Low-Level Design

Design

- High Level Design
  - public classes (used by clients/users)
  - public methods
  - public attributes
  - exceptions
- Low Level Design
  - high level design info plus
  - private classes, private methods, private attributes
  - data structures
  - algorithms

Low Level Design Document

- provide the interface for all classes
  - public and private methods, including parameters, return values and exceptions thrown
  - types defined
- describe and justify your choice of data structures
  - describe the major alternatives that you considered and why you made the choice that you did
- describe all interesting algorithms
  - describe any alternatives that you considered and justify the choice that you made
- provide the inheritance relationships for your classes

Low Level Design Presentation

- too much detail to present the full design
- what to present?
  - Review HLD
  - select representative aspects of the LLD
    - after seeing these should understand how similar parts of the design are done
  - select the most interesting aspects of the design
    - often the most controversial
    - demonstrates that you have thought about the issues

How to do Low Level Design

- must consider alternatives
- make well-reasoned choices among alternatives
- when choice is not clear, pursue multiple solutions until
  - a choice becomes clear
  - it is too costly to pursue multiple solutions
    - make your best guess
    - plan for change
      - Try to keep the same interface so that the implementation is isolated

Data structure and algorithm design

- still an art
- consider how the data structure will be used
  - types of operations
  - frequency of operations
- select a structure that will be
  - efficient for the proposed uses
  - easy to implement
  - easy to maintain
- usually see the finished product, not the alternatives that were explored and rejected

Goals

- Figure out how to implement API defined during HLD:
  - What classes to create
  - How to relate those classes
  - What methods you will need
- Start to make some design decisions:
  - Data structures
  - Algorithms
  - Policies

Goals (continue)

- Explore using existing software
  - What does it do?
  - How to extend it to do what you need?
  - How to fit it into API?
- Add/Keep Flexibility
  - Isolate low-level decisions in even lower-level structure
  - Try to make it easy to make changes

An Incremental Approach

- Make some decisions
- Assess the consequences
  - Do other parts need to be changed now?
  - Will it be easy enough to implement?
  - Are all requirements met?
- Change these decisions or earlier ones
- Repeat

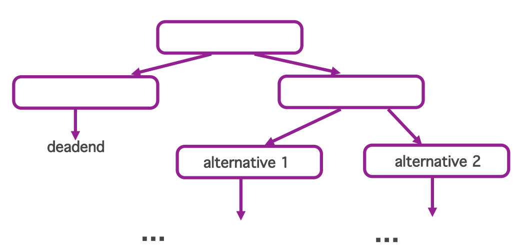
Stepwise Refinement

- a well-known problem solving technique
- divide and conquer approach
- used in many disciplines
  - mathematics--lemmas and corollaries
  - house design--floor plans, wiring, etc.
- also called top down design, incremental development

Guidelines for Stepwise Refinement

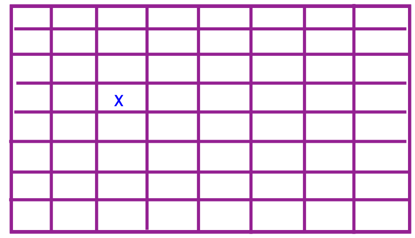
- decompose decisions
- untangle interdependencies
- defer representations and detailed algorithm decisions as long as possible
  - wait until there is information to help make these decision
- be prepared to undo previous steps and restart

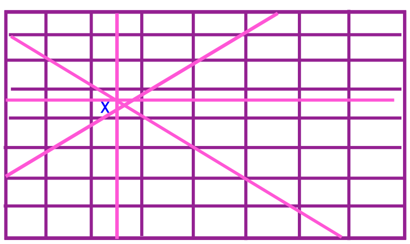
Decision Tree



Example:8 Queens problem

- Given an 8x8 chess board and 8 queens, find a safe position for each queen
  - i.e., every row, column, and diagonal contains at most one queen





First Solution

x an element of the set of configurations
safe(x) is true if x satisfies the problem

```
solution := false
initialize configuration
repeat
      x := next configuration
until safe (x) or no more configurations
if safe(x) then
     solution := true
     print (" solution", x)
else print ( "no solution" )
endif
```

First alternative

consider every possible board configuration

$$\binom{64}{8}$$

$$\frac{64!}{(64-8)!\ 8!}$$

$$= \frac{64 \times 63 \times \ldots \times 57}{8!} \sim 2^{32}$$

$$= 4{,}426{,}165{,}368$$

First alternative

- Could refine this solution but know it is expensive
  - Need to consider alternative solutions
- Considering efficiency at the algorithm level and data structure level

Second alternative

- 1 queen in every column
  - $8^8 = 2^{24} = 16,777,216$
- previous algorithm is still adequate
- set of configurations is now restricted to configurations with 1 queen per column

Second alternative elaborated

x an element of the set of configurations, where a configuration can only have one queen in each column
safe(x) is true if x satisfies the problem

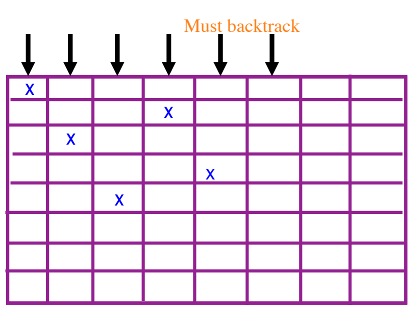
```
solution := false
initialize configuration
repeat
      x := next configuration
until safe (x) or no more configurations
if safe(x) then
      solution := true
      print (" solution", x)
else print ( "no solution" )
endif
```

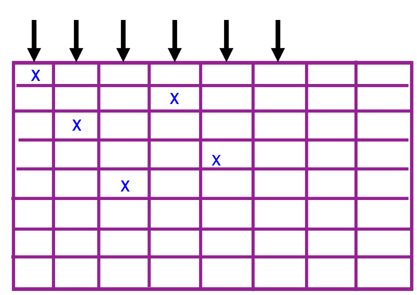
Third alternative

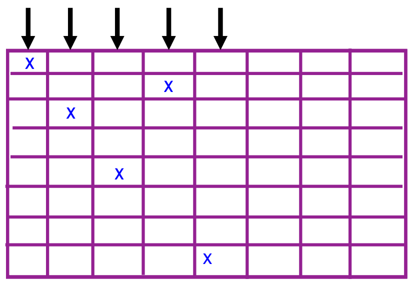
- While forming next configuration, could be checking that it is safe
- safe (for a partial configuration)
  1<= i <= 8
      x1
      x1, x2
      x1, x2, x3
- backtrack one column if there is no safe configuration for the next column

Third Alternative

- similar to a counter
  - each solution can only increment the counter
  - must find a solution for first J columns before going to J+1







Third alternative

```
initialize to first column
repeat
    try column
    if successful then
       set queen
       increment to next column
    else backtrack
    endif
until last column done or backtracking done
```

Third Alternative Elaborated

```
//initialize to first column
     column :=1
     row:= 0
repeat
     //try column
     row:= row + 1
     if row  > 8
        then
           successful  := false
        else
          successful :=  safe (row, column)
     endif
```

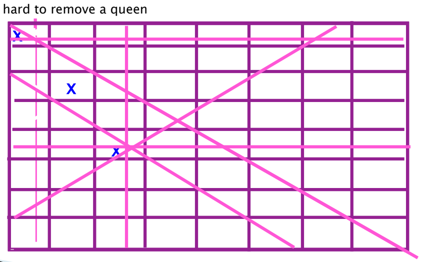
```
if successful then
   update partial configuration
      //inc to next column
      column := column + 1
else backtrack
endif
until last column done or
backtracking done
```

must now make decisions about data structures

- alternative 1
  - board [row, column]
    8x8 Boolean matrix
  - set and remove queen are easy
  - safe requires checking all squares in column, row and diagonals
  - safe done frequently

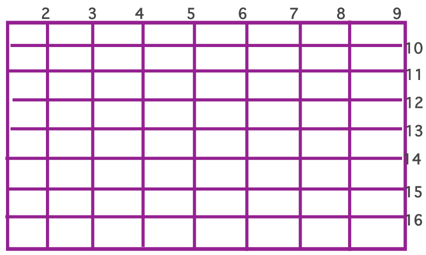
alternative 2

- when setting a queen, mark all unusable squares
- safe is easy
- hard to remove a queen

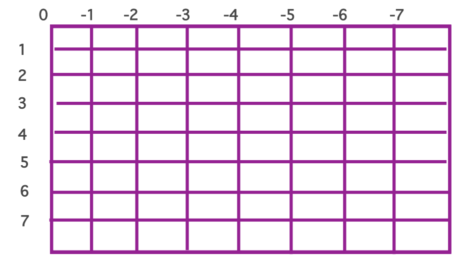


alternative 3

- integer columnselected [1:8]
  - columnselected(i) is row selected for column i
- boolean rowempty[1:8]
  - rowempty (i) = T if there is no queen in row i
- boolean uptothe right[2:16]
  - uptotheright (i) = T if there is no queen in the ith right diagonal
  - row + column = diagonal
- boolean uptothe left[-7:7]
  - uptotheleft (i) = T if there is no queen in the ith left diagonal
  - row-column = diagonal



row + column = diagonal



row-column = diagonal

Third alternative solution with refinements

```
//initialize to first column
    column :=1
    row:= 0
Repeat
   //try column
   row:= row + 1
   if row  > 8
      then
       successful  := false
    else
      successful :=  safe (row, column)
   endif
```

```
    if successful then
        update configuration
         //increment to next
        column
         column := column + 1
    else backtrack
    endif
until last column done or
   backtracking done
```

Refinements

```
//safe (row, column)
    safe :=  rowempty (row) and
        uptothe right(row + column)
    and
        uptothe left (row-column)

//update configuration
     columnselected(column) := row
     rowempty (row) := false
     uptothe right(row + column) :=
         false
     uptothe left (row-column) :=
        false
```

```
//backtrack
     //remove queen
     column := column-1
     If column >= 0 then
       row :=columnselected(column)
       columnselected(column) :=0
       rowempty (row) := true
       uptothe right(row + column) :=
            true
       uptothe left (row-column) :=true
     endif
```

Problems with the proposed solution

- could be more general
  - what if the board size changes
- could be better decomposed
  - could provide methods to set configuration and to backtrack

Third alternative solution with improvements

```
//initialize to first column
    column :=1
    row:= 0
    max := 8
Repeat
   //try column
   row:= row + 1
   if row  > max
      then
       successful  := false
    else
      successful :=  safe (row, column)
   endif
```

```
    if successful then
        update configuration
         //increment to next column
         column := column + 1
    else backtrack(column, row,
                  no  solution)
    endif
until column > max or no solution
```

Refinements

```
backtrack (column, row, no solution)
    //remove queen
    column := column - 1
    no solution := false
    If column > 0 then
        row := columnselected (column)
        columnselected (column) :=0
        rowempty (row) := true
        uptothe right (row + column) := true
        uptothe left (row-column) :=true
    else no solution := true
    endif
```

Low Level Design Summary

- requires an exploration of the implementation alternatives
  - stepwise refinement is a general problem solving approach
- investigate each alternative until
  - the best approach becomes clear or
  - resources demand that you make your best guess
    - plan for future change if this choice must be revisited
- document your decisions for future developers
- consider data structures and algorithms together
  - consider kinds and frequency of operations
  - consider efficiency at the algorithm and data structure level

Optimization

- Very hard to predict where the bottlenecks will be ahead of time
- Proposed strategy
  - Create a clean decomposition of the system
    - Easier to develop and maintain
    - If performance is a problem, evaluate for optimizations later
  - For each low-level component, can select an appropriate data structure
    - Easy to optimize later

Optimizations

- Based on changes to low level structures are relatively easy to implement
- Based on high level decomposition are usually much more costly and can reduce future extensibility