Recursion (III): Multiple Recursion

CSCD 300 – Data Structures

Eastern Washington University

Outline

1. Multiple recursion
2. Example: print the permutations of a given string

Multiple recursion

A multiple recursion is one that has more than two recursive function calls inside the function body.

Example: print the permutations of a given string

Definition: permutation

A permutation of a sequence of elements is a particular shuffle of the elements in the sequence.

An example

There are six different permutations of the sequence "abc":

$$\texttt{abc}, \texttt{acb}, \texttt{bac}, \texttt{bca}, \texttt{cab}, \texttt{cba}$$

Our task

Given a sequence of elements, print all its permutations.

- We assume all the elements in the sequence are distinct.
- We have no requirement on the order at which the permutations are printed.

Theorem

Given a sequence of n distinct elements, there are a total of $n!$ distinct permutations of these n distinct elements.

Proof idea

- We have n choice for the first element in the permutation.
- After the first element is chosen, we have $n-1$ choices for the second element in the permutation.
- After the second element is chosen, we have $n-2$ choices for the third element in the permutation.
- ...
- After $(n-2)$th element is chosen, we have 2 choices for the $(n-1)$th element in the permutation.
- After $(n-1)$th element is chosen, we have 1 choice for the nth element in the permutation.

So, altogether the number of different permutations we can choose is

$$n \cdot (n-1) \cdot (n-2) \cdots 2 \cdot 1 = n!$$

The minimum time needed to print all perms

Because we are trying to print out all the $n!$ permutations, the time cost of any program to do this job is expected to take at least $O(n!)$ time.

A recursion-based idea for printing all permutations

- **Input:** A sequence of n distinct elements stored in array $S[0 \ldots n-1]$.
- **Idea:** `permute(StringBuffer S, int k)` prints those permutations of S obtained from by only shuffling $S[k, k+1, \ldots, n-1]$ [1].

The function call `permute(S,0)` prints all the permutations of S, which can be recursively implemented by:

1. swapping $S[0]$ with each element in $S[0 \ldots n-1]$, so every element in S will have the chance to occupy the $S[0]$ spot.
2. after the swap of each pair, shuffle $S[1 \ldots n-1]$ by calling `permute(S,1)` to get all the permutations with $S[0]$ fixed.
3. `permute(S,1)` will also be recursively implemented ...

Pseudo code follows ...

[1] In Java, the data type String is immutable, so we use StringBuffer.

```
permute(StringBuffer S, int k) //s.length = n
{
   if (k == n-1) //all positions in the perm are fixed,
      print S;   //so it's ready to be printed.
   else
      //multiple recursive calls within this loop
      for(i = k; i < n; i++){
         swap(S[k],S[i]);  // try a different element for the k^th
                           // position in the perm, and fix it.

         permute(S, k+1);  //recursive call
         swap(S[k],S[i]);
      }
}
```

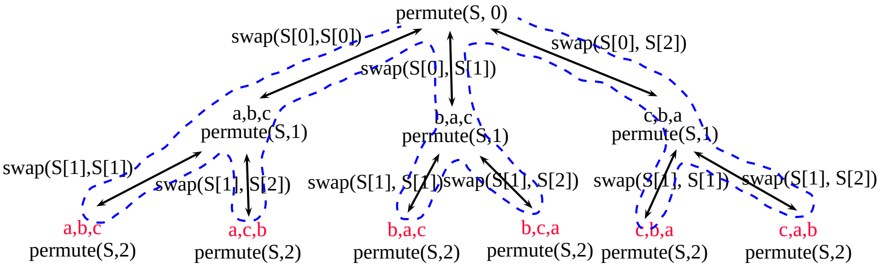
Java code follows ...

Java code [2]

```
public class test_permutation{
   public static void main(String[] args){
      String s = "ABC";
      if(args.length > 0) s = args[0];
      print_perms(s);
    }
    public static void print_perms(String s){
       permute(new StringBuffer(s), 0);
    }
    private static void permute(StringBuffer s, int k){
       int n = s.length();
       if(k == n-1) System.out.println(s);
       else
          for(int i = k; i < n; i++){
             swap(s, i, k); permute(s, k+1); swap(s, i, k);
          }
    }
    private static void swap(StringBuffer s, int i, int j){
       if(i != j) {
          char ch = s.charAt(i); s.setCharAt(i, s.charAt(j)); s.setCharAt(j, ch);
       }
    }
}
```

[2] Code is borrowed and modified from *Data Structures with Java* by Hubbard&Huray, page 316.

An example: input string $S[0...2] = [a, b, c]$



- All those in red are screen prints of the permutations.
- The blue dash line shows the trace of recursive calls.

Time complexity

Let's look at the initial call that works with the original array S of size n.

```
permute(StringBuffer S, int k = 0) //T(n) = ?
{                       _
   if (k == n-1)         | time cost: c_1
      print S;          _|
   else
      for(i = k; i < n; i++){  //#steps = n
         swap(S[k],S[i]);      //time cost: c_2
         permute(S, k+1);      //time cost: T(n-1) = ?
         swap(S[k],S[i]);      //time cost: c_3
      }
}
```

So ...

$T(n) = c_1 + n(c_2 + c_3 + T(n-1)) = c_1 + (c_2 + c_3)n + nT(n-1) \geq nT(n-1)$

(continue ...)

$$
\begin{aligned}
T(n) &\geq nT(n-1) \\
T(n-1) &\geq (n-1)T(n-2) \\
T(n-2) &\geq (n-2)T(n-3) \\
&\vdots \\
T(2) &\geq 2T(1)
\end{aligned}
$$

Therefore:

$$
\begin{aligned}
T(n) &\geq nT(n-1) \\
&\geq n(n-1)T(n-2) \\
&\geq n(n-1)(n-2)T(n-3) \\
&\geq \ldots \\
&\geq n(n-1)(n-2)\ldots 2T(1) \\
&= c \cdot n! \qquad (T(1) \text{ is just some constant } c.) \\
&= O(n!)
\end{aligned}
$$

This says, the time cost of the program for an input array of size n is at least $O(n!)$, which is consistent with our expectation from Page 6.