More Recursion: Permutations

Computer Science Department
University of Central Florida

COP 3502 – Computer Science I

Permutations

- The Permutation problem:
- Given a list of items,
  - List ALL the possible orderings of those items
  - Often, we work with permutations of letters
- For example:
  - Here are all the permutations of the letters CAT:

CAT	ATC
CTA	TAC
ACT	TCA

  - The question: can we write a program to do this?

Permutations

- The Permutation algorithm:
  - There are several different permutation algorithms
  - Since recursion is an emphasis of the course,
    - we will present a recursive algorithm to solve this
  - Permutations of the letters CAT:

CAT	ATC
CTA	TAC
ACT	TCA

Permutations

- The Permutation algorithm:
  - The idea is as follows:
    - We want to list ALL the permutations of CAT
    - So we split our work into **3 groups of permutations**:

1) Permutations that start with C
2) Permutations that start with A
3) Permutations that start with T

Permutations

- The Permutation algorithm:
  - The idea is as follows:
    - Notice what happens:
    - What can we say about ALL of the permutations that start with the letter C?
      - Think about recursion…
      - Think about the idea of wanting to reduce your problem to a smaller problem of the same form…
    - ALL of the permutations that start with the letter C,
      - Are SIMPLY three-character strings that are formed by attaching C to the front of ALL permutations of “AT”
    - So this is nothing but another, smaller permutation problem of the same form!!!

Permutations – Recursive Calls

- The Permutation algorithm:
  - The # of recursive calls needed:
  - General “rule of thumb” for recursion:
    - “recursive functions don’t have loops”
      - cuz we use recursion!
      - Either you have iteration, hence loops
      - Or recursion…no need for loops
  - However, this rule of thumb is just that
    - It’s not always true
    - One exception is this permutation algorithm

Permutations – Recursive Calls

- The Permutation algorithm:
  - The # of recursive calls needed:
  - Look at the example with three letters, CAT
    - We need THREE recursive calls, one for each letter
    - Remember, we said we split the work into three groups:
      1) Permutations that start with C
      2) Permutations that start with A
      3) Permutations that start with T
  - But what if we were permuting the letters of the word “computer”
    - EIGHT recursive calls would be needed
    - 1 for each possible starting letter

Permutations – Recursive Calls

- The Permutation algorithm:
  - The # of recursive calls needed:
  - So we see the need for a loop in our algorithm:

```
for (each possible starting letter) {
        list all permutations that start
        with that letter
}
```

  - Now, what is the terminating condition?

Permutations – Recursive Calls

- The Permutation algorithm:
  - The # of recursive calls needed:
  - Terminating condition:
    - Permuting either 0 or 1 element
    - Right.?.
      - Cause if there is only 1 element or 0 elements, **then there is nothing to permute!**
    - In our code, we will use 0 as the terminating condition
      - When there are 0 elements left
      - This can only be done in one way

Permutations – Extra Parameter

- The Permutation algorithm:
  - Use of an extra parameter:
    - As seen previously, some recursive functions take in an extra parameter
      - When compared to their iterative counterparts
      - Usually for the purpose of reducing towards the terminating, or base, case
    - This is the case for our permutation algorithm
      - In order for the recursive permutation to work correctly
      - We must specify one additional piece of information
  - And now to our function…

Permutations – Recursive Function

- The Permutation algorithm:
  - Function Prototype
    - With Pre-conditions and Post-conditions:

```
// Pre-condition:  str is a valid C String, and
//                 k is non-negative and less than
//                 or equal to the length of str.
// Post-condition: All of the permutations of str
//                 with the first k characters fixed
//                 in their original positions are
//                 printed.  Namely, if n is the
//                 length of str, then (n-k)!
//                 permutations are printed.
void RecursivePermute(char str[], int k);
```

    - So k refers to the **first k characters that are fixed in their original positions**

Permutations – Recursive Function

- The Permutation algorithm:
  - Terminating condition:
    - Terminate when k is equal to the length of the string, **`str`**
      - Think about that:
      - k refers to the first k characters in the string that are fixed
      - So if k is equal to the length of the actual string
      - This means that ALL of the letters in str are fixed!
      - If/when this becomes the case
        - We simply want to print out that permutation
  - If we do NOT terminate:
    - **We want a for loop that tries each character at index k**

Permutations – Recursive Function

- The Permutation algorithm:
  - The main for loop within the recursive algorithm:

```
for (j=k; j<strlen(str); j++) {
        ExchangeCharacters(str, k, j);
        RecursivePermute(str, k+1);
        ExchangeCharacters(str, j, k);
}
```

  - But how do we get different characters (the ‘C’, the ‘A’, and the ‘T’) at the first position???
    - C is the first character in the word CAT
    - So how do we make ‘A’ become the first character

Permutations – Recursive Function

- The Permutation algorithm:
  - The main for loop within the recursive algorithm:

```
for (j=k; j<strlen(str); j++) {
        ExchangeCharacters(str, k, j);
        RecursivePermute(str, k+1);
        ExchangeCharacters(str, j, k);
}
```

  - **`ExchangeCharacters`** function:
    - This function will take in our string (**`str`**) and it will SWAP two characters within that string.
    - Which two characters:
      - The character at **index k** will SWAP with the one at **index j**

Permutations – Recursive Function

- The Permutation algorithm:
  - Let's take a closer look at this specific function:

```
// Pre-condition: str is a valid C String and i and j are
//                valid indexes to that string.
// Post-condition: The characters at index i and j will
//                 be swapped in str.
void ExchangeCharacters(char str[], int i, int j) {
       char temp = str[i];
       str[i] = str[j];
       str[j] = temp;
}
```

    - So we send over a string and then SWAP the characters at the two specified indices

Permutations – Recursive Function

- The Permutation algorithm:
  - Again, the main loop within the recursive algorithm:

```
for (j=k; j<strlen(str); j++) {
        ExchangeCharacters(str, k, j);
        RecursivePermute(str, k+1);
        ExchangeCharacters(str, j, k);
}
```

  - ExchangeCharacters function:
    - Remember the three letter example, CAT
    - We said that we need to find ALL permutations with C as the first character, A as the first, and with T as the first

Permutations – Recursive Function

- The Permutation algorithm:
  - Again, the main loop within the recursive algorithm:

```
for (j=k; j<strlen(str); j++) {
        ExchangeCharacters(str, k, j);
        RecursivePermute(str, k+1);
        ExchangeCharacters(str, j, k);
}
```

  - ExchangeCharacters function:
    - This function SWAPS the two characters at the indices passed in as the last two arguments to the function
    - We then **recursively call the permute function**
    - Then we SWAP the characters back to their spots

Brief Interlude: Human Stupidity

Permutations – Recursive Function

```
void RecursivePermute(char str[], int k) {
        int j;

        // Base-case: All fixed, so print str.
        if (k == strlen(str))
                printf("%s\n", str);
        else {
                // Try each letter in spot j.
                for (j=k; j<strlen(str); j++) {
                        // Place next letter in spot k.
                        ExchangeCharacters(str, k, j);

                        // Print all with spot k fixed.
                        RecursivePermute(str, k+1);

                        // Put the old char back.
                        ExchangeCharacters(str, j, k);
                }
        }
}
```

Let's look at this in more detail.

Permutations – Recursive Function

- The Permutation algorithm:
  - Code in detail:
    - We send over two parameters to the function:
    1) The actual string we want to permute
    2) And the integer k
       - Represents the first k characters that are FIXED at their spots

```
void RecursivePermute(char str[], int k) {
        int j;

        // Base-case: All fixed, so print str.
        if (k == strlen(str))
                printf("%s\n", str);
```

Permutations – Recursive Function

- The Permutation algorithm:
  - Code in detail:
    - Using CAT as our example string:
    1) We send over the string, CAT
    2) And the integer k (currently set to zero)
       - Representing that **ZERO characters are initially FIXED**.

```
void RecursivePermute(char str[], int k) {
        int j;

        // Base-case: All fixed, so print str.
        if (k == strlen(str))
                printf("%s\n", str);
```

Permutations – Recursive Function

- The Permutation algorithm:
  - Code in detail:
    - Base case:
      - If k is equal to the length of our string
        - Meaning that **ALL characters are fixed**
      - Then **there is no more characters to permute**
      - Just print out the resulting string!

```
void RecursivePermute(char str[], int k) {
        int j;

        // Base-case: All fixed, so print str.
        if (k == strlen(str))
                printf("%s\n", str);
```

Permutations – Recursive Function

- The Permutation algorithm:
  - Code in detail:
    - ALL other cases (non-base cases):
      - If k does NOT equal the length of the string
      - Means there are some characters that have not been FIXED
      - Means that there are more options to permute
      - We have to try those unused characters at index k

```
void RecursivePermute(char str[], int k) {
        // PREVIOUS CODE
        else {
                // Try each letter in spot j.
                for (j=k; j<strlen(str); j++) {
                        //
                        // ... code here
                        //
```

Permutations – Recursive Function

- The Permutation algorithm:
  - Code in detail:
    - ALL other cases (non-base cases):
      - So we call this for loop
      - **It iterates the number of times EQUAL to the number of possible characters that can go into index k**

```
for (j=k; j<strlen(str); j++) {
        // Place next letter in spot k.
        ExchangeCharacters(str, k, j);
        // Print all perms. with spot k fixed.
        RecursivePermute(str, k+1);
        // Put the old char back.
        ExchangeCharacters(str, j, k);
}
```

Permutations – Recursive Function

- The Permutation algorithm:
  - Code in detail:
    - ALL other cases (non-base cases):
      - Again, k refers to the number of FIXED positions
      - For example, if k is 2
        - Meaning, index 0 and index 1 are FIXED
      - Then the first NON-FIXED location is index 2 …**the value of k!**

```
for (j=k; j<strlen(str); j++) {
        // Place next letter in spot k.
        ExchangeCharacters(str, k, j);
        // Print all perms. with spot k fixed.
        RecursivePermute(str, k+1);
        // Put the old char back.
        ExchangeCharacters(str, j, k);
}
```

Permutations – Recursive Function

- The Permutation algorithm:
  - Code in detail:
    - For all possible characters that could be placed at index k (the next possible NON-FIXED spot):
      - ExchangeCharacters(str, k, j)
        - Means SWAP the characters at index k and j
        - **Meaning, try all possible (remaining) values at index k**

```
for (j=k; j<strlen(str); j++) {
        // Place next letter in spot k.
        ExchangeCharacters(str, k, j);
        // Print all with spot k fixed.
        RecursivePermute(str, k+1);
        // Put the old char back.
        ExchangeCharacters(str, j, k);
}
```

Permutations – Recursive Function

- The Permutation algorithm:
  - Code in detail:
    - For all possible characters at index k:
      - So if we had just started this function
        - Input was CAT for the string and **k equal to zero**
      - this for loop would run three times (length of CAT)
        - Each time, the first line would try each character at index 0

```
for (j=k; j<strlen(str); j++) {
        // Place next letter in spot k.
        ExchangeCharacters(str, k, j);
        // Print all with spot k fixed.
        RecursivePermute(str, k+1);
        // Put the old char back.
        ExchangeCharacters(str, j, k);
}
```

Permutations – Recursive Function

- The Permutation algorithm:
  - Code in detail:
    - For all possible characters at index k:
      - This is what we said earlier, split the work into 3 parts:
        - Permutations that start with C
        - Permutations that start with A
        - Permutations that start with T

```
for (j=k; j<strlen(str); j++) {
        // Place next letter in spot k.
        ExchangeCharacters(str, k, j);
        // Print all with spot k fixed.
        RecursivePermute(str, k+1);
        // Put the old char back.
        ExchangeCharacters(str, j, k);
}
```

Permutations – Recursive Function

- The Permutation algorithm:
  - Code in detail:
    - So the for loop iterates three times (for CAT)
      - First line of code makes each letter the first spot of the string
      - The second line then recursively calls the function
        - The arguments are the string (updated with a new, 1st character)
        - And the new value for k (referring to the # of FIXED spots)

```
for (j=k; j<strlen(str); j++) {
        // Place next letter in spot k.
        ExchangeCharacters(str, k, j);
        // Print all with spot k fixed.
        RecursivePermute(str, k+1);
        // Put the old char back.
        ExchangeCharacters(str, j, k);
}
```

Permutations – Recursive Function

- The Permutation algorithm:
  - Code in detail:
    - So the for loop iterates three times (for CAT)
      - Third and final line of code
      - Simply switches back the characters that we swapped with the first line of code (of the for loop)

```
for (j=k; j<strlen(str); j++) {
        // Place next letter in spot k.
        ExchangeCharacters(str, k, j);
        // Print all with spot k fixed.
        RecursivePermute(str, k+1);
        // Put the old char back.
        ExchangeCharacters(str, j, k);
}
```

Recursion

WASN'T THAT BODACIOUS!

Daily Demotivator

More Recursion: Permutations

Computer Science Department
University of Central Florida

COP 3502 – Computer Science I