APS 105 Lecture 28 Notes

Last time: Printing different patterns recursively

Today: Doing recursion on strings (or arrays)

Recap:

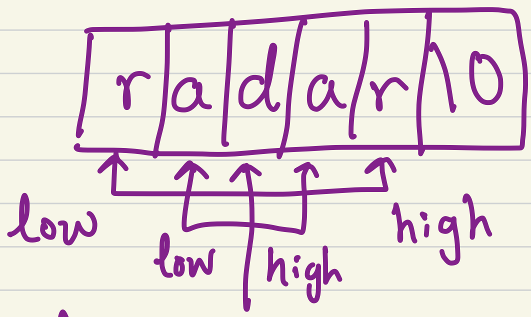
Recursion with strings

A String is an array of characters. To think of strings recursively, think of a string as a 1 character followed by a string OR 2 char preceded by a string OR 3 two characters enclosing a string.

The smaller problem is a smaller string, the base case is when you should stop solving on a smaller problem.

For example,

Write a "recursive" function to determine if a string is a palindrome.



Check edges then the string enclosed is a smaller problem.

```
bool isPalindromeRecursive(char *s, int low, int high){
        bool result;
        if(low == high)
                result = true;
        else if(s[low] != s[high])
                result = false;
        else
                result = isPalindromeRecursive(s,
                                                low+1,
                                                high-1);
        return result;
}
```

Sometimes it can be in convient to call a function with 3 arguements, so it is better to deal with isPalindromeRecursive as a helper function and call inside a function that takes only the string as an arguement

```
bool isPalindrome(char *s){
    return isPalindromeRecursive(s, 0, strlen(s)-1);
}
```

to make the value of high be the index of last character before '\0'.

Home work: How can we implement a function that reverses an array?

Another example (Final exam 2018, Q14)

Write a recursive function that takes a string and a character and returns the index of the 1st occurrence of the character.

Function Prototype is:

```
int recursiveFindIndex(char *str, char c);
```

Question 14 [10 Marks]

Consider the following function that returns the index of a char `c` in a string `string` (i.e., the position of the first c in the string), or returns -1 if c does not occur in `string`:

```
int findIndex(char *string, char c) {
    int n = 0;
    while (*string != c && *string != '\0') {
        string = string + 1;
        ++n;
    }
    if (*string == '\0')
        return -1;
    return n;
}
```

Write a C function `recursiveFindIndex(char *string, char c)` that does not use any loops and yet behaves like the `findIndex()` function above. Your function may have additional parameters, but at the minimum must include the parameters `string` and c.

str

a	p	p	l	e	\0

c 'l'

Recursive Call:

1. We check if 'a' is 'l' and then can think of the smaller problem as $\boxed{\text{pple\0}}$
2. <u>Base case</u> is either when we found the character, <u>OR</u> when we have reached end of string and didn't find it.

```
int recursiveFindIndexHelper(char *str, char c, int ind){
        if ( str[ind] == c)          ] Base Case :
                return ind;          ]  c is found
        else if (str[ind] == '\0')   ] Base Case:
                return -1;           ] c is not found
        else {
                ind++;
                return recursiveFindIndexHelper(str, c, ind);
        }
}
```

```
int recursiveFindIndex(char *str, char c){
    return recursiveFindIndexHelper(str, c, 0);
}
```

Start with ind=0

Count # of odd numbers in an array:

arr

3	7	5	8	10	1	9

Write a recursive function that counts the number of odd numbers in an array (i.e. if (num%2 == 1) num is odd)

<u>Recursive Call</u>: see if arr[i] is odd, then count number of odd numbers in the rest of the array

<u>Base Case</u>: we reached end of the array

3	2	1

Initial Call left=0 right=2 (arr[left]%2==1) +recursiveOddCount(arr,left+1,right)

Second call left=1 right=2 (arr[left]%2==1) +recursiveOddCount(arr,left+1,right)

Third call left=2 right=2 (arr[left]%2==1)

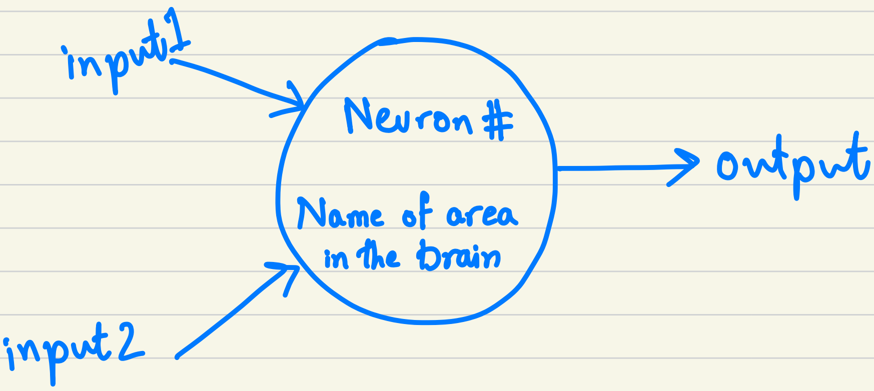
```
int recursiveOddCount (int *arr, int left , int right){
Base case { if (right == left)
          {     return (arr[left]% 2==1);
          else
Recursive {    return (arr[left]%2==1) +      recursiveOddCount(
Call      {           if odd -> true                    arr, left +1,
          {           if even -> false                      right);
}
```

Introduce more complex data structures

So far we dealt with int, char, bool, double and a collection of one data type in an array (1D or 2D).

But often we want to define more complex types with a combination of data types.

Example: we want to model a neuron in the brain



Why not have all these features kept together in one data structure, and we can name it neuron.

```
struct NStruct{
        int neuronNum;
        double input1, input2, output;
        char areaName[50];
};
int main() {
  struct Nstruct neuron1, neuron2, neurons[100];
  neuron1.input1 = 7.3;
  return 0;
}
```

nothing here

not declaring variables

declare variables here!

access every member of neuron1 (member operator)

OR

```
struct NStruct{
        int neuronNum;
        double input1, input2, output;
        char areaName[50];
 } neuron1, neuron2, neurons[100];
```

declare variables here

```
int main(){
    neuron1.input1 = 7.3;
    strcpy(neurons[3].areaName, "Cortex");
    return 0;
}
```

OR

To create an alias for the data type, we use typedef

For example,

cannot be used without struct before it

```
struct NStruct{
        int neuronNum;
        double input1, input2, output;
        char areaName[50];
};
```

nothing here

not declaring variables

```
typedef struct NStruct Neuron;
int main(){
    Neuron neuron1;
    neuron1.input1 = 3.2;
    return 0;
}
```

OR

Without typedef

```
struct Distance {
    int feet;
    double inches;
};
int main(){
    struct Distance d1,d2;
    return 0;
}
```

With typedef

```
typedef struct Distance {
    int feet;
    double inches;
}distance;
int main()
    distance d1,d2;
    return 0;
}
```

In general typedef works as follows:

typedef < data type name / definition if it was struct > < alias Name>;

Hence,

```
typedef struct Distance {
    int feet;
    double inches;
}distance;
int main()
    distance d1,d2;
    return 0;
}
```

equivalent

```
struct Distance {
    int feet;
    double inches;
};
typedef struct Distance distance;
```