# Unit 26: Permutations

## Learning Objectives

After taking this unit, students should

- be familiar with using recursion to generate all possible permutations of a given array

## Generating Permutations

We have been using recursions to either compute or find a solution to a problem. In this unit, let's look at another useful application of recursion: to generate all possible permutations or combinations of items.

Let's see a simple example of this: generate all permutations of the characters in a string of length $n$. For simplicity, we assume that there is no repetition in the string. We leave the problem where there is repetition as an exercise in Problem 26.1.

## Recursive Formulation

To formulate a recursive solution, as usual, let's simplify the problem. A simpler version of the problem is to permute a string of length $n-1$. because we assume we know how to already do n-1 permutation, then means only left 1 more permutation (last one) which is easy to find and solve

The trivial case is when we generate the permutation of a string with one character. There is only one possible permutation! Now, by wishful thinking, we assume that we know how to generate all permutations of a string of length $n-1$. Here is what we do to generate all permutations of a string of length $n$.

For each character $i$ in the string, we move $i$ to the "first" position in the string. We have $n-1$ characters left, which we solve recursively, generating all possible permutations with $i$ as the first character.

For example, consider a string length 3, `abc`.

- We start with `a` as the "first" character, and generate all the permutations of the string `bc`. We get two permutations `abc` and `acb`. a (bc/cb)
- The next character is `b`. We generate all permutations of the string `ac`. We get `bac` and `bca`. b (ac/ca)
- Similarly, we get the permutations `cab` and `cba` by considering `c` as the "first" character and permutating `ba`. c (ab/ba)

Now, let's consider a longer example, say a string of length 5, `abcde`.

- Just like above, we start with `a` as the first character and permute `bcde`.
- As we recursively permute `bcde`, `a` should always remain in its position.
- To recursively permute `bcde`, we first fix `b` as the "first" character (but `b` is the second character in `abcde`) and permute `cde`.
- As we recursively permute `cde`, `ab` should remain in their positions.

We see that, as we recursively permute the shorter string, the prefix of the string should remain fixed.

## The Code

The idea above is implemented as the following. The function `permute` takes in the array `a` to permute, the length `len` of the array, and the location `curr`, where the substring `a[curr]` to `a[len - 1]` is what this function will permute. The function will print out all permutations of `a` where `a[0]` to `a[curr - 1]` are fixed, and `a[curr]` to `a[len - 1]` are permutated.

```
1  /**
2   * Fix a[0]..a[curr - 1] but permute characters a[curr]..a[len - 1]
3   * Print out each permutation.
4   *
5   * @param[in,out] a  The array to permute
6   * @param[in]     len The size of the array
7   * @param[in]     curr The starting index at which we will permute
8   *
9   * @post The string a remains unchanged
10  */
11 void permute(char a[], long len, long curr) {
12   if (curr == len - 1) {
13     cs1010_println_string(a);
14     return;
15   }
16
17   permute(a, len, curr + 1);
18   for (long i = curr + 1; i < len; i += 1) {
19     swap(a, curr, i);
20     permute(a, len, curr + 1);
21     swap(a, i, curr);
22   }
23 }
```

curr = 0

abc

base case cause nothing left to permute

this is to fix the starting letter of

swapping directly on the array?

will it affect the other perms?

this is to swap the affected characters with each other, without affecting starting character

Lines 3-6 above corresponds to the base case, where we have reached the end of the string, there is only one character left to permutate. Since there is only one possible permutation, we do not need to do anything except to print out the permutated string.

Line 8 permutes the remaining string, `a[curr + 1]` to `a[len - 1]`, with character `a[curr]` intact. Lines 9-13 is a for loop which loops through all characters `a[curr + 1]` to `a[len - 1]`, and swaps each one to the position of `a[curr]`, and recursively permutes the string `a[curr + 1]`..`a[len - 1]`. When we are done, we swap back the original `a[curr]`, this is to ensure that the string remains unchanged after `permute` is called.

## Running Time

How efficient is the function `permute`? We know that for a string of length $n$, there are $n!$ permutations, so the function `permute` should be at least $n!$. If it runs in $O(n!)$ steps, then our function above is as efficient as it can get.

Let the number of times `permute` is called when we pass in a string of length $n$ be $T(n)$. Each invocation of `permute(a, n, k)` calls `permute(a, n, k+1)` recursively $n-k$ times. The first call to `permute` is `permute(a, n, 0)`. We stop when we call `permute(a, n, n-1)`. So $T(1) = 1$.

So, we have:

$$T(n) = nT(n-1) = n(n-1)T(n-2) = n(n-1)(n-2)T(n-3) = .. = n(n-1)(n-2)..T(1) = n!$$

We made $n!$ calls to `permute`, so the function `permute` is as efficient as it gets.

## Problem Set

### Problem Set 26.1

In the code above, we assume that the string contains distinct characters. If there are duplicate characters in the string, duplicate permutations will be generated. For instance, if the input is `aaa`, the code above would print `aaa` six times.

We can fix this by making a small change to the function `permute` above so that it does not generate duplicate permutations. This can be done by adding a condition (Line A). Write a boolean function that we can call in Line A to check if we should continue to permute the rest of the string, and therefore avoid generating duplicate permutations when the input string contains duplicate characters.

```
void permute(char a[], long len, long curr) {
  // permute characters a[curr]..a[len-1] and print out a for each permutation.
  if (curr == len-1) {
    cs1010_println_string(a);
    return;
  }

  permute(a, len, curr + 1);
  for (long i = curr + 1; i < len; i += 1) {
    if (...) { // Line A
      swap(a, curr, i);
      permute(a, len, curr + 1);
      swap(a, i, curr);
    }
  }
}
```

## Appendix: Complete Code Written in Lecture

```
#include "cs1010.h"
#include <stdbool.h>
#include <stdio.h>
#include <string.h>

void swap(char a[], long i, long j) {
  char temp = a[i];
  a[i] = a[j];
  a[j] = temp;
}

void permute(char str[], long n, long k) {
  if (k == n-1) {
    cs1010_println_string(str);
    return;
  }
  permute(str, n, k+1);
  for (long i = k+1; i < n; i+= 1) {
    swap(str, k, i);
    permute(str, n, k+1);
    swap(str, i, k);
  }
}

int main() {
  char *str = cs1010_read_word();
  permute(str, strlen(str), 0);
  free(str);
}
```