USC Viterbi
School of Engineering

CSCI 104
Recursion

Mark Redekopp
David Kempe

USC

Recursion

- Problem in which the solution can be expressed in terms of itself (usually a smaller instance/input of the same problem) ***and a base/terminating case***
- Input to the problem must be categorized as a:
  - Base case: Solution known beforehand or easily computable (no recursion needed)
  - Recursive case: Solution can be described using solutions to smaller problems of the same type
    - Keeping putting in terms of something smaller until we reach the base case
- Factorial: n! = n * (n-1) * (n-2) * … * 2 * 1
  - n! = n * (n-1)!
  - Base case: n = 1
  - Recursive case: n > 1 => n*(n-1)!

Recursive Functions

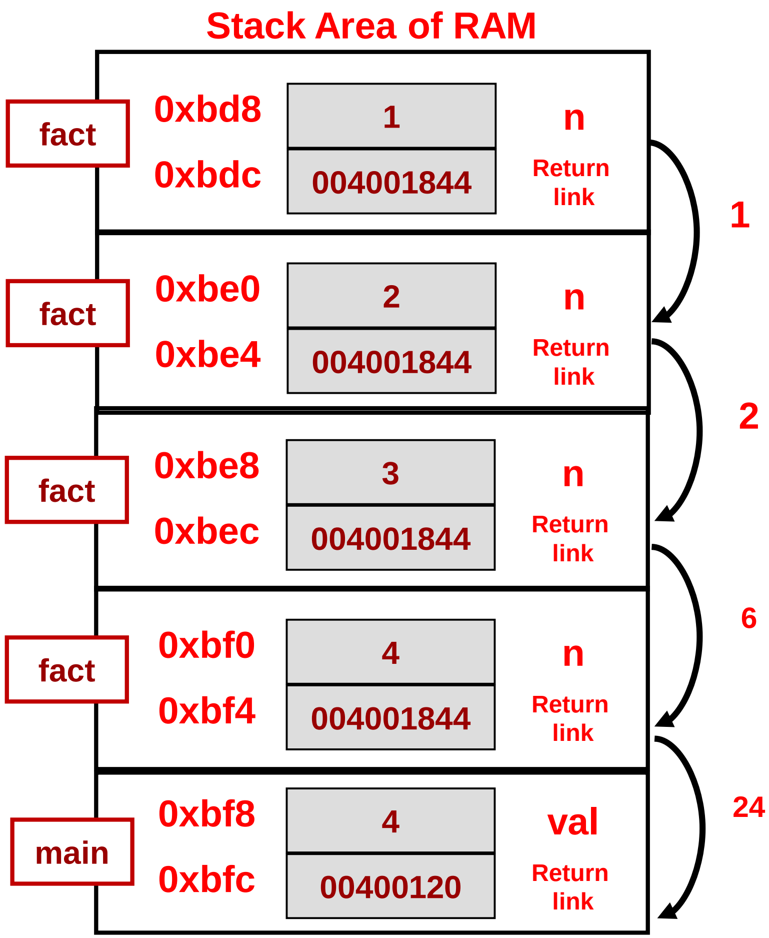
- Recall the system stack essentially provides separate areas of memory for each 'instance' of a function
- Thus each local variable and actual parameter of a function has its own value within that particular function instance's memory space

C Code:

```
int fact(int n)
{
   if(n == 1){
      // base case
      return 1;
   }
   else {
      // recursive case
      return = n * fact(n-1);

   }
}
```

Recursion & the Stack

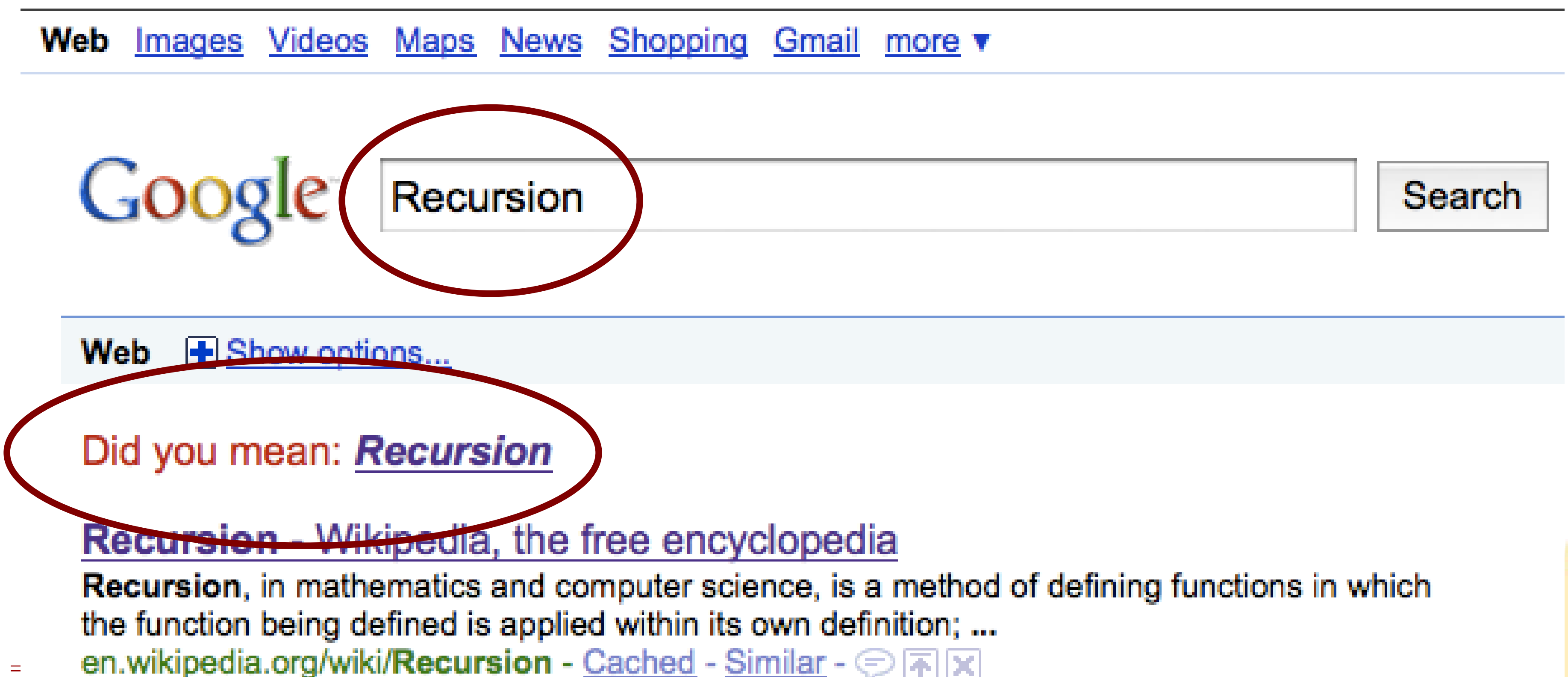
- Must return back through the each call




```
int fact(int n)
{
  if(n == 1){
     // base case
     return 1;
  }
  else {
     // recursive case
     return = n * fact(n-1);
  }
}


int main()
{
  int val = 4;
  cout << fact(val) << endl;
}
```

Recursion

- **Google is in on the joke too...**



Recursive Functions

- Many loop/iteration based approaches can be defined recursively as well

C Code:

```
int main()
{
  int data[4] = {8, 6, 7, 9};
  int size=4;
  int sum1 = isum_it(data, size);
  int sum2 = rsum_it(data, size);
}

int isum_it(int data[], int len)
{
  sum = data[0];
  for(int i=1; i < len; i++){
    sum += data[i];
  }
}

int rsum_it(int data[], int len)
{
  if(len == 1)
    return data[0];
  else
    int sum = rsum_it(data, len-1);
    return sum + data[len-1];
}
```

Recursive Call Timeline

```
int main(){
  int data[4] = {8, 6, 7, 9};
  int size=4;
  int sum2 = rsum_it(data, size);
  ...
}
```

```
int rsum_it(int data[], int len)
{
  if(len == 1)
    return data[0];
  else
    int sum = rsum_it(data, len-1);
    return sum + data[len-1];
}
```

Time

len = 4

rsum_it(data,4)
int sum=
rsum_it(data,4-1)

len = 3

rsum_it(data,3)
int sum=
rsum_it(data,3-1)

len = 2

rsum_it(data,2)
int sum=
rsum_it(data,2-1)

len = 1

rsum_it(data,1)
return data[0];

8

int sum = 8
return 8+data[1];

14

int sum = 14
return 14+data[2];

21

int sum = 21
return 21+data[3];

30

Each instance of rsum_it has its own len argument and sum variable

Every instance of a function has its own copy of local variables

USC Viterbi School of Engineering

System Stack & Recursion

- The system stack makes recursion possible by providing separate memory storage for the local variables of each running instance of the function

```
int main()
{
  int data[4] = {8, 6, 7, 9};
  int size=4;
  int sum2 = rsum_it(data, size);
}

int rsum_it(int data[], int len)
{
  if(len == 1)
    return data[0];
  else
    int sum =
          rsum_it(data, len-1);
    return sum + data[len-1];
}
```

System Memory

(RAM)

Code for all functions
Data for rsum_it (data=800, len=1, sum=??) and return link
Data for rsum_it (data=800, len=2, sum=8) and return link
Data for rsum_it (data=800, len=3, sum=14) and return link
Data for rsum_it (data=800, len=4, sum=21) and return link
Data for main (data=800, size=4, sum1=??,sum2=??) and return link
System stack area

800

	0	1	2	3
data[4]:	8	6	7	9

USC

Head vs. Tail Recursion

- Head Recursion: Recursive call is made before the real work is performed in the function body
- Tail Recursion: Some work is performed and then the recursive call is made

Tail Recursion

```
void doit(int n)
{
  if(n == 1) cout << "Stop";
  else {
   cout << "Go" << endl;
   doit(n-1);
  }
}
```

Head Recursion

```
void doit(int n)
{
  if(n == 1) cout << "Stop";
  else {
   doit(n-1);
   cout << "Go" << endl;
  }
}
```

Head vs. Tail Recursion

- Head Recursion: Recursive call is made before the real work is performed in the function body
- Tail Recursion: Some work is performed and then the recursive call is made

Tail Recursion

```
Void doit(int n)
{
  if(n == 1) cout << "Stop";
  else {
   cout << "Go" << endl;
   doit(n-1);
  }
}
```

doit(3) Go	return	Go Go Stop
doit(2) Go	return	
doit(1) Stop	return	

Head Recursion

```
Void doit(int n)
{
  if(n == 1) cout << "Stop";
  else {
   doit(n-1);
   cout << "Go" << endl;
  }
}
```

doit(3)	Go return	Stop Go Go
doit(2)	Go return	
doit(1) Stop	return	

Head or Tail

```
int main()
{
  int data[4] = {8, 6, 7, 9};
  int size=4;
  int sum2 = rsum_it(data, size);
}

int rsum_it(int data[], int len)
{
  if(len == 1)
    return data[0];
  else
    int sum =
         sum_them(data, len-1);
    return sum + data[len-1];
}
```

Head

```
int main()
{
  int data[4] = {1, 6, 7, 9};
  int target = 3
  bsearch(data,0,4,target);
}

int bsearch(int data[],
            int start, int end,
            int target)
{
  if(end >= start)
    return -1;
  int mid = (start+end)/2;
  if(target == data[mid])
    return mid;
  else if(target < data[mid])
    return bsearch(data, start, mid,
                   target);
  else
    return bsearch(data, mid, end,
                   target);
}
```

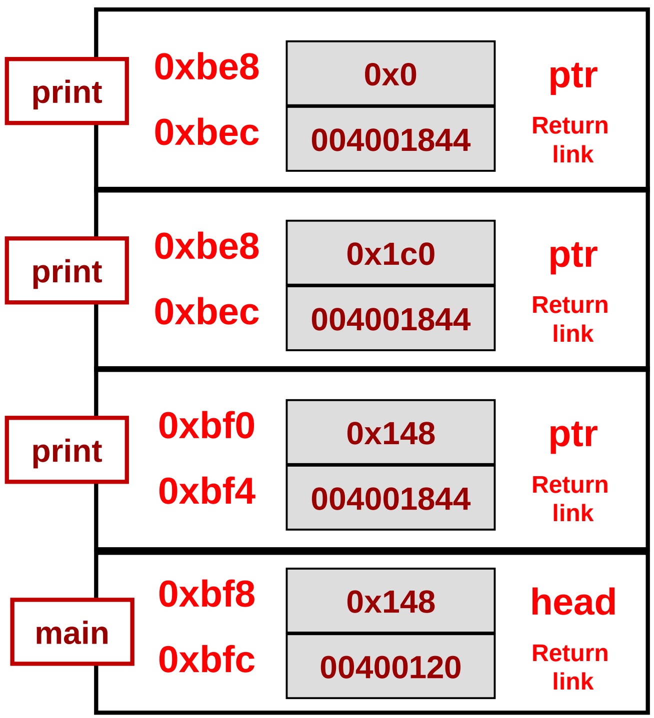
Tail

Loops & Recursion

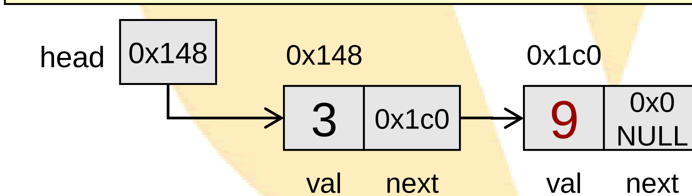
- Is it better to use recursion or iteration?
  - ANY problem that can be solved using recursion can also be solved with iteration and vice versa
  - Usually, a routine with a single recursive call can be implemented just as well or better with iteration
  - When multiple recursive calls are made in the definition (i.e. in a loop), recursion often becomes **much** simpler
- Why use recursion?
  - Usually clean & elegant. Easier to read.
  - Sometimes generates much simpler code than iteration would
  - Sometimes iteration will be almost impossible
- How do you choose?
  - Iteration is usually faster and uses less memory
  - However, if iteration produces a very complex solution, consider recursion

Recursive Operations on Linked List

- Many linked list operations can be recursively defined
- Can we make a recursive iteration function to print items?
  - Recursive case: Print one item then the problem becomes to print the n-1 other items.
    - Notice that any 'next' pointer can be though of as a 'head' pointer to the remaining sublist
  - Base case: Empty list (i.e. Null pointer)
- How could you print values in reverse order?

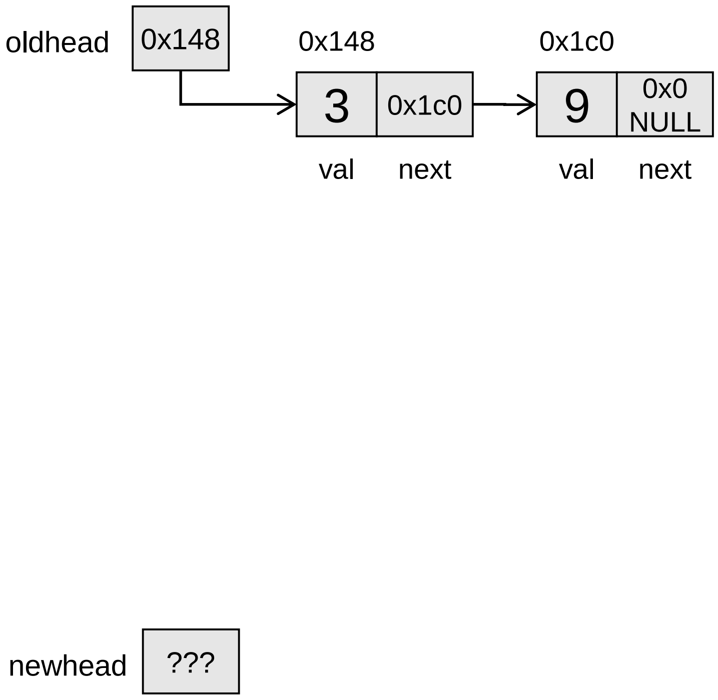



```
void print(Item* ptr)
{
  if(ptr == NULL) return;
  else {
    cout << ptr->val << endl;
    print(ptr->next);
  }
}
int main()
{ Item* head;
  ...
  print(head);
}
```



Recursive Copy

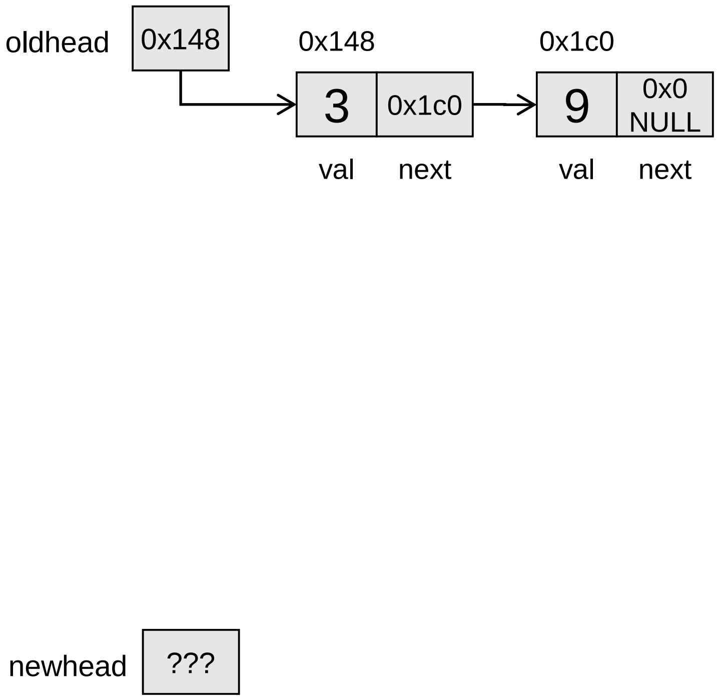
- How could you make a copy of a linked list using recursion




```
struct Item {
 int val;
 Item* next;
 Item(int v, Item* n){
   val = v; next = n;
 }
};

Item* copyLL(Item* head)
{
  if(head == NULL) return NULL;
  else {




  }
}
int main()
{ Item* oldhead, *newhead;
  ...
  newhead = copyLL(oldhead);
}
```

Recursive Copy

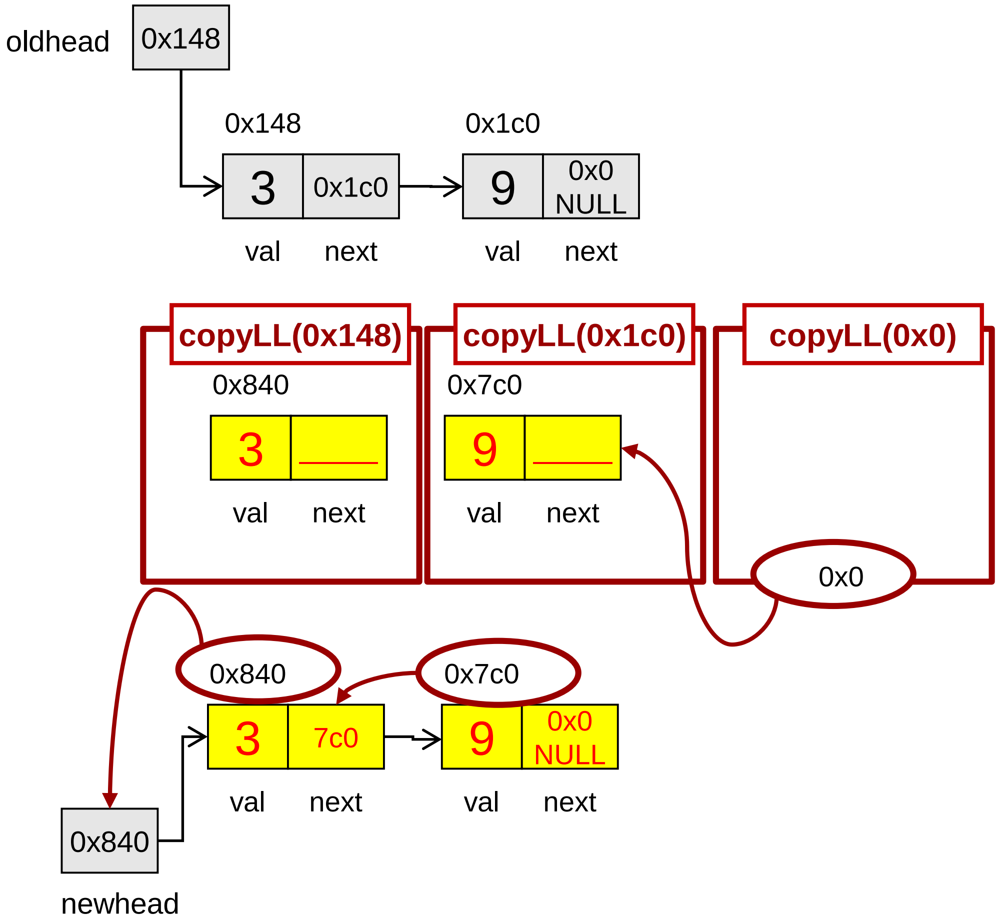
- How could you make a copy of a linked list using recursion




```
struct Item {
 int val;
 Item* next;
 Item(int v, Item* n){
   val = v; next = n;
 }
};

Item* copyLL(Item* head)
{
  if(head == NULL) return NULL;
  else {
    return new Item(head->val,
                    copyLL(head->next));
  }
}
int main()
{ Item* oldhead, *newhead;
  ...
  newhead = copyLL(oldhead);
}
```

Recursive Copy

- How could you make a copy of a linked list using recursion




```
struct Item {
 int val;
 Item* next;
 Item(int v, Item* n){
   val = v; next = n;
 }
};

Item* copyLL(Item* head)
{
  if(head == NULL) return NULL;
  else {
    return new Item(head->val,
                    copyLL(head->next));
  }
}
int main()
{ Item* oldhead, *newhead;
  ...
  newhead = copyLL(oldhead);
}
```

BACKTRACK SEARCH ALGORITHMS

Generating All Combinations

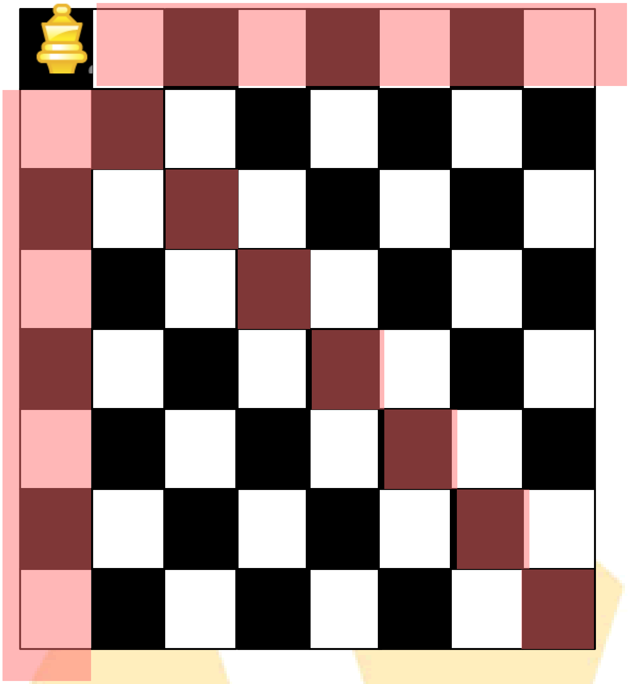
- Recursion offers a simple way to generate all combinations of N items from a set of options, S
  - Example: Generate all 2-digit decimal numbers (N=2, S={0,1,…,9})

Recursive Backtracking Search

- Recursion allows us to "easily" enumerate all solutions to some problem
- Backtracking algorithms…
  - Are often used to solve constraint satisfaction problem or optimization process
    - Several items that can be set to 1 of N values under some constraints
  - Stop searching down a path at the first indication that constraints won't lead to a solution
- Some common and important problems can be solved with backtracking
- Knapsack problem
  - You have a set of objects with a given weight and value. Suppose you have a knapsack that can hold N pounds, which subset of objects can you pack that maximizes the value.
  - Example:
    - Knapsack can hold 35 pounds
    - Object A: 7 pounds, $12 ea. Object B: 10 pounds, $18 ea.
    - Object C: 4 pounds, $7 ea. Object D: 2.4 pounds, $4 ea.
- Other examples:
  - Map Coloring
  - Traveling Salesman Problem
  - Sudoku
  - N-Queens

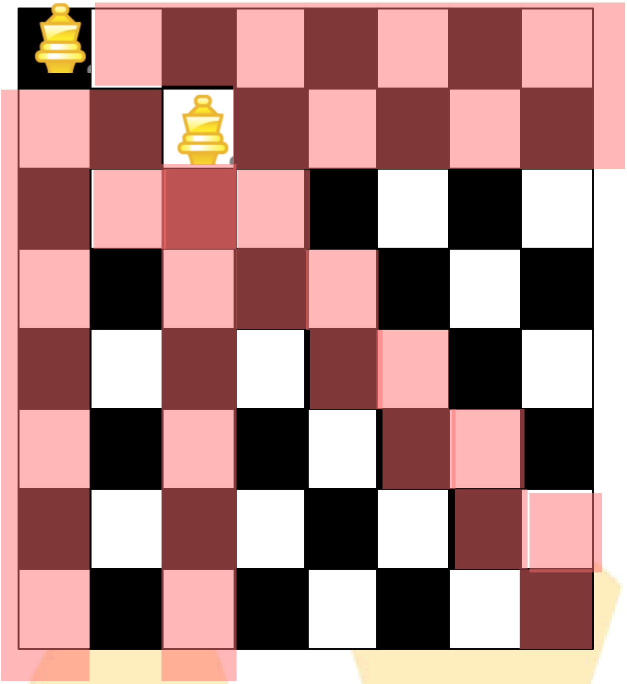
N-Queens Problem

- Problem: How to place N queens on an NxN chess board such that no queens may attack each other
- Fact: Queens can attack at any distance vertically, horizontally, or diagonally
- Observation: Different queen in each row and each column
- Backtrack search approach:
  - Place 1st queen in a viable option then, then try to place 2nd queen, etc.
  - If we reach a point where no queen can be placed in row i or we've exhausted all options in row i, then we return and change row i-1



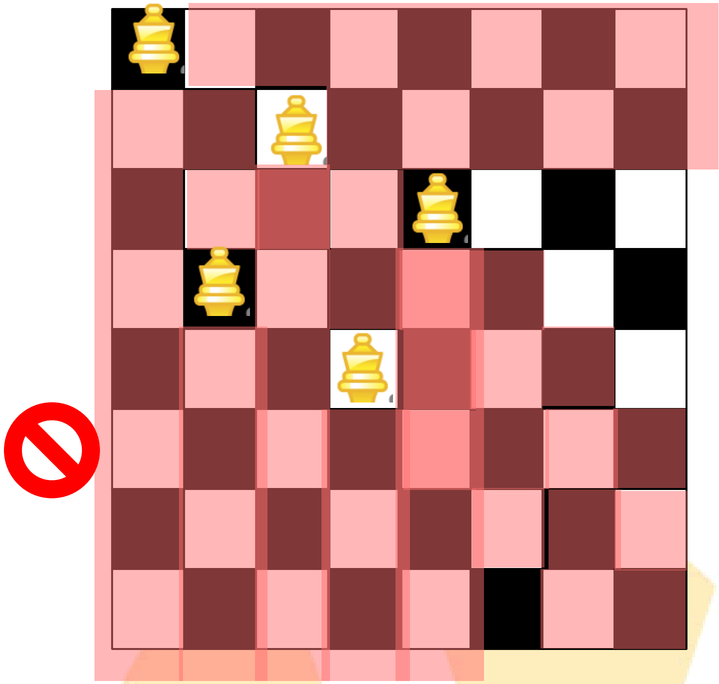
8x8 Example of N-Queens

- Now place 2nd queen



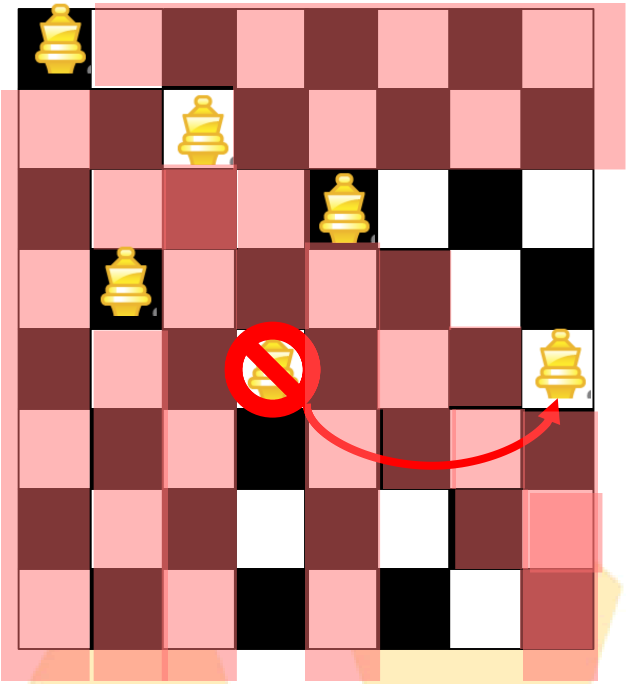
8x8 Example of N-Queens

- Now place others as viable
- After this configuration here, there are no locations in row 6 that are not under attack from the previous 5
- BACKTRACK!!!



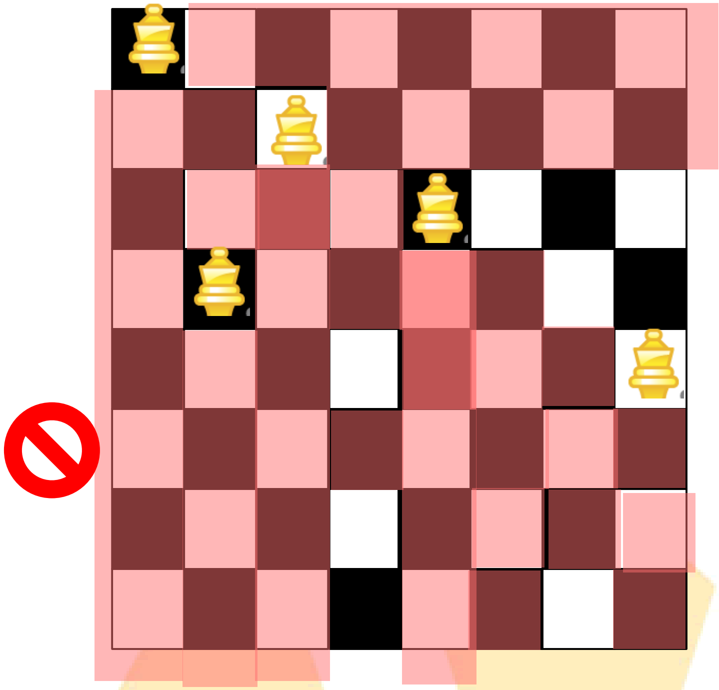
8x8 Example of N-Queens

- Now place others as viable
- After this configuration here, there are no locations in row 6 that is not under attack from the previous 5
- So go back to row 5 and switch assignment to next viable option and progress back to row 6



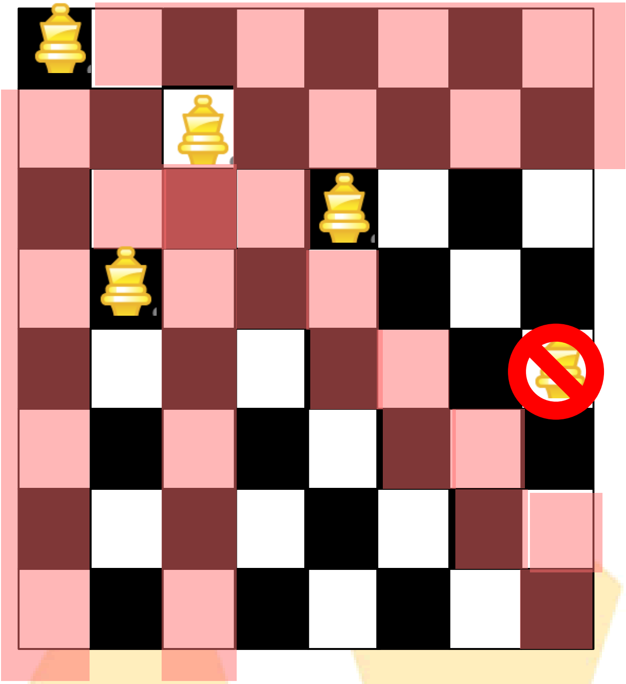
8x8 Example of N-Queens

- Now place others as viable
- After this configuration here, there are no locations in row 6 that is not under attack from the previous 5
- Now go back to row 5 and switch assignment to next viable option and progress back to row 6
- But still no location available so return back to row 5



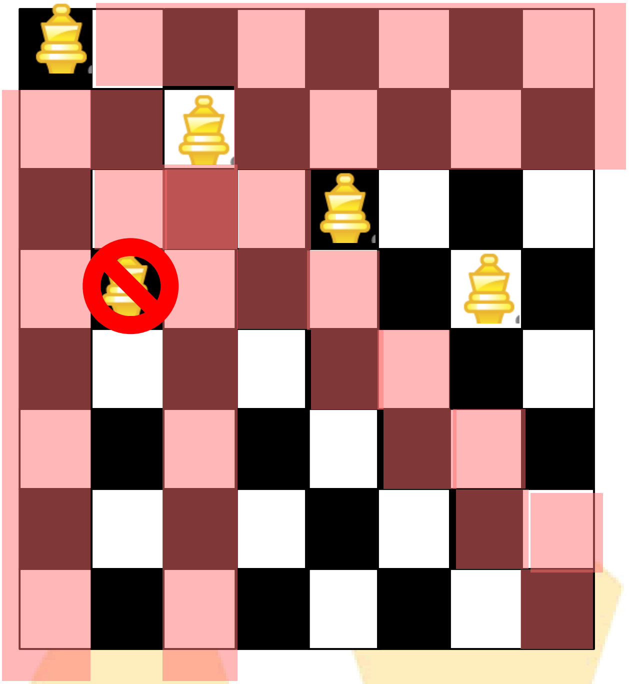
8x8 Example of N-Queens

- Now place others as viable
- After this configuration here, there are no locations in row 6 that is not under attack from the previous 5
- Now go back to row 5 and switch assignment to next viable option and progress back to row 6
- But still no location available so return back to row 5
- But now no more options for row 5 so return back to row 4
- BACKTRACK!!!!



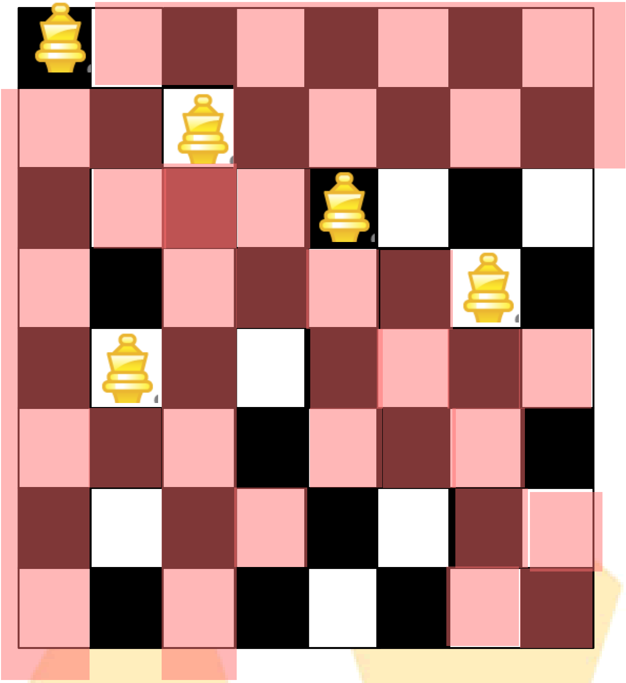
8x8 Example of N-Queens

- Now place others as viable
- After this configuration here, there are no locations in row 6 that is not under attack from the previous 5
- Now go back to row 5 and switch assignment to next viable option and progress back to row 6
- But still no location available so return back to row 5
- But now no more options for row 5 so return back to row 4
- Move to another place in row 4 and restart row 5 exploration



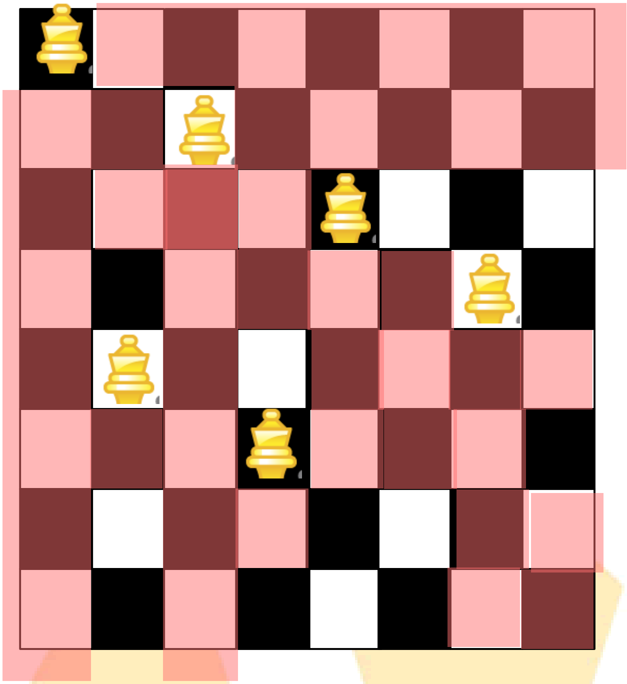
8x8 Example of N-Queens

- Now place others as viable
- After this configuration here, there are no locations in row 6 that is not under attack from the previous 5
- Now go back to row 5 and switch assignment to next viable option and progress back to row 6
- But still no location available so return back to row 5
- But now no more options for row 5 so return back to row 4
- Move to another place in row 4 and restart row 5 exploration



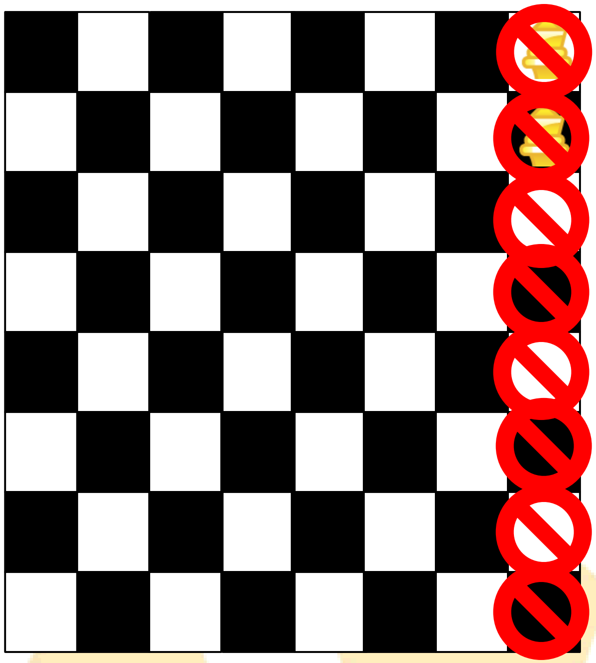
8x8 Example of N-Queens

- Now a viable option exists for row 6
- Keep going until you successfully place row 8 in which case you can return your solution
- What if no solution exists?



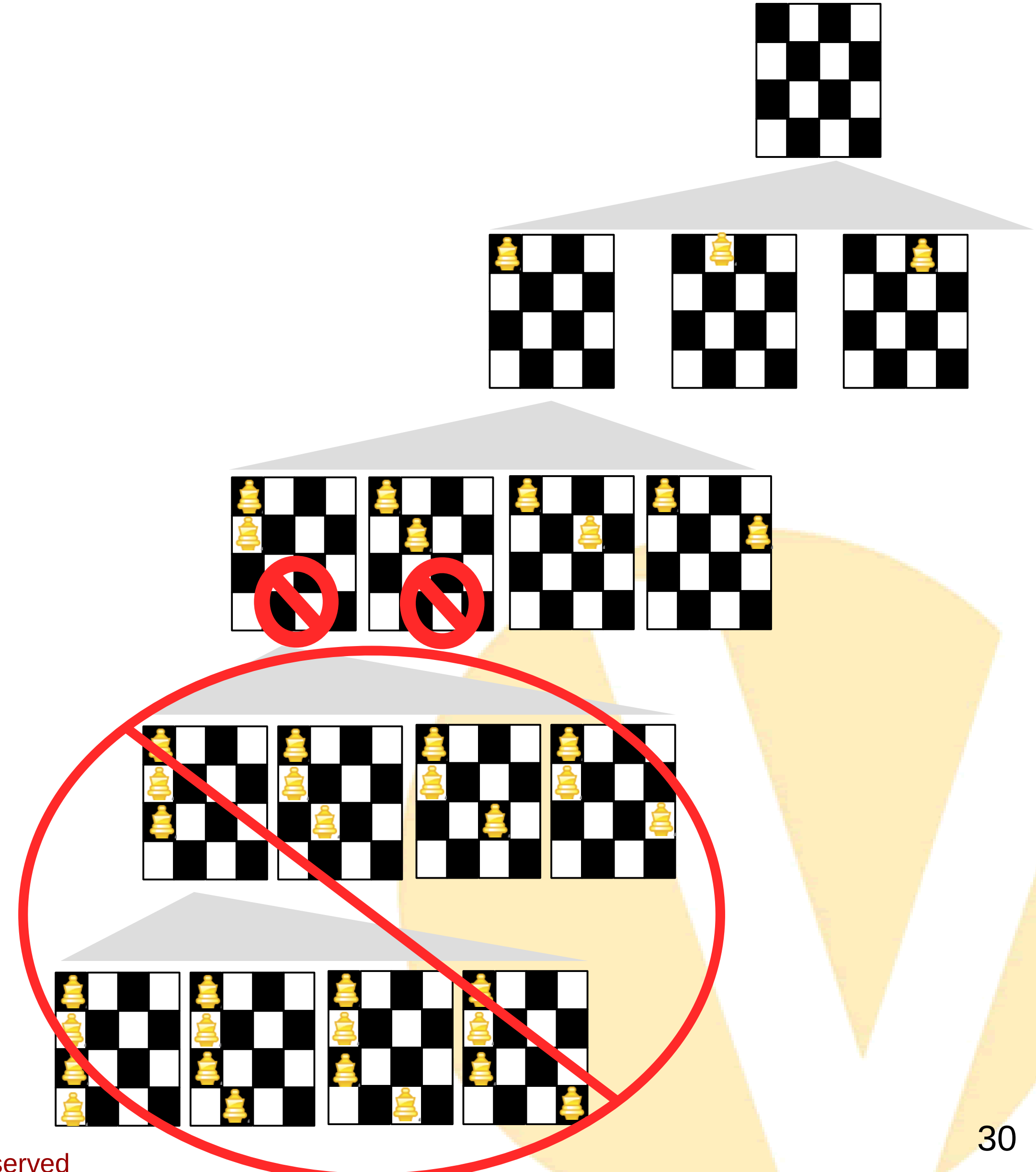
8x8 Example of N-Queens

- Now a viable option exists for row 6
- Keep going until you successfully place row 8 in which case you can return your solution
- What if no solution exists?
  - Row 1 queen would have exhausted all her options and still not find a solution



Backtracking Search

- Recursion can be used to generate all options
  - 'brute force' / test all options approach
  - Test for constraint satisfaction only at the bottom of the 'tree'
- But backtrack search attempts to 'prune' the search space
  - Rule out options at the partial assignment level

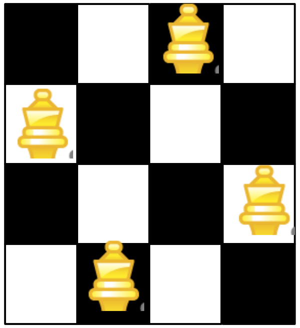


Brute force enumeration might test only once a possible complete assignment is made (i.e. all 4 queens on the board)

N-Queens Solution Development

- Let's develop the code
- 1 queen per row
  - Use an array where index represents the queen (and the row) and value is the column
- Start at row 0 and initiate the search [i.e. search(0)]
- Base case:
  - Rows range from 0 to n-1 so STOP when row == n
  - Means we found a solution
- Recursive case
  - Recursively try all column options for that queen
  - But haven't implemented check of viable configuration…

i
0
1
2
3

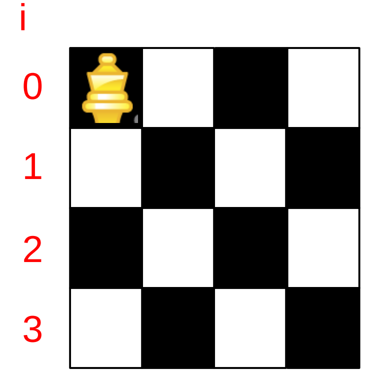


Index = Queen i in row i	0	1	2	3
q[i] = column of queen i	2	0	3	1

```
int *q;   // pointer to array storing
          // each queens location
int n;    // number of board / size

void search(int row)
{
  if(row == n)
    printSolution(); // solved!
  else {
   for(q[row]=0; q[row]<n; q[row]++){
     search(row+1);
  }
}
```

N-Queens Solution Development

- To check whether it is safe to place a queen in a particular column, let's keep a "threat" 2-D array indicating the threat level at each square on the board
  - Threat level of 0 means SAFE
  - When we place a queen we'll update squares that are now under threat
  - Let's name the array 't'
- Dynamically allocating 2D arrays in C/C++ doesn't really work
  - Instead conceive of 2D array as an "array of arrays" which boils down to a pointer to a pointer



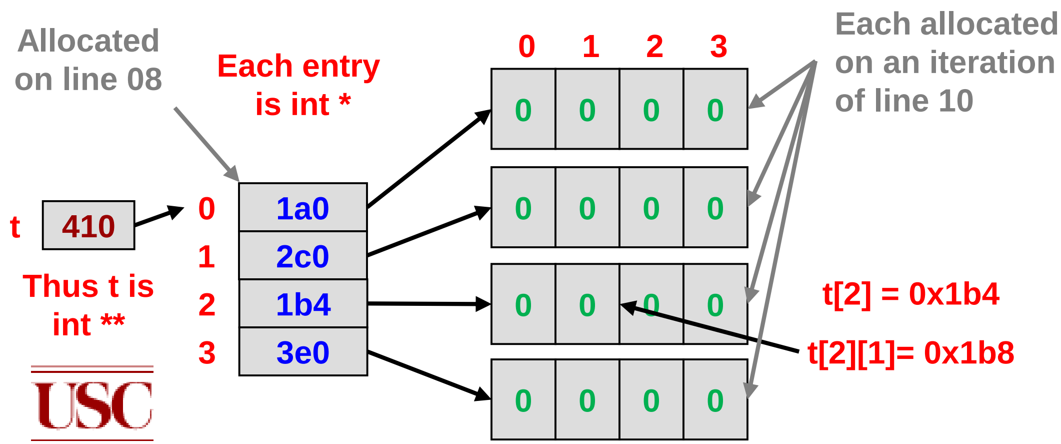
0	1	1	1
1	1	0	0
1	0	1	0
1	0	0	1

Index = Queen i in row i

q[i] = column of queen i

0	1	2	3
0			

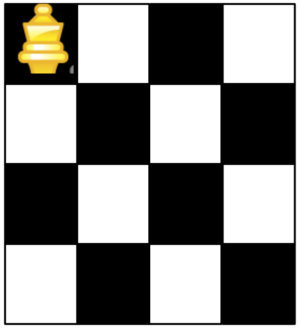
```
00  int *q;   // pointer to array storing
01            // each queens location
02  int n;    // number of board / size
03  int **t;  // thread 2D array
04
05  int main()
06  {
07    q = new int[n];
08    t = new int*[n];
09    for(int i=0; i < n; i++){
10      t[i] = new int[n];
11      for(int j = 0; j < n; j++){
12        t[i][j] = 0;
13      }
14    }
15    search(0); // start search
16    // deallocate arrays
17    return 0;
18  }
```



N-Queens Solution Development

- After we place a queen in a location, let's check that it has no threats
- If it's safe then we "place" it by adding the new threats (+1) assuming we place it there
- Now recurse to next row
- If we return it means the problem was solved, or more often, that no solution existed given our placement so we remove the threats (-1)
- Then we iterate to try the next location for this queen

i



Index = Queen i in row i

q[i] = column of queen i

0	1	2	3
0			

```
int *q;   // pointer to array storing
          // each queens location
int n;    // number of board / size
int **t; // n x n threat array
void search(int row)
{
  if(row == n)
    printSolution(); // solved!
  else {
   for(q[row]=0; q[row]<n; q[row]++){
     // check that col: q[row] is safe
     if(t[row][q[row]] == 0){
       // if safe place and continue
       addToThreats(row, q[row], 1);
       search(row+1);
       // if return, remove placement
       addToThreats(row, q[row], -1);
} } }
```

t	0	1	2	3
0	0	0	0	0
1	0	0	0	0
2	0	0	0	0
3	0	0	0	0

Safe to place queen in upper left

t	0	1	2	3
0	0	1	1	1
1	1	1	0	0
2	1	0	1	0
3	1	0	0	1

Now add threats

t	0	1	2	3
0	0	0	0	0
1	0	0	0	0
2	0	0	0	0
3	0	0	0	0

Upon return, remove threat and iterate to next option

addToThreats Code

- Observations
  - Already a queen in every higher row so addToThreats only needs to deal with positions lower on the board
    - Iterate row+1 to n-1
  - Enumerate all locations further down in the same column, left diagonal and right diagonal
  - Can use same code to add or remove a threat by passing in change
- Can't just use 2D true/false array as a square might be under threat from two places and if we remove 1 piece we want to make sure we still maintain the threat

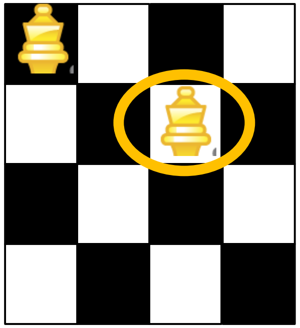
i

0

1

2

3



Index = Queen i in row i

q[i] = column of queen i

0	1	2	3
0			

```
void addToThreats(int row, int col, int change)
{
  for(int j = row+1; j < n; j++){
    // go down column
    t[j][col] += change;
    // go down right diagonal
    if( col+(j-row) < n )
       t[j][col+(j-row)] += change;
    // go down left diagonal
    if( col-(j-row) >= 0)
       t[j][col-(j-row)] += change;
  }
}
```

t	0	1	2	3
0	0	1	1	1
1	1	1	0	0
2	1	0	1	0
3	1	0	0	1

t	0	1	2	3
0	0	1	1	1
1	1	1	0	0
2	1	1	2	1
3	2	0	1	1

N-Queens Solution

```
int *q;   // queen location array
int n;    // number of board / size
int **t; // n x n threat array

int main()
{
  q = new int[n];
  t = new int*[n];
  for(int i=0; i < n; i++){
    t[i] = new int[n];
    for(int j = 0; j < n; j++){
      t[i][j] = 0;
    }
  }
  // do search
  if( ! search(0) )
     cout << "No sol!" << endl;
  // deallocate arrays
  return 0;
}
```

```
void addToThreats(int row, int col, int change)
{
  for(int j = row+1; j < n; j++){
    // go down column
    t[j][col] += change;
    // go down right diagonal
    if( col+(j-row) < n )
       t[j][col+(j-row)] += change;
    // go down left diagonal
    if( col-(j-row) >= 0)
       t[j][col-(j-row)] += change;
  }
}

bool search(int row)
{
  if(row == n){
    printSolution(); // solved!
    return true;
  }
  else {
   for(q[row]=0; q[row]<n; q[row]++){
     // check that col: q[row] is safe
     if(t[row][q[row]] == 0){
       // if safe place and continue
       addToThreats(row, q[row], 1);
       bool status = search(row+1);
       if(status) return true;
       // if return, remove placement
       addToThreats(row, q[row], -1);
     }
    return false;
  }
}
```

General Backtrack Search Approach

- Select an item and set it to one of its options such that it meets current constraints
- Recursively try to set next item
- If you reach a point where all items are assigned and meet constraints, done…return through recursion stack with solution
- If no viable value for an item exists, backtrack to previous item and repeat step 1
- If viable options for the 1st item are exhausted, no solution exists

General Outline of Backtracking
Sudoku Solver

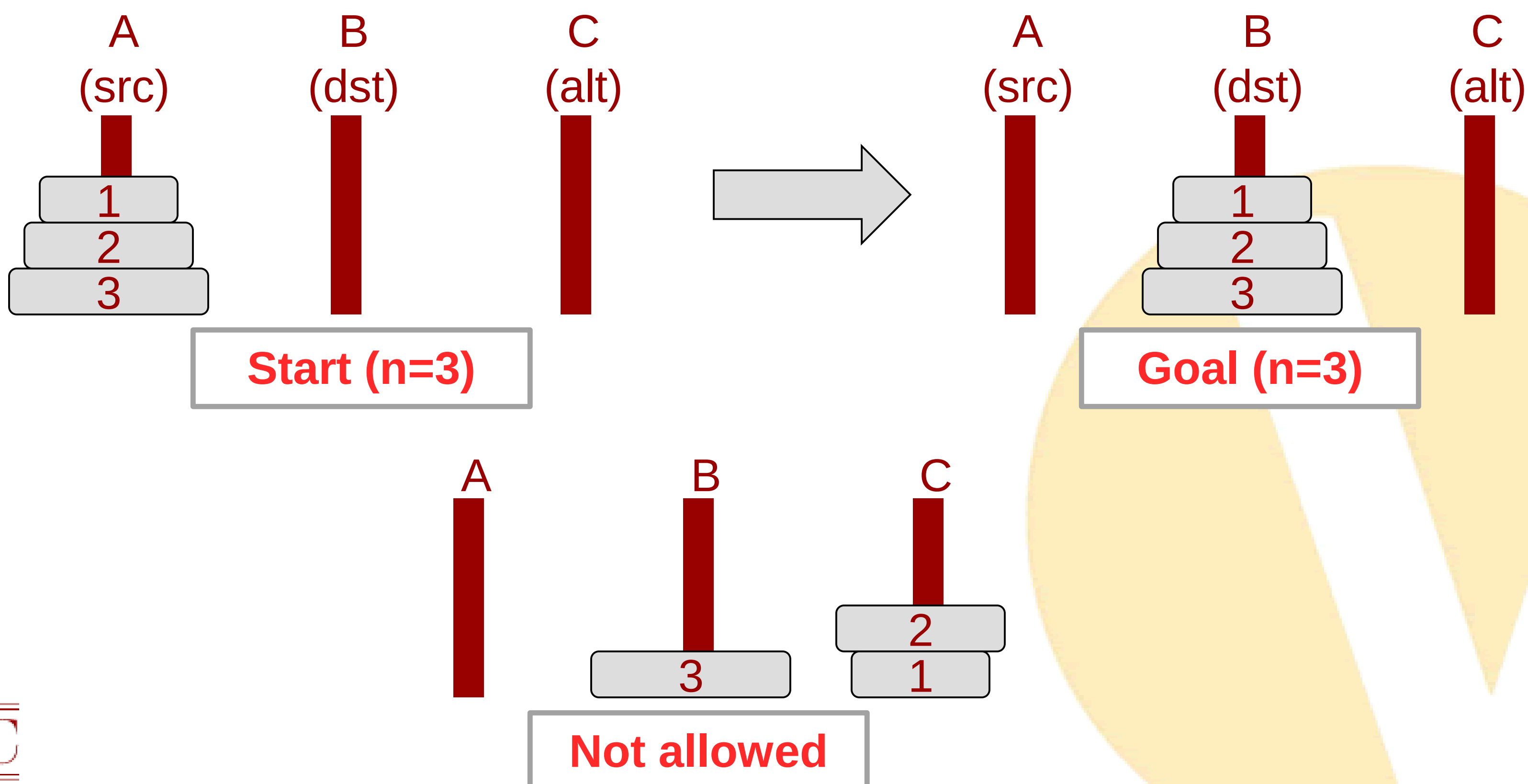
```
bool sudoku(int **grid, int r, int c)
{
  if( allSquaresComplete(grid) )
    return true;
  }
  // iterate through all options
  for(int i=1; i <= 9; i++){
    grid[r][c] = i;
    if( isValid(grid) ){
      bool status = sudoku(...);
      if(status) return true;
    }
  }
  return false;
}
```

Assume r,c is current square to set and grid is the 2D array of values

OTHER RECURSIVE EXAMPLES

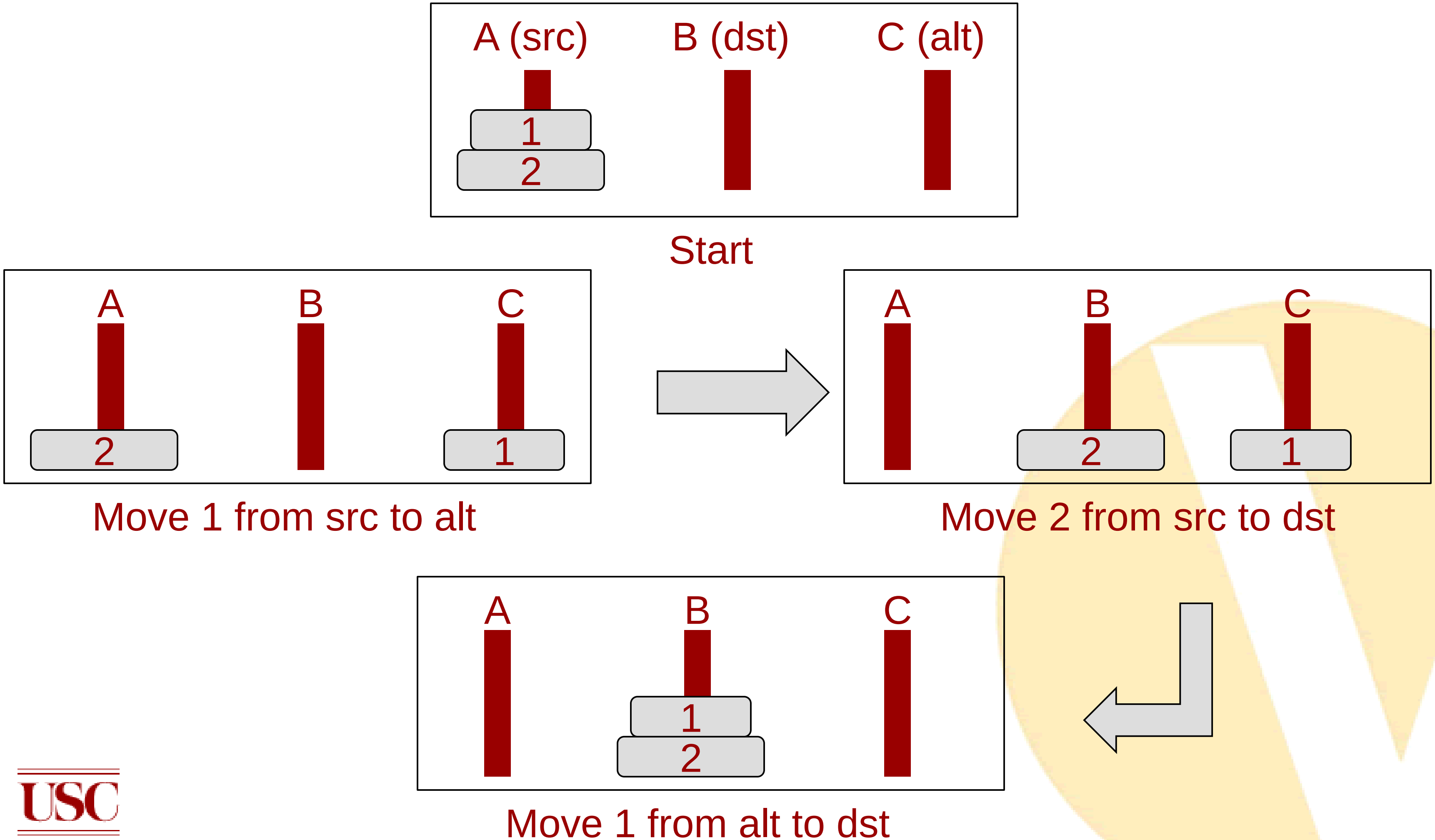
Towers of Hanoi Problem

- Problem Statements: Move n discs from source pole to destination pole (with help of a 3rd alternate pole)
  - Cannot place a larger disc on top of a smaller disc
  - Can only move one disc at a time



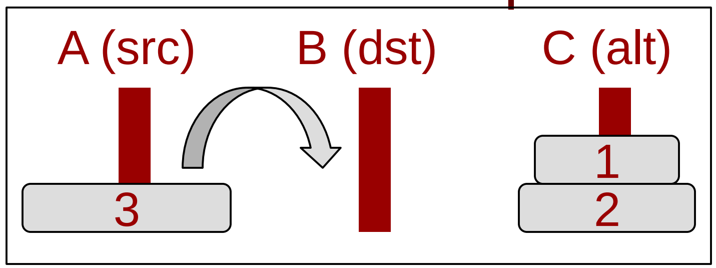
Observation 1

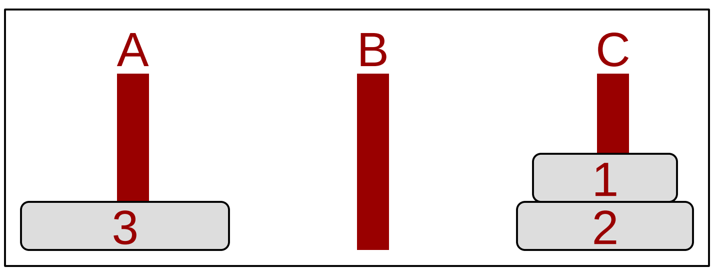
- Observation 1: Disc 1 (smallest) can always be moved
- Solve the n=2 case:



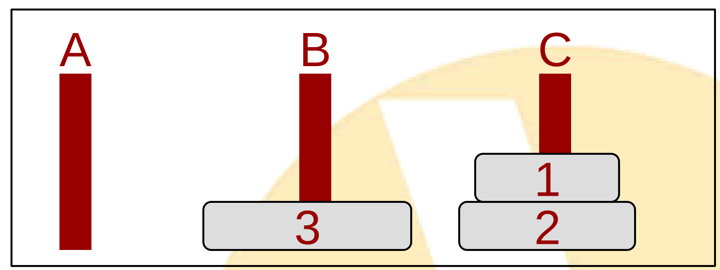
Observation 2

- Observation 2: If there is only one disc on the src pole and the dest pole can receive it the problem is trivial

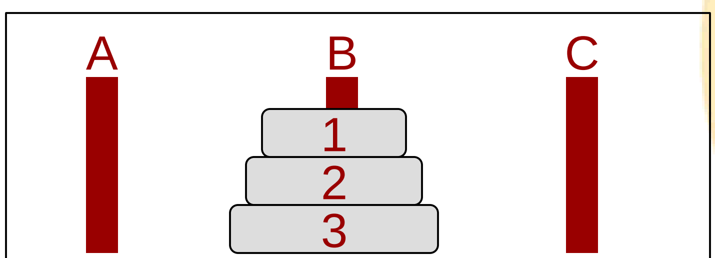




Move n-1 discs from src to alt



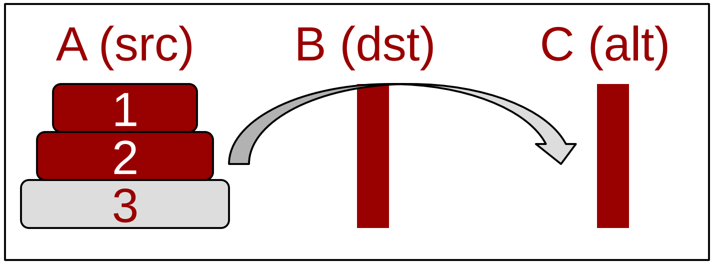
Move disc n from src to dst



Move n-1 discs from alt to dst

Recursive solution

- But to move n-1 discs from src to alt is really a smaller version of the same problem with
  - n => n-1
  - src=>src
  - alt =>dst
  - dst=>alt



- Towers(n,src,dst,alt)
  - Base Case: n==1   // Observation 1: Disc 1 always movable
    - Move disc 1 from src to dst
  - Recursive Case: // Observation 2: Move of n-1 discs to alt & back
    - Towers(n-1,src,alt,dst)
    - Move disc n from src to dst
    - Towers(n-1,alt,dst,src)

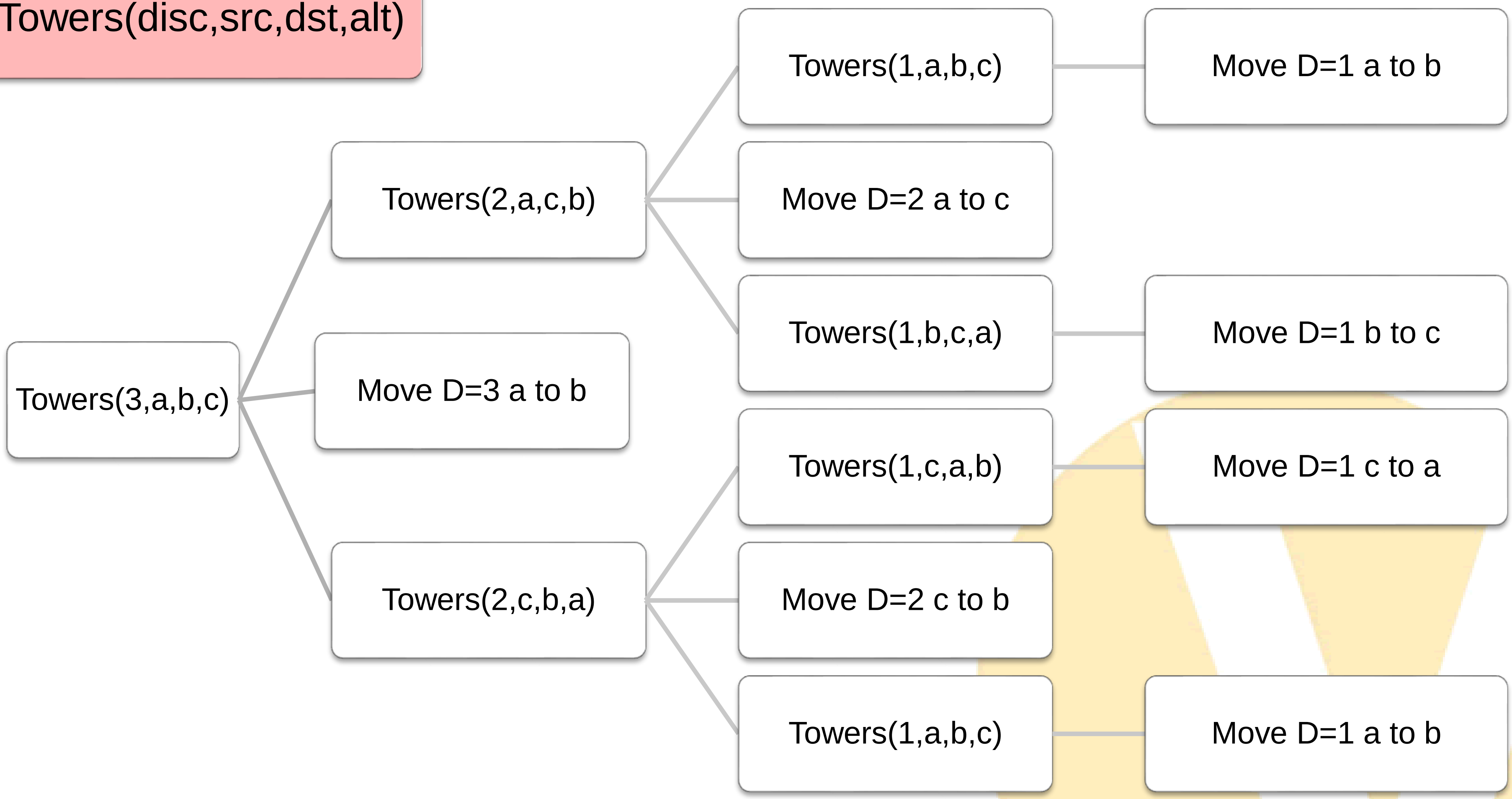
Exercise

- Implement the Towers of Hanoi code
  - $ wget http://ee.usc.edu/~redekopp/cs104/hanoi.cpp
  - Just print out "move disc=x from y to z" rather than trying to "move" data values
    - Move disc 1 from a to b
    - Move disc 2 from a to c
    - Move disc 1 from b to c
    - Move disc 3 from a to b
    - Move disc 1 from c to a
    - Move disc 2 from c to b
    - Move disc 1 from a to b

Recursive Box Diagram

Towers Function Prototype

Towers(disc,src,dst,alt)



Recursive Definitions

- N = Non-Negative Integers and is defined as:
  - The number 0
  - n + 1 where n is some non-negative integer
- Palindrome (string that reads the same forward as backwards)
  - Example: dad, peep, level
  - Defined as:
    - Empty string
    - Single character
    - xPx where x is a character and P is a Palindrome
- Recursive definitions are often used in defining grammars for languages and parsers (i.e. your compiler)

Simple Paragraph Grammar

Substitution	Rule
subject	"I" | "You" | "We"
verb	"run" | "walk" | "exercise" | "eat" | "play" | "sleep"
sentence	subject verb '.'
sentence_list	sentence | **sentence_list** sentence
paragraph	[TAB = \t] sentence_list [Newline = \n]

Example:

I run. You walk. We exercise.
subject verb. subject verb. subject verb.

sentence sentence sentence
sentence_list sentence sentence
sentence_list sentence
sentence_list
paragraph

Example:

I eat You sleep
Subject verb subject verb
Error

C++ Grammar

Rule	Expansion
expr	constant | variable_id | function_call | assign_statement | ‘(‘ expr ‘)’ | expr binary_op expr | unary_op expr
expr_statement	‘;’ | expr ‘;’
assign_statement	variable_id ‘=‘ expr

Example:

5 * (9 + max);
expr * (*expr* + *expr*);
expr * (*expr*);
expr * *expr*;
expr;
expr_statement

Example:

x + 9 = 5;
expr + *expr* = *expr*;
expr = *expr*;

NO SUBSTITUTION
Compile Error!

C++ Grammar

Rule	Substitution
statement	expr_statement | compound_statement | if (expr) statement | while (expr) statement …
compound_statement	'{' statement_list '}'
statement_list	statement | statement_list statement

Example:

```
while(x > 0) { doit(); x = x-2; }
while(expr) { expr; assign_statement; }
while(expr) { expr; expr; }
while(expr) { expr_statement expr_statement }
while(expr) { statement statement }
while(expr) { statement_list statement }
while(expr) { statement_list }
while(expr) compound_statement
while(expr) statement
statement
```

Example:

```
while(x > 0)
   x--;
   x = x + 5;
```

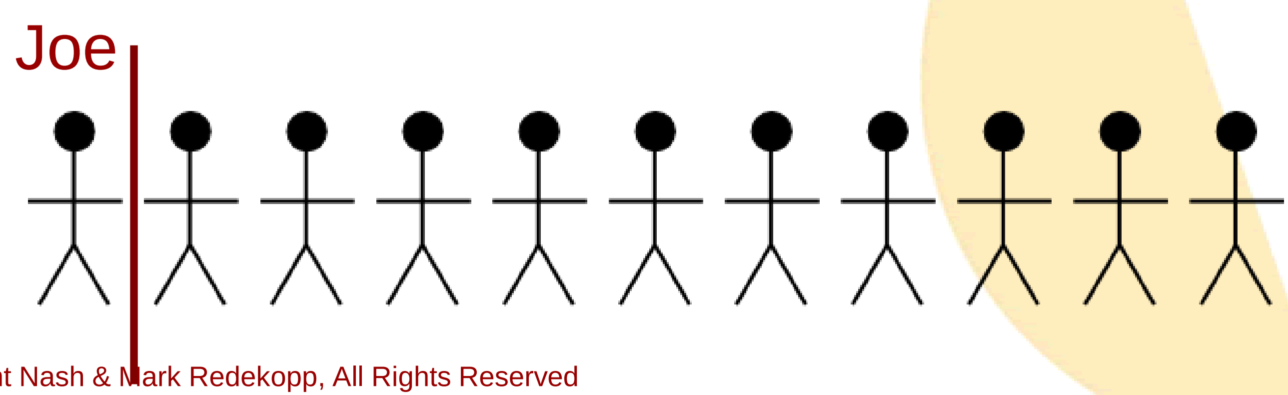
```
while(expr)
   statement
   statement
```

```
statement
statement
```

BACKUP

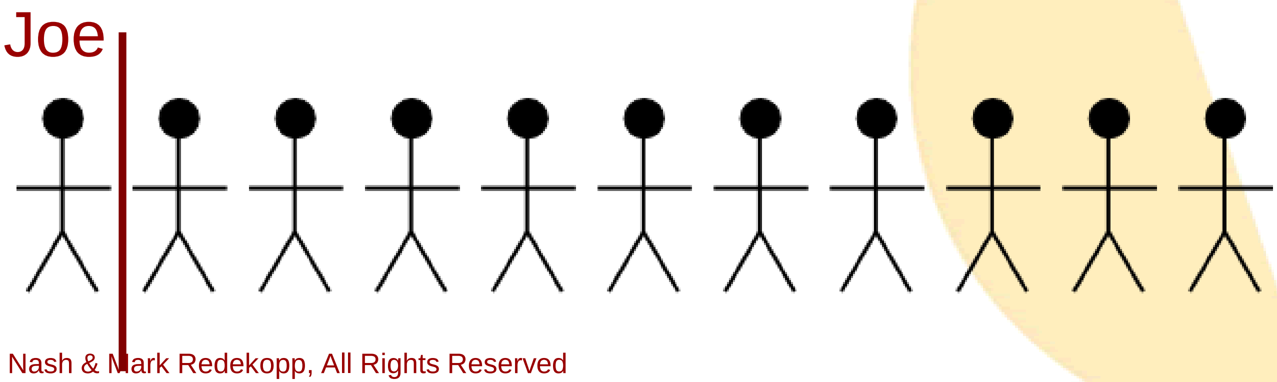
Combinatorics Examples

- Given n things, how can you choose k of them?
  - Written as C(n,k)
- How do we solve the problem?
  - Pick one person and single them out
    - Groups that contain Joe => ________________
    - Groups that don't contain Joe => ________________
  - Total number of solutions: ______________________
  - What are base cases?



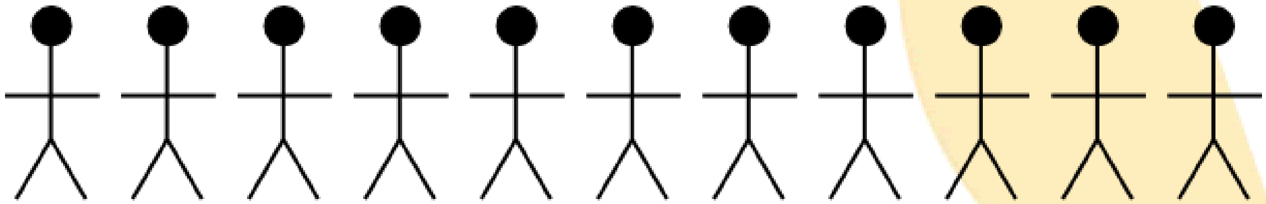
Combinatorics Examples

- Given n things, how can you choose k of them?
  - Written as C(n,k)
- How do we solve the problem?
  - Pick one person and single them out
    - Groups that contain Joe => C(n-1, k-1)
    - Groups that don't contain Joe => C(n-1, k)
  - Total number of solutions: C(n-1,k-1) + C(n-1,k)
  - What are base cases?



Combinatorics Examples

- You're going to Disneyland and you're trying to pick 4 people from your dorm to go with you
- Given n things, how can you choose k of them?
  - Written as C(n,k)
  - Analytical solution: C(n,k) = n! / [k! * (n-k)!]
- How do we solve the problem?



Recursive Solution

- Sometimes recursion can yield an incredibly simple solution to a very complex problem
- Need some base cases
  - C(n,0) = 1
  - C(n,n) = 1

```
int C(int n, int k)
{
if(k == 0 || k == n)
  return 1;
else
  return C(n-1,k-1) + C(n-1,k);
}
```