In order to understand recursion, one must first understand recursion.

Let’s see what Google tells us about recursion:

That is, in order to find recursion, one must first find recursion.
Recursion

A way of solving problems:

- If the problem is trivial, return the result.
- If not trivial, break the problem into small sub-problems.
- Solve sub-problems recursively.
- Use solutions of sub-problems to solve the problem.

```java
Solve(Problem) {
    if (Problem is trivial) return result
    sol1 = Solve(Smaller Problem);
    sol2 = Solve(Smaller Problem);
    return combine(sol1, sol2);
}
```
Recursion Rules

Two rules of recursion:

- Base step (stopping condition).
- Simplifying step.

You must have those two steps. Otherwise, your recursive functions will run forever.
When dealing with recursion:

- Don’t delve into tracing down every level of the recursion.
- Instead, make the **recursive leap of faith**: Any recursive call to a smaller problem than the one you were given works.
Write a recursive function to calculate factorials, that is:

\[
Fact(n) = \begin{cases} 
1, & \text{if } n = 0 \\
 n \times Fact(n - 1), & \text{if } n > 0 
\end{cases}
\]

```c
int factorial(int n) {
    // base case
    if(n==0) return 1;

    // simplifying step
    int part1 = n;
    int part2 = factorial(n-1);

    // combine sub-problems
    return part1*part2;
}
```
Write a recursive function to compute $b^e$, for $b > 0$ and $e \geq 0$:

$$Power(b, e) = \begin{cases} 
1, & \text{if } e = 0 \\
 b \times Power(b, e - 1), & \text{if } e > 0
\end{cases}$$

```c
int power(int b, int e) {
    // base case
    if (e == 0) return 1;

    // simplifying step and combining sub-problems
    return b * power(b, e - 1);
}
```

How many times does this function recurse? $e$ times.
Can we optimize $\text{power}(b, e)$?

$$
\text{Pow}(b, e) = \begin{cases} 
1, & \text{if } e = 0 \\
\text{Pow}(b, e/2) \times \text{Pow}(b, e/2), & \text{if } e \text{ is even} \\
b \times \text{Pow}(b, \lfloor e/2 \rfloor) \times \text{Pow}(b, \lfloor e/2 \rfloor), & \text{if } e \text{ is odd}
\end{cases}
$$

```c
int power(int b, int e) {
    // base case
    if (e == 0) return 1;

    // simplifying step and combining sub-problems
    int half = power(b, e/2);
    if ((e % 2) == 0) return half*half;
    else return b*half*half;
}
```

How many times does this function recurse? $\log_2(e)$ times.
Do we always have one stopping condition? No.
Do we always have one recursive call? No.

Write a recursive function to calculate Fibonacci sequence:

\[
Fib(n) = \begin{cases} 
0, & \text{if } n = 0 \\
1, & \text{if } n = 1 \\
Fib(n - 1) + Fib(n - 2), & \text{if } n > 1 
\end{cases}
\]

```c
int fib(int n) {
    // base case
    if (n == 0) return 0;
    if (n == 1) return 1;
    // simplifying step and combining sub-problems
    return fib(n - 1) + fib(n - 2);
}
```
How to recurse on arrays?

- By passing indices as arguments.

Write a recursive function to find the max value in an array:

```c
int max(int[] arr, int start, int end) {
    //base case
    if(start==end) return arr[start];

    //simplifying step and combining sub-problems
    int first = arr[start];
    int rest = max(arr,start+1,end);
    return ((first>rest)? first:rest);
}
```

For the sake of readability, define a simpler function:

```c
int maxValue(int[] arr, int size) {
    return max(arr,0,size-1);
}
```
Palindrome is a word that reads the same backward or forward, e.g., eye, Bob, sees, kayak, etc.

**tattarrattat**

Longest one in the Oxford English Dictionary. Coined by James Joyce in Ulysses for a knock on the door.
Write a recursive function to decide if a string is palindrome:

Some useful functions:
- `s.length()` returns the length of `s`.
- `s.substr(start,len)` returns a substring of `s` starting from `start` having length of `len`.

```cpp
bool isPalindrome(string s) {
    int n = s.length();
    // base case
    if(n==0) return true;
    if(n==1) return true;

    // simplifying step and combining sub-problems
    if(s[0]==s[n-1] &&
       isPalindrome(s.substr(1,n-2))) return true;
    else return false;
}
```
Write a recursive function to get all permutations of a string.

E.g., given \texttt{abc}, we want to get: (in any order)

\texttt{abc, acb, bac, bcb, cab, cbd}

How to break down the problem into smaller problems?

- What if we have all permutations of \texttt{bc}?
  \texttt{bc, cb}

- Isn’t it enough to insert \texttt{a} into each possible position?
  \texttt{abc, bac, bca, acb, cab, cba}

Some useful functions:

- \texttt{s.length()} returns the length of \texttt{s}.
- \texttt{s.substr(start, len)} returns a substring of \texttt{s} starting from \texttt{start} having length of \texttt{len}.
vector is defined in STL: to represent resizable arrays. We’ll store permutations using this class.

```cpp
vector<string> permute(string s) {
    int n = s.length();
    //base case
    if (n==1) return vector<string>(1, s);

    //simplifying step and combining sub-problems
    char first = s[0];
    string rest = s.substr(1, n-1);
    vector<string> rperm = permute(rest);
    vector<string> results;
    for (int i=0; i<rperm.size(); i++) {
        string p = rperm[i]; //p is of size (n-1)
        for (int j=0; j<n; j++) //n locations to insert first
            results.push_back(p.substr(0, j) + first + p.substr(j, n-1-j));
    }
    return results;
}
```
A *k-way permutation* of a string is a permutation of size $k$ obtained from the string.

Write a recursive function to get all $k$-way permutations of a string.

E.g., given $abcd$ and $k = 2$, we want to get: (in any order)

$$ab, ac, ad, ba, bc, bd, ca, cb, cd, da, db, dc$$

How to break down the problem into smaller problems?

- **a** is in $k$-way permutations?
  - We need $(k - 1)$-way permutations of $bcd$.

- **a** is not in $k$-way permutations?
  - We need $k$-way permutations of $bcd$. 
Assume we can use permute(string s).

```cpp
vector<string> kWayPermute(string s, int k) {
    int n = s.length();
    //base case
    if(k==0) return vector<string>(1,"");
    if(n==k) return permute(s); //permute all letters

    //simplifying step and combining sub-problems
    char first = s[0];
    string rest = s.substr(1,n-1);
    vector<string> results = kWayPermute(rest,k);
    vector<string> rperm = kWayPermute(rest,k-1);
    for(int i=0; i<rperm.size(); i++) {
        string p = rperm[i]; //p is of size (k-1)
        for(int j=0; j<k; j++) //k locations to insert first
            results.push_back(p.substr(0,j)+first+p.substr(j,k-1-j));
    }
    return results;
}
```
We should solve it without using permute(string s):

```cpp
vector<string> kWayPermute(string s, int k) {
    int n = s.length();
    //base case
    if(k==0) return vector<string>(1,"");
    if(n==1) return vector<string>(1,s);

    //simplifying step and combining sub-problems
    char first = s[0];
    string rest = s.substr(1,n-1);
    vector<string> results;
    if(n>k) results = kWayPermute(rest,k);
    vector<string> rperm = kWayPermute(rest,k-1);
    for(int i=0; i<rperm.size(); i++) {
        string p = rperm[i]; //p is of size (k-1)
        for(int j=0; j<k; j++) //k locations to insert first
            results.push_back(p.substr(0,j)+first+p.substr(j,k-1-j));
    }
    return results;
}
```
Make sure base case is defined.
Make sure simplifying step is defined.
Don’t forget to test your function.
  - For each base case.
  - At least one recursive calls.
Slides will be available at

http://www.cs.ucla.edu/~umut/cs32