CS 201
RECITATION 4

- RECURSION
- TIME COMPLEXITY

Palindrome question

- A palindrome is a sequence of characters or numbers that looks the same forwards and backwards. For example, "**dennis sinned**" is a palindrome because it is spelled the same reading it from front to back as from back to front. The number **12321** is a numerical palindrome. Write a function that takes a string and its length as arguments and recursively determines whether the string is a palindrome:

- int isPalindrome(char *s, int len);

Palindrome question

Thinking Recursively

- **Step 1: Consider various ways for simplifying inputs.**

How can you simplify the inputs in such a way that the same problem can be applied to simpler input.

Here are several possibilities for the palindrome test problem.

- Remove the first character.
- Remove the last character.
- Remove both the first and last character.
- Remove a character from the middle.
- Cut the string into two halves.

Palindrome question

Thinking Recursively

- **Step 2: Combine solutions with simpler inputs to a solution of the original problem.**

  Don't worry how those solutions are obtained. These are simpler inputs, so someone else will solve the problem for you.

  Removing the first and last characters seems promising:

  "rotor"

  becomes

  "oto"

  A word is a palindrome if

  the first and last letters match

  the word obtained by removing the first and last letters is a palindrome.

Palindrome question

Thinking Recursively

- **Step 3: Find solutions to the simplest inputs.**

  To make sure that the recursion comes to a stop, you must deal with the simplest inputs separately.

  Sometimes you get into philosophical questions dealing with degenerate inputs: empty strings, shapes with no area, and so on.

  The simplest strings for the palindrome test are:

  strings with two characters

  strings with a single character

  the empty string.

  We don't need a special case for strings with two characters - by removing both the first and last characters, it becomes a string of length 0.

  - A single character string is a palindrome.
  - An empty string is a palindrome.

Palindrome question

```
bool isPalindrome( char *s, int len) {
    if (len <= 1) {
                return true;
    }
    else
                return ((s[0] == s[len-1]) && isPalindrome(s+1, len-2));
    }
}
```

What is the time complexity of this function in terms of Big-O notation?

Palindrome question

```
bool isPalindrome( char *s, int len) {
    if (len <= 1) {
                    return true;
    }
    else
                    return ((s[0] == s[len-1]) && isPalindrome(s+1, len-2));
    }
}
```

What is the time complexity of this function in terms of Big-O notation?

T(0) = 1 // base case
T(1) = 1 // base case

T(n) = 1 + T(n-2)// general case →*T(n-2) = 1 + T(n-4)*
T(n) = 2 + T(n-4)
T(n) = 3 + T(n-6)
T(n) = k + T(n-2k) ... n-2k = 1 → k= (n-1)/2

T(n) = (n-1)/2 + T(1) → O(n)

Old Midterm Question

Consider the problem of finding the k element subsets of a set with n elements. Write a recursive function that takes an array of integers representing the set, the number of integers in the set (n), and the required subset size (k) as input, and displays all subsets with k elements on the screen. You may assume that the elements in the array have unique values.

For example, if the array (set) contains the elements [8 2 6 7], n is 4, and k is 2, then the output is

82
86
87
26
27
67

Old Midterm Question

```
void findSubsets( int *array, int n, int k, int *taken, int i) {
    if (k == 0) {
                for (int i = 0; i < n; i++)
                        if (taken[i])
                                cout << array[i];
                cout << endl;
    }
    else if (i < n) {
                taken[i] = 1;
                findSubsets(array,n,k-1,taken,i+1);
                taken[i] = 0;

                findSubsets(array,n,k,taken,i+1);
    }
}
```

What is the time complexity of this function in terms of Big-O notation?

Old Midterm Question

```
void findSubsets( int *array, int n, int k, int *taken, int i) {
     if (k == 0) {
                          for (int i = 0; i < n; i++)
                                       if (taken[i])
                                                    cout << array[i];
                          cout << endl;
     }
     else if (i < n) {
                          taken[i] = 1;
                          findSubsets(array,n,k-1,taken,i+1);
                          taken[i] = 0;

                          findSubsets(array,n,k,taken,i+1);
     }
}
```

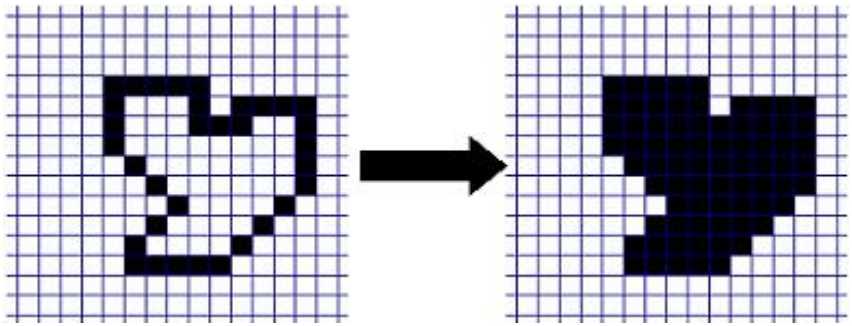
What is the time complexity of this function in terms of Big-O notation?

Each combination that is generated is printed, and it takes $O(n)$ recursive invocations for each combination printed, so an upper bound on the number of recursive calls is $O(n*k)$, which is $O(n^2)$ in the worst-case, (when $k = n/2$).

The Flood Fill Algorithm

- We are working on a 2D drawing package, and we want to make a bucket fill tool similar to that found in Microsoft Paint. Assume that the picture can only be black and white for simplicity. The user draws a picture, clicks the flood fill button, and then clicks on a point, and the black color spreads to fill the area bounded by the picture. Have a look at the figure.

- We want to write a function to implement this feature. We are given a 2 dimensional array of integers of size 100 x 100 (board) where a 0 in the 2D array indicates a white pixel and a 1 indicates a black pixel, and a point to start painting from (int x, int y). We should fill the fill area with the color as mentioned above.



The Flood Fill Algorithm

- There is no easy non-recursive solution to this problem, but we can solve this recursively in just 8 lines of code. How? The idea is simply to make a flood fill from a point do the following:
  - If the point we want to color is already filled, don't do anything and stop, else color it.
  - If the point above it is not colored, call flood fill from that point (recursive step).
  - If the point to the right is not colored, call flood fill from that point (recursive step).
  - If the point to the left is not colored, call flood fill from that point (recursive step).
  - If the point below it is not colored, call flood fill from that point (recursive step).

The Flood Fill Algorithm

- Now, you see, the whole solution is just 5 steps, and 4 of them are recursive calls. It is important to understand how this actually works.

```
void Fill(int board[100][100], int x, int y)
{
   if(board[x][y] == 1)
        return;
         board[x][y] = 1;
         if(x < 99 && board[x+1][y] != 1)
                  Fill(board, x+1, y);
         if(x > 0 && board[x-1][y] != 1)
                  Fill(board, x-1, y);
         if(y < 99 && board[x][y+1] != 1)
                  Fill(board, x, y+1);
         if(y>0 && board[x][y-1] != 1)
                  Fill(board, x, y-1);
}
```

What is the time complexity of this function in terms of Big-O notation?

The Flood Fill Algorithm

- Now, you see, the whole solution is just 5 steps, and 4 of them are recursive calls. It is important to understand how this actually works.

```
void Fill(int board[100][100], int x, int y)
{
   if(board[x][y] == 1)
        return;
         board[x][y] = 1;
         if(x < 99 && board[x+1][y] != 1)
                  Fill(board, x+1, y);
         if(x > 0 && board[x-1][y] != 1)
                  Fill(board, x-1, y);
         if(y < 99 && board[x][y+1] != 1)
                  Fill(board, x, y+1);
         if(y>0 && board[x][y-1] != 1)
                  Fill(board, x, y-1);
}
```

What is the time complexity of this function in terms of Big-O notation?

Since there are if-clauses whole operation only considers empty cells, which have a static number say n. Therefore worst-case complexity becomes O(n).

Complexity Analysis - 1

```
for (i = 0; i <= n; i++){
    j = n;
    while (j >= i)
          j--;
}

Answer: O(n^2)
```

```
for (i = 0; i <= n; i++){
    j = 0;
    while (j < 10000)
          j++;
}

Answer: O(n)
```

Complexity Analysis - 2

```
void display(int n){
    for (int i = 0; i <= n; i++)
                for (int j = 0; j < 2*n; j++)
                        cout << i * j <<
endl;
}
void myfunction(int m){
    for (int i = 0; i < m; i++)
                display(m);
}
Answer: O(n^3)
```

```
void display(int n){
    for (int i = 0; i <= n; i++)
                for (int j = 0; j < 2*n; j++)
                        cout << i * j <<
endl;
}
void myfunction(int m){
    for (int i = 0; i < m; i++)
                display(i);
}
Answer: O(n^3)
```

Complexity Analysis - 3

```
int *arr = new int[n];
int k = 0;
for (int *i = arr; i < arr + n; i++)
    k++;
Answer: O(n)
```

```
int f1(int n){
    if (n < 100)
                    return 1;
    return n * f1(n-3);
}
```

T(99) = 1 // base case

T(n) = 1 + T(n-3)// general case →*T(n-3) = 1 + T(n-6)*
T(n) = 2 + T(n-6)
T(n) = 3 + T(n-9)
T(n) = k + T(n-3k) ... n-3k = 99 → k= (n-99)/3

T(n) = (n-99)/3 + T(99) → O(n)

```
Answer: O(n)
```

Complexity Analysis - 4

```
int f2(int n){
    if (n <= -8)
                    return 1;
    return n * f2(n/3);
}
Answer: O(log n)
```

T(-8) = 1 // base case

T(n) = 1 + T(n/3)// general case →*T(n/3) = 1 + T(n/9)*
T(n) = 2 + T(n/9)
T(n) = 3 + T(n/27)
T(n) = k + T($n/3^k$) ... $n/3^k = -8$ → $n= -8 * 3^k$ → $\log_3 n = k$

T(n) = $\log_3 n$ + T(1) → O(log n)

```
int f3(int n){
    if (n < 100)
                    return 1;
    return n * f3(n-1) * f3(n-2) * f3(n-3);
}
Answer: O(3^n)
```

T(99) = 1 // base case

T(n) = 1 + T(n-1) + T(n-2) + T(n-3) // general case
→*T(n-1)* = 1 + T(n-2) + T(n-3) + T(n-4)
→*T(n-2)* = 1 + T(n-3) + T(n-4) + T(n-5)
→*T(n-3)* = 1 + T(n-4) + T(n-5) + T(n-6)
T(n) = 4+ T(n-2) + T(n-3) + **T(n-4)** + T(n-3) + **T(n-4)** + T(n-5)
+ **T(n-4)** + T(n-5) + T(n-6)

In each step you call T three times, thus will provide eventual asymptotic barrier of:

T(n) = 3 * 3 * ... * 3 = 3^n → O(3^n)