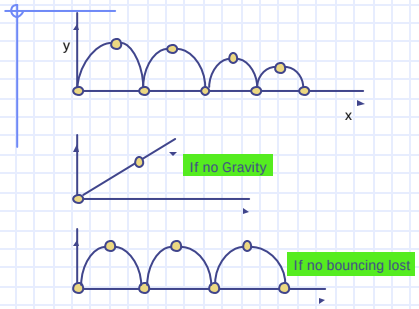
Problem Solving

- Understand the problem
- Math model
- Analysis: top down or bottom up
- Develop an algorithm
- Implement in a programming language

The Bouncing Ball Problem

- Calculate and draw the path of a ball for several bounces along a level floor, given its initial thrown angle and velocity.
- Assume that the external force is gravity and the acceleration is 9.8 m/s^2.
- Assume that 10% of the vertical velocity is lost when the ball bounces, but no loss in the horizontal velocity.

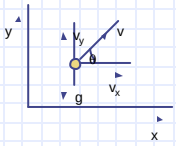


Problem Discussion:

- $v_x = v \cos \theta$
- $v_y = v \sin \theta$

$V_y(t + \Delta t) = y(t) + g^* \Delta t$

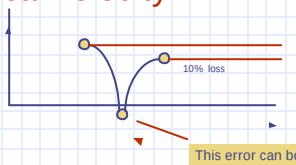
Calculate new x, y coordinates from t → t + Δt



$x(t + \Delta t) = x(t) + V_x(t)^* \Delta t$

$y(t + \Delta t) = y(t) + (V_y(t + \Delta t) + V_x(t))/2 * \Delta t$

Vertical velocity



This error can be ignored If Δt is small enough, or use Zero-finding algorithm for accuracy.

if $(y(t + \Delta t) < 0)$

$v_y(t + \Delta t) = v_y(t + \Delta t) * 0.9$

From Math Model to Program

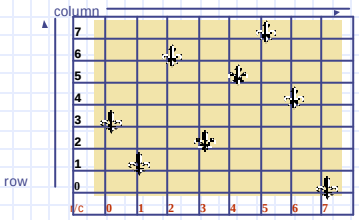
```
Function bouncing_ball(v, degree, n, deltat)
Inputs:  v: initial velocity; degree: throwing angle;
         n: number of bounces; deltat: time increment;

   radians = degree*3.1415926/180;
   vx = v*cos(radians);          /* initial x-velocity */
   vy_old = v*sin(radians);      /* initial y-velocity */
   x = 0;                        /* initial x coordinate */
   y = 0;                        /* initial y coordinate */
   g = 9.8;                      /* gravity */
   j = 0;                        /* initial bounce counter */
```

```
while (j < n){
        draw(x, y);
        x = x + vx*deltat;
        vy_new = vy_old + g*deltat;
        y = y + deltat *(vy_old + vy_new)/2;
        vy_old = vy_new;
        if (y < 0){
                vy_old = abs(0.9*vy_old);
                y = 0;
                j++;
        }
}
```


Eight Queens Problem:

- To place eight Queens on an empty chess board such that no Queen attacks and other Queens.



Math Problem:

- If arrange Queens on the row-wise, no two Queens on the same row for each arrangement.
  - first arrange row 0; if succeed;
  - arrange row 1; and so on;
  - however, if an arrangement at row i failed, we rollback to previous row to rearrange.

Math Problem:

- For column-wise, no attack conditions:

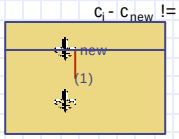
(1) No two Queens on the same column;

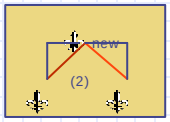
c_{new} != c_j for j = 0..n-1

(2) No two Queens on the same diagonal.

$c_{new} - c_j$!= $r_{new} - r_j$ and

$c_j - c_{new}$!= $r_{new} - r_j$





let $d = c_{new} - c_j$

then the negation conditions (attack) are:

(1) d == 0; /* same column */

(2) (r_{new} == r_i + d) or (r_{new} == r_i – d)/* same diagonal */

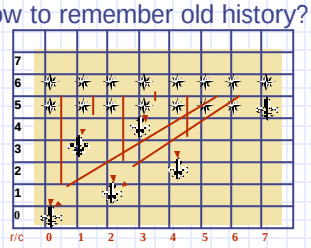
that is:

if ((d == 0) || (r_{new} == r_i + d) || (r_{new} == r_i – d))

{ attack!}

Backtracking Case:

- How to remember old history?



Programming:

```
int board[8][8]; /* a chess board, 0 not used, 1 occupied */

int find(int r){        /* find a Queen in a given row, returns column */
    for (int c = 0; c <= 7; c++)
        if (board[r][c]) return c;
}

int no_attack( int r_new;  int c_new){
int c, r, d;
    if (r_new == 0) return 1;
    for (r = r_new – 1; r >= 0; r--){ /* starting from the row below r_new */
        c = find( r);
        d = c_new  – c;
        if (( d==0) || (r + d == r_new) || (r – d) == r_new))
                return 0;
    }
    return 1;
}
```



```
int addQueen(int row){
int col, good;
    if (row > 7) return 1;                      /* recursion termination, done */
    col = 0;
    good = 0;
    while ((col < = 7) && !good){
        board[row][col] = 1;                    /* we try to place here */
        if (no_attack(row, col))
            good = addQueen(row + 1);   /* so far so good, try next */
        if (!good){             /* either this does not work, or the try failed */
            board[row][col] = 0;            /* give up this */
            col++;                          /* how about next column? */
        }
    }
    return good;
}

void main(){
    init_board();                               /* all zeros */
    if (addQueen(0)) print_board();
    else printf("%s", "no solution!  ");
}
```


Refine Program:

```
#include <stdio.h>
#include <stdlib.h>
 int x[8]; /* x[i] = x coordinate of queen in row i. */

 /* prints field */
void print () {
  int i, j;
        printf ("+----------+\n");
        for (i=0;i<8; i++) {
                printf ("|");
                for (j=0;j<8; j++)
                        if (j==x[i]) printf ("<>");
                        else printf (" ");
                printf ("|\n");
        }
        printf ("+----------+\n\n");
}
```



```
/* tests, whether (ix, iy) is beaten by queens 0...(iy-1) */
int is_free (int ix, int iy) {
  int i;
        for (i=0;i<iy; i++)
                if ((x[i]==ix) || (abs(x[i]-ix)==abs(i-iy)))
                        return 0;
        return 1;
}
/* tries to place queen n on row n */
void try (int n) {
  int i;
        if (n==8) print();
        else
                for (i=0;i<8; i++)
                        if (is_free(i,n)) {
                                x[n]=i;
                                try(n+1);
                        }
}
 int main () { try (0); return 0; }
```


it takes over a week for an 800 MHz PC to calculate the number of solutions for a 21 x 21 board. (there are 314,666,222,712 solutions.)

# queens	# solutions
1	1
2	0
3	0
4	2
5	10
6	6
7	40
8	92
9	352

# queens	# solutions
10	724
11	2680
12	14200
13	73712
14	365596
15	2279184
16	14772512
17	95815104
18	666090624

Advanced Recursion

- Directly recursive function P, is one in which P calls itself within itself.
- Mutually recursive functions, say P1, P2, P3, P4, is one in which P1 calls P2, P2 calls P3, P3 calls P4, and P4 calls P1 again, in a circular fashion.

Recursion as a Descriptive Method

- By an example of a Context-Free Grammar(CFG) for arithmetic expressions.
  - E: Expression
  - T: Term
  - F: Factor
  - P: Primary

CFG: Production Rules

Production rules are a set of substitution rules that allow us to replace a symbol on the left of → with the symbols on the right.

left_symbol → right_symbols

For example, CFG for arithmetic expressions:

E → E + T	E → E – T	E → T
T → T * F	T → T / F	T → F
F → F ↑ P	F → P /* ↑ exponentiation */	
P → (E)	P → a	

Example:

Rewrite right-most	Applying Rule
(1) E	E → T
(2) T	T → T/F
(3) T/F	F → P
(4) T/P	P → (E)
(5) T/(E)	E → E – T
(6) T/(E – T)	T → F
(7) T/(E – F)	F → P
(8) T/(E – P)	P → a
(9) T/(E – a)	E → T
(10) T/(T – a)	T → F

Rewrite	Applying Rule
(11) T/(F – a)	F → P
(12) T/(P – a)	P → a
(13) T/(a – a)	T → T*F
(14) T*F/(a – a)	F → F ↑ P
...	...
(n) a*a ↑a/(a – a)	no more substitution

sentential form

symbol a is called terminal symbol whereas E,T,F,P are nonterminal symbols. From a nonterminal symbol generating a sentential form expression is called production

Definition:

- A Grammar G is defined as

  $\{S, V_N, V_T, P\}$

  where S is the start symbol (such as S = E)

  V_N is the set of nonterminals (V_N = {E, T, F, P}

  V_T is the set of terminals (V_T = {+, -, *, /, ↑}

  P is the set of productions
- The Language L(G) generated by G is the set of all such sentences that can be generated from S using P.

Recursion in P:

- Directly Recursive

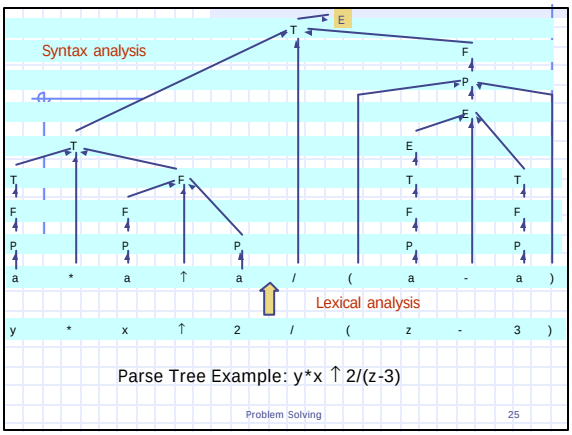
  E → E – T T → T * F

- Indirectly (Mutually) Recursive

  E → T T → F F → P P → (E)

Parsing:

- The process of parsing a sentence is to recognize if it belongs to the language L(G).
- Passing a sentence is the reverse of the process generating it.
- The first two phases of a Compiler are lexical analysis followed by syntax analysis.



Parse Tree Example: y*x ↑ 2/(z-3)

Railroad Diagrams

E → E + T | E – T | T

F → F↑P | P

T → T * F | T / F | F

P → (E) | a

Generating a recursive Parser

- Parser class: Shift-reduce parsers with a lookahead of 1.
- Three basic actions:
  shift: move to the next token
  reduce: apply production rule
  lookahead: examine the next token

Example:

String being parsed:	Remainder of input:	
T*F ↑	2/(z – 3) next == 2	since we can not reduce the the these rightmost lexemes(string being parsed), we shift the next input
T*F ↑2	/(z – 3) next == /	since the rightmost lexeme '2' is a terminal "a" , form rule P→ a, call reduce(1, P), which replaces the last lexeme to "P"
T*F ↑P	/(z – 3) next == /	the rightmost F ↑P is the production of F → F ↑P , call reduce(3, F), which replaces the last 3 lexemes to "F"
T*F	/(z – 3) next == /	the rightmost T * F is the production of T → T * F , call reduce(3, T), which replaces the last 3 lexemes to "T"

Example:

String being parsed:	Remainder of input:	
T	/(z – 3) next == /	since we still have remainder of input, so we shift the next input
T/	(z – 3) next == (	keep going...

Recursive Descent Parser Program :

```
/* Assume that the variable Next and the functions Reduce(k,L) */
/* and Shift(c) are defined externally. */
extern void ParseE(void);       /* extern needed for functions that are */
                                /* called before they are declared */

void ParseP(void)               /* parse a Primary */
{
    if (Next == '(') {
        Shift('(');
        ParseE();
        Shift(')');
        Reduce(3,'P');          /* P --> ( E ) */
    } else {
        Shift('a');             /* 'a' stands for any letter or digit */
        Reduce(1,'P');          /* P --> a */
    }
}
```

```
void ParseF(void)                       /* parse a Factor */
{
    ParseP();
    Reduce(1,'F');                      /* F --> P */
    while (Next == '^') {
        Shift(Next);
        ParseP();
        Reduce(3,'F');                  /* F --> F ^ P */
    }
}

void ParseT(void)                       /* parse a Term */
{
    ParseF();
    Reduce(1,'T');                      /* T --> F */
    while ((Next == '*') || (Next == '/')) {
        Shift(Next);
        ParseF();
        Reduce(3,'T');                  /* T --> T * / F */
    }
}
```



```
void ParseE(void)                       /* parse an Expression */
{
    ParseT();
    Reduce(1,'E');                      /* E --> T */
    while ((Next == '+') || (Next == '-')) {
        Shift(Next);
        ParseT();
        Reduce(3,'E');                  /* E --> E +- T */
    }
}

int main(void)
{
    do {
        GetInputFromUser();
        ParseE();           /* call the parser to parse the input string */
    } while (UserWantsToContinue());
}
```


Utilities functions:

```
typedef StackType Stack;        /* the StackType will depend on the representation */

extern void InitializeStack (Stack *S);  /* Initialize the stack S to be the empty stack */

extern int Empty (Stack *S);        /* Returns TRUE if and only if the stack S is empty */

extern int Full (Stack *S);         /* Returns TRUE if and only if the stack S is full */

extern void Push (ItemType X, Stack *S); /* If S is not full push X onto stack S */

extern void Pop (Stack *S, ItemType *X); /* If S is non-empty, pop an item off the */
                                         /* top of the stack S and save it in X */
```



```
extern typedef enum {false, true} Boolean;

/* gets the Input string from the user          */
extern void GetInputStringFromUser(void);

/* pops n items off Stack, then pushes S on Stack */
extern void Reduce(int n, char S);

/* reads c from Input and pushes c on Stack      */
extern void Shift(char c);

/* returns "true" iff user wants to continue     */
extern Boolean UserWantsToContinue(void);

char   *Answer = "blank",       /* a reply from the user */
       *InputString             /* the input string */
        = "a blank input string long enough for an expression!";

char *FetchNext(void);        /* returns next non-blank character in the Input */

void PrintErrorMessageIfNecessary(char c, char d);  /* write a Shift error    */
                                                    /* message, if c != d     */
```



```
char    Next;                   /* the next character in the input string */

Stack  InputStack, ParseStack;

void GetInputStringFromUser(void)     /* gets an Input string from user to parse */
{
    int i;

    InitializeStack(&ParseStack);  /* initialize the ParseStack to the empty stack */
    InitializeStack(&InputStack);  /* initialize the InputStack to the empty stack */
    Push('#',&InputStack);         /* put a 'pad' character at bottom of InputStack */
    printf("give input string: ");
    gets(InputString);                      /* scanf("%s", InputString); */
    for (i=strlen(InputString)-1; i>=0; --i) Push(InputString[i], &InputStack);
    Next = '#';
    Shift('#');                             /* get things started */
}
```



```
char FetchNext(void)            /* returns next non-blank character in the Input */
{
  char d, f;
  d = Next;                     /* find the next non-space character in the Input */
  do {                          /* find the next non-space character in the Input */
     Pop(&InputStack,&f);
  } while (f == ' ');
  Next = f;             /* let Next be this next non-space character in the Input */
  return d;                     /* return d as the value of the function */
}

void PrintErrorMessageIfNecessary(char c, char d)
{
/* if d isn't equal to the expected character c, write an error message. */
  if (c == 'a') {           /* recall that 'a' stands for any <letter> or <digit> */
     if (! (islower(d) || isdigit(d)) ) {
        printf("Syntax Error: expected <letter> or <digit> but got %c\n",d);
     }
  } else if (c != d) {
     printf("Syntax Error: expected %c but got %c\n",c,d);
  }
}
```

```
int UserWantsToContinue(void)
 {
   printf("Do you want to give another string? (y/n) ");
   gets(Answer);       /* scanf("%s",Answer); */
   return (Answer[0] == 'y');
 }

void Shift(char c)
 {
   char d;                    /* d holds the character Shifted from the Input string */
   d = FetchNext();           /* read the next character d from the Input string */
   Push(d, &ParseStack);                          /* push d on top of ParseStack */
   PrintErrorMessageIfNecessary(c, d); /* if c != d then print Shift error message */
 }
```



```
void Reduce(int n, char S)                    /* pop n items from Stack, push S  */
 {
   int i; char *Output;
     Output = "                ";
     Output[2*n] = '\0';
     for (i=n-1; i>=0; --i) {
       Pop(&ParseStack, &Output[2*i]); /* Output example: " T * F \0"  */
       Output[2*i+1] = ' ';
     }
     if (n == 1) {
       if (S == 'P') {
         printf("  %c  -->    a     { where  a = %s}\n", S, Output);
       } else {
         printf("  %c  -->    %s\n", S, Output);
       }
     } else {
       printf("  %c  -->  %s\n", S, Output);
     }

   /* push S on stack */

       Push(S,&ParseStack);
 }
```