04-630
Data Structures and Algorithms for Engineers

Lecture 5: Searching and Sorting Algorithms

Agenda

Searching and Sorting Algorithms

- Linear Search & Binary Search
- In-place sorts
  - Bubble Sort
  - Selection Sort
  - Insertion Sort
- Not-in-place sort
  - Quicksort
  - Mergesort
- Characteristics of a good sort

SEARCHING ALGORITHMS

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 *(element being searched for)* is encountered
  or
  Until 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); //returns a negative index in this case.
   }
}
```

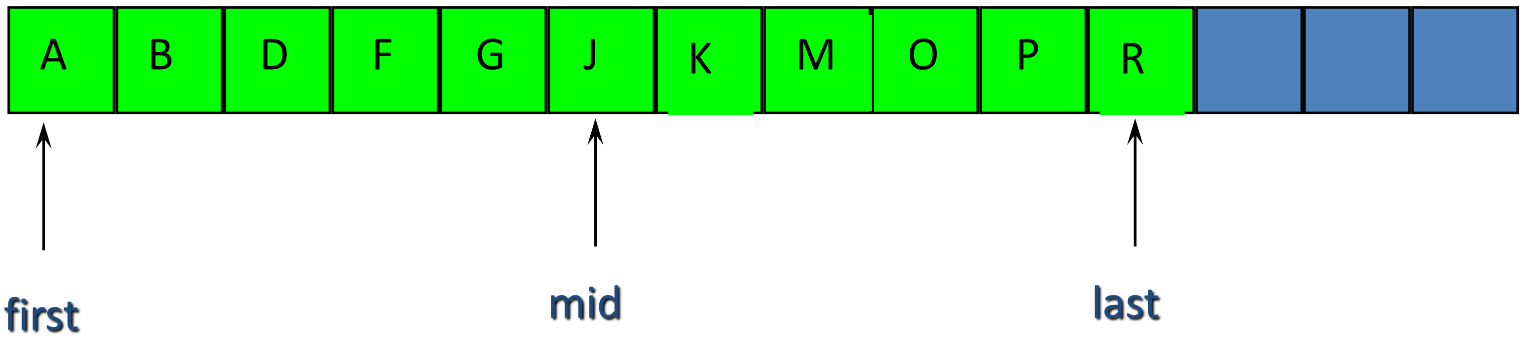
Binary Search

- If the list is sorted, we can use a more efficient $\mathrm{O}(\log_2(n))$ search strategy

- Check to see whether the key is
  - equal to
  - less than
  - greater than

the middle element

Binary Search



Binary Search

- If key is equal to the middle element, then terminate (found)
- 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

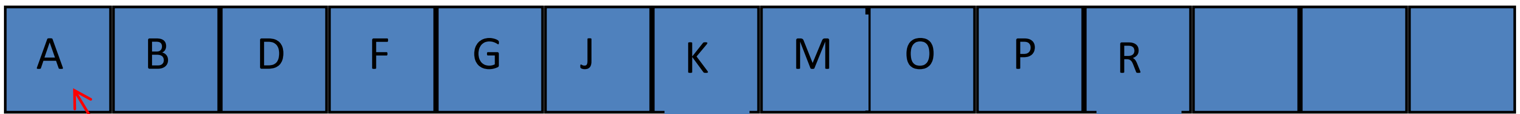
Pseudo-code

```
binary_search(list, key, lower_bound, upper_bound)

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
```

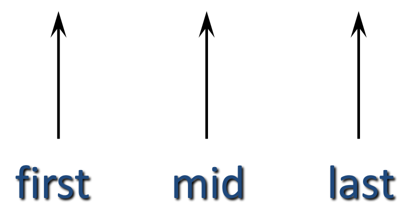
Implementation of binary search in C (iterative approach)

```
typedef char item_type;

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

   int first, last, mid;

    first = low;
    last  = high;

    do {
        mid = (first + last) / 2;
        if (s[mid] < key) { //search top half
            first = mid + 1;
        }
        else {//search bottom half
            last = mid - 1;
        }
    } while ( (first <= last) && (s[mid] != key) );

    if (s[mid] == key)
        return (mid);
    else
        return (-1); //returns a negative index in this case.
}
```

Binary Search

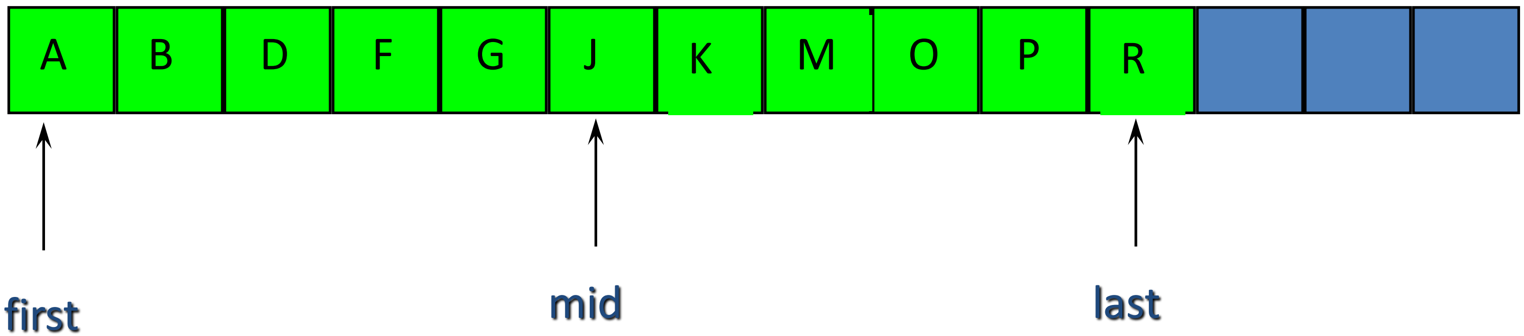



```
first:
last:
mid:
list[mid]:
key:        P
```



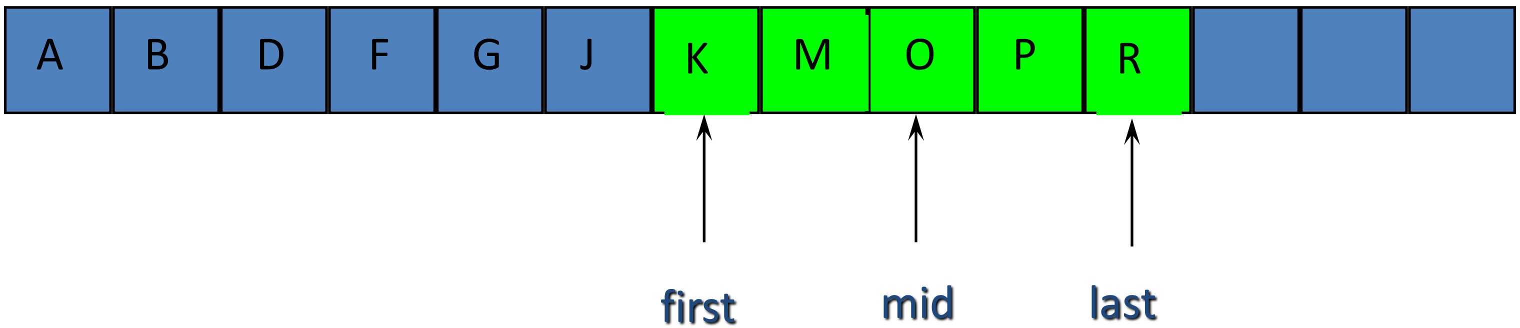
Should really have written these as 'P', 'A', 'B', ... because they are character values

Binary Search



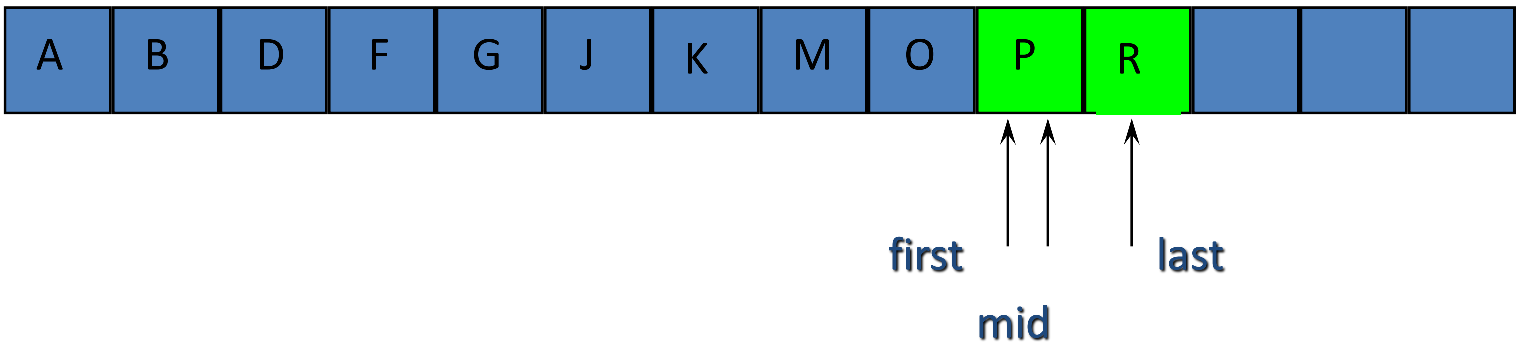

```
first:      1
last:      11
mid:        6
list[mid]: J
key:        P
```

Binary Search




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

Binary Search



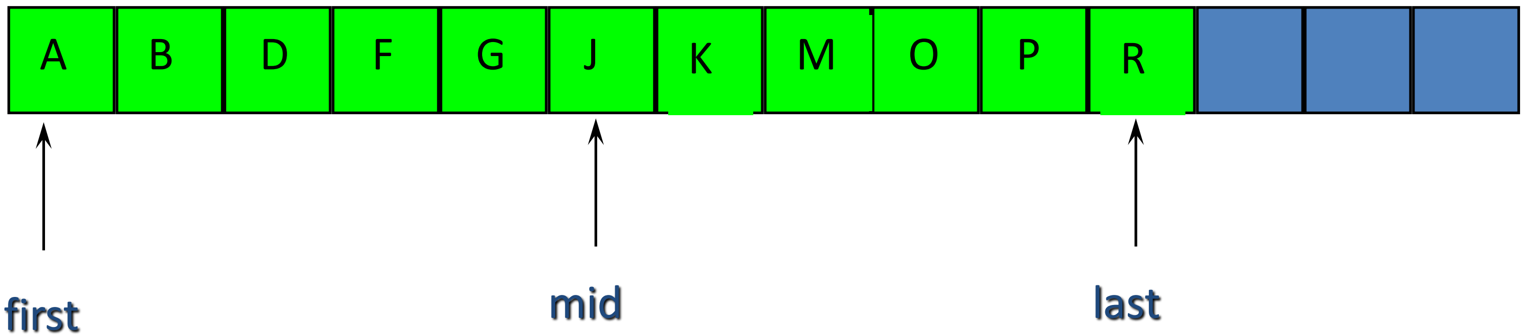
`first:`	1	7	10	
`last:`	11	11	11	
`mid:`	6	9	10	
`list[mid]:`	J	O	P	FOUND!
`key:`	P	P	P	

Binary Search

A	B	D	F	G	J	K	M	O	P	R			

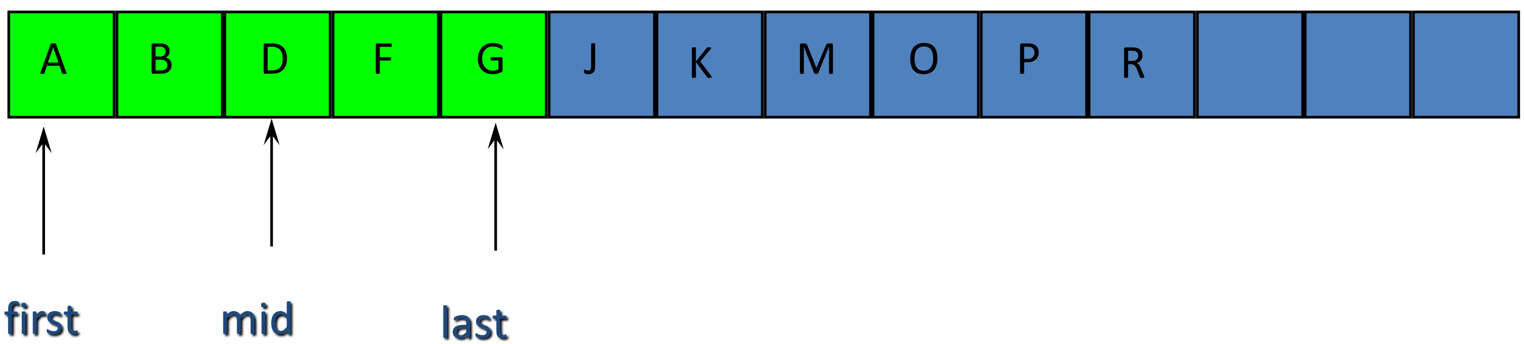
```
first:
last:
mid:
list[mid]:
key:        E
```

Binary Search



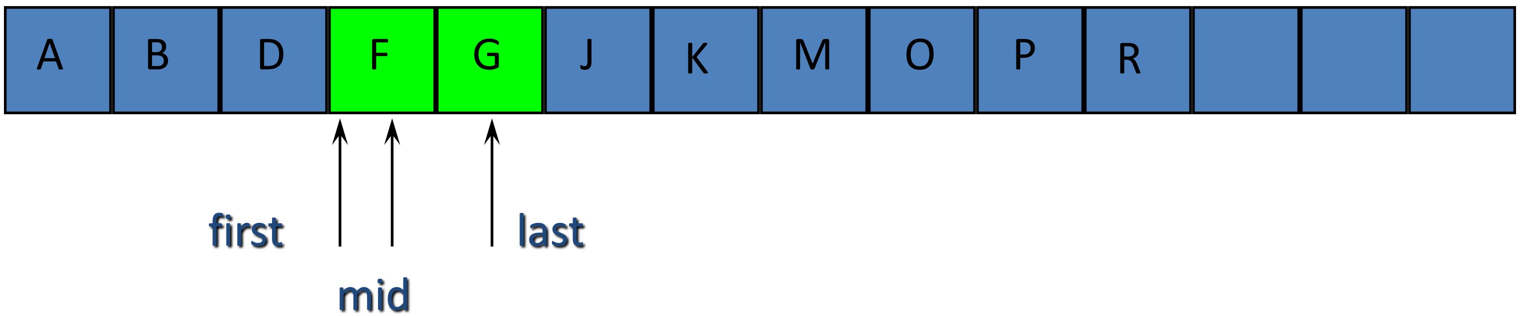

```
first:      1
last:      11
mid:        6
list[mid]: J
key:        E
```

Binary Search



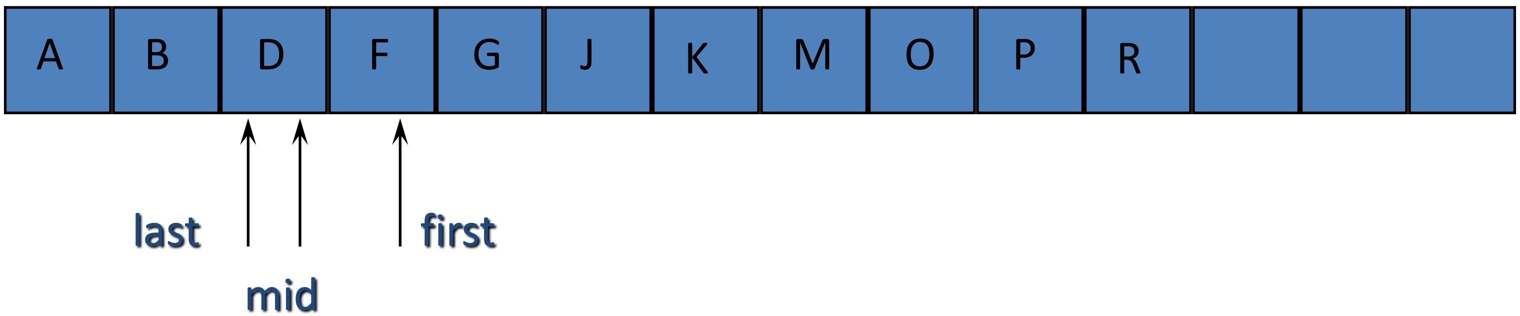

```
first:       1   1
last:       11   5
mid:         6   3
list[mid]: J   D
key:         E   E
```

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)); //search bottom half
    }
    else {
       return(binary_search(s, key, mid+1, high)); //search top half
    }
}
```

Agenda

Searching and Sorting Algorithms

- Linear Search & Binary Search
- In-place sorts
  - Bubble Sort
  - Selection Sort
  - Insertion Sort
- Not-in-place sort
  - Quicksort
  - Mergesort
- Characteristics of a good sort

SORTING ALGORITHMS

The Sorting Problem

Input: A sequence of *n* numbers $< a_1, a_2, \ldots a_n >$

Output: the permutation (reordering) of the input sequence such that $a_1 \leq a_2 \leq \ldots \leq a_n$

Sorting Algorithms

- In-place sorts
  - Small number of elements stored outside the input data structure
  - Additional space requirements $O(1)$
  - **Tradeoff**: more computationally-complex algorithms (slower sorts)
    - Bubble Sort
    - Selection Sort
    - Insertion Sort

Bubble Sort

- Assume we are sorting a list represented by an array A of n integer elements

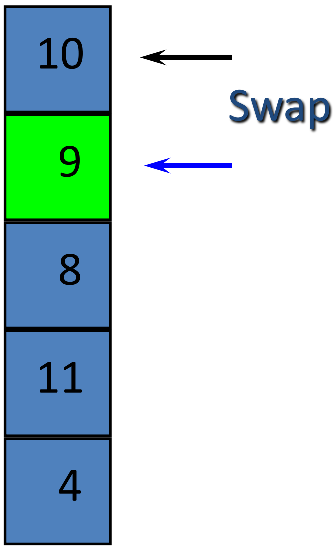
- Bubble sort algorithm in pseudo-code

```
FOR every element in the list,
    proceeding from the first to the last

   WHILE list element > previous list element
      bubble element back (up) the list
      by successive swapping with
      the element just above/prior it
```

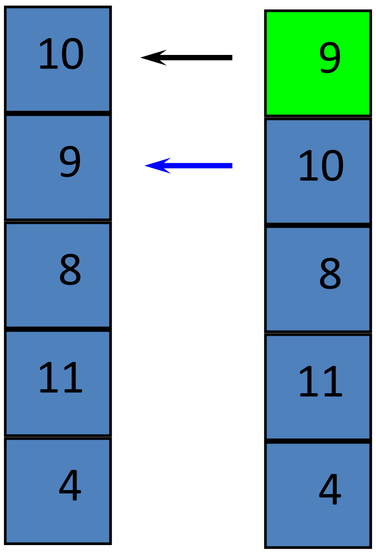
Bubble Sort

10	9	8	11	4

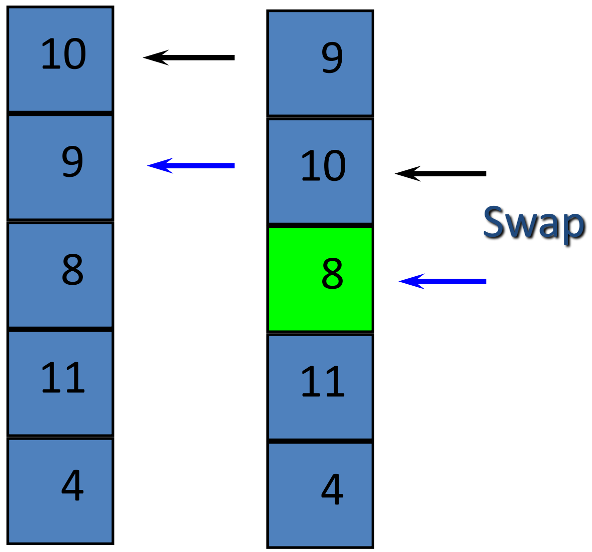
Bubble Sort



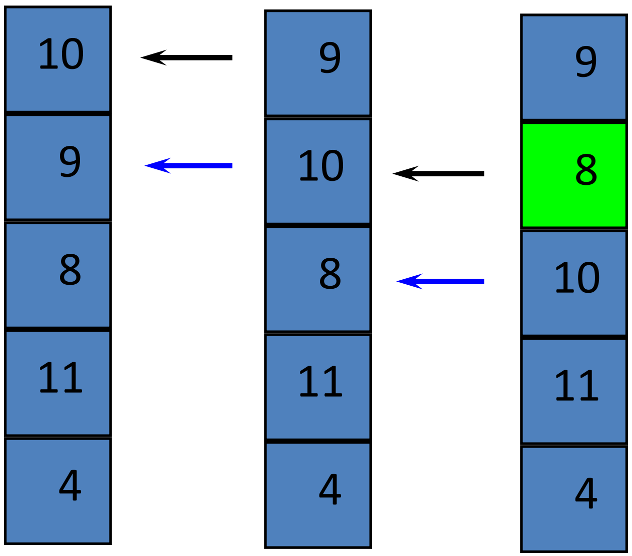
Bubble Sort



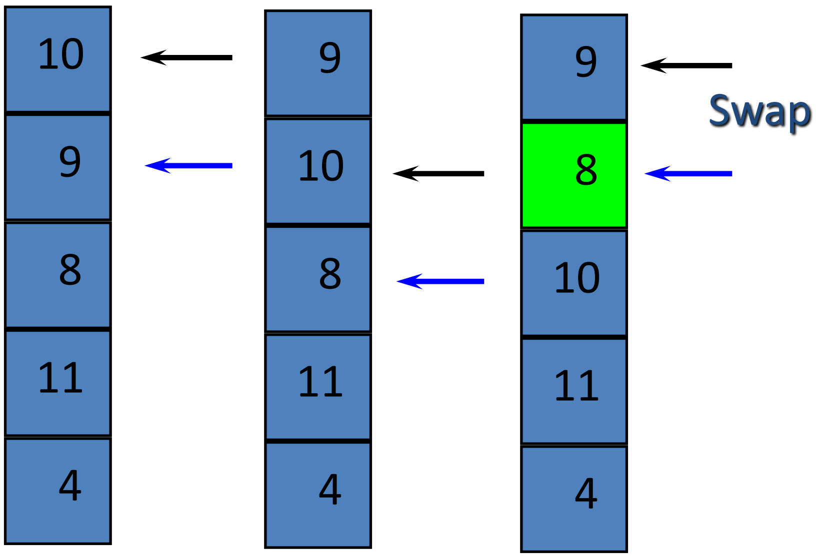
Bubble Sort



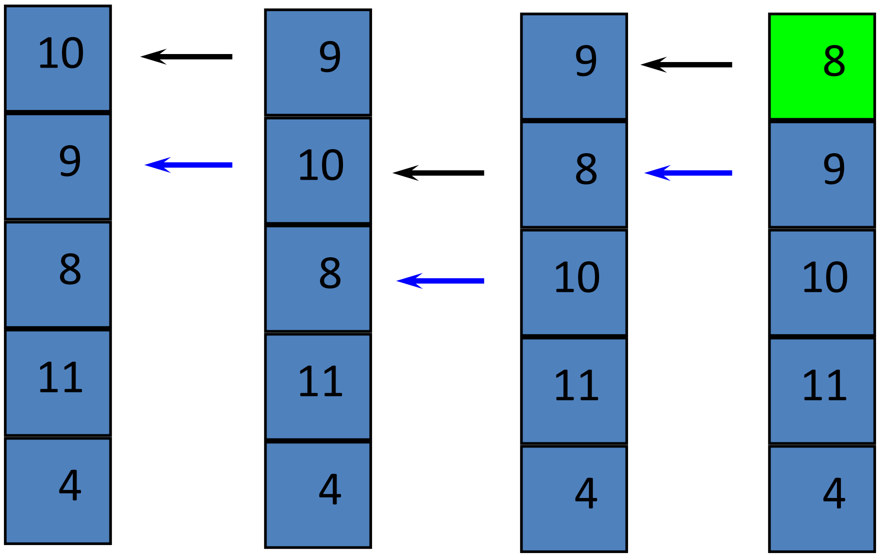
Bubble Sort



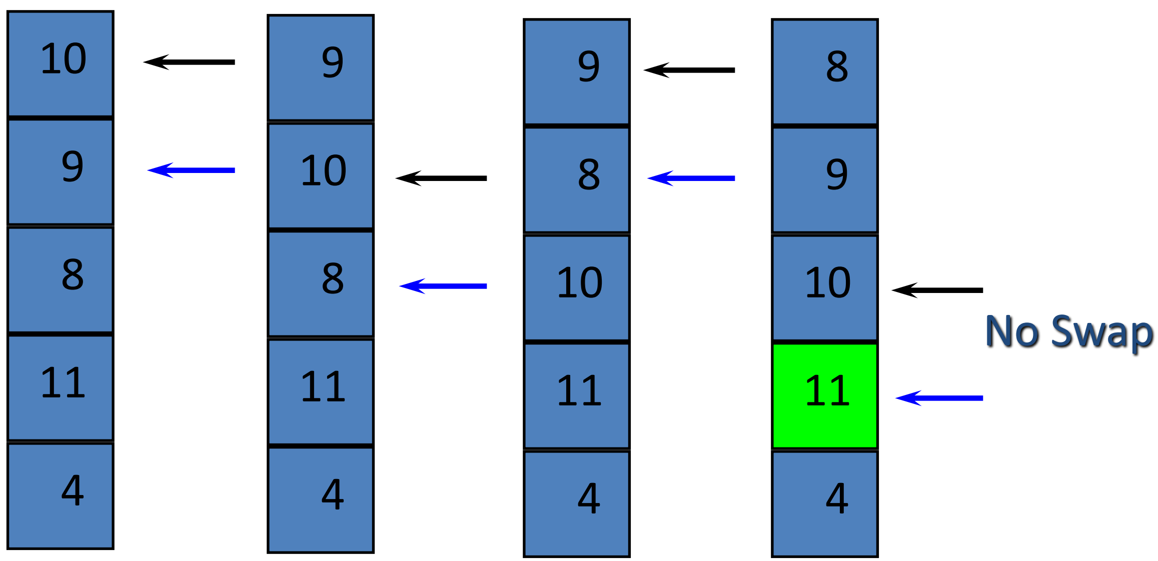
Bubble Sort



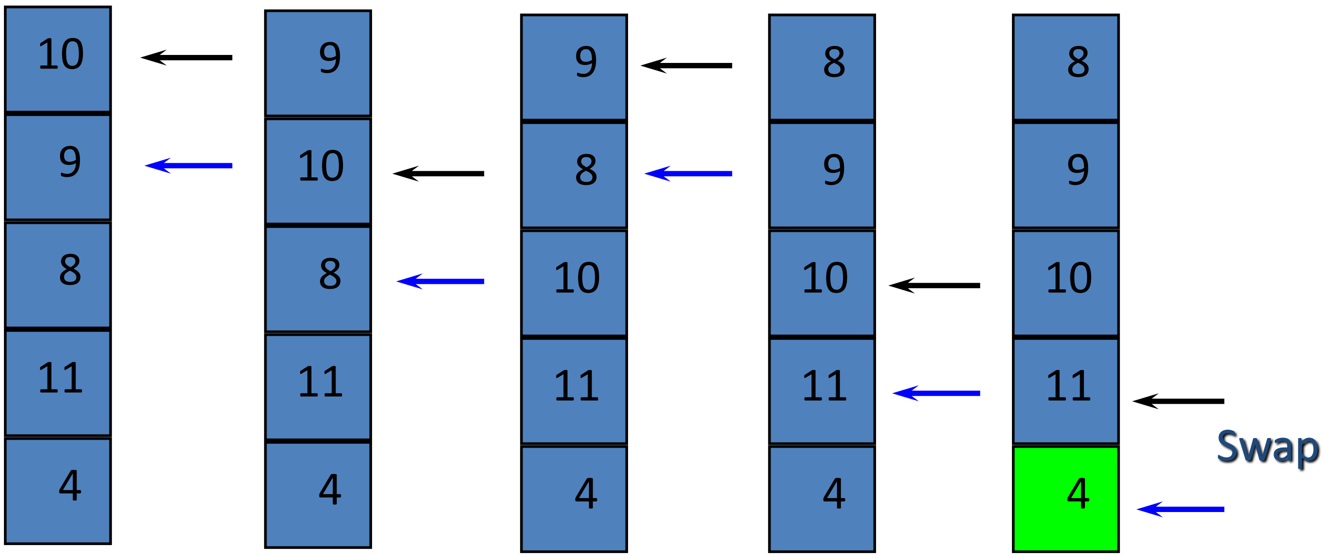
Bubble Sort



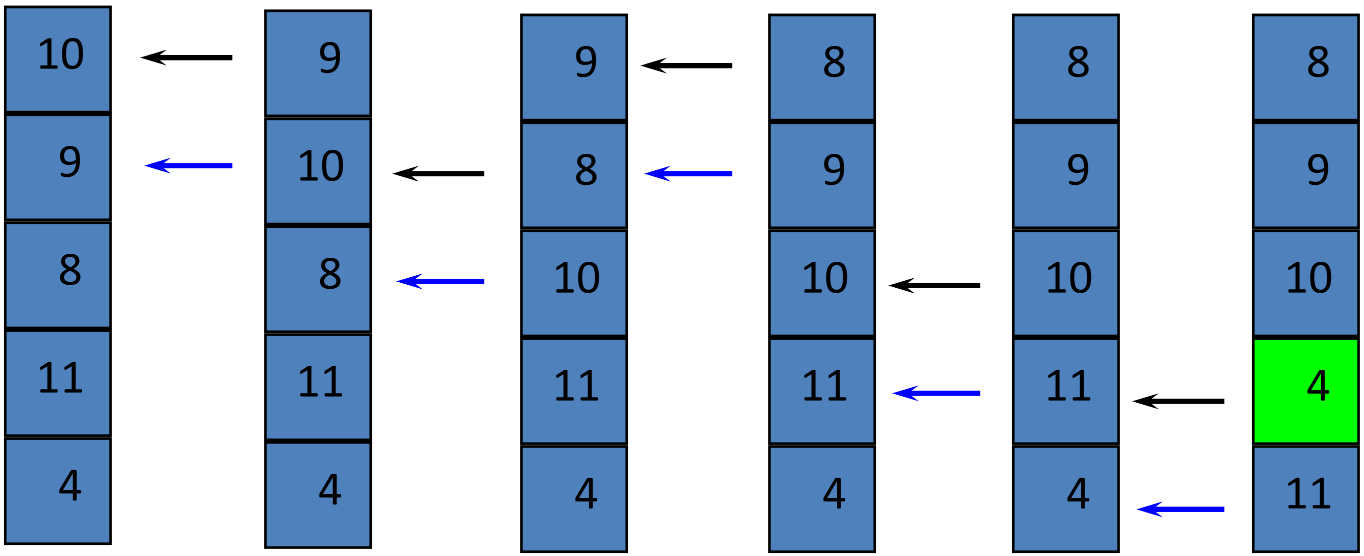
Bubble Sort



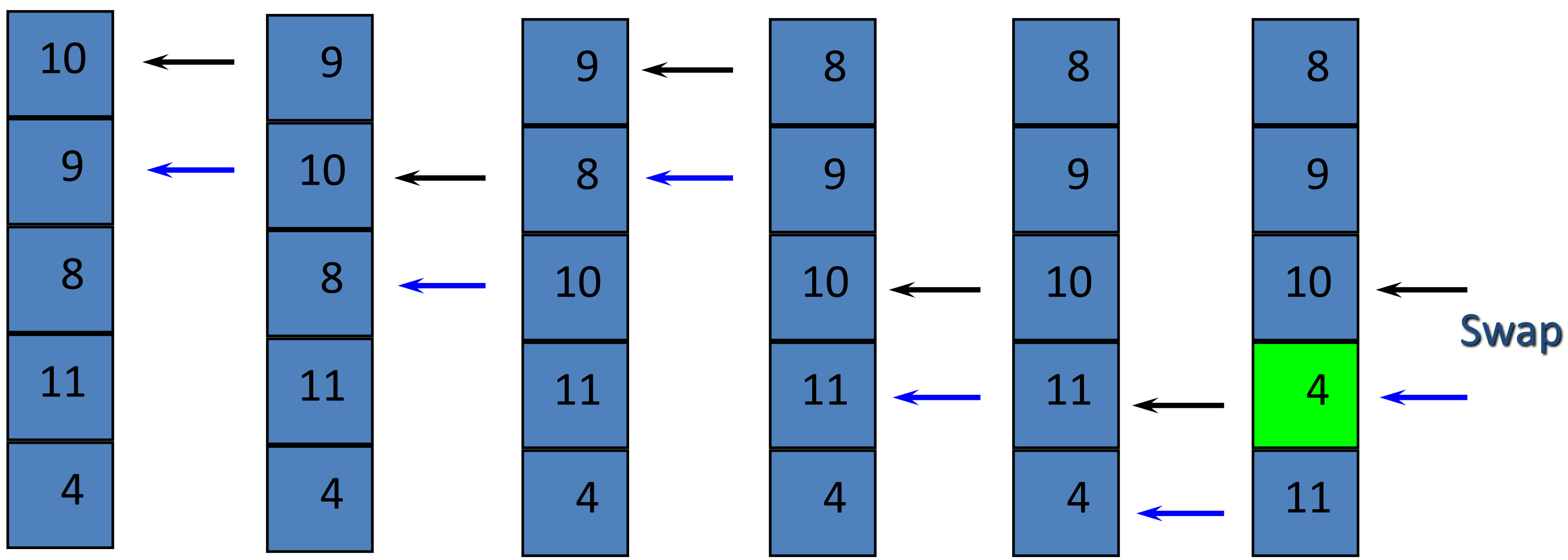
Bubble Sort



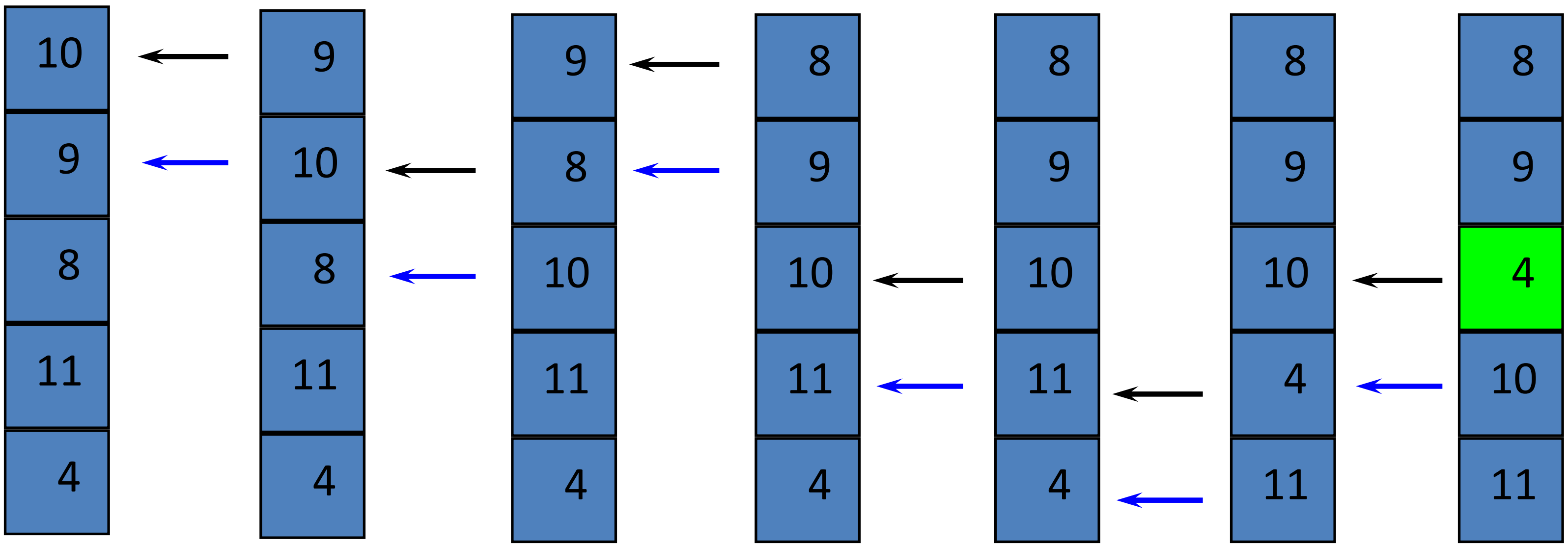
Bubble Sort



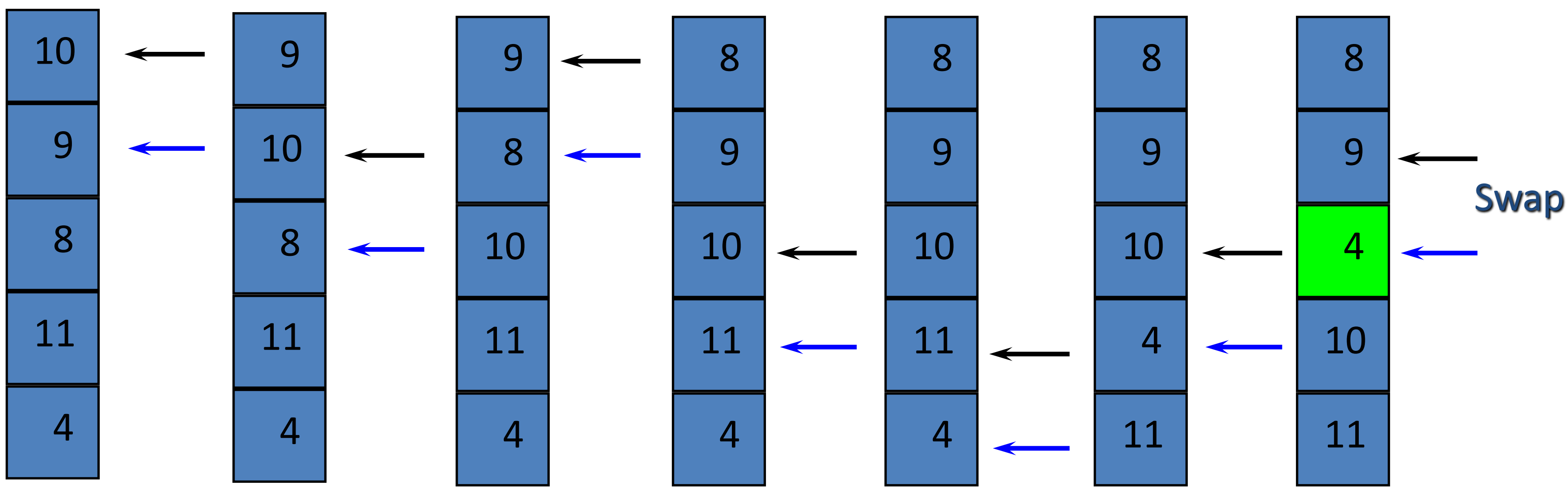
Bubble Sort



Bubble Sort

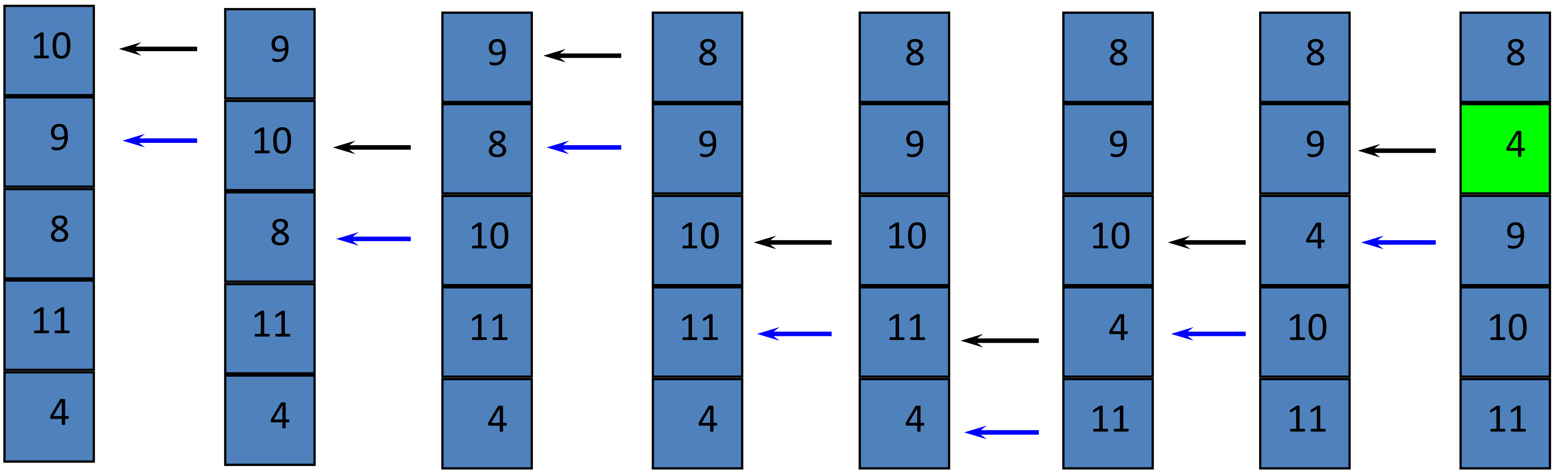


Bubble Sort



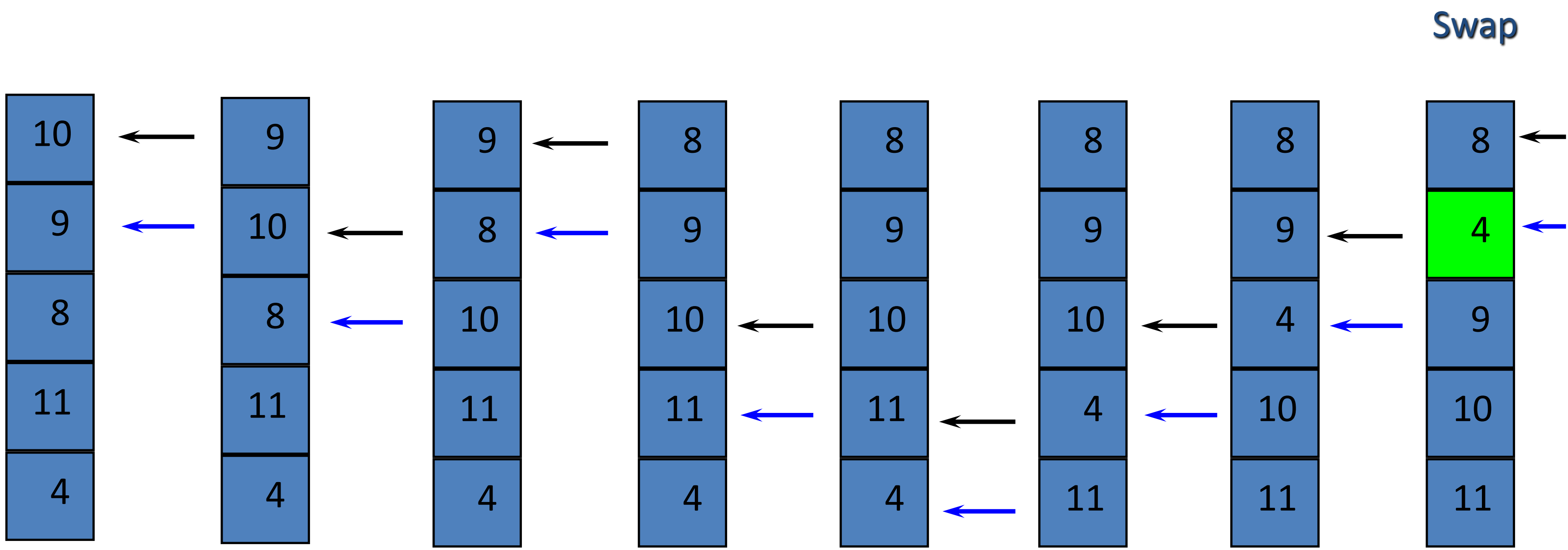
10
9
8
11
4
9
10
8
11
4
9
8
10
11
4
8
9
10
11
4
8
9
10
11
4
8
9
10
4
11
8
9
4
10
11
Swap

Bubble Sort



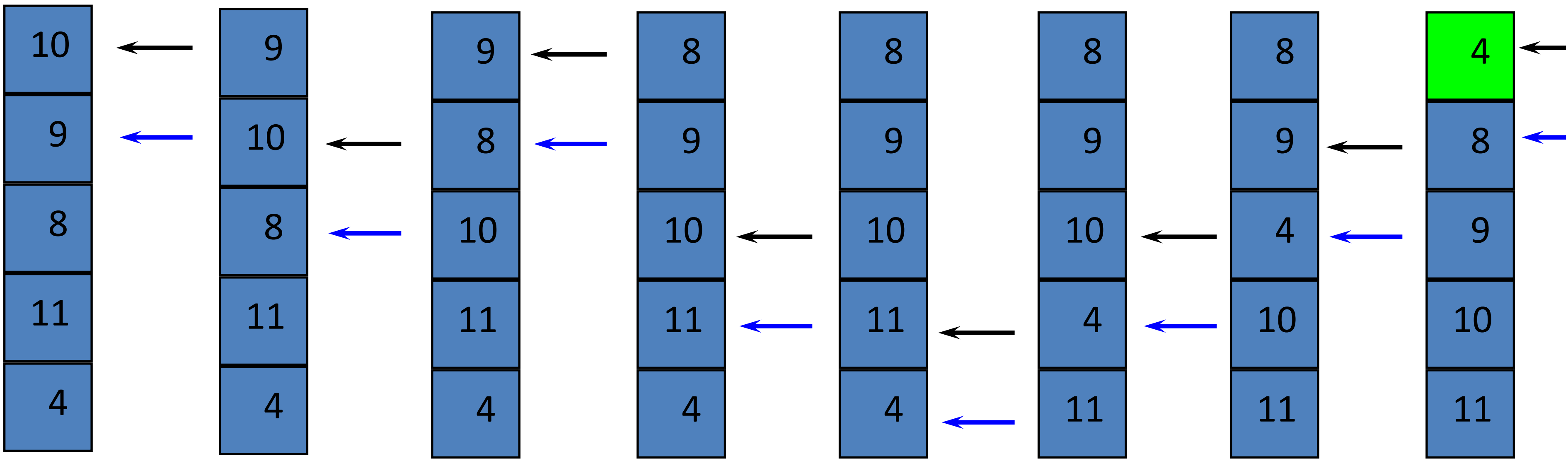
10
9
8
11
4
9
10
8
11
4
9
8
10
11
4
8
9
10
11
4
8
9
10
11
4
8
9
10
4
11
8
9
4
10
11
8
4
9
10
11

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++) { // why?
        for (j=i; j >= 0; j--) {  // Because initially j=i
            if (a[j] > a[j+1]) {  // and we access element j+1

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

Note that this is an inefficient naive implementation.
It doesn't use the while condition in the pseudo-code:

```
WHILE list element > previous list element
```

It uses a for loop and blindly compares all elements right back to the beginning of the list, swapping when necessary.

Exercise: reimplement this more efficiently with the while loop.

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)

- We can make the preceding algorithm more efficient. How? (**hint**: do we always have to bubble back to the top?)

Bubble Sort

Exercise: implement these changes and write a driver program to test:

- the original bubble sort
- the more efficient bubble sort
- 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  item_type;

void selection_sort(item_type a[] , int n) {

 item_type 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;
   }
}
```

Insertion sort

- Step 1 − If it is the first element, it is already sorted. return 1;
- Step 2 − Pick next element
- Step 3 − Compare with all elements in the sorted sub-list
- Step 4 − Shift all the elements in the sorted sub-list that is greater than the value to be sorted
- Step 5 − Insert the value
- Step 6 − Repeat until list is sorted

Insertion Sort

```
typedef  int  item_type;

insertion_sort(item_type a[], int n) {

   int i,j;
   int temp;

   for (i=1; i<n; i++) {
      j=i;
      while ((j>0) && (a[j] < a[j-1])) {
         temp = a[j-1]; // swap
         a[j-1] = a[j];
         a[j] = temp;

         j = j-1;
      }
   }
}
```

Agenda

Searching and Sorting Algorithms

- Linear Search & Binary Search
- In-place sorts
  - Bubble Sort
  - Selection Sort
  - Insertion Sort
- Not-in-place sort
  - Quicksort
  - Mergesort
- Characteristics of a good sort

Sorting Algorithms

- Not-in-place sort
  - Additional space requirements not $O(1)$
  - **Tradeoff**: less computationally-complex algorithms but greater memory requirements (possibly unpredictable)
    - Quick Sort
    - Merge Sort

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 (called a "pivot")
- and create two (sub)lists
    1. Left sublist: all elements are less than partitioning element,
    2. Right sublist: all elements are greater than it
- Now repeat this partitioning effort on each of these two sublists
- This is a divide-and-conquer strategy

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 by
  - scanning left to right
  - scanning 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

Quicksort

- Options for selecting the pivot:
  - Random
  - select the first element
  - select the last element
  - select the middle element
  - find the median of the first, last and middle.

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()

```
void quicksort(item_type s[ ], int size, int low, int high){

   if ((high-low)>=1){
    int pivot = (low + high) / 2;

    int middle = partition(s, low,high,s[pivot]);

    quicksort(s,size, low, middle);
    quicksort(s,size, middle+1, high);
   }
}
```

Choose the middle item as the pivot.

Implementation of Quicksort()

```
void quicksort(item_type s[ ], int size, int low, int high){

   if ((high-low)>=1){
    int pivot = (low + high) / 2;

    int middle = partition(s, low,high,s[pivot]);

    quicksort(s,size, low, middle);
    quicksort(s,size, middle+1, high);
   }
}
```

Partition and find the new middle.

Implementation of Quicksort()

```
void quicksort(item_type s[ ], int size, int low, int high){

   if ((high-low)>=1){
    int pivot = (low + high) / 2;

    int middle = partition(s, low,high,s[pivot]);

    quicksort(s,size, low, middle);
    quicksort(s,size, middle+1, high);
   }
}
```

Recursively sort both sub-lists.

Implementation of Quicksort()

```
int partition(item_type s[], int low, int high, int pivot){
   int temp;
   int i = low-1;
   int j = high+1;
   while(true){
         do{
        i++;
    }while(s[i] < pivot);
    do{
        j--;
    }while(s[j] > pivot);
    if (i < j){
        temp = s[i];
        s[i] = s[j];
        s[j] = temp;
    }else{
        return j;        }
   }
}
```

Step along until you find an item on the left that is larger tha the pivot.

Implementation of Quicksort()

```
int partition(item_type s[], int low, int high, int pivot){
    int temp;
    int i = low-1;
    int j = high+1;
    while(true){
           do{
          i++;
     }while(s[i] < pivot);
     do{
          j--;
     }while(s[j] > pivot);
     if (i < j){
          temp = s[i];
          s[i] = s[j];
          s[j] = temp;
     }else{
          return j;
     }
    }
}
```

On the right side look for elements less than the pivot.

Implementation of Quicksort()

```
int partition(item_type s[], int low, int high, int pivot){
   int temp;
   int i = low-1;
   int j = high+1;
   while(true){
         do{
        i++;
    }while(s[i] < pivot);
    do{
        j--;
    }while(s[j] > pivot);
    if (i < j){
        temp = s[i];
        s[i] = s[j];
        s[j] = temp;
    }else{
        return j;
    }
   }
}
```

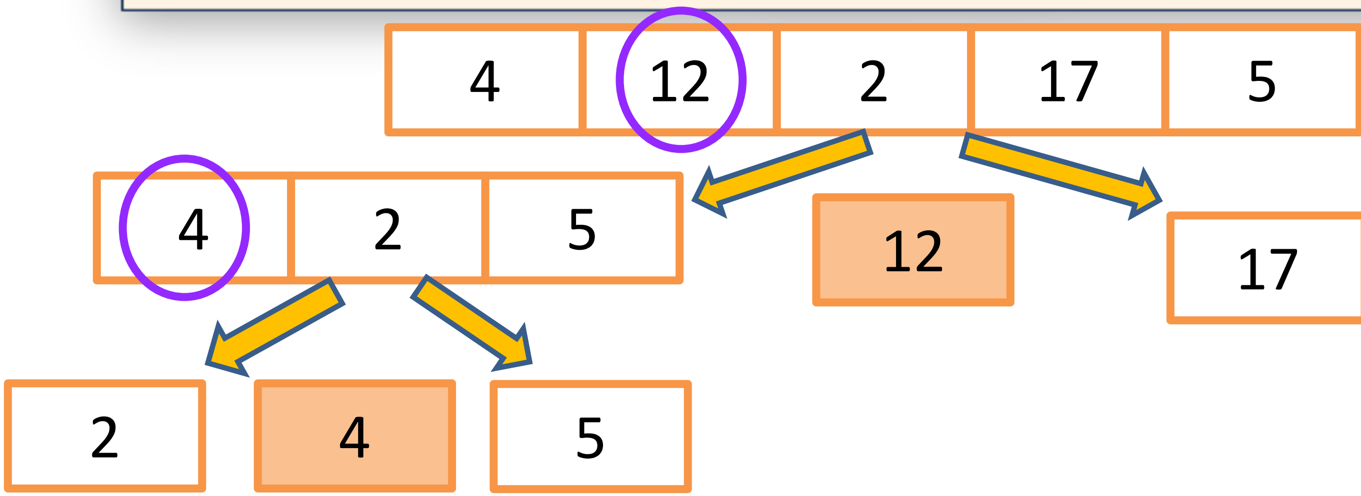
Exchange them.

Implementation of Quicksort()

```
int partition(item_type s[], int low, int high, int pivot){
   int temp;
   int i = low-1;
   int j = high+1;
   while(true){
         do{
         i++;
     }while(s[i] < pivot);
     do{
         j--;
     }while(s[j] > pivot);
     if (i < j){
         temp = s[i];
         s[i] = s[j];
         s[j] = temp;
     }else{
         return j;
     }
   }
}
```

End the loop and return the new middle.

Quick Sort



Here the pivot has been selected at random.

Partitioning

5	7	13	9	11	16	2	4	14

i

Moving the pointers.

j

5	7	13	9	11	16	2	4	14

i

13 is larger than 11 so stop.
4 is smaller than 11 so stop.

j

Partitioning

5	7	4	9	11	16	2	13	14

Swap them.

i

j

5	7	4	9	11	16	2	13	14

11 is equal to 11 so stop.
2 is smaller than 11 so stop.

i

j

Partitioning

5	7	4	9	2	16	11	13	14

Swap them.

i j

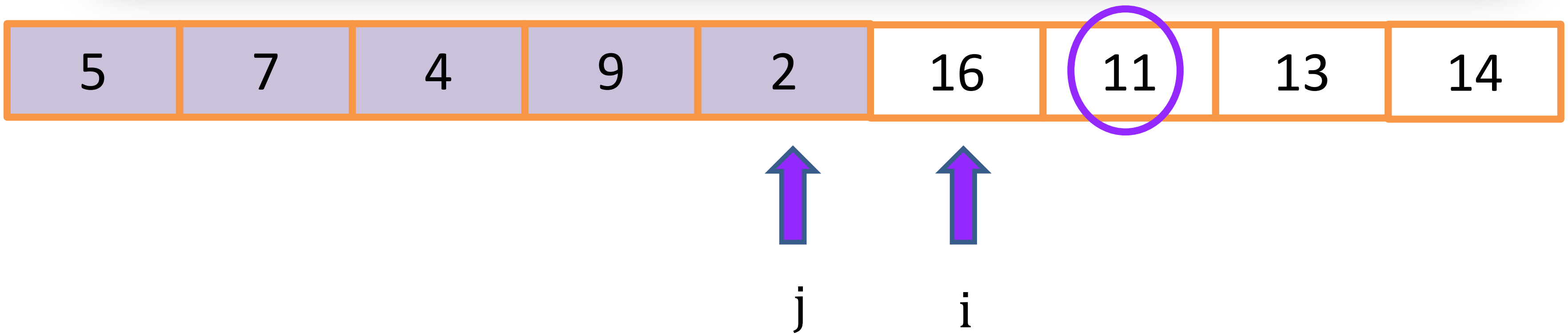
5	7	4	9	2	16	11	13	14

16 is larger than 11 so stop.
2 is smaller than 11 so stop.

j i

i and j crossed over so return j.

Partitioning



The shaded side is all less than or equal to the pivot and the other side is all greater than or equal to the pivot.

Now run quicksort again on each half, selecting new pivots.

Quicksort

- Performance depends on which element is selected as the pivot

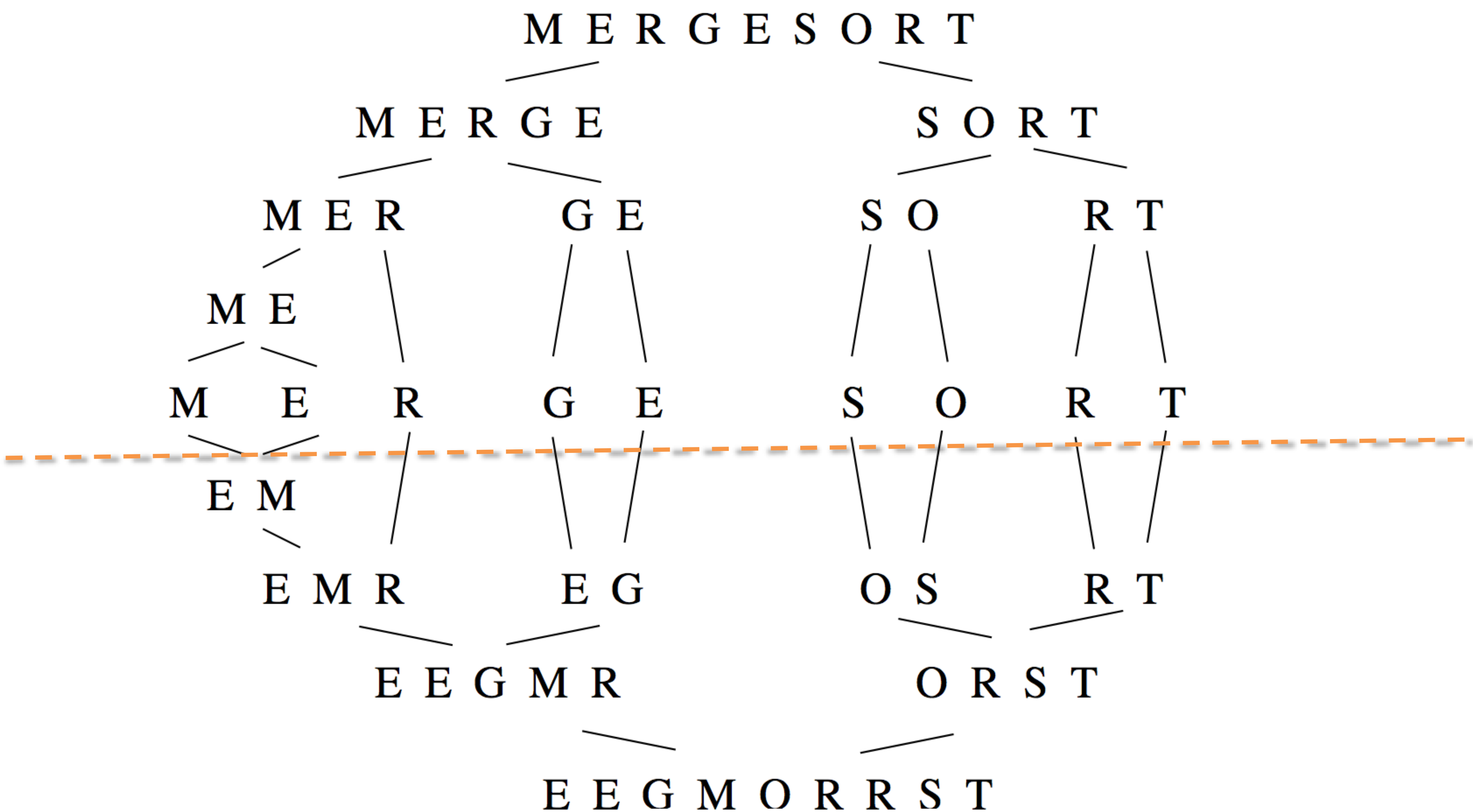
- The worst-case occurs when the list is sorted and the left-most element is selected as the pivot

- Space complexity is $O(n^2)$ in the worst case

Mergesort

- Divide-and-conquer, recursive, $O(n \log n)$
- Recursively partition the list into two lists L1 and L2
  - L1 and L2 approx. n/2 elements each
- Stop when we have a collection of lists of 1 element
- Now, each L1 and L2 is merged into a list S
  - the elements of L1 and L2 are put in S in order
- Pairs of sorted lists S1 and S2 are, in turn, merged as we ascend back up through the recursion

Mergesort


M E R G E S O R T
M E R G E
S O R T
M E R
G E
S O
R T
M E
M
E
R
G
E
S
O
R
T
E M
E M R
E G
O S
R T
E E G M R
O R S T
E E G M O R R S T

Merge Sort

```
mergesort(item_type s[ ], int size, int low, int high){
    if(low<high)
    {
        int middle = (low + high) / 2;
        //int middle = low+(high-low) / 2;//alternate
        mergesort(s, size,low, middle);//recurse
        mergesort(s, size,middle+1, high);//recurse
        merge(s,low, middle, high);
    }
}
```

Merge Sort

- The efficiency of mergesort depends on how we combine the two sorted halves into a single sorted list

- The key is to realize that each half (i.e. each sublist) is sorted

- So we just have to repeatedly do the following

  – Take the ”front” element of either one list or the other (depending on which is smaller) and

  – Move it to the merged list (thus keeping the elements in order)

Merge Sort

```
merge(item_type s[], int low, int middle, int high){
   int size=high-low+1;
   int temp[size];
   int i = low;
   int j = middle;
   int k = 0;
   while(i < middle && j <= high){
       if (s[i] <= s[j]){
           temp[k] = s[i];
           i++;
       }else{
           temp[k] = s[j];
           j++;
       }
       k++;
   }
```

Merge Sort

```
    //from previous slide
    // copy any remaining items on the left
    while (i < middle){
        temp[k] = s[i];
        k++;
        i++;
    }
    //copy any remaining items on the right
    while (j <= high){
        temp[k] = s[j];
        k++;
        j++;
    }
    for(int x = 0; x < size; x++){
        s[low+x] = temp[x];
    }

} // end of merge function.
```

Mergesort

Why is mergesort $O(n \log n)$?

How many times do we merge and how big are the data sets?

Let's assume that n is a power of two

At level 0

2^1 calls to `mergesort` & merge 2^1 lists of size $\sim n/2$

At level 1

2^2 calls to `mergesort` & merge 2^2 lists of size $\sim n/4$

…

At level k

2^{k+1} calls to `mergesort` & merge 2^{k+1} lists of size $\sim n/ 2^{k+1}$

Mergesort

How many levels k?

$k = \log_2 n,\ e.g.$ if $n = 8$, $k = 3$

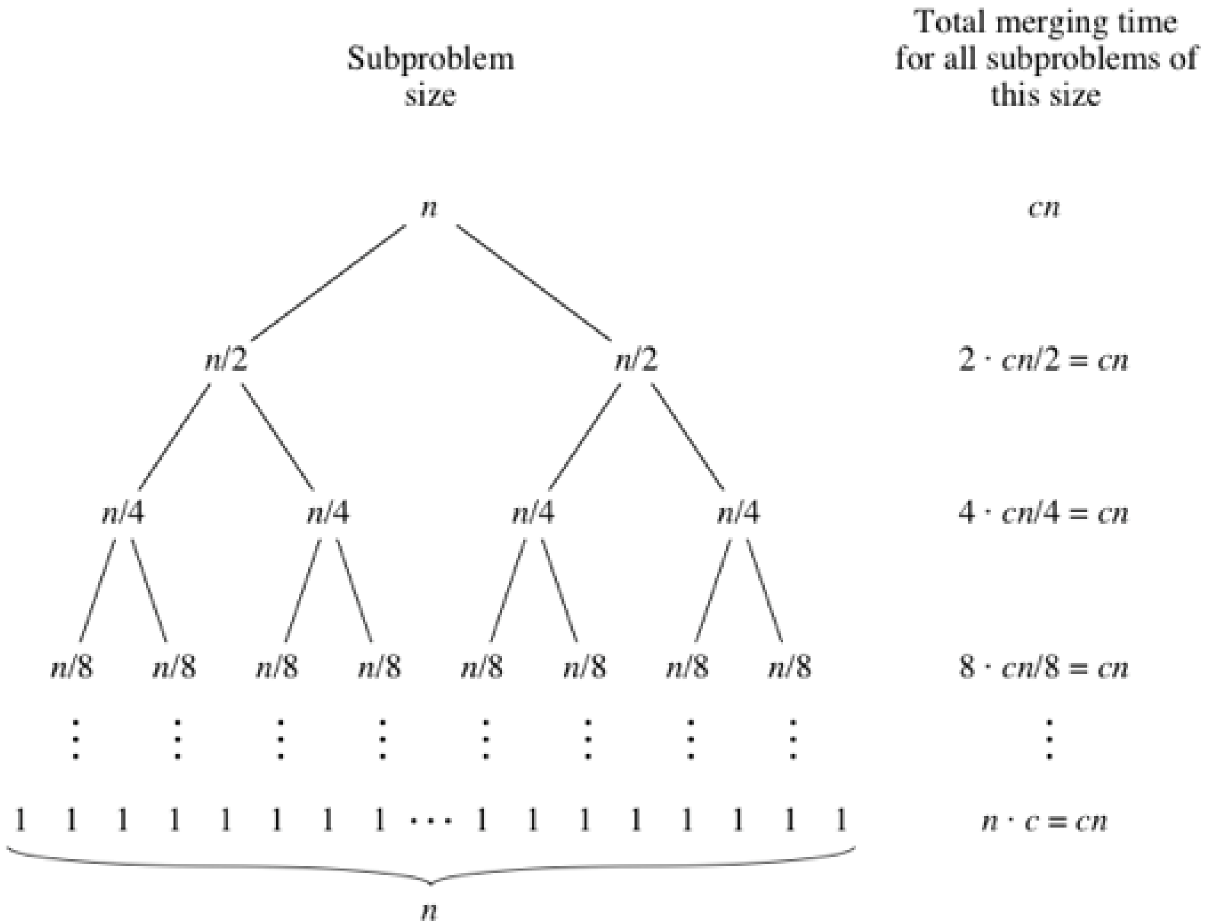
At level k, the sub-lists are of size 1.

So we merge on $k = \log_2 n$ levels (level $0 - k$-1)

Each level we merge 2^{k+1} lists of size $\sim$n/ 2^{k+1} i.e. total size $\sim$n

So the total complexity is $O(n \log_2 n)$

Mergesort



https://www.khanacademy.org/computing/computer-science/algorithms/merge-sort/a/analysis-of-merge-sort

Agenda

Searching and Sorting Algorithms

- Linear Search & Binary Search

- In-place sorts
  - Bubble Sort
  - Selection Sort
  - Insertion Sort

- Not-in-place sort
  - Quicksort
  - Mergesort

- Characteristics of a good sort

Characteristics of a Good Sort

Code Complexity

- Short, simple algorithms are appealing because they are easy to implement and debug
- Algorithms that can easily be applied to all data types are convenient to use but often come at the cost of implementation complexity
- Although they may not be as fast as more specialized algorithms, simple algorithms are always appealing especially when maintenance is an issue

Characteristics of a Good Sort

Stability

- A stable sorting algorithm maintains the relative order of records with equal keys
  - Let records R and S have the same key
  - R appears before S in the original list,
  - R will always appear before S in the sorted list
- This is particularly important when sorting based on multiple keys

Characteristics of a Good Sort

Stability

- Assume that the following pairs of numbers are to be sorted by their first component (two different results are possible)

(4, 2) (3, 7) (3, 1) (5, 6)

(3, 7) (3, 1) (4, 2) (5, 6) (stable: order maintained)
(3, 1) (3, 7) (4, 2) (5, 6) (unstable: order changed)

- Unstable sorting algorithms change the relative order of records with equal keys, but stable sorting algorithms do not

Characteristics of a Good Sort

Stability

- Unstable sorting algorithms can be specially implemented to be stable

- Stability usually comes with an additional computational cost

Acknowledgement

- Adopted and Adapted from Material by:
- David Vernon: vernon@cmu.edu ; www.vernon.eu