Searching

Linear (Sequential) Search

- Linear (Sequential) Search
- Begin at the beginning of the list
- Proceed through the list, sequentially and element by element,
- Until the key is encountered
- Or the end of the list is reached

Linear (Sequential) Search

- Note: we treat a list as a general concept, decoupled from its implementation
- The order of complexity is O(n)
- The list does not have to be in sorted order

Implementation of linear search in C

```
int linear_search(item_type s[], item_type key, int low, int high) {

  int i;

   i = low;

   while ((s[i] != key) && (i < high)) {
       i = i+1;
   }

   if (s[i] == key) {
       return (i);
   }
   else {
       return(-1);
   }
}
```

Binary Search

- The main point to note here is that the elements of the array must be sorted
  - just as the binary search tree was

Binary Search

- The essential idea is to begin in the beginning of the list
- Check to see whether the key is
  - equal to
  - less than
  - greater than
- the middle element

Binary Search

- If key is equal to the middle element, then terminate
- If key is less than the middle element, then search the left half
- If key is greater than the middle element, then search the right half
- Continue until either
  - the key is found or
  - there are no more elements to search

Implementation of Binary_Search

```
Pseudo-code first

Binary_Search(list, key, upper_bound, index, found)

identify sublist to be searched by setting bounds on
  search

REPEAT
   get middle element of list
   if middle element < key
      then reset bounds to make the right sublist
           the list to be searched
      else reset bounds to make the left sublist
           the list to be searched
UNTIL list is empty or key is found
```

Implementation of Binary_Search in Modula

```
CONST n = 100;

TYPE bounds_type = 1..n;
     key_type    = INTEGER;
     list_type   = ARRAY[bounds_type] OF key_type;

PROCEDURE binary_search(list: list_type,
                        key: key_type,
                        bounds: bounds_type,
                        VAR index: bounds_type,
                        VAR found: BOOLEAN);

VAR first, last, mid : bounds_type
```

Implementation of Binary_Search in Modula

```
(* assume at least one element in the list *)
BEGIN
   first :=  1;
   last  := bounds;

   REPEAT
      mid :=  (first + last) DIV 2;
      IF list[mid] < key
         THEN
            first := mid + 1
         ELSE
            last := mid - 1
      END
   UNTIL (first > last) OR (list[mid]  = key);
```

Implementation of Binary_Search in Modula

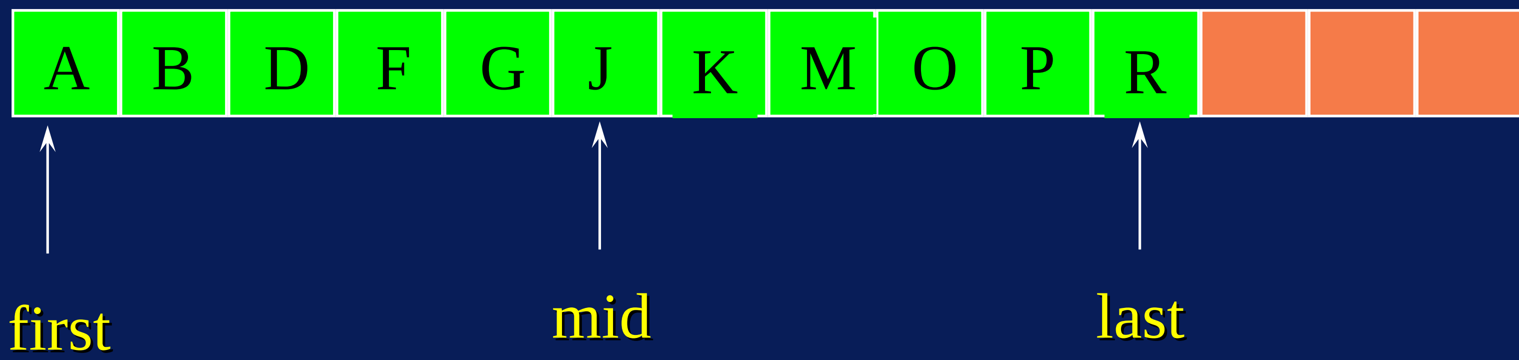
```
   found := key = list[mid];
   index := mid
END binary_search
```

Binary Search

A	B	D	F	G	J	K	M	O	P	R			

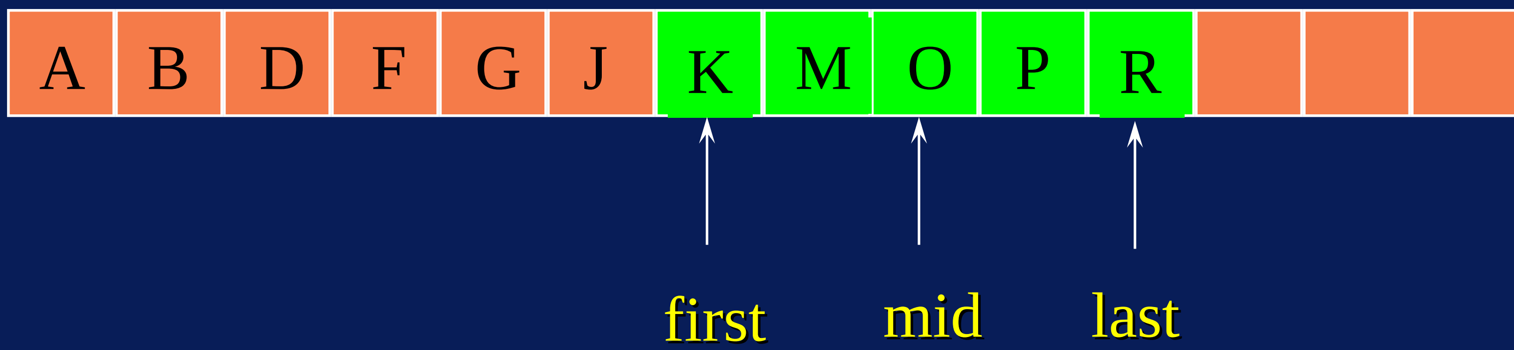
```
first:
last:
mid:
list[mid]:
key:        P
```

Binary Search



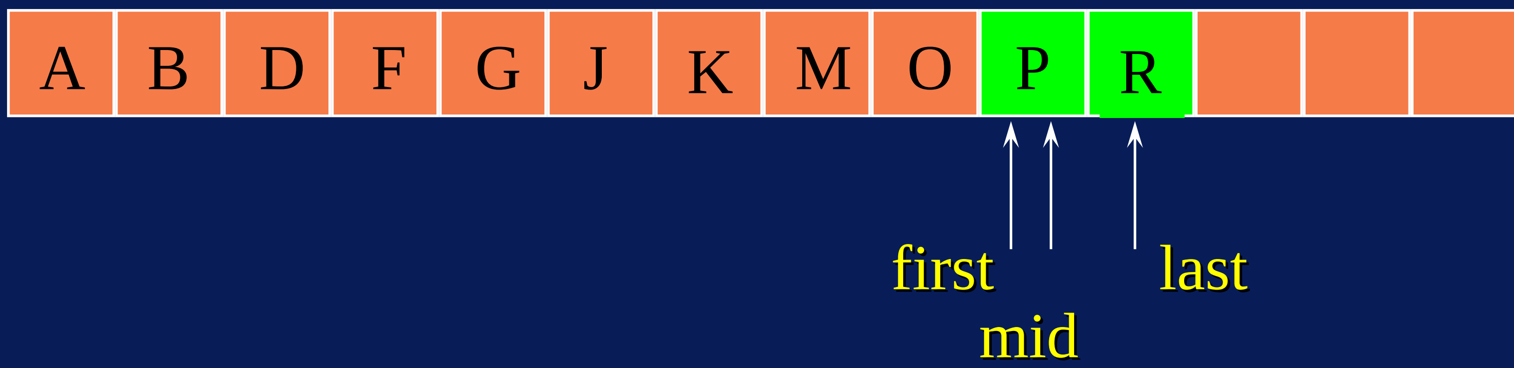

```
first:       1
last:       11
mid:         6
list[mid]:  J
key:         P
```

Binary Search




```
first:       1   7
last:       11  11
mid:         6   9
list[mid]:  J   O
key:         P   P
```

Binary Search




```
first:       1   7   10
last:       11  11   11
mid:         6   9   10
list[mid]:  J   O    P
key:         P   P    P
```

FOUND!

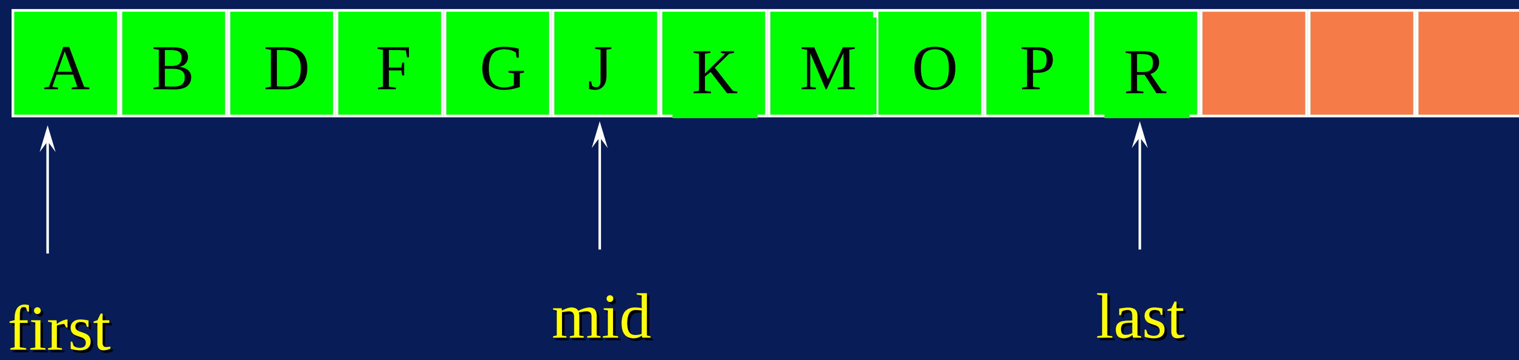
Binary Search

A	B	D	F	G	J	K	M	O	P	R			

```
first:
last:
mid:
list[mid]:
key:        E
```

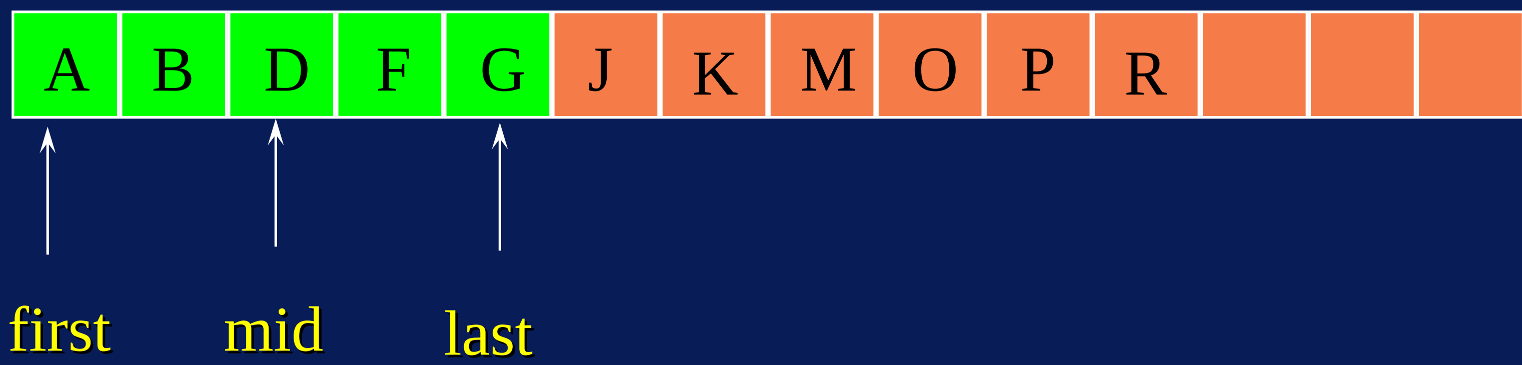
first mid last

Binary Search



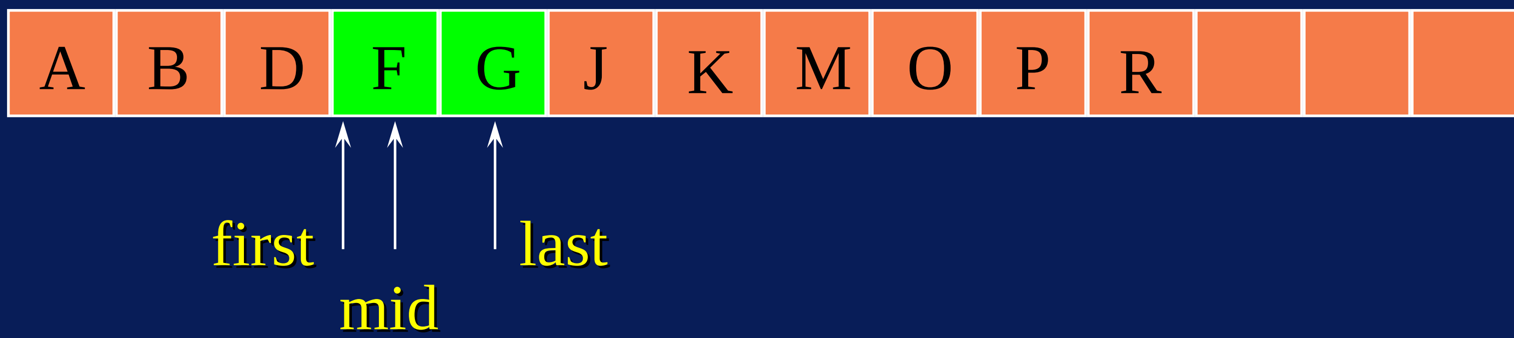

```
first:       1
last:       11
mid:         6
list[mid]: J
key:         E
```

Binary Search



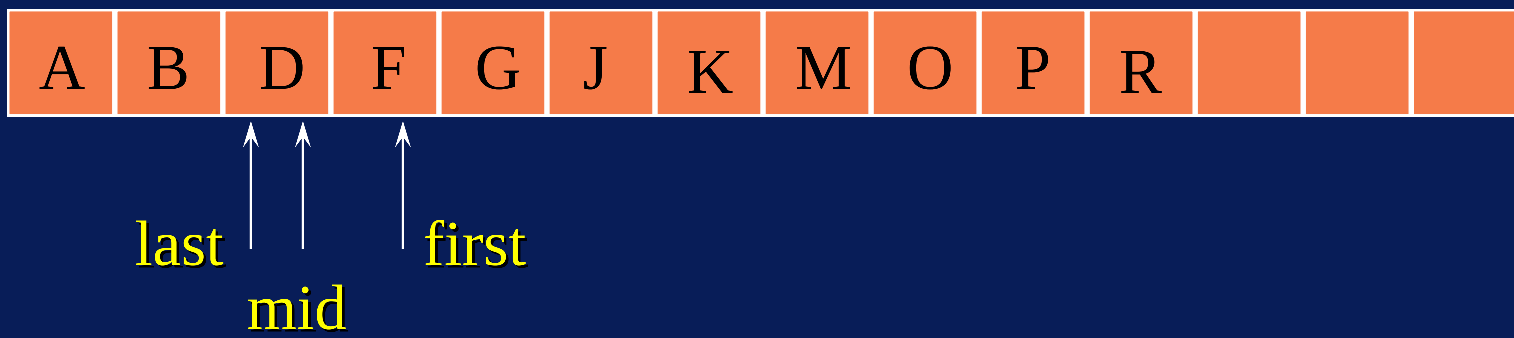

```
first:       1   1
last:       11   5
mid:         6   3
list[mid]:  J    D
key:         E   E
```

Binary Search




```
first:       1   1   4
last:       11   5   5
mid:         6   3   4
list[mid]:  J   D   F
key:         E   E   E
```

Binary Search



first:	1	1	4	4
last:	11	5	5	3
mid:	6	3	4	3
list[mid]:	J	D	F	D
key:	E	E	E	E

first > last: NOT FOUND!

Implementation of binary search in C (recursive approach)

```
typedef char item_type;

int binary_search(item_type s[], item_type key, int low, int high) {

   int mid;

    if (low > high)  return (-1); /* key not found */

    mid = (low + high) / 2;

    if (s[mid] == key) return(mid);

    if (s[mid] > key) {
         return(binary_search(s, key, low, mid-1));
    }
    else {
       return(binary_search(s, key, mid+1, high));
    }
}
```

Sorting Algorithms

Sorting Algorithms

- Bubble Sort
- Quick Sort

Bubble Sort

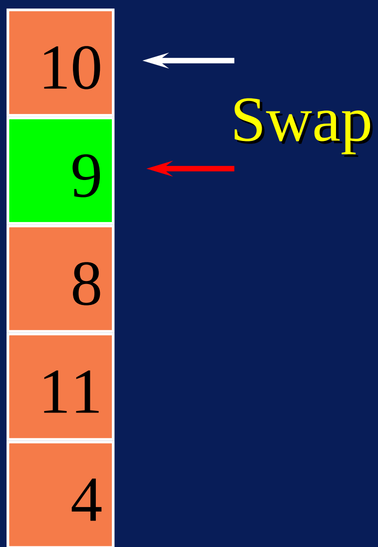
- Assume we are sorting a list represented by an array A of n integer elements
- Bubble sort algorithm in pseudo-code

```
FOR every element in the list,
    proceeding for the first to the last
DO
   WHILE list element > previous list element
      bubble element back (up) the list
      by successive swapping with
      the element just above/prior it
```

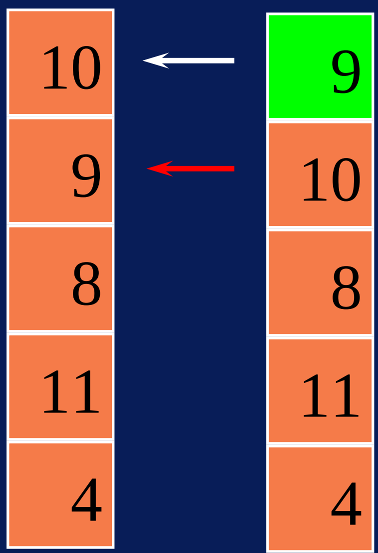
Bubble Sort

10	9	8	11	4

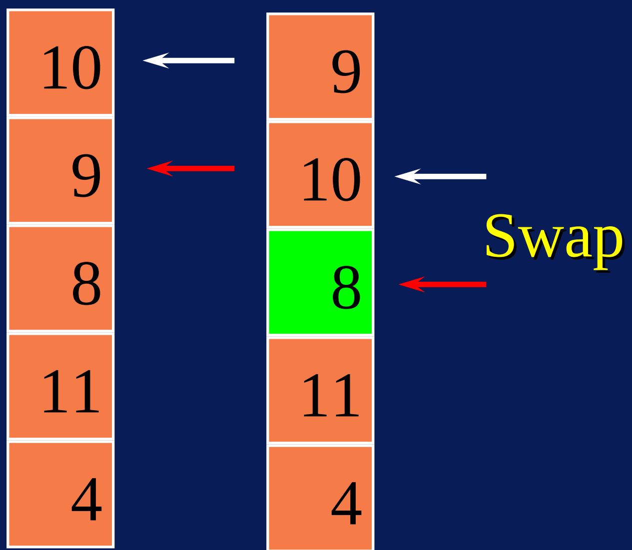
Bubble Sort



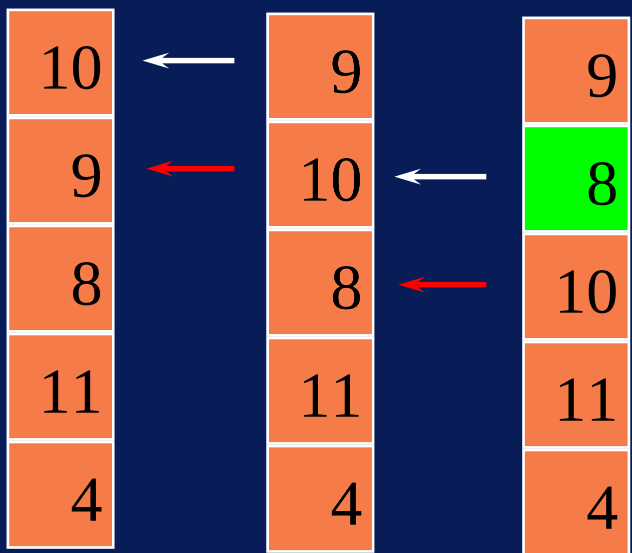
Bubble Sort



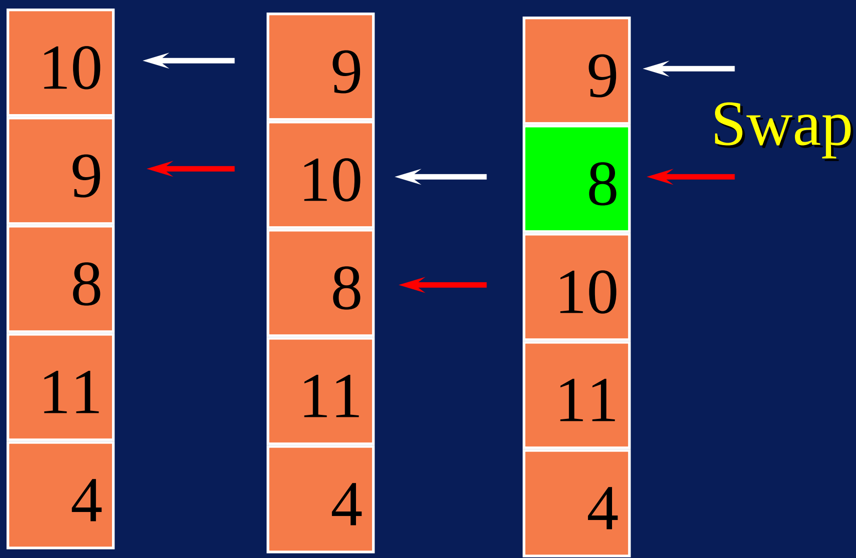
Bubble Sort



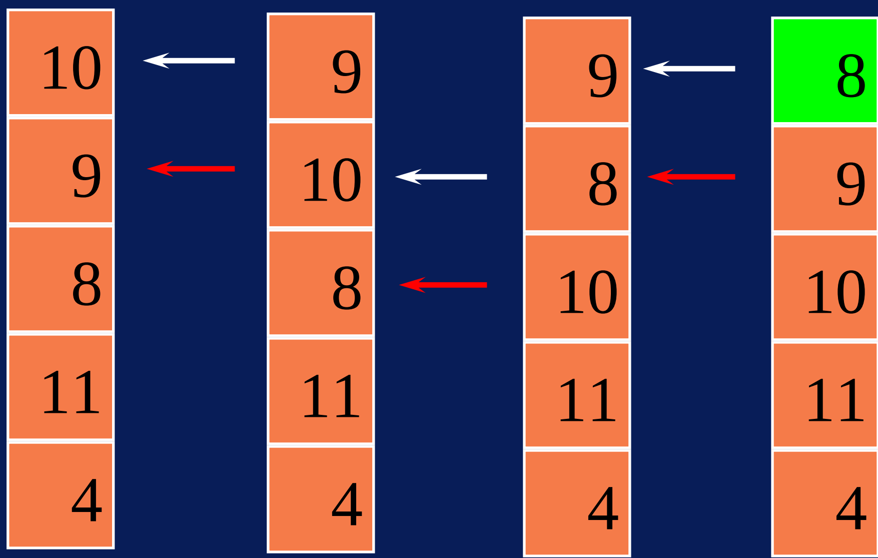
Bubble Sort



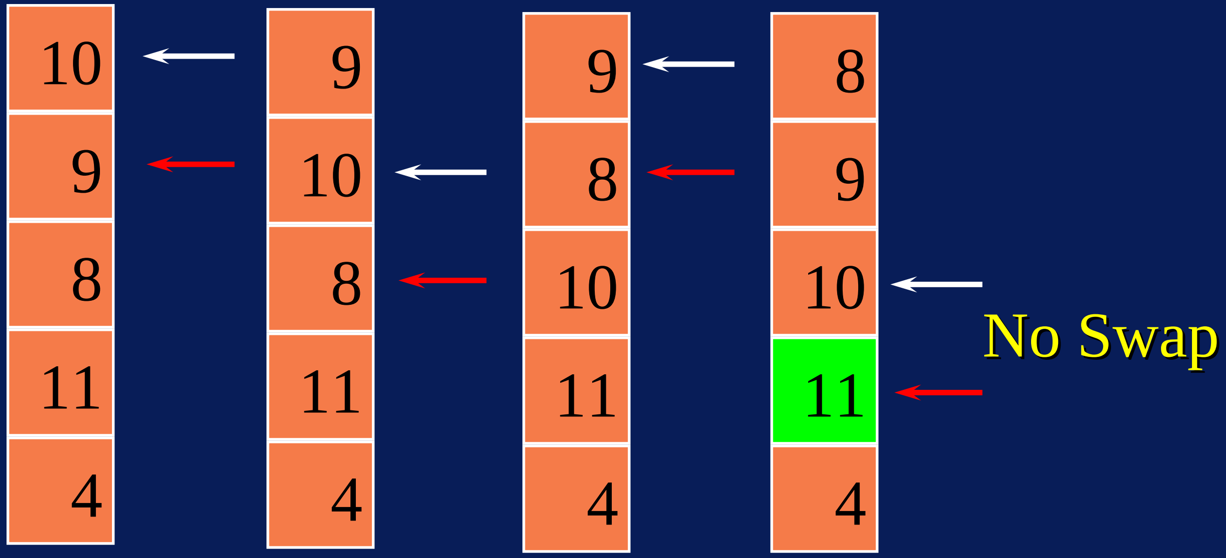
Bubble Sort



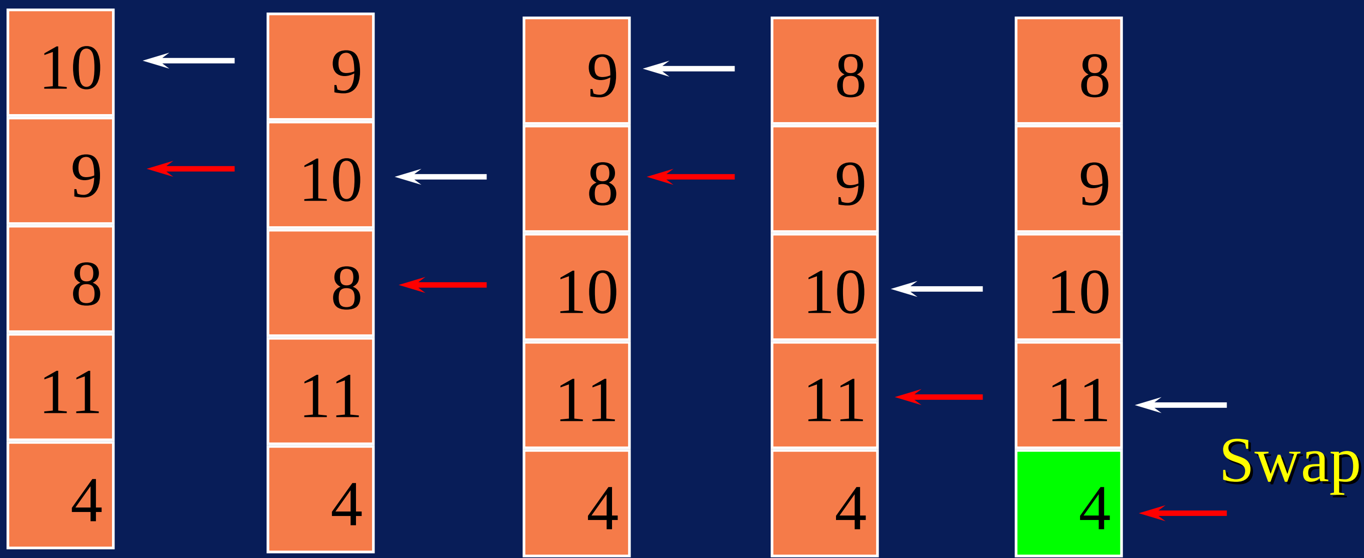
Bubble Sort



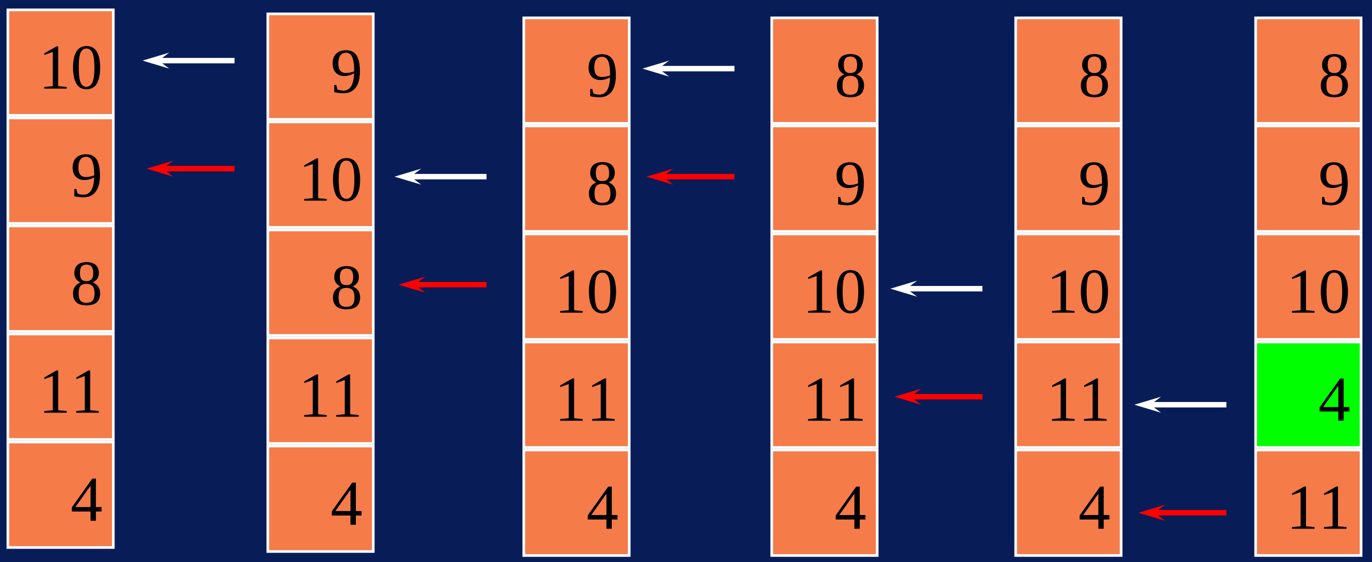
Bubble Sort



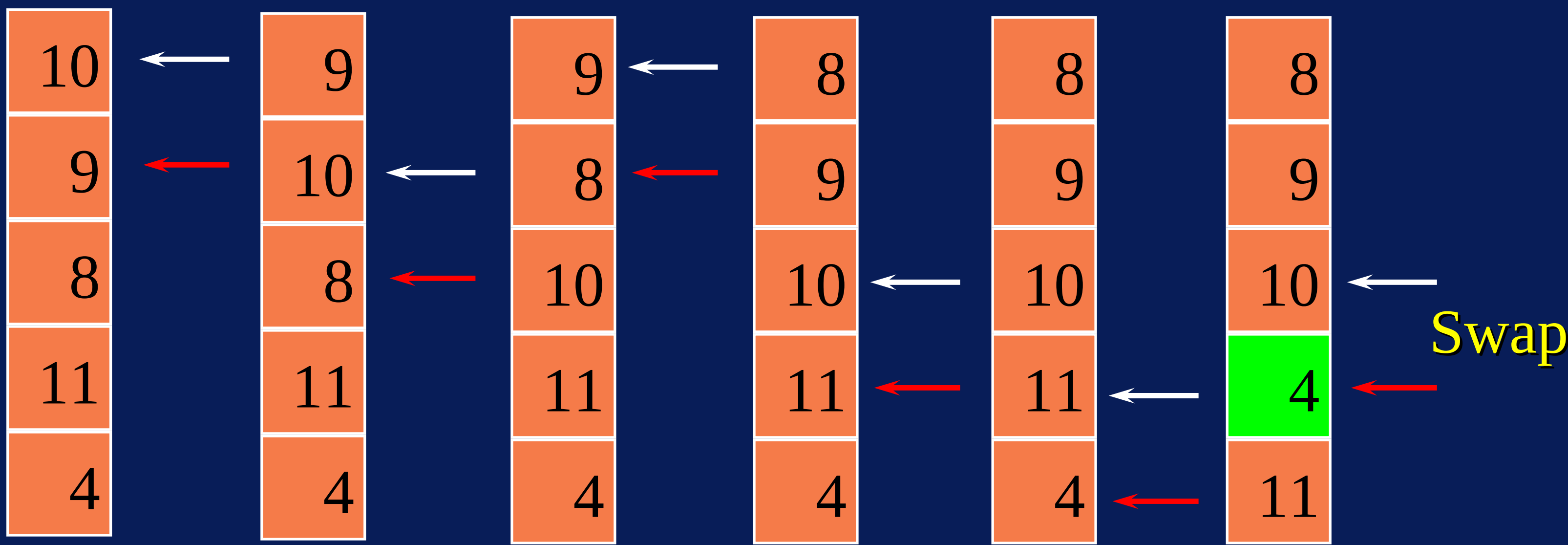
Bubble Sort



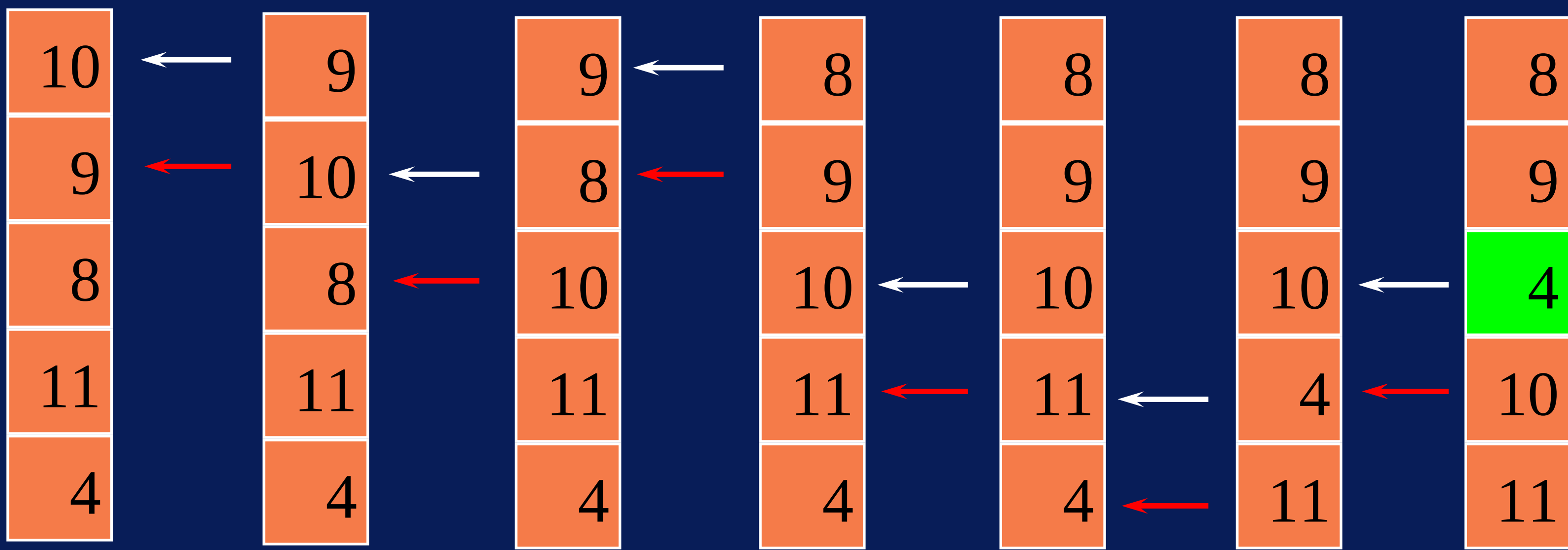
Bubble Sort



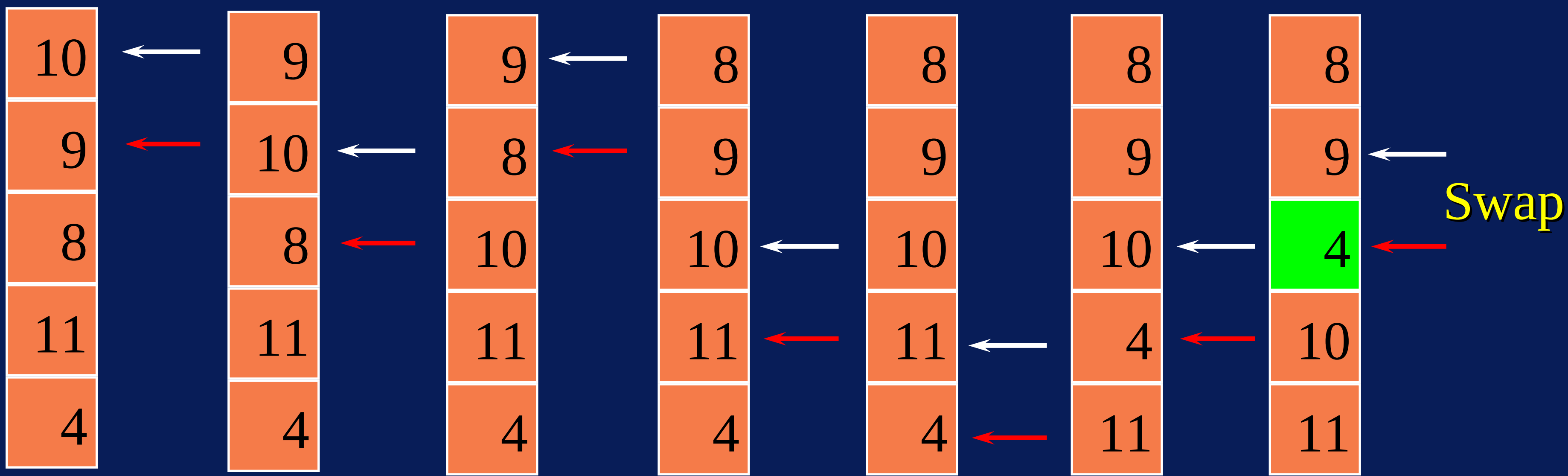
Bubble Sort



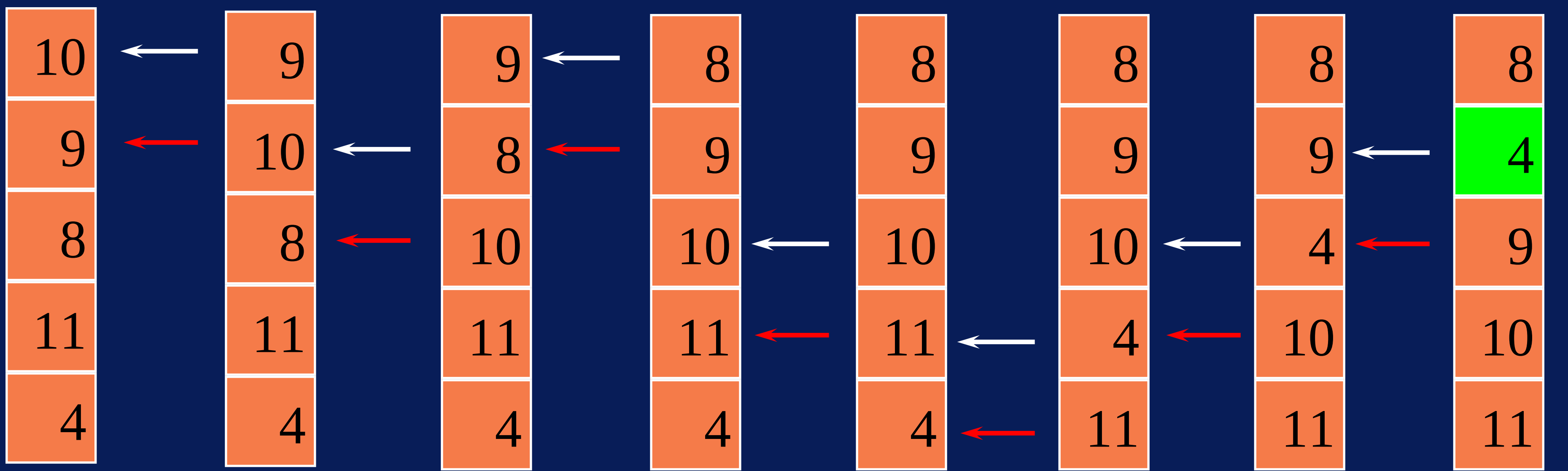
Bubble Sort



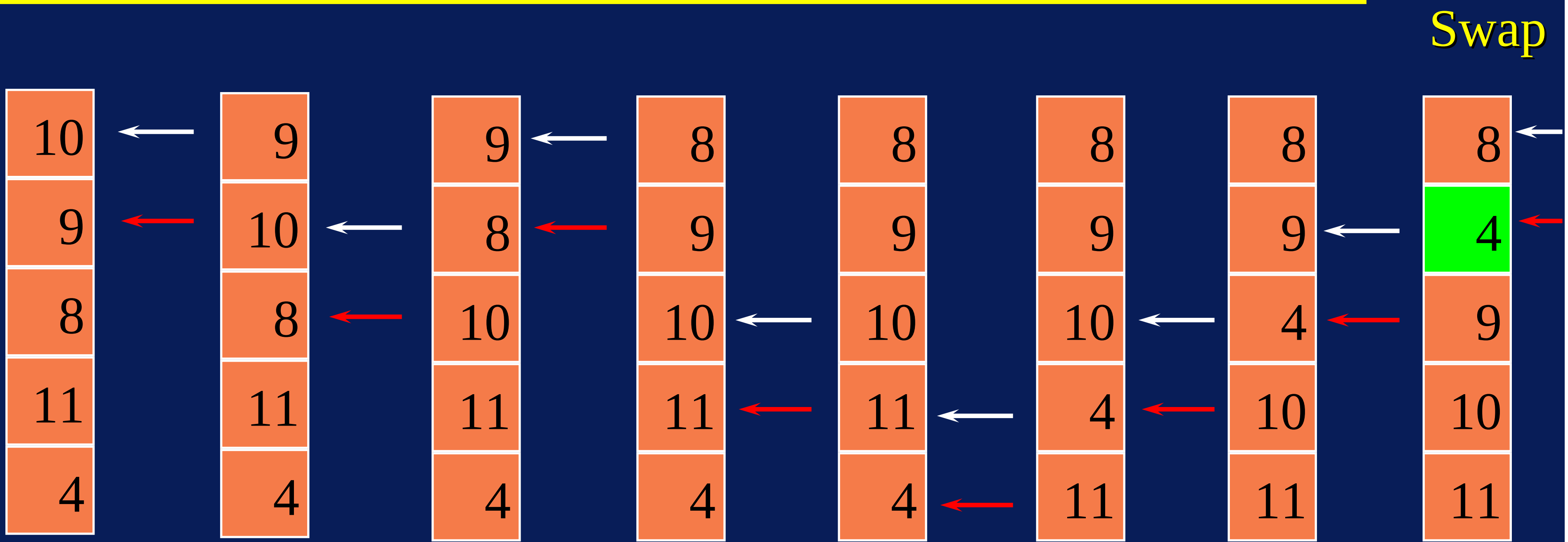
Bubble Sort


10
9
8
11
4
9
10
8
11
4
9
8
10
11
4
8
9
10
11
4
8
9
10
11
4
8
9
10
4
11
8
9
4
10
11
Swap

Bubble Sort

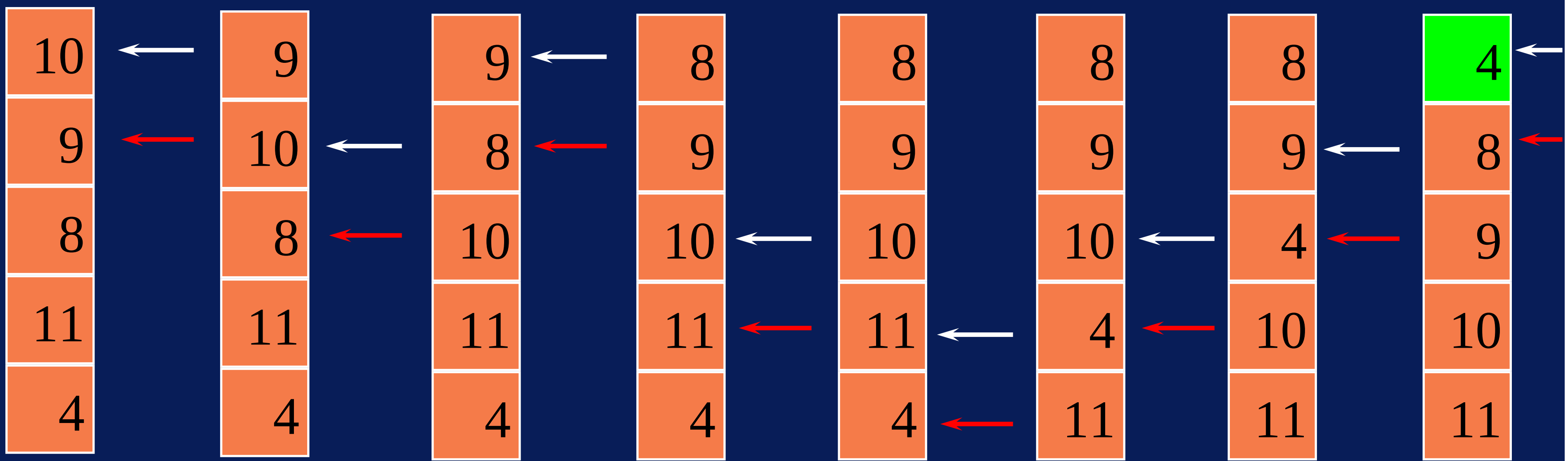

10 9 8 11 4
9 10 8 11 4
9 8 10 11 4
8 9 10 11 4
8 9 10 11 4
8 9 10 4 11
8 9 4 10 11
8 4 9 10 11

Bubble Sort



Swap
10 9 8 11 4
9 10 8 11 4
9 8 10 11 4
8 9 10 11 4
8 9 10 11 4
8 9 10 4 11
8 9 4 10 11
8 4 9 10 11

Bubble Sort



10
9
8
11
4
9
10
8
11
4
9
8
10
11
4
8
9
10
11
4
8
9
10
11
4
8
9
10
4
11
8
9
4
10
11
4
8
9
10
11

Implementation of Bubble_Sort()

```
Int bubble_sort(int *a, int size) {
   int i,j, temp;

   for (i=0; i < size-1; i++) {
      for (j=i; j >= 0; j--) {
         if (a[j] > a[j+1]) {

            /* swap */
            temp = a[j+1];
            a[j+1] = a[j];
            a[j] = temp;
         }
      }
   }
}
```

Bubble Sort

- A few observations:
  - we don’t usually sort numbers; we usually sort records with keys
    - » the key can be a number
    - » or the key could be a string
    - » the record would be represented with a `struct`
  - The swap should be done with a function (so that a record can be swapped)

Bubble Sort

- write a driver program to test:
  - the bubble sort with a swap function
  - the bubble sort with structures
  - compute the order of time complexity of the bubble sort

Selection Sort

- Example:

  - Shaded elements are selected
  - Boldface elements are in order

Initial Array	29	10	14	37	13
After 1st swap	29	10	14	13	**37**
After 2nd swap	13	10	14	**29**	**37**
After 3rd swap	13	10	**14**	**29**	**37**
After 4th swap	**10**	**13**	**14**	**29**	**37**

Selection Sort

- Assume we are sorting a list represented by an array A of n integer elements
- Selection sort algorithm in pseudo-code

```
last = n-1
Do
    Select largest element from a[0..last]
    Swap it with a[last]
    last = last-1
While (last >= 1)
```

Selection Sort

```
typedef  int  DataType;

void selectionSort(DataType a[] , int n) {

   DataType temp;
   int index_of_largest, index, last;

   for(last= n-1; last >= 1;  last--) {

       // select largest item in a[0..last]
      index_of_largest = 0;
      for(index=1; index <= last; index++) {
         if (a[index] > a[index_of_largest])
            index_of_largest = index;
      }

      // swap largest item with last element
      temp = a[index_of_largest];
      a[index_of_largest] = a[last]);
      a[last]) = temp;
   }
}
```

Quicksort

- The Quicksort algorithm was developed by C.A.R. Hoare. It has the best average behaviour in terms of complexity:

  Average case: $O(n \log_2 n)$
  Worst case: $O(n^2)$

Quicksort

- Given a list of elements,
- take a partitioning element
- and create a (sub)list
  - such that all elements to the left of the partitioning element are less than it,
  - and all elements to the right of it are greater than it.
- Now repeat this partitioning effort on each of these two sublists

Quicksort

- And so on in a recursive manner until all the sublists are empty, at which point the (total) list is sorted
- Partitioning can be effected simultaneously, scanning left to right and right to left, interchanging elements in the wrong parts of the list
- The partitioning element is then placed between the resultant sublists (which are then partitioned in the same manner)

Implementation of Quicksort()

```
In pseudo-code first

If anything to be partitioned
   choose a pivot
   DO
      scan from left to right until we find an element
       > pivot: i points to it

       scan from right to left until we find an element
       < pivot: j points to it

       IF  i < j
          exchange ith and jth element
   WHILE i <= j
```

Implementation of Quicksort()

exhange pivot and j^{th} element

partition from 1^{st} to j^{th} elements

partition from i^{th} to r^{th} elements

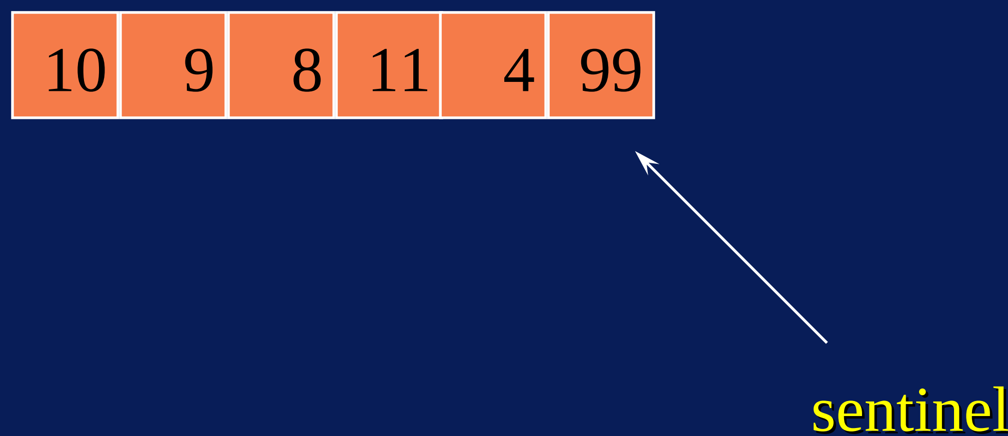
Implementation of Quicksort()

```
/* simple quicksort to sort an array of integers */

void quicksort (int A[], int L, int R)
{
   int i, j, pivot;

/* assume A[R] contains a number > any element,    */
/* i.e. it is a sentinel.                          */
```

Implementation of Quicksort()

```
if ( R > L) {
   i = L; j = R;
   pivot = A[i];
   do {
      while (A[i] <= pivot) i=i+1;
      while ((A[j] >= pivot) && (j>l)) j=j-1;
      if (i < j) {
         exchange(A[i],A[j]); /*between partitions*/
         i = i+1; j = j-1;
      }
   } while (i <= j);
   exchange(A[L], A[j]); /* reposition pivot */
   quicksort(A, L, j);
   quicksort(A, i, R);   /*includes sentinel*/
}
```

Quicksort



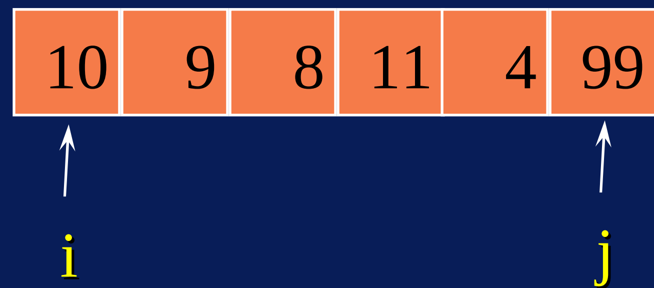
Quicksort

10	9	8	11	4	99

```
QS(A, , )

L:
R:
i:
j:
pivot:
```

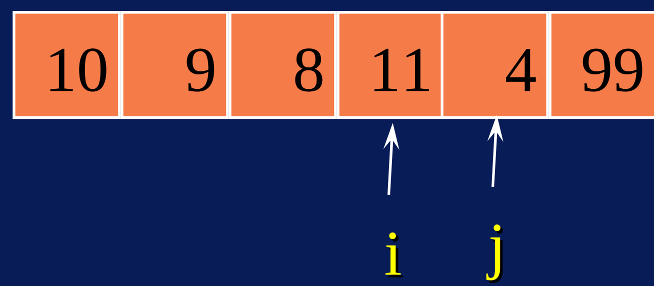
i j

Quicksort



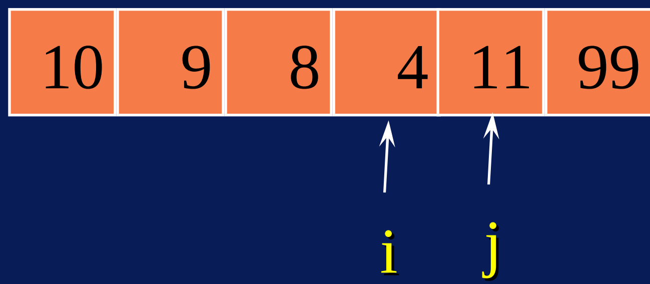

```
QS(A,1,6)

L:       1
R:       6
i:       1
j:       6
pivot: 10
```

Quicksort



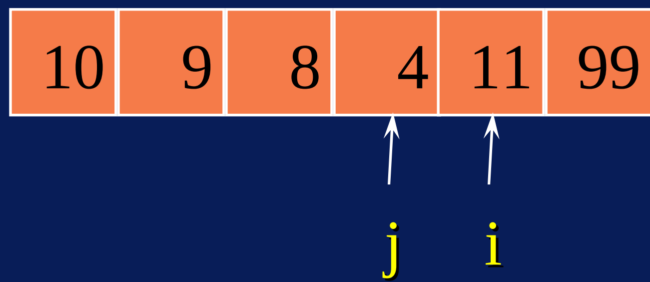

```
QS(A,1,6)

L:       1
R:       6
i:       1 2 3 4
j:       6 5
pivot: 10
```

Quicksort



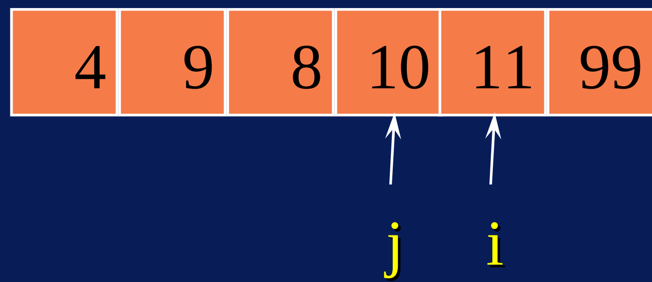

```
QS(A,1,6)

L:      1
R:      6
i:      1 2 3 4
j:      6 5
pivot: 10
```

Quicksort




```
QS(A,1,6)

L:      1
R:      6
i:      1 2 3 4 5
j:      6 5 4
pivot: 10
```

Quicksort




```
QS(A,1,6)

L:       1
R:       6
i:       1 2 3 4 5
j:       6 5 4
pivot: 10
```

Quicksort

4	9	8	10	11	99

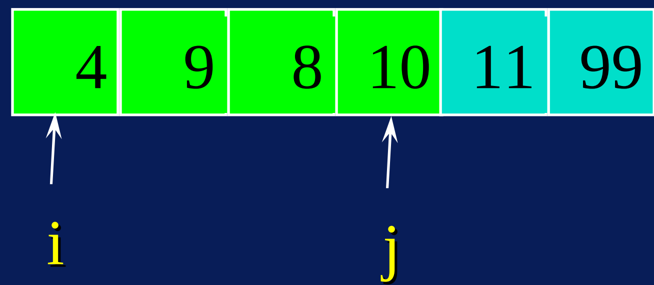
i j

```
QS(A,1,6)

L:      1
R:      6
i:      1 2 3 4 5
j:      6 5 4
pivot: 10
```

```
QS(A,1,4)

L:      1
R:      4
i:
j:
pivot:  4
```

```
QS(A,5,6)

L:      5
R:      6
i:
j:
pivot:  11
```

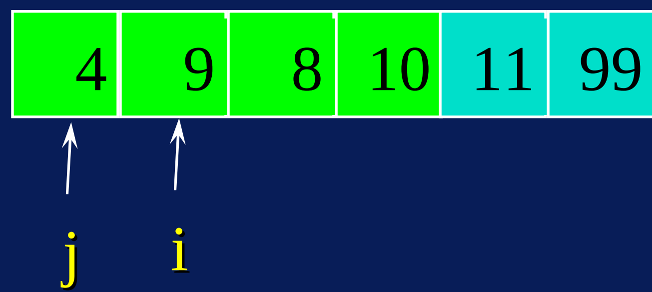
Quicksort




```
QS(A,1,4)

L:      1
R:      4
i:      1
j:      4
pivot:  4
```

```
QS(A,5,6)

L:      5
R:      6
i:      5
j:      6
pivot:  11
```

Quicksort




```
QS(A,1,4)

L:      1
R:      4
i:      1 2
j:      4 3 2 1
pivot:  4
```

```
QS(A,5,6)

L:      5
R:      6
i:      5
j:      6
pivot:  11
```

Quicksort

4	9	8	10	11	99

i j

QS(A,1,4)

L: 1
R: 4
i: 1 2
j: 4 3 2 1
pivot: 4

QS(A,1,1)

L: 1
R: 1
i:
j:
pivot: 4

QS(A,2,4)

L: 2
R: 4
i:
j:
pivot: 9

QS(A,5,6)

L: 5
R: 6
i: 5
j: 6
pivot: 11

Quicksort

4	9	8	10	11	99

↑ ↑

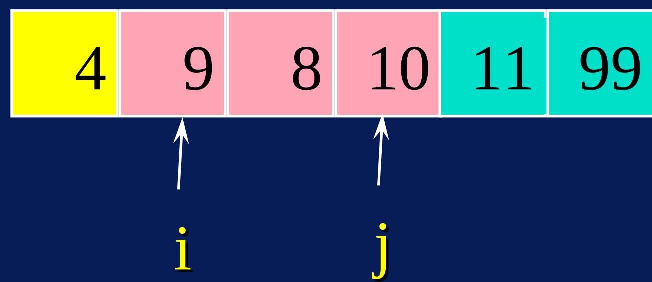
i j

```
QS(A,1,1)

L:      1
R:      1
i:
j:
pivot:  4
```

```
QS(A,2,4)

L:      2
R:      4
i:
j:
pivot:  9
```

```
QS(A,5,6)

L:      5
R:      6
i:      5
j:      6
pivot:  11
```

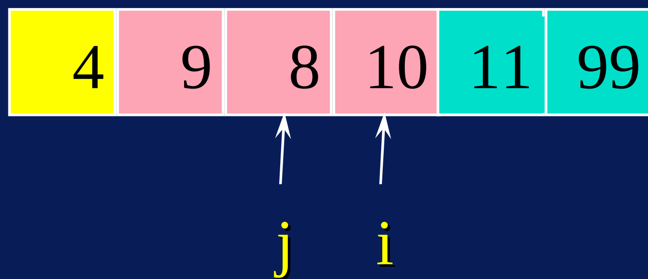
Quicksort




```
QS(A,2,4)

L:      2
R:      4
i:      2
j:      4
pivot:  9
```

```
QS(A,5,6)

L:      5
R:      6
i:      5
j:      6
pivot:  11
```

Quicksort




```
QS(A,2,4)

L:      2
R:      4
i:      2 3 4
j:      4 3
pivot:  9
```

```
QS(A,5,6)

L:      5
R:      6
i:      5
j:      6
pivot:  11
```

Quicksort

4	8	9	10	11	99

i j

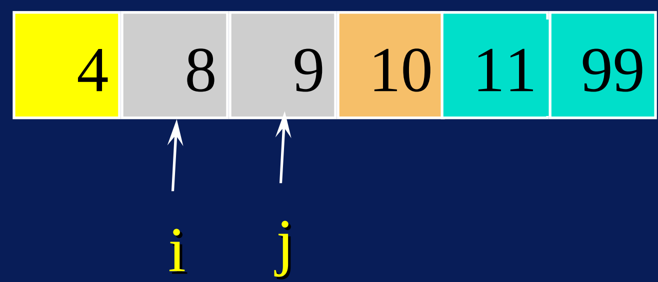
```
QS(A,2,4)

L:      2
R:      4
i:      2 3 4
j:      4 3
pivot:  9
```

```
QS(A,2,3)

L:      2
R:      3
i:
j:
pivot:  8
```

```
QS(A,4,4)

L:      4
R:      4
i:
j:
pivot:  10
```

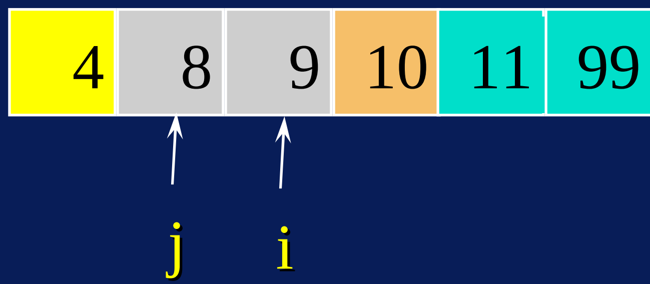
```
QS(A,5,6)

L:      5
R:      6
i:      5
j:      6
pivot:  11
```

Quicksort



QS(A,2,3)		QS(A,4,4)		QS(A,5,6)	
L:	2	L:	4	L:	5
R:	3	R:	4	R:	6
i:	2	i:		i:	5
j:	3	j:		j:	6
pivot:	8	pivot:	10	pivot:	11

Quicksort




```
QS(A,2,3)

L:      2
R:      3
i:      2 3
j:      3 2
pivot:  8
```

```
QS(A,4,4)

L:      4
R:      4
i:
j:
pivot:  10
```

```
QS(A,5,6)

L:      5
R:      6
i:      5
j:      6
pivot:  11
```

Quicksort

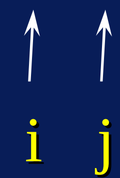
4	8	9	10	11	99

i j

QS(A,2,3)		QS(A,2,2)		QS(A,3,3)		QS(A,4,4)		QS(A,5,6)	
L:	2	L:	2	L:	3	L:	4	L:	5
R:	3	R:	2	R:	3	R:	4	R:	6
i:	2 3	i:		i:		i:		i:	5
j:	3 2	j:		j:		j:		j:	6
pivot:	8	pivot:	8	pivot:	9	pivot:	10	pivot:	11

Quicksort

4	8	9	10	11	99



QS(A,4,4)

L: 4
R: 4
i:
j:
pivot: 10

QS(A,5,6)

L: 5
R: 6
i: 5
j: 6
pivot:
11

Quicksort

4	8	9	10	11	99

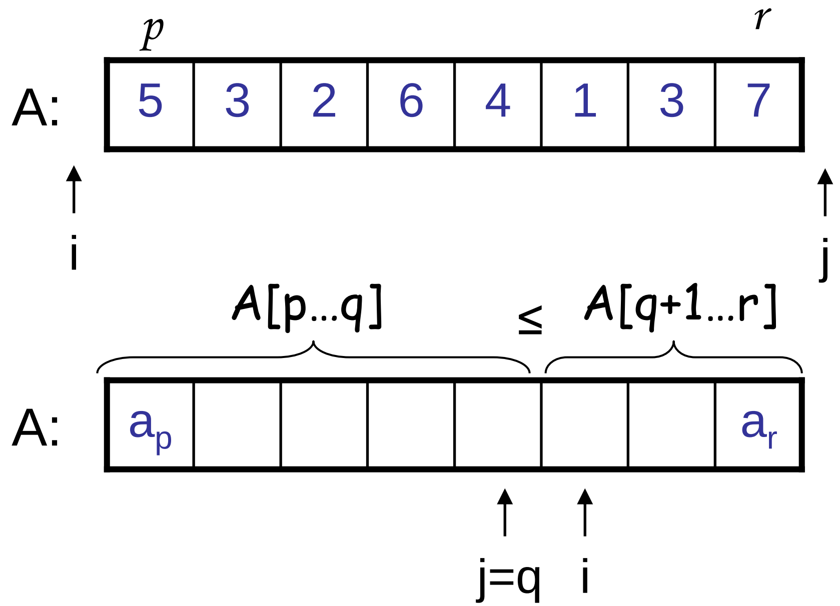
i j

QS(A,5,6)		QS(A,5,5)		QS(A,6,6)	
L:	5	L:	5	L:	6
R:	6	R:	5	R:	6
i:	5	i:		i:	
j:	6	j:		j:	
pivot:	11	pivot:	11	pivot:	99

Partitioning the Array

Alg. PARTITION (A, p, r)

```
x ← A[p]
i ← p - 1
j ← r + 1
while TRUE
        do repeat j ← j - 1
                until A[j] ≤ x
        do  repeat i ← i + 1
                until A[i] ≥ x
         if i < j
                then exchange A[i] ↔ A[j]
         else return j
```



Each element is visited once!

Running time: $O(n)$

$n = r - p + 1$

Recurrence

Alg.: QUICKSORT(A, p, r) Initially: p=1, r=n

if p < r

then $q \leftarrow$ PARTITION(A, p, r)

QUICKSORT (A, p, q)

QUICKSORT (A, q+1, r)

Recurrence:

$$T(n) = T(q) + T(n - q) + n$$

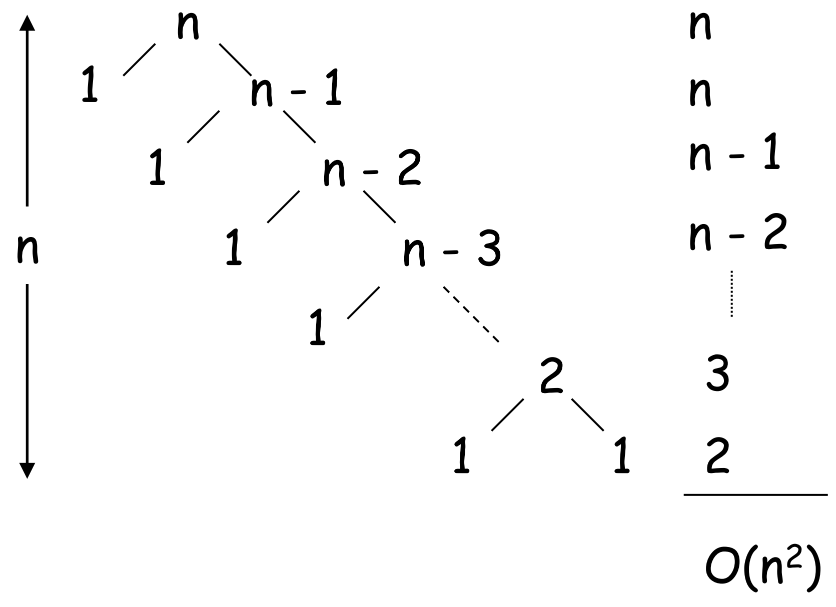
Worst Case Partitioning

- Worst-case partitioning
  - One region has one element and the other has n – 1 elements
  - Maximally unbalanced
- Recurrence: q=1

  T(n) = T(1) + T(n – 1) + n,

  T(1) = O(1)

  T(n) = T(n – 1) + n



When does the worst case happen?

Worst Case Partitioning

- Worst-case partitioning
  - One region has one element and the other has $n - 1$ elements
  - Maximally unbalanced
- Recurrence: $q=1$

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

$T(1) = c1$

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

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

$= T(n - 3) + 3n-3$

$= T(n - 4) + 4n-6$

$=T(n-k)+kn-(1+2+\ldots k-1)$

$=T(n-k)+kn-k(k-1)/2$

If $n-k=1 \Rightarrow k=n$

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

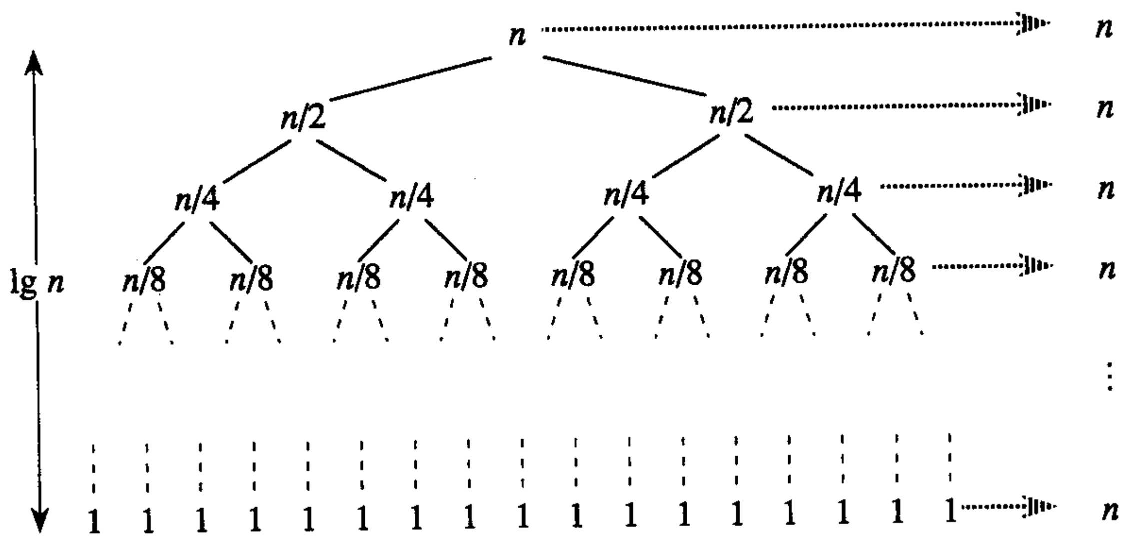
$\Rightarrow O(n^2)$

Best Case Partitioning

- Best-case partitioning
  - Partitioning produces two regions of size $n/2$
- Recurrence: q=n/2

$T(n) = 2T(n/2) + n$,

$T(n) = O(n\lg n)$ (Master theorem)



Best Case Partitioning

- Best-case partitioning
  - Partitioning produces two regions of size $n/2$
- Recurrence: q=n/2

$T(n) = 2T(n/2) + n$ & $T(1)=c1$

$=4T(n/4) + 2n$

$=8T(n/8) + 3n$

$=2^kT(n/2^k) + kn$

$n/2^k = 1 \Rightarrow k=\log_2 n$

$=2^{\log 2n}T(1) + \log_2 n \,.\, n$

$= n.c1 + n\log_2 n$

$\Rightarrow O(n\log n)$