22C:21 Lecture Notes
Recursion: generating permutaions

Sept 7, 2005

Today we will deal with a more substantial example of recursion. The problem is to write a program that takes as input a positive integer n and prints out all possible permutations of the numbers $1, 2, \ldots, n$.

```
1 2 3
1 3 2
2 1 3
2 3 1
3 1 2
3 2 1
```

So we need to write a function

```
public static void genPerms(int n)
{
    ...
}
```

How do we break this problem up into smaller problems? One way to do it is the following. The

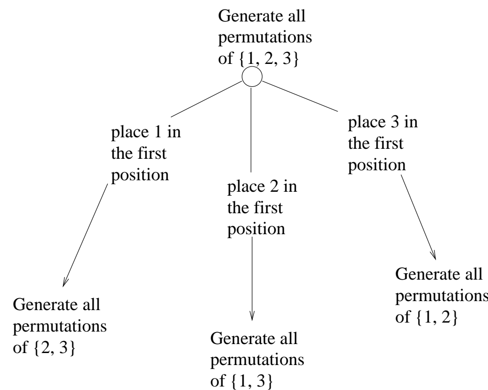

Figure 1: The problem of generating all permutations of $\{1, 2, 3\}$ has been reduced to the problems of generating all permutations of $\{1, 2\}$, $\{1, 3\}$, and $\{2, 3\}$.

typical problem that we need to solve is the following. Let $A \subseteq \{1, 2, \ldots, n\}$ be an arbitrary subset of size m. Let $B = \{1, 2, \ldots, n\} - A$. The elements in A have been placed in the first m slots of an array. We now need to generate all the permutations of elements in B.

The pseudocode for our solution is

```
for each element x in B do
      place x in position m+1
      recursively generate all permutations of B - {x}
```

We now need to figure out what information we need to pass to each recursive call. There are several ways to do this. One simple option is to keep two arrays, one called *perms* to keep track of the actual permutation being generated and the other called B (as in the above code) which keeps track of the subset of elements whose permutations we need to generate. We may also want to send in m, the number of elements which have already been placed. So the header of the recursive version of `genPerms` function will look like

```
public static void recursiveGenPerms(int n, int[] perms,
                                     int m, int[] B)
```

The code that implements the above idea is below.

```
int rest = n - m; // number of elements in B

// for each element in B do
for(int i = 0; i < rest; i++)
{
      int x = B[i];

      // Place the element from B into position m+1 in perms
      perms[m] = x;

      // Make a temporary copy of B without the element x
      // So this function does both: copy and delete
      int[] temp = new int[rest-1];
      copyAndDelete(B, temp, i);


      // Delete x from temp
      delete(temp, x);

      // Make the recursive call to generate all permutations
      // of elements in perm
      recursiveGenPerms(n, perms, m+1, temp);
 }
```

We still need to figure out the base cases. Clearly, when there are no elements in B, then we are done. This can be checked by testing if *rest* (which equals the number of elements in B) is 0. If $rest == 0$ then it means that an entire permutation has been generated in the array *perms* and it is time to print this out. The code for this is below.

```
int rest = n - m; // number of elements in B

if (rest == 0)
{
      printArray(perms, n);
      return;
}

// the rest of the code goes here
```

This completes `recursiveGenPerms`. We now just need to complete the "wrapper" function `genPerm`.

```
public static void genPerms(int n)
{
        // perms has size n, but initially nothing is
        // placed in it
        int[] perms = new int[n];

        int[] B = new int[n];
```

```
        // Initially B contains everything
        for(int i = 0; i < n; i++)
                B[i] = i+1;

        recursiveGenPerm(n, perms, 0, B);
}
```