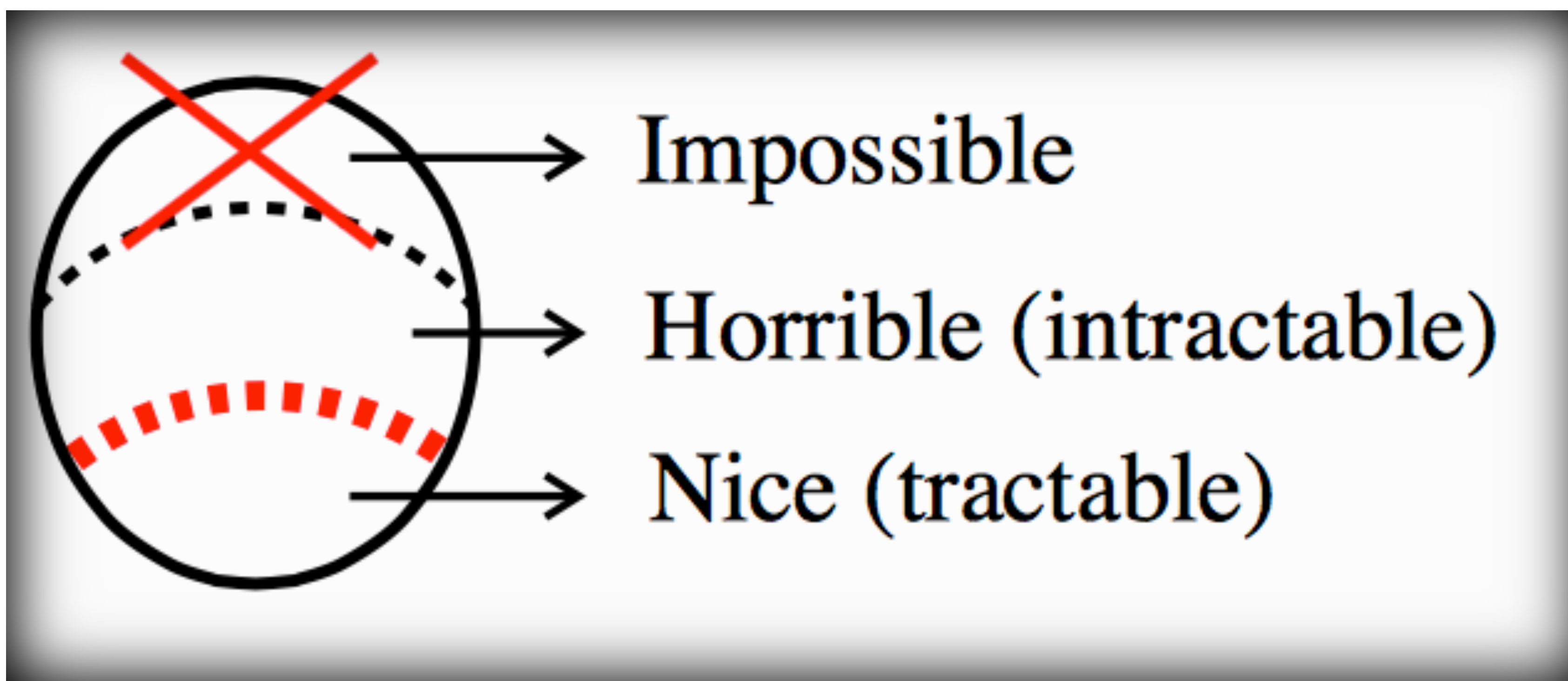
INF2220 Review

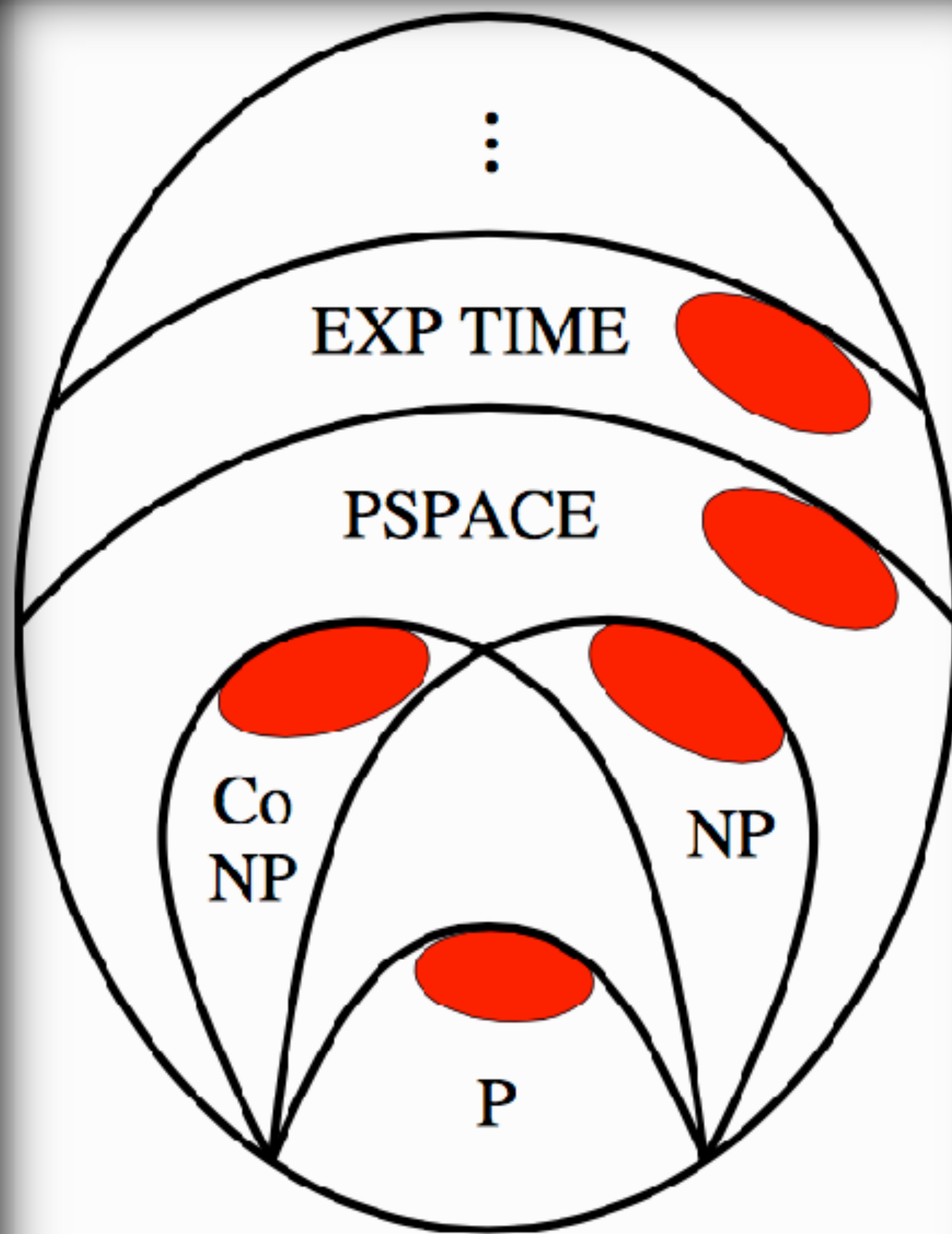
by Dino Karabeg

Remember what this is all about…

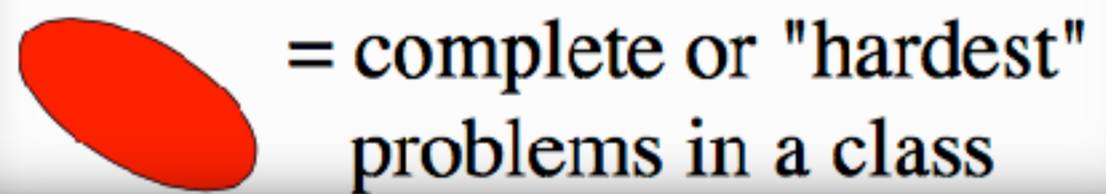
Recall the 3 kinds of complexity…

COMPLEXITY KIND 1


Impossible
Horrible (intractable)
Nice (tractable)



Map of classes



7f (vekt 2%)

Ved å representere problemer som formelle språk oppnår vi et systematisk og enhetlig syn på problemer og algoritmer.

Oppgave 7 Diverse oppgaver (vekt 10%)

Gi et kort svar, ikke mer enn 3 setninger, til hver av følgende spørsmål. Hvert spørsmål teller 2%.

(Fortsettes på side 12.)

1. Hvorfor bruker vi Big-O ($O(f(n))$) for å estimere kompleksitet av algoritmer? Hvorfor ikke bruke eksakte funksjoner, eller faktiske kjøretid?
2. Hva oppnår vi ved å dele problemer i kompleksitetsklasser?
3. Er uavgjørbarheten av Stoppe-problemet (*Halting problem*, det første problemet bevist å være uløsbart) bevist ved hjelp av reduksjon?
4. I hvilke situasjoner er det best å bruke utvidbar hashing?
5. Når sier vi at et problem er "intractable"?

Exercise 1: Assuming the alphabet {0, 1, ‘,’}, what is the formal language that corresponds to the (lexicographic) sorting problem?

Exercise 2: Correct the following statements and give a short explanation

a. A formal language is a formal way of speaking a conventional language such as English or Norwegian.
b. The Turing machine is a hardware computational device built by Alan Turing.
c. By reducing a problem we make it smaller.
d. We classify problems into classes based on how important they are.
e. Church’s thesis is Alonso Church’s Ph.D. dissertation.

Exercise 3:

a. Show that there are more real numbers than natural numbers, by diagonalization.
b. Show how the same technique may be applied to produce an alternative proof that the Halting problem is unsolvable.

COMPLEXITY
KIND 2

Oppgave 2 Kompleksitet (vekt 10%)

Hva er *worst-case* kompleksiteten av følgende implementasjoner:

2a For-løkker (vekt 3%)

```
for i = 1 to n {
  for j = i to n {
    for k = i to j {
        print (i, j, k);
    }
  }
}
```

1a Time complexity (weight 4%)

What is the *worst-case* complexity of the following piece of code, in terms of the input parameter n?

```
int z = 0;
for (int i = n; i >= 1; i = i/2) {
  for (int j = 1; j <= n ; j++) {
      z = z + i + j ;
  }
}
```

2b Rekursjon (vekt 7%)

```
float foo(A) {
  n = A.length;
  if (n==1) {
    return A[0];
  }
  let A1,A1,A3,A4 be arrays of size n/2
  for (i=0; i <= (n/2)-1; i++) {
    for (j=0; j <= (n/2)-1; j++) {
      A1[i] = A[i];
      A2[i] = A[i+j];
      A3[i] = A[n/2+j];
      A4[i] = A[j];
    }
  }
  b1 = foo(A1);
  b2 = foo(A2);
  b3 = foo(A3);
  b4 = foo(A4);
}
```

Solution: $\mathcal{O}(N^2\, logN + N^2) = \mathcal{O}(N^2\, log\, N)$

Oppgave 4 Binærtre (vekt 8%)

Et komplett binærtre med N elementer er lagret i en array, i posisjon 1 til N. Hvor stor må arrayen være for

1. et binærtre med to ekstra nivåer?
2. et binærtre med N noder hvor den dypeste noden er på dybde $2 \log N$?

Solution:

1. Når vi legge til et nivå, dobler vi antall node-plasser. 2 ekstra nivåer krever 4 ganger antall plasser
2. Et komplett binærtre har dypeste noden på $log\ N$, $2^{log\ N} \approx N$. Arrayen blir N^2 stor siden $N * N \approx (2^{log\ N})(2^{log\ N}) = 2^{2\ log\ N}$. (Den eksakte relasjonen er $2^{log\ (N+1)} = N + 1$)

7a (vekt 2%)

Vi vet at vi kan finne et element i en liste med n elementer på *O(log n)* tid ved å holde listen sortert og gjøre et binærsøk. Vi vet også at vi kan sortere n elementer på *O(n log n)* tid. Men vi kan aldri vite hvorvidt dette er den ideelle ytelsen vi kan oppnå og hvorvidt det er mulige å lage bedre algoritmer.

Oppgave 2 Binære søketrær (vekt 15%)

Fullfør de to programsegmentene under slik at de implementerer følgende oppgaver korrekt:

2a Finn noder i søketrær (vekt 5%)

Gitt et binært søketre og et heltall x. Metoden `find` skal returnere noden V som har verdi lik x (du kan forutsette at den finnes). Fullfør de tre kodelinjene som

(Fortsettes på side 3.)

starter med `return`.

```
class BinNode {
  int value;
  BinNode leftChild;
  BinNode rightChild;

  BinNode find(int x) {
    if (value == x) {
      return ........;
    }
    else if (value < x) {
      return ........;
    }
    else {
      return ........;
    }
  }
}
```

Solution:

```
class BinNode {
  int value;
  BinNode leftChild;
  BinNode rightChild;

  BinNode find(int x) {
    if (value == x) {
      return this;
    }
    else if (value < x) {
      return rightChild.find(x);
    }
    else {
      return leftChild.find(x);
    }
  }
}
```

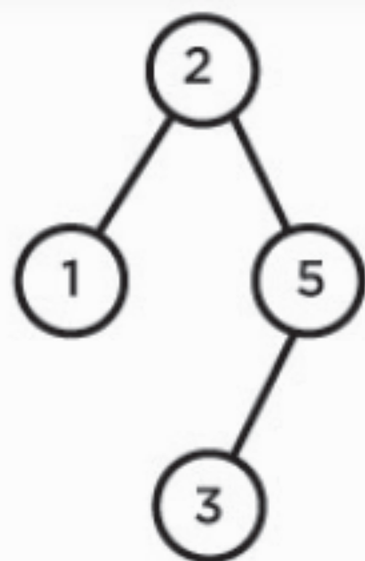
2b Rotering i søketrær (vekt 6%)

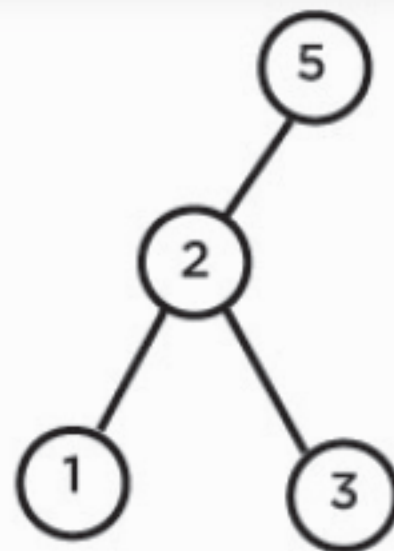
Under er koden til en del av et program som skal gjøre en node V til den nye rotnoden av et binært søketre ved gjentatte ganger å bytte V med foreldrenoden sin (se eksempelet i Figur 1). Koden nedenfor utfører kun et enkelt bytte. Du skal fullføre koden og sørge for at søketre-egenskapen opprettholdes. Du kan anta at V er den noden vi ønsker å bytte, og at P er foreldrenoden til V.

```
if (V == P.leftChild) {
    P.left = ..................;
     ................;
     ................;
}
```

(Fortsettes på side 4.)


```
else { // V is the right child of its parent P
     ................;
     ................;
     ................;
}
```



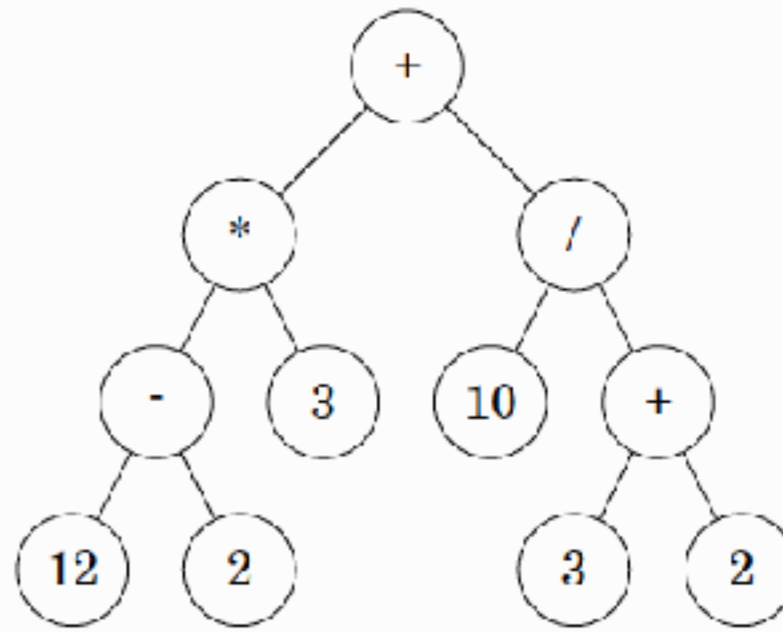


Figur 1: Noden med verdi 5 i figur **a** blir den nye rotnoden i figur **b**

Solution:

```
if (V == P.left)  {
  P.left = V.right;
  V.right = P;
}
else {
  P.right = V.left
  V.left = P
}
```

Oppgave 2 Uttrykkstrær og rekursjon (vekt 11%)



Figur 1

(Fortsettes på side 3.)

Figur 1 viser et *uttrykkstre*. Et uttyksetre er en entydig måte å representere et regnestykke, der verdien til en node tilsvarer enten resultatet av operanden den innholder utført på nodens to barn eller verdien til tallet i noden.

2a Evaluering av uttrykkstreet (vekt 2%)

Evaluer uttrykkstreet i Fig. 1, og skriv resultatet.

Solution:

32

2b Rekursjon (vekt 7%)

Implementer en rekursiv metode med signaturen `public void evalTree (BinTree t)` som skal evaluere et uttrykstre. Programmet skal kombinere klassene `MyStack` og `BinTree` med *postorder traversering* av trær. Du kan anta at klassen metoden ligger i har initialisert en instanse av klassen `MyStack` med tilstrekkelig størrelse med navn `stack`.

```
public class MyStack {
    private int maxSize;
    private int[] stackArray;
    private int top;
    public MyStack(int s) {
        maxSize = s;
        stackArray = new int[maxSize];
        top = -1;
    }
    public void push(int j) {
        stackArray[++top] = j;
    }
    public long pop() {
        return stackArray[top    ];
    }
}

class BinTree {
    String value;
    BinTree left;
    BinTree right;
}
```

En stack er en implementasjon av konseptet *LIFO datahåndtering*. Se for deg en stabel med tallerkener der du enten setter en tallerken på toppen av stabelen med *push* eller henter en tallerken fra toppen av stabelen med *pop*. Verdiene som skal lagres i treet er representert med strenger og inneholder enten en av de aritmetiske operatorene "+" , "-" , "*" og "/ " eller en tallverdi. Programmet skal fungere som følger: Når en node inneholder en streng med et tall så skal integerverdien til strengen pushes på stacken. Når en node inneholder en operand så skal man først evalure verdien til de to barna før man poper høyre- og venstrebarnets verdier av stacken og utfører operatoren på de to operandene.

Resultatet av verdien skal pushes på stacken. Når programmet har kjørt ferdig skal stacken kun inneholde verdien uttrykkstreet evaluerer til.

Solution:

Solution:

```
class EvalClass {
   private MyStack stack; //the stack is initalized somewhere else

   ...

   public void evalTree(BinTree t){
      int op1;
      int op2;
      int result;

      if (t == null) {
         return;
      }
      evalTree(t.left);
      evalTree(t.right);
      if (value.equals("+")) {
         op2 = stack.pop();
         op1 = stack.pop();
         result = op1 + op2;
      }
      else if (value.equals(" ")) {
         op2 = stack.pop();
         op1 = stack.pop();
         result = op1 - op2;
      }
      else if (value.equals("*")) {
         op2 = stack.pop();
         op1 = stack.pop();
         result = op1 * op2;
      }
      else if (value.equals("/")) {
         op2 = stack.pop();
         op1 = stack.pop();
         result = op1 / op2;
      }
      else {
         result = Integer.parseInt(t.value));
      }
      stack.push(result);
   }
}
```

7c (vekt 2%)

Man kan gjøre søk mer effektivt med hashing enn det som er mulig med binærsøk og binære søketrær

7d (vekt 2%)

Dobbel hashing er brukt for å unngå sekundære kollisjoner (secondary clustering).

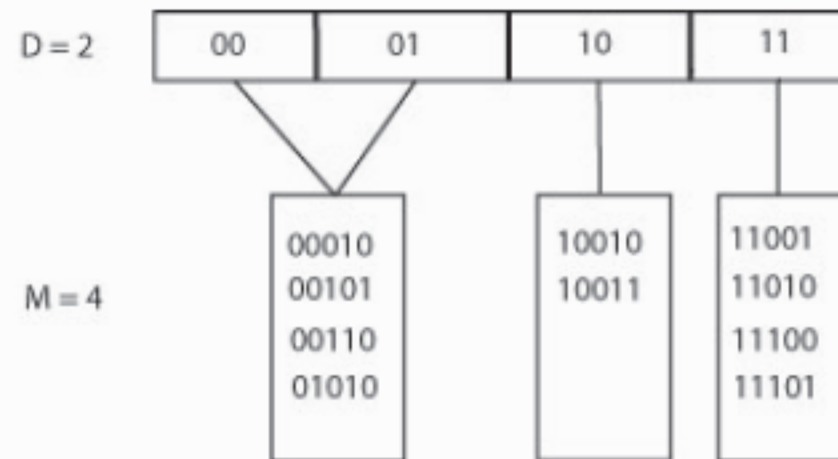
COMPLEXITY
KIND 3

7b (vekt 2%)

B-trær tillater oss å ha en mer effektiv bruk av lagringsplass, men det er ingen forskjell i kjøretiden til b-trær sammenlignet med andre balanserte søketrær.

Oppgave 4 Utvidbar hashing (vekt 10%)

Figuren under viser en utvidbar hashtabell



Figur 3: En utvidbar hashtabell

1. Vis hashtabellen du får ved å sette inn **00000** i Figur 3?
2. Vis hashtabellen du får ved å sette inn **11111** i Figur 3?

Solution: Solution should be obvious: a) splits the first block, b) splits the index and then the block accordingly.