Data Structures and Algorithms

XMUT-COMP 103 - 2024 T1

A bit about sorting

A/Prof. Pawel Dmochowski

School of Engineering and Computer Science

Victoria University of Wellington

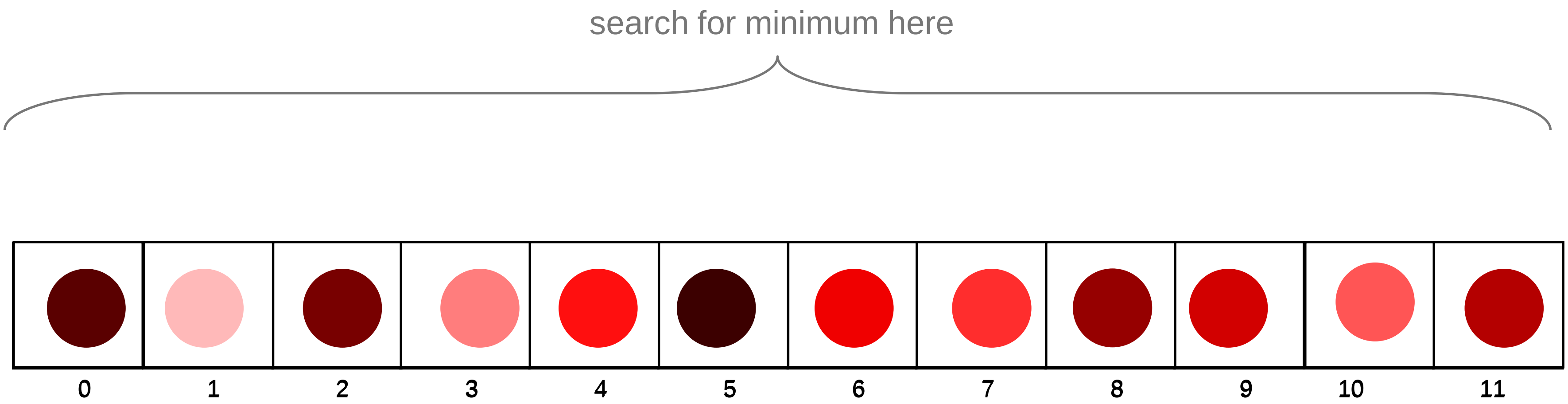
Ways of sorting

- 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

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?)

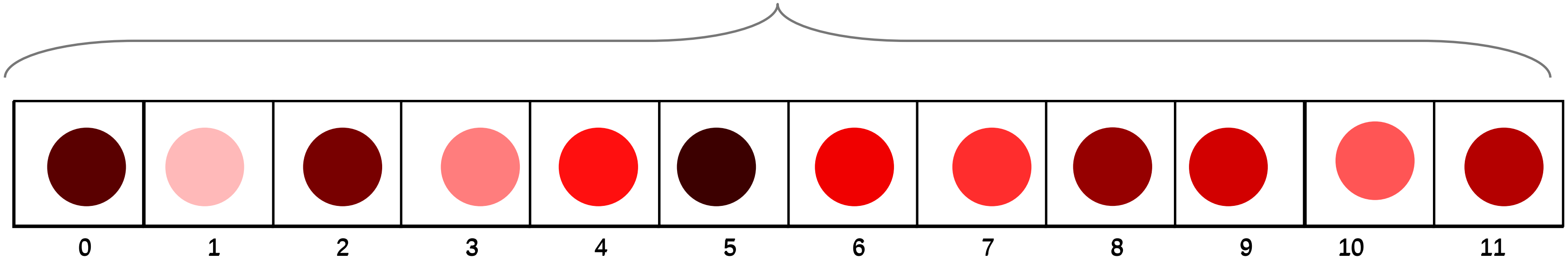
Selection-based Sorts



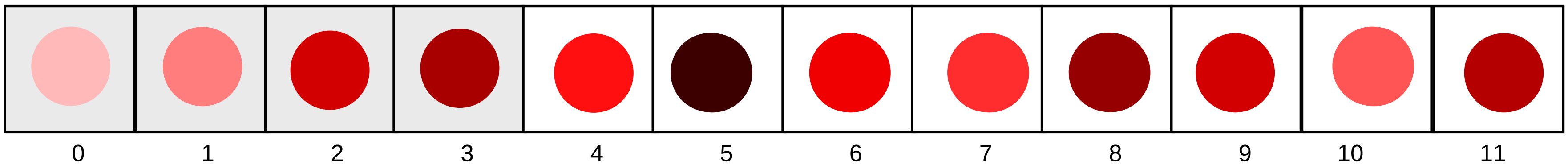
- Selection Sort (slow)
- HeapSort (fast)

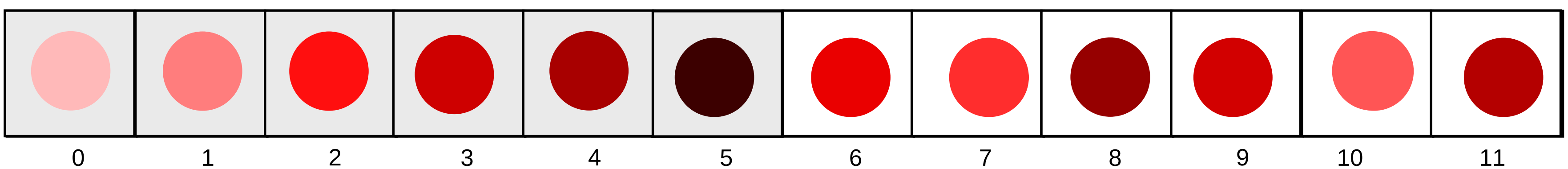
Selection Sorts

```
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);
    }
}
```



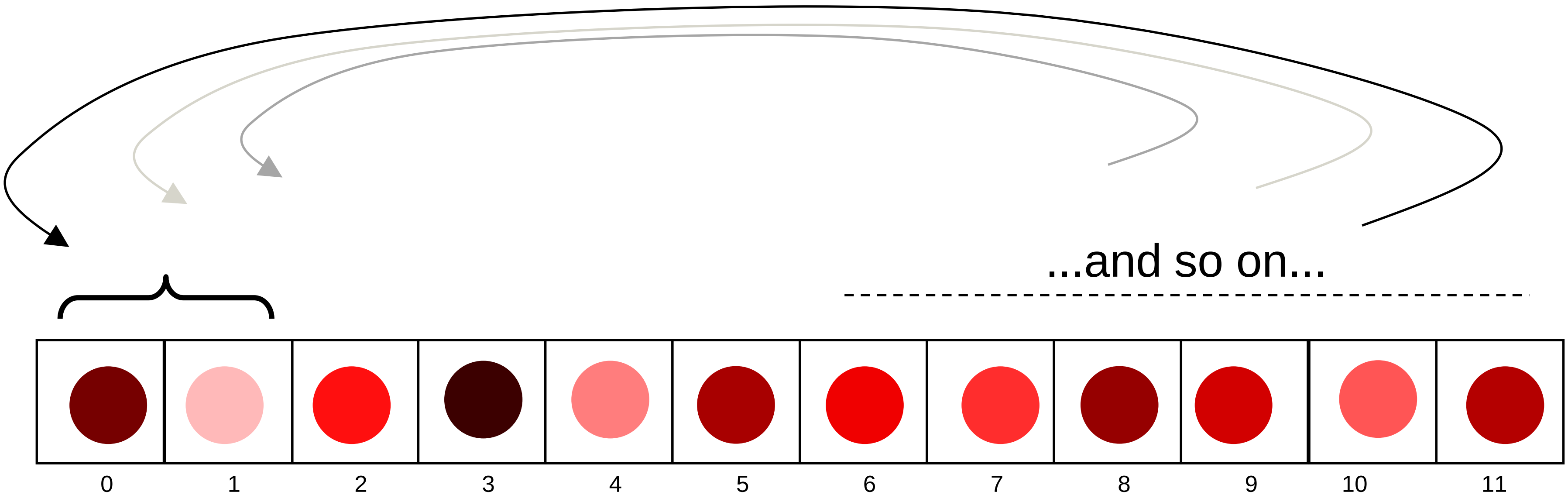
Insertion-based Sorts





- Insertion Sort (slow)
- Shell Sort (pretty fast)
- Merge Sort (fast)
  (Divide and Conquer)

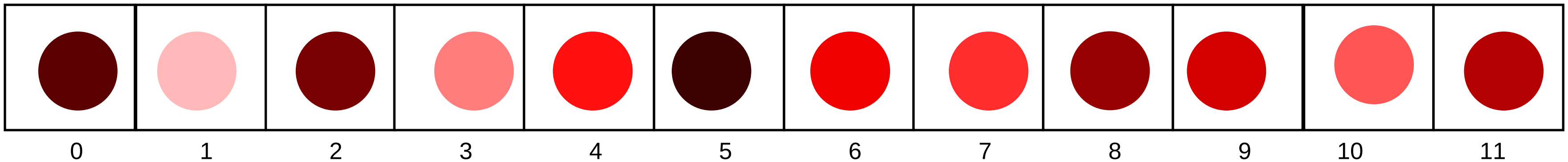
Compare and Swap Sorts



things bubble *up* quickly,
but bubble *down* slowly

- Bubble Sort (easy but terrible performance)
- QuickSort (very fast)
  (Divide and Conquer)

Other Sorts

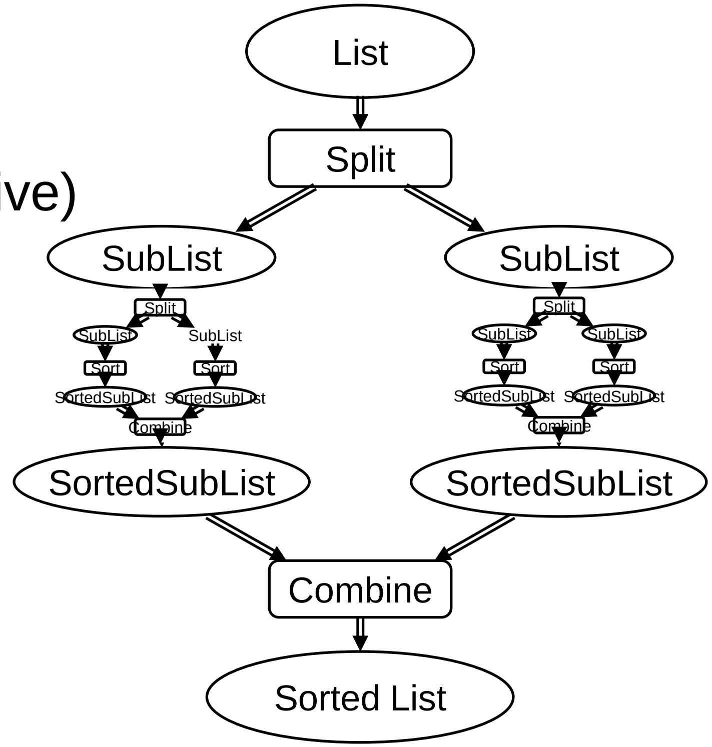


- Radix Sort (only works with certain data types)
- Permutation Sort (very slow)
- Random Sort (Generate and Test)

Divide and Conquer Sorts

To Sort:

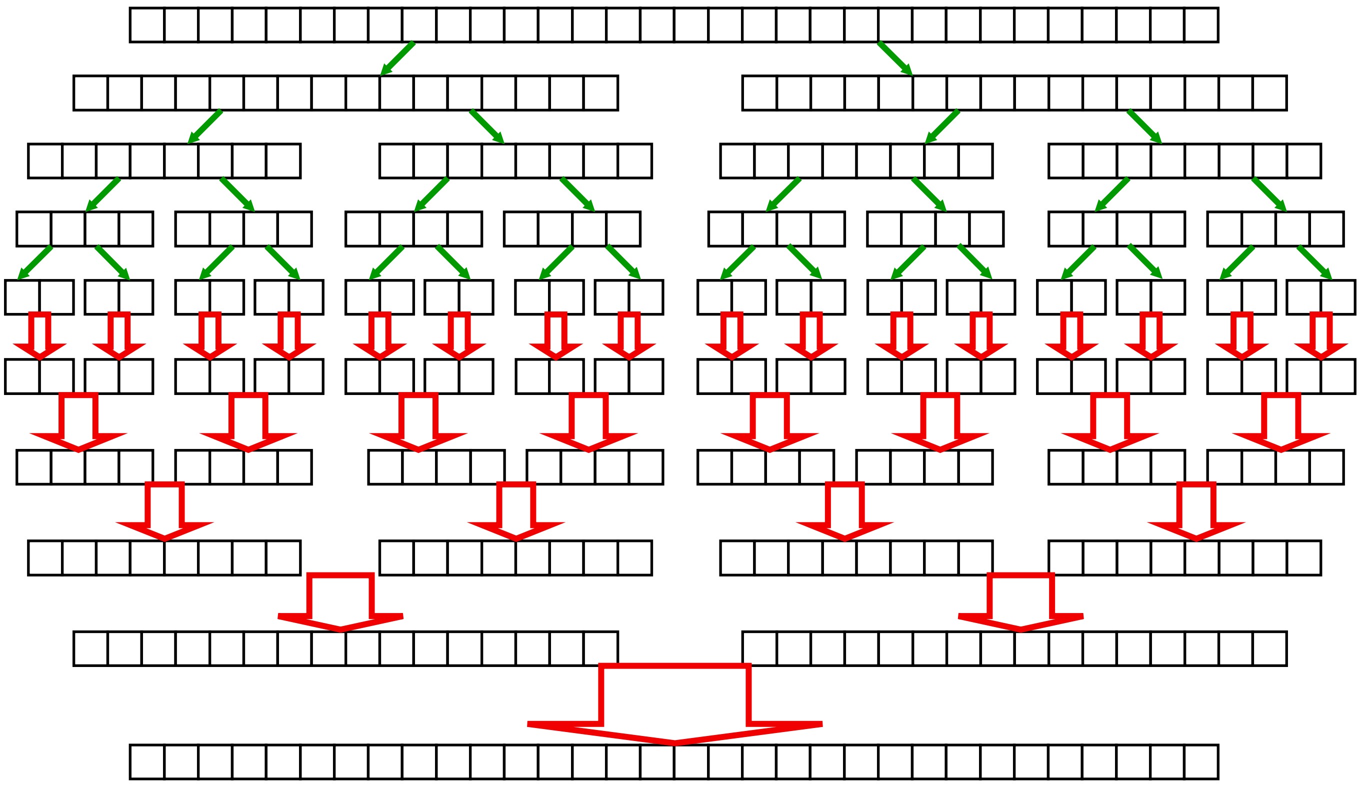
- Split
- Sort each part (recursive)
- Combine

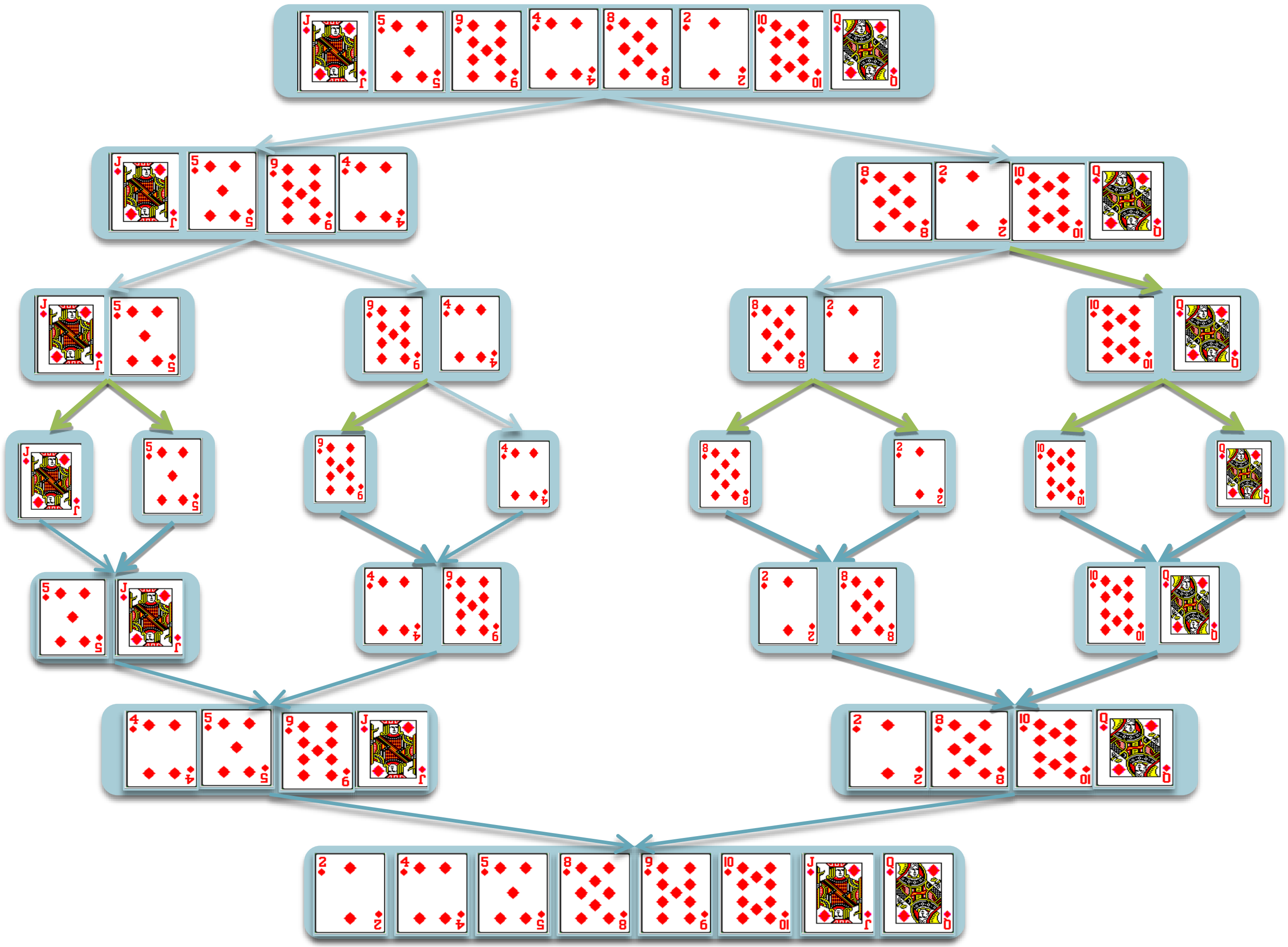


Where does the work happen?

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

MergeSort : the concept





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++));
     }
```

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 one (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”)

Sorting Algorithm costs:

- Insertion sort, Selection Sort:
  - All slow (except Insertion sort on almost-sorted lists)
  - $O(n^2)$

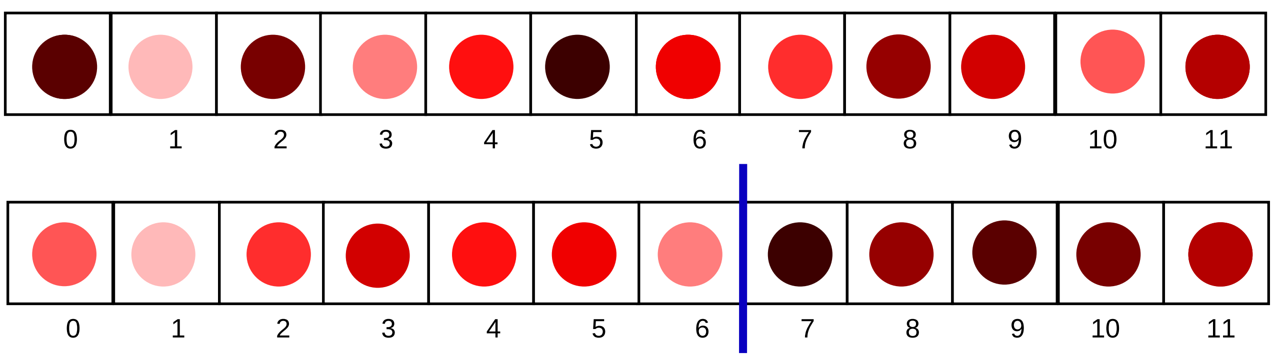
- Merge Sort
  - $\log_2(n)$ levels, n comparisons at each level to merge.
  - therefore cost = $O(n \log(n))$

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

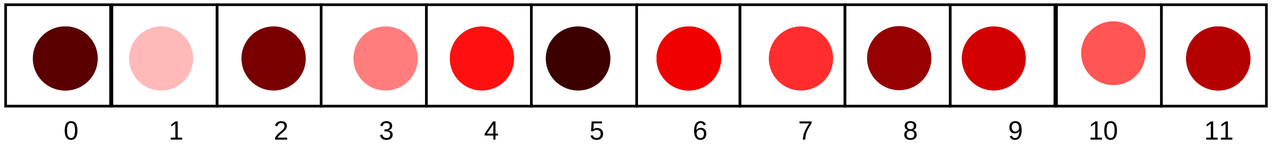
note: it won’t usually be an *equal* split

- The work is done in the partition method:

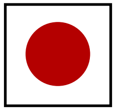


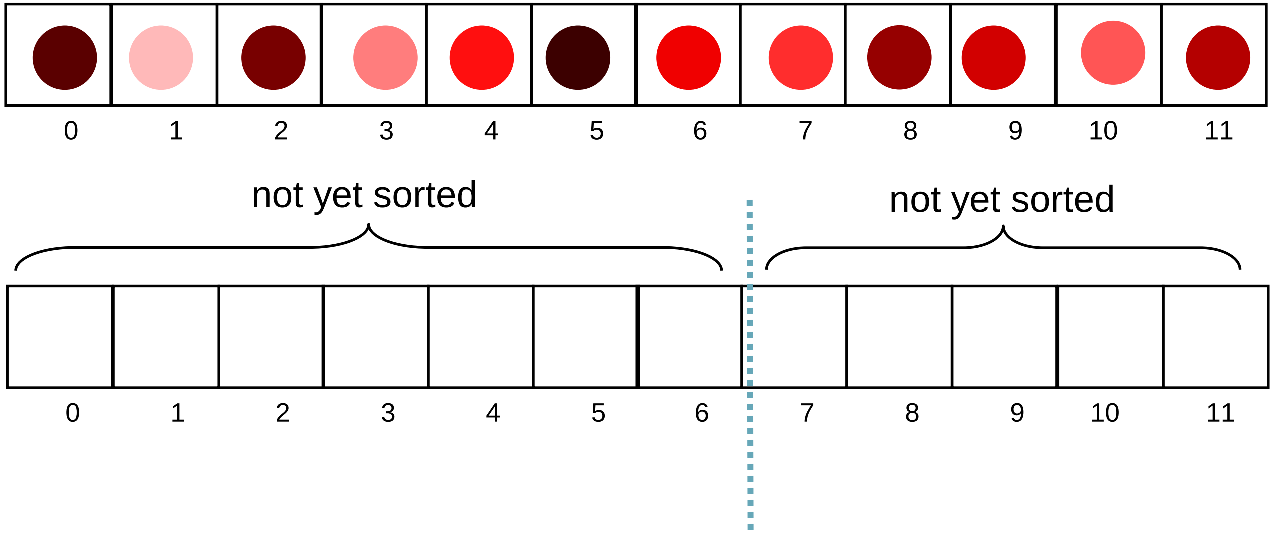
QuickSort: simplest version

1. Choose a pivot:



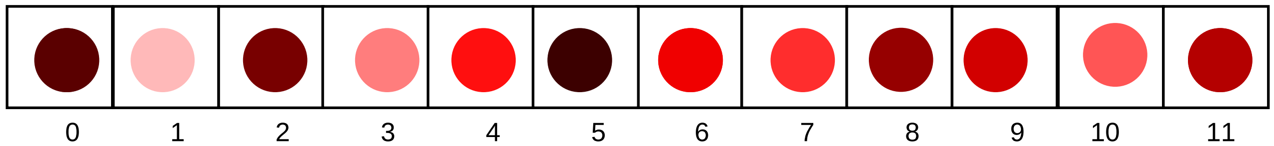
2. Use pivot to partition the array:



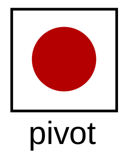


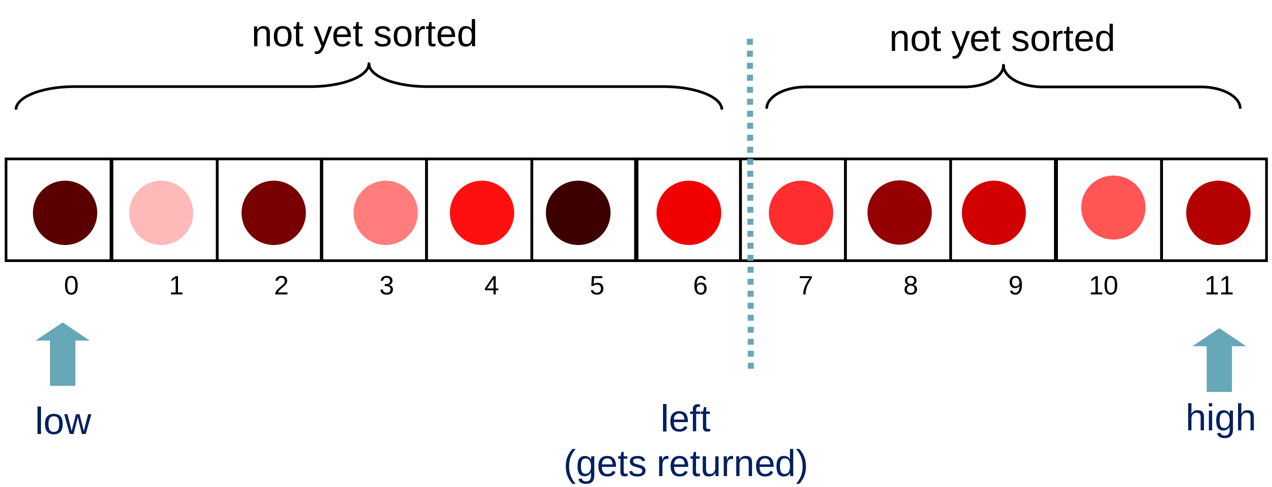
QuickSort: in-place version

1. Choose a pivot:



2. Use pivot to partition the array:





QuickSort

Here's how we start it off:

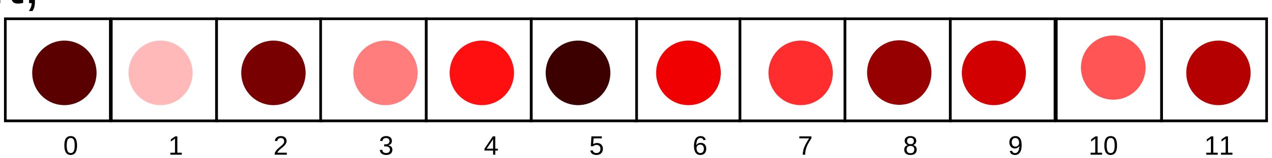
```
public static <E> void quickSort( List<E>] data, Comparator<E> comp)  {
        quickSort (data,   0,   data.size(),   comp);
}
```

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;
}
```



QuickSort: cost

```
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);
     }
}
```

Cost of Quick Sort:

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

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.

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