Recursion

Introduction

Each time a recursive function calls itself, it must pass in a simpler sub-problem than the one it was asked to solve

Eventually reaches the simplest subproblem, which it solves without needing to call itself

Idea

```
SolveAProblem(problem)
    /
  Is the problem trivially solved ->Yes-> return the answer
    \
    No
     \
     Break the problem into two or more simpler sub-problems
       \
      Solve each sub-problem j by calling some other function
      on the sub-problem j
        \
    Collect all the solution(s) to the sub-probems
        \
    Use the sub-solutions to construct
    a solution to the complete problem -> Return the solution
```

Merge Sort

Idea: Divide the pile in half and in half and in half... and sort it, when only one, return because it is already sorted

```
void MergeSort(an array)
{
    if (array's size == 1)
    {
        return;
    }
    MergeSort(first half of array); //process the 1st half
    MergeSort(second half of array); //process the 2nd half

    Merge(the two array halves);
    //now the complete array is sorted
}
```

```
void sort(int a[], int start, int end)
{
    if (end - start >= 2) //if array has at least 2 elements
    {
        int mid = (start + end) / 2; //midpoint
        sort(a, start, mid);
        sort(a, mid, end);
        merge(a, start, mid, end); //assuming we have the merge function
    }

    //if array only has 1 element -> already sorted
}
```

- `end` is the position just past the array

Two Rules of Recursion

1. Base Case: Every recursive function must have a "stopping condition"

2. Every resursive function must have a "simplifying step"

    - Every time a recursive function calls itself, it must pass in a smaller sub-problem that ensures the algorithm will eventually reach its stopping condition

- Recursive functions should never use global, static, or member variables

Write Recursive Functions

1. Write the function header
2. Define your magic function
3. Add your base case code
4. Solve the problem with the magic function
5. Remove the magic
6. Validate your function

Example 1: Factorial

#1 Write Function Header

- Factorial function takes in an integer as a parameter, e.g. factorial(6)

- Factorial computes and should return an integer result -> return type `int`

```
int main()
{
    int n = 6;
    int result;
```

```
    result = fact(n);
}
```

#2 Define Your Magic Function

- Pretend that you are given a magic function that can compute a factorial, it's already been written for you

```
int magicfact(int x)
{
    int f = 1; (an example)
    while (x > 1)
    {
        f *= x;
        x--;
    }
    return f;
}
```

```
int fact(int n)
{

}
```

- Takes the same parameters as your factorial function and returns the same type of result/value
- But CANNOT pass in a value of `n` (other words cannot use it to compute `n`)
    - CAN compute smaller problems like `(n-1)!`

#3 Add base case code

- Determine base case and write code to handle without recursion
- Identify the simplest possible input(s) to our function
- Have our function process those inputs without calling itself
- Pass `0` to our function: `0! == 1`

```
int fact(int n)
{
    if (n == 0)
        return 1;
    ...
}
```

> Always consider all possible base cases

#4 Solve problem using magic function

- Break problem into two or more simpler sub-problems and use magic function to solve those
- `N! = N(N-1)!` -> already split into two parts
  - Compute `n` is trivial: `int part1 = n;`
  - Use magic function to solve `n-1`: `int part2 = magicfact(n - 1);`

```
int fact(int n)
{
    if (n == 0)
        return 1;

    int part1 = n;
    int part2 = magicfact(n-1);

    return part1 * part2;
}
```

#5 Remove the magic

```
int fact(int n)
{
    if (n == 0)
        return 1;

    int part1 = n;
    int part2 = fact(n-1);

    return part1 * part2;
}
```

#6 Validating our function

- Test function with simplest possible input:

  ```
  fact(0);
  ```

- Test function incrementally

Example 2: Recursion on an array

- Recursion to get the sum of all the items in an array

#1 Write the function header

- Figure out arguments and return type
- To sum all the items in array: need array and size
- Return `int` total sum

```
int main()
{
    const int n = 5;
    int arr[n] = { 10, 100, 42, 72, 16 }, s;
    s = sumArr(arr, n);
}
```

#2 Define magic function

- Sum up smaller arrays (e.g. n-1 elements)

```
int magicsumArr(int arr[], int x)
```

```
//first n - 1
s = magicsumArr(arr, n-1);
//last n - 1
s = magicsumArr(arr+1, n-1);
//sums 1st half
s = magicsumArr(arr, n/2);
//2nd half
s = magicsumArr(arr+n/2, n-n/2);
```

#3 Add your base case code

1. Empty array of size `n=0` -> sum = 0
2. Array of size `n=1`

```
int sumArr(int arr[], int n)
{
    if (n == 0) return 0;
    if (n == 1) return arr[0];
}
```

#4 Solve problem using magic function

- Strategy 1: Front to back

  - Process first `n-1` elements of array, ignoring the last element
  - Get result from magic function, combine with the last element

  ```
  int front = magicsumArr(arr, n-1);
  int total = front + arr[n - 1];
  return total;
  ```

- Strategy 2: Back to front
  - Process last n-1 elements, ignore first element

  ```
  int rear = magicsumArr(arr+1, n-1);
  int total = arr[0] + rear;
  ```

- Strategy 3: Divide and Conquer
  - Process first half of the array
  - Also process last half of the array
  - Combine both and return

  ```
  int first = magicsumArr(arr, n/2);
  int scnd = magicsumArr(arr+n/2, n-n/2);
  return first + scnd;
  ```

#5 Remove the Magic

```
int sumArr(int arr[], int n)
{
    if (n == 0) return 0;
    if (n == 1) return arr[0];
    int first = sumArr(arr, n/2);
    int scnd = sumArr(arr + n/2, n - n/2);
    return first + scnd;
}
```

Example 3: Print Array in Reverse

Write a recursive function `printArr` that prints out an array of integers in reverse from bottom to top

Write function header:

```
void printArr(int arr[], int size)
```

Magic Function:

```
const int size = 5;
int arr[size] = {7, 9, 6, 2, 4};
magicprintArr(arr+1, size - 1); //print last n-1 elems
```

Add Base Code:

```
if (size == 0)
    return;
```

Solve problem using magic function:

```
magicprintArr(arr+1, size-1);
cout << arr[0];
```

Remove magic:

```
void printArr(int arr[], int size)
{
    if (size == 0)
        return;
    printArr(arr+1, size-1);
    cout << arr[0];
}
```

Example 4: Recursion on a Linked List

- Much like processing an arry with strategy #2, but
    - Instead of passing in a pointer to an array element, pass in a pointer to a node
    - Don't need to pass in a size value for list
- Write a function that finds the biggest number in a non-empty linked list

Write function header:

- Needs to take a `Node` parameter
- Returns biggest value in list -> type `int`

```
struct Node
{
```

```
    int val;
    Node* next;
};

int biggest(Node* cur);
```

Define magic function:

- Can use it to find the biggest item in a *partial* list

```
int magicbiggest(Node* n);
```

```
int main()
{
    Node* cur = createLinkedList();
    int biggest = magicbiggest(cur->next);
}
```

- `cur->next` points to a linked list with `n-1` elements

Add base case:

- (Assume linked list at least one element)
- Linked list with *only one node*
  - Then by def that node must hold the biggest value in the list -> just return `cur->value`

```
if (cur->next == nullptr) //the only node
    return cur->val;
```

Solve problem using magic function:

- Use magic function to process the last `n-1` elements of the list, ignore the first element
- Combine it with the first element

```
int rest = magicbiggest(cur->next);
//pick biggest of 1st node and last n-1 nodes
return max(rest, cur->val);
```

Remove the magic:

```
int biggest(Node* cur)
{
    if (cur->next == nullptr)
        return cur->val;

    int rest = biggest(cur->next);
    return max(rest, cur->val);
}
```

Validate function

```
int main()
{
    Node* head1 = mkLstWith1Item();
    cout << biggest(head1);

    Node* head2 = mkLstWith2Items();
    cout << biggest(head2);
}
```

> Critical Tip: Your recursive function should generally **only access the current node/array cell** passed into it

Never/rarely access the values in the node/cells below

```
int recursiveGood(Node *p)
{
    if (p->value == someValue)
        do something;
    if (p == nullptr || p->next == nullptr)
        do something;
    int v = p->value + recursiveGood(p->next);

    if (p->value > recursiveGood(p->next))
        do something;
}
```

```
int recursiveBad(Node *p)
{
    if (p->next->value == someValue)

    if (p->next->next == nullptr)

    int v = p->value + p->next->value +
    recursiveBad(p->next->next);
```

```
    if (p->value > p->next->value)
}
```

For arrays

```
int recursiveGood(int a[], int count)
{
    if (count == 0 || count == 1)
    if (a[0] == someValue)
    int v = a[0] + recursiveGood(a+1, count-1);
    if (a[0] > recursiveGood(a+1, count-1))
}
```

```
int recursiveBad(int a[], int count)
{
    if (count == 2)
    if (a[1] == someValue)
    int v = a[0] + a[1] + recursiveBad(a+2, count-2);
    if (a[0] > a[1])
        recursiveBad(a+2, count-2);
}
```

Example 5: Count number of times a number in array

Write a recursive function called `count` that counts the number of times a number appears in an array

Write function header:

```
int count(int arr[], int size, int val)
```

Define magic function:

```
void magiccount(int a[], int s, int v)
```

```
int main()
{
    const int size = 5;
    int arr[size] = {7, 9, 6, 7, 7};
    int val = 7;
    int b = magiccount(arr + 1, size - 1, val);
}
```

Add base case:

```
if (size == 0)
    return 0;
```

Solve problem with magic function:

```
int total;
total = magiccount(arr+1, size-1, val);
if (arr[0] == val)
    ++total;
return total;
```

Remove magic:

```
int count(int arr[], int size, int val)
{
    if (size == 0)
        return 0;
    int total;
total = count(arr+1, size-1, val);
if (arr[0] == val)
    ++total;
return total;
}
```

Example 6: Find earliest position of a number in linked list

Write a function `findPos` that finds and returns the earliest position of a number in a linked list. If the number is not in the list or the list is empty, function should return `-1` to indicate

Write function header:

```
int findPost(Node* cur, int val)
```

Magic function:

```
void magicfindPos(Node* n, int v)
```

```
int main()
{
```

```
    Node* cur = <make a linked list>;
    int val = 3;

    //search last n-1 nodes for value using magic func
    int a = magicfindPost(cur->next, val);
}
```

Add base case:

```
if (cur == nullptr) //number is not in list
    return -1;
if (cur->value == val) //number found in top node
    return 0;
```

Solve using magic function:

```
int posInRestOfList = magicfindPos(cur->next, val);
if (posInRestOfList == -1) //not in list
    return -1;
else
    return posInRestOfList + 1;
```

Remove magic:

```
int findPost(Node* cur, int val)
{
    if (cur == nullptr) //number is not in list
        return -1;
    if (cur->value == val) //number found in top node
        return 0;
    int posInRestOfList = findPos(cur->next, val);
    if (posInRestOfList == -1) //not in list
        return -1;
    else
        return posInRestOfList + 1;
}
```

Test:

```
int main()
{
    Node* head1 = nullptr;
    cout << findPos(head1, 5);

    Node* head2 = createSingleNode(5);
```

```
    cout << findPos(head2, 5);

    Node* head3= createTwoNodes(5, 6);
    cout << findPos (head3, 6);
}
```

Recursion: Binary Search

Goal: Search a *sorted* array of data for a particular item

Idea: Use recursion and a divide-and-conquer approach

Pseudocode:

```
bool Search(sortedArray, findMe)
{
    if (array.size == 0) //empty array
        return false;
    middle_word = sortedArray.size / 2;
    if (findMe == sortedArray[middle_word])
        return true;
    if (findMe < sortedArray[middle_word])
        return Search(first half of sortedArray);
    else
        return Search(second half of sortedArray);
}
```

- Notice how binary search code recurses on either the first half or the second half of the array, but never both

Code:

```
int BS(string A[], int top, int bot, string f)
{
    if (numItemsBetween(top, bot) == 0)
        return (-1); //value not found
    else
    {
        int Mid = (top + bot) / 2;
        if (f == A[Mid])
            return Mid; //found - return
        else if (f < A[Mid])
            return BS(A, top, Mid - 1, f);
        else if (f > A[Mid])
            return BS(A, Mid + 1, bot, f);
    }
}
```

Code Trace Through:

```
main()
{
    string names[11] = {"Albert", ... };
    if (BS(names, 0, 10, "David") != -1)
    {
        cout << "Found it!";
    }
}
```

Recursive helper functions

- Hide complexities and provide user with simple function

```
int SimpleBinarySearch(string A[], int size, string findMe)
{
    return BS(A, 0, size-1, findMe);
}
```

- This simple function call the complex recursive function

Backtracking (Depth First Search)

Solving a Maze

```
bool solvable;
int dcol, drow;
char m[11][11] = {
    "**********",
    "*        *",
    "* * * ** *",
    "*** *  * *",
    "* * ** * *",
    "*    *** *",
    "*  *   * *",
    "*  ***** *",
    "*     *  *",
    "**********"
};

main()
{
    solvable = false;
    drow = dcol = 10;

    solve(1,1);
```

```
    if (solvable == true)
        cout << "possible!";
}
```

```
void solve(int row, int col)
{
    m[row][col] = '#';
    if (row == drow && col == dcol)
        solvable = true;
    if (m[row - 1][col] == ' ')
        solve(row-1, col);
    if (m[row + 1][col] == ' ')
        solve(row+1, col);
    if (m[row][col - 1] == ' ')
        solve(row, col - 1);
    if (m[row + 1][col + 1] == ' ')
        solve(row, col + 1);
}
```

Approach to Solve the Maze

```
bool solve(start, end)
{
    if (start == end)
        return true;
    mark start as visited
    for each direction
    {
        if (moving in that direction is possible AND that spot has not been
visited)
            if (solve(position reached by moving that step, end))
                return true
    }
    return false
}
```

- Is every call solving a smaller problem?
    - Measure of size of problem: getting closer to the goal
- How to measure size of a problem?
    - Number of unvisited places will decrease each time a recursive call is made AND there is a finite number of spots
    - If everyspot has been visited and we are still not at goal -> maze not solvable
    - Size of problem is the number of unvisited places

- Base Cases
    - Either I reach the goal
    - Or I visited the place or everywhere surrounding is a wall

Writing a Tic Tac Toe Game

Use recursion to write function `getBestMoveForX()`

1. Try each X move

    a. if that ends the game, log the result (base case)

    b. otherwise, see how O would respond

2. Return the best move we found for X

Co-recursion

- The function for X calls the function for O (to simulate how O would move)
    - And this function for O calls the function for X, and the function for X calls the function for O...
    - Until we hit the bottom

--

Other Topics

Infinite Recursion

An infinite loop: going to consume time

- If there is nothing in the loop that consumes resources (dynamically allocate something) -> then actually infinite -> just consume time

Infinite Recursion: involves function call, which needs memory to set up new environment

- Every recursion consumes a little more memory
- Infinite recursion will after some time get the program to crash

Divide and Conquer (Approach to Recursion)

Break big problem into small problems

E.g. Merge sort

- The same algorithm will work if pile not split into evenly piles
    - Just not as efficient

The first and the rest OR The last and the rest

Some other problems want to divide unevenly

Ways to Go Wrong

Example: Missing Base Case

```
bool has(int a[], int n, int target)
{
    if (a[0] == target)
        return true;
    return has(a+1, n-1, target);
}
```

- How could this function ever return false?
- Can I fully prove termination?
    - If array does not have target, then no base case
    - We are eventually going to reach the end and try to access past the array

```
if (n == 0)
    return false;
```

- What if someone passes negative `n`

```
if (n <= 0)
    return false;
```