Notes for "Algorithms and datastructures"

Simon Holm

DM578 - Algorithms and datastructures

Teacher: Rolf Fagerberg

Contents

1 Introduktion

Notes and exercises for lectures and TA-sessions. Note that mistakes in note and/or exercises may occur.

2 Notes

2.1 Introduction

Analasis recipe
We need the following

- **Model problem.** Individually for all problems (usually pretty basic).
- **Model machine.** Use the RAM-model.
- **Quality** Focus on time consumption.
- **Mathematical tools**

Analysing an algorithm
Note: This course is centered around the analysing, rather than implimenting and testing algorithms

1. Invent (come up with an idear)
2. Compare (how does 1 idear compare with another?)
3. Use the best one/ones (often: quality = time spent)

note: use argumentation and proof

2.2 Jigsaw puzzle and cycles (*kredse*)

Various methods of solving puzzle

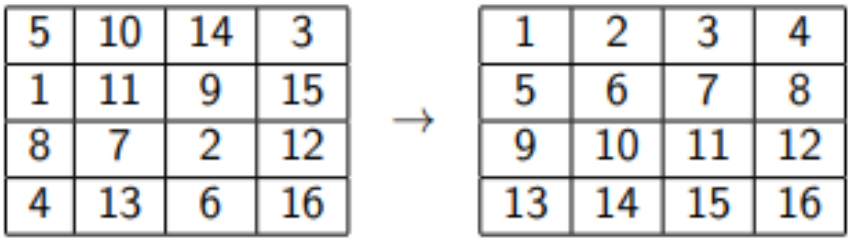


Figure 1:

One way would be using an array

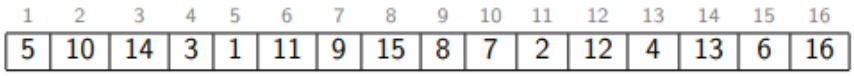


Figure 2:

Note: A cycle is a series of moves to move an element from one position to the correct one.
Using cycles (*kredse*) we can determind when a puzzle is solved. Once all elements have their own uniqe cycle pointing to their own curret position, the puzzle is solved.

2.3 RAM-model

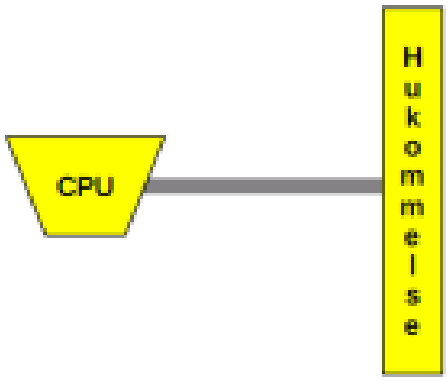


Figure 3: RAM-model

- Memory: A large row of cells, each able to contain a number or symbol. (NB: list/array)
- A CPU has a number of basic operations:
  - +, -, ·, compare, ... the contents of two cells
  - Move the contents of two cells between them.
  - jump in program (⇒ loop, branching, function/method-call)

  Note: all the basic operations are assumed to have the same time complexity

  - **Time:** number of basic operations
  - **Space:** max number of occupied memory-cells.

2.4 Sorting algorithms

2.4.1 Insertionsort

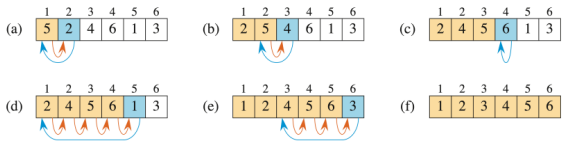


Figure 4: How to do Insertionsort

Algorithm 1 INSERTIONSORT(A)

```
1: for i = 1 to A.length do
2:     key = A[i]
3:     j = i − 1
4:     while j > 0 and A[j] < key do
5:         A[j + 1] = A[j]
6:         j = j − 1
7:     A[j + 1] = key
```

Insertionsort time complexity:

- Best: $c \cdot (n-1) = c \cdot n$
- Worst: $\frac{(n+1) \cdot n}{2}$

Analyzing time complexity in Insertionsort

2.4.2 Selectionsort

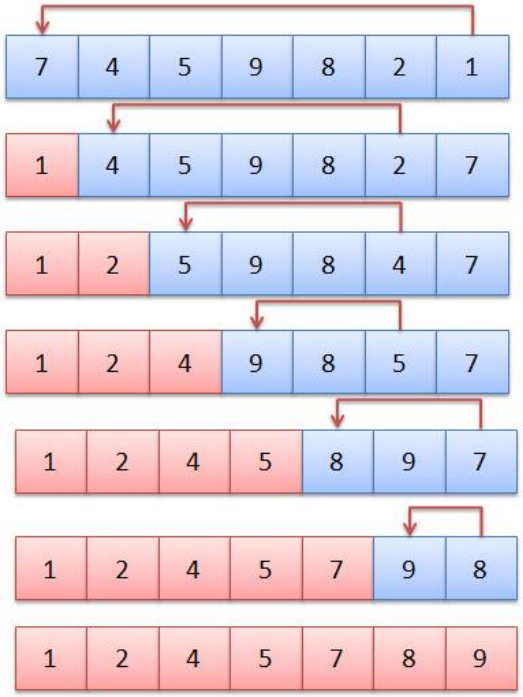


Figure 5: How to do Selectionsort

sorts an array by repeatedly selecting the smallest (or largest) element from the unsorted portion and swapping it with the first unsorted element. This process continues until the entire array is sorted.

Algorithm 2 SELECTIONSORT(A)

```
1: for i = 0 to A.length do
2:     123
3:     for 123 do
4:         if 123 then
5:     123
```

- time complexity: $c \cdot n^2$

2.4.3 MERGE-SORT(A, p, r)

Sorting a list by spiting the list in smaller lists, and merging them. *Example below*

Algorithm 3 MERGESORT(A, p, r)

```
1: if p ⩾ r then                                   ▷ zero or one element
2:     return
3: q = ⌊(p + r)/2⌋)
4: MERGE-SORT(A, p, q)
5: MERGE-SORT(A, q + 1, r)
6: MERGE(A, p, q, r)
```

Merging two sorted lists as one sorted list
While we could just sort the union of two lists
It is fast to:

- **Repeat:** Move the smallest element of the two first element from List A and B to the front of the new list.
- **time complexity:** $n \cdot log(n)$

Algorithm 4 MERGE(A, p, q, r)

```
1: n_L = q − p + 1                                 ▷ length of A[p : q]
2: n_R = r − q                                     ▷ length of A[q − 1 : r]
3: let L[0 : n_L − 1] and R[0 : n_R − 1] be new arrays
4: for i = 0 to n_L − 1 do                         ▷ copy A[p : q] into L[0 : n_L − 1]
5:     L[i] = A[p + i]
6: for j = 0 to n_R − 1 do                         ▷ copy A[q + 1 : r] into R[0 : n_R − 1]
7:     R[j] = A[q + j + 1]
8: i = 0                                           ▷ i indexes the smallest remaining element in L
9: j = 0                                           ▷ j indexes the smallest remaining element in R
10: k = p                                          ▷ k indexes the location in A to fill
11: while i < n_L and j < n_R do                   ▷ As long as L and R is not empty
12:     if L[i] ⩽ R[j] then
13:         A[k] = L[i]
14:         i = i + 1
15:     else if A[k] = R[j] then
16:         j = j + 1
17:     k = k + 1
18: while i < n_L do
19:     A[k] = L[i]
20:     i = i + 1
21:     k = k + 1
22: while j < n_R do
23:     A[k] = L[i]
24:     j = j + 1
25:     k = k + 1
```

2.4.4 Quicksort

Quicksort is a divide-and-conquer sorting algorithm that selects a pivot, partitions the array into elements less than and greater than the pivot, and recursively sorts both parts. The process repeats until the array is sorted. Its average and best-case time complexity is O(n log n), but worst-case is O(n^2) if poorly partitioned.

Algorithm 5 QUICK-SORT$(A, low, high)$

```
1: if low < high then
2:     q = PARTITION(A, low, high)
3:     QUICK-SORT(A, low, q − 1)
4:     QUICK-SORT(A, q + 1, high)
```

Algorithm 6 PARTITION$(A, low, high)$

```
1: x = A[high]
2: for j = low to high do
3:     if A[j] ⩽ x then
4:         i = i + 1
5:         swap A[i] and A[j]
```

2.4.5 Counting sort

Counting Sort is a non-comparative sorting algorithm that sorts elements by counting occurrences and using that count to place elements in the correct order. It works efficiently for integers within a known range but requires extra space for the count array.

Algorithm 7 COUNTING-SORT(A, p, r)

```
1: let B[0 : n] and C[0 : k] be new arrays
2: for i = 0 to k do
3:     C[i] = 0
4: for i = 0 to n do
5:     C[A[i]] = C[A[j]] + 1
6: for i = 1 downto k do
7:     C[i] = C[i] + C[−1]
8: for j = n downto 0 do
9:     B[C[j]] = A[j]
10:    C[A[j]] = C[A[j]] − 1
```

2.4.6 Radix sort

Radix Sort sorts numbers by processing digits from least to most significant using a stable sort (like Counting Sort). It groups numbers by each digit's value, starting from the rightmost digit, and repeats for all digits until fully sorted.

Algorithm 8 RADIX(A, n, d)

```
1: for i = 1 to d do                    ▷ go through all digits
2:     Use a stable sort to sort A on digit i.
```

- Time complexity is $O(nk)$

2.5 Asymptotic notation

For the following: Consider $f(n)$ and $g(n)$ functions $N \to R$

O : **Big O**: $f(n) = O(g(n))$

$$f(n) \leqslant g(n)$$

Note: $f(n)$'s growth rate is equal or as fast as the functions from $g(n)$'s class.

o : **Small o**: $f(n) = o(g(n))$

$$f(n) < C \cdot g(n)$$

Note: $f(n)$'s growth rate is slower than all the functions from $g(n)$'s class.

For this

$$\lim_{x \to \infty} \frac{f(n)}{g(n)} = 0$$

Ω : **Omega**: $f(n) = \Omega(g(n))$

$$f(n) \geqslant C \cdot g(n)$$

Note: $f(n)$'s growth rate is at least as fast as the functions from $g(n)$'s class.

ω : **Small omega**: $f(n) = \omega(g(n))$

$$f(n) > g(n)$$

Note: $f(n)$'s growth rate is faster than all the functions from $g(n)$'s class.

For this

$$\lim_{x \to \infty} \frac{f(n)}{g(n)} = \infty$$

Θ : **Theta**: $f(n) = \Theta(g(n))$

$$f(n) = g(n)$$

Note: $f(n)$s growth rate is equal to the functions from $g(n)$'s class.
For this, there exits constants $C_1, C_2 > 0$ such that:

$$C_1 \leqslant \frac{f(n)}{g(n)} \leqslant C_2$$

2.6 Tools for algorithm analysis

- Constants does not matter.
- $f(n)$'s class functions are all functions on the form $\{c \cdot f(n) \mid \forall c > 0\}$
- Good to know.

  If $\frac{f(n)}{g(n)} \to k < 0$ when $n \to \infty$, then it holds that $f(n) = \Theta(g(n))$

  If $\frac{f(n)}{g(n)} \to 0$ when $n \to \infty$, then it holds that $f(n) = o(g(n))$

  Good reminder: $\forall a, b \mid a > 0,\ b > 1$ it holds that $\frac{n^a}{b^n} \to 0$ for $n \to \infty$

  Note: Any polynomial is $o(n)$ to any exponential

  Also: $\forall a, b \mid a, d > 0,\ c > 1$ it holds that $\frac{(log_c\, n)^a}{n^d} \to 0$ for $n \to \infty$

  Note: Any logarithm (even when raised in a potency) is $o(n)$ to any polynomial

- **Note:** Any term with a higher power can dominate easily.

  $2^n > n^{++} > n^+ > log_c\, n > n$

2.6.1 Extending time complexity

- When extending time complexity for $time(n)$ ::

  $time(n) = c \cdot n^3$

  $time(2n) = c \cdot (2n)^3 = c \cdot 2^3 \cdot n^3 = c \cdot n^3 \cdot 8 = time(n) \cdot 8$

  Here the extended time depends heavily on the exponent.

2.6.2 Heap

We know that for a binary tree $n(x) = 2^h - 1$
A binary tree with value in all leaves might not have at flat floor. If all the leaves on the floor are fixed the left side of the tree, the tree is heap-shaped.
Note: A full tree is also heap-shaped

Heap order means that a tree is sorted with parents being more valuable than children.
For a knot (n_1) children are defined as $c_2 = n_1 \cdot 2$ and $c_3 = n_1 \cdot 2 + 1$ Parent knots that therefor shown as $\lfloor c/2 \rfloor$

- Operations
  - MAX-HEAPIFY:
    For a know with two heap-shaped trees underneath it. Make the whole tree heap-shaped
    The parent switches value with the strongest child, if weaker than that child.
  - BUILD-HEAP:
    Arrange element in a heap-shape, then sort by heap-order
    Note: A single knot is both heap-shaped and in heap-order

2.7 Lower bound for comparator based sorting

Lower bound for ALL sorting-algorithms. **Note**: Does require an exact definition of a sorting-algorithm.
Definition:
All elements are comparable
For some algorithms we annotate each element along with its original position such as:
$F,A,C,B,E,D \rightarrow (F,1),(A,2),(C,3),(B,4),(E,5),(D,6) \rightarrow_{sort} (A,2),(B,4),(C,3),(D,6),(E,5),(F,1)$
A tree with the height of h has at most 2^h leaves.

$$2^h \geqslant \text{ antal blade} \geqslant n!$$
$$h \geqslant log(n!) = log(1 \cdot 2 \cdot 3 \cdot \cdots \cdot n)$$
$$\text{Since we know that: } log(n!) = log(1) + log(2) + \cdots + log\left(\frac{n}{2}\right) \cdot \cdots log(n)$$
$$log(n!) \geqslant \frac{n}{2} \cdot log\left(\frac{n}{2}\right) = \frac{n}{2} \cdot (log(n) - 1)$$

Therefore worst-case time complexity $= h = \Omega(n \cdot log(n))$

2.8 Datastructures

Goals for datastructure: flexibility and efficiency

- Datastructure level 1

  the operations offered (in java: an interface)

- Datastructure level 2

  specific implementation of the operations offered

2.8.1 Priority queues

Where Q is the priority queue
The central operations are:

- Q.EXTRACT-MAX: returns the max value of Q and removes it from Q
- Q.EXTRACT-MIN: returns the min value of Q and removes it from Q
- Q.INSERT($e : element$): adds the element e to the queue
- Q.INCREASE-KEY($r : index, k : key$): change an element in Q
- Q.DECREASE-KEY($r : index, k : key$): change an element in Q
- Q.BUILD($l : List$): Builds an priority queue from list L
- Also: Q.CREATENEWEMPTY(), Q.REMOVEEMPTY() and Q..ISEMPTY()

2.8.2 Time complexity for heap operations

	Heap	*Usorteret liste*	*Sorteret liste*
EXTRACT-MAX	$O(\log n)$	$O(n)$	$O(1)$
BUILD	$O(n)$	$O(1)$	$O(n \log n)$
INCREASE-KEY	$O(\log n)$	$O(1)$	$O(n)$
INSERT	$O(\log n)$	$O(1)$	$O(n)$

Figure 6: time complexities of heap-operations

2.8.3 Dictionaries

- Search(key): return key element (or tell if not found)
- Insert(key): Insert new element on key
- Delete(key): Delete element on key
- Predecessor(key): find element before key (largest key $<$ key)
- Successor(key): find element after key (smallest key $>$ key)
- OrderedTraversal(): prints ordered elements

For Predecessor(key), Successor(key) and OrderedTraversal() keys must be ordered.

2.8.4 Binary Tree

For a binary tree, if left side children $\leqslant$ parent and right side children are $\geqslant$, it is inorder.

- **INORDER-TREE-WALK**(x)

  Print a tree,

  Time complexity: $O(n)$

Algorithm 9 INORDER-TREE-WALK(x)

```
1: if x ≠ NIL then
2:     INORDER-TREE-WALK(x.left)
3:     print(key[x])
4:     INORDER-TREE-WALK(x.right)
```

- **TREE-SEARCH**(x)

  Pricipal: if x exists, it is under me.

  If i can reach the buttom without finding x, it does not exist.

Algorithm 10 TREE-SEARCH(x)

```
1: if x = NIL or k = key[x] then
2:     return x
3: if k < x.key then
4:     return TREE-SEARCH(x.left, k)
5: else
6:     return TREE-SEARCH(x.right, k)
```

- **TREE-MAXIMUM**(x) **and TREE-MINIMUM**(x)

  Both the following operations have a time complexity of O(h), where h is the height of the tree.

Algorithm 11 TREE-MAXIMUM(x)

```
1: while x.right ≠ NIL do
2:     x = x.right
3: return x
```

TREE-MAXIMUM(x) finds the largest key in a binary search tree by following the right child pointers until reaching a node with no right child.

Algorithm 12 TREE-MINIMUM(x)

```
1: while x.left ≠ NIL do
2:     x = x.left
3: return x
```

TREE-MINIMUM(x) finds the smallest key by following the left child pointers until reaching a node with no left child.

- **TREE-SUCCESSOR**(x)

Algorithm 13 TREE-SUCCESOR(x)

```
1: if x.right ≠ NIL then
2:     return TREE-MINIMUM(x.right)
3: y = x.parrent
4: while y ≠ NIL and x = y.right do
5:     x = y
6:     y = y.parent
7: return y
```

TREE-SUCCESOR finds the node with the smallest key larger than x. If x has a right subtree, the successor is the minimum node in that subtree; otherwise, it is the lowest ancestor of x whose left child is also an ancestor of x.

- **TREE-PREDECESSOR**(x)

Algorithm 14 TREE-PREDECESSOR(x)

```
1: if x.left ≠ NIL then
2:     return TREE-MAXIMUM(x.left)
3: y = x.parrent
4: while y ≠ NIL and x = y.left do
5:     x = y
6:     y = y.parent
7: return y
```

TREE-PREDECESSOR finds the node with the largest key smaller than x. If x has a left subtree, the predecessor is the maximum node in that subtree; otherwise, it is the lowest ancestor of x whose right child is also an ancestor of x.

2.8.5 Deleting nodes in binary tree

If one of Zs children is "null" redirect Zs parent to Zs child.
Else replace Z with its successor.

2.8.6 Red-black trees

For a red-black tree:

- Every node is either red or black.
- The root is black.
- If a node is red, then both its children are black.
- For each node, all simple paths from the node to descendant leaves contain the same number of black nodes.

2.8.7 Balancing a red-black tree

- Uncle is **red**

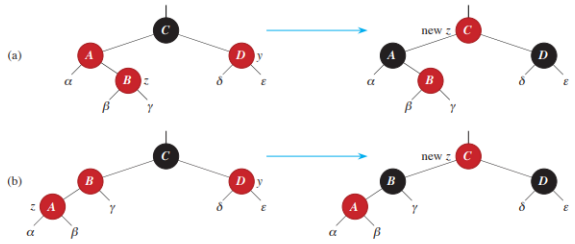


Figure 7: Rebalancing a red-black tree when uncle is red

- Uncle is **Black**

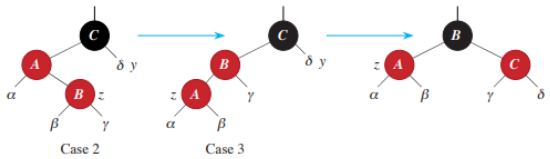


Figure 8: Rebalancing a red-black tree when uncle is black

2.8.8 Balancing after deletion in a red-black tree

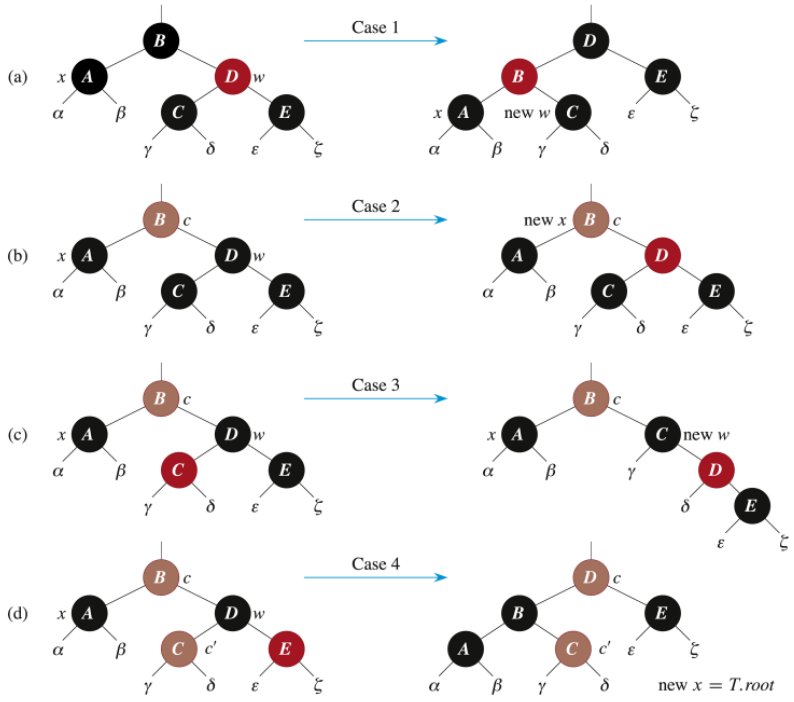


Figure 9: Re-balancing a red-black tree after deletion

2.8.9 Hashing

Use hashfunction to store elements. By doing modulus on the hashcodes we can store the elements using less space.
When duplicates occur, we can linklist duplicates in the stored space.(*Chaining*)
Another way is move duplicate to next empty space.

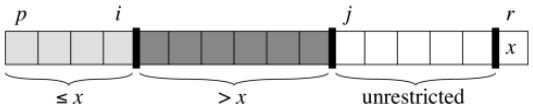


Figure 10: For Quicksort: $[p:i] \leqslant [x] < [j:r]$

2.9 Invariants

A relationship maintained by an algorithm throughout its execution. Often forms the core of the idea behind the algorithm.

2.9.1 Using induction to prove a invariant (*korrekthed*)

Prove that invariant hold in base case → prove for the all the next steps in the algorithm.
Invariant levels:

Level 1
1. Idea
2. Drawing

Level 2
1. Pseudocode
2. Code

2.9.2 Recursion equaltions for time complexity

$$T(n) = \begin{cases} a \cdot T(n/b) + \Theta(f(n)) & \text{for } n > 1 \\ \Theta(1) & \text{for } n \leqslant 1 \end{cases}$$

The following are good to remember

$$1 + c + c^2 + c^3 + \cdots + c^k = \frac{c^{k+1} - 1}{c - 1} = c^k \cdot \frac{c - 1/c^k}{c - 1} = \Theta(c^k)$$

$$(a^b)^c = a^{bc} = (a^c)^b$$

$$a^{log_b(n)} = n^{log_b(a)}$$

$$log_b(a) = \frac{log_c(a)}{log_c(b)}$$

2.9.3 The Master Theorem

The recurtion equation $T(n) = a \cdot T(n/b) + f(n)$ has the three following solutions.

1. If $f(n) = O(n^{log_b(a)-\varepsilon})$, where $\varepsilon > 0$ it holds that $T(n) = \Theta(n^{log_b(a)})$. (less than)
2. If $f(n) = \Theta(n^{log_b(a)} \cdot log(n)^k)$, where $k \geqslant 0$ it holds that $T(n) = \Theta(n^{log_b(a)} \cdot log(n)^{k+1})$. (equal)
3. If $f(n) = \Omega(n^{log_b(a)+\varepsilon})$, where $\varepsilon > 0$ it holds that $T(n) = \Omega(f(n))$. (greater than)

Note: Case is determined by the ratio of $f(n)$ and $n^{log_b(a)} \cdot log(n)^k$, where $k \geqslant 0$

3 Exercises

Apart from certain exercises, exercises are copied directly from the exercise sheets. Therefore A-exercises are exercises from the lesson and B-exercises are the one due before a lesson. Some weeks have two lessons therefore I and II.

3.1 Opgaver uge 7

3.1.1 A. 1

Exercise 1-1: Comparison of running times
For each function f (n) and time t in the following table, determine the largest size n of a problem that can be solved in time t, assuming that the algorithm to solve the problem takes f (n) nanoseconds $\left(10^{-9}\right)$.

(n)	1 second	1 hour	year	century
n	$1 \cdot 10^9$	$3.60 \cdot 10^{12}$		
$n \cdot log(n)$	$3.96 \cdot 10^7$	$8.69 \cdot 10^{10}$		
n^2	$3.16 \cdot 10^4$	$1.897 \cdot 10^6$		
n^3	$1.0 \cdot 10^3$	$1.53 \cdot 10^4$		
2^n	$2.99 \cdot 10^1$	$4.17 \cdot 10^1$		

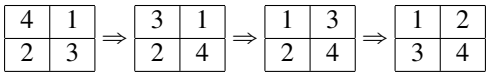
3.1.2 Brodals puslespil: Opgave 1

SShow that if all the pieces are not in their correct places at the start, then the minimum is required $n/2$ swiches to solve the puzzle.

Note that one swich can place two pieces in their correct place. Therfore if lucky you only need to swich $n/2$ times

3.1.3 Brodals pulsepsil: Opgave 2

State a puzzle with 4 pieces and an optimal sequence of swiches for the given puzzle (a sequence of swiches containing the least possible number of swiches), but where not all exchanges move at least one piece to the correct place.



```
import java.util.ArrayList;
import java.util.Collections;
import java.util.Scanner;
public class puzzle {
    /**
     * Takes n as an input and calls perm(n)
     * @param args
     */
    public static void main(String[] args) {
        try (Scanner sc = new Scanner(System.in)) {
            System.out.print("Please input an interger as n: ");
            int n = sc.nextInt();

            System.out.println(perm(n));
        }
    }

    /**
     * Returns a random permutation from
     * @param n
     * @return
     */
    public static ArrayList<Integer> perm(int n) {
        ArrayList<Integer> myList = new ArrayList<Integer>();
        for (int i = 0; i < n; i++) {
            myList.add(i);
        }
        Collections.shuffle(myList);
        return myList;
    }
}
```

OUTPUT

```
Please input an interger as n: 5
[4, 0, 1, 3, 2]
```

3.1.4 B. 1

Fill the coloums year and centry.

(n)	1 second	1 hour	year	century
n	$1 \cdot 10^9$	$3.60 \cdot 10^{12}$	$3.15 \cdot 10^{16}$	$3.15 \cdot 10^{18}$
$n \cdot log(n)$	$3.96 \cdot 10^7$	$8.69 \cdot 10^{10}$	$6.41 \cdot 10^{14}$	$5.67 \cdot 10^{16}$
n^2	$3.16 \cdot 10^4$	$1.897 \cdot 10^6$	$1.78 \cdot 10^8$	$1.78 \cdot 10^9$
n^3	$1.0 \cdot 10^3$	$1.53 \cdot 10^4$	$3.16 \cdot 10^5$	$1.47 \cdot 10^6$
2^n	$2.99 \cdot 10^1$	$4.17 \cdot 10^1$	$5.48 \cdot 10^1$	61.5

3.1.5 B. 2

3.2 Opgaver Uge 8

3.2.1 IA. 1

Consider linear search again (see Exercise 2.1-4). How many elements of the input array need to be checked on the average, assuming that the element being searched for is equally likely to be any element in the array? How about in the worst case? Using Θ-notation, give the average-case and worst-case running times of linear search. Justify your answers.

worst case is $\Theta(n)$
best case is $\Theta(1)$
average case is $\Theta(n)$

3.2.2 IA. 2

Using Figure 2.2 as a model, illustrate the operation of INSERTIONSORT on an array initially containing the sequence $\langle 31, 41, 59, 26, 41, 58 \rangle$. Answer:

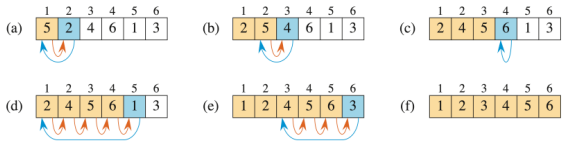


Figure 11: Model 2.2: Insertionsort

31	41	59	26	41	58
26	31	41	59	41	58
26	31	41	41	59	58
26	31	41	41	58	59

3.2.3 IA. 3

Implement insertion sort in java

```
    public class InsertionSort {
    public static void main(String[] args) {
        int[] arr = {5, 2, 4, 6, 1, 3};
        insertionSort(arr);
    }

    public static int[] insertionSort(int[] arr) {
        for (int i = 1; i < arr.length; i++) {
            // define key as the current element i.
            int key = arr[i];

            // define j as the element before i.
            int j = i - 1;
            while (j > 0 && arr[j] > key) {
                // move the element to the right.
                arr[j+1] = arr[j];
                j = j - 1;
            }
            // insert the key at the right position.
            arr[j+1] = key;
        }
        return arr;
    }
}
```

3.2.4 IA. 4

Let $A[1:n]$ be an array of n distinct numbers. If $i < j$ and $A[i] > A[j]$, then the pair (i, j) is called an inversion of A.

a. List the five inversions of the array $< 2,3,8,6,1 >$.

   1. $(1,5)$ 2. $(2,5)$ 3. $(3,4)$ 4. $(3,5)$ 5. $(4,5)$

b. What array with elements from the set $< 1,2,3\ldots,n$ has the most inversions? How many does it have?

$$\sum_{i=0}^{n-1} i = \frac{(n-1)n}{2}$$

c. What is the relationship between the running time of insertion sort and the number of inversions in the input array? Justify your answer.

   For every inversion, insertion moves the key once. to Run time of insertion sort = the number of inversions in the input array.

3.2.5 IB. 2

Describe an algorithm that, given a set S of n integers and another integer x, determines whether S contains two elements that sum to exactly x. Your algorithm should take $\Theta(log_2(n)n)$ time in the worst case. **Pseudocode:**

Algorithm 15 findSumX($A[\,],x$)

```
1: for i = 0 to A[].length − 1 do
2:     key ← A[i]
3:     for j = i + 1 to A[].length − 1 do
4:         if key + A[j] = x then return true
5: return false
```

Java:

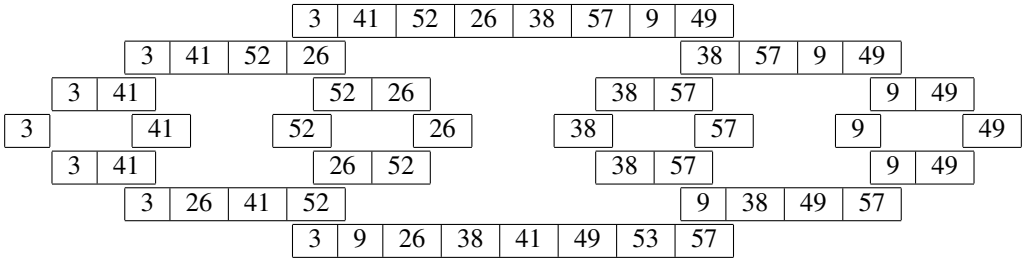
```java
public class sumX {
    public static void main(String[] args) {
        int[] arr = {5, 2, 4, 6, 1, 3};
        int x = 11;
        System.out.println(findSumX(arr,x));
    }

    public static boolean findSumX(int[] arr, int x) {
        for (int i = 0; i < arr.length; i++) {
            // define key as the current element i.
            int key = arr[i];

            // check next elements in list.
            for (int j = i + 1; j < arr.length; j++) {
                // check if elements sum to x.
                if (key + arr[j] == x) {
                    return true;
                }
            }
        }
        return false;
    }
}
```

OUTPUT:

```
TRUE
```

3.2.6 IIA 1

Using Figure 2.4 as a model, illustrate the operation of merge sort on an array initially containing the sequence $< 3,41,52,26,38,57,9,49 >$.



3.2.7 IIA 2

Show for

$$f(n) = 0.1n^2 + 5n + 25$$

that $f(n) = \Theta(n^2)$ and $f(n) = o(n^3)$

$$\frac{0.1n^2+5n+25}{n^2} = \frac{0.1+5/n+5/n^2}{1} = \frac{0.1+0+0}{1} = 20 \text{ for } n \to \infty$$

$$\frac{0.1n^2+5n+25}{n^3} = \frac{0.1/n+5/n^2+25/n^3}{1} = \frac{0+0+0}{1} = 0 \text{ for } n \to \infty$$

3.2.8 IIA 3

Show that the following functions are written in growing asymptotic growth rate. These are written in the form

$$\frac{f(n)}{g(n)} \to 0 \text{ for n} \to \infty$$

1) $\frac{1}{log\ n} \to 0$ for n $\to \infty$

$log\ n$) $\frac{log\ n}{\sqrt{n}} = \frac{log\ n}{n^{\frac{1}{2}}} \to 0$ for n $\to \infty$

$\sqrt{n}$) $\frac{\sqrt{n}}{n} = \frac{n^{\frac{1}{2}}}{n} \to 0$ for n $\to \infty$

n) $\frac{n}{n \cdot log\ n} = \frac{1}{log\ n} \to 0$ for n $\to \infty$

$n \cdot log\ n$) $\frac{n \cdot log\ n}{n \cdot \sqrt{n}} = \frac{log\ n}{n^{\frac{1}{2}}} \to 0$ for n $\to \infty$

$n \cdot \sqrt{n}$) $\frac{n \cdot \sqrt{n}}{n^2} = \frac{n^{\frac{1}{2}}}{n} \to 0$ for n $\to \infty$

n^2) $\frac{n^2}{n^3} = \frac{n}{n^2} \to 0$ for n $\to \infty$

n^3) $\frac{n^3}{n^{10}} = \frac{1}{n^7} \to 0$ for n $\to \infty$

n^{10}) $\frac{n^{10}}{2^n} \to 0$ for n $\to \infty$

2^n) $\frac{b^n}{2^n} \to 0$ for n $\to \infty$

3.2.9 IIB 1

From Exercise 2.3-6 (p.44) from the book Referring back to the searching problem (linear seach) . First draw the algrothm, then write pseudocode, either iterative or recursive, for binary search. Then write it in python or java. Argue that the worst-case running time of binary search is $\Theta(log_2 n)$.

Choose and $x = 4$
And choose array $A = \{1,2,3,4,5,6,7,8\}$
Now find x in $A[]$ using the binary search.
Note: For binary search $A[]$ must be a sorted list, and round up if even length
Notation: start (p), middle (r) and end (q)

p ↓				r ↓			q ↓
1	2	3	4	5	6	7	8
p ↓		r ↓		q ↓			
1	2	3	4	5			
		p ↓	r ↓	q ↓			
		3	4	5			
			x ↓				
			4				

Table 1: Finding x = 4 in array using binary search

Pseudocode:

Algorithm 16 BinSearch($A[\,],x$)

```
1:  low = 0                                   ▷ A[] is a sorted array and x is an int
2:  high = A.length − 1
3:  while low ≠ high do
4:      mid = ⌈(high − low)/2⌉ − 1
5:      if A[mid] = x then return mid
6:      else if A[mid] < x then
7:          low = mid
8:      else
9:          high = mid
10: return NOT FOUND
```

```java
public class BinSearch {
    public static void main(String[] args) {
        int[] arr = {1, 2, 3, 4, 5, 6};
        int x = 4;
        System.out.println(binSearch(arr, x));
    }

    public static int binSearch(int[] arr, int x) {
        int low = 0;
        int high = arr.length - 1;
        while (low != high) {
            int mid = (int) Math.ceil((low + high) / 2);
            if (arr[mid] == x) {
                return mid;
            } else if (arr[mid] < x) {
                low = mid;
            } else {
                high = mid;
            }
        }
        return -1;
    }
}
```

3.2.10 IIB 2

The while loop of lines 5-7 of the INSERTION-SORT procedure in Section 2.1(book p.30) uses a linear search to scan (backward) through the sorted subarray $A[1 : j-1]$. What if insertion sort used a binary search (see Exercise 2.3-6) instead of a linear search? Would that improve the overall worst-case running time of insertion sort to $\Theta(n \cdot log\ n)$?

- It could be fasterm, yet worst case is still. $\Theta(n^n)$

3.2.11 IIB 3

Determind these statemends as true or false?

- $2^{n+1} = O(2^n)$?

  As $\frac{2^{n+1}}{2^n} = 2$ when $n \to \infty$ Therefore Θ

  This statement true.

- $2^{2n} = O(2^n)$?

  We try and prove the following $\frac{2^n}{2^{2n}} = \frac{2^n}{2^n \cdot 2^n} = \frac{1}{2^n}$.

  Therefore the statement is false.

3.2.12 IIB 4

Prove the following

- Equation 3.21 ($a^{log_b\ c} = c^{log_b\ a}$) This proof uses the change of base rule $log_b\ x = \frac{log_a\ x}{log_a\ b}$

$$\begin{aligned}
\text{Let: } x &= a^{log_b\ c} \\
log\ x &= log(a^{log_b\ c}) \\
log\ x &= log\ a \cdot log_b\ c \\
\text{Because: } log_b\ c &= \frac{log\ c}{log\ b} \\
log\ x &= log\ a \cdot \frac{log\ c}{log\ b} \\
\text{Now switch a and c: } log\ x &= log\ c \cdot \frac{log\ a}{log\ b} \\
\text{Now go back to base b: } log\ x &= log\ c \cdot log_b\ a \\
log\ x &= log(c^{log_b\ a}) \\
\text{Substitute x from earlier: } log\ a^{log_b\ c} &= log(c^{log_b\ a}) \\
\text{Therefore: } a^{log_b\ c} &= c^{log_b\ a}
\end{aligned}$$

- Equation (3.26)-(3.28)
  - Equation 3.26 ($n! = o(n^n)$) REWRITE

$$\begin{aligned}
\prod_{i=1}^{n} i &< \prod_{i=1}^{n} n \\
(1 \cdot 2 \cdot 3 \cdots n) &< (n \cdot n \cdot n \cdots n) \\
\text{Therefore } n! &= o(n^n)
\end{aligned}$$

  - Equation 3.27 ($n! = \omega(2^n)$) REWRITE

    We must find both $log(n!) = O(n \cdot log\ n)$ and $log(n!) = \Omega(n \cdot log\ n)$

$$\begin{aligned}
\prod_{i=1}^{n} i &< \prod_{i=1}^{n} 2 \\
(1 \cdot 2 \cdot 3 \cdots n) &> (2 \cdot 2 \cdot 2 \cdots 2) \text{ n times} \\
\text{Therefore for n} > 1\text{, it holds that } n! &> 2^n
\end{aligned}$$

- Equation 3.28 ($log_2(n!) = \Theta(n \cdot log_2\ n)$)
  - $log(n!) = O(n \cdot log(n)$

$$\begin{aligned} log(n!) &= log(n) + log(n-1)0 \cdots + log(2) + log(1) \\ &\leqslant log(n) + log(n) \cdots + log(n) + log(n) \\ log(n!) &\leqslant n \cdot log(n) \end{aligned}$$

  - $log(n!) = \Omega(n \cdot log(n)$

For this note that $log\left(\frac{n}{2}\right) \geqslant \frac{1}{2} \cdot log(n)$ (1)

$$\begin{aligned} log(n!) &= log(n) + log(n-1) + \ldots log\left(\frac{n}{2}\right) + \cdots + log(1) \\ &\geqslant log(n) + log(n-1) + \ldots log\left(\frac{n}{2}\right) \\ &\geqslant log\left(\frac{n}{2}\right) + log\left(\frac{n}{2}\right) + \ldots log\left(\frac{n}{2}\right) \\ &\geqslant \frac{n}{2} \cdot log\left(\frac{n}{2}\right) \\ &\geqslant \frac{n}{2} \cdot \frac{1}{2} \cdot log\,(n) \qquad (1) \\ &\geqslant \frac{1}{4} \cdot n \cdot log\,(n) \end{aligned}$$

Therefore: $log_2(n!) = \Omega(n \cdot log_2\ n)$

When both $f = O(g)$ and $f = \Omega(g)$ is true, this means that $f = \Theta(g)$

3.3 Opgaver Uge 9

3.3.1 A. 1

Write the asymptotic order of the following functions.

$$\sqrt{n}, 2^n, (log_{10}(n))^2, n, log(n)$$

Using the rule of base change we get that

$$\frac{log_2(n)}{(log_{10}(n))^2}$$

$$\text{Base change rule: } \frac{\frac{log_{10}(n)}{log_{10}(2)}}{(log_{10}(n))^2}$$

$$\text{This results to: } \frac{1}{log_{10}(n) \cdot log_{10}(2)}$$

This means that to:

$$log_2(n) > (log_{10}(n)) < \sqrt{n} < n < 2^n$$

3.3.2 A. 2

Which of the following statements are true?

a) $2n+5 = \Theta(n)$

Remove constants. So true

b) $n = o(n^2)$

n will always grow less than n^2

c) $n = O(n^2)$

because of b) this is true

d) $n = \Theta(n^2)$

because of b) this must be false

e) $n = \Omega(n^2)$

because of b) this must be false

f) $n = \omega(n^2)$

because of b) this must be false

g) $\sqrt{n}\ log(n) = O(\sqrt{n})$

no constant can hold up to $log(n)$, so false

h) $\sqrt{n}\ log(n) = \omega(\sqrt{n})$

no constant can hold up to $log(n)$, so true

i) $\sqrt{n} + log(n) = O(\sqrt{n})$

since $\sqrt{n}$ is bigger $log(n)$ than , so true

j) $(log(n))^3 = O(n \cdot log(n))$

Any logarithm no matter the exponent is worse than a polynomial.

3.3.3 A. 3

Given that. $f_1(n) \in O(g_1(n))\ f_2(n) \in O(g_2(n))$ Which of the following statements are true

a) $f_1(n) + f_2(n) \in O(g_1(n) + g_2(n))$

$f_1(n) + f_2(n) \leqslant c_1 \cdot g_1(n) + c_2 \cdot g_2(n)) \leqslant max(c_1, c_2) \cdot (g_1(n) + g_2(n)) \mid \forall n \in max\{\aleph_0, \aleph_0\}$

This statement is true

b) $g_1(n) + g_2(n) \in \Omega(f_1(n) + f_2(n))$

Bacause of a) we know that $f(n) = O(g(n)) \iff g(n) = \Omega(f(n))$

c) $\frac{f_1(n)}{f_2(n)} = O\left(\frac{g_1(n)}{g_2(n)}\right)$

Counter example $\frac{n^2}{1} = O(\frac{n^2}{n}) \iff n^2 = O(n)$

3.3.4 A. 4

Write the time complexity for the following two algorthms

Algorithm 17 "Algorithm 1(n)"

```
1: s = 0
2: for i = 1 do ⌊n/2⌋
3:     for j = i do n − i
4:         s = s + 1
5: return s
```

- First loop is approximately $n/2$ so $O(n)$
- In worst case j = i, therfore forloop with j must also be $O(n)$
- In total $O(n^2)$

Algorithm 18 "Algorithm 2(n)"

```
1: s = 0
2: for i = 1 do n
3:     for j = 1 do n
4:         for k = 1 do n
5:             s = s + 1
6: return s
```

- Going trough 3 loops, so n^3

3.3.5 B. 1

Which of the following statements are true?

1. $n^2 \in \Omega(n)$

   Yes. Since $n^2 \geqslant C \cdot n$

2. $n \in \Theta(n^2)$

   No. Since for no constants C_1, C_2 it holds that $C_1 \leqslant \frac{n}{n^2} \leqslant C_2$

3. $n \cdot log(n) \in o(n^2)$

   Yes. Since $\frac{n \cdot log(n)}{n^2} = \frac{log(n)}{n} = 0$ as $n \to \infty$

4. $log(n) \in O(\sqrt{n})$

   Yes. Since $\frac{log(n)}{\sqrt{n}} = 0$ as $n \to \infty$

5. $n^2 \in \omega(n)$

   Yes. Since $\frac{n!}{2^n} = \infty$ as $n \to \infty$

3.3.6 B. 2

Write the time complexity for the following two algorithm.

Algorithm 19 "Algorithm 3(n)"

```
1: s = 0
2: for i = 1 to n do
3:     for j = 1 to n do
4:         if i = j then
5:             for k = 1 to n do
6:                 s = s + 1
7: return s
```

- This is $\Theta(n^2 + n^2)$
- Which is just $\Theta(n^2)$

Algorithm 20 "ALGORITHM3(n)"

```
1: s = 0
2: for i = 1 to n do
3:     for j = 1 to n do
4:         if i ≠ j then
5:             for k = 1 to n do
6:                 s = s + 1
7: return s
```

- This is $\Theta(n^2 + (n^2 - n) \cdot n) = \Theta(n^2 + n^3 - n^2)$
- Which is $\Theta(n^3)$

3.3.7 B. 3

Continue the task above with the implementation of MERGESORT.

```
    import java.util.Random;
public class MergeSort {
    public static void main(String[] args) {
        int n = 0;
        int[] arr = arrGen(n);
        long startTime = System.currentTimeMillis();
        arr = mergeSort(arr);
        long endTime = System.currentTimeMillis();
        //for (int i : arr) {
        //    System.out.println(i);
        //}
        System.out.println("arr lenth: " + n);
        System.out.println("time complexity: " + (endTime - startTime));
    }

    public static int[] mergeSort(int[] arr) {
        if (arr.length < 2) {
            return arr;
        }

        int mid = arr.length / 2;
        int[] left = new int[mid]; // 0 to mid
        int[] right = new int[arr.length - mid]; // mid to arr.length

        for (int i = 0; i < mid; i++) { // make new array from 0 to mid
            left[i] = arr[i];
        }

        for (int i = 0; i < arr.length - mid; i++) { // make new array from mid to arr.length
            right[i] = arr[i + mid];
        }

        left = mergeSort(left); // recursively sort left
        right = mergeSort(right); // recursively sort right

        return merge(left, right);
    }
```

```
    public static int[] merge(int[] left, int[] right) {
        int[] result = new int[left.length + right.length];
        int i = 0, j = 0;

        while (i < left.length && j < right.length) {
            if (left[i] <= right[j]) {
                result[i+j] = left[i];
                i++;
            }
            else {
                result[i+j] = right[j];
                j++;
            }
        }

        while (i < left.length) {
            result[i+j] = left[i];
            i++;
        }

        while (j < right.length) {
            result[i+j] = right[j];
            j++;
        }
        return result;
    }

    public static int[] arrGen(int n) {
        int[] result = new int[n];
        Random random = new Random();
            for (int i = 0; i < n; i++) {
                result[i] = random.nextInt(10);
            }
        return result;
    }
}
```

TESTS

```
arr lenth: 1.000.000
AVG: 112.67
```

$$\frac{112.67}{10 \cdot 10^5 \cdot log_2(10 \cdot 10^5)} = 5.65 \cdot 10^{-6}$$

```
arr lenth: 10.000.000
AVG: 766.67
```

$$\frac{766.67}{10 \cdot 10^6 \cdot log_2(10 \cdot 10^6)} = 3.30 \cdot 10^{-6}$$

```
arr lenth: 25.000.000
AVG: 1951.33
```

$$\frac{1951.33}{2.5 \cdot 10^7 \cdot log_2(2.5 \cdot 10^7)} = 3.18 \cdot 10^{-6}$$

```
arr lenth: 50.000.000
AVG: 3765.67
```

$$\frac{3765.67}{5.0 \cdot 10^7 \cdot log_2(5.0 \cdot 10^7)} = 2.94 \cdot 10^{-6}$$

```
arr lenth: 65.000.000
AVG: 5060.33
```

$$\frac{5060.33}{6.5 \cdot 10^7 \cdot log_2(6.5 \cdot 10^7)} = 2.99 \cdot 10^{-6}$$

3.3.8 B. 4

Let $A[1:n]$ be an array of n distinct numbers. If $i < j$ and $A[i] > A[j]$, then the pair (i, j) is called an inversion of A.

d Give an algorithm that determines the number of inversions in any permutation on n elements in $\Theta(n \cdot log(n))$ worst-case time. (Hint: Modify merge sort.)

3.4 Opgaver Uge 10

3.4.1 IA. 1

Exercises 7.1-1 (p.186) from book Using Figure 7.1 as a model, illustrate the operation of PARTITION on the array $A = [13, 19, 9, 5, 12, 8, 7, 4, 21, 2, 6, 11]$.

13	19	9	5	12	8	7	4	21	2	6	11
13	19	9	5	12	8	7	4	21	2	6	11
13	19	9	5	12	8	7	4	21	2	6	11
9	19	13	5	12	8	7	4	21	2	6	11
9	5	13	19	12	8	7	4	21	2	6	11
9	5	13	19	12	8	7	4	21	2	6	11
9	5	8	19	12	13	7	4	21	2	6	11
9	5	8	7	12	13	19	4	21	2	6	11
9	5	8	7	4	13	19	12	21	2	6	11
9	5	8	7	4	2	19	12	21	13	6	11
9	5	8	7	4	2	6	12	21	13	19	11
9	5	8	7	4	2	6	11	21	13	19	12

Figure 12: excel bad

3.4.2 IA. 2

Exercise 7.1.2 (p187) from book:
What value of q does PARTITION return when all elements in the subarray $A[i : j]$ have the same value?
This would be equivalent to the last element of A

Modify PARTITION so that $q = \lfloor (p+r)/2 \rfloor$ when all elements in the subarray $A[p : r]$ have the same value.

Algorithm 21 PARTITION(A, p, r)

```
1:  x = A[r]                                   ▷ the pivot
2:  p − 1                                      ▷ highest index into the low side
3:  for j = p to r − 1 do                      ▷ process each element other than the pivot
4:      if A[j] ⩽ x then                       ▷ does this element belong on the low side?
5:          i = i + 1                          ▷ index of a new slot in the low side
6:          exchange A[i] with A[j]            ▷ put this element there
7:  exchange A[i + 1] with A[r]                ▷ pivot goes just to the right of the low side
8:  return i + 1                               ▷ new index of the pivot
```

Algorithm 22 ALTERED FOR EXERCISE PARTITION(A, p, r)

```
1:  x = A[r]                                   ▷ the pivot
2:  p − 1                                      ▷ highest index into the low side
3:  c = 0                                      ▷ counter for later
4:  for j = p to r − 1 do                      ▷ process each element other than the pivot
5:      if A[j] < x then                       ▷ does this element belong on the low side?
6:          i = i + 1                          ▷ index of a new slot in the low side
7:          exchange A[i] with A[j]            ▷ put this element there
8:      else if A[j] = x then                  ▷ checks if element is equal value to x
9:          c = c + 1                          ▷ counter switching from even/odd
10:         if (c mod 2) = 0 then              ▷ checks if c is even
11:             i = i + 1                      ▷ index of a new slot in the low side
12:             exchange A[i] with A[j]        ▷ corrects x position
13: exchange A[i + 1] with A[r]                ▷ pivot goes just to the right of the low side
14: return i + 1                               ▷ new index of the pivot
```

3.4.3 IA. 3

What is the running time of QUICKSORT when all elements of array A have the same value?
recursion n times with 1 for-loop mean that $\Theta(n^2)$

3.4.4 IA. 4

Give a brief argument that the running time of PARTITION on a subarray of size n is $\Theta(n)$
Only going through the list once per PARTITION

3.4.5 IA. 5

Enter the best-case and worst-case driving time as well as the driving time for sorted input for Insertion Sort, Merge Sort and Quicksort.

	Best	Worst	Sorted input
InsertionsSort	n	n^2	n
MergeSort	$n \cdot log(n)$	$n \cdot log(n)$	$n \cdot log(n)$
QuickSort	$n \cdot log(n)$	n^2	n^2

3.4.6 IA. 6

The PARTITION procedure returns an index q such that each element of $A[p : q-1]$ is less than or equal to $A[q]$ and each element of $q+1 : r$ is greater than $A[q]$. Modify the PARTITION procedure to produce a procedure PARTITION'(A, p, r), which permutes the elements of $A[p : r]$ and returns two indices q and t, where $p \leqslant q \leqslant t \leqslant r$, such that

- all elements of $A[q : t]$ are equal
- each element of $A[p : q-1]$ is less than $A[q]$, and
- each element of $A[t+1 : r]$ is greater than $A[q]$.

Like PARTITION, your PARTITION' procedure should take $\Theta(r-p)$ time.

Algorithm 23 PARTITION'(A, p, r)

```
1:  x = A[r]                                   ▷ the pivot
2:  i = p − 1                                  ▷ highest index into the low side
3:  t = p − 1
4:  for j = p to r − 1 do                      ▷ process each element other than the pivot
5:      if A[j] = x then                       ▷ does this element belong on the low side?
6:          t = t + 1                          ▷ index of a new slot in the low side
7:          exchange A[t] with A[j]            ▷ put this element there
8:      else if A[j] < x then
9:          i = i + 1
10:         exchange A[i] with A[j]
11:         exchange A[t] with A[j]
12: exchange A[t + 1] with A[r]                ▷ moves r to the = part
13: t = t + 1 return (i + 1, t)                ▷ new index of the pivot
```

3.4.7 IB.1

Implement Quicksort in Java. Test code like in Opgave Uge 9, B. 3.

```
arr lenth: 10000
AVG: 12
```

$$\frac{12}{10 \cdot 10^3 \cdot log_2(10 \cdot 10^3)} = 9.03 \cdot 10^{-5}$$

```
arr lenth: 25000
AVG: 79
```

$$\frac{79}{2.5 \cdot 10^4 \cdot log_2(2.5 \cdot 10^4)} = 2.16 \cdot 10^{-4}$$

```
arr lenth: 50000
AVG: 312
```

$$\frac{312}{5 \cdot 10^4 \cdot log_2(5 \cdot 10^4)} = 3.99 \cdot 10^{-4}$$

```
arr lenth: 75000
AVG: 685
```

$$\frac{685}{7.5 \cdot 10^4 \cdot log_2(7.5 \cdot 10^4)} = 5.64 \cdot 10^{-4}$$

```
arr lenth: 100000
AVG: 1247
```

$$\frac{1247}{10 \cdot 10^4 \cdot log_2(10 \cdot 10^4)} = 7.51 \cdot 10^{-4}$$

3.4.8 IIA. 1

Exercise 6.1-6 (p. 164) from book
Is an array that is in sorted order a min-heap?
Since $i-1$ is always less than i a sorted array must be min-heap

3.4.9 IIA. 2

Exercise 6.1-7 (p. 164) from book
Is the array with values $(33, 19, 20, 15, 13, 10, 2, 13, 16, 12)$ a max-heap?

> No since $16 < 15$
> For a max heap all parent $>$ child

3.4.10 IIA. 3

Exam from january 2008, 1.b
Insert the key 9 into the binary max-heap represented by the array below. Show the result before each iteration of the while loop in the algorithm on page 140 of the textbook. You may draw it as a tree structure instead of an array.

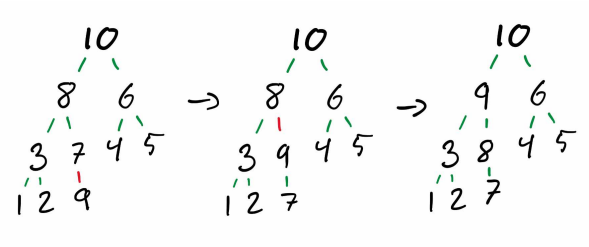


Figure 13: 9 added to a min heap

3.4.11 IIA. 4

Exam from january 2006, b
Consider the following binary heap:
Now an element with priority 2 is inserted. Draw the heap as it looks after this insertion.

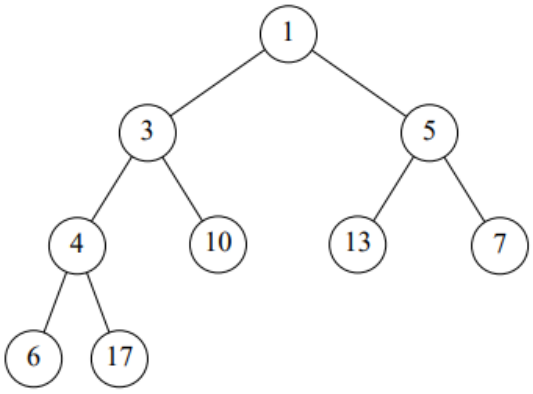


Figure 14: Tree from exercise

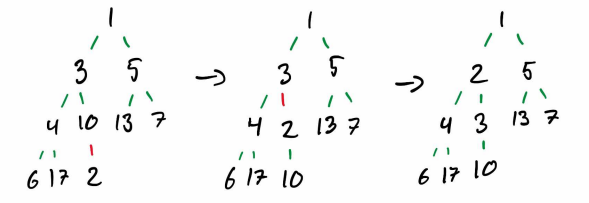


Figure 15: 2 added to a min heap

3.4.12 IIA. 5

Exercise 6.2-1 (p. 166) from book
Using Figure 6.2 as a model, illustrate the operation of MAX-HEAPIFY$(A, 3)$ on the array $A(27, 17, 3, 16, 13, 10, 1, 5, 7, 12, 4, 8, 9, 0)$.

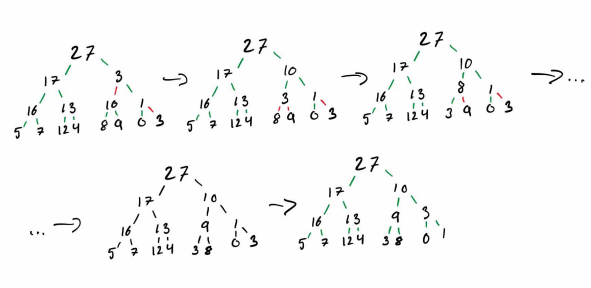


Figure 16: 3 added to a heap A

3.4.13 IIA. 6

Perform HEAP-EXTRACT-MAX(A) on max-heap A below.
$A(21, 18, 10, 12, 8, 9, 4, 7, 5, 2)$

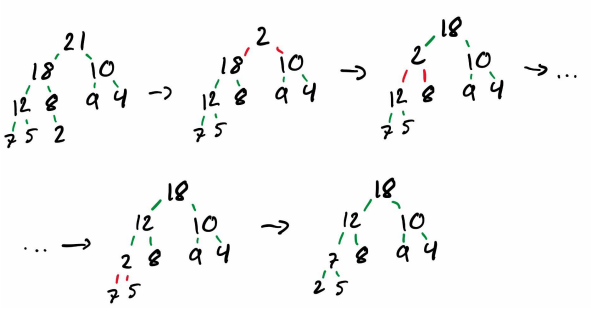


Figure 17: HEAP-EXTRACT-MAX(A)

3.4.14 IIA. 7

Exercise 6.1-4 (p. 164)
Where in a max-heap might the smallest element reside, assuming that all elements are distinct?

In a leaf

3.4.15 IIB. 1

Exam from june 2008, 4.a Draw all possible binary min-heaps that contain four nodes with priorities 1, 2, 3, and 4.

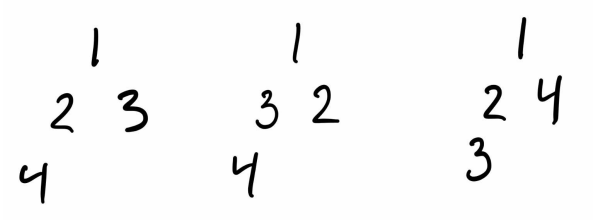


Figure 18: All possible min-heaps containing the priorities 1, 2, 3 and 4

3.4.16 IIB. 2

Exexercise 6.5-11 (p. 178)
Give an $O(n \cdot log(k))$-time algorithm to merge k sorted lists into one sorted list, where n is the total number of elements in all the input lists. (Hint: Use a minheap for k-way merging.)

Algorithm 24 K-SORTEDMERGE(A[k sorted lists])

```
1: tree = empty tree
2: result = empty list
3: for i = 0 to < k do
4:     tree.addNode(A[i[0]] and A[i.remove(0)]
5:
6:     result += ROOT(tree)
7: return result
```

3.4.17 IIB. 3

Problem 6.2 (p. 179)
A *d-ary* heap is like a binary heap, but (with one possible exception) nonleaf nodes have d children instead of two children. In all parts of this problem, assume that the time to maintain the mapping between objects and heap elements is O.1/ per operation.

a. Describe how to represent a d-ary heap in an array.

for a d-ary heap to an array

Algorithm 25 D-ARYHEAP($tree, d$)

```
1: arr[0] = root node
2: row = 0
3: for i = 1 to < height do
4:     row = row + 1
5:     for j = 1 to ⩽ d^row do
6:         A.add(node)
7:         i = i + 1
8: return tree
```

For array A as a d-ary heap. For each height $N(h) = d^h$ where N is amount of nodes in a layer, and h is the height of the tree.

This means that $A[0]$ is the very first node of the tree, now we can continue adding the nex d nodes as children to $A[0]$.

After this continue for each child as for $A[0]$. Continue this.

Algorithm 26 D-ARYHEAP

```
1: P(i) = ⌊i/2⌋
2: P(i) = 2i
3: P(i) = 2i + 1
4: Child(k,i) = i · d + k
5: Parent(k,i) = ⌊(i−1)/k⌋
```

b. Using Θ-notation, express the height of a *d-ary* heap of n elements in terms of n and d. For question b, use e.g. formula (A.5) page 1147 as well as inspiration from analysis on the height of slides in binary heaps.

$$
\begin{aligned}
n &= \frac{d^{h+1}-1}{d-1} \\
n \cdot d - 1 &= d^{h+1} - 1 \\
n \cdot d &= d^{h+1} \\
log(n\cdot)d &= log(d) \cdot (h+1) \\
\frac{log(n \cdot d)}{log(d)} &= h+1 \\
\frac{log(n \cdot d)}{log(d)} - 1 &= h \\
Log_d(nd) - 1 &= h
\end{aligned}
$$

c.,d. and e. are implemented in Java and/or Python

3.5 Opgaver Uge 11

3.5.1 A.1

Exercise 6.4-4 (p. 172) Show that the worst-case running time of HEAPSORT is $\Omega(n \cdot log(n))$
For HEAPSORT, we start with build heap. then heapify for each element. $n + n \cdot log(n)$

3.5.2 A.2

Exercise 8.2-1 (p. 210) Using Figure 8.2 as a model, illustrate the operation of COUNTINGSORT on the array $A = (6,0,2,0,1,3,4,6,1,3,2)$.

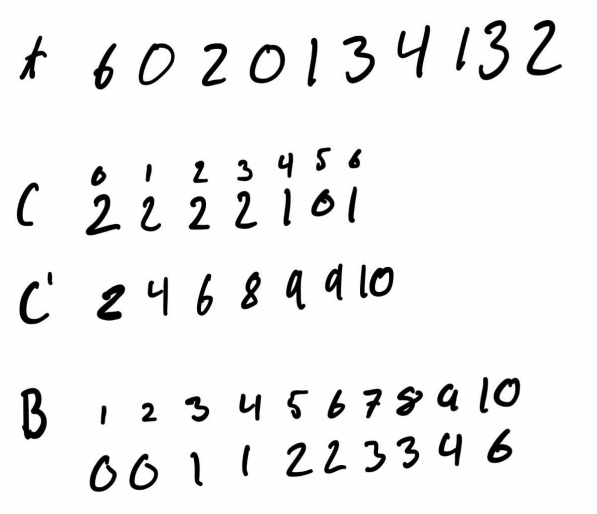


Figure 19: Counting sort illustrated on an array

3.5.3 A.3

Exercise 8.2-3 (p. 211) Suppose that we were to rewrite the for loop header in line 11 of the COUNTINGSORT as

11 **for** $j = 1$ **to** n

Show that the algorithm still works properly, but that it is not stable. Then rewrite the pseudocode for counting sort so that elements with the same value are written into the output array in order of increasing index and the algorithm is stable.

Algorithm 27 COUNTINGSORT(A, n, k)

```
1: let B[1 : n] and C[0 : k] be new arrays
2: for i = 0 to k do
3:     C[i] = 0
4: for j = 1 to n do
5:     C[A[j]] = C[A[j]] + 1                ▷ C[i] now contains the number of elements equal to i.
6: for j = 1 to k do
7:     C[i] = C[i] + C[i − 1]               ▷ C[i] now contains the number of elements less than or equal to i.
8: for j = 1 to n do
9:     B[C[A[j]]] = A[j]
10:    C[A[j]] = C[A[j]] − 1
11: return B
```

This will be unstable due to order. We can fix this with:

Algorithm 28 COUNTINGSORT(A, n, k)

```
1: let B[1 : n] and C[0 : k] be new arrays
2: for i = 0 to k do
3:     C[i] = 0
4: for j = 1 to n do
5:     C[A[j]] = C[A[j]] + 1                ▷ C[i] now contains the number of elements equal to i.
6: for i = k to 0 do
7:     C[i] = C[i + 1] − C[i]               ▷ C[i] now contains the number of elements greater than or equal to i.
8: for j = 1 to n do
9:     B[C[A[j]]] = A[j]
10:    C[A[j]] = C[A[[j] + 1
11: return B
```

3.5.4 A.4

Exercise 8.2-6 (p. 211) Describe an algorithm that, given n integers in the range 0 to k, preprocesses its input and then answers any query about how many of the n integers fall into a range $[a : b]$ in $O(1)$ time. Your algorithm should use $\Theta(n+k)$ preprocessing time.

Algorithm 29 PREPROCESS(A, n, k)

```
1: C = [0] · n                          ▷ makes an "empty" array from 0 to k
2: for i = 0 to n do
3:     C[A[i]] = C[A[i]] + 1            ▷ first steps of counting sort. Count elements
4: for j = 1 to n do                    ▷ add elements as in counting sort
5:     C[j] = C[j]] + C[j − 1]
6: return C
```

Algorithm 30 QUERY(C, a, b)

```
1: if a − 1 < 0 then                    ▷ ensure that a is not 0
2:     return C[b]
3: else
4:     return C[b] − C[a − 1]
```

3.5.5 A.5

Exercise 8.3-2 (p. 215) Which of the following sorting algorithms are stable: insertion sort, merge sort, heapsort, and quicksort?

- INSERTIONSORT is Stable
- MERGESORT is Stable
- HEAPSORT is Un-stable
- QUICKSORT is Un-stable

Give a simple scheme that makes any comparison sort stable.
How much additional time and space does your scheme entail?
Hint: extend the keys of the elements.

Assign an indext-list for for the list to be sorted, this way you can sort the indexes in the end with. The same sorting time again and another n space

3.5.6 A.6

Exam from June 2008, 1a
Perform radix sort 10 on the numbers: $(747, 765, 544, 754, 431, 231, 222)$

Algorithm 31 RADIX(A, n, d)

1: **for** $i = 1$ **to** d **do** ▷ go through all digits
2: Use a stable sort to sort A on digit i.

Show the result after each iteration

For each for iteration each numbers are sorted using counting sort, based on the underlined numbers. (e.g $22\underline{2}, 43\underline{1} \rightarrow 43\underline{1}, 22\underline{2}$)

74$\underline{7}$	76$\underline{5}$	54$\underline{4}$	75$\underline{4}$	43$\underline{1}$	23$\underline{1}$	22$\underline{2}$
43$\underline{1}$	23$\underline{1}$	22$\underline{2}$	54$\underline{4}$	75$\underline{4}$	76$\underline{5}$	74$\underline{7}$
4$\underline{3}$1	2$\underline{3}$1	2$\underline{2}$2	5$\underline{4}$4	7$\underline{5}$4	7$\underline{6}$5	7$\underline{4}$7
2$\underline{2}$2	4$\underline{3}$1	2$\underline{3}$1	5$\underline{4}$4	7$\underline{4}$7	7$\underline{5}$4	7$\underline{6}$5
$\underline{2}$22	$\underline{4}$31	$\underline{2}$31	$\underline{5}$44	$\underline{7}$47	$\underline{7}$54	$\underline{7}$65
$\underline{2}$22	$\underline{2}$31	$\underline{4}$31	$\underline{5}$44	$\underline{7}$47	$\underline{7}$54	$\underline{7}$65

3.5.7 A.7

Exercise 8.3-5 (p. 215)
Show how to sort n integers in the range 0 to $n^3 - 1$ in $O(n)$ time.
Hint: look at the number made from 3 digits $d_2d_1d_0$ in radix (*grundtal*) n. So. $x = d_2 \cdot n^2 + d_1 \cdot n^1 + d_0 \cdot n^0$, where $0 \leqslant d_i \leqslant n-1$ for $i = 0, 1, 2$. To refresh your knowledge of number systems, see pages 5–7 of these slides about number systems - our normal number system uses base 10, as you know.
By the way, for a number x, you can find these digits $d_2d_1d_0$ by using integer division and modulus (remainder of integer division) by n repeatedly, just like in the algorithm on pages 10–12 on the same slides (where radix/base n is equal to 2)).

We know that the largest possible integer is $n^3 - 1$ using $d_2d_1d_0$ we know that it at most have 3 digits. so we can perform radix sort with 3 iterations.

Radix sort uses counting sort is $O(n+k)$. Since k is at most n This is means that time complexity is at most $O(n)$

3.5.8 B.1

Exercise 8.3-1 (p. 215)
Using Figure 8.3 as a model, illustrate the operation of RADIX-SORT on the following list of English words: **COW, DOG, SEA, RUG, ROW, MOB, BOX, TAB**, BAR, EAR, TAR, DIG, BIG, TEA, NOW, FOX. Use only the first eight words as input (otherwise the exercise will be too long) (**These are highlighted in black**).

CO$\underline{W}$	DO$\underline{G}$	SE$\underline{A}$	RU$\underline{G}$	RO$\underline{W}$	MO$\underline{B}$	BO$\underline{X}$	TO$\underline{B}$
SE$\underline{A}$	MO$\underline{B}$	TO$\underline{B}$	DO$\underline{G}$	RU$\underline{G}$	CO$\underline{W}$	RO$\underline{W}$	BO$\underline{X}$
S$\underline{E}$A	M$\underline{O}$B	T$\underline{O}$B	D$\underline{O}$G	R$\underline{U}$G	C$\underline{O}$W	R$\underline{O}$W	B$\underline{O}$X
S$\underline{E}$A	M$\underline{O}$B	T$\underline{O}$B	D$\underline{O}$G	C$\underline{O}$W	R$\underline{O}$W	B$\underline{O}$X	R$\underline{U}$G
$\underline{S}$EA	$\underline{M}$OB	$\underline{T}$OB	$\underline{D}$OG	$\underline{C}$OW	$\underline{R}$OW	$\underline{B}$OX	$\underline{R}$UG
$\underline{B}$OX	$\underline{C}$OW	$\underline{D}$OG	$\underline{M}$OB	$\underline{R}$OW	$\underline{R}$UG	$\underline{S}$EA	$\underline{T}$OB

3.5.9 B.2

Problem 7-5 (p. 202)
Hint for part c: choose which part to call recursively, instead of always using the left part.

Algorithm 32 TRE-QUICKSORT(A, p, r)

```
1: while p < r do
2:     q = PARTITION(A, p, r)
3:     TRE-QUICKSORT(A, p, q − 1)
4:     p = q + 1
```

a. Argue that TRE-QUICKSORT(A, l, n) correctly sorts the array $A[1 : n]$.

partition until all $q-1$ is biggets element in partition. Then that part of the list is put away at it repeats from the next index after latest partition.

b. Describe a scenario in which TRE-QUICKSORT stack depth is $\Theta(n)$ on an n-element input array.

in worst case we switch two elements only and $p+1$ wich eventually solves.

c. Modify TRE-QUICKSORT so that the worst-case stack depth is $\Theta(log(n))$. Maintain the $O(n \cdot log(n))$ expected running time of the algorithm.

Algorithm 33 NEW-TRE-QUICKSORT(A, p, r)

```
1: q = PARTITION(A, p, r)
2: TRE-QUICKSORT(A, p, q − 1)
3: TRE-QUICKSORT(A, q, r)
```

3.6 Opgaver Uge 12

3.6.1 IA.1

Exam june 2008, 4b
Draw all possible binary search trees that have height 2 and contain four nodes with keys 1, 2, 3, and 4.

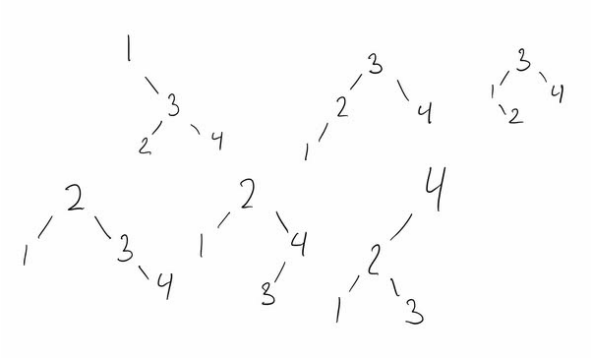


Figure 20: Binary trees with keys 1, 2, 3 and 4

3.6.2 IA.2

Exercise 12.2-1 (p. 319)
You are searching for the number 363 in a binary search tree containing numbers between 1 and 1000. Which of the following sequences cannot be the sequence of nodes examined? **ONLY a-c**.

a. $2, 252, 401, 398, 330, 344, 397, 363.$

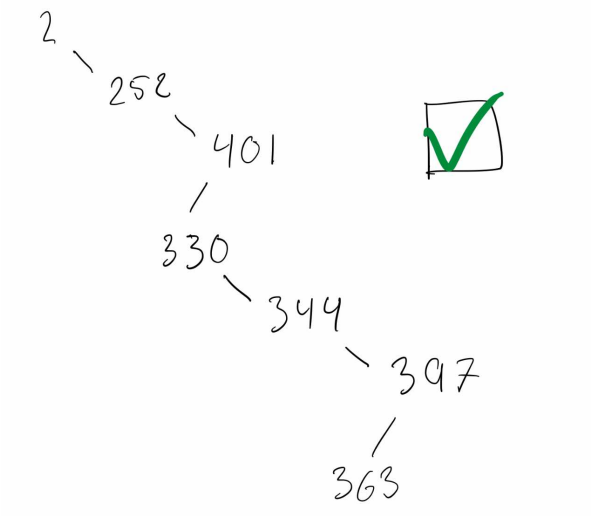


Figure 21: Correct search for 363 in binary tree

b. $924, 220, 911, 244, 898, 258, 362, 363.$

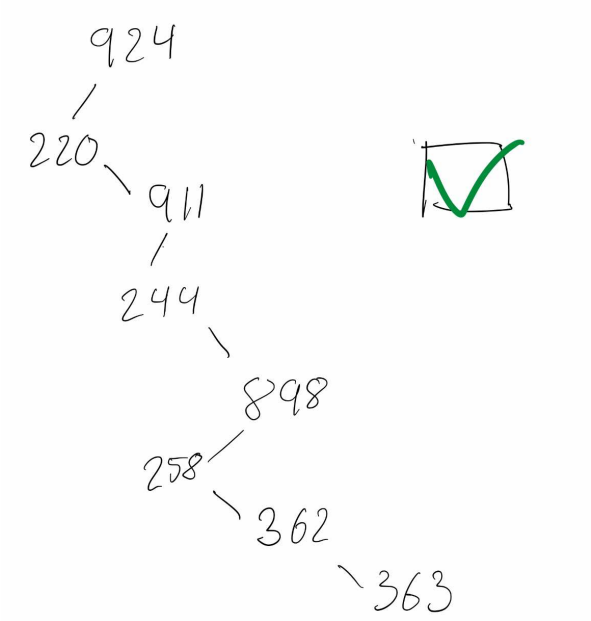


Figure 22: Correct search for 363 in binary tree

c. $925, 202, 911, 240, 912, 245, 363.$

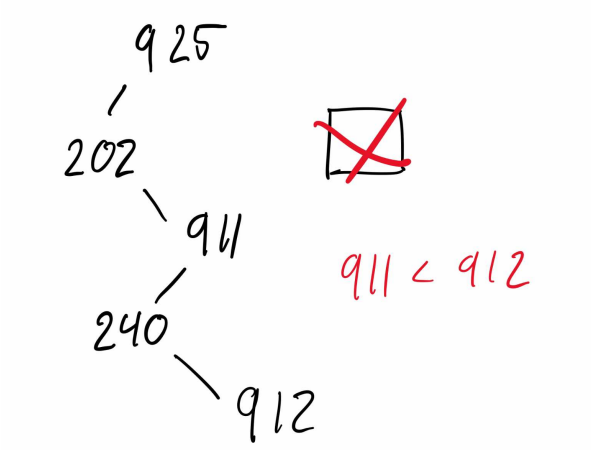


Figure 23: False search for 363 in binary tree

3.6.3 IA.3

Exercise 12.2-3 (p. 320)
Write the TREE-PREDECESSOR procedure.

Algorithm 34 TREE-PREDECESSOR(x)

```
1: if x.left ≠ NILL then
2:     return TREE-MAXIMUM(x.left)
3: else
4:     while y ≠ NILL and x = y.left do
5:         x = y
6:         y = y.p
7:     return y
```

3.6.4 IA.4

Exercise 12.1-5 (p. 315)
Argue that since sorting n elements takes $\Omega(n \cdot log(n))$ time in the worst case in the comparison model, any comparison-based algorithm for constructing a binary search tree from an arbitrary list of n elements takes $\Omega(n \cdot log(n))$ time in the worst case.
For each element in the list, the worstcase walk for placing an element is $log(n)$ so $O(n \cdot log(n))$

3.6.5 IA.5

Exam june 2013, 5

a Here we are looking at sorting n integers with values between 0 and n^4. State the asymptotic worst-case running time on this type of data for the following sorting algorithms:

i) CountingSort
Time complexity: $O(n+k)$ when $k = n^4$, $O(n^4)$

ii) RadixSort, when integers are considered to consist of 4 digits with values between 0 and n.
Time complexity: Counting sort 4 times $O(4(n+k)) = O(n)$ when $k = n$

iii) QuickSort
Time complexity: $O(n^2)$

iv) MergeSort
Time complexity: $O(n \cdot log(n))$

v) InsertionSort
Time complexity: $O(n^2)$

b If CountingSort sorts integers with values between 0 and 7 (inclusive), what are the contents of array C after the completion of the algorithm (with pseudo-code as given in the textbook page 195), when the input A is as follows?

- $A = [7,4,1,2,6,4,0,4,4,4,7,2]$

Answer by listing the elements of array C in order from left to right.

- $C = [1,2,4,4,9,9,10,12]$

3.6.6 IB.1

Exercise 12.1-2 (p. 315)
What is the difference between the binary-search-tree property and the min-heap property on page 163? Can the min-heap property be used to print out the keys of an n-node tree in sorted order in $O(n)$ time? Show how, or explain why not.

> For a min-heap, all childen under a parent must be bigger than the parrent, while in a binary tree we know that trees left of a node are smaller than the node, and those to the right are bigger. To print in you would need to check all leafs n times. This is $O(n \cdot log(n))$ and not $O(n)$

3.6.7 IB.2

Exercise 13.4-9 (p. 334)
Consider the operation RB-ENUMERATE(T,r,a,b), which outputs all the keys k such that $a \leqslant k \leqslant b$ in a subtree rooted at node r in an n-node red-black tree T. Describe how to implement RB-ENUMERATE in $\Theta(m+log(n))$ time, where m is the number of keys that are output. Assume that the keys in T are unique and that the values a and b appear as keys in T . How does your solution change if a and b might not appear in T ? The operation is much more often called RangeSearch than Enumerate. Hint: either make a variant of Inorder-Tree-Walk.

Algorithm 35 RANGE-SEARCH(r,a,b)

```
1:  if r = NILL then
2:      return
3:  if r.key < a then
4:      RANGE-SEARCH(r.right, a, b)
5:  if r.key > b then
6:      RANGE-SEARCH(r.left, a, b)
7:  else
8:      RANGE-SEARCH(r.left, a, b)
9:      print(r.key)
10:     RANGE-SEARCH(r.right, a, b)
```

3.6.8 IIA.1

Exercise 13.1-2 (p. 334)
Draw the red-black tree that results after TREE-INSERT is called on the tree in Figure 13.1 with key 36. If the

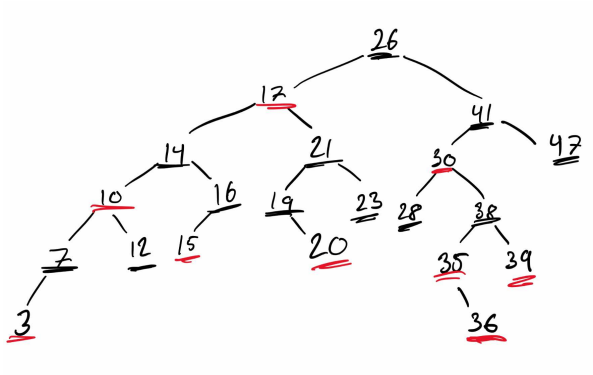


Figure 24: RedBlack tree (figure 13.1 from book)

inserted node is colored red, is the resulting tree a red-black tree?

> No, since two red notes follow each other. What if it is colored black?
> No since there are more more unbalance in amount of black nodes per path.

3.6.9 IIA.2

Exam January 2005, 1
Below are some binary trees with red and black nodes. Gray circles symbolize black nodes, and white circles symbolize red nodes. Question

a: Which of the following trees are red-black trees? Justify your answers.

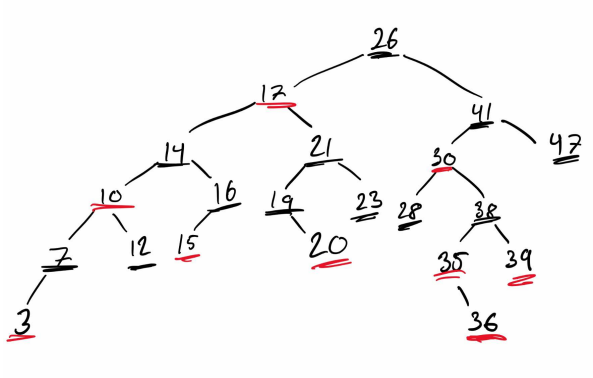


Figure 25: RedBlack tree from exercise IIA.2a

T_1 Since there are precisely 4 black nodes in each path, and no red childparent, the tree is a RB-Tree

T_2 This tree violates the rule about amount of black nodes in every path from root. So not a RB-Tree

T_3 No, RB-Tree must be Inoreder.

T_4 This tree violates the rule about no red childparent. So not a RB-Tree

b: Consider the following red-black tree.

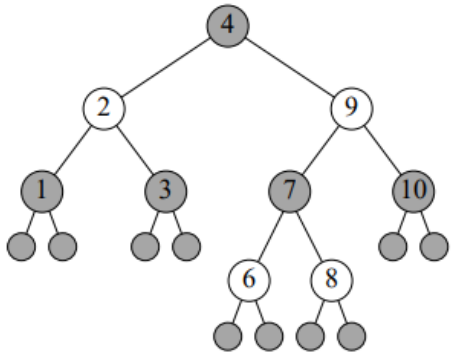


Figure 26: the following red-black tree

Now insert an element with key 5. Draw the tree as it looks after the insertion. Show the individual steps in the insertion.

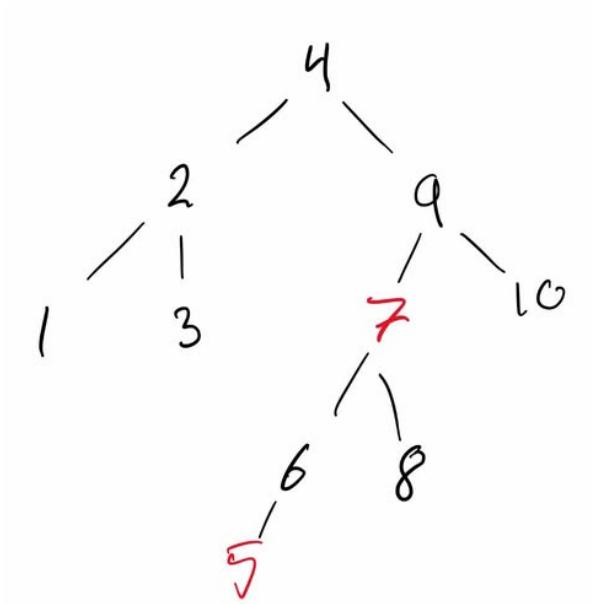


Figure 27: RedBlack tree from exercise IIA.2b

3.6.10 IIA.3

Exercise 13.3-2 (p. 346)
Show the red-black trees that result after successively inserting the keys $41, 38, 31, 12, 19, 8$ into an initially empty red-black tree.

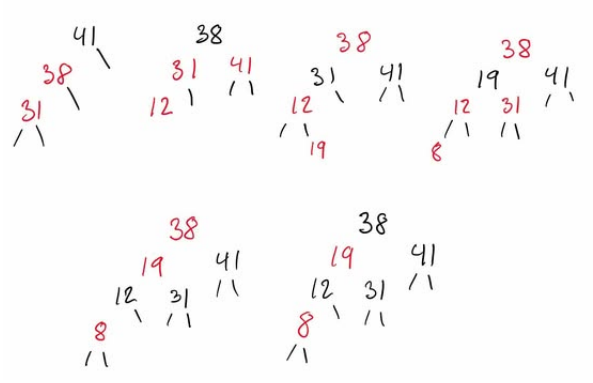


Figure 28: RedBlack tree build from exercise IIA.3

3.6.11 IIA.4

Exercise 13.1-6 (p. 335)
What is the largest possible number of internal nodes in a red-black tree with blackheight k? What is the smallest possible number?

We can count the number nodes as $2^0 + 2^1 \dots 2^{2(k-2)}$
Use -2 because we dont want leafs

$$\sum_{i=0}^{k-2} 2i = k^{2(k-1)}$$

3.6.12 IIA.5

Exercise 12.3-3 (p. 326)
You can sort a given set of n numbers by first building a binary search tree containing these numbers (using TREE-INSERT repeatedly to insert the numbers one by one) and then printing the numbers by an inorder tree walk. What are the worstcase and best-case running times for this sorting algorithm?

- Worst case is always n^2
- Best is then $n \cdot log(n)$

Now, answer this problem again, but now with red-black trees instead of unbalanced binary search trees.

- Worst case must be $n \cdot log(n)$ since rebalancing is $log(n)$
- Best case $n \cdot log(n)$

3.6.13 IIB.1

Exam June 2011, 1.

a) Consider the eight trees in Figure 1. Nodes in bold are black, and the rest are red. State which of the eight trees are red-black trees.

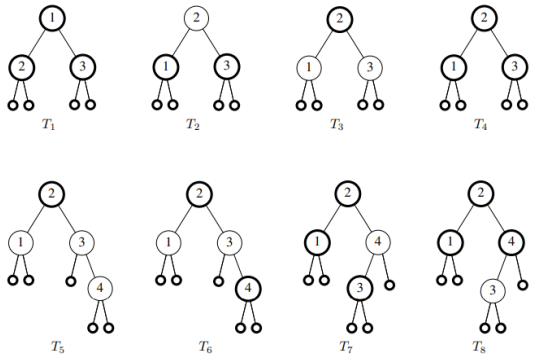


Figure 29: RB-trees from exercise

T_1 No, no red nodes

T_2 No, root cannot be red

T_3 Yes, all rules are held

T_4 Yes, all rules are held

T_5 No, two red nodes

T_6 No, not correct amount of black nodes in each path

T_7 No, not correct amount of black nodes in each path

T_8 Yes, all rules are held

b) Consider the red-black tree T below.

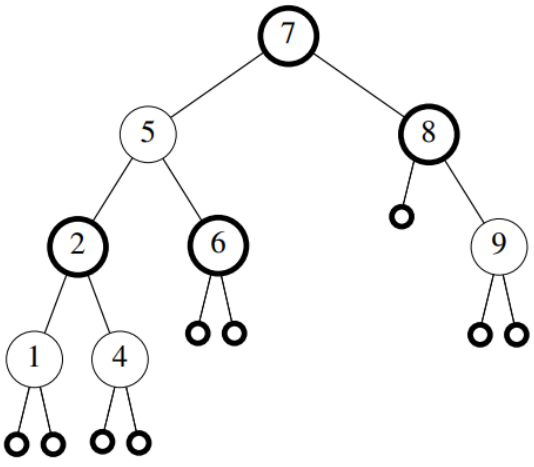


Figure 30: Tree before inserting 3-node

Now a node with key 3 is inserted. Which of the trees in figures 2 and 3 corresponds to T after the node with key 3 is inserted?

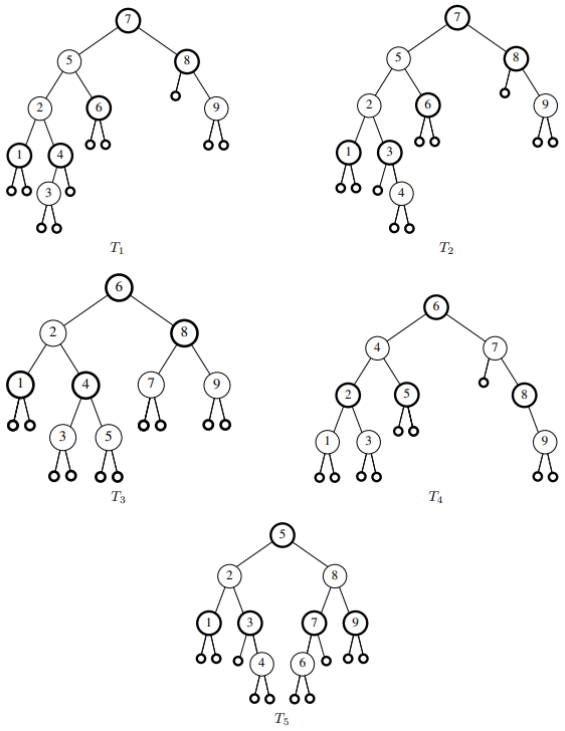


Figure 31: First five trees

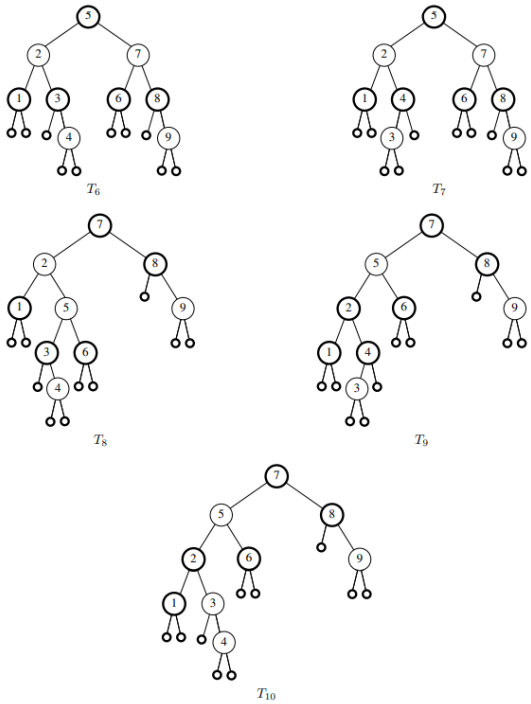


Figure 32: Last five trees

must be the 7th

3.6.14 IIB.2

useless timing exercise

3.7 Opgaver Uge 13

3.7.1 A. 1

Exam June 2015, 8.

a Specify a coloring of the nodes in the tree below that makes it a red-black tree. Answer by writing a list of

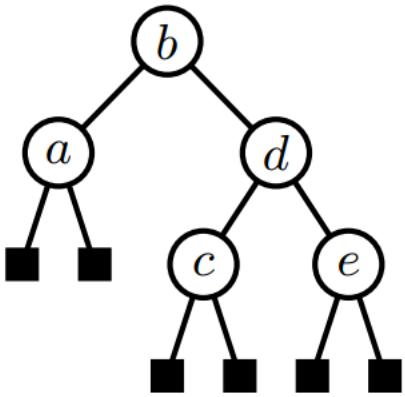


Figure 33: Tree from exam 2015 8a

the names of the black nodes and a list of the names of the red nodes.

$black = (a, b, c, e)$ and $red = (d)$

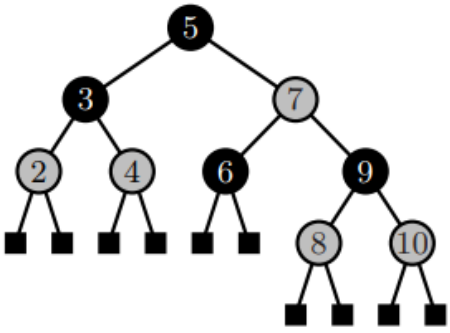


Figure 34: Tree from exam 2015 8b

b In the above red-black tree, 11 is inserted using the algorithm from the textbook. State which of the four trees below is the result.

Must be T_3

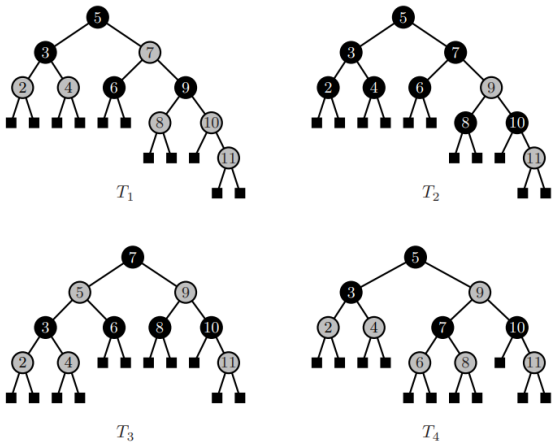


Figure 35: Trees from exam 2015 8b

3.7.2 A. 2

Exam June 2009, 1b.
Consider the red-black tree below, where black nodes are drawn in bold. Draw the tree as it appears after the node with key 2 is deleted.

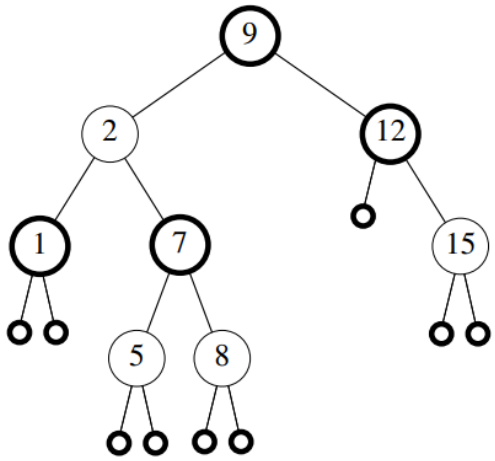


Figure 36: Tree from exam 2009 1b

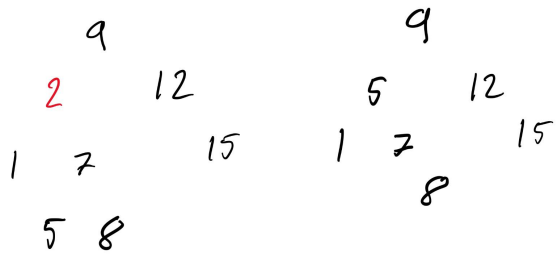


Figure 37: Tree after deletion

3.7.3 A. 3

Exam June 2016, 4.
Below is a hash table H that uses the hash function

$$h'(x) = (7x+4) \mod 11$$

and linear probing (in the textbook $h'(x)$ is called the "auxiliary hash function" in this context).

$$H = (67, 20, 17, \quad, 33, \quad, 16, 2, \quad, \quad, 15)$$

Insert the values 18 and 26 (in that order). Give the appearance of the hash table after each of the two insertions. Answer by writing the contents of H in order from left to right, with blank spaces indicated as x.

- $h'(18) = 9$ so $H = (67, 20, 17, \quad, 33, \quad, 16, 2, \quad, 18, 15)$
- $h'(26) = 10$ so $H = (67, 20, 17, 26, 33, \quad, 16, 2, \quad, 18, 15)$

3.7.4 A. 4

Exercise 11.2-2 (p. 281)
Consider a hash table with 9 slots and the hash function $h(k) = k \mod 9$. Demonstrate what happens upon inserting the keys $5, 28, 19, 15, 20, 33, 12, 17, 10$ with collisions resolved by chaining.

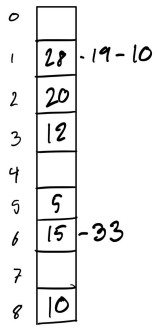


Figure 38: Hashtable 11.2-2

3.7.5 A. 5

Exercise 12.4-1 (p. 300) *Only fist **5 keys***
Consider inserting the keys $10, 22, 31, 4$ into a hash table of length $m = 11$ using open addressing. Illustrate the result of inserting these keys using linear probing with $h(k,i) = (k+i) \mod m$ and using double hashing with $h_1(k) = k$ and $h_2(k) = 1 + (k \mod (m-1))$. in both instances

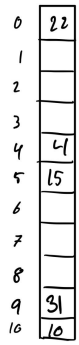


Figure 39: Hashtable 12.4-1

3.7.6 A. 6

Exam January 2008, 1c.
Do linea probing on $H = (\quad, 11, 31, \quad, 24, 15, \quad, \quad, 48, \quad)$
and the hashfunction $h(k) = k \mod 10$
Now insert 4.

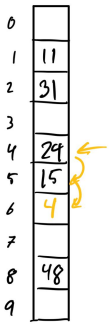


Figure 40: Hashtable after insertion

3.7.7 A. 7

Exam January 2006, 1a.
Given the following hashtable $H = (\quad, \quad, \quad, 14, \quad, \quad, \quad, 3, \quad, 41, \quad)$
and the following hashfunction: $h(k,i) = (h_1(k) + i \cdot h_2(k)) \mod 11$
Where $h_1(k) = k \mod 11$ and $h_2(k) = 1 + (k \mod 10)$
Now insert 18 into the hashtable

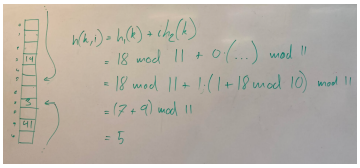


Figure 41: Hashtable after insertion

3.7.8 B. 1

Exercise 13.4-4 (p. 354) In Exercise 13.3-2 on page 346, you found the red-black tree that results from successively inserting the keys $41, 38, 31, 12, 19, 8$ into an initially empty tree. Now show the red-black trees that result from the successive deletion of the keys in the order $8, 12, 19, 31, 38, 41$.

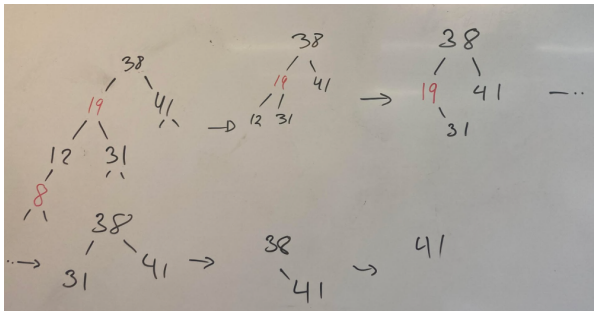


Figure 42: Deleting elements in a RB tree

3.7.9 B. 2

Exam June 2013, 3b. Given the following hashtable $H = (3, \quad, \quad, 59, \quad, 23, 38, 53, \quad, 72, 87)$
and the following hashfunction: $h(k) = (3x + 2) \mod 11$
Now insert 60 and 45 into the hashtable

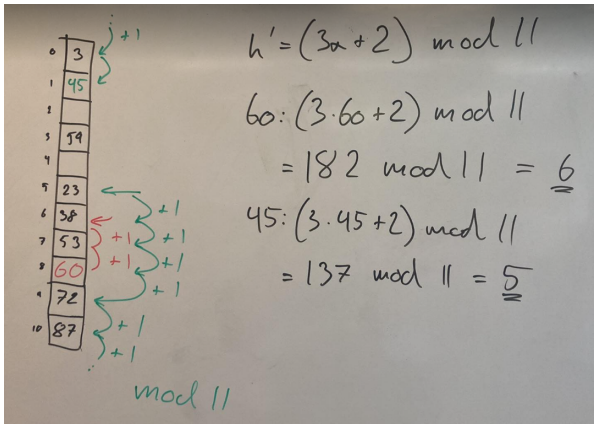


Figure 43: Hashtable after insertion

3.8 Opgaver Uge 14

3.8.1 IA. 1

Exercise 17.1-1 (p 485)
Show how OS-SELECT$(T.root, 10)$ operates on the red-black tree T shown in Figure 17.1.

Algorithm 36 OS-SELECT(x, i)

```
1: r = x.left.size + 1
2: if i = r then
3:     return x
4: else if i < r then
5:     return OS-SELECT(x.left, i)
6: else
7:     return OS-SELECT(x.right, i)
```

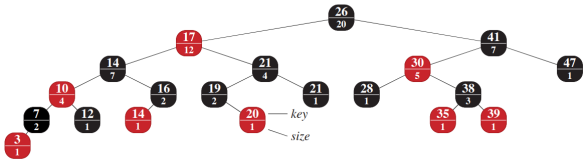


Figure 44: red-black tree. Figure 17.1

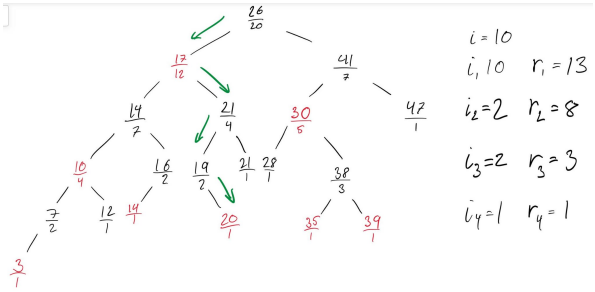


Figure 45: OS-SELECT$(T.root, 10)$

3.8.2 IA. 2

Exercise 17.1-2 (p 485)
Show how OS-RANK(T, x) operates on the red-black tree T shown i n Figure 17.1 and the node x with $x.key = 35$.

Algorithm 37 OS-RANK(T, x)

```
1: r = x.left.size + 1
2: y = x
3: while y ≠ T.root do
4:     if y < y.p.right then
5:         r = r + y.p.left.size + 1
6:     y = y.p
7: return r
```

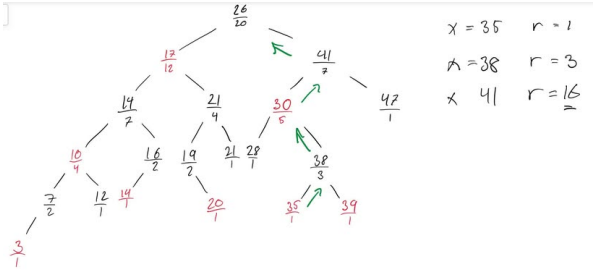


Figure 46: OS-RANK$(T, 35)$

3.8.3 IA. 3

Exercise 17.1-5 (p 486)
Given an element x in an n-node order-statistic tree and a natural number i, show how to determine the ith successor of x in the linear order of the tree in $O(n \cdot log(n))$time.
Reading "find" instead of "determine" may make the task a little clearer. You don't necessarily need to provide pseudo-code, but can also simply describe the idea of your algorithm, e.g. with figures.

Algorithm 38 ITH-SUCCESSOR(T, x, i)

1: $r =$ OS-RANK(T, x)
2: **return** OS-SELECT$(T.root, r + i)$

3.8.4 IA. 4

Exam january 2008, 3
Let $(x1, y1), (x2, y2), \ldots, (xn, yn)$ be points in the plane. Assume that the coordinates are positive integers and that all the x-coordinates are distinct. We will look at the MINABOVE(t) operation. If at least one point is found whose y-coordinate is greater than or equal to t, the one of those points with the least x-value is returned. Otherwise, it is reported that no such point was found. In the example below, the point with the circle around it is returned.

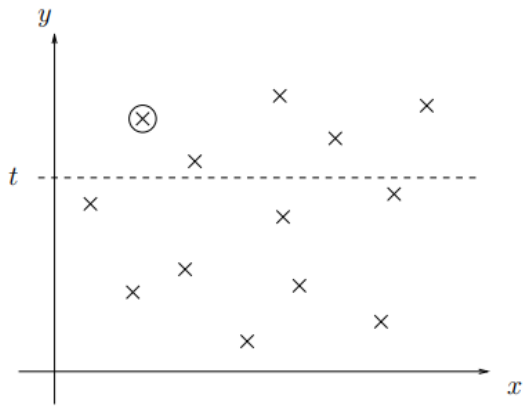


Figure 47: Coordinates

a) What is MINABOVE(10) for the points $(4,9), (5,17), (19,6), (23,10), (25,15), (40,7)$?

$(5,17)$

In the following, an extension of red-black trees for storing the points is considered. Each node contains a point, i.e. x-coordinate and y-coordinate, and the x-coordinates are used as keys in the tree. In addition, each node contains a number y_{max}, which indicates the largest y-coordinate in the node's subtree. Below is an extended red-black tree for the points in question a. The nodes are indicated by x, y at the top and y_{max} at the bottom. Black nodes are drawn in bold.

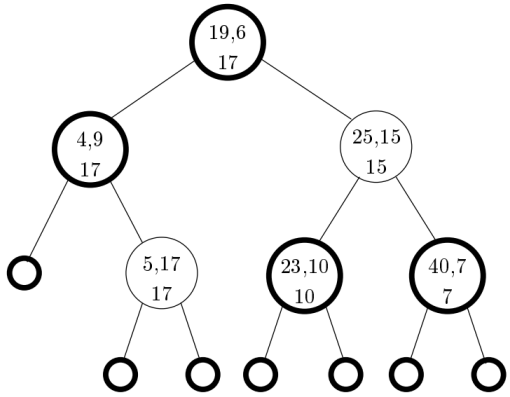


Figure 48: Extended RB-Tree

b) Describe how an extended red-black tree can be maintained during insertion and deletion of points.

update y_{max} then rebalance if needed

c) Illustrate part of the answer from question b by inserting the point $(30, 11)$ into the extended red-black tree shown at the top of this page.

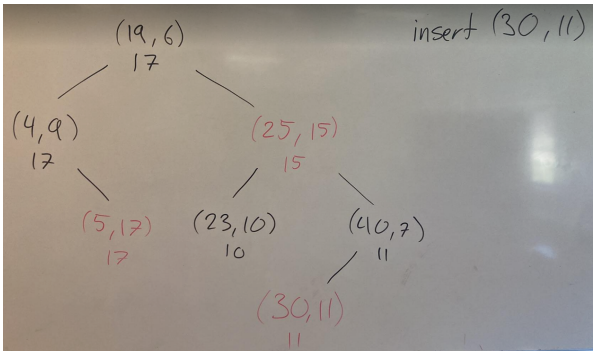


Figure 49: Extended RB-Tree

d) Describe how MINABOVE(t) can be executed in time $O(log(n))$.

Algorithm 39 MINABOVE(t)

```
1: node = T.root
2: while node.max_y < t do
3:     node = node.left
4: return node.x
```

3.8.5 IA. 5

Exercise 2.1-4 (p 25)
Consider the *searching problem*:

Input: A sequence of n numbers $<a_1, a_2, \ldots a; n>$ stored in array $A[1:n]$ and a value x. **Output:** An index i such that x equals $A[i]$ or the special value NIL if x does not appear in A. Write pseudocode for *linear search*, which scans through the array from beginning to end, looking for x. Using a loop invariant, prove that your algorithm is correct. Make sure that your loop invariant follows the three necessary properties.

Algorithm 40 LINEARSEARCH(A, x)

```
1: i = 1
2: while i ⩾ A.length and A[i] ≠ x do
3:     i = i + 1
4: if i = A.length then
5:     return NIL
6: else
7:     return i
```

Invariant:

- intial: x is not in $A[1 \ldots 0] = \varnothing$
- Maintenance:
  1. $i \geqslant A.length \wedge A[i] \neq x \wedge$ "x is not in $A[1 \ldots (1 \ldots i-1)]$"
  2. $i' = i + 1$it now holds that "x is not in $A[1 \ldots (1 \ldots i)]$" = "x is not in $A[1 \ldots (1 \ldots i' - 1)]$"
- Termination
  - Loop stopped because $A[i] = x$, so $x \in A[1 \ldots (i-1)]$
  - Loop stopped because $i = A.length \Rightarrow A[1 \ldots (i-1)] = A[1 \ldots (n+1-1)] = A$

3.8.6 IA. 6

Exercise 2.2-2 (p 33)
Consider sorting n numbers stored in array $A[1:n]$ by first finding the smallest element of $A[1:n]$ and exchanging it with the element in $A[1]$. Then and the smallest element of $A[2:n]$, and exchange it with $A[2]$. Then and the smallest element of $A[3:n]$, and exchange it with $A[3]$. Continue in this manner for the first $n-1$ elements of A. Write pseudocode for this algorithm, which is known as selection sort.

Algorithm 41 SELECTIONSORT(A)

```
1: n = A.length
2: for i = 1 to n do
3:     min_index = i
4:     for j = i + 1 to n do
5:         if then A[j] < A[min_index]
6:             min_index = j
7:     swap element A[min_value] and A[i]
8: return A
```

What loop invariant does this algorithm maintain?

- intial: $i = 1$ and $A[1 \ldots 0] = \varnothing$ is sorted and $A[i \ldots n] \geqslant A[0]$
- Maintenance:
  1. $i \geqslant n-1 \wedge$ "$A[1 \ldots (i-1)]$ is sorted and $A[i \ldots n] \geqslant A[i-1]$

2. "$A[i \ldots 1]$ is sorted and $A[i+1 \ldots n] \geqslant A[i]$
3. $i' = i+1$ "$A[1 \ldots (i'-1)]$ is sorted and $A[i' \ldots n] \geqslant A[i'-1]$

- Termination
  - Loop stopped because $i = n.A[1 \ldots (i-1)] = A[1 \ldots (n-1)]$ is sorted and $A[n] \geqslant A[n-1]$

Why does it need to run for only the first $n-1$ elements, rather than for all n elements?

Because of the invariant, the last element must be the largest, so the array will be sorted at $n-1$

Give the worst-case running time of selection sort in ,-notation. Is the best-case running time any better?

$\Theta(n^2)$

3.8.7 IB. 1

Exercise 17.1-7 (p 486)
Show how to use an order-statistic tree to count the number of inversions (see Problem 2-4 on page 47) in an array of n distinct elements in $O(n \cdot log(n))$ time. Hint: think of insertion sort, but insert into a tree. The elements in the array are all assumed to be different, cf. exercise 2-4 page 47

3.8.8 IB. 2

Exam june 2013, 6
The following code is intended to compute $n! = 1 \cdot 2 \cdot 3 \cdots (n-1) \cdot n$ for integers $n \geqslant 1$.

Algorithm 42 FACTORIAL(n)

```
1: i = n
2: r = 1
3: while i > 1 do
4:     r = r · i
5:     i = i − 1
6: return r
```

Indicate for each of the following statements whether it is a loop invariant for the algorithm (i.e., always true when the test at the beginning of the while loop is executed) for input where n is an integer $n \geqslant 1$. (You do not need to justify your answers.)

3.8.9 IIA. 1

Solve the recursion-equation

$$T(n) = 4 \cdot T(n/3) + n$$

Solve it both via the recursion-tree-method and via the Master Theorem.

$$\begin{gathered}
4 \cdot n/3 = n(4/3) \\
4^2 \cdot n/3^2 = n(4/3)^2 \\
\Downarrow \\
4^i \cdot n/3^i = n(4/3)^i \\
\Downarrow \\
4^{log_3(n)} \cdot n/3^{log_3(n)} = 4^{log_3(n)} \cdot n/n \\
\Downarrow \\
\text{Last layer dominates, so } \Theta(4^{log_3(n)}) \qquad \Downarrow \\
n^{log_3(4)} = n^{ln(4)/ln(3)} \\
\Downarrow \\
n^{1.2619}
\end{gathered}$$

Now for the Master theorem, we first find a, b and $f(n)$

- $a = 4$
- $b = 3$
- $f(n) = n$

step 1 $\alpha = log_3(4) = 1.2619$

step 2 compare $n^{1.2619}$ and $f(n) = n$

step 3 Case 1 says: $\Theta(n^{1.2619})$

3.8.10 IIA. 2

Solve the recurrence equation

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

Solve it both via the recursion-tree-method and via the Master Theorem. Since $T(n/2)$ is decreasing n^2 dominates to $\Theta(n^2)$ Now for the Master theorem, we first find a, b and $f(n)$

- $a = 1$
- $b = 2$
- $f(n) = n^2$

step 1 $\alpha = log_2(1) = 0$

step 2 compare n^0 and $f(n) = n$

step 3 Case 3 says: $O(n^2)$

3.8.11 IIA. 3

Solve the recurrence equation

$$T(n) = 16 \cdot T(n/2) + n^4 + n^2$$

Solve it via the Master Theorem.

- $a = 16$
- $b = 2$
- $f(n) = n^4$

step 1 $\alpha = log_2(16) = 4$

step 2 compare n^4 and $f(n) = n^4$

step 3 Case 2 says: $\Theta(n^4 \cdot log(n))$

3.8.12 IIB. 1

Exam january 2006, 1c State the asymptotic solution to this recurrence equation. $T(n) = 3T(n/3) + n^2$ Solve it via the Master Theorem.

- $a = 3$
- $b = 3$
- $f(n) = n^2$

step 1 $\alpha = log_3(3) = 0$

step 2 compare $n^0 = 1$ and $f(n) = n^2$

step 3 Case 2 says: $\Theta(n^2)$

3.8.13 IIB. 2

Solve the recurrence equation

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

Solve it both via the recursion-tree-method and via the Master Theorem.
First the recursion tree

$$4 \cdot n/2 = n(4/2)$$
$$4^2 \cdot n/2^2 = n(4/2)^2$$
$$\Downarrow$$
$$4^i \cdot n/2^i = n(4/2)^i$$
$$\Downarrow$$
$$4^{log_2(n)} \cdot n/2^{log_2(n)} = 4^{log_2(n)} \cdot n/n$$
$$\Downarrow$$
Compare $4^{log_2(n)}$ and n
Last layer dominates, so $\Theta(4^{log_2(n)})$ $\Downarrow$
$$n^{log_2(4)} = n^{ln(4)/ln(2)}$$
$$\Downarrow$$
$$\Theta(n^2)$$

Now via the Master Theorem.

- $a = 4$
- $b = 2$
- $f(n) = n$

step 1 $\alpha = log_2(4) = 2$

step 2 compare n^2 and $f(n) = n$

step 3 Case 2 says: $\Theta(n^2)$

3.9 Opgaver Uge 15

3.9.1 A. 1

Exercise 4.5-1 (p 106) Use the Master Theorem to give tight asymptotic bounds for the following recurrences.

a. $T(n) = 2 \cdot T(n/4) + 1$

   - $a = 2$
   - $b = 4$
   - $f(n) = 1$

   step 1 $\alpha = log_4(2) = 0.5$

   step 2 compare $n^{0.5}$ and $f(n) = 1$

   step 3 Case 1 says: $\Theta(n^{0.5})$

b. $T(n) = 1 \cdot T(n/4) + \sqrt{n}$

- $a = 2$
- $b = 4$
- $f(n) = 1$

step 1 $\alpha = log_4(2) = 0.5$
step 2 compare $n^{0.5}$ and $f(n) = \sqrt{n}$
step 3 Case 2 says: $\Theta(n^{0.5} \cdot log(n))$

c. $T(n) = 2 \cdot T(n/4) + \sqrt{n} \cdot log^2(n)$

- $a = 2$
- $b = 4$
- $f(n) = 1$

step 1 $\alpha = log_4(2) = 0.5$
step 2 compare $n^{0.5}$ and $f(n) = \sqrt{n} \cdot log^2(n)$
since $\sqrt{n} \cdot log^2(n) = n^{0.5} \cdot log(n)^k$
step 3 Case 2 says: $\Theta(n^{0.5} \cdot log^3(n))$

d. $T(n) = 2 \cdot T(n/4) + n$

- $a = 2$
- $b = 4$
- $f(n) = 1$

step 1 $\alpha = log_4(2) = 0.5$
step 2 compare $n^{0.5}$ and $f(n) = n$
step 3 Case 3 says: $\Theta(n)$

e. $T(n) = 2 \cdot T(n/4) + n^2$

- $a = 2$
- $b = 4$
- $f(n) = 1$

step 1 $\alpha = log_4(2) = 0.5$
step 2 compare $n^{0.5}$ and $f(n) = n^2$
step 3 Case 3 says: $\Theta(n^2)$

3.9.2 A. 2

Exercise 4.5-3 (p 106) Use the master method to show that the solution to the binary search recurrence $T(n) = T(n/2) + \Theta(1)$ is $T(n) = \Theta(log(n))$ (See Exercise 2.3-6 for a description of binary search.)

- $a = 1$
- $b = 2$
- $f(n) = \Theta(1)$

step 1 $\alpha = log_2(1) = 0$

step 2 compare n^0 and $f(n) = \Theta(1)$

step 3 Case 2 says: $\Theta(n^0 \cdot log(n)^{0+1}) = \underline{\underline{\Theta(log(n))}}$

3.9.3 A. 3

Using the recursion tree method solve the following recurrence equation

$$T(n) = 2 \cdot T(n-1) + 1$$

$$\begin{gathered}
2 \cdot n - 1 = 1 \\
\Downarrow \\
2^2 \cdot (n-1)^2 = 2 \\
\Downarrow \\
2^i \cdot (n-1)^i = 2^i \\
\Downarrow \\
2^n \cdot (n-1)^n = 2^n - 1 \\
\Downarrow \\
\text{since } \sum_{i=0}^{n-1} 2^i = 2^{(n-1)+1} - 1 = 2^n - 1 \\
\Downarrow \\
T(n) = \Theta(2^n - 1) = \Theta(2^n)
\end{gathered}$$

Can this be solved using the Master Theorem?
No because there is no b variable

3.9.4 A. 4

Exercise 2.3-5 (p 106) You can also think of insertion sort as a recursive algorithm. In order to sort $A[1:n]$, recursively sort the subarray $A[1:n-1]$ and then insert $A[n]$ into the sorted subarray $A[1:n-1]$. Write pseudocode for this recursive version of insertion sort. Give a recurrence for its worst-case running time.

Algorithm 43 INSERTIONSORTRECURSIVE(A, i)

```
1: if i ⩾ 2 then
2:     key = A[i]
3:     INSERTIONSORTRECURSIVE(A, i − 1)
4:     j = i − 1
5:     while j ⩾ 1 and key⩾ A[j] do
6:         A[j + 1] = A[j]
7:         j = j − 1
8: return r
```

Now solve the recurrence equation using the recursion tree method.

$$T(n) = T(n-1) + n$$

$$\begin{gathered}
n \\
\Downarrow \\
n-1 \\
\Downarrow \\
n-i \\
\Downarrow \\
\text{since } 1 - \sum_{i=1}^{n} i = \frac{n \cdot (n-1)}{2} \\
\Downarrow \\
T(n) = \Theta(2^n - 1) = \Theta(2^n)
\end{gathered}$$

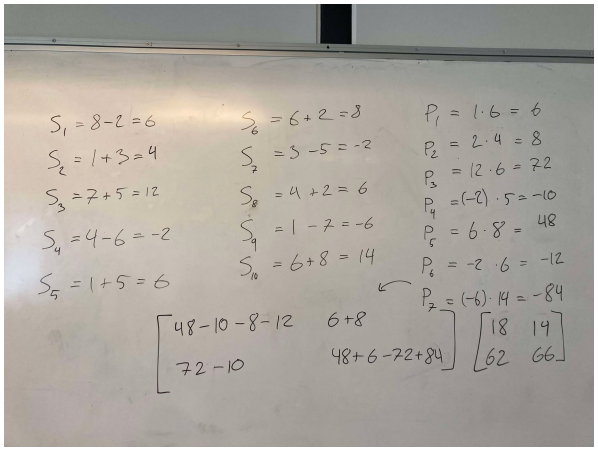


Figure 50: matrix multiplication using strassens algorithm

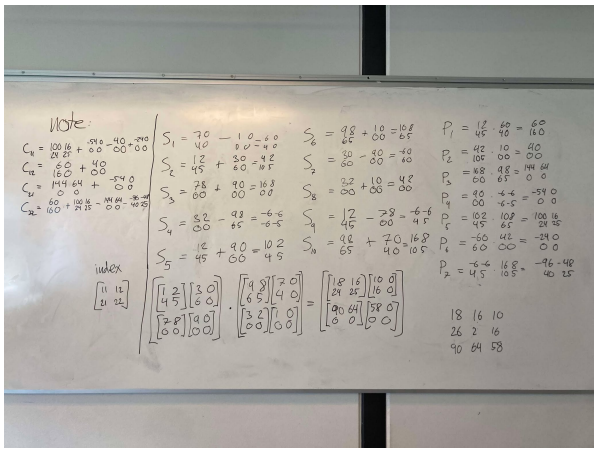


Figure 51: We tried

Can the recurrence equation be solved using the Master Theorem?

3.9.5 A. 5

Exercise 4.5-1 (p 106) Use Strassen's algorithm to compute the matrix product

$$\begin{bmatrix} 1 & 3 \\ 7 & 5 \end{bmatrix} \begin{bmatrix} 6 & 8 \\ 4 & 2 \end{bmatrix}$$

Show your work.

That is, compute $S_1, \ldots, S_{10}$, $P_1, \ldots, P_7$, as well as C_{11}, C_{12}, C_{21}, and C_{22} for the simple case where A_{ij}, B_{ij}, and C_{ij} are 1×1 matrices, i.e., just numbers.

Note that here we see that Strassen's algorithm can be said to originate from discovering how the matrix product of two 2×2 matrices can be computed using only 7 multiplications of numbers. Strassen then simply replaces numbers in 2×2 matrices with $\frac{n}{2} \times \frac{n}{2}$ submatrices in $n \times n$ matrices. This helps in understanding the formulation of Exercise 4.2-4.

3.9.6 A. 6

How can Strassen's algorithm be extended so that it can multiply $n \times n$ matrices when n is not a power of two? [Hint: pad with zeros.] Argue that the runtime is still $\Theta(n^{\log_2 7})$.

3.9.7 B. 1

Exam june 2013, 1
This assignment is about the following recurrence relations: Solve only using the Master Theorem

i $T(n) = 8 \cdot T(n/3) + n^2$

- $a = 8$
- $b = 3$
- $f(n) = n^2$

step 1 $\alpha = log_3(8) \approx 1.9$
step 2 compare $n^{1.9}$ and $f(n) = n^2$
step 3 Case 3 says: $\Theta(n^2)$

furthermore $a \cdot f(n/b) \leqslant c \cdot f(n)$
so... $8 \cdot f(n/3) \leqslant c \cdot f(n)$

ii $T(n) = 9 \cdot T(n/3) + n^2$

- $a = 9$
- $b = 3$
- $f(n) = n$

step 1 $\alpha = log_3(3) = 1$
step 2 compare n^1 and $f(n) = n$
step 3 Case 3 says: $\Theta(n \cdot log(n))$

iii $T(n) = 10 \cdot T(n/3) + n^2$

- $a = 10$
- $b = 3$
- $f(n) = n \cdot log(n)$

step 1 $\alpha = log_4(4) = 1$
step 2 compare n^1 and $f(n) = n \cdot log(n)$
step 3 Case 1 says: $\Theta(n \cdot log(n))$

3.9.8 B. 2

Solve the following recurrence equation using the Master Theorem

$$T(n) = 4 \cdot T(n/2) + n^2 \cdot log(n)$$

- $a = 4$
- $b = 2$
- $f(n) = n^2$

step 1 $\alpha = log_2(4) = 2$

step 2 compare n^2 and $f(n) = n^2$

step 3 Case 1 says: $\Theta(n \cdot log(n))$

Can the recurrence relation be solved using the Master Theorem? *[Technical detail: for this recurrence relation, the base case (and thus the input size at the leaves) must be $n \leqslant 5$ (for example, $n = 4$ gives $4/2 + 2 = 4$, which shows that the recursive call does not decrease, i.e., the recursion will never stop if the base case threshold is set lower).]* Hint: take inspiration from the last example in the slides (about floors and ceilings).

3.9.9 B. 3

Solve the following recurrence equation using the recursion tree method.

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

$$4 \cdot n/2 + 2$$
$$4^2 \cdot (n/2+2)^2$$
$$\Downarrow$$
$$4^i \cdot (n/2+2)^i$$
$$\Downarrow$$
$$4^{log_2(n)} \cdot (n/2+2)^{log_2(n)}$$
$$\Downarrow$$
$$\text{Compare } 4^{log_2(n)} \text{ and } n$$
$$\text{Last layer dominates, so } \Theta(4^{log_2(n)})$$
$$\Downarrow$$
$$n^{log_2(4)} = n^{ln(4)/ln(2)}$$
$$\Downarrow$$
$$\Theta(n^2)$$

3.9.10 B. 4

Exercise 4.2-4 (p.89)
Write pseudocode for Strassen's algorithm.

Algorithm 44 STRASSEN(A,B)

if A and B is 2x2 **then**
    compute the following
    $S_1 = (a_{11} + a_{22}) \cdot (b_{11} + b_{22})$
    $S_2 = (a_{21} + a_{22}) \cdot b_{11}$
    $S_3 = a_{11} \cdot (b_{12} - b_{22})$
    $S_4 = a_{22} \cdot (b_{21} - b_{11})$
    $S_5 = (a_{11} + a_{12}) \cdot b_{22}$
    $S_6 = (a_{21} + a_{22}) \cdot (b_{11} + b_{12})$
    $S_7 = (a_{12} + a_{22}) \cdot (b_{21} + b_{22})$
    now $C = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} \cdot \begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{bmatrix} = \begin{bmatrix} S_1 + S_4 - S_5 + S_7 & S_3 + S_5 \\ S_2 + S_4 & S_1 + S_2 - S_3 + S_6 \end{bmatrix}$
    return C
else if if A and B is equal in size and their number of rows an columns is n^2 **then**
    Divide matrices into 4 parts
    STRASSEN(A,B) on the new two matrices
else
    Fill the matriceis with 0's so they are the same size, and that their number of rows an columns is 4^k, where k is an positive interger
    STRASSEN(A,B)

3.9.11 B. 5

Exercise 4.5-4 (p.106)
Consider the function $f(n) = log(n)$. Argue that although $f(n/2) < f(n)$, the regularity condition $a \cdot f(n/b) \leqslant c \cdot f(n)$ with $a = 1$ and $b = 2$ does not hold for any constant $c < 1$. Argue further that for any $\varepsilon > 0$, the condition in case 3 that $f(n) = \Theta(n^{log_b(a+\varepsilon)})$ does not hold.

3.10 Opgaver uge 17

3.10.1 IA. 1

Exam june 2016, 1

i $T(n) = 2 \cdot T(n/2) + 1$

- $a = 2$
- $b = 2$
- $f(n) = 1$

step 1 $\alpha = log_2(2) = 1$

step 2 compare n^1 and $f(n) = 1$

step 3 Case 3 says: $\Theta(n)$

furthermore $a \cdot f(n/b) \leqslant c \cdot f(n)$

so... $8 \cdot f(n/3) \leqslant c \cdot f(n)$

ii $T(n) = 3 \cdot T(n/3) + n^2$

- $a = 3$
- $b = 3$
- $f(n) = n$

step 1 $\alpha = log_3(3) = 1$

step 2 compare n^1 and $f(n) = n$

step 3 Case 2 says: $\Theta(n \cdot log(n))$

iii $T(n) = 4 \cdot T(n/4) + n^2$

- $a = 4$
- $b = 4$
- $f(n) = n \cdot log(n)$

step 1 $\alpha = log_4(4) = 1$

step 2 compare n^1 and $f(n) = n \cdot log(n)$

step 3 Case 1 says: $\Theta(n \cdot log(n))$

iv $T(n) = 5 \cdot T(n/5) + n^2$

- $a = 5$
- $b = 5$
- $f(n) = n^2$

step 1 $\alpha = log_4(4) = 1$

step 2 compare n^1 and $f(n) = n^2$

step 3 Case 1 says: $\Theta(n^2$

3.10.2 IA. 2

Exam june 2016, 2 Which of the following statements are true

i 1 is $O(2)$ *True*

ii 1 is $\Theta(2)$ *True*

iii n is $O(n^2)$ *True*

iv n is $\Theta(n^3)$ *False*

v $3x+2x^2+x^3$ is $\Theta(x+2x^2+3x^3)$ *True*

vi $log(n)$ is $o(n/log(n))$ *True*

vii $n^{1/2}$ is $o(n/2^n)$ *False*

viii $log(n)$ is $\omega(log(n))$ *True*

ix $2^n \cdot log(n)$ is $\omega(2^n)$ *False*

x $n^2/log(n)$ is $O(n \cdot log^2(n)$ *False*

3.10.3 IA. 3

Exam june 2016, 3 Do BUILD-MAX-HEAP(A) on the following array A.
$< 2,1,5,4,8,6,7,9,3 > < 1,2,3,4,5,6,7,8,9 >$

3.10.4 IA. 4

Exam june 2016, 7
Binary trees are considered different if they have different shapes. The figure below shows two different binary trees, each with five nodes.

Figure 52: The figure below

- a)
  How many different binary trees with three (3) nodes are there? Answer by stating the number.

  In the rest of the assignment, we consider binary trees with keys in the nodes. Two trees are considered different if they have different shapes, or if they have the same shape but differ in the key of at least one node. The figure below shows three different binary trees with five nodes and the keys $1,2,3,4,5$.

- b)
  How many different binary trees with three nodes and the keys 1, 2, 3 satisfy the max-heap order? Answer by stating the number.

- c)
  How many different binary trees with three nodes and the keys 1, 2, 3 satisfy the inorder sequence? Answer by stating the number.

- d)
  How many different binary trees with three nodes and the keys 1, 2, 3 satisfy both the inorder sequence and the max-heap order? Answer by stating the number.

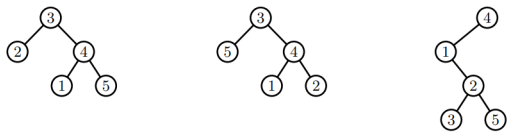


Figure 53: binary trees

3.10.5 IA. 5

Exam june 2016, 8
The following code is intended to compute $7 \cdot n$ for integers $n \geqslant 0$.

Algorithm 45 MULTSEVEN(n)

```
1: x = n
2: r = 0
3: while x > 0 do
4:     x = x − 1
5:     r = r + 7
6: return r
```

State for each of the following statements whether it is a *loop invariant* for the algorithm (i.e., always true when the condition at the start of the while loop is evaluated) for all inputs where the input is an integer $n \geqslant 0$. (You do not need to justify your answers.)

i) $r = 7n$ *not a loop invariant*

ii) $r < 7n$ *not a loop invariant*

iii) $n - x = 7r$ *not a loop invariant*

iv) $7x - r = 7n$ *loop invariant*

v) $x \geqslant 0$ *loop invariant*

3.10.6 IA. 6

Exam june 2016, 9
State for each of the following algorithms their asymptotic runtime in Θ-notation as a function of n. The input n is a positive integer for all algorithms.

Algorithm 46 ALGORITHM1(n)

```
1: for i = 1 to n do
2:     j = i
3:     while j > 0 do
4:         j = j − 1
```

This will be n^2

Algorithm 47 ALGORITHM2(n)

```
1: j = 1
2: while i > n do
3:     i = i · 2
```

This will be $log(n)$

Algorithm 48 ALGORITHM3(n)

```
1: i = 1
2: while i < n do
3:     i = i · n
```

This will be $O(1)$

Algorithm 49 ALGORITHM4(n)

```
1: i = 1
2: while i < n do
3:     s = 0
4:     while s < n do
5:         s = s + i
6:     i = i · 2
```

This will be $n + n/2 + n/4 + n/8 \cdots + 1 = 2n - 1 = O(n)$

3.10.7 IB. 1

Exam june 2012, 6
For $n \geqslant 1$, the integer logarithm is the value $\lfloor \log n \rfloor$. This is the largest power of two that does not exceed n (i.e., the integer k for which $2^k \leqslant n < 2^{k+1}$). The following is easily seen (and may be used without proof):

1) $\left\lfloor \log \frac{n}{2} \right\rfloor = \lfloor \log n \rfloor - 1$

2) $\lfloor \log(n-1) \rfloor = \lfloor \log n \rfloor$ when n is an odd integer

We wish to compute the integer logarithm of n for an arbitrary integer $n \geqslant 1$. Use the following algortitm for this.

Algorithm 50 INTEGERLOG(n)

```
1: k = 0
2: i = n
3: while i > 1 do
4:     if i%2 = 0 then
5:         i = i/2
6:         k = k + 1
7:     else
8:         i = i − 1
9: return k
```

a State the values of i and k at each test at the entry of the `while` loop in the algorithm above when the algorithm is run with input $n = 53$.

1 $i = 53$ and $k = 0$

2 $i = 52$ and $k = 0$

3 $i = 26$ and $k = 1$

4 $i = 13$ and $k = 2$

5 $i = 12$ and $k = 2$

6 $i = 6$ and $k = 3$

7 $i = 3$ and $k = 4$

8 $i = 2$ and $k = 4$

9 $i = 1$ and $k = 5$

b) Show that the following is an invariant for the `while` loop when the algorithm is started with input as an integer $n \geqslant 1$: When the test at the entry of the `while` loop is performed, the following holds

i) $\lfloor \log(i) \rfloor + k = \lfloor \log(n) \rfloor$

When taking the logarithm of i and n only difference is k.

$$\left\lfloor log_2\left(\frac{n}{2}\right)\right\rfloor = \lfloor log_2(n) \rfloor - k$$

For each iteration we know that $i = \frac{n}{2}$

$$\lfloor log_2(i) \rfloor = \lfloor log_2(n) \rfloor - k$$
$$\lfloor log_2(i) \rfloor + k = \lfloor log_2(n) \rfloor$$

ii) i is an integer

i is only divided when even, so no decimals

c) Argue for the correctness of the algorithm, i.e., that it terminates for all integers $n \geqslant 1$ and returns $\lfloor \log n \rfloor$.

Initial At initial where $i = n$ and $k = 0$ we know that:
$\lfloor \log(i) \rfloor + k = \lfloor \log(n) \rfloor \Leftrightarrow \lfloor \log(n) \rfloor + 0 = \lfloor \log(n) \rfloor$
Also since n is an integer i must be as well

Maintenance while the algorithm is running.

`while` $i > 1$
ensures termination at $i < 2$
`if` $i\%2 = 0$. Ensures that i stays an integer.
$i' = i/2$ and $k' = k+1$ so $\lfloor \log \frac{n}{2} \rfloor = \lfloor \log n \rfloor - 1$ stays true
`else` We know that $\lfloor \log(n-1) \rfloor = \lfloor \log n \rfloor$ when n is an odd integer
So since $i\%2 = 1$. $i' = i - 1$ ensures that i becomes even

Termination When out of while
Termination because $i = 1$. $\lfloor \log 1 \rfloor + k = \lfloor \log n \rfloor$

d) Give an analysis of the asymptotic running time (as a function of n) for the algorithm. time complexity as a function of n must be $f(n) = 2 \cdot \log n$ which is $\Theta(\log n)$

3.10.8 IB. 2

Exercise 4.2-5 (p 90)
The product of two complex numbers $a + bi$ and $c + di$ is $(ac - bd) + (ad + bc)i$, i.e., the task is to compute $ac - bd$ and $ad + bc$ from a, b, c, and d, but using only three multiplications.
Note. The solution resembles Strassen's method (but is much simpler).

3.11 Opgaver Uge 18

3.11.1 IIA. 1

Exercise 14.1-3 (p 373)
Consider a modification of the rod-cutting problem in which, in addition to a price p_i for each rod length i, each cut incurs a fixed cost of c. The revenue associated with a solution is now the sum of the prices of the pieces minus the total cost of making the cuts. Give a dynamic programming algorithm to solve this modified problem.

Algorithm 51 CUT-ROD(p, n, c)

```
1: if n = 0 then
2:     return 0
3:     q = −∞
4: for i = 1 to n do
5:     q = max{q, p[i] + CUT-ROD(p, n − 1) − c
```

3.11.2 IIA. 2

Exercise 14.4-1 (p 399)
Determine an LCS of $< 1,0,0,1,0,1,0,1 >$ and $< 0,1,0,1,1,0,1,1,0 >$. $< 1,0, >$

3.11.3 IIA. 3

Exercise 14.4-2 (p 399)
Give pseudocode to reconstruct an LCS from the completed c table and the original sequences $X = (x_1; x_2, \ldots, x_m)$ and $= y(y_1; y_2, \ldots, y_m)$ in $O(m+n)$ time, without using the b table.

3.11.4 IIA. 4

Exam june 2010, 4

3.11.5 IIA. 4

Exam june 2013, 7