Recursion stacks the value of all local variables and parameters; on return it restores all the local variables and parameters from the top level of the stack.

Problem solving using Backtracking

Algorithm uses trial and error

Brute force
Systematic approach to trying alternative “choices” and eliminating them if they don’t work out
Never try a “choice” more than once
Eventually find a solution if one exists

Recursion is natural because when a choice is no longer viable, we go “erase” the choice, go back to the previous choice and pick a new alternative, if no alternative is available we recurse backwards

Decompose the trial and error into partial tasks which are best done recursively

Stacking mechanism of recursive solutions automatically records everything and makes it available at the appropriate point when needed for retry.

Consider a Maze

Explore paths until reach a dead end, and then retrace the points in reverse order until you reach a point where you can try taking an alternative path

Need to keep track of all paths tried as we go through the maze so we don’t retry them

Well Known Backtracking Problems

Maze, walk thru a graph, Knights tours, 8 queens are some, map coloring problems
General Backtracking Algorithm

```java
boolean try (move)
{
    initialize selection of possibilities
    if current choice for move is not acceptable return false
    record choice for move
    if done with all moves return true
    loop while (moves are available)
    {
        pick next move
        success = try (next move)
        if successful return true
    }
    unrecord move (unsuccessful)
    return false
}
```

The data representation of the problem is not passed as a parameter to the recursive method as it would then stack the data representation adding further to the inefficiency of recursion.

Consider the Maze

Data Representation – char [10][10] maze
   Each element contains P – path, H – hedge, or E – exit

Algorithm
```
boolean findWayOut (int r, int c)
{
    Initialize
    Placement ok (in bounds and not a Hedge)
    No – return false
    Done (at exit) – return true
    Record move (turn it into a H, so we don’t take it again)
    Loop ---
        Determine Next move out of (set r and c for next call)
        Above
        Right
        Below
        Left
        Call with next move
        Success – return true
    End loop when no moves left to take
    Unrecord – turn into P
    Return false
```
Test Cases

- No Exit
- No backtracking needed
- On edge
- Not on edge

Sample Program: Maze.java

Knights tour

Consider an eight by eight matrix representing a chess board. The problem at hand is to place a knight anywhere on the board and have it travel around (tour) the board (using one of the eight possible moves of the knight) without occupying the same square twice.

The eight possible moves of the knight are:

- 2 up, 1 to the left
- 2 up, 1 to the right
- 2 down, 1 to the left
- 2 down, 1 to right
- 2 to the left, 1 down
- 2 to the left, 1 up
- 2 to the right, 1 up
- 2 to the right, 1 down

Data representation

```
int [][] board --- nxn board

   each element holds moveNum visited on or -1 if not yet visited
```
Algorithm

Initialize board to unvisited (-1)

boolean ktour (int r, int c, int moveNum)
  is r,c acceptable move? (not already visited and on board)
    yes -- record move (board [r][c]=moveNum)
  all moves made? (moveNum == boardSize*boardSize)
    yes -- return true
  loop -- pick next move
    switch between 8 possible moves setting nr, nc
      if (ktour (nr, nc,moveNum+1) is successful) return true
  end loop
  unrecord move not-successful (board [r][c]=-1)
  return false

Sample Programs: Knights.java, KnightsTour.java

Heuristic

A way to speed up the solution process, when an exhaustive search is impractical.
Used to increase performance, loose precision.
  Maze – Right Hand rule
  Knights Tour – Warndorff’s Rule