UNIVERSITY OF SKÖVDE

Computation: Algorithmic Strategies

Göran Falkman
School of Informatics
University of Skövde

goran.falkman@his.se

Algorithmic Strategies

UNIVERSITY OF SKÖVDE

- Algorithms: Recipes for problem solving
- Brute force
- Greedy algorithms
- Recursion
- Divide-and-conquer
- Dynamic programming
- State space search
- Backtracking

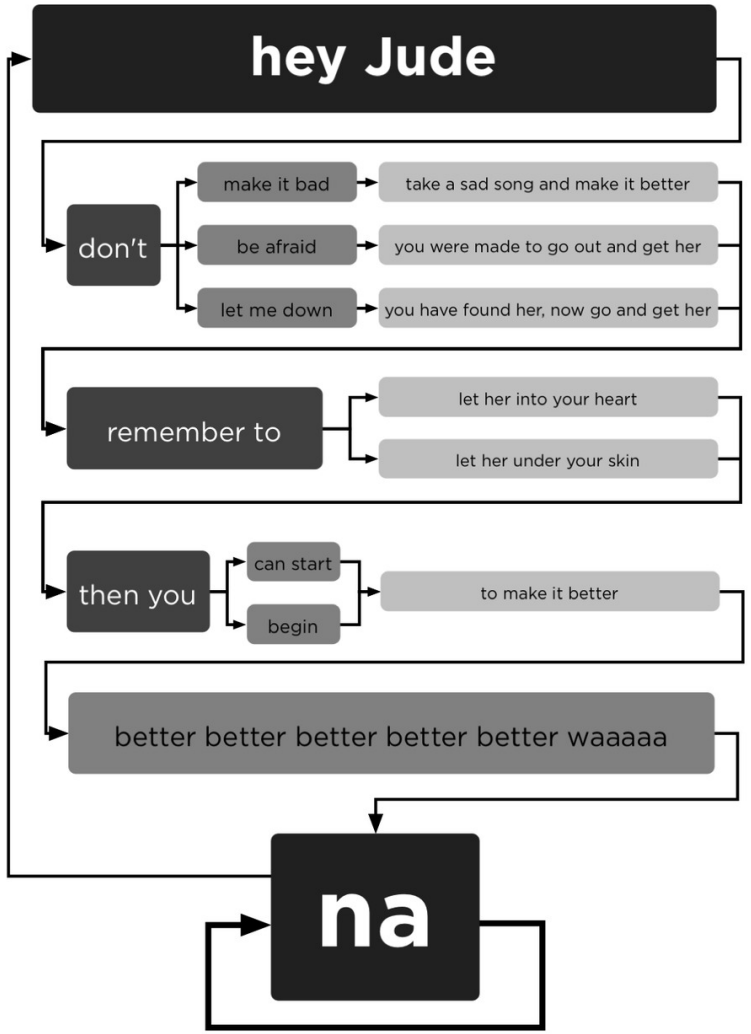
What is an Algorithm?

UNIVERSITY OF SKÖVDE

- Abu Jafar Mohammed ibn Musa al Khowarizimi (circa 825):

  "All complex problems of science must be and can be solved by means of five simple steps."

- Intuitively: A recipe or a step-by-step instruction:
  - A **method for solving a problem** or a class of problems
- Formal definition:
  - A **finite sequence** of instructions, each of which has a **clear meaning** and can be performed with a **finite amount of effort** in a **finite length of time**



Algorithm Design Paradigms

UNIVERSITY OF SKÖVDE

- **General approaches** to the construction of **correct** and **efficient** solutions to problems
- The study of such approaches is of interest because:
  - They provide **templates** (or **patterns**) suited to solving a broad range of diverse problems
  - They can be translated into the control and data structures provided by **any high-level programming language**
  - The temporal and spatial requirements of the algorithms can be **precisely and formally analyzed**
- Although more than one technique may be applicable to a specific problem, it is often the case that an algorithm constructed by one approach is clearly superior to equivalent solutions built using alternative techniques

Brute Force

UNIVERSITY OF SKÖVDE

- Brute force is a straightforward – "the first that comes to mind" – approach to solve a problem based on the problem formulation and concepts involved
- Perhaps **the easiest approach to apply** and is useful for solving small-size instances of a problem
- May result in **naïve solutions with poor performance**
- Some examples of brute force algorithms are:
  - Computing a^n ($a > 0$, n a non-negative integer) by multiplying $a \times a \times \cdots \times a$
  - Computing $n!$
  - Selection sort, Bubble sort
  - Sequential search

Greedy Algorithms

UNIVERSITY OF SKÖVDE

- "Take what you can get now" strategy
- The solution is constructed through a sequence of steps, each expanding a partially constructed solution obtained so far. At each step the choice must be **locally optimal** – this is the central point of this technique
- Examples of problems that can be solved using a greedy algorithm:
  - Finding the minimal spanning tree of a graph
  - Finding the shortest distance in a graph
  - The Knapsack problem
  - The coin exchange problem
  - Using Huffman trees for optimal encoding of information

Divide-and-Conquer

UNIVERSITY OF SKÖVDE

- Divide-and conquer (D&Q) is a method of designing algorithms that proceed as follows:
  - Given an instance of the problem, **split this into several smaller sub-instances**, independently **solve each of the sub-instances** and then **combine the sub-instance solutions** so as to yield a solution for the original instance
- With the D&Q method, **the size of the problem instance is reduced** by a factor (e.g. half the input size)
- Examples of D&Q algorithms:
  - Computing a^n ($a > 0$, n a nonnegative integer) by recursion
  - Mergesort algorithm
  - Quicksort algorithm

Dynamic programming

UNIVERSITY OF SKÖVDE

- One disadvantage of using D&Q is that the process of recursively solving separate sub-instances can result in **the same computations being performed repeatedly**
- The idea behind dynamic programming is to avoid calculating the same quantity twice, usually by **maintaining a table of sub-instance results**
- Dynamic programming is a bottom-up technique in which the smallest **sub-instances are explicitly solved first** and the results of these used to **construct solutions to progressively larger problem instances**
- Examples of dynamic programming algorithms:
  - Fibonacci numbers computed by iteration
  - Warshall's algorithm for finding the shortest path in a graph implemented by iteration

Backtracking

UNIVERSITY OF SKÖVDE

- Backtracking is a **generate-and-test** method used for **state space search problems**
- Based on the construction of a **state space tree**, whose nodes represent states, the root represents the initial state, one or more leaves are goal states and each edge represents the application of an operator:
  - The solution is found by expanding the tree until a goal state is found
- Examples of problems that can be solved using backtracking:
  - **Puzzles** (e.g., eight queens puzzle, crosswords, Sudoku)
  - **Combinatorial optimization problems** (e.g., parsing and layout problems)
  - **Logic programming languages** such as Icon, Planner and Prolog, which use backtracking internally to generate answers

Grenander's Problem

UNIVERSITY OF SKÖVDE

- Given a sequence, $I_1, I_2, ..., I_n$, of integers find the sub-sequence with the maximum sum:
  - If all numbers are negative the result is 0
- Examples:

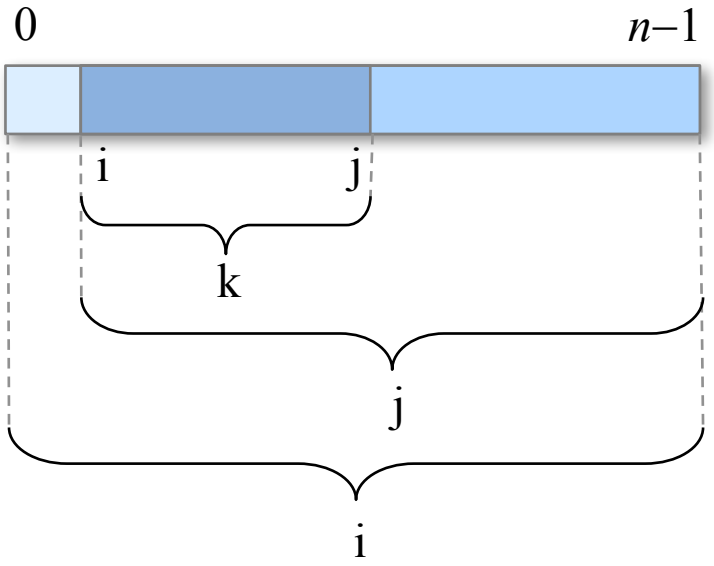
  –2, 11, –4, 13, –4, 2 gives the solution 20

  1, –3, 4, –2, –1, 6 gives the solution 7

Brute Force Solution to Grenander's

UNIVERSITY OF SKÖVDE

```
int grenanderBF(int a[], int n) {
    int maxSum = 0;
    for (int i = 0; i < n; i++) {
        for (int j = i; j < n; j++) {
            int thisSum = 0;
            for (int k = i; k <= j; k++) {
                thisSum += a[ k ];
            }
            if (thisSum > maxSum) {
                maxSum = thisSum;
            }
        }
    }
    return maxSum;
}
```



- Results in **a cubic algorithm**, $O(n^3)$

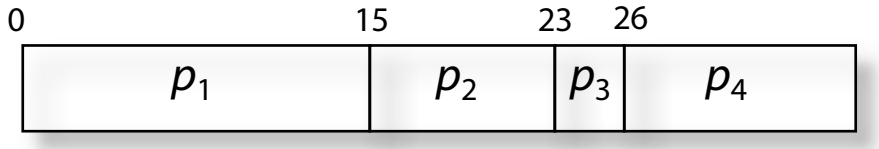
Greedy Algorithms

UNIVERSITY OF SKÖVDE

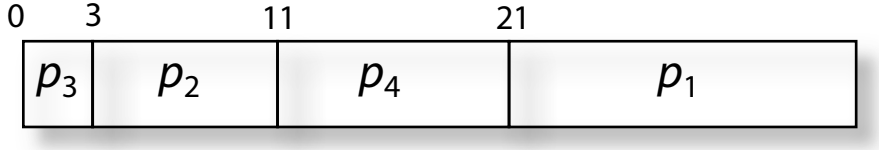
- Try to find solutions to problems **step-by-step:**
  - A **partial solution** is incrementally expanded towards a complete solution
- In each step, there are several ways to expand the partial solution:
  - The **best alternative for the moment** is chosen, the others are discarded

- **Example:** Four processes p_1, p_2, p_3, p_4 take 15, 8, 3 and 10 time units, respectively, to run.

  In which order should the processes be allocated to the CPU in order to minimize the average waiting time?

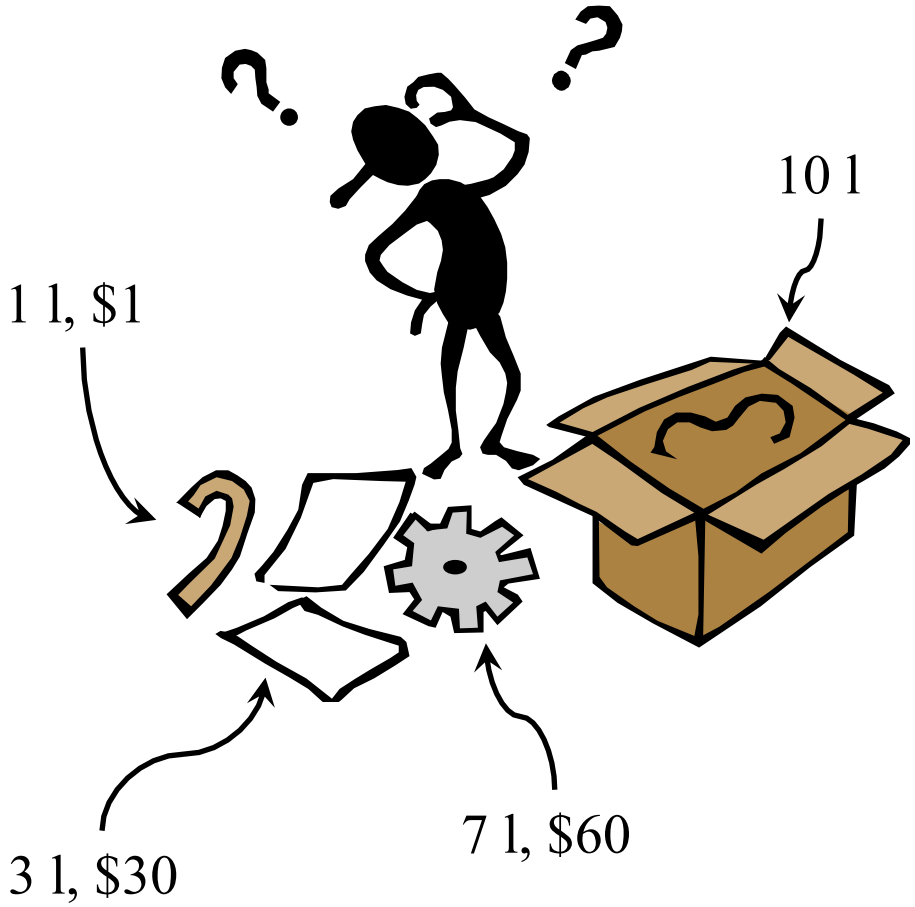


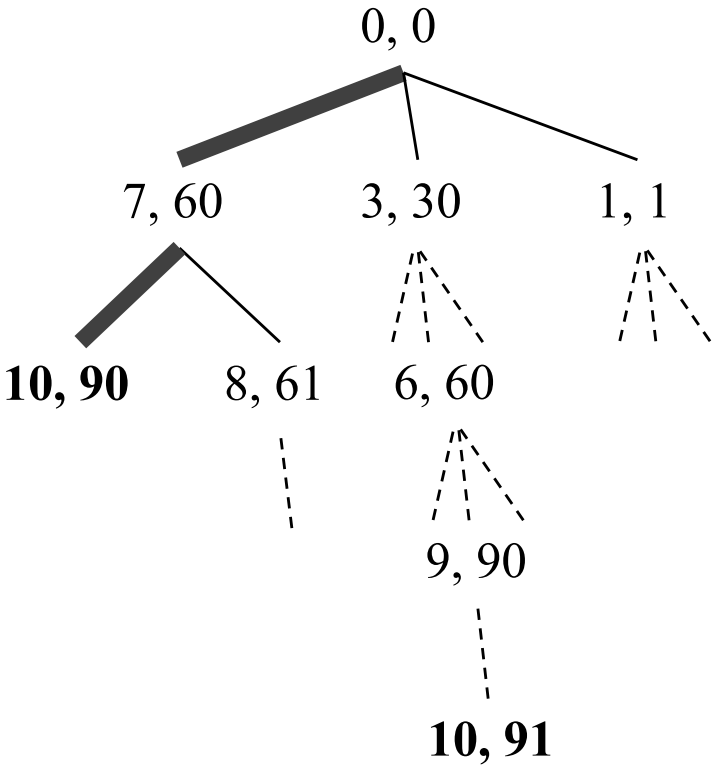
$(0+15+23+26)/4 = 16$



$(0+3+11+21)/4 = 8.75$

To Pack a Knapsack



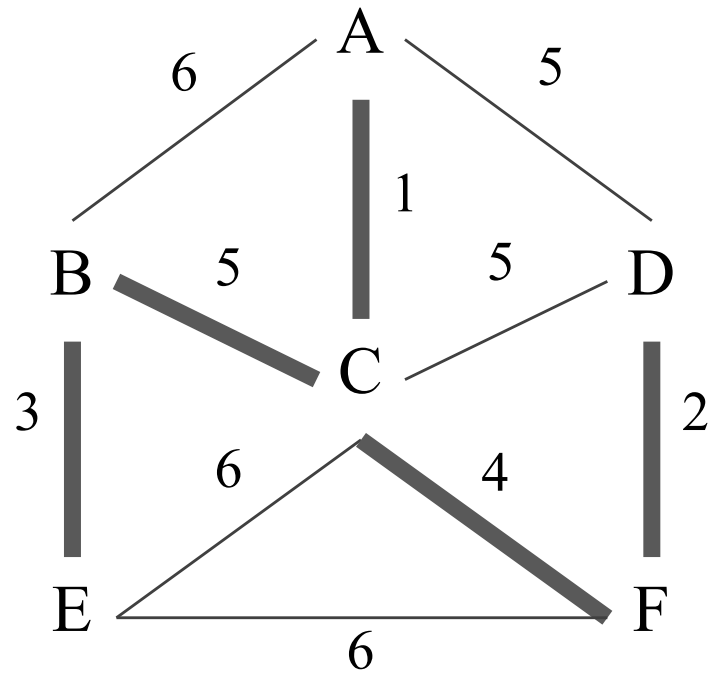


Minimum Spanning Trees

UNIVERSITY OF SKÖVDE

- **Example:** Computers located at different physical locations are to be connected through a network.

  How to build the network using as little cable as possible?



Two Greedy Solutions

UNIVERSITY OF SKÖVDE

1. Let $U = \{\text{some node}\}$ and $T = \emptyset$
2. Find smallest $<u, v> \in E$ such that $u \in U$ and $v \in V-U$
3. Let $T = T \cup \{<u, v>\}$
4. Let $U = U \cup \{v\}$
5. Repeat from 2 until $U = V$

1. Let $S = \{\{v_1\}, ..., \{v_n\}\}$ and $T = \emptyset$
2. Select smallest $<u, v> \in E$
3. If $u \in S_u$ and $v \in S_v$, where $S_u \neq S_v$:
   Let $T = T \cup \{<u, v>\}$
   Let $E = E-\{<u, v>\}$
   Replace S_u and S_v with $S_u \cup S_v$ in S
4. Repeat from 2 until $E = \emptyset$

General Greedy Algorithm

UNIVERSITY OF SKÖVDE

```
Set greedy(Set cands) {
    Set partial = ∅;
    while (!solution(partial) && cands != ∅) {
        Candidate c = bestCandidate(cands);
        cands.remove(c);
        partial.add(c);
        if (!feasibleSolution(partial))
            partial.remove(c);
    }
    if (solution(partial)) return partial;
    else return ∅;      // No solution
}
```

Solving Problems Recursively

UNIVERSITY OF SKÖVDE

- How to define a problem in terms of smaller instances of the same problem?
- How to reduce the size of a problem?
- For which instances of the problem are solutions known (which instances are trivial)?
- How to make sure that we reach trivial instances?

The Fibonacci sequence:

0, 1, 1, 2, 3, 5, 8, 13, 21, ...

$Fib_0 = 0$

$Fib_1 = 1$

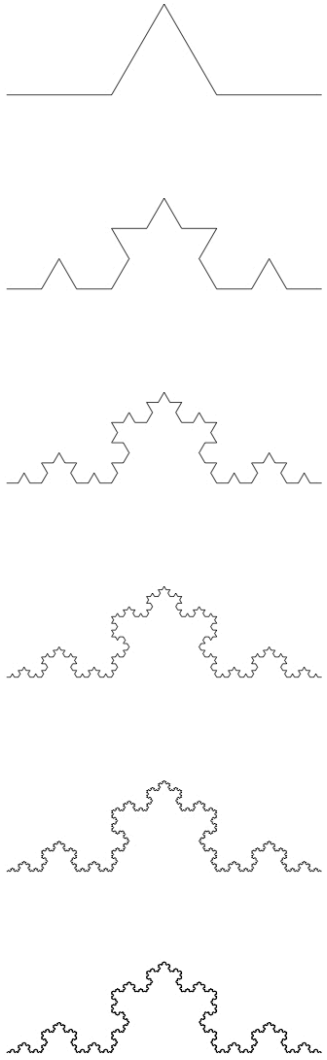
$Fib_i = Fib_{i-1} + Fib_{i-2}$

```
long fib(int n) {
    if (n <= 1)
        return n;
    else
        return fib(n−1) + fib(n−2);
}
```

Why Use Recursion?

UNIVERSITY OF SKÖVDE

- Complex problems can often be expressed **simply and elegantly** using recursion
- Results in **compact and clear** algorithms
- Is closely **coupled to important data structures** (e.g., trees)
- Easier to **prove the correctness** of algorithms



```
/kochR {
   2 copy ge {dup 0 rlineto} {
      3 div
      2 copy kochR
      60 rotate
      2 copy kochR
      -120 rotate
      2 copy kochR
      60 rotate
      2 copy kochR
   } ifelse
   pop pop
} def
```

There is No Magic

UNIVERSITY OF SKÖVDE

even(n) =

true if odd(n–1) is true

false otherwise

odd(n) =

true if even(n–1) is true

false otherwise

Not **workable definitions** of *even* and *odd*

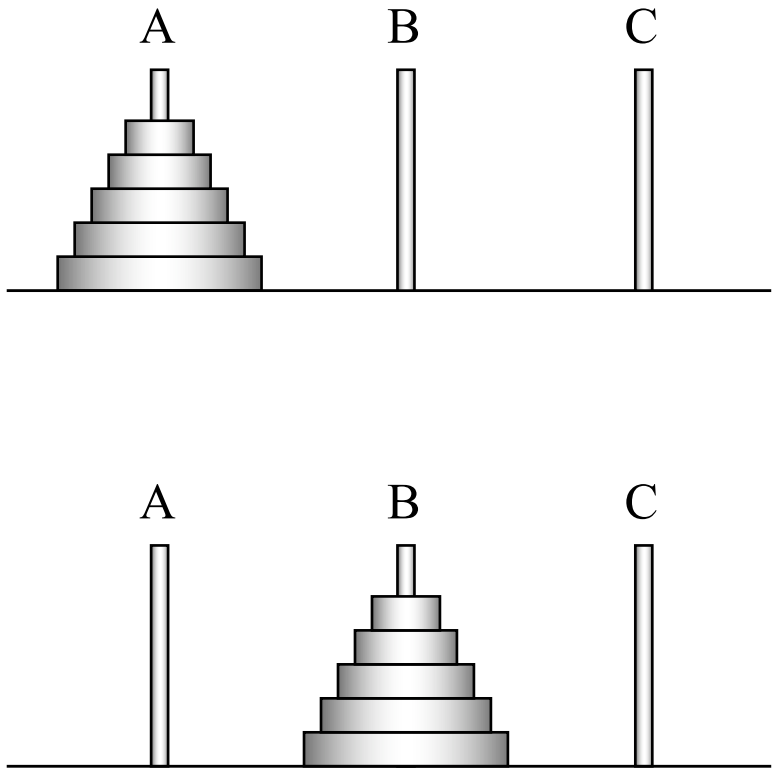
```
int puzzle(int n) {
    if (n == 1)
        return 1;
    else if ((n % 2) == 0)
        return puzzle(n/2);
    else
        return puzzle(3*n + 1);
}
```

The **termination** of *puzzle* for any n has not been proved

The Towers of Hanoi

UNIVERSITY OF SKÖVDE

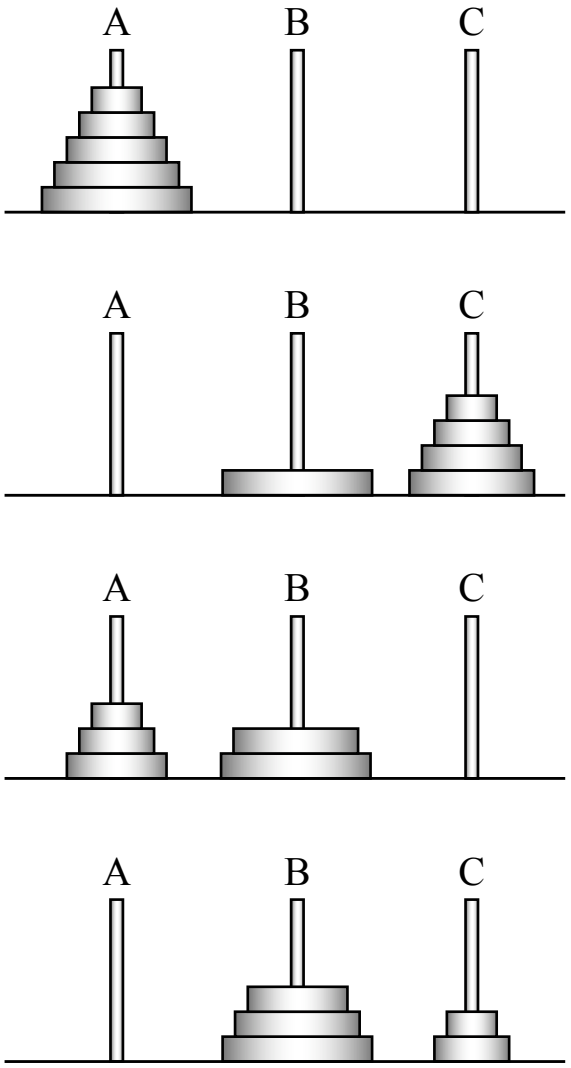
- Three rods A, B and C
- *n* disks are sorted onto A with the largest disk at the bottom
- All disks are to be moved from A to B with C as an auxiliary:
  - Can only move one disk at a time
  - Cannot place a larger disk upon a smaller one



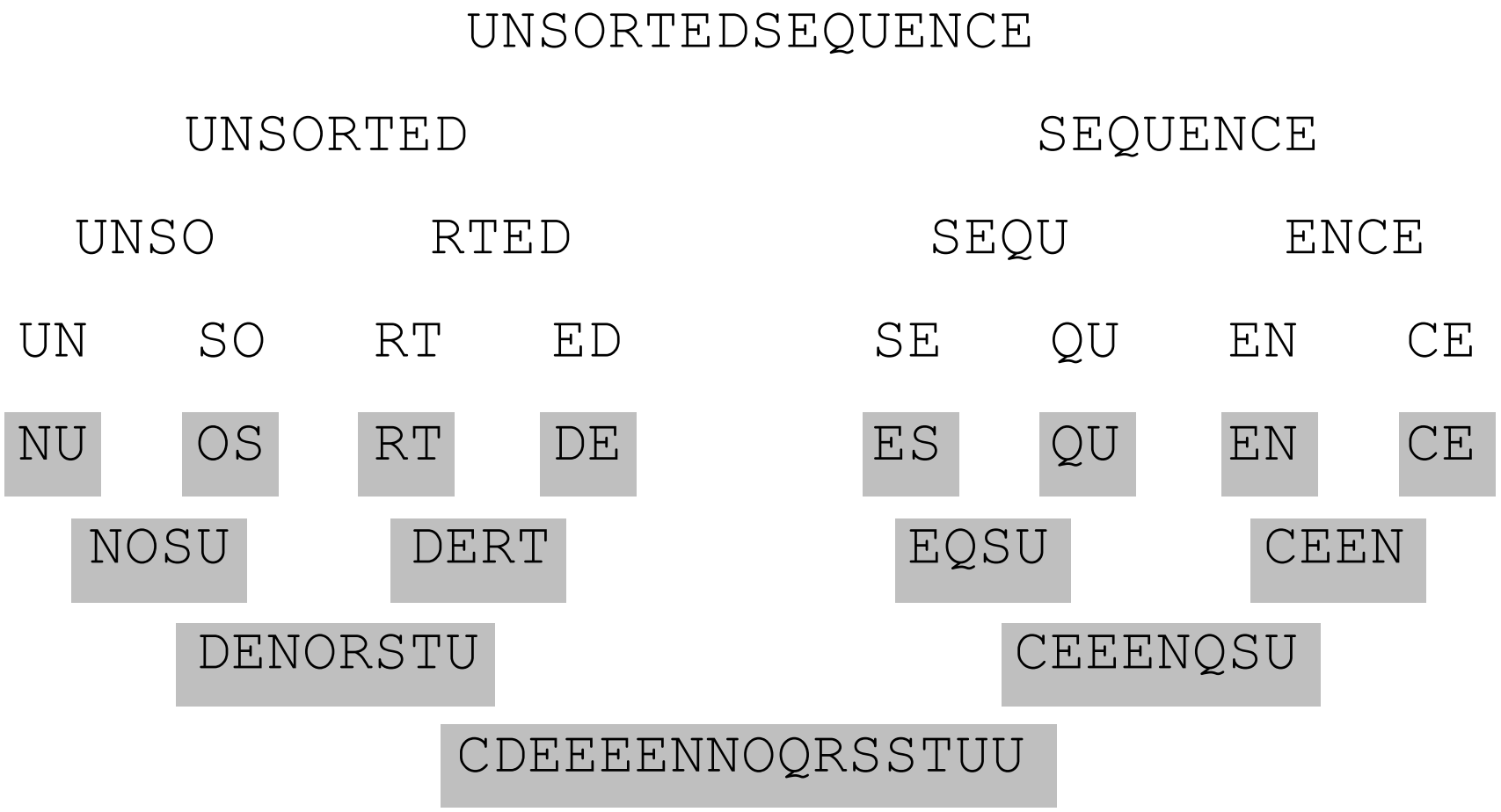
Towers of Hanoi using Divide and Conquer

UNIVERSITY OF SKÖVDE

- Can be very hard to find a direct – brute force – solution to the problem of size *n*
- However, there is a very **simple and elegant recursive solution**:
  - Assume that we can solve the problem of size *n*−1, i.e., we can move *n*−1 disks from one rod to another using a third rod as auxiliary
  - To move *n* disks from A to B:
    - » Move the top *n*−1 disks from A to C using B (we know how to do this)
    - » Move the remaining disk on A to rod B
    - » Move the *n*−1 disks from C to B using A (we know how to do this)



Typical D&Q: Merge Sort



Merge Sort

UNIVERSITY OF SKÖVDE

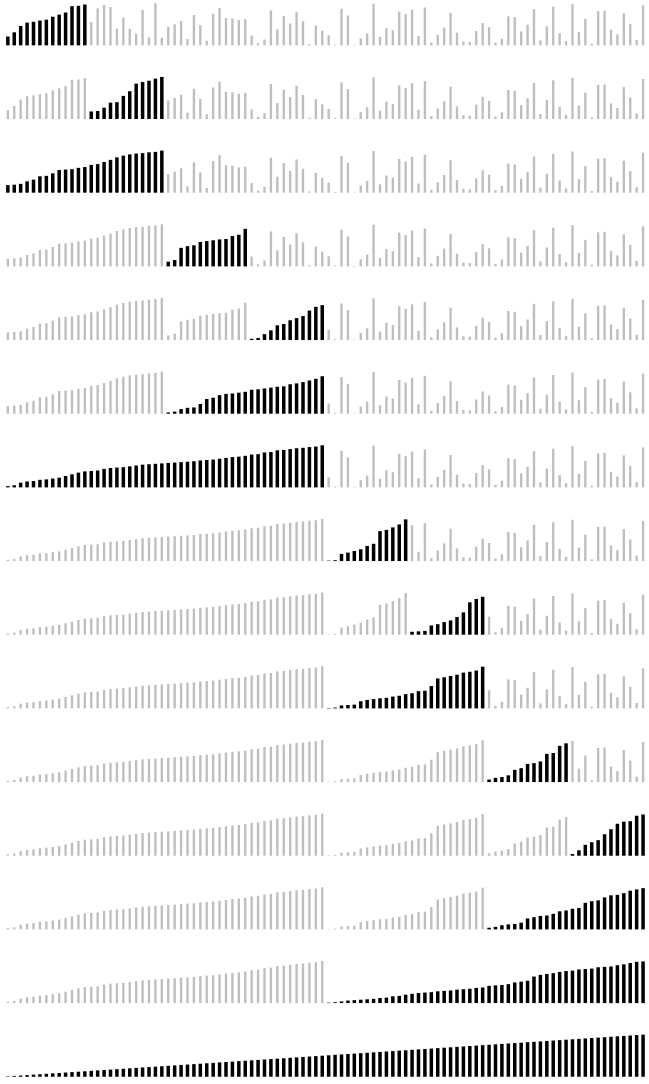
```
void mergesort(Item a[], int l, int r) {
    if (r–l <= 1) {
        return;
    } else {
        int m = (r + l) / 2;
        mergesort(a, l, m);
        mergesort(a, m+1, r);
        merge(a, l, m, r);
    }
}

void mergesort(Item a[], int size) {
    mergesort(a, 0, size–1);
}
```

Already sorted?

Divide the list into two equal parts

Sort the two halves recursively

Merge the sorted halves into a sorted whole



General D&Q Algorithm

UNIVERSITY OF SKÖVDE

```
divideAndConquer(Problem p) {
    if (p is simple or small enough) {
        return simpleAlgorithm(p);
    } else {
        divide p in smaller instances p1, p2, ..., pn
        Solution solutions[n];
        for (int i = 0; i < n; i++) {
            solutions[i] = divideAndConquer(pi);
        }
        return combine(solutions);
    }
}
```

Grenander's Problem Using D&Q

UNIVERSITY OF SKÖVDE

```
int grenanderDQ(int a[], int l, int h) {
    if (l > h) return 0;
    if (l = h) return max(0, a[l]);
    int m = (l + h) / 2;
    int sum = 0;
    int maxLeft = 0;
    for (int i = m; i >= l; i--) {
        sum += a[i];
        maxLeft = max(maxLeft, sum);
    }
    sum = 0;
    int maxRight = 0;
    for (int i = m + 1; i <= h; i++) {
    sum += a[i];
        maxRight = max(maxRight, sum);
    }
    int maxL = grenanderDQ(a, l, m);
    int maxR = grenanderDQ(a, m+1, h);
    int maxC = maxLeft + maxRight;
    return max(maxC, max(maxL, maxR));
}
```

Divide the problem

Solve the sub-problem

Solve the sub-problem

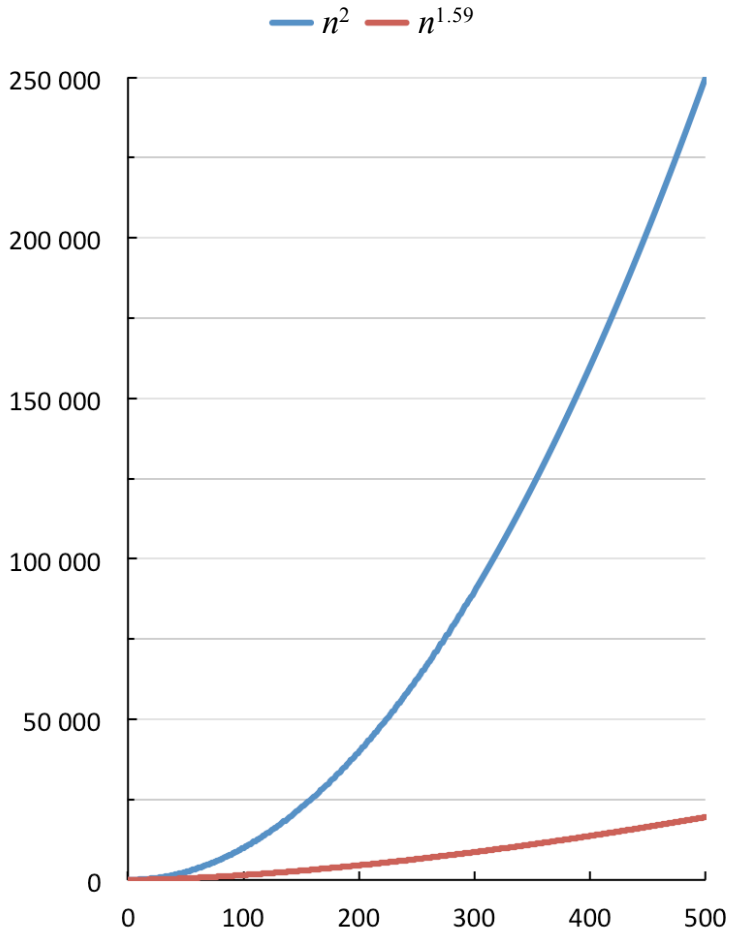
Solve the sub-problems

Combine the solutions

Multiplying Large Numbers Using D&Q

UNIVERSITY OF SKÖVDE

- Brute force algorithm for X·Y:
  - Multiply each of the n bits of X with each of the n bits of Y, summing the parts, $\Theta(n^2)$
- Faster algorithm using D&Q:
  - Let $X = A \cdot 2^{n/2} + B$ and $Y = C \cdot 2^{n/2} + D$
  - Then $X \cdot Y = AC \cdot 2^n + (AD+BC) \cdot 2^{n/2} + BD$, which is $O(n^2)$
  - Rewrite: $X \cdot Y = AC \cdot 2^n + ((A-B)(D-C) + AC+BD) \cdot 2^{n/2} + BD$, which is $O(n^{\log_2 3}) \approx O(n^{1.59})$



Dynamic Programming and the Principle of Optimality

UNIVERSITY OF SKÖVDE

- In an optimal sequence of choices, actions or decisions each sub-sequence must also be optimal:
  - Examples include finding the shortest path between two cities
  - **An optimal solution to a problem is a combination of optimal solutions to some of its sub-problems**
  - Not all optimization problems adhere to this principle
- Dynamic programming is similar to D&Q:
  - Divides the original problem into smaller sub-problems
  - Many times it is hard to know beforehand which sub-problems that are needed to be solved in order to solve the original problem
  - Dynamic programming **solves a large number of sub-problems** and uses some of the sub-solutions to form a solution to the original problem
  - Is **a bottom-up technique**

Dynamic Programming

UNIVERSITY OF SKÖVDE

- The same sub-problems may reappear
- To avoid solving the same sub-problem more than once, **sub-results are saved** in a data structure that is updated dynamically
- Sometimes the result structure (or parts of it) may be **computed beforehand**

```
long fib(int n) {
    if (n < 2)
        return 1;
    else
        return fib(n–1) + fib(n–2);
}
```

fib(4) ⇒ fib(3) + fib(2)
⇒ fib(2) + fib(1) + fib(2)
⇒ fib(1) + fib(0) + fib(1) + fib(2)
⇒ fib(1) + fib(0) + fib(1) + fib(1) + fib(0)

Examples of Dynamic Programming

UNIVERSITY OF SKÖVDE

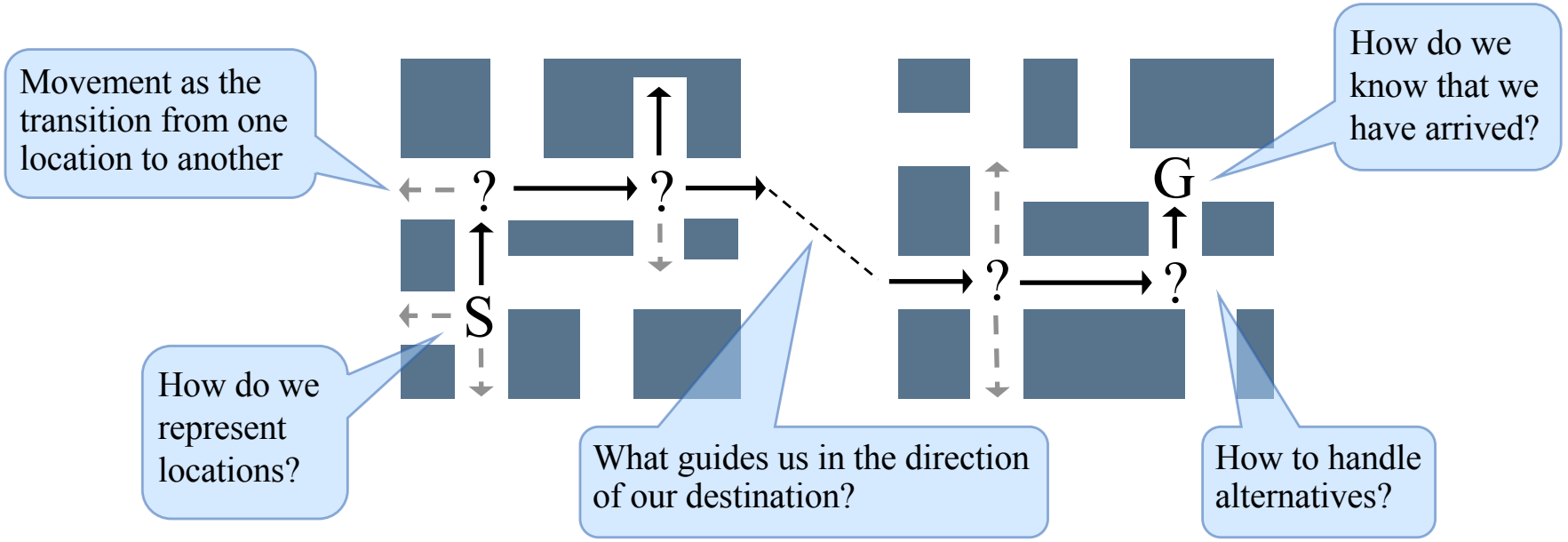
```
Table t = new Table();

int fib(int n) {
    if (n < 2) return 1;
    else {
        bool ok;
        int answer = t.retrieve(n, ok)
        if (!ok) {
            answer = fib(n−1) + fib(n−2);
            t.insert(n, answer);
        }
        return answer;
    }
}
```

```
int grenanderDP(int a[], int n) {
    int table[n+1];
    table[0] = 0;
    for (int k = 1; k <= n; k++)
        table[k] = table[k−1] + a[k−1];
    int maxSoFar = 0;
    for (int i = 1; i <= n; i++)
        for (int j = i; j <= n; j++) {
            thisSum = table[ j ] − table[i−1];
            if (thisSum > maxSoFar)
                maxSoFar = thisSum;
        }
    return maxSum;
}
```

State Space Search

UNIVERSITY OF SKÖVDE

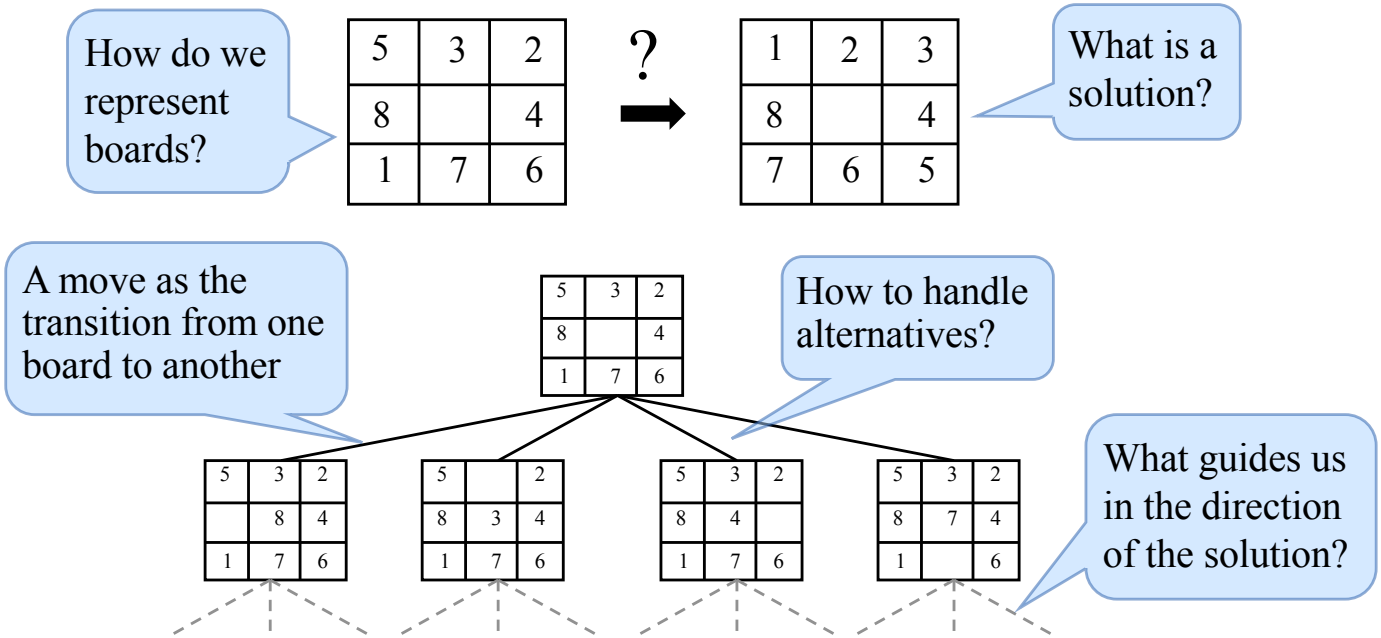
- Solving problems through the **systematic search** for solutions in a (large) **state space**
- The general idea is to incrementally **extend partial solutions until a complete solution is obtained**
- Example: Navigating a city



State Space Search (cont.)

UNIVERSITY OF SKÖVDE

- Search is the systematic process of **choosing one of many possible alternatives**, saving the rest in case the alternative selected first would not lead to the goal
- Search as the construction and traversal of **search trees**
- Example:
  The 8-puzzle



State Space Search (cont.)

UNIVERSITY OF SKÖVDE

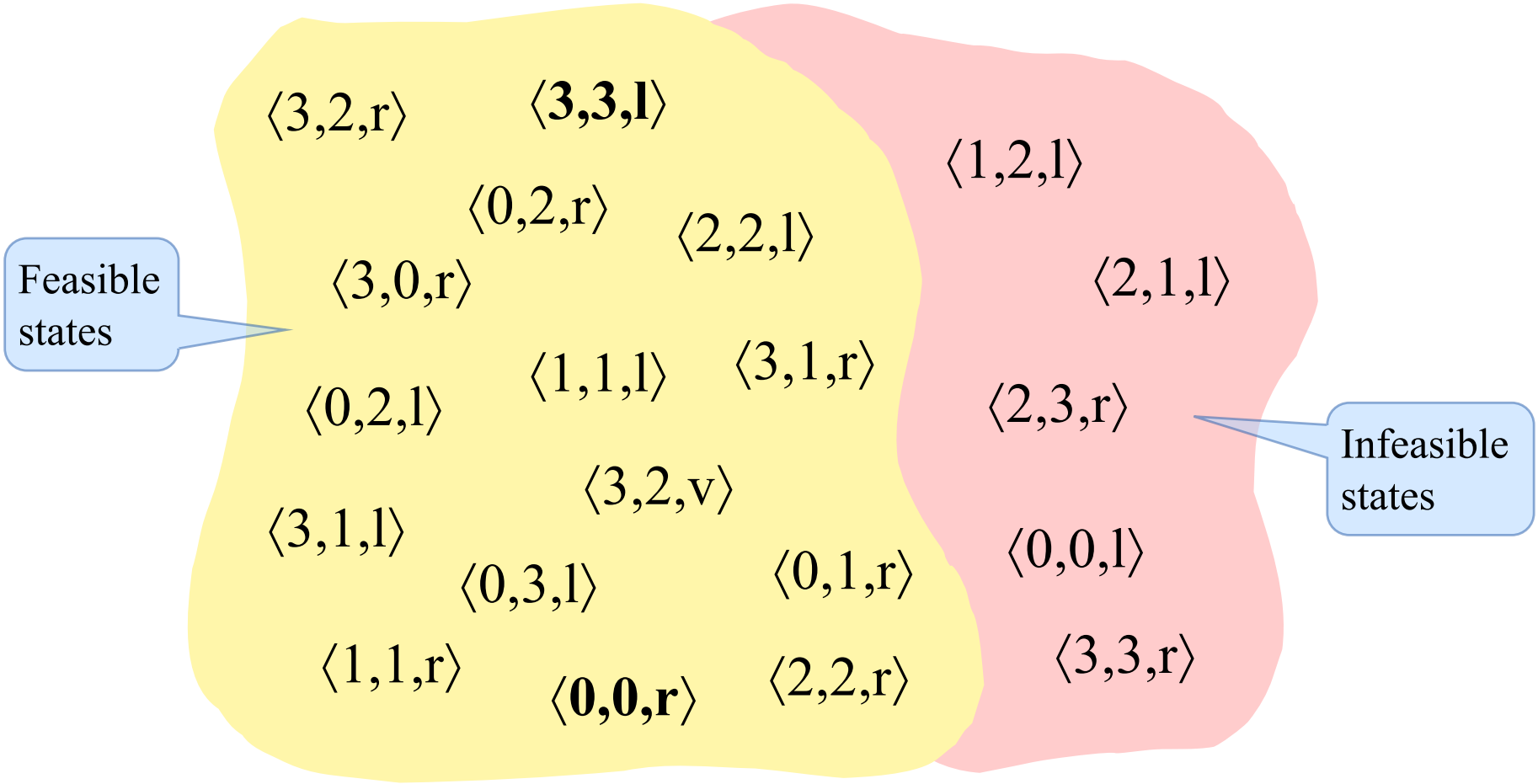
- The **initial state** (e.g., an address or a board)
- A **set of operators** which take us from one state to another state (e.g., walk or slide a tile)
- A **goal-test** which decides when the goal is reached (e.g., comparing addresses or boards):
  - Explicit states (e.g., a specific address or board)
  - Abstractly described states (e.g., any post office)
- A **description of a solution** (e.g., the address, the path between locations or the moves used):
  - The search path (e.g., the shortest path between your home and your office or the series of moves)
  - Just the final state (e.g., the post office)
- A **cost function** (e.g., time, money, distance or number of moves) = *true cost* for going from start to where we are now + *estimated cost* for going from we are now to the nearest goal:
  - Search cost, the cost for concluding that a certain operator should be used (e.g., the time it takes to ask someone for directions or thinking about a move) +
  - Path cost, the cost for using a operator (e.g., the energy it takes to walk or time)

Example: Missionaries & Cannibals

UNIVERSITY OF SKÖVDE

- Transport three missionaries and three cannibals from the left bank of a river to the right bank, using a canoe with room for two. The number of missionaries may never be less than the number of cannibals:
  - **State**: ⟨# M to the left, # C to the left, location of the canoe⟩
  - **Start state**: ⟨3, 3, l⟩
  - **Goal state**: ⟨0, 0, r⟩
  - **Operators**: move one of each, move one M, move one C, move two M or move two C
  - **Cost**: number of river crossings

State Space of M & C



A Note on State Spaces

UNIVERSITY OF SKÖVDE

- Propositional satisfiability problem (SAT):
  - Decide if there is an assignment to the variables of a propositional formula that satisfies it:

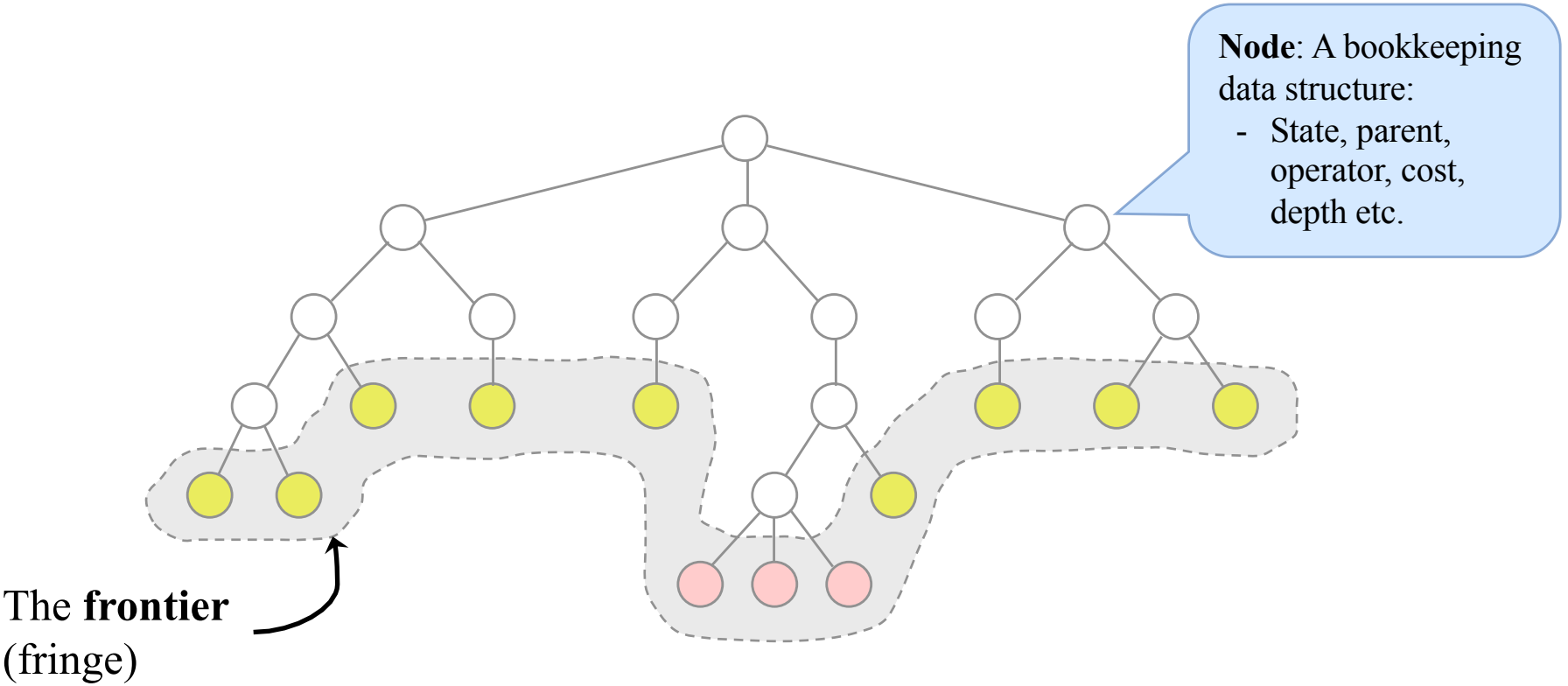
$$f = (\bar{x}_2 + \bar{x}_4)(x_3 + x_4)(\bar{x}_3 + x_4)(x_1 + x_2)(\bar{x}_1 + \bar{x}_3)$$

  - 100 variables $\Rightarrow 2^{100} \approx 10^{30}$ combinations 1000 evaluations/second $\Rightarrow$ **31,709,791,983,764,586,504 years** required to evaluate all combinations!

- Traveling salesman problem (TSP):
  - Given a number of cities along with the cost of travel between each pair of them, find the cheapest way of visiting all the cities exactly once and returning to the starting point
  - There are 2^n identical tours for each permutation of n cities ☞ the number of tours are $n!/(2n) = (n-1)!/2$
  - A 50-city TSP therefore has about **$3 \cdot 10^{62}$ potential solutions**

Search Trees

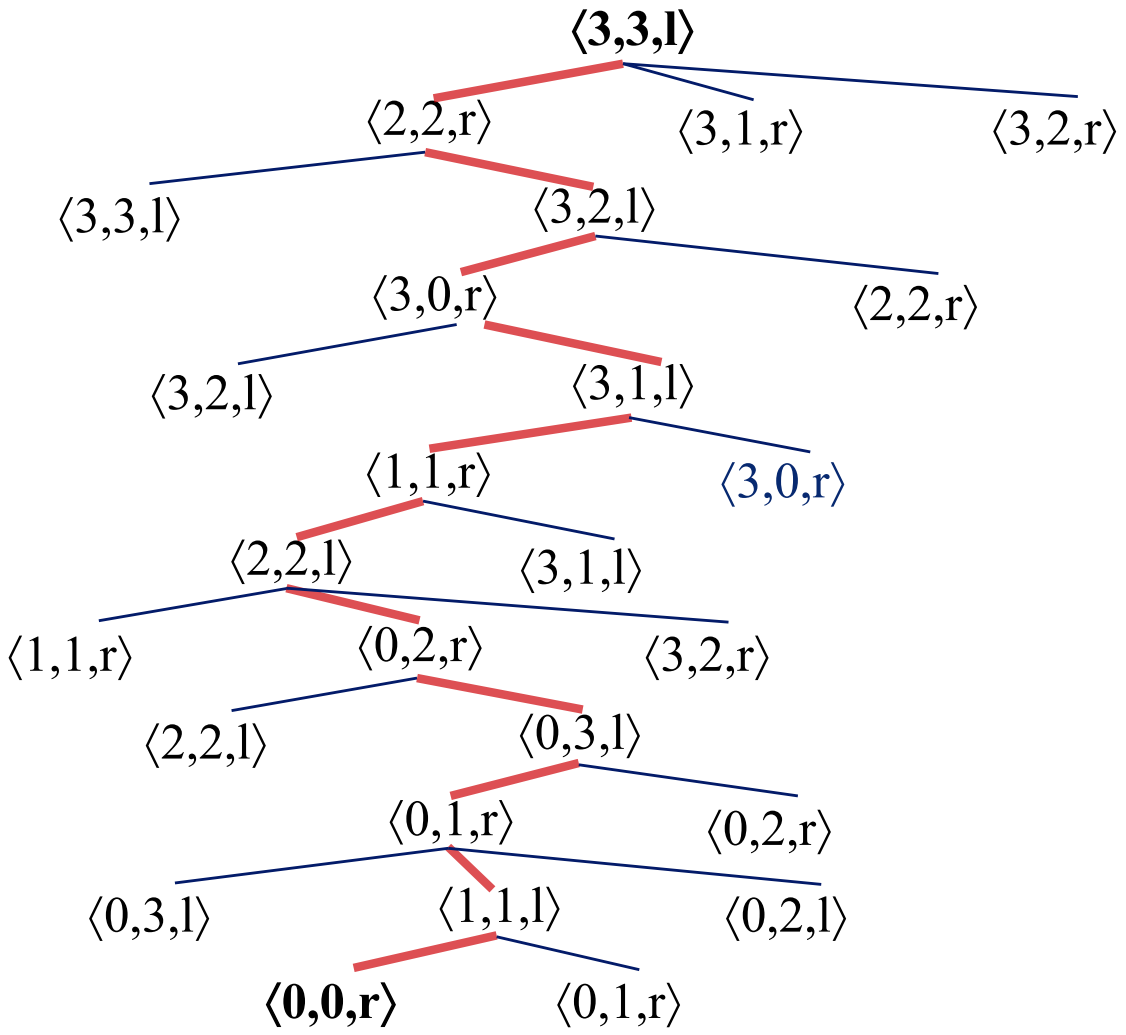
UNIVERSITY OF SKÖVDE



- Already expanded nodes (CLOSED)
- Nodes waiting to be expanded (OPEN)
- Recently generated nodes waiting to be expanded (NODES)

Search Tree of M & C

UNIVERSITY OF SKÖVDE



General State Space Search Algorithm

UNIVERSITY OF SKÖVDE

Create a node *S* from the start state
Create a list, *OPEN* := [*S*], containing all nodes that have not yet been expanded
Create a set *CLOSED* := [], containing all nodes that have already been expanded
loop
    if *OPEN* = [] **then return** failure
    else
        N := first element in *OPEN*
        Remove *N* from *OPEN*
        Add *N* to *CLOSED*
        if *N* represents a goal state **then return** SOLUTION
        else
            NODES := the nodes we get by applying all applicable operators to *N*
            Remove all nodes in *NODES* that are members of *OPEN* or *CLOSED*
            OPEN := expand(*OPEN*, *NODES*)

Expand evaluates all open nodes according to how **promising** they are and may **sort and truncate** the resulting list of nodes

Backtracking

UNIVERSITY OF SKÖVDE

- In each step, **a promising alternative is chosen**
- If chosen alternative does not lead to a solution we **backtrack to the most recent step with untried promising alternatives**
- If all alternatives of the first step have been tried without reaching a complete solution we have failed to find a solution
- Backtracking is an instance of general state space search (or a realization of state space search):
  - Uses **depth-first** search, where *NODES* are just put in front of *OPEN*, i.e., the fringe is used as a **stack**, usually without a cost function

General Backtracking Algorithm

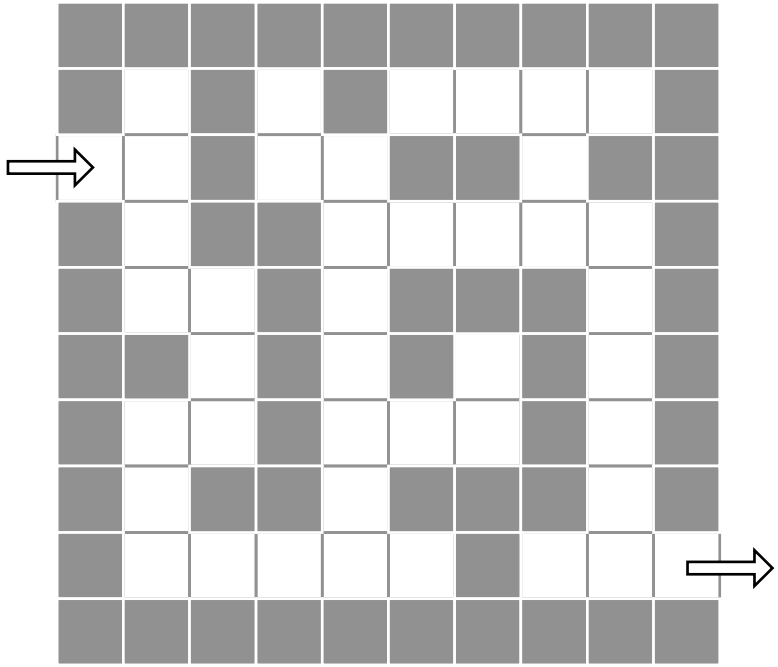
UNIVERSITY OF SKÖVDE

```
Solution backtrack(Solution partial) {
    if (solution(partial)) return partial;          // Solution found!
    else {
        Solution next;
        while (extend(partial, next))               // Choose an alternative
            if (promising(next)) {                  // If promising:
                Solution s = backtrack(next);       // expand recursively
                if (solution(s))
                    return s;                       // Solution found!
            }                                       // Else, try next alternative
        }
        return partial;     // Have tried all alternatives – backtrack!
    }
}
```

Navigating a Maze

UNIVERSITY OF SKÖVDE

- Finding a way through a maze is a problem for which backtracking is suitable
- If we reach a dead end, we backtrack to the most recent intersection and choose another path



Problem Representation – States

UNIVERSITY OF SKÖVDE

- The maze is represented by a matrix, where 1 means an opening and 0 means a wall
- A **state** – description of a (partial) solution – consists of a maze and the current position

```
typedef struct s_struct {
      int maze[10][11];
      int x, y;
} state;

void main() {
      state s = {
            { {0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0},
            {0,  0,  1,  0,  1,  0,  1,  1,  1,  1,  0},
            {0,  1,  1,  0,  1,  1,  0,  0,  1,  0,  0},
            ...,
            {0,  0,  1,  1,  1,  1,  1,  0,  1,  1,  1},
            {0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0} },
            1, 2  }; // Starting position

      srand(time(0));
      state g = backtrack(s);
      cout << "Final position: " << g.x << ", "
            << g.y << endl;
      exit(0);
}
```

Backtracking Function

UNIVERSITY OF SKÖVDE

```
state backtrack(state partial) {
    if (solution(partial)) return partial;
    else {
        state next;
        while (extend(partial, next))
            if (promising(next)) {
                state s = backtrack(next);
                if (solution(s)) return s;
            }
        return partial;
    }
}
```

Next Available Position – Operators

UNIVERSITY OF SKÖVDE

```
// If not all four neighbours of s are
// occupied, we choose one of them
bool nextPos(state s, int &x, int &y) {
    int dirs[] = { 0, 1, 2, 3 };
    shuffle(dirs, 4);
    for (int i = 0; i < 4; i++) {
        x = s.x;
        y = s.y;
        if (dirs[i] == 0) x -= 1;
        else if (dirs[i] == 1)  y -= 1;
        else if (dirs[i] == 2)  x += 1;
        else if (dirs[i] == 3)  y += 1;
        if (s.maze[y][x] == 1)
            return true;
    }   // for
    return false;
}   // nextPos

void shuffle(int a[], int n) {
    int i1, i2, t;
    for (int i = 0; i < 17; i++) {
        i1 = rand() % n;
        i2 = rand() % n;
        t = a[i1];
        a[i1] = a[i2];
        a[i2] = t;
    }
}   // shuffle
```

Solution, Promising and Extend

UNIVERSITY OF SKÖVDE

```
// A partial solution is a complete
// solution if we've reached the exit
bool solution(state s) {
    return (s.x == 10) && (s.y == 8);
}

// A state is promising if we can
// extend our path or if we have
// a solution
bool promising(state s) {
    int x, y; // Dummies – not used
    return solution(s) || nextPos(s, x, y);
}
```

```
bool extend(state &s, state &next) {
    // Copy s to next
    for (int i = 0; i < 10; i++)
        for (int j = 0; j < 11; j++)
            next.maze[i][j] = s.maze[i][j];
    // Find a untried path from s
    if (nextPos(s, next.x, next.y)) {
        // "Close the door behind us"
        next.maze[s.y][s.x] = 0;
        // Mark path as tried
        s.maze[next.y][next.x] = 0;
        return true;
    }
    return false; // No untried paths left
}
```

Conclusions

UNIVERSITY OF SKÖVDE

- Usually, a given problem can be solved using various approaches, where some approaches result in much **more efficient solutions** and some approaches result in **clearer, more easy to understand, algorithms**
- Greedy algorithms don't regret anything:
  - Suitable when **sub-optimal solutions** are acceptable
  - A locally optimal solution can often be optimized globally afterwards
- D&Q is **suitable for (nearly) dividable problems**, consisting of independent sub-problems:
  - The complexity is typically $O(n \cdot \log n)$, but this is dependent on the branching factor (e.g., 2 in mergesort), the cost of dividing the problem (e.g., O(1) in mergesort) and the cost of combining sub-solutions (e.g., $O(n)$ in mergesort)

Conclusions (cont.)

UNIVERSITY OF SKÖVDE

- Dynamic programming can be regarded as **a technique for improving efficiency**:
  - Similar to D&Q, but without predefined criteria for dividing a problem into subproblems
- In contrast to greedy algorithms, backtracking doesn't discard untried alternatives, but keeps them for the future if the chosen alternative turns out not to lead to a solution:
  - A way of performing a **systematic and exhaustive search** of the state space for a solution