Data Structures and Algorithms

XMUT-COMP 103 - 2026 T1

Sort Algorithms

Agatha Rachmat

School of Engineering and Computer Science

Victoria University of Wellington

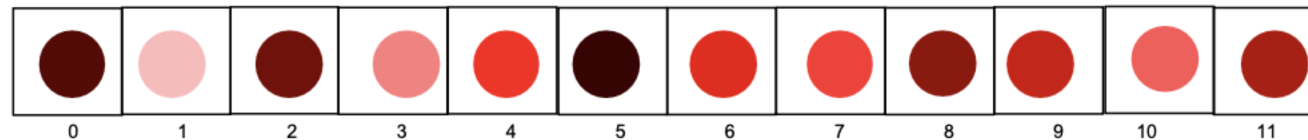
© Peter Andreae and Mohammad Nekooei

Topics

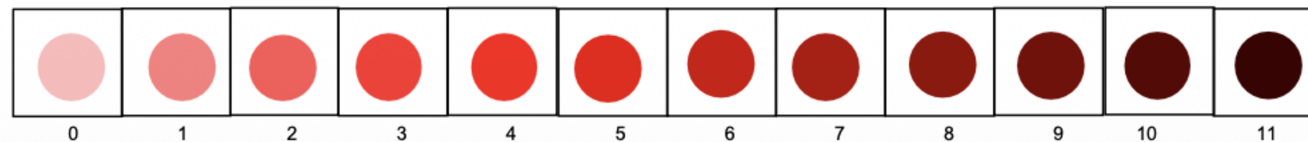
- Intro to sorting.
- Ways to rate sorting algorithms.
- Ways of sorting.
  - Selection-based sorts.
  - Insertion-based sorts.
  - Compare and swap sorts.
- Other sorts.
- Examples of sorting algorithm:
  - Selection sorts.
  - Quick sorts.
  - Merge sorts.

Intro to Sorting

- Use to arrange elements of an array/list in a specific order.
- Operator or criterion is used to decide the order i.e. value, index, etc.
- Sorting is useful for:
  - Determining the smallest and largest values.
  - Searching is easy when in order.
  - Solving problems.



Sorting



Ways to rate sorting algorithms

- Efficiency
  - What is the (worst-case) order of the algorithm?
  - How does the algorithm deal with border cases?
- Requirements on Data
  - Does the algorithm need random-access to data?
  - Does it need anything more than “compare” and “swap”?
- Space Usage
  - Can the algorithm sort in-place, or does it need extra space?
- Stability
  - Is the algorithm “stable”
    (will it ever reverse the order of equivalent items?)

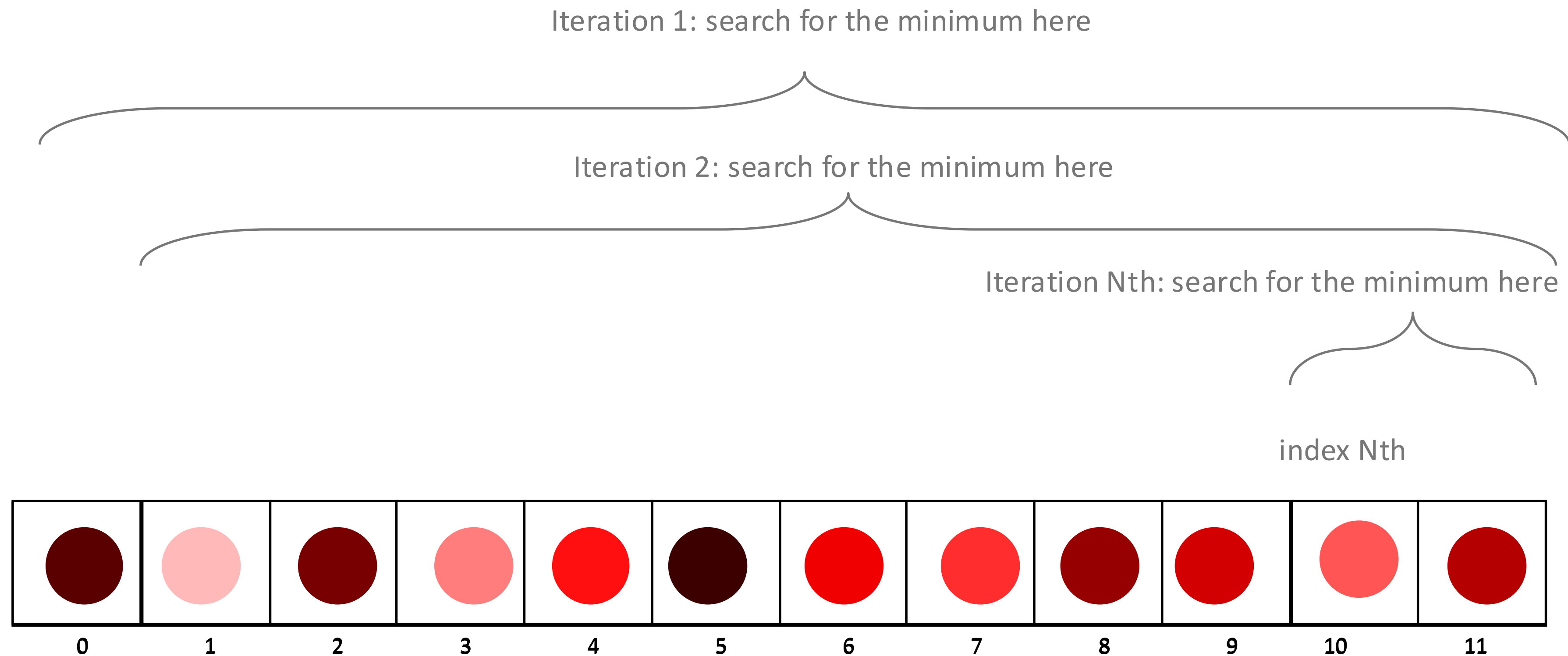
Examples of Sorting Algorithms

Name	Best Case	Average Case	Worst Case	Space Usage	Stability	Method Used
Selection Sort	n^2	n^2	n^2	1	No	Selection
Heap Sort	n log n	n log n	n log n	1	No	Selection
Insertion Sort	n	n^2	n^2	1	Yes	Insertion
Shell Sort	n log n	n^4/3	n^3/2	1	No	Insertion
Merge Sort	n log n	n log n	n log n	n	Yes	Merging
Bubble Sort	n	n^2	n^2	1	Yes	Swapping
Quick Sort	n log n	n log n	n^2	log n	No	Partitioning
Tree Sort	n log n	n log n	n log n	n	Yes	Insertion

Ways of sorting

- Selection-based sorts:
  - find the next largest/smallest item and put in place
  - build the correct list in order incrementally
- Insertion-based sorts:
  - for each item, insert it into an ordered sublist
  - build a sorted list, but keep changing it
- Compare-and-Swap-based sorts:
  - find two items that are out of order, and swap them
  - keep “improving” the list

Selection-based Sorts



Examples:

- Selection sort (slow) - repeatedly finding the minimum element from the unsorted part of the array and moving it to the beginning.
- Heap sort (fast) – sort based on heap data structure (partially ordered binary tree).

Insertion-based Sorts

Index 2

Index Nth

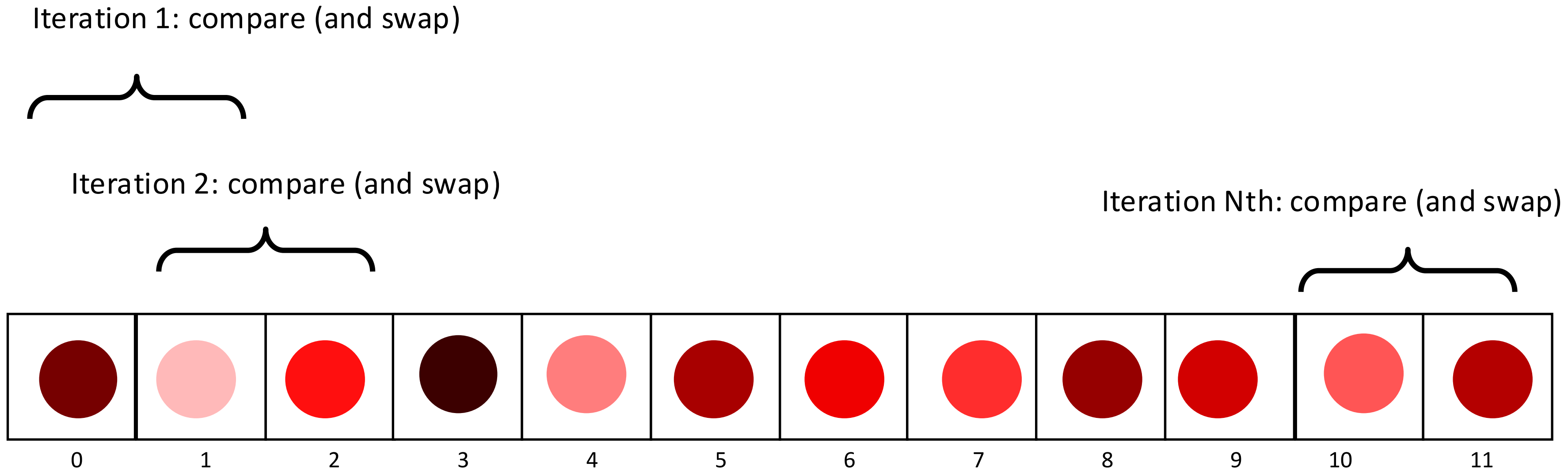
Index 1

0 1 2 3 4 5 6 7 8 9 10 11

Examples:

- Insertion sort (slow) - iteratively inserting each element in an unsorted list into its correct position in a sorted portion of the list.
- Shell sort (pretty fast) - Sorts the elements far apart from each other and successively reduces the interval between the elements to be compared.
- Merge sort (fast) – based on the divide and conquer algorithm.

Compare and Swap Sorts

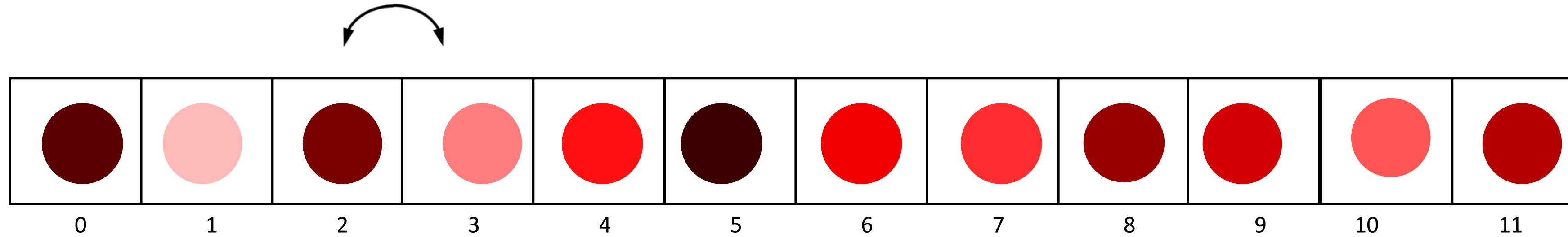


Examples:

- Bubble sort (easy but terrible performance) - things bubble up quickly, but bubble down slowly.
- Quick sort (very fast) – based on the divide and conquer algorithm.

Other (Hybrid) Sorts

Auxiliary algorithms i.e. permutation,
randomisations, radix digit, etc.



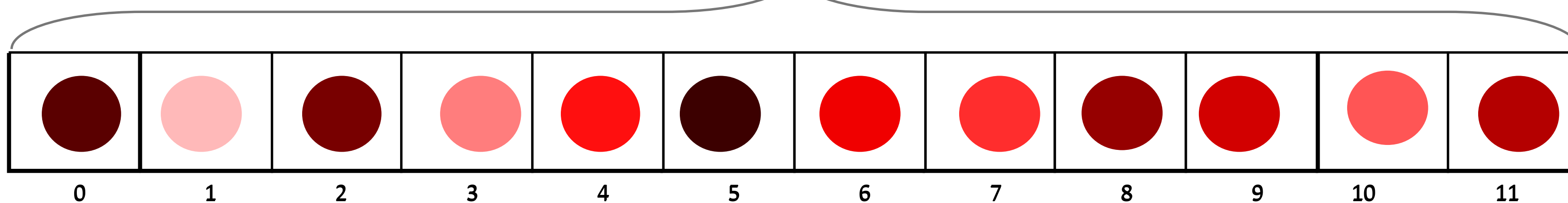
Examples:

- Permutation sort (very slow) – permutation of its input (a.k.a. Bogo sort).
- Random sort (i.e. generate and test) (fast) - uses randomisation of numbers.
- Radix sort (fast) - sorts numbers by processing individual digits, efficient for those represented in a certain radix. (i.e. only works with certain data types).

Selection Sorts

```
import java.util.*;

public void selectionSort(E[ ] data, int size, Comparator<E> comp) {
    // for each position, from 0 up, find the next smallest item
    // and swap it into place
    for (int i=0; i<size-1; i++) {
        int minIndex = i;

        for (int j=i+1; j<size; j++)
            if (comp.compare(data[j], data[minIndex]) < 0)
                minIndex=j;

        swap(data, i, minIndex);
    }
}
```

```
Private void swap(List<E> data, int from, int to) Collection.swap(data, from, to);
```



Selection Sorts: Cost

- Algorithm of selection sort:

```
selectionSort(array, size)
  for i from 0 to size - 1 do
    set i as the index of the current minimum
    for j from i + 1 to size - 1 do
        if array[j] < array[current minimum]
            set j as the new current minimum index
    if current minimum is not i
        swap array[i] with array[current minimum]
end selectionSort
```

- Analysis of an algorithm
  - Algorithm complexity: O(n^2) – 2 loops, so structural complexity is n*n = n^2.
  - Time complexity: Worst case – O(n^2) – array is ascending or descending order; Best case (Ω) – O(n^2) – when array sorted already; Average case (Θ) – O(n^2) – array is jumbled.
  - Space complexity: O(1) due to extra variable min_idx is used.

Selection Sorts: Stable or Unstable?

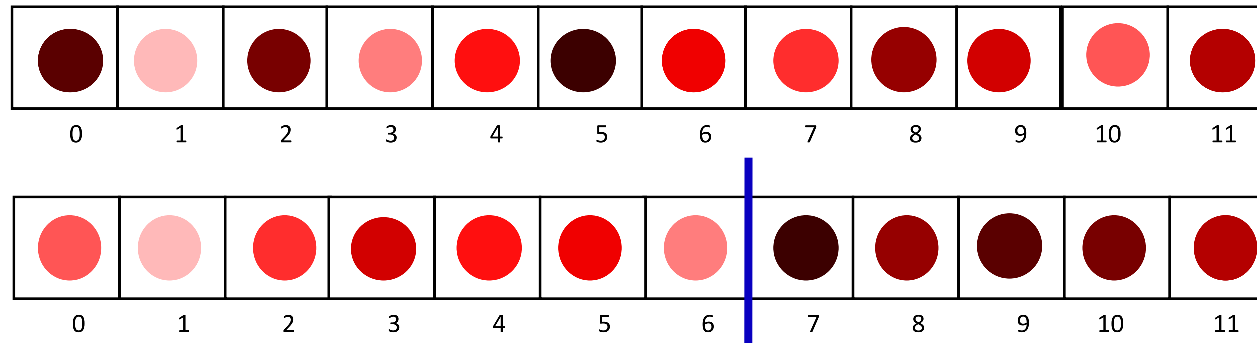
- Stable: if two objects with equal keys appear in the same order in sorted output as they appear in the input data set, i.e. the equivalent elements retain their relative positions, after sorting.
- Other stable sorting algorithms:
  - Selection, bubble sort, insertion sort, and merge sort maintain stability by ensuring that the sorted array is filled in reverse order so that elements with equivalent keys have the same relative position.
- Other unstable sorting algorithms:
  - Quick sort: relative order of equal elements can change during the partitioning process.
  - Heap sort: heap operations can change the relative order of equal elements
  - Radix sort: its stability depends on another sort, with the only requirement that the other sort should be stable.

QuickSort

- Uses Divide and Conquer, but does its work in the “split” step
- Split the array into parts, by choosing a “pivot” item, and making sure that:
  - all items < pivot are in the left part
  - all items > pivot are in the right part
- Then (recursively) sort each part

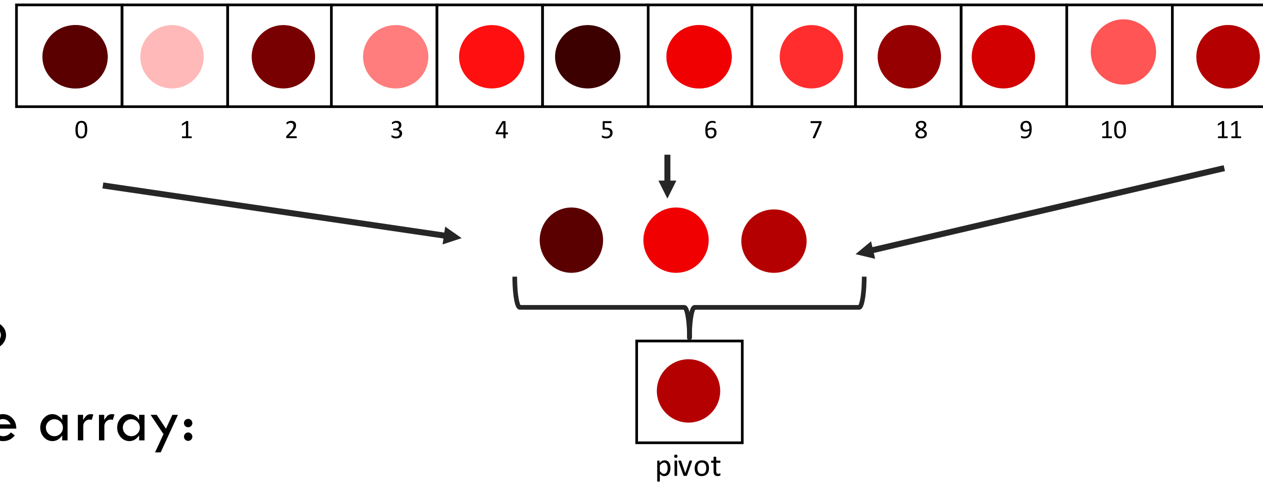
note: it won’t usually be an equal split

- The work is done in the partition method:

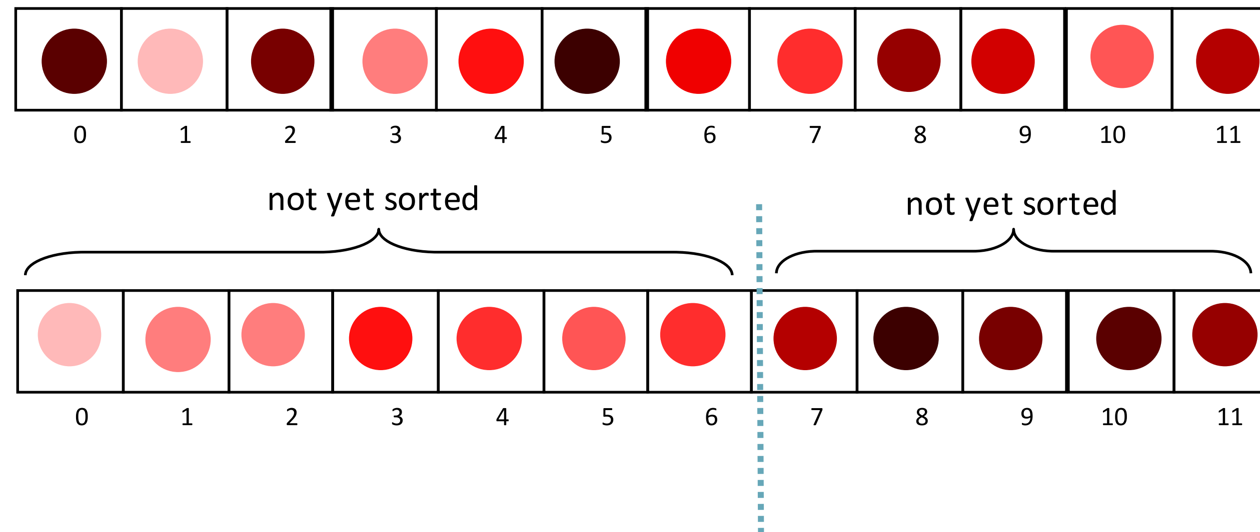


QuickSort: simplest version

1. Choose a pivot:

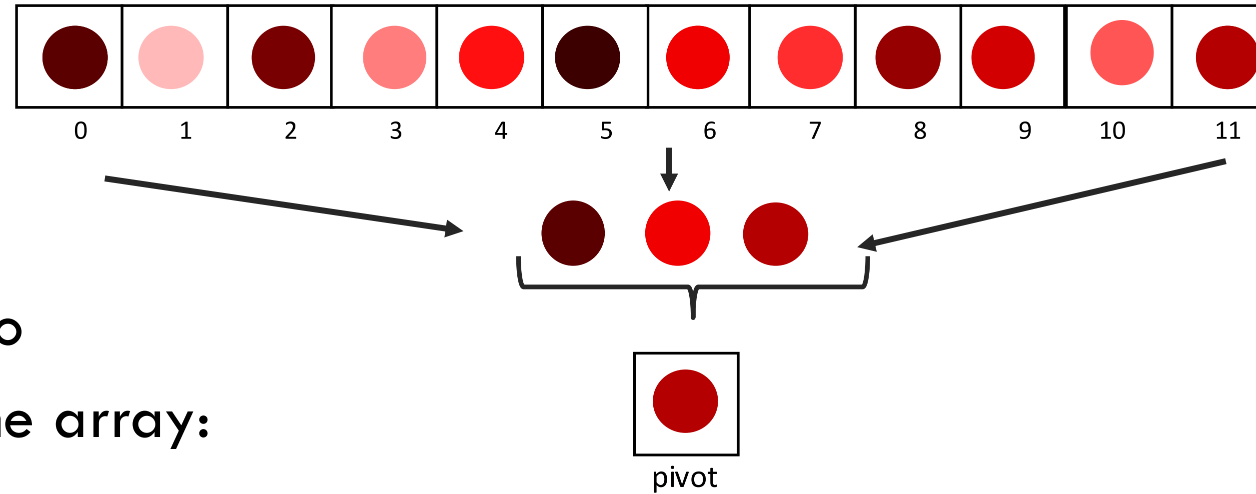


2. Use pivot to partition the array:

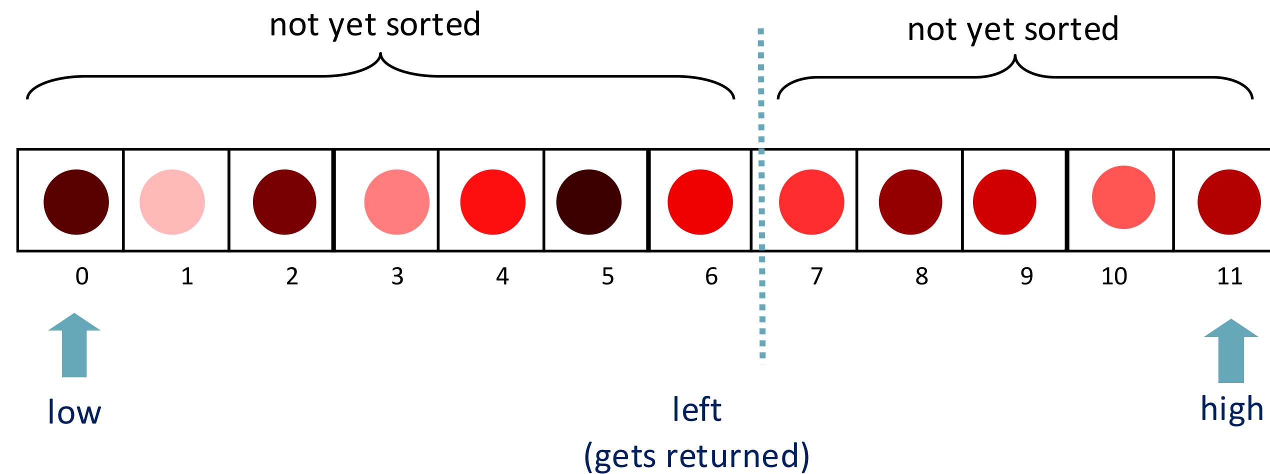


QuickSort: in-place version

1. Choose a pivot:



2. Use pivot to partition the array:



QuickSort

Here's how we start it off:

```
public static <E> void quickSort( List<E>] data, Comparator<E> comp)  {
        quickSort (data,   0,   data.size(),   comp);
}
```

QuickSort

```
public static <E> void quickSort( List<E>] data, Comparator<E> comp)  {
     quickSort (data,   0,   data.size(),   comp);
}

public static <E> void quickSort(List<E>] data, int low, int high,
                                          Comparator<E> comp){
    if (high-low < 2) { return; }    // only one item to sort.
    if (high-low < 4) { sort3(data, low, high, comp);}  // only 2 or 3 items to sort.
    else {
        int mid = partition(data, low, high, comp);      // split:  mid = boundary
        quickSort(data, low, mid, comp);
        quickSort(data, mid, high, comp);
    }
}
```

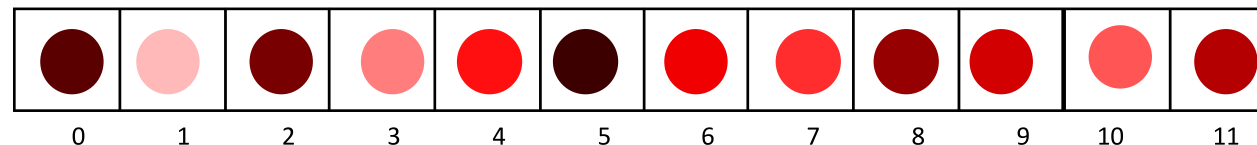
QuickSort: partition

```
/** Partition into small items (low..mid-1) and large items (mid..high-1)
private static <E> int partition(List<E> data, int low, int high, Comparator<E> comp){
    E pivot = medianOf3(data, low, high-1, low+high)/2, comp);
    int left = low-1;
    int right = high;
    while( left <= right ){
        do {  left++;                    // on left, skip over items < pivot
        } while (left<high &&comp.compare(data.get(left), pivot)< 0);

        do { right--;                    // on right, skip over items > pivot
        } while (right>=low && comp.compare(data.get(right), pivot)> 0);

        if (left < right)   { Collections.swap(data, left, right); }
    }
    return left;
}
```

0 1 2 3 4 5 6 7 8 9 10 11

QuickSort: partition

```
/** Partition into small items (low..mid-1) and large items (mid..high-1)
private static <E> int partition(List<E> data, int low, int high, Comparator<E> comp){
    E pivot = medianOf3(data, low, high-1, low+high)/2, comp);
    int left = low-1;
    int right = high;
    while( left <= right ){
        do {  left++;                          // on left, skip over items < pivot
        } while (left<high &&comp.compare(data.get(left), pivot)< 0);
        do { right--;                          // on right, skip over items > pivot
        } while (right>=low && comp.compare(data.get(right), pivot)> 0);
        if (left < right)   { Collections.swap(data, left, right); }
    }
    return left;
}
```



QuickSort: algorithm

```
Algorithm of quick sort:
function quickSort(array, low, high) {
   if (low < high) {
    // Choose a pivot i.e. pivotIndex = partition(array, low, high);
    // Recursively sort sub-arrays i.e. quickSort(array, low, pivotIndex - 1);
       and quickSort(array, pivotIndex + 1, high);
   }
}

function partition(array, low, high) {
       // (Implementation details for partitioning the array around the pivot)
       // Return the index where the pivot should be placed
}
```

```
public static <E> void quickSort (List<E> data,  int low,  int high, Comparator<E> comp) {
    if (high > low +2) {
        int mid = partition(data, low, high, comp);
        quickSort(data, low, mid, comp);
        quickSort(data, mid, high, comp);
    }
}
```

QuickSort: cost

Cost of Quick Sort:

- three steps:
  - partition: has to compare (high-low) pairs
  - first recursive call
  - second recursive call

- If quicksort divides the array exactly in half, then:

  C(n) = log(n) x n

  = n log(n) comparisons

  = O(n log(n)) (best case)

QuickSort Cost:

- If Quicksort divides the array exactly in half, then:
  - C(n) = log(n) x n
    $\rightarrow$ n log(n) comparisons
    = O(n log(n)) (best case)

- If Quicksort divides the array into 1 and n-1:
  - C(n) = n + (n-1) + (n-2) + (n-3) + … + 2 + 1
    = n(n-1)/2 comparisons
    = $O(n^2)$ (worst case)

- Average case?
  - very hard to analyse.
  - still O(n log(n)), and very good.

Insertion and selection-based sorts are:

- All slow (except insertion sort on almost-sorted lists).
- O(n2).

QuickSort: Stable or Unstable?

- QuickSort:
  - Unstable: Partition “jumps” items to the other end
    ⇒ two equal items likely to reverse their order
  - Cost depends on choice of pivot.
    - Simplest choice is very slow: $O(n^2)$ even on almost sorted lists
    - Better choice (median of three) ⇒ $O(n \log(n))$ on almost sorted lists
  - In-place
    - A sorting process that requires only a constant amount of extra memory space, apart from the input data.
    - An algorithm rearranges the elements of the array or list directly within the original data structure, without needing significant additional storage.

Stable or Unstable? Almost-sorted?

- MergeSort:
  - Stable: doesn’t jump any item over an unsorted region
    ⇒ two equal items preserve their order
  - Same cost on all input
  - “natural merge” variant doesn’t sort already sorted regions
    ⇒ will be very fast: O(n) on almost sorted lists
  - Needs extra space
- QuickSort:
  - Unstable: Partition “jumps” items to the other end
    ⇒ two equal items likely to reverse their order
  - Cost depends on choice of pivot.
    - Simplest choice is very slow: $O(n^2)$ even on almost sorted lists
    - Better choice (median of three) ⇒ O(n log(n)) on almost sorted lists
  - In-place

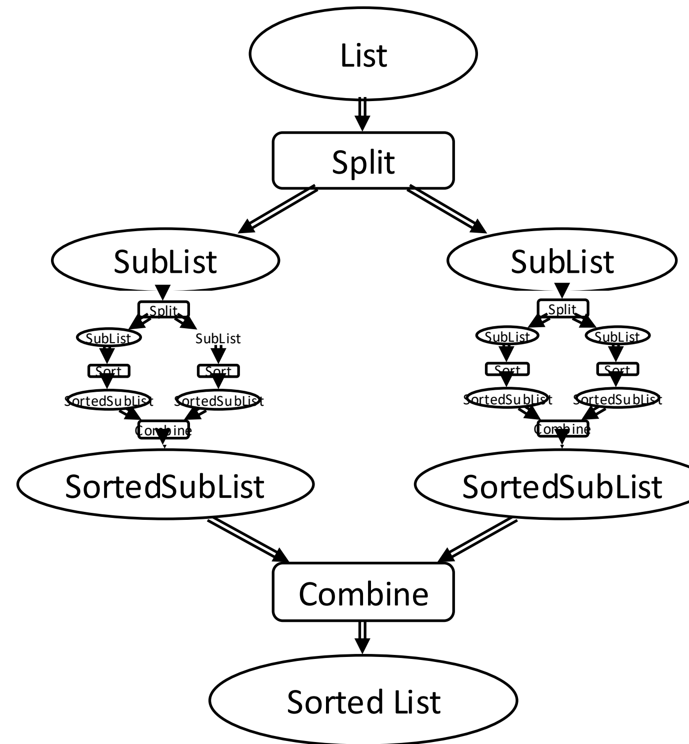
Merge Sort: Divide and Conquer Sorts

To Sort:

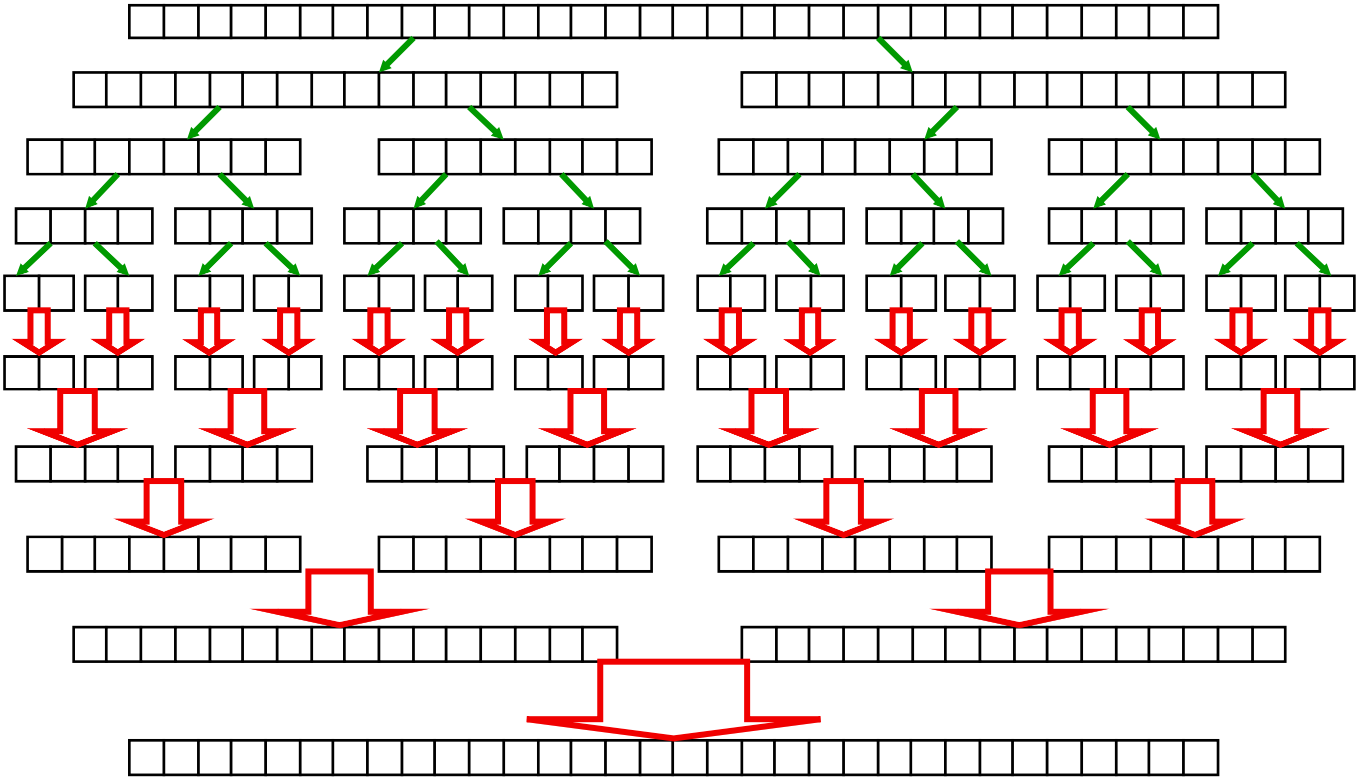
- Split
- Sort each part (recursive)
- Combine

Where does the work happen?

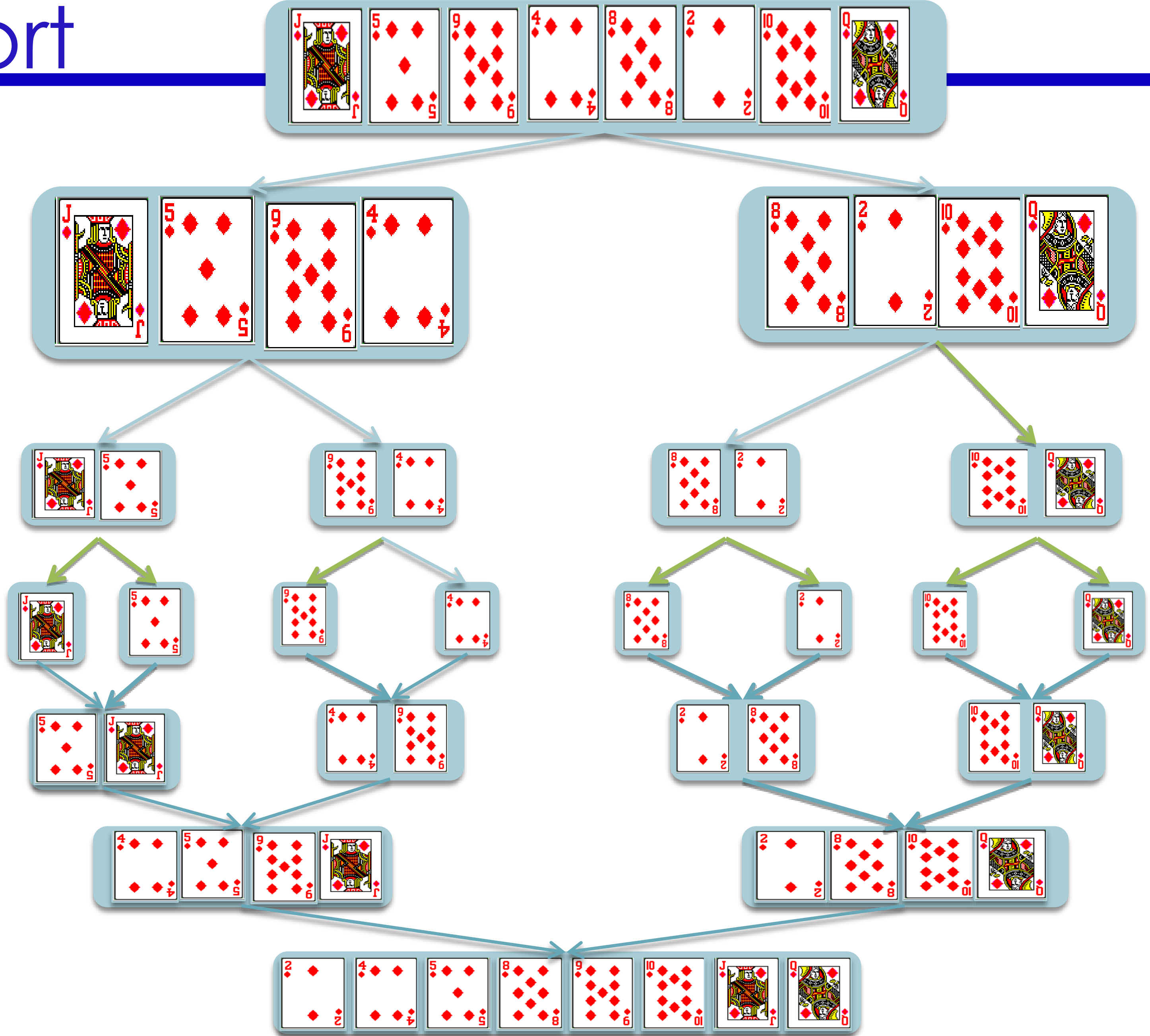
- MergeSort:
  - split is trivial
  - combine does all the work
- QuickSort:
  - split does all the work
  - combine is trivial



Merge Sort : the concept



Merge Sort



Merge

```
/** Merge from[low..mid-1] with from[mid..high-1] into  to[low..high-1.*/
private static <E> void merge(List<E> from, List<E> to, int low, int mid, int high,
 Comparator<E> comp){
    int index = low;     // where we will put the item into "to‟
    int indxLeft = low; // index into the lower half of the "from" range
    int indxRight = mid; // index into the upper half of the "from" range
    while (indxLeft<mid && indxRight < high){
        if (comp.compare(from.get(indxLeft), from.get(indxRight)) <=0)
            to.set(index++, from.get(indxLeft++));
        else
            to.set(index++, from.get(indxRight++));
    }

    // copy over the remainder. Note only one loop will do anything.
    while (indxLeft<mid)     {  to.set(index++, from.get(indxLeft++)); }
    while (indxRight<high)  {  to.set(index++, from.get(indxRight++)); }
}
```

MergeSort – a wrapper method that starts it

- It looks like we an extra temporary array for each "level" (how many levels are there?)
- Only need <u>one</u> (extra): at each layer, treat the other array as "storage"
- We start with a wrapper to make this second array, and fill it with a copy of the original data.

```
public static <E> void mergeSort(List<E> data, Comparator<E> comp){
        List<E> other = new ArrayList<E>(data);
        mergeSort(data, other, 0, data.size(), comp);
}
```

MergeSort – the recursive method

```
private static <E> void mergeSort(List<E> data, List<E> other, int low,
 int high, Comparator<E> comp){

    // sort items from low..high-1, using the other array
    if (high > low+1){
        int mid = (low+high)/2;
        // mid = low of upper 1/2, = high of lower half.
        mergeSort(other, data, low, mid, comp);
        mergeSort(other, data, mid, high, comp);
        merge(other, data, low, mid, high, comp);
    }
}
```

- there are multiple calls to the recursive method in here.
- this will make a "tree" structure
- we swap other and data at each recursive call (= each “level”)

Merge Sort costs:

Algorithm of merge sort:

1. Divide the unsorted array into two sub-arrays, half the size of the original.
2. Continue to divide the sub-arrays as long as the current piece of the array has more than one element.
3. Merge two sub-arrays together by always putting the lowest value first.
4. Keep merging until there are no sub-arrays left.

- Insertion sort, Selection Sort:
  - All slow (except Insertion sort on almost-sorted lists)
  - $O(n^2)$
- Merge Sort
  - $\log_2(n)$ levels, n comparisons at each level to merge.
  - therefore cost = $O(n \log(n))$

Merge Sort : Stable? Unstable?

- Stable: doesn’t jump any item over an unsorted region.
  - ⇒ Two equal items preserve their order.
- Same cost for all inputs.
- Needs extra space
- “natural merge” variant doesn’t sort already sorted regions
  - ⇒ will be very fast: