

# Introduction to Programming Concepts and Tools

Lecture 7:

## Sorting, Algorithm Analysis

Peter Sestoft: Searching and sorting with Java

Section 3, 4.1-4.6, 5.1-5.7

Pohl & McDowell: Java by Dissection.

Chp. 5.5-5.7

# Today's Lecture:

- Sorting
  - Selection Sort
  - Quick Sort
  - Heap Sort
  
- Analyzing Algorithm
  - Time Complexity measures
  - Big Oh

# Selection Sort

Find the least element in the array

move it to the front

Find the next to least element in the array

move it second to the front

....

Repeat until all elements have been selected  
in order of size.

# Selection Sort

Find the least element in the array

swap it with the first

Find the next to least element in the array

swap it with the second

....

Repeat until all elements have been selected  
in order of size.

# Selection Sort

```
for (int i = 0; i < arr.length; i++){  
    find the least element in arr within the range  
        arr[i] to arr[arr.length-1]; say that it is arr[j].  
    Exchange arr[i] and arr[j].  
}  
  
/
```

# Selection Sort

```
// Sorts the elements of its array parameter
Public void selectionSort(int[] arr){
    for (int i = 0; i < arr.length; i++)
        // invariant: subarray arr[0]..arr[i-1] is sorted
        {int j = findLeast(arr, i, arr.length-1);
         int temp = arr[i];
         arr[i] = arr[j];
         arr[j] = temp;
        }
}
```

# Selection Sort

```
// finds the index of the smallest element
// in arr[start]..arr[end]
private int findLeast(int[] arr, int start, int end){
    int least = start;
    for (int i = start+1; i <= end; i++){
        // invariant: least is index of the least element
        // in range arr[start]..arr[i-1]
        {if (arr[i] < arr[least])
            least = i;
        }
    }
    return least;
}
```

# efficiency of Selection Sort

- How many comparisons are required to sort  $n$  elements?
  - the first iteration requires  $n-1$  comparisons  
the next requires  $n-2$  comparisons....

together:  $(n-1) + (n-2) + \dots + 2+1 = n * (n-1)/2$

# Quicksort

invented by C.A.R.Hoare 1962

Quicksort solves the sorting problem by

## **Divide and Conquer**

a problem is solved by

- splitting it into smaller sub-problems
- solving the sub-problems (by divide and conquer, very small problems may have trivial solutions)
- combining the solutions for the sub-problems to get a solution to the initial problem

# Quicksort

- Divide the data into two sets,  
    ‘small elements’ and ‘large elements’
- Sort each of the two parts separately
- Combine the solutions

Quicksort sorts an array without using another array

# Quicksort

- Chose an element  $x$  from the array.  
 $x$  is called the pivot
- Move the elements of the array such that:  
elements less than or equal to  $x$  are to the left,  
elements greater than or equal to  $x$  are to the  
right
- Quicksort the smaller elements (left sub-array)  
Quicksort the larger elements (right sub-array)

# Quicksort

```
// Sorts the elements of its array parameter
```

```
public static void quickSort(int[] arr){
```

```
    qSort(arr, 0, arr.length-1);
```

```
}
```

```
private static void qSort(int[] arr, int start, int end){
```

```
    .....
```

```
}
```

# Quicksort

```
private static void qSort(int[] arr, int start, int end){
    if (start < end){
        int i = start, j = end, x = arr[(i+j)/2];
        do {
            while (arr[i] < x) i++;
            while (arr[j] > x) j--;
            if (i <= j) {
                swap(arr, i, j); i++; j--;
            }
        } while (i <= j);
        qSort(arr, start, j);
        qSort(arr, i, end);
    }
}
```

# Correctness of Quicksort

How can we argue that Quicksort will actually sort the array?

# Correctness of Quicksort

Claim:

- a call to qsort operates on a sub-array and does not touch the rest of the array
- if the sub-array has 0 or 1 elements, then it is already sorted and nothing is done

else

# Correctness of Quicksort

Claim:

- before each iteration of the do-loop it holds that  
all elements in  $\text{arr}[\text{start}..i-1] \leq x$   
all elements in  $\text{arr}[j+1..\text{end}] \geq x$
- each iteration of the do-loop increases  $i$  and decreases  $j$  by at least 1.
- the do-loop will terminate and after it has terminated  $i > j$

# Correctness of Quicksort

- before each iteration of the do-loop it holds that

all elements in  $\text{arr}[\text{start}..i-1] \leq x$

all elements in  $\text{arr}[j+1..\text{end}] \geq x$

- it holds before the first iteration
- when it holds before an iteration, then after

```
while (arr[i] < x) i++;
```

```
while (arr[j] > x) j--;    it holds that
```

$\text{arr}[\text{start}..i-1] \leq x, \text{arr}[i] \geq x, \text{arr}[j+1..\text{end}] \geq x, \text{arr}[j] \leq x$

- if  $(i \leq j)$  holds then after

```
if (i <= j) {swap(arr, i, j); i++; j--;}
```

it holds that  $\text{arr}[\text{start}..i-1] \leq x, \text{arr}[j+1..\text{end}] \geq x$

before the next iteration of the do-loop

# Correctness of Quicksort

- after `if (i <= j) {swap(arr, i, j); i++; j--;}`  
it holds always no matter if swap has been executed or not,  
that  $\text{arr}[\text{start}..i-1] \leq x$  and  $\text{arr}[j+1..\text{end}] \geq x$
- If the do-loop terminates then  $(i > j)$  and it holds that  
 $j \leq i-1$ ,  $j+1 \leq i$   
 $\text{arr}[\text{start}..j] \leq x$ ,  $(\text{arr}[j+1/i-1] = x,)$   $\text{arr}[i..\text{end}] \geq x$

# Correctness of Quicksort

- after the do-loop:

```
qSort(arr, start, j);  
qSort(arr, i, end);
```

- these recursive calls are with smaller sub-arrays  
we can inductively assume that they sort the sub-arrays

then after these calls it will hold that  $(i > j)$  and  
 $\text{arr}[\text{start}..j] \leq x$  and sorted,  $(\text{arr}[j+1/i-1] = x)$ ,  
 $\text{arr}[i..end] \geq x$  sorted

hence:  $\text{arr}[\text{start}..end]$  is sorted

# Quicksort

```
private static void qSort(int[] arr, int start, int end){
    if (start < end){
        int i = start, j = end, x = arr[(i+j)/2];
        do {
            while (arr[i] < x) i++;
            while (arr[j] > x) j--;
            if (i <= j) {
                swap(arr, i, j); i++; j--;
            }
        } while (i <= j);
        qSort(arr, start, j);
        qSort(arr, i, end);
    }
}
```

# termination of Quicksort

How can we know that the two while loops

```
while (arr[i] < x) i++;
```

```
while (arr[j] > x) j--;
```

will terminate?

-----

when you have a while loop in a program,  
ensure yourself and others that it will terminate

# termination of Quicksort

```
do { // it holds at this point for the sub-array before each round
// that arr[i..end] holds an element  $\geq x$ 
// and arr[start..j] holds an element  $\leq x$ 
    while (arr[i] < x) i++;
    while (arr[j] > x) j--;
    if (i <= j) {
        swap(arr, i, j); i++; j--;
    }
} while (i <= j);
```

In the first round  $i=start$ ,  $j=end$  and  $x$  is in  $arr[start..end]$

In later rounds it holds that  $arr[start] \leq x$  and  $arr[end] \geq x$

# termination of Quicksort

How can we know that the do-loop  
will terminate?

# termination of Quicksort

```
// The do-loop terminates when (i > j)
do {
    while (arr[i] < x) i++;
    while (arr[j] > x) j--;
    if (i <= j) {
// if the loop-termination condition is not fulfilled
// then i will increase by 1 and j will decrease by 1
        swap(arr, i, j); i++; j--;
    }
} while (i <= j);

- - - - -
// in each iteration i is increased by at least 1 and
// j is decreased by at least 1

// In finite time we will have i > j
```

# efficiency of Quicksort

- How many comparisons are required to sort  $n$  elements?
  - each partition requires that each element in the sub-array is compared to  $x$  so this makes 'the length of the sub array' number of comparisons

worst case: the pivot is the least or greatest element in each iteration. Total number of comparisons

$$n + (n-1) + (n-2) + \dots + 2 + 1 = (n+1)*n/2$$

best case: the pivot end up in the middle in each iteration

each sub-sort sort half the number of elements

$$T_{\text{best}}(n) = n + 2 * T_{\text{best}}(n/2) = n + 2*(n/2 + 2 * T_{\text{best}}(n/4))$$

$$n + n + 4 * T_{\text{best}}(n/4) = n + n + \dots + n = n * \log(n)$$

In practice Quicksort is fast, but it can be slow in cases special

# Improving Quicksort

Quicksort is slow when the pivot is the least or greatest element in the sub-array

The risk can be reduced by choosing the pivot as the median between 3 values from the array

# Analyzing Algorithms

- problem,  
instance of a problem  
algorithm to solve the problem
- time complexity
  - theoretical: mathematical analysis of the time used by the algorithm as a function of the size of the instance
  - empirical: measuring time for different input sets

# Analyzing Algorithms

## Comparing Algorithms

- machine independence
- model of computation
  - which operations are considered to be elementary and be performed in constant time
  - size of problem instans
  - asymptotical analysis
- expressed in *order of  $f(n)$*

# Big Oh

Let  $f$  and  $g$  be functions  $N \rightarrow R_+$

$f$  is at most the order of  $g$ ,

$f(n)$  in  $O(g(n))$ ,  $f(n) = O(g(n))$  iff

there exists positive numbers  $c, m$  such that

for all  $n \geq m$  it holds that  $f(n) \leq c * g(n)$

# Big Oh

- An algorithm that requires  $O(\log n)$  time is better than an algorithm that requires  $O(n)$  time

there might be small values  $n$  for which the second is faster

A table of running times was shown at the lecture.

We prefer algorithms with a time complexity  
from the top of this list

$\log n$   
 $n$   
 $n \cdot \log n$   
 $n^2$   
 $n^3$   
 $2^n$   
 $3^n$

We do not want algorithms with an exponential time complexity

# Analyzing Algorithms

- size of an instance of a problem
- worst case / average case analysis
- Example:
  - instance: an actual array to be sorted
  - size, number of elements in an array
  - the performance of an algorithm may depend on the actual elements, e.g. if the array is already partly sorted.

- Comparing algorithms
- Heap Sort

# Heap Sort

Williams,/Floyd 1962/1964

Binary tree

An array thought of as a binary tree:

node in  $arr[i]$  has children  $arr[2i+1], arr[2i+2]$

0

1 2

3 4 5 6

7 8 9 10 11 12 13 14

# Heap

heap condition: the value of a node in the tree is  $\geq$  the value of its children

leaves satisfy the heap condition

heap: binary tree where all nodes satisfy the heap condition

# Heapifying

If the root-node is the only node not satisfying the heap condition, then we can repeatedly swap it with the largest of its children.

The index  $i$  doubles each time we go down the tree, so this requires at most  $\log(n)$  operations, when the tree holds  $n$  nodes

A whole binary tree can be heapified, by heapifying from below. The leafs are already heaps. Heapify those nodes which children are leafs. ect.

A tree with  $n$  nodes has  $n/2$  inner nodes. A single node requires at most  $\log(n)$  operations. The tree requires at most  $n/2 * \log(n)$

# Heap Sort

- think of the array as a binary tree
- heapify the array, now the largest element is in `arr[0]`
- Swap `arr[0]` with `arr[last]`, now `arr[last]` is sorted
- Heapify `arr[0..last-1]`, now `arr[0]` holds the largest element in this heap
- Repeat until all elements have been extracted
- The sorted array is build from right to left

# Efficiency of Heap Sort

- heapifying the array takes time proportional to  $n \cdot \log(n)$
- there are  $n$  extractions, each heapifying of the root takes  $\log(n)$  operations. This gives  $n \cdot \log(n)$
- Together this gives  $2 \cdot n \cdot \log(n)$ , proportional to  $n \cdot \log(n)$
- Heap sort is better than Quicksort in the worst case and of same order in the average case. In practice Quicksort is often faster.

# Heap Sort

```
public static void heapsort(int[] arr, int n){
    // heapify the array from below, leafs are in largest indexes in arr
    for (int m = n/2-1; m > 0; m--){
        heapify(arr,m,n-1);
    }
    // repeat: extract the root by swapping,
    // heapify the new root in the remaining part of the heap
    for (int m = n-1; m >= 1; m--){
        swap(arr,0,m);
        heapify(arr,0,m-1);
    }
}
```

-----

the initial heap is in arr[0..n-1]

# Heap Sort

```
private static void heapify(int[] arr, int i, int k){
    int j = 2*i+1;
    if (j<= k){    //at least one child
        // if rightmost child exists and is larger take that
        if (j+1 <= k && arr[j] < arr[j+1]) j++;
        if (arr[i] < arr[j]){
            swap(arr,i,j);    //now arr[j] holds the value we are moving down
            heapify(arr,j,k);
        }
    }
}
```

-----

k is the last index in the heap arr[0..k]

i is the index we want to heapify

j, j+1 are indexes of the children of arr[i] if they exists