326.041 (2015S) – Practical Software Technology

(Praktische Softwaretechnologie)

Searching, Big O Notation, Sorting

Alexander Baumgartner
Alexander.Baumgartner@risc.jku.at

Research Institute for Symbolic Computation (RISC)
Johannes Kepler University, Linz, Austria

Linear Search int[]


```
public static int search(int[] array, int searchKey,
        int fromIdx, int toIdx) {
    for (; fromIdx < toIdx; fromIdx++) {
        if (array[fromIdx] == searchKey)
            return fromIdx;
    }
    return -1;
}
```

- Algorithm needs time linear to the size of the array.
- Can we do better if the array is sorted?
  - Yes, we can use binary search.
  - For large arrays, it is much faster than a linear search.

Binary Search int[] – Iterative


```java
public static int binarySearch(int[] array, int searchKey,
        int fromIdx, int toIdx) {
    while (fromIdx < toIdx) {
        int mid = (fromIdx + toIdx − 1) / 2;
        if (array[mid] == searchKey) return mid;
        else if (array[mid] > searchKey) toIdx = mid;
        else fromIdx = mid + 1;
    }
    return −1;
}
```

As long as the interval is not empty:

1. Set mid to the middle of the interval.
2. If the value can be found at position mid, then we are done.
3. If the value at position mid is greater, then continue searching in the lower half interval $[fromIdx, mid[$.
4. If the value at position mid is smaller, then continue searching in the upper half interval $]mid, toIdx[$.

Binary Search int[] – Recursive


```
public static int binarySearch(int[] array, int searchKey,
        int fromIdx, int toIdx) {
    if (fromIdx >= toIdx) return −1;
    int mid = (fromIdx + toIdx − 1) / 2;
    if (array[mid] == searchKey) return mid;
    if (array[mid] > searchKey)
        return binarySearch(array, searchKey, fromIdx, mid);
    return binarySearch(array, searchKey, mid + 1, toIdx);
}
```

1. Return -1 if the interval is empty.
2. Set mid to the middle of the interval.
3. If the value can be found at position mid, then we are done.
4. If the value at position mid is greater, then continue searching in the lower half interval $[fromIdx, mid]$.
5. If the value at position mid is smaller, then continue searching in the upper half interval $(mid, toIdx]$.

Logarithmic Complexity

- Binary search is a **divide and conquer** algorithm.
- It divides the size by two for each iteration.
- It needs only $\log_2(n)$ recursions, where n is the input size.

Size	Recursions Needed
10	4
100	7
1 000	10
10 000	14
100 000	17
1 000 000	20
10 000 000	24
100 000 000	27
1 000 000 000	30

Alexander.Baumgartner@risc.jku.at

Express the Efficiency Of Algorithms

- Shorthand way to say how efficient a computer algorithm is.
- In computer science, this rough measure is called “Big O” notation.
- Tells how an algorithms speed is related to the number of items.
- For example:
  - Time T needed to set an item of an array a of length n:
    - $a[i] = value$ – Does not depend on the length. T is constant.
  - Time T needed for linear search:
    - for all $x \in a$ if $x = value...$ – Test all elements. T is proportional to n.
  - Time T needed for binary search:
    - Divide the size by two for each iteration. T is proportional to $\log_2(n)$.
    - Logarithms are related by constants: $\log_2(n) = K \log(n)$, for some K.

Big O Notation

Notation	Name
$O(1)$	constant
$O(\log(n))$	logarithmic
$O(n)$	linear
$O(n\log(n))$	loglinear
$O(n^2)$	quadratic
$O(n^3)$	cubic
$O(n^c)$	polynomial
$O(c^n)$	exponential
$O(n!)$	factorial

Constants C and "lower" contribution do not matter. E.g.:

$$O(C * n) = C * O(n) = O(n) \tag{1}$$

$$O(2\log(n) + n) < O(3 * n) = O(n) \tag{2}$$

$$O(n + 2n^2 + n^3) < O(4 * n^3) = O(n^3) \tag{3}$$

Sorting an Array

Figure: Unsorted team of players

Figure: Sorted team of players

Bubble Sort

- **Compare** two items.
- If the one on the left is greater, **swap** them.
- Move one position right.

Figure: First step

Bubble Sort

- Compare two items.
- If the one on the left is greater, swap them.
- Move one position right.

Figure: Second step

Bubble Sort

- Compare two items.
- If the one on the left is greater, swap them.
- Move one position right.

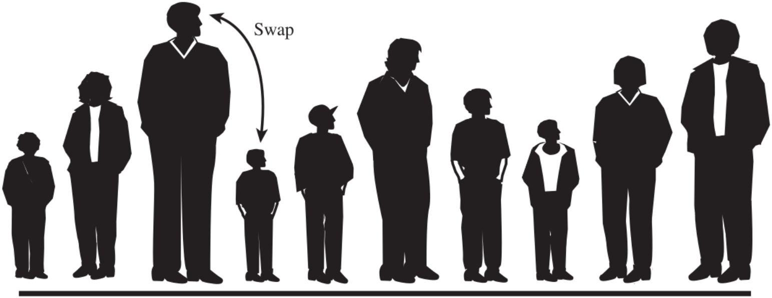


Figure: Third step

Bubble Sort

- Compare two items.
- If the one on the left is greater, swap them.
- Move one position right.

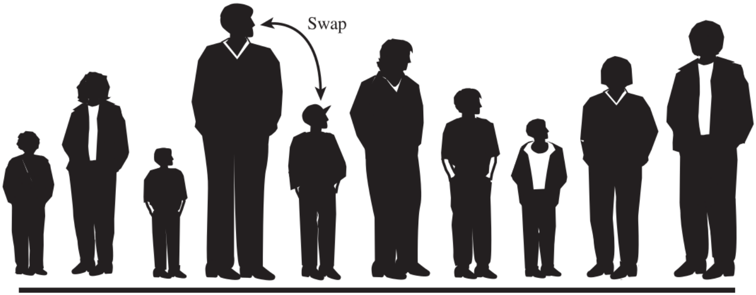


Figure: Fourth step

Bubble Sort

- Compare two items.
- If the one on the left is greater, swap them.
- Move one position right.

Figure: Bubble sort – One iteration

Bubble Sort – Java Implementation


```
int[] array = ...
...
public void bubbleSort() {
    for (int out = array.length - 1; out > 1; out--) {
        for (int in = 0; in < out; in++) {
            if (array[in] > array[in + 1])
                swap(in, in + 1);
        }
    }
}
```

- Items behind position out are always sorted.
- Bubble sort runs in $O(n^2)$ time.
    - A nested loop often leads to runtime complexity $O(n^2)$.

Insertion Sort

- Works by **partial sorting** and a **marker**.
- Items to the left of the marker are partially sorted.
- Items to the right of the marker are unsorted.

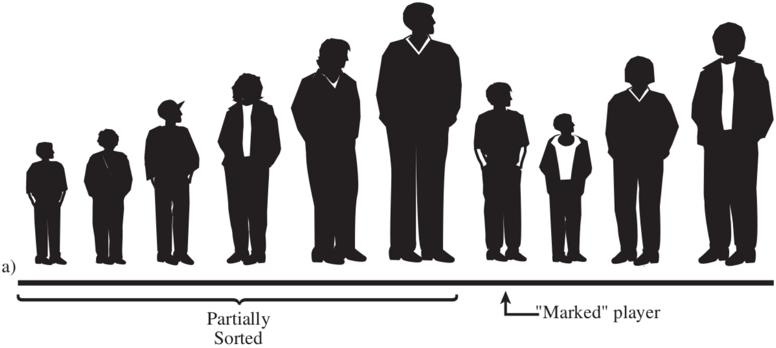


Figure: Insertion sort – Players to the left are sorted

Insertion Sort

- Works by partial sorting and a marker.
- Items to the left of the marker are partially sorted.
- Items to the right of the marker are unsorted.

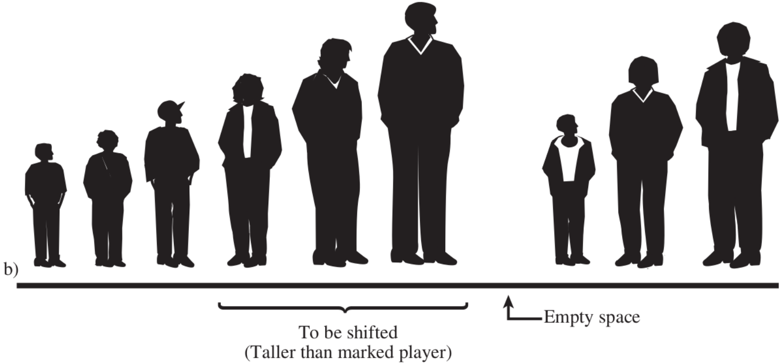


Figure: Insertion sort – Insert marked player at right position

Insertion Sort

- Works by partial sorting and a marker.
- Items to the left of the marker are partially sorted.
- Items to the right of the marker are unsorted.

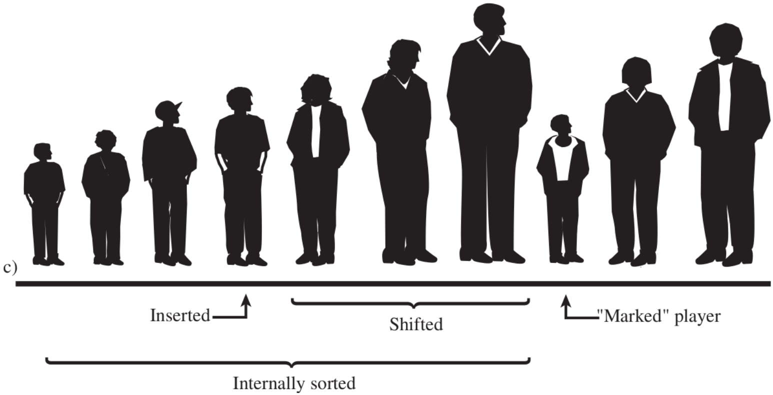


Figure: Insertion sort – Mark next player and repeat the process

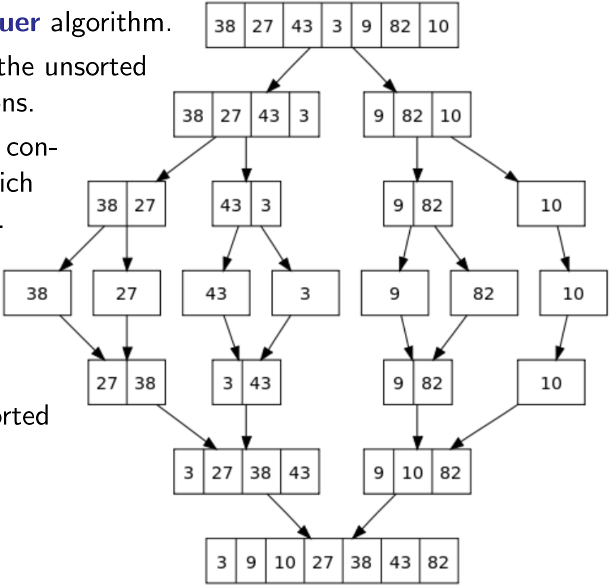
Insertion Sort – Java Implementation


```
int[] array = ...
...
public void insertionSort() {
    for (int out = 1; out < array.length; out++) {
        int temp = array[out];
        int in = out;
        for (; in > 0 && array[in-1] >= temp; in--)
            array[in] = array[in-1];
        array[in] = temp;
    }
}
```

- out starts at 1 and moves right.
- $temp$ marks the leftmost unsorted item.
- in starts at out and moves left.
- Insertion sort still runs in $O(n^2)$ time.

Mergesort

- Is **divide and conquer** algorithm.
- Successively divide the unsorted array into n partitions.
- Until each partition contains 1 element, which is considered sorted.
- Repeatedly **merge** partitioned units.
- Until there is only 1 sublist remaining which will be the sorted list.



Alexander.Baumgartner@risc.jku.at

Mergeing Arrays

- The heart of the mergesort algorithm is the merging of two already-sorted arrays.
- Merging requires $O(n)$ time.

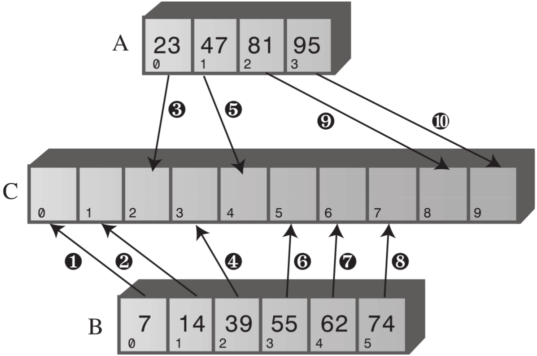


Figure: Merging the arrays A and B into C

Mergeing – Java Implementation

```
public static int[] merge(int[] arrayA, int[] arrayB) {
    int[] arrayC = new int[arrayA.length+arrayB.length];
    int aDex = 0, bDex = 0, cDex = 0;
    while (aDex < arrayA.length && bDex < arrayB.length){
        if (arrayA[aDex] < arrayB[bDex])
            arrayC[cDex++] = arrayA[aDex++];
        else
            arrayC[cDex++] = arrayB[bDex++];
    }
    while (aDex < arrayA.length)
        arrayC[cDex++] = arrayA[aDex++];
    while (bDex < arrayB.length)
        arrayC[cDex++] = arrayB[bDex++];
    return arrayC;
}
```

Mergesort – Java Implementation

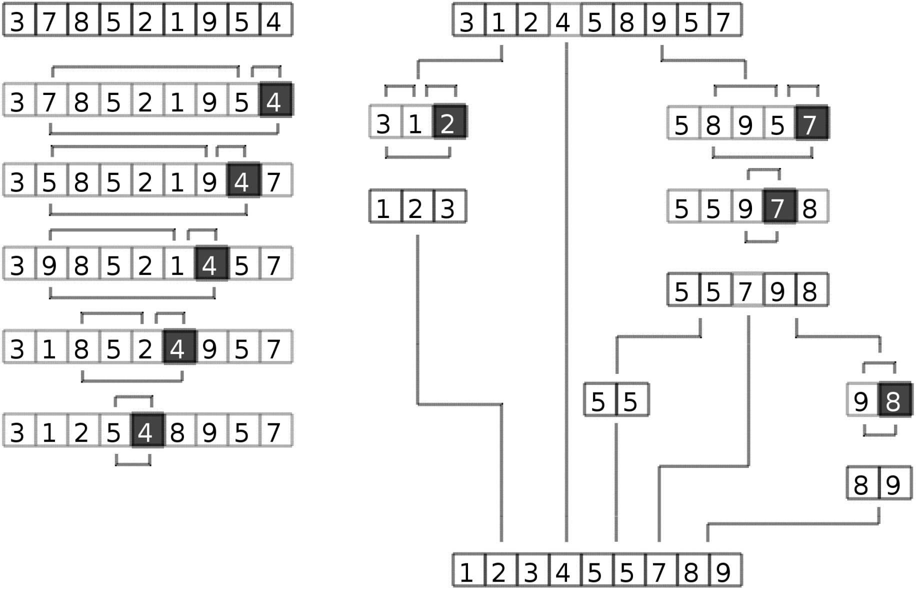

```
public void mergeSort() {
    doMergeSort(0, size - 1);
}

private void doMergeSort(int left, int right) {
    if (left < right) {
        int mid = left + (right - left) / 2;
        doMergeSort(left, mid);
        doMergeSort(mid + 1, right);
        mergeParts(left, mid, right);
    }
}
```

- If $left \geq right$, then it is either one or no element.
- mid divides the array in two parts.
- Recursively sort the left part.
- Recursively sort the right part.
- Merge the sorted parts.
- Mergesort runs in $O(n \log(n))$ time.

Quicksort – Basic Idea

- Choose an element, called **pivot**, from the list.
- **Partition** the list so that:
  - The pivot is in its final place.
  - All elements to the left of pivot are smaller.
  - All elements to the right of pivot are larger.
- **Recursively** apply the above steps to the two partitions.
- Is also a divide and conquer algorithm.

Quicksort – Illustration



Quicksort – Java Implementation


```java
public void quickSort(int left, int right) {
    if (right - left > 0) {
        int partition = partitionIt(left, right);
        quickSort(left, partition - 1);
        quickSort(partition + 1, right);
    }
}
```

- Quicksort runs in $O(n^2)$ time for the worst case.
- Quicksort runs in $O(n \log(n))$ time for the average case.
- Quicksort can be faster than Mergesort for the average case.
  - Different selection strategies for pivot.
  - Random pivot.
  - Median-of-3 pivot.

Exercise

- Create a class IntArray which represents a dynamically growing integer array for storing values of primitive type int without autoboxing. (Like demonstrated in the lecture.)
- Implement a modified version of the insertionSort() method from the lecture so that it removes duplicates as it sorts.