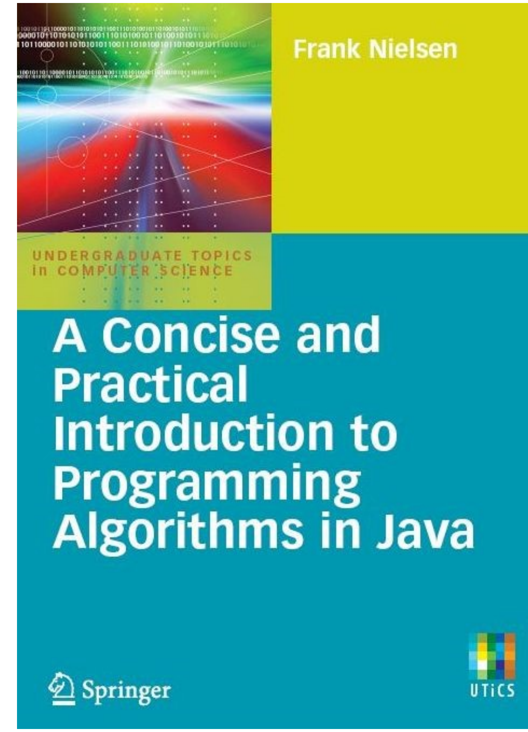
A Concise and Practical Introduction to Programming Algorithms in Java



Chapter 9: Combinatorial optimization algorithms

Frank NIELSEN

nielsen@lix.polytechnique.fr

Agenda

A few algorithms and paradigms:

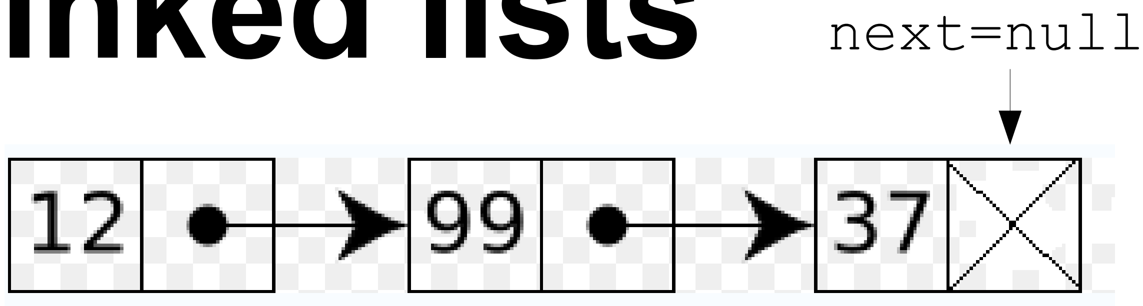
- I/ **Exhaustive search**

- II/ **Greedy algorithm** (set cover problems)

- III/ **Dynamic programming** (knapsack)

+...*Merging two lists*...

Linked lists




```
public class List
{
int container;
List next;  // a reference to a cell of a list

// Constructor  List(head, tail)
// Build a cell so that
//          the reference to the next cell is the tail
List(int element, List tail)
    {
    this.container=element;
    this.next=tail;
    }

}
```

Merging ordered linked lists

- Two *linked lists* `u` and `v` of increasing integers
- Build a *new linked list* in increasing order...
- ...using **only cells** of `u` and `v` (no cell creation, `new`)

For example:

```
U           | 3-->6-->8-->null
V           | 2-->4-->5-->7-->9-->null
Merge(U,V)  | 2-->3-->4-->5-->6-->7-->8-->9-->null
```

Linked lists

```
class List
{
int container;
List next;

// Constructor  List(head, tail)
List(int element, List tail)
    {
    this.container=element;
    this.next=tail;
    }

List insert(int el)   // insert element at the head of the list
{
return new List(el,this);
}

void Display()
{
List u=this;

while(u!=null)
    {
    System.out.print(u.container+"-->");
    u=u.next;
    }
System.out.println("null");
}
```

```
U     | 3-->6-->8-->null
V     | 2-->4-->5-->7-->9-->null
```

```java
class MergeList
{
//
// Merge two ordered lists
//
static List mergeRec(List u, List v)
{
if (u==null) return v;
if (v==null) return u;

if (u.container < v.container)
    {
    // Recycle cells/ no new
    u.next=mergeRec(u.next,v);
    return u;
    }
else
    {
    // Recycle cells/no new
    v.next=mergeRec(u,v.next);
    return v;
    }

}
```

```java
public static void main(String [] args)
{
List u=new List(8,null);
u=u.insert(6);u=u.insert(3);
u.Display();

List v=new List(9,null);
v=v.insert(7);v=v.insert(5);
v=v.insert(4);v=v.insert(2);
v.Display();

List w=mergeRec(u,v);
w.Display();
}
```

```java
static List sortRec(List u)
{
int i,l=u.length(), lr;
List l1, l2, split, psplit; // references to cells

if (l<=1)
   return u;
else
   {
   l1=u;

   psplit=split=u;
   i=0;lr=l/2;


   while (i<lr)
      {i++;
      psplit=split;
      split=split.next;}

   l2=split; // terminates with a null
   psplit.next=null;

   return mergeRec( sortRec(l1), sortRec(l2) );
   }
}
```

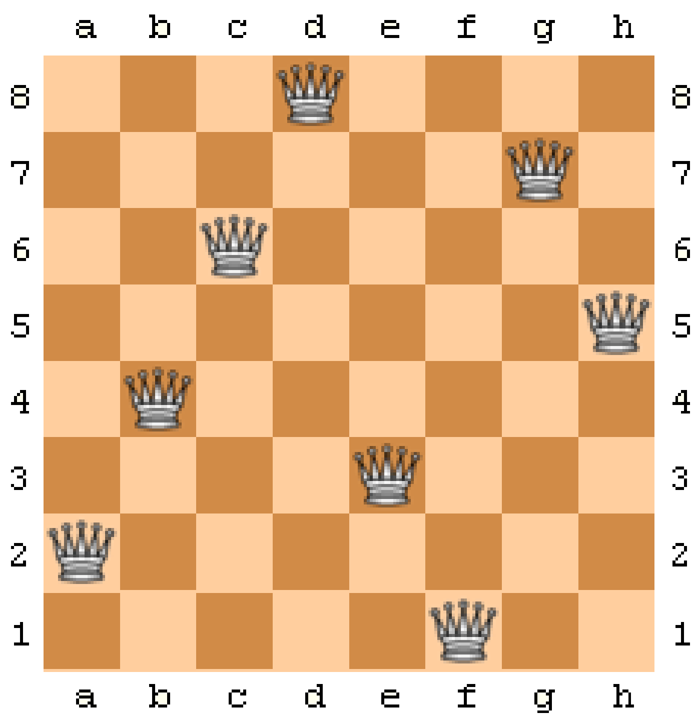
Sort in O(n log n) time:
- Split list in two halves
- Recursively apply sorting
- Merge ordered lists

```
public static void main(String [] args)
{

List u=new List(3,null);
u=u.insert(2);
u=u.insert(9);
u=u.insert(6);u=u.insert(1);
u=u.insert(15);u=u.insert(17);
u=u.insert(23);u=u.insert(21);
u=u.insert(19);u=u.insert(20);

u.Display();

List sortu=sortRec(u);
System.out.println("Sorted linked list:");
sortu.Display();
}
```

```
20-->19-->21-->23-->17-->15-->1-->6-->9-->2-->3-->null
Sorted linked list:
1-->2-->3-->6-->9-->15-->17-->19-->20-->21-->23-->null
```

Exhaustive search (Brute force search)

The Eight Queens puzzle:



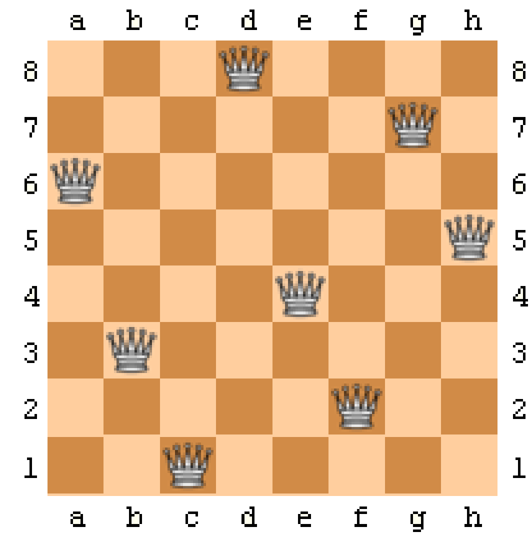
Max Bezzel (1848, chess player)
Find safe positions of 8 queens
on a 8x8 chessboard

- 92 distinct solutions
- 12 non-naive distinct solutions (rotation/symmetry)
- Good exercise for *designing* algorithms
- Generalize to n-queens

Exhaustive search & Backtracking

Brute force (naive) algorithm:
Check all *64x63x...x57/8! = 283,274,583,552* ?! solutions...

Easy to check that a **configuration** is not safe
(*check horizontal/vertical/diagonal lines*)



→ Two queens cannot be on the same line...

Therefore, *incrementally place queen* i (0...7)
on the i-th row, on the first **free** column

If there is no more free columns left, then **backtrack...**

Exhaustive search & Backtracking

Incrementally place queen i (0...7) on the first **free** column
If there is no more free columns left, then **backtrack:**

Consider the previous queen position and increment its column position, etc., etc., etc.

... until we find a solution
(=reach a successful location for queen indexed 7)

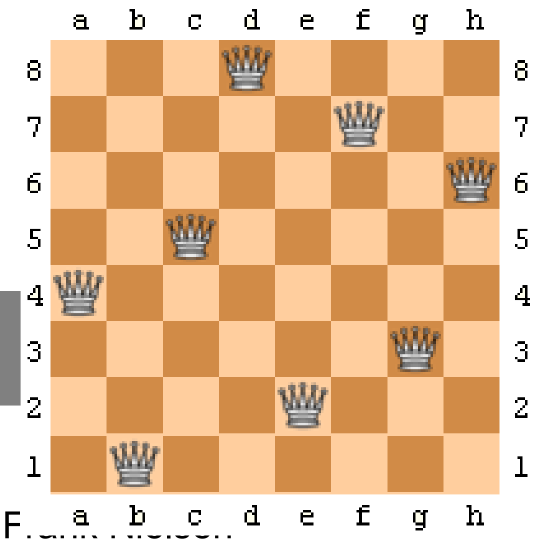
`queen`: 1D Array that specifies the column position
Queen i is located at position `(i,queen[i])`
(with i ranging from 0 to 7)

`search`: Static function that returns a/all solution(s).

```java
// Are queen (i1,j1) and queen (i2,j2) safe ?
static boolean WrongPos(int i1, int j1, int i2, int j2)
{ // same row?, same col?, same diag?
return (i1==i2 ||
        j1==j2 ||
        Math.abs(i1-i2) == Math.abs(j1-j2));
}

// Place safely queen i at column j?
static boolean FreeMove(int i, int j)
{
boolean result=true;

for(int k=0;k<i;k++)
   result=result&&!WrongPos(i,j,k,queen[k]);

return result;
}
```

Check for the queens placed so far on the chessboard

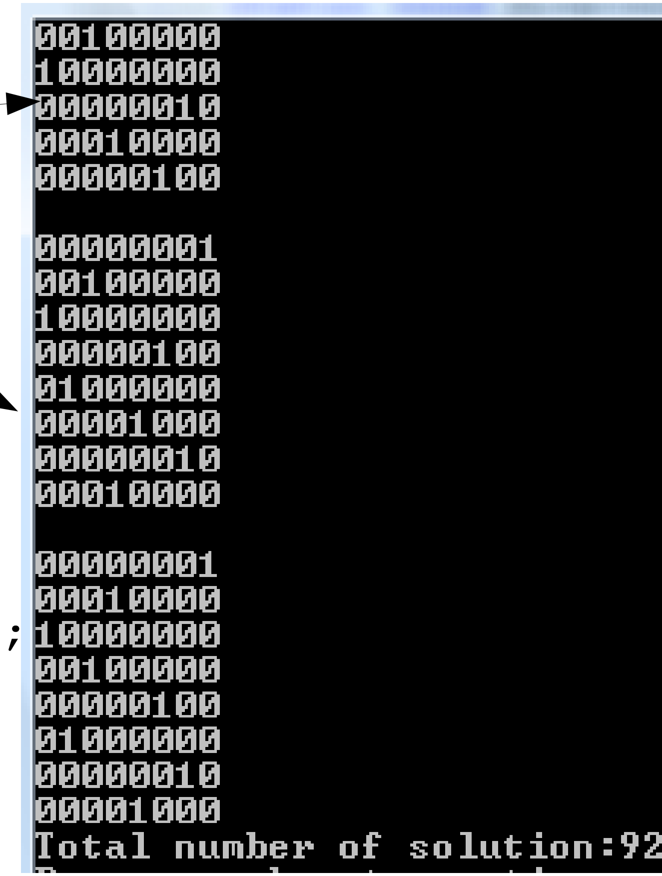
Static functions to check for collisions



```java
static boolean search(int row)
{
boolean result=false;

if (row==n)
   {// Terminal case
    DisplayChessboard();
    nbsol++;
   }
   else
   {// Exhaustive search
   int j=0;

      while(!result && j<n)
      {
         if (FreeMove(row,j))
         {
            queen[row]=j;
            result=search(row+1); // RECURSION/BACKTRACK
         }
         j++; // explore all columns
      }
    }
   return result;
```

Increment the number of found solutions (static class variable)

```
static final int n=8;
static int [] queen=new int[n];
static int nbsol;

static void DisplayChessboard()
{
int i,j;

System.out.println("");

for(i=0;i<n;i++)
    {
        for(j=0;j<n;j++)
        {
            if (queen[i]!=j) System.out.print("0");
                else System.out.print("1");
        }
        System.out.println("");
    }
}
```



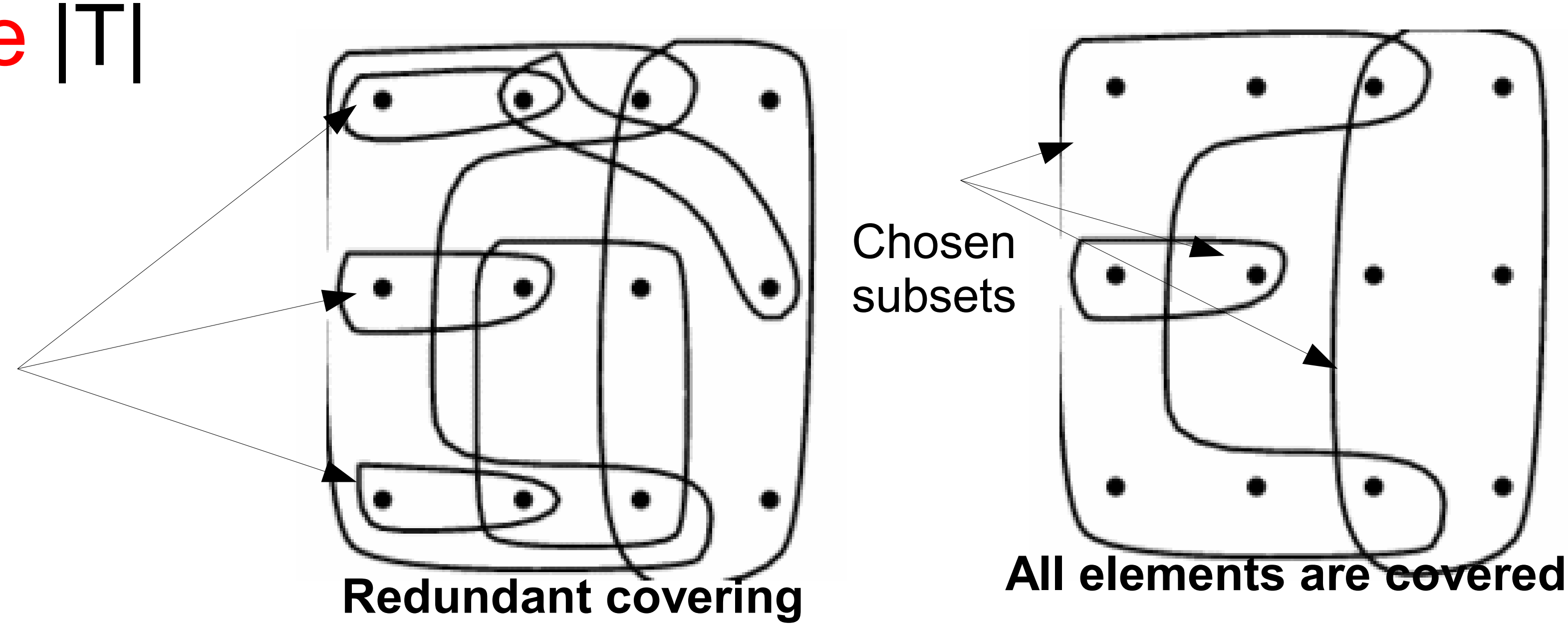

```
public static void main(String [] arguments)
{
nbsol=0;
search(0); // Call exhaustive search procedure
System.out.println("Total number of solutions:"+nbsol);
}
```

Optimization: Set Cover Problem (SCP)

- Given a graph:
  - A **finite set** X = {1, …, n}
  - A **collection of subsets** of S: $S_1, S_2, \ldots, S_m$

- **Problem**:
  - Find a subset T of {1, …, m} such that $\bigcup_{j \text{ in } T} S_j = X$
  - Minimize |T|

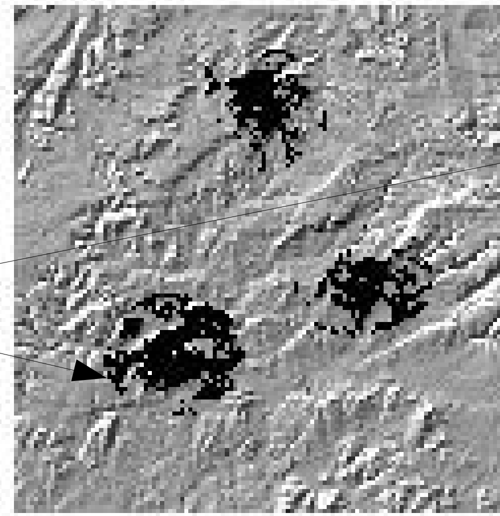
Elements of X



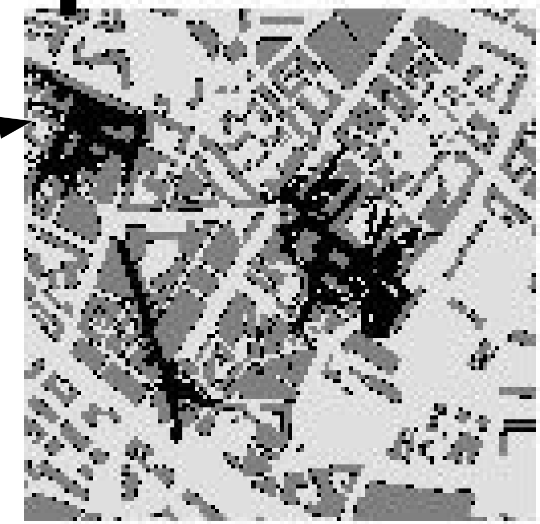
http://www.cs.sunysb.edu/~algorith/files/set-cover.shtml

Applications of set cover problems

Wave propagation patterns
of a single antenna



Mountains



City

- Choose base stations for quality of service
  (cell phone network, etc)

- Discretize terrain using a regular grid (set X)

- Each position of antenna -> subset of covered grid areas (S)

- Minimize the number of chosen antennas (X)

- Given a graph:
  - A **finite set** X = {1, …, n} (=regular grid elements)
  - A **collection of subsets** of S (=antenna patterns)

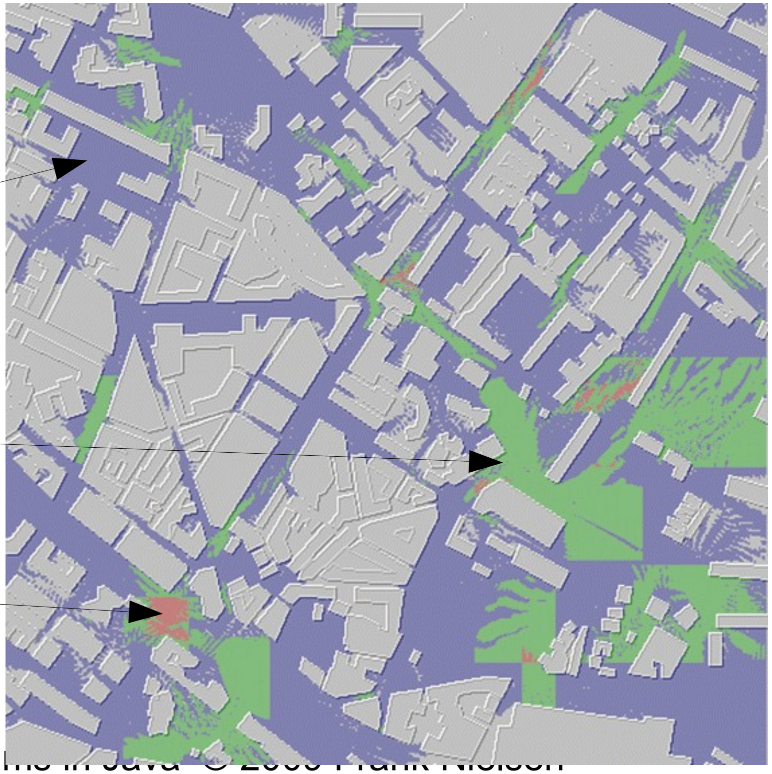
$$S_1, S_2, \ldots, S_m$$

- **Problem**:
  - Find a subset T of {1, …, m} (=subset of antennas) such that $U_{j \text{ in } T} S_j = X$
  - Minimize |T|

Covered once

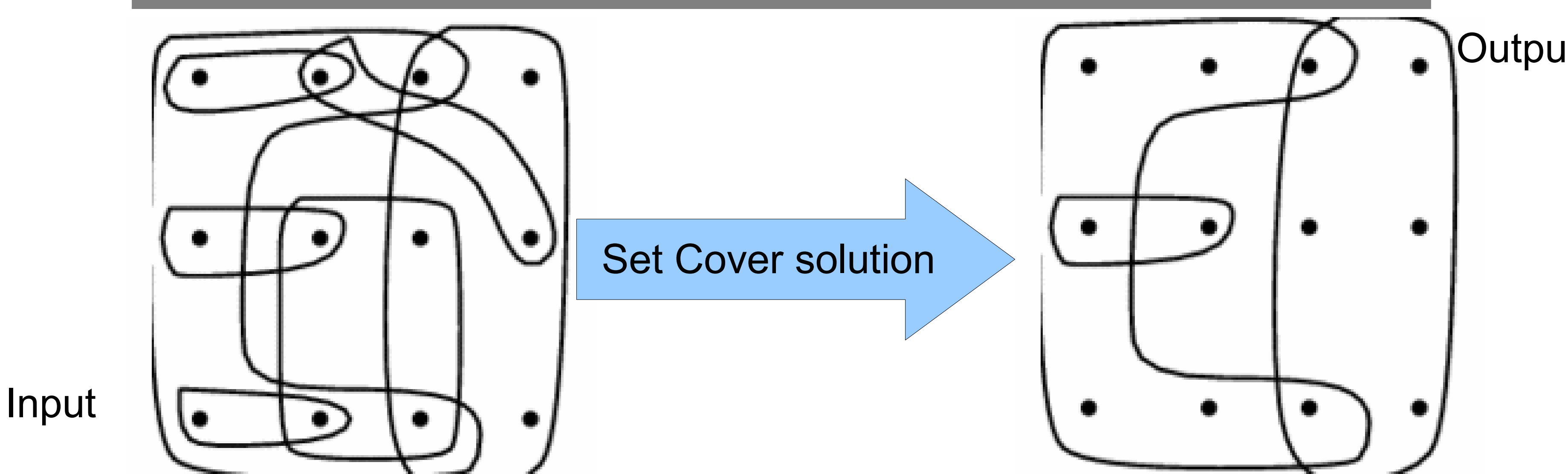
Covered twice

Covered thrice



A greedy algorithm (algorithme glouton)

```
GREEDY-SET-COVER(X,S)
M ← X        // all elements must be covered
C ← Ø        //  chosen subsets
while M ≠ Ø do
      select an Q ∈ S that maximizes |Q ∩ M|
        M ← M – Q
        C ← C U {Q}
return C
```



Visualizing a « range set »

<u>Elements:</u>

X = {1, …, 6}

<u>Subsets:</u>

S_1 = {1,2,4}

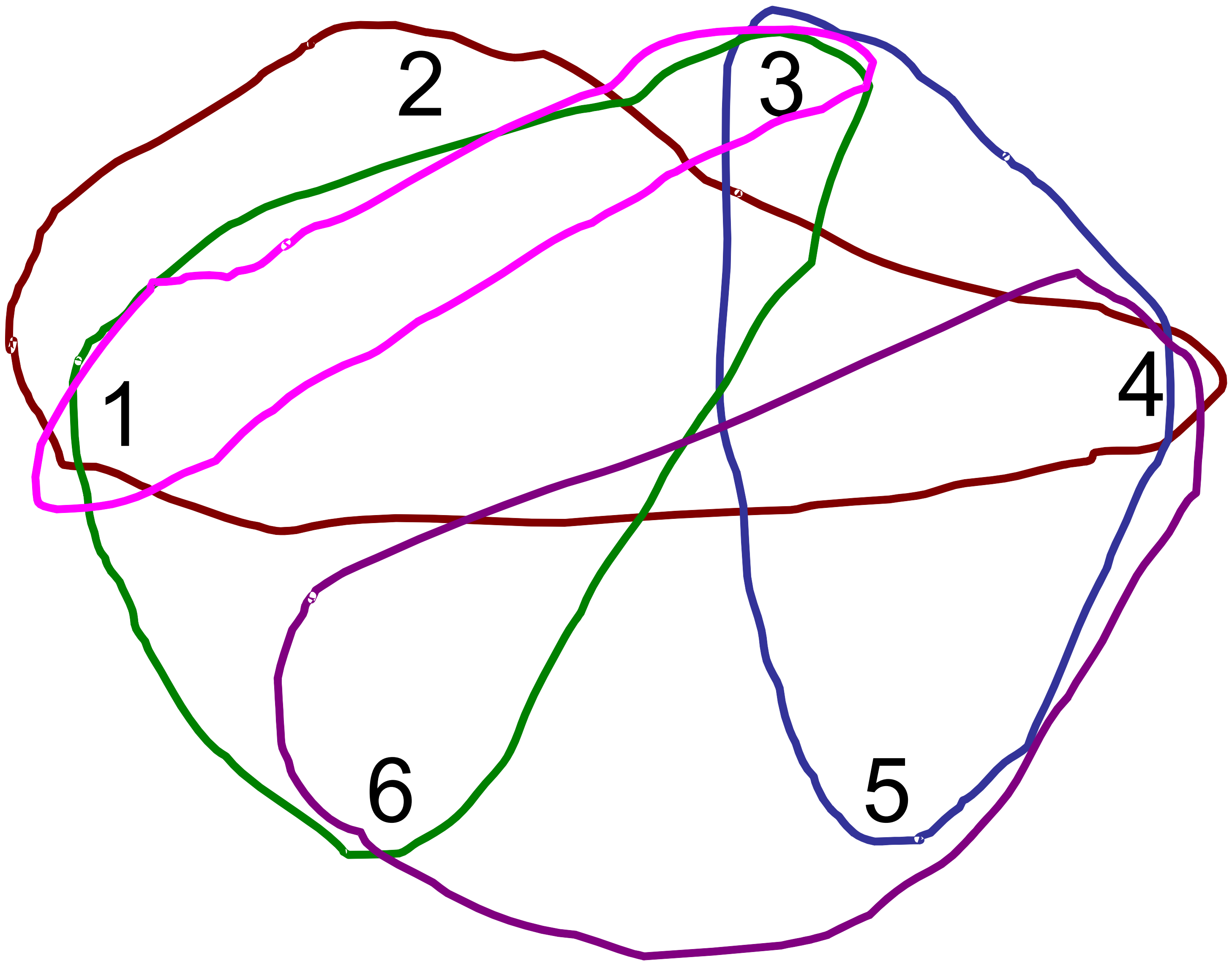
S_2 = {3,4,5}

S_3 = {1,3,6}

S_4 = {2,3,5}

S_5 = {4,5,6}

S_6 = {1,3}



Data-structure for the set cover problem

$X = \{1, \ldots, 6\}$

$S_1 = \{1,2,4\}$
$S_2 = \{3,4,5\}$
$S_3 = \{1,3,6\}$
$S_4 = \{2,3,5\}$
$S_5 = \{4,5,6\}$
$S_6 = \{1,3\}$

M=6 SUBSETS

$$
\begin{array}{cccccc|c}
1 & 2 & 3 & 4 & 5 & 6 & \\
\hline
1 & 1 & 0 & 1 & 0 & 0 & S_1 \\
0 & 0 & 1 & 1 & 1 & 0 & S_2 \\
1 & 0 & 1 & 0 & 0 & 1 & S_3 \\
0 & 1 & 1 & 0 & 1 & 0 & S_4 \\
0 & 0 & 0 & 1 & 1 & 1 & S_5 \\
1 & 0 & 1 & 0 & 0 & 0 & S_6
\end{array}
$$

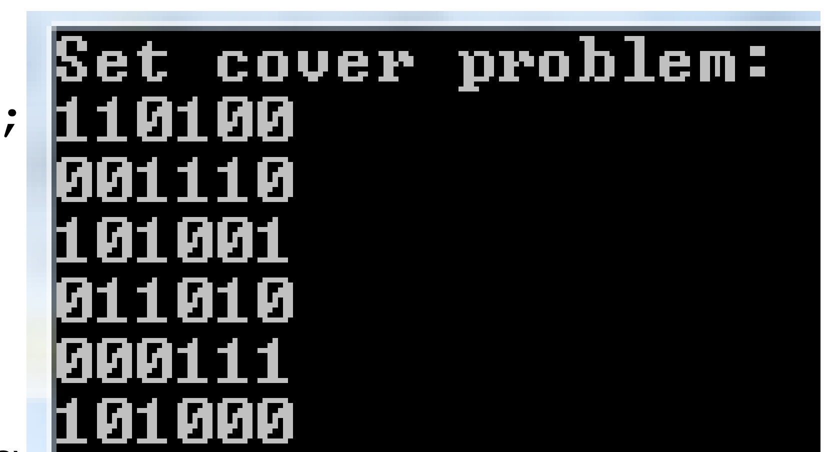
N=6 ELEMENTS

Incidence matrix: boolean matrix

```java
class SetCover
{
int nbelements;
int nbsubsets;
boolean [][] incidenceMatrix;

//Constructor
SetCover(int nn, int mm)
{
this.nbelements=nn; this.nbsubsets=mm;

incidenceMatrix=new boolean[nbsubsets][nbelements];

for(int i=0;i<nbsubsets;i++)
   for(int j=0;j<nbelements;j++)
      incidenceMatrix[i][j]=false;
}

void SetSubsets(int [] [] array) // Set incidence matrix
{for(int j=0;j<array.length;j++)
   {for(int i=0;i<array[j].length;i++)
         incidenceMatrix[j][array[j][i]]=true;
   }
}

}
```

```
void Display()
{
for(int i=0;i<nbsubsets;i++){

   for(int j=0;j<nbelements;j++)
      if (incidenceMatrix[i][j]) System.out.print("1");
         else System.out.print("0");
         System.out.println("");
         }
}
```

```
public static void main(String [] args)
{
int [][] subsets={{0,1,3},{2,3,4}, {0,2,5},{1,2,4},{3,4,5},{0,2}};

SetCover setcover=new SetCover(6,6);
setcover.SetSubsets(subsets);

System.out.println("Set cover problem:");
setcover.Display();
}
```



```java
static boolean [] GreedySCP(SetCover problem)
{
boolean [] result=new boolean[problem.nbsubsets];
int cover=0; int select;

for(int i=0;i<problem.nbsubsets;i++)  // initially no subsets
      result[i]=false;

while(cover!=problem.nbelements)
{
// Choose largest not-yet covered subset
select=problem.LargestSubset();
result[select]=true;

// Update covered matrix
cover+=problem.Cover(select);

// Update incidence matrix
problem.Update(select);

System.out.println("Selected "+select+" Number of covered
elements="+cover);
problem.Display();
}

return result;
}
```

Greedy algorithm

```java
// Number of covered element by subset i
int Cover(int i)
{
int nbEl=0;

for(int j=0;j<nbelements;j++)
      if (incidenceMatrix[i][j]) ++nbEl;

return nbEl;
}

// Report the current largest subset
int LargestSubset()
{
int i, nbel, max, select;

max=-1;select=-1;

for(i=0;i<nbsubsets;i++)
   {
   nbel=Cover(i);
   if (nbel>max) {max=nbel; select=i;}
   }

return select;
}
```

Methods of class SetCover

Methods of class SetCover

```
// Update the incidence matrix
void Update(int sel)
{
int i,j;

for(i=0;i<nbsubsets;i++)
{
   if (i!=sel) //use sel below so don't modify it
      {
      for(j=0;j<nbelements;j++)
         if (incidenceMatrix[sel][j])
            incidenceMatrix[i][j]=false;
      }
}
// Remove the chosen subset as well
for(j=0;j<nbelements;j++)
      incidenceMatrix[sel][j]=false;
}
```

```
public static void main(String [] args)
{
int [] [] subsets ={{0,1,3},{2,3,4},
                   {0,2,5},{1,2,4},
                   {3,4,5},{0,2}};
SetCover setcover=new SetCover(6,6);

setcover.SetSubsets(subsets);

System.out.println("Set cover problem:");
setcover.Display();

boolean [] solution=GreedySCP(setcover);

System.out.print("Solution:");
for(int i=0;i<setcover.nbsubsets;i++)
if (solution[i]) System.out.print(" "+i);
System.out.println("");
}
```

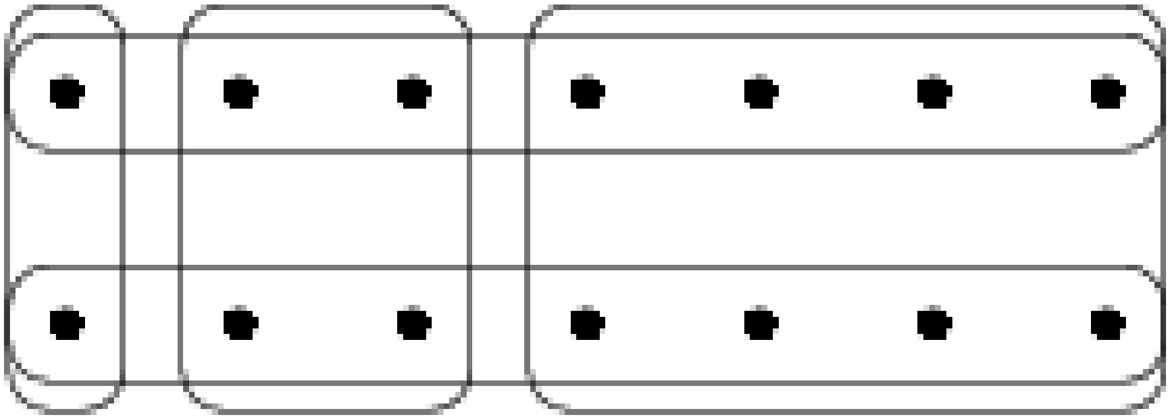
```
Set cover problem:
110100
001110
101001
011010
000111
101000
Selected 0 Number of covered elements=3
000000
001010
001001
001010
000011
001000
Selected 1 Number of covered elements=5
000000
000000
000001
000000
000001
000000
Selected 2 Number of covered elements=6
000000
000000
000000
000000
000000
000000
Solution: 0 1 2
```

Optimization bounds for greedy algorithm

CoptT<= Cgreedy <= ApproximationFactor x Copt

Upper bound:
Approximation factor is at most H(n)<=log(n)

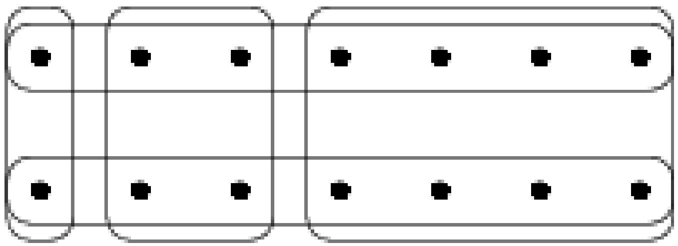
2 sets is optimal solution
but greedy chooses 3 sets here

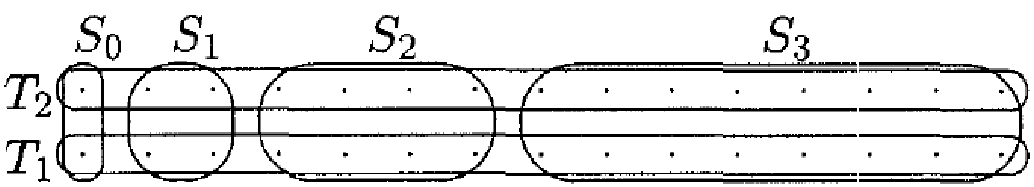


Lower bound:
Approximation factor can be as big as Omega(log(n))

Difficult to approximate: cannot beat (1-eps)Opt unless P=NP

```
int [] [] subsets={{0,1,2,3,4,5,6},{7,8,9,10,11,12,13},
                  {0,7},{1,2,8,9},{3,4,5,6,10,11,12,13}};
   SetCover setcover=new SetCover(14,5);
```





Etc...

Easy to build generic examples where greedy does not behave well with O(log n) approximation ratio

```
Set cover problem:
11111110000000
00000001111111
10000001000000
01100000110000
00011110001111
Selected 4 Number of covered elements=8
11100000000000
00000001110000
10000001000000
01100000110000
00000000000000
Selected 3 Number of covered elements=12
10000000000000
00000001000000
10000001000000
00000000000000
00000000000000
Selected 2 Number of covered elements=14
00000000000000
00000000000000
00000000000000
00000000000000
00000000000000
Solution: 2 3 4
```

Knapsack problem (sac a dos)

(First version)

Given:

- A set of n Objects `O1, ..., On` with corresponding weights `W1, ..., Wn`
- And a bag (knapsack) of capacity `W`

Find:

All the ways of choosing objects to **fully fill** the bag

Filling the knapsack

Need to enumerate all possibilities:

n objects => 2^n choices

(2, 4, 8, 16, 32, 64, ...)

How to program this?

n **is** a variable

(cannot fix the number of nest loops)

Need to enumerate all combinations:

= **Exhaustive search**

n=4
2^4=16

1 1 1 1
1 1 1 0
1 1 0 1
1 1 0 0
1 0 1 1
1 0 1 0
1 0 0 1
1 0 0 0
0 1 1 1
0 1 1 0
0 1 0 1
0 1 0 0
0 0 1 1
0 0 1 0
0 0 0 1
0 0 0 0

Enumerating: A recursive approach

```
static void Display(boolean [] tab)
{
for(int i=0;i<tab.length;i++)
   if (tab[i]) System.out.print("1 ");
      else
      System.out.print("0 ");

System.out.println("");
}

static void Enumerate(boolean [] selection, int pos)
{
if (pos==selection.length-1)
      Display(selection);
   else
   {
   pos++;
   selection[pos]=true;
   Enumerate(selection,pos);
   selection[pos]=false;
   Enumerate(selection,pos);
   }
}
```

```
public static void main(String[] args)
{
int n=4;
int i;
boolean [] select=new boolean[n];
for(i=0;i<n;i++)
   select[i]=false;

Enumerate(select,-1);
}
```

Fully filling the knapsack

```
final static int n=10; // 10 objects
static int [] weight={2,3,5,7,9,11,4,13,23,27};

static void SolveKnapSack(boolean [] chosen, int goal, int
i, int total)
{
numbercall++; // keep track of total number of calls

if ((i>=chosen.length)&&(total!=goal)) return;

if (total==goal)
    {  Display(chosen, goal);
           numbersol++; // total number of solutions
       }
else
    {
    chosen[i]=true;// add item first
    SolveKnapSack(chosen,goal,i+1,total+weight[i]);
    chosen[i]=false;  // and then remove it
    SolveKnapSack(chosen,goal,i+1,total);
    }
}
```

```java
final static int n=10; // 10 objects
static int [] weight={2,3,5,7,9,11,4,13,23,27};

public static void main(String [] args)
{
int totalweight=51;
numbersol=0;
numbercall=0;

System.out.println("Knapsack:");
boolean [] chosen=new boolean[n];

SolveKnapSack(chosen, totalweight, 0, 0);
System.out.println("Total number of
 solutions:"+numbersol);
System.out.println(" #calls="+numbercall);
}
```

```
Knapsack:
2+3+5+7+11+23+0=51
2+5+7+9+11+4+13+0=51
2+5+4+13+27+0=51
2+7+11+4+27+0=51
2+9+4+13+23+0=51
2+9+13+27+0=51
3+5+7+9+4+23+0=51
3+5+7+9+27+0=51
3+5+7+13+23+0=51
3+5+9+11+23+0=51
7+4+13+27+0=51
9+11+4+27+0=51
11+4+13+23+0=51
11+13+27+0=51
Total number of solutions:14
 #calls=2029
```

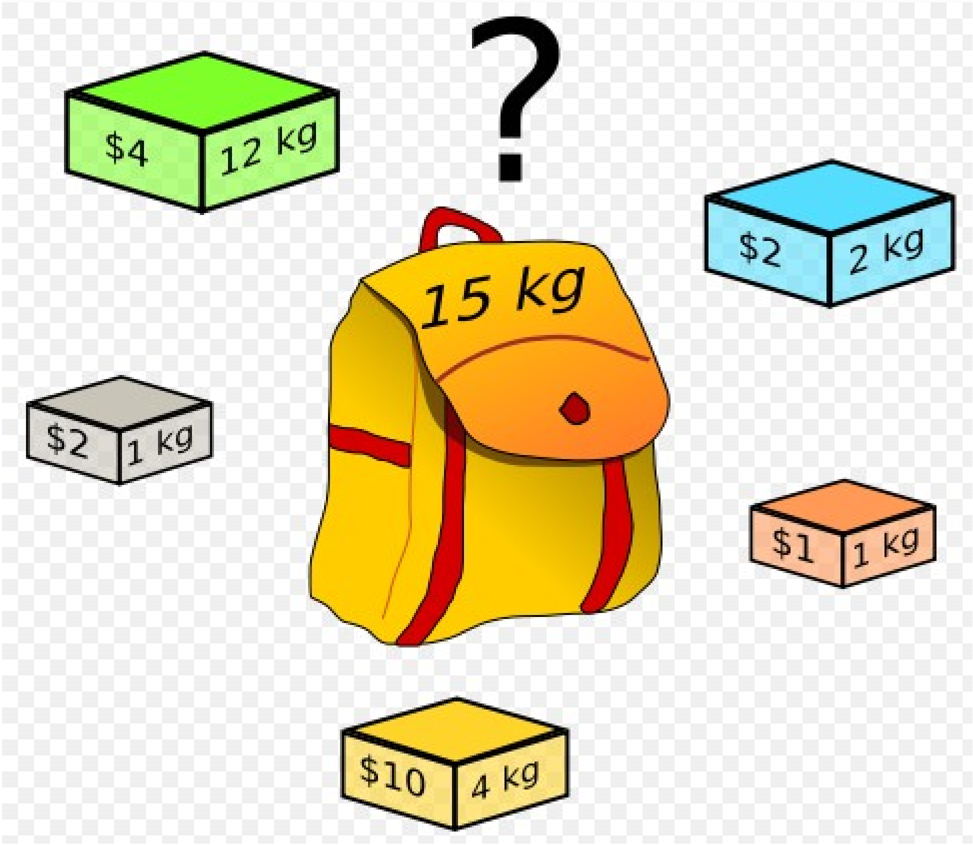
Exhaustive search: Branch & bound

```
static void SolveKnapSack(boolean [] chosen, int goal, int i,
int total)
{
numbercall++;

if (total>goal) return; // cut

if ((i>=chosen.length)&&(total!=goal)) return;

if (total==goal)
   {  Display(chosen, goal);
         numbersol++;
      }
else
   {
   chosen[i]=true;// add item first
   SolveKnapSack(chosen,goal,i+1,total+weight[i]);
   chosen[i]=false;  // and then remove it
   SolveKnapSack(chosen,goal,i+1,total);
   }

}
```

Stop recursion if we already exceed the weight amount

```
Knapsack:
2+3+5+7+11+23+0=51
2+5+7+9+11+4+13+0=51
2+5+4+13+27+0=51
2+7+11+4+27+0=51
2+9+4+13+23+0=51
2+9+13+27+0=51
3+5+7+9+4+23+0=51
3+5+7+9+27+0=51
3+5+7+13+23+0=51
3+5+9+11+23+0=51
7+4+13+27+0=51
9+11+4+27+0=51
11+4+13+23+0=51
11+13+27+0=51
Total number of solutions:14 #calls=1791
```

Knapsack: Fundamental optimization problem



Given a bag capacity (15 kg),
maximize the **utility** (price) of selected objects

(NP-hard problem)

Knapsack optimization problem

Given $p_1, p_2, \ldots p_n$ ← weights

$a_1, a_2, \ldots, a_n$ ← utility

$Pmax$ ← Maximum weight (capacity of bag)

Optimize $\sum_{i \in I} \quad a_i$

such that

$$\sum_{i \in I} p_i \quad \leq \quad Pmax$$

(Maybe there exists several solutions)

Knapsack: Example

$Pmax$ = 12

8 objects

weight	2	3	5	2	4	6	3	1
utility	5	8	14	6	13	17	10	4

Dynamic programming (Knapsack)

Dynamic programming computes a table...
... from which a solution can be retrieved.

Requires a relational equation to deduce
solutions progressively.

Dynamic programming (Knapsack)

Let u(i,j) be the maximum utility by
taking objects in {1, ...,i} with total weight <=j

- If i=1 then u(i,j)=0 for j<p1
  and u(i,j)=A1 for j>=P1

- If i>1
  u(i,j)=max(u(i-1,j) , u(i-1,j-Pi)+Ai)

Do not take object Oi

Take object Oi:
- gain Ai
- but leave room: Pi

weight	2	3	5	2	4	6	3	1
utility	5	8	14	6	13	17	10	4

Pmax= 12

u(i,j)=max(u(i-1,j) , u(i-1,j-Pi)+Ai)

$u_{i,j}$	0	1	2	3	4	5	6	7	8	9	10	11	12
$i=1$	0	0	5	5	5	5	5	5	5	5	5	5	5
$i=2$	0	0	5	8	8	13	13	13	13	13	13	13	13
$i=3$	0	0	5	8	8	14	14	19	22	22	27	27	27
$i=4$	0	0	6	8	11	14	14	20	22	25	28	28	33
$i=5$	0	0	6	8	13	14	19	21	24	27	28	33	35
$i=6$	0	0	6	8	13	14	19	21	24	27	30	33	36
$i=7$	0	0	6	10	13	16	19	23	24	29	31	34	37
$i=8$	0	4	6	10	14	17	20	23	27	29	33	35	38

Optimization result: Maximum utility, given Pmax

Reading *back:* Solution

2	3	5	2	4	6	3	1
5	8	14	6	13	17	10	4

$u_{i,j}$	0	1	2	3	4	5	6	7	8	9	10	11	12	
$i=1$	0	0	5	5	5	5	5	5	5	5	5	5	5	Choose (38=33+5, 2-2=0)
$i=2$	0	0	5	8	8	13	13	13	13	13	13	13	13	5=5 do not choose
$i=3$	0	0	5	8	8	14	14	19	22	22	27	27	27	5=5 do not choose
$i=4$	0	0	6	8	11	14	14	20	22	25	28	28	33	Choose (33=27+6,4-2=2)
$i=5$	0	0	6	8	13	14	19	21	24	27	28	33	35	Choose (27=14+13,8-4=4)
$i=6$	0	0	6	8	13	14	19	21	24	27	30	33	36	24=24, do not choose
$i=7$	0	0	6	10	13	16	19	23	24	29	31	34	37	Choose (14, 11-3=8)
$i=8$	0	4	6	10	14	17	20	23	27	29	33	35	38	Choose (4, 11)

Choose (Utility=0, Pmax=12)

From the table, chosen solution: O8, O7, O5, O4, O1

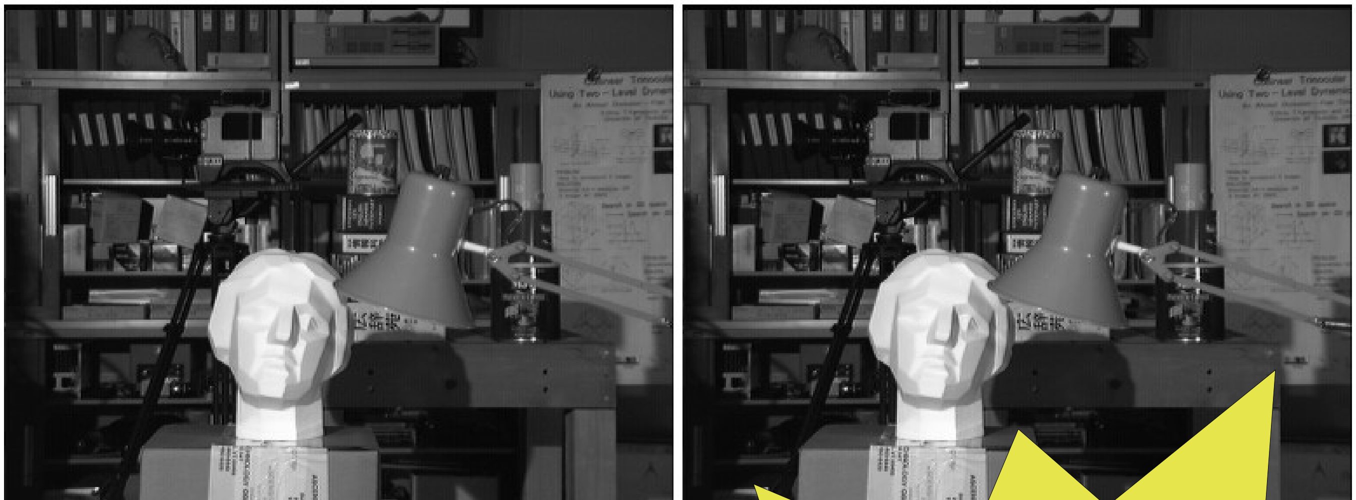
```java
static int nbObjects=8;
static int [] weight={2,3,5,2,4,6,3,1};
static int [] utility={5,8,14,6,13,17,10,4};
static int weightmax=12;
static int [] [] array;

static void SolveDP()
{
int i,j;
array=new int[nbObjects][weightmax+1];

// initialize the first row
for(j=0;j<=weightmax;j++)
   if (j<weight[0]) array[0][j]=0;
         else array[0][j]=utility[0];
// for all other rows
for(i=1;i<nbObjects;i++)
   {
   for(j=0;j<=weightmax;j++)
      if (j-weight[i]<0) array[i][j]=array[i-1][j];
       else
         array[i][j]=max( array[i-1][j],
                     array[i-1][j-weight[i]]+utility[i]);
   }
}
```

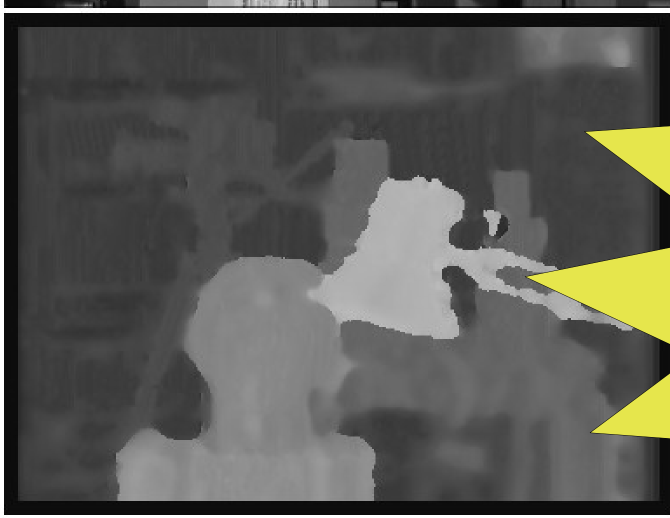
```
static void InterpretArray()
{
int i,u,w;
u=0;
w=weightmax;

for(i=nbObjects-1;i>=1;i--)
{
   if (array[i][w]!=array[i-1][w])
         {System.out.print((i+1)+" ");
         w=w-weight[i];
         u=u+utility[i];
         }
}

if (array[0][w]!=0);
   {System.out.println("1");
   w=w-weight[0];
   u=u+utility[0];
   }
System.out.println("Cross check:"+u+" remaining weight "+w);
}
```

```
public static void main(String[] args)
{
System.out.println("Solving knapsack using the dynamic
                programming paradigm.");
SolveDP();
Display();
System.out.println("Reading solution:");
InterpretArray();
}
```

```
Solving knapsack using the dynamic programming paradigm.
0       0       5       5       5       5       5       5       5       5       5       5       5

0       0       5       8       8       13      13      13      13      13      13      13      13

0       0       5       8       8       14      14      19      22      22      27      27      27

0       0       6       8       11      14      14      20      22      25      28      28      33

0       0       6       8       13      14      19      21      24      27      28      33      35

0       0       6       8       13      14      19      21      24      27      30      33      36

0       0       6       10      13      16      19      23      24      29      31      34      37

0       4       6       10      14      17      20      23      27      29      33      35      38

Reading solution:
8 7 5 4 1
Cross check:38 remaining weight 0
```

Dynamic programming: binocular stereo matching





Dynamic Programming

Benchmark:
http://vision.middlebury.edu/~schar/stereo/web/results.php

Optimization: A brief summary

- Exhaustive search: recursion but O(2^n) complexity

- Can be improved by backtracking (cuts)

- Greedy algorithm: Polynomial O(n^3)
- but yields an approximation

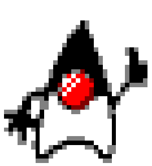
- Dynamic programming yields an exact solution
  but requires O(weightxobjects) time
  (weights should not be too bigs)

Last but not least: Java applets!

Applets are special java programs...
...that can run into your favorite Internet browser

You need to:
(1) write a web page with `<APPLET> </APPLET>` tags
(2) write and compile the Java applet code (`javac`)

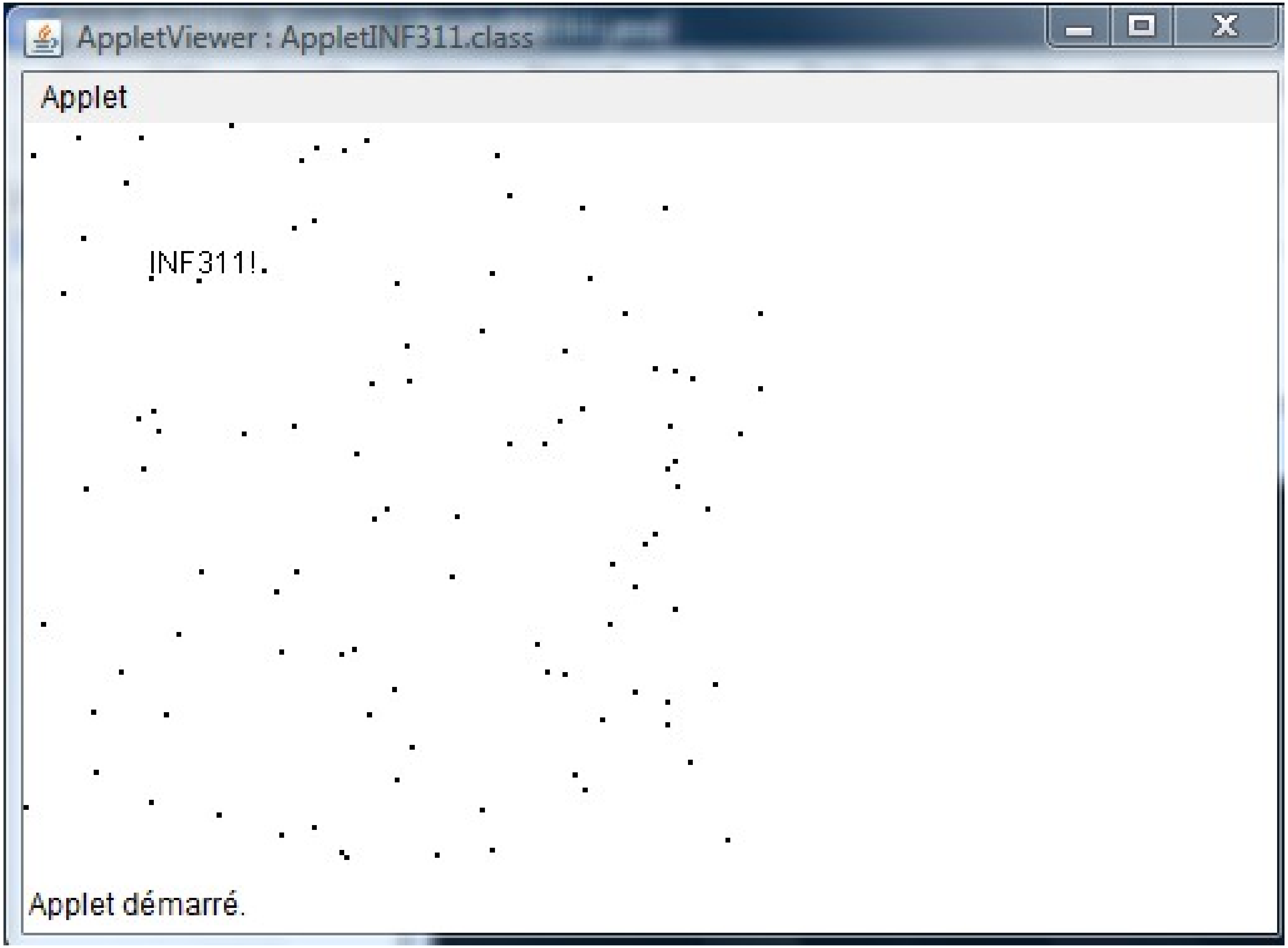
Advantages of applets are:
(1) to be accessible worldwide
(2) to provide graphics



```java
import java.awt.*;
import java.applet.*;

class Point2D
{double x,y;
Point2D()
{this.x=300.0*Math.random();
this.y=300.0*Math.random();}}

public class AppletINF311 extends Applet {
final static int n=100;
static Point2D [] set;

    public void init() {
        int i;
        set=new Point2D[n];
        for(i=0;i<n;i++)
            set[i]=new Point2D();}

    public void paint(Graphics g) {int i;

        for(i=0;i<n;i++)
        {
        int xi, yi;
        xi=(int)set[i].x; yi=(int)set[i].y;
        g.drawRect(xi, yi,1,1);
        }
        g.drawString("INF311!", 50, 60 );
    }
}
```

Java applets

```
<APPLET
    code   = "AppletINF311.class"
    width  = "500"
    height = "300"
    >
</APPLET>
```



Java applets for online demos...

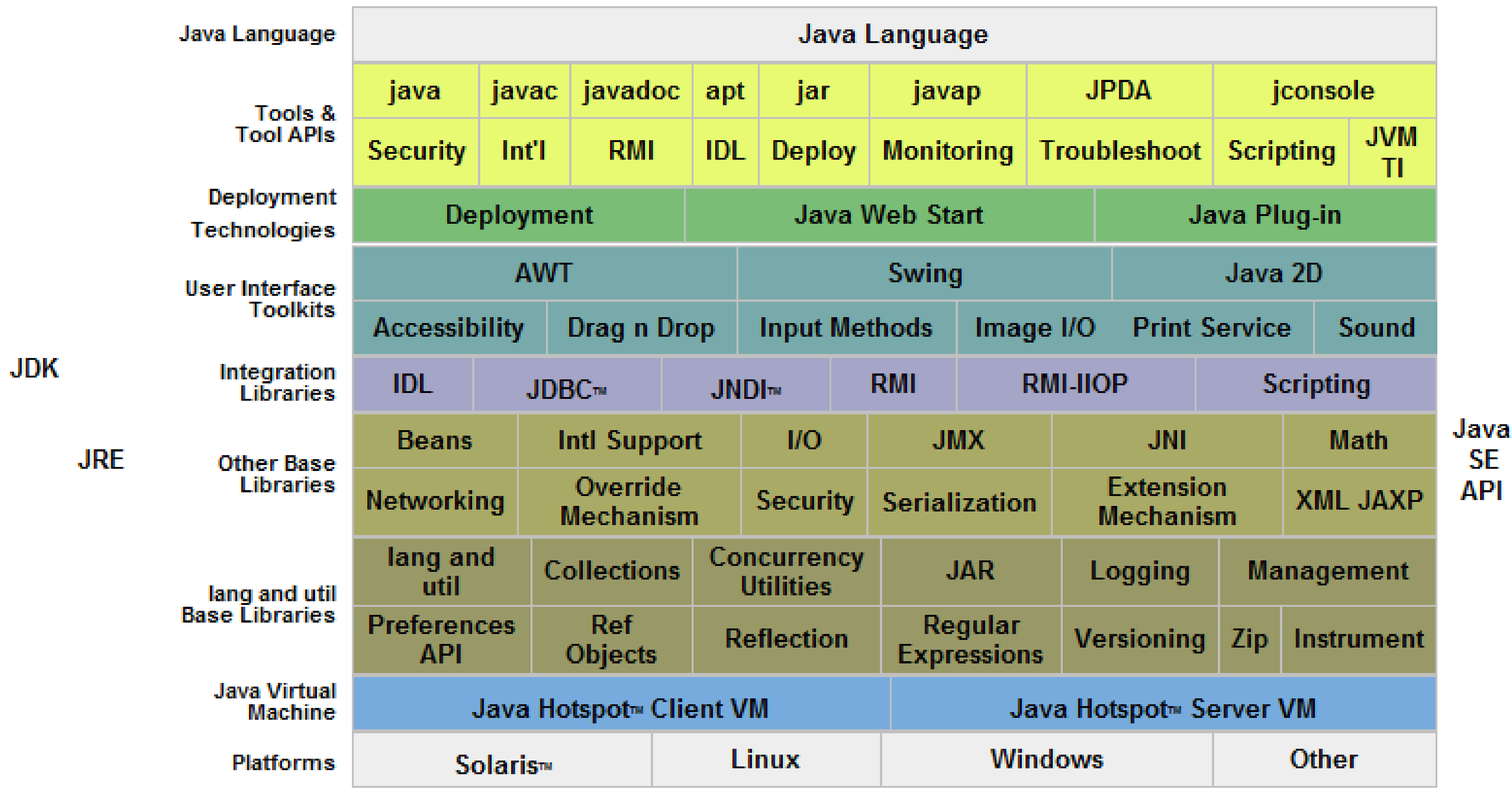
Hallucination (mirage)

Two technologies in use:
- Image segmentation
- Texture synthesis (hole filling)

Try it !!! http://www.sonycsl.co.jp/person/nielsen/ClickRemoval/

F. Nielsen, R. Nock ClickRemoval: interactive pinpoint image object removal. ACM Multimedia 2005

There is much more in Java!



JDK™ 6 Documentation

A glimpse at object inheritance

All objects inherit from the topmost object: `Object`
...meaning some methods are already **predefined**

java.lang
Class Object

`java.lang.Object`

`public class Object`

Class `Object` is the root of the class hierarchy. Every cl
the methods of this class.

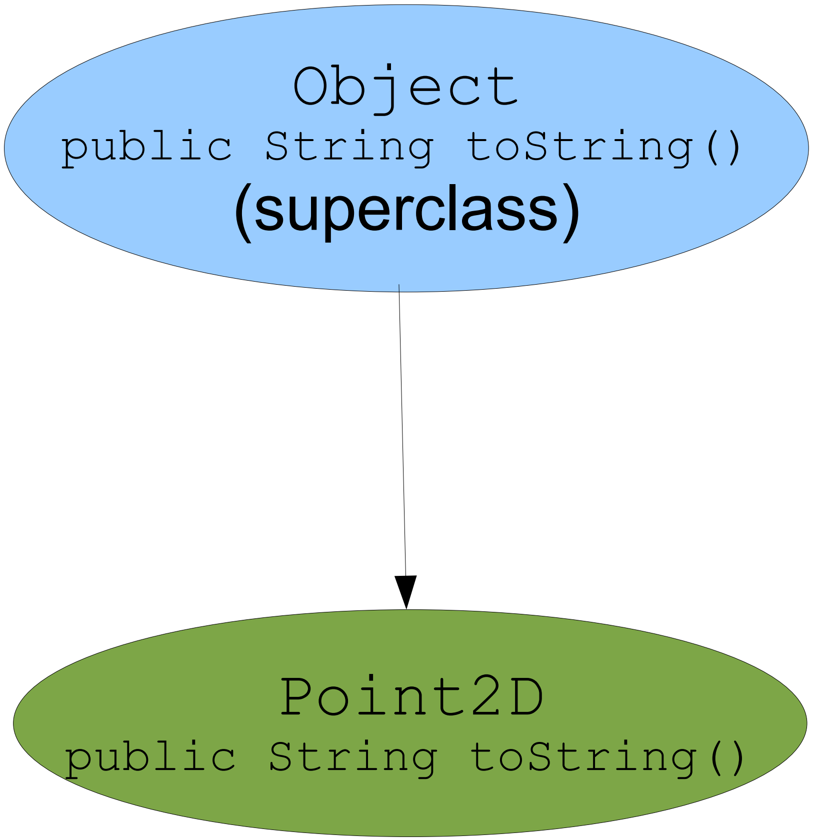
Method Summary

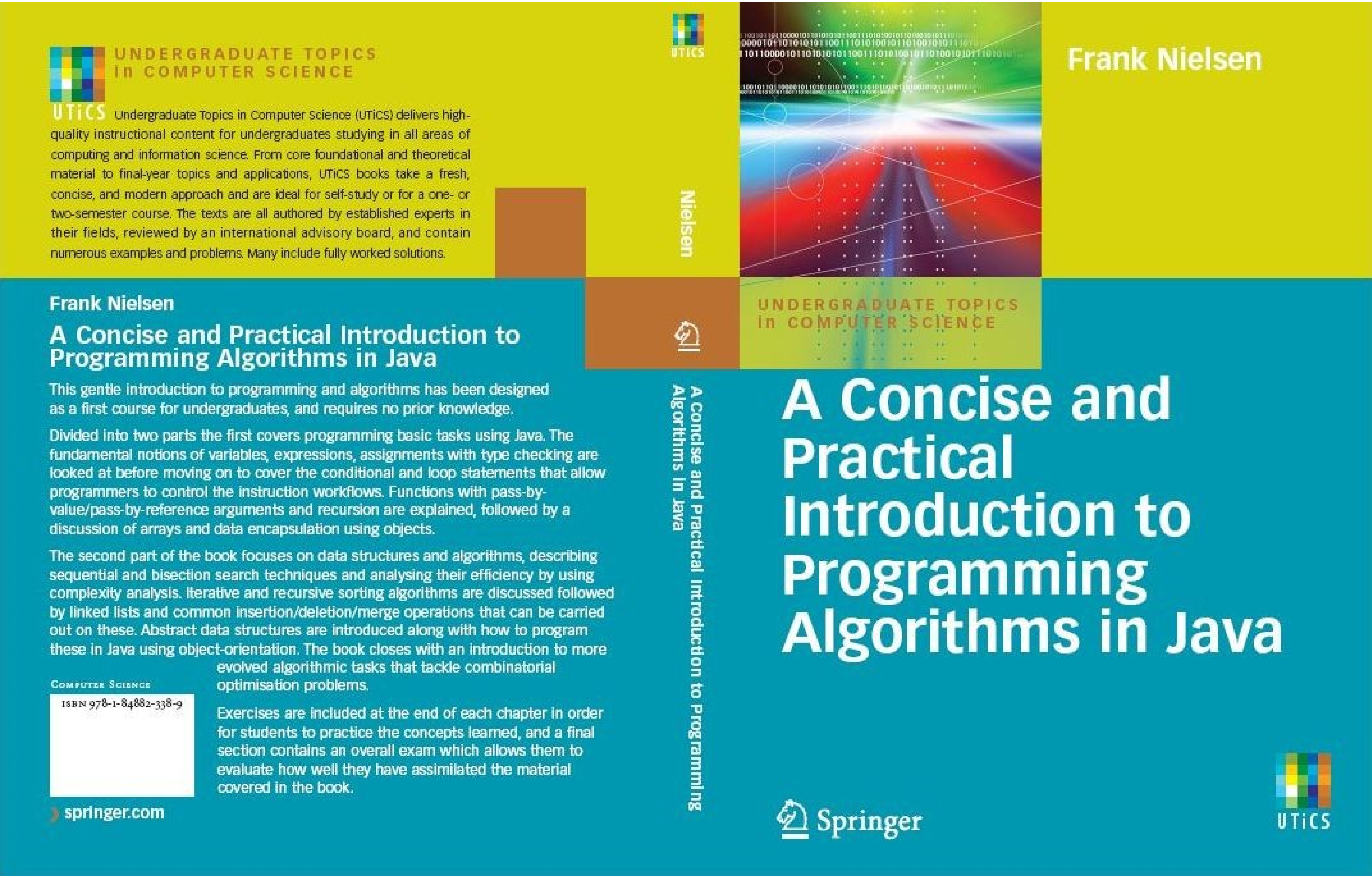
protected Object	clone() Creates and returns a copy of this object.
boolean	equals(Object obj) Indicates whether some other object is "equal to" this one.
protected void	finalize() Called by the garbage collector on an object when garbage collection determines that there are no more references to the object.
Class<?>	getClass() Returns the runtime class of this `Object`.
int	hashCode() Returns a hash code value for the object.
void	notify() Wakes up a single thread that is waiting on this object's monitor.
void	notifyAll() Wakes up all threads that are waiting on this object's monitor.
String	toString() Returns a string representation of the object.

Can overwrite the method `toString`

A glimpse at object inheritance

```
class Point2D
{double x,y;
// Constructor
Point2D(double xi, double yi)
{this.x=xi;this.y=yi; }
// Overrides default method
public String toString()
{return "["+x+" "+y+"]"; }
}

class SuperClass
{
public static void main(String [] a)
{
  Point2D myPoint=new Point2D(Math.PI, Math.E);
  System.out.println("Point:"+myPoint);
}
}
```




UNDERGRADUATE TOPICS in COMPUTER SCIENCE
UTiCS Undergraduate Topics in Computer Science (UTiCS) delivers high-quality instructional content for undergraduates studying in all areas of computing and information science. From core foundational and theoretical material to final-year topics and applications, UTiCS books take a fresh, concise, and modern approach and are ideal for self-study or for a one- or two-semester course. The texts are all authored by established experts in their fields, reviewed by an international advisory board, and contain numerous examples and problems. Many include fully worked solutions.
Frank Nielsen
A Concise and Practical Introduction to Programming Algorithms in Java
This gentle introduction to programming and algorithms has been designed as a first course for undergraduates, and requires no prior knowledge.
Divided into two parts the first covers programming basic tasks using Java. The fundamental notions of variables, expressions, assignments with type checking are looked at before moving on to cover the conditional and loop statements that allow programmers to control the instruction workflows. Functions with pass-by-value/pass-by-reference arguments and recursion are explained, followed by a discussion of arrays and data encapsulation using objects.
The second part of the book focuses on data structures and algorithms, describing sequential and bisection search techniques and analysing their efficiency by using complexity analysis. Iterative and recursive sorting algorithms are discussed followed by linked lists and common insertion/deletion/merge operations that can be carried out on these. Abstract data structures are introduced along with how to program these in Java using object-orientation. The book closes with an introduction to more evolved algorithmic tasks that tackle combinatorial optimisation problems.
Exercises are included at the end of each chapter in order for students to practice the concepts learned, and a final section contains an overall exam which allows them to evaluate how well they have assimilated the material covered in the book.
Computer Science
ISBN 978-1-84882-338-9
springer.com
UTiCS
Nielsen
A Concise and Practical Introduction to Programming Algorithms in Java
Frank Nielsen
UNDERGRADUATE TOPICS in COMPUTER SCIENCE
A Concise and Practical Introduction to Programming Algorithms in Java
Springer
UTiCS