58	94	35	96	95	75	81
1	2	3	4	5	6	7	8	9	10	11	12	13

- hk = 3
- i = 12
- tmp = 75

Sort the ff. using shellsort

15	12	11	17	41	28	58	94	35	96	95	75	81
1	2	3	4	5	6	7	8	9	10	11	12	13

- hk = 3
- i = 13
- tmp = 81

Sort the ff. using shellsort

15	12	11	17	41	28	58	94	35	81	95	75	96
1	2	3	4	5	6	7	8	9	10	11	12	13

- hk = 3
- i = 13
- tmp = 81

Sort the ff. using shellsort

15	12	11	17	41	28	58	94	35	81	95	75	96
1	2	3	4	5	6	7	8	9	10	11	12	13

- hk = 1
- i = 2
- tmp = 12

Sort the ff. using shellsort

12	15	11	17	41	28	58	94	35	81	95	75	96
1	2	3	4	5	6	7	8	9	10	11	12	13

- hk = 1
- i = 2
- tmp = 12

Sort the ff. using shellsort

11	12	15	17	41	28	58	94	35	81	95	75	96
1	2	3	4	5	6	7	8	9	10	11	12	13

- hk = 1
- i = 1
- tmp = 11

Sort the ff. using shellsort

11	12	15	17	28	41	58	94	35	81	95	75	96
1	2	3	4	5	6	7	8	9	10	11	12	13

- hk = 1
- i = 6
- tmp = 28

Sort the ff. using shellsort

11	12	15	17	28	35	41	58	94	81	95	75	96
1	2	3	4	5	6	7	8	9	10	11	12	13

- hk = 1
- i = 9
- tmp = 35

Sort the ff. using shellsort

11	12	15	17	28	35	41	58	81	94	95	75	96
1	2	3	4	5	6	7	8	9	10	11	12	13

- hk = 1
- i = 10
- tmp = 81

Sort the ff. using shellsort

11	12	15	17	28	35	41	58	75	81	94	95	96
1	2	3	4	5	6	7	8	9	10	11	12	13

- hk = 1
- i = 12
- tmp = 75

Linear Time Sorting

- Sorting in linear time is possible in some special cases.
- Extra information must be available.

Bucket sort

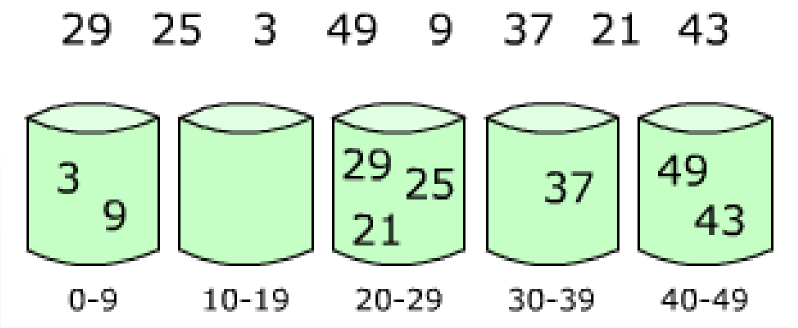
- Works by partitioning an array into a number of buckets.
- Each bucket is then sorted individually, either using a different sorting algorithm, or by recursively applying the bucket sort algorithm.

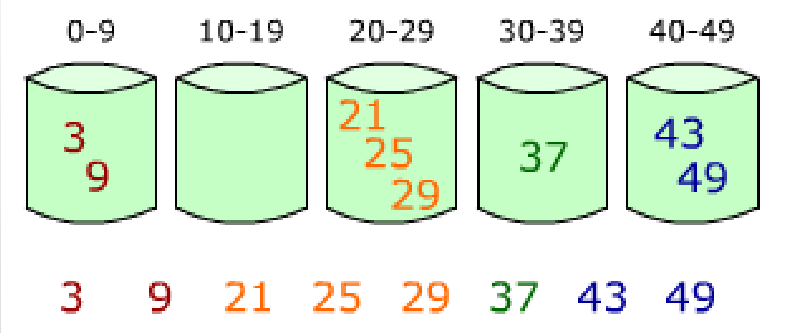
Bucket sort

- Algorithm:
  - Set up an array of initially empty "buckets."“
  - Go over the original array, putting each object in its bucket.
  - Sort each non-empty bucket.
  - Visit the buckets in order and put all elements back into the original array.

Bucket Sort

```
Bucketsort(A,n) {
    for(i=1; i<=n; i++)
        insert(A[i], B[getpos(A[i])])
    for(i=0; i<n; i++)
        insertion_sort(B[i])
    concatenate B[0], B[1],...B[n]
}
```





Counting Sort

- Bucket sort can be seen as a generalization of counting sort.
- If each bucket has size 1 then bucket sort degenerates to counting sort.

Counting Sort

- Suppose that the values to be sorted are all between 0 and k, where k is some (small) integer. Assume the values are in the array A[1..n].
- Algorithm:

  1. Use an array C[0..k] to count how many times each key occurs in A.
  2. Calculate cumulative totals in C.
  3. Copy data into the target array B.

Counting Sort

```
Counting Sort(A,n) {
    k=findmax(A)
    for(i=0; i<=k; i++)
        C[i]=0;
    for(i=1; i<=n; i++)
        C[A[i]]+=1;
    for(i=1; i<=k; i++)
        C[i]+=C[i-1];
    for(i=n; i>=1; i--) {
        B[C[A[i]]]=A[i];
        C[A[i]]-=1
    }
}
```

A

2	5	3	0	2	3	0	3
1	2	3	4	5	6	7	8

C

0	0	0	0	0	0
0	1	2	3	4	5

Counting Sort

```
Counting Sort(A,n) {
    k=findmax(A)
    for(i=0; i<=k; i++)
        C[i]=0;
    for(i=1; i<=n; i++)
        C[A[i]]+=1;
    for(i=1; i<=k; i++)
        C[i]+=C[i-1];
    for(i=n; i>=1; i--) {
        B[C[A[i]]]=A[i];
        C[A[i]]-=1
    }
}
```

A

2	5	3	0	2	3	0	3
1	2	3	4	5	6	7	8

C

0	0	1	0	0	0
0	1	2	3	4	5

Counting Sort

```
Counting Sort(A,n) {
    k=findmax(A)
    for(i=0; i<=k; i++)
        C[i]=0;
    for(i=1; i<=n; i++)
        C[A[i]]+=1;
    for(i=1; i<=k; i++)
        C[i]+=C[i-1];
    for(i=n; i>=1; i--) {
        B[C[A[i]]]=A[i];
        C[A[i]]-=1
    }
}
```

A

1	2	3	4	5	6	7	8
2	5	3	0	2	3	0	3

C

0	1	2	3	4	5
0	0	1	0	0	1

Counting Sort

```
Counting Sort(A,n) {
    k=findmax(A)
    for(i=0; i<=k; i++)
        C[i]=0;
    for(i=1; i<=n; i++)
        C[A[i]]+=1;
    for(i=1; i<=k; i++)
        C[i]+=C[i-1];
    for(i=n; i>=1; i--) {
        B[C[A[i]]]=A[i];
        C[A[i]]-=1
    }
}
```

A

2	5	3	0	2	3	0	3
1	2	3	4	5	6	7	8

C

2	0	2	3	0	1
0	1	2	3	4	5

Counting Sort

```
Counting Sort(A,n) {
    k=findmax(A)
    for(i=0; i<=k; i++)
        C[i]=0;
    for(i=1; i<=n; i++)
        C[A[i]]+=1;
    for(i=1; i<=k; i++)
        C[i]+=C[i-1];
    for(i=n; i>=1; i--) {
        B[C[A[i]]]=A[i];
        C[A[i]]-=1
    }
}
```

A

2	5	3	0	2	3	0	3
1	2	3	4	5	6	7	8

C

2	0	2	3	0	1
0	1	2	3	4	5

C

2	2	4	7	7	8
0	1	2	3	4	5

Counting Sort

```
for(i=n; i>=1; i--) {
    B[C[A[i]]]=A[i];
    C[A[i]]-=1
}
```

A

2	5	3	0	2	3	0	3
1	2	3	4	5	6	7	8

C

2	2	4	7	7	8
0	1	2	3	4	5

B

1	
2	
3	
4	
5	
6	
7	
8	

Counting Sort

```
for(i=n; i>=1; i--) {
      B[C[A[i]]]=A[i];
      C[A[i]]-=1
}
```

A

2	5	3	0	2	3	0	3
1	2	3	4	5	6	7	8

C

2	2	4	6	7	8
0	1	2	3	4	5

B

1	
2	
3	
4	
5	
6	
7	3
8	

Counting Sort

```
for(i=n; i>=1; i--) {
    B[C[A[i]]]=A[i];
    C[A[i]]-=1
}
```

A

1	2	3	4	5	6	7	8
2	5	3	0	2	3	0	3

C

0	1	2	3	4	5
1	2	4	6	7	8

B

1	
2	0
3	
4	
5	
6	
7	3
8	

Counting Sort

```
for(i=n; i>=1; i--) {
      B[C[A[i]]]=A[i];
      C[A[i]]-=1
}
```

A

2	5	3	0	2	3	0	3
1	2	3	4	5	6	7	8

C

1	2	4	5	7	8
0	1	2	3	4	5

B

1	
2	0
3	
4	
5	
6	3
7	3
8	

Counting Sort

```
for(i=n; i>=1; i--) {
    B[C[A[i]]]=A[i];
    C[A[i]]-=1
}
```

A

2	5	3	0	2	3	0	3
1	2	3	4	5	6	7	8

C

1	2	3	5	7	8
0	1	2	3	4	5

B

1	
2	0
3	
4	2
5	
6	3
7	3
8	

Counting Sort

```
for(i=n; i>=1; i--) {
      B[C[A[i]]]=A[i];
      C[A[i]]-=1
}
```

A

1	2	3	4	5	6	7	8
2	5	3	0	2	3	0	3

C

0	1	2	3	4	5
0	2	3	5	7	8

B

1	0
2	0
3	
4	2
5	
6	3
7	3
8	

Counting Sort

```
for(i=n; i>=1; i--) {
      B[C[A[i]]]=A[i];
      C[A[i]]-=1
}
```

A

2	5	3	0	2	3	0	3
1	2	3	4	5	6	7	8

C

0	2	3	4	7	8
0	1	2	3	4	5

B

1	0
2	0
3	
4	2
5	3
6	3
7	3
8	

Counting Sort

```
for(i=n; i>=1; i--) {
    B[C[A[i]]]=A[i];
    C[A[i]]-=1
}
```

A

1	2	3	4	5	6	7	8
2	5	3	0	2	3	0	3

C

0	1	2	3	4	5
0	2	3	4	7	7

B

1	0
2	0
3	
4	2
5	3
6	3
7	3
8	5

Counting Sort

```
for(i=n; i>=1; i--) {
      B[C[A[i]]]=A[i];
      C[A[i]]-=1
}
```

A

2	5	3	0	2	3	0	3
1	2	3	4	5	6	7	8

C

0	2	2	4	7	7
0	1	2	3	4	5

B

1	0
2	0
3	2
4	2
5	3
6	3
7	3
8	5