CS111
Introduction to Computer Science

- Sorting
  - Insertion sort
  - Selection sort
  - Mergesort

Sorting

- If we keep our data sorted in an array we can use Binary search O(log n) instead of Linear search O(n) to find an item in the array
- How to keep the data sorted?
  - We'll learn two algorithms to sort arrays
    - Insertion sort
    - Selection sort

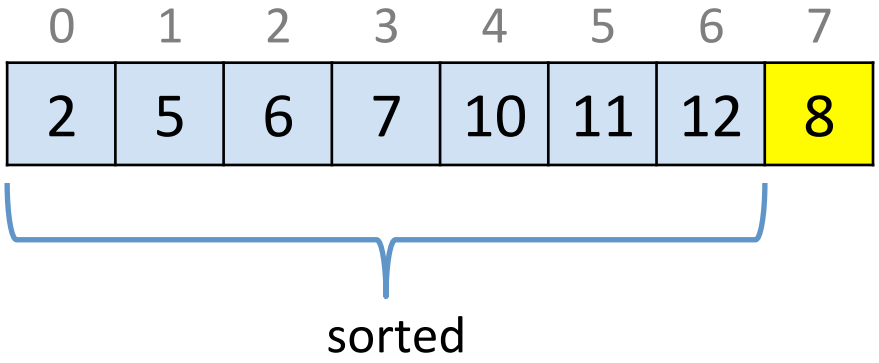
Insertion Sort

One way of thinking of inserting sort:

- Imagine you are playing a card game.
- The cards you are holding on your hand are sorted.
- The dealer hands you one card.
- You have to put it into the correct place so that the cards you're holding are sorted.

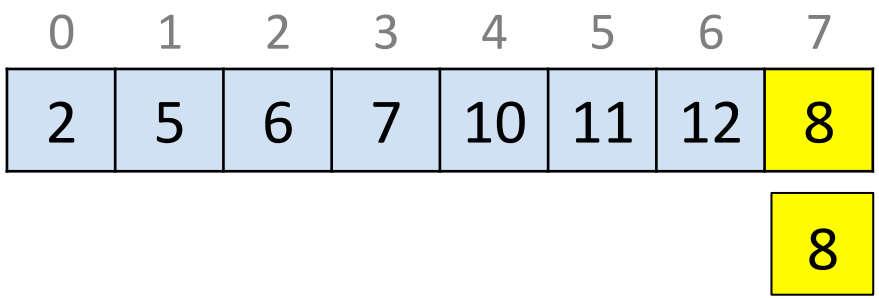
Insertion Sort: Insert into sorted array

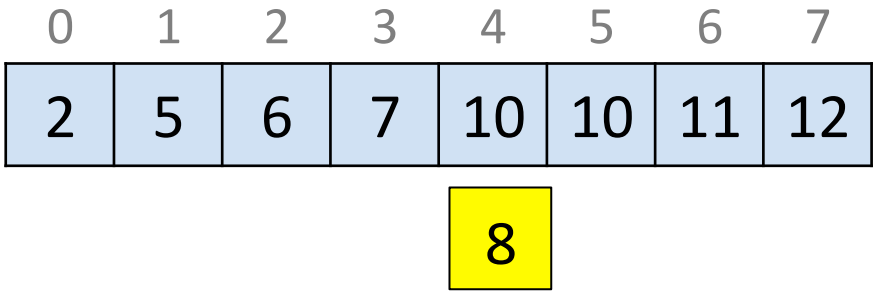
- If we keep the cards in an array
  - sub-array from index 0 through index 6 is sorted
  - insert the element currently at index 7

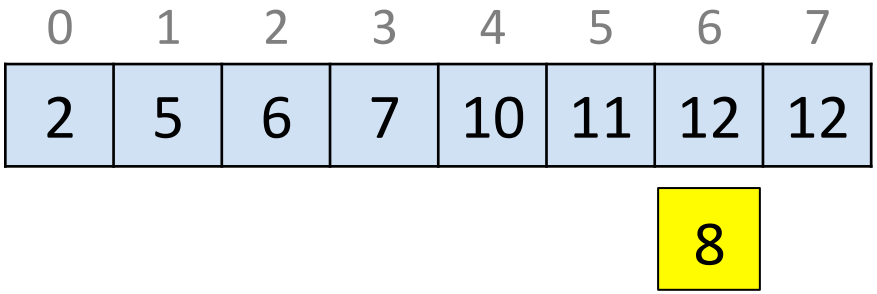


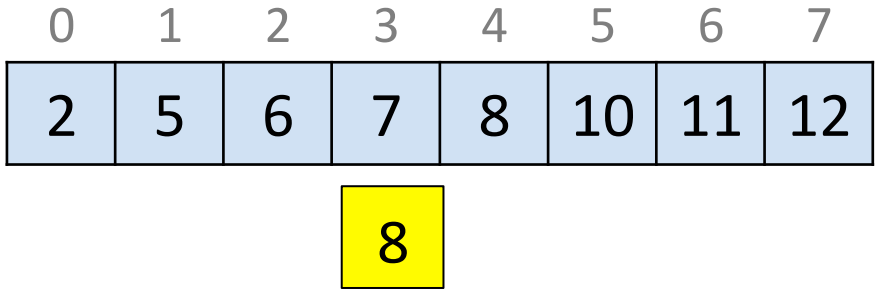
  - To move 8 into index 3, shift elements 10, 11, and 12 right by one position

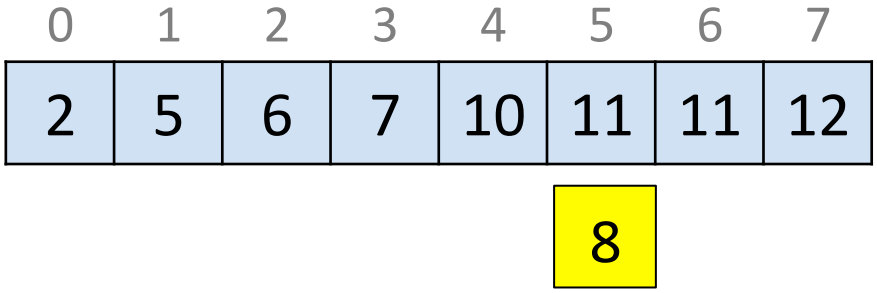
Insertion Sort: Insert into sorted array









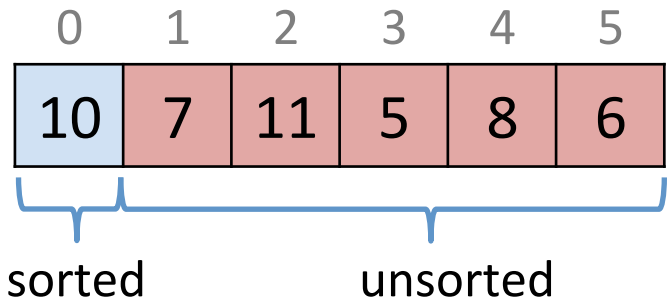


Compare 8 with each item from right to left until a smaller item is found

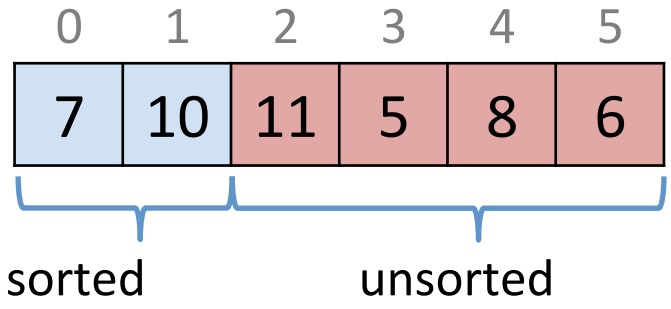
Insertion Sort: sort entire array

- Now you are given an unsorted array

  mark two regions sorted (only the first item) and unsorted (rest of the array)

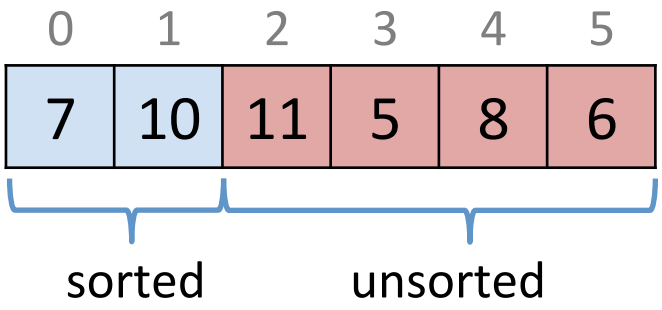


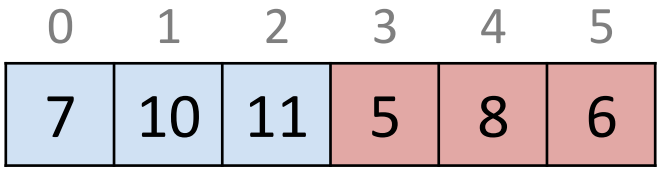
  take the first element from the unsorted region and insert into the sorted region

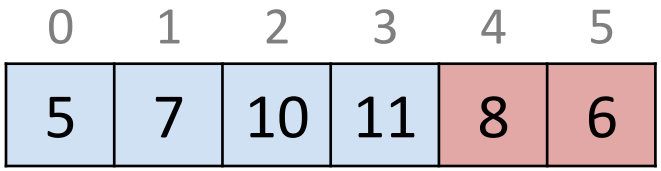


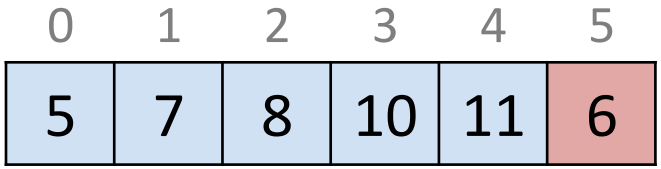
Insertion Sort: sort entire array

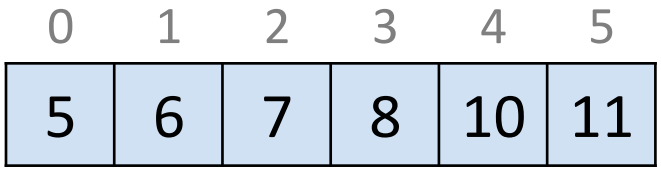
- Do the same for the rest of the unsorted region
  - take the first element from the unsorted region and insert into the sorted region











Insertion Sort: Efficiency Analysis

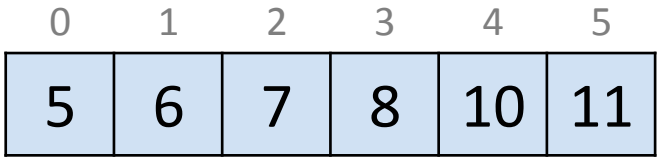
```
void insertionSort(int[] a, int n) {

    for (int i = 1; i < n; i++) {
        int itemToInsert = a[i]; // start of unsorted region
        int loc = i – 1; //end of sorted region

        while (loc >= 0) {
            if (a[loc] > itemToInsert) {
                a[loc+1] = a[loc];
                loc--;
            } else {
                break;
        }
        a[loc+1] = itemToInsert;
    }
}
```

Done *loc+1* times for each number

Basic Operation

Insertion Sort: Efficiency Analysis

Best case: array is sorted in increasing order



– count the number of comparisons

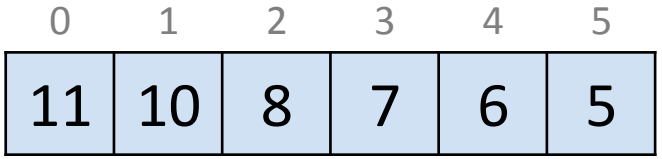
1st insertion	0 compares
2nd insertion	1 compares
3rd insertion	1 compares
nth insertion	1 compares

– f(n) = 0+1+1+...+1

– f(n) = n-1 ➡ O(n)

Insertion Sort: Efficiency Analysis

Worst case: array is sorted in decreasing order



– count the number of comparisons

1st insertion	0 compares
2nd insertion	1 compares
3rd insertion	2 compares
nth insertion	n-1 compares

$$f(n) = 0 + 1 + 2 + 3 + \ldots + n - 1 = \sum_{i=1}^{n-1} i = (n-1)\frac{((n-1)+1)}{2} = O(n^2)$$

Arithmetic Series

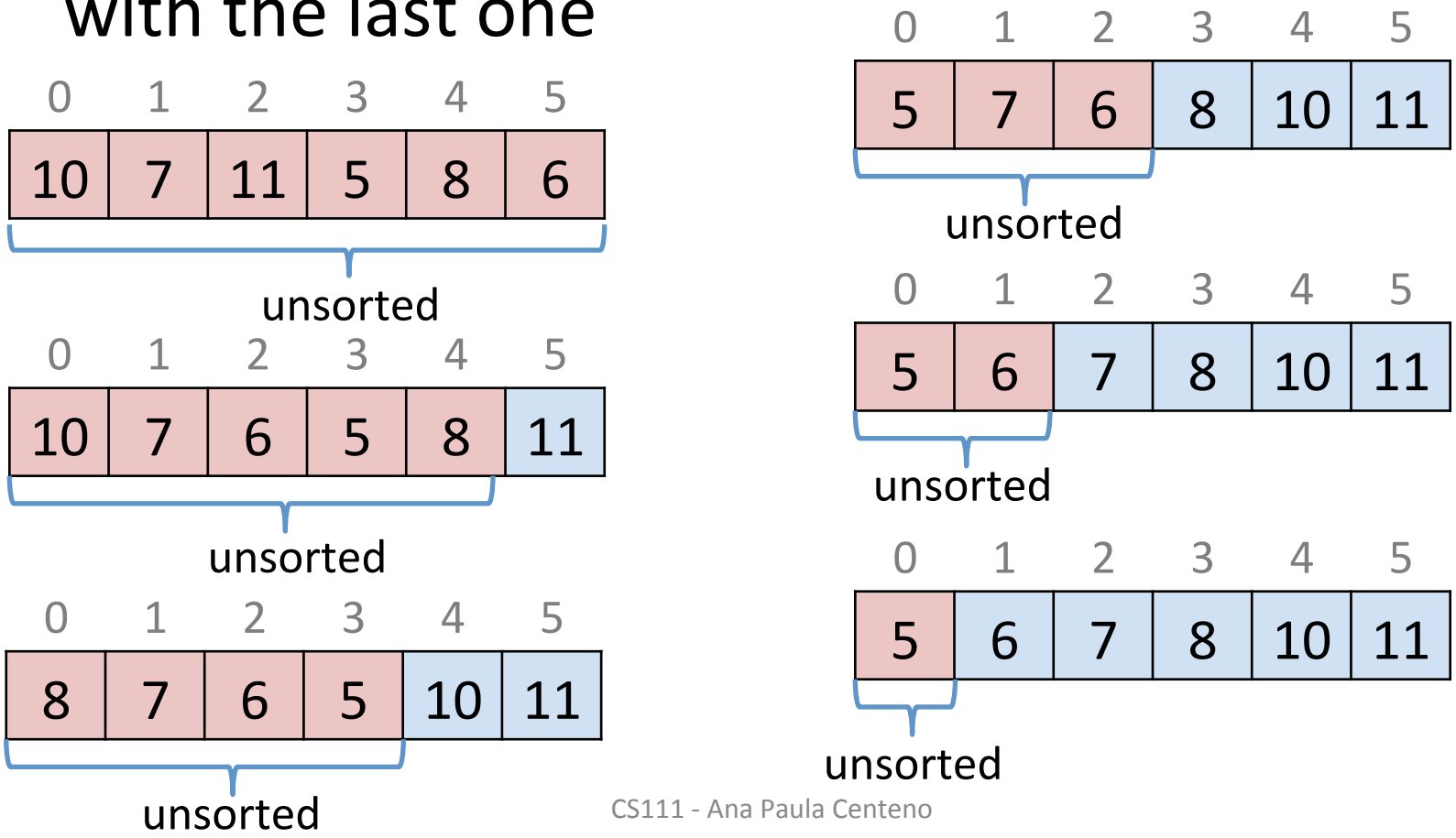
Selection Sort

- The second sorting algorithm we'll study

- The idea is to repeatedly find the biggest item in the array and move it to the end
  - to keep the array sorted in increasing order

Selection Sort

- Find the biggest item in the array and swap it with the last one



Selection Sort: Efficiency Analysis

```
void selectionSort(int[] a, int n) {
    for (int i = n-1; i > 0; i--) {

        int maxLoc = 0; //Location of the largest value

        for (int j = 1; j <= i; j++) {
            if (a[j] > a[maxLoc]) {
                maxLoc = j;
            }
        }
        int temp = a[maxLoc];
        a[maxLoc] = a[i];
        a[i] = temp;
    }
}
```

Keeps track of the unsorted region
i is the first unsorted item

Keeps track of the sorted region.
Done *i-1* times for each number

Basic Operation

Selection Sort: Efficiency Analysis

Worst case and Best case are the same

– count number of comparisons to find largest value

1st largest	n-1 compares
2nd largest	n-2 compares
3rd largest	n-3 compares
nth largest	0 compares

$$f(n) = 0 + 1 + 2 + 3 + \ldots + n - 1 = \sum_{i=1}^{n-1} i = (n-1)\frac{((n-1)+1)}{2} = O(n^2)$$

Arithmetic Series

Sorting Algorithms

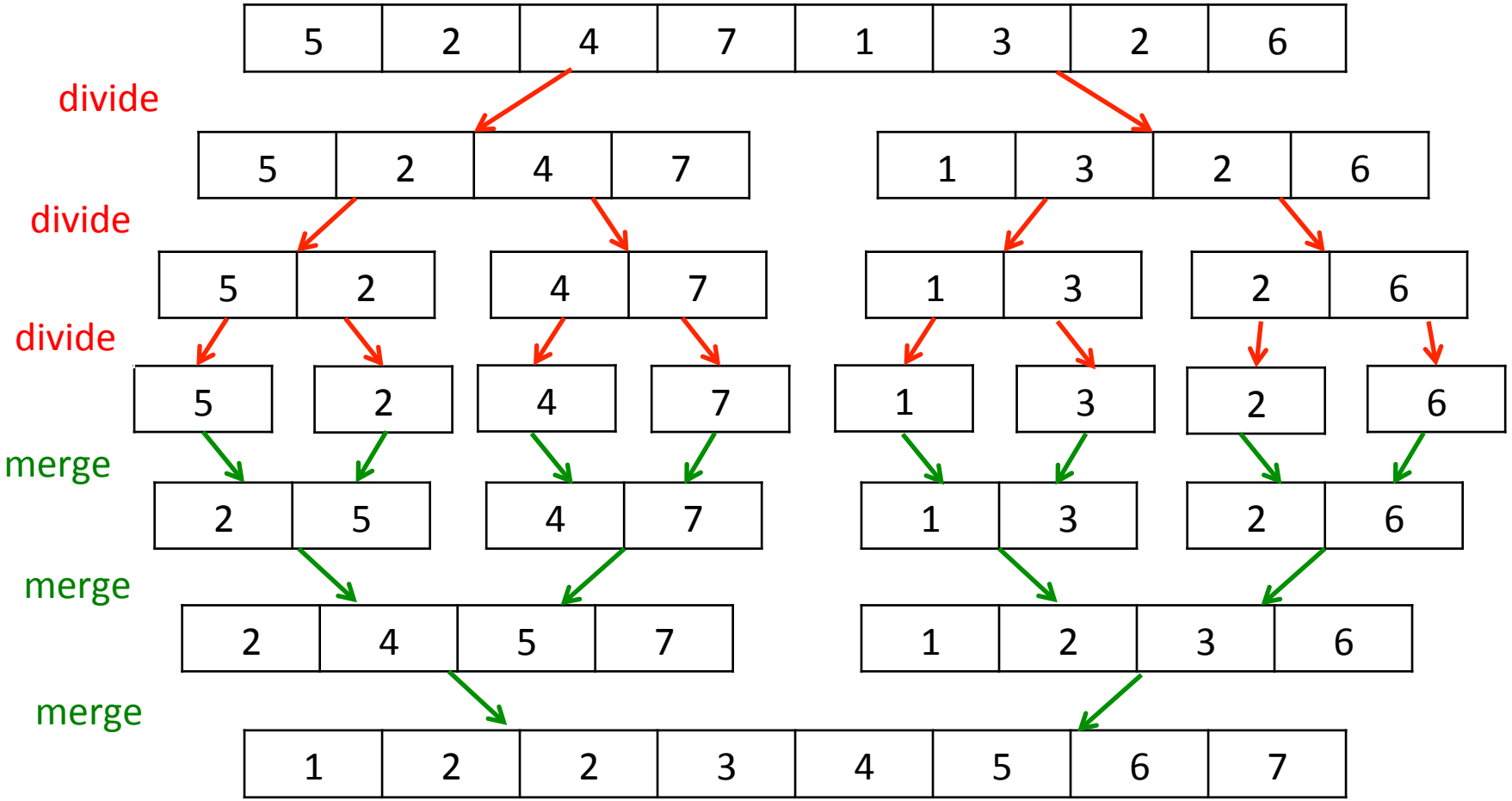
- Insertion sort $O(n^2)$
  - best if data is partially sorted
- Selection sort $O(n^2)$
  - best and worst case are the same

Mergesort

Follows the divide-and-conquer approach

1. break the problem into several sub-problems that are similar to the original problem but smaller in size
2. solve the sub-problems recursively
3. combine the sub-problems solution to create a solution to the original problem

Mergesort


5 2 4 7 1 3 2 6
divide
5 2 4 7
1 3 2 6
divide
5 2
4 7
1 3
2 6
divide
5
2
4
7
1
3
2
6
merge
2 5
4 7
1 3
2 6
merge
2 4 5 7
1 2 3 6
merge
1 2 2 3 4 5 6 7

Mergesort

Main idea for an array of n elements:

- Divide the unsorted array until there are n arrays, each containing 1 element.
  - An array with one element is considered sorted.
- Repeatedly merge two sorted arrays until there is only 1 array remaining.
  - This will be the sorted array at the end.

Mergesort

```
void mergesort (int[] a, int l, int r){
   if (l > r) return;
   int middle = (l+r)/2;
   mergesort(a, l, m);
   mergesort(a, m+1, r);
   merge(a, l, m, r);
}
```

```
void merge (int[] a, int l, int m, int r){
   int[] b = new int[m-l+1]; //copy half array
   for (int i=0; i<=m; i++) b[i] = a[i];
   int i = 0, j = m+1, k = 0;
   while (i <= m && j <= r) {
      if (a[j] < b[i]) {a[k++] = a[j++];}
      else {a[k++] = b[i++];}
   }
   while (i <= m) {a[k++] = b[i++];}
   while (j <= r) {a[k++] = a[j++];}
}
```

Mergesort: Analysis

Merge: merges two sorted halves of the array

Worst case: last two elements of the output are on different halves

- making a copy of half of the array is O(n)
  - constant time for each assignment: done n/2 times
- comparisons O(n)
  - each comparison takes constant time
  - in the worst case there are n-1 comparisons
- O(n) + O (n) = O (n)

Mergesort: Analysis

Mergesort

- the comparison and the computation of middle is done in constant time O(1)
- two calls to mergesort: each on half of the array
- one call to merge O(n)

Recurrence relation is

$T(n) = T(n/2) + T(n/2) + O(n) + O(1)$

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

Mergesort: Analysis

- Repeat the recurrence

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

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

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

  $\ldots = 2^k\,T(n/2^k) + kn$

- Let $n = 2^k$

  $= 2^k T(2^k/2^k) + kn$

  $= n\,T(1) + kn$

  $= n*1 + kn$

  if $n = 2^K$ then $k = \log n$ and $T(n) = n + \log n * n = O(n \log n)$