Recursive Counting

CS125 Spring 2007

Arthur Kantor
Paths through a lattice

- You are allowed to step only down or to the right.
- How many paths are there from (0,0) to (2,3)?
Paths through a lattice

- More generally: How many paths are there from \((i,j)\) to \((R,C)\)?
- Let the function \(\text{CountPaths}(i,j,R,C)\) give us that answer
• What self similar sub-problems would assist us in solving CountPaths(i,j,R,C)?
What self similar sub-problems would assist us in solving $\text{CountPaths}(i, j, R, C)$?

- Knowing $\text{CountPaths}(i+1, j, R, C)$ and $\text{CountPaths}(i, j+1, R, C)$ would be useful.

Given the solutions to the two sub-problems what’s the solution to $\text{CountPaths}(i, j, R, C)$?
General case

\begin{itemize}
  \item \texttt{CountPaths}(i,j,R,C) = \texttt{CountPaths}(i+1,j,R,C) + \texttt{CountPaths}(i,j+1,R,C)
\end{itemize}

\begin{verbatim}
public static int CountPaths(int i, int j, int R, int C) {
    return CountPaths(i + 1, j, R, C) +
           CountPaths(i, j + 1, R, C);
}
\end{verbatim}
Base case

- If you are beyond the edge of the lattice, then there are no legal paths to \((R,C)\)
  - If \(i>R\) or \(j>C\) \(\text{CountPaths}(i,j,R,C)\) must return 0

```java
public static int CountPaths(int i, int j, int R, int C) {
    if (i>R || j>C)
        return 0;
    else
        return CountPaths(i + 1, j, R, C) +
               CountPaths(i, j + 1, R, C);
}
```
Will this work?

```
public static int CountPaths(int i, int j, int R, int C)
{
    if (i > R || j > C)
        return 0;
    else
        return CountPaths(i + 1, j, R, C) +
                CountPaths(i, j + 1, R, C);
}
```
Will this work? No.

- 0 is the result of every sub-problem!

Need one more special case:
If we are at (R,C), there is one path to (R,C): it requires 0 steps.

```java
public static int CountPaths(int i, int j, int R, int C) {
    if (i>R || j>C)
        return 0;
    else
        return CountPaths(i + 1, j, R, C) +
               CountPaths(i, j + 1, R, C);
}
```
This will work

public static int CountPaths(int i, int j, int R, int C)
{
    if (i>R || j>C)
        return 0;
    else if(i==R && j == C)
        return 1;
    else
        return CountPaths(i + 1, j, R, C) +
               CountPaths(i, j + 1, R, C);
}

This figure illustrates the function call tree for the CountPaths function.
Variations

- What if you were also allowed to move diagonally?
Variation on the problem

- How would the general case change?
- How would the base case change?

```java
public static int CountPaths(int i, int j, int R, int C) {
    if (i>R || j>C) {
        return 0;
    } else if (i==R && j == C) {
        return 1;
    } else {
        return CountPaths(i + 1, j, R, C) + CountPaths(i, j + 1, R, C);
    }
}
```
Variation on the problem

- How would the general case change?
- How would the base case change?

```java
public static int CountPaths(int i, int j, int R, int C)
{
    if (i>R || j>C)
        return 0;
    else if(i==R && j == C)
        return 1;
    else
        return CountPaths(i + 1, j, R, C) +
                CountPaths(i + 1, j + 1, R, C) +
                CountPaths(i, j + 1, R, C);
}
```
Making change

- You have various coin amounts.
- What is the maximum numbers of ways you can have a given quantity of money?
Making change: example

• You have 1$, 2$, 5$ and 10$ bills.
• How many ways can you get 62$ ?

• If I have 62$ in my hand? How many 10$ bills can I have in my hand?
  – 0 or 1 or 2 or 3 or 4 or 5 or 6. That’s it.
Making change: example

• If I have 62$ in my hand? How many 10$ bills *can* I have in my hand?
  – 0 or 1 or 2 or 3 or 4 or 5 or 6. That’s it.

• If I have 62$ in my hand *and* 4 10$ bills, how many 5$ bills can I have in my hand?
  – The same number of 5$ bills if I had 22$ and 0 10$ bills.
  – 0 or 1 or 2 or 3 or 4. That’s it.
Finding the sub-problem

• Say I use N 10$ bills in my way to make up 62$, this way will always be different from any way where I use M\neq N 10$ bills to make up 62$.

• The sub-problem then is to use N units of one kind of bills, and disallow the use of those bills in the future.

• The size-of-the-problem parameter is the number of kinds of bills you can use
makeChange() function

- int makeChange(int amount, int[] denom, int allowedDenom);
  - amount: desired amount of money
  - denom: array of denominations: {1, 2, 5, 10}
  - allowedDenom: the last usable denomination in denom.
    - E.g. if allowedDenom is 2, you can use bills of denom[0], denom[1], and denom[2], or 1$, 2$ and 5$ bills.
Base cases

- if allowedDenom < 0, I cannot use any denominations.
  - makeChange() must return 0 in that case

```c
int makeChange(int amount, int[] denom, int allowedDenom){
    if (allowedDenom<0)
        return 0;
}
```
Base cases

• if allowedDenom < 0, I cannot use any denominations.
  – makeChange() must return 0 in that case

• Otherwise, if amount == 0, there is exactly one way to make change, using any bills
  – Use 0 bills of all denominations

```java
int makeChange(int amount, int[] denom, int allowedDenom){
    if (allowedDenom<0)
        return 0;
    else if (amount==0)
        return 1;
    }
```
General case

- Use N bills of type d
- Decrease the target amount appropriately
- Disallow the use of bills of type d and solve the sub-problem with d disallowed and the new amount.
- For each allowed N, add up all the ways returned by the sub-problems

```c
int makeChange(int amount, int[] denom, int allowedDenom)
{
    if (allowedDenom<0)
        return 0;
    else if (amount=0)
        return 1;
    else{
        int ways =0;
        int d = denom[allowedDenom]; //denom we will disallow in the sub-problem
        for(int N=0; N*d<=amount;N++)
            ways = ways + makeChange(amount-N*d, denom, allowedDenom-1);
        return ways;
    }
}
```
Sum of all path weights

- Weight of a path = sum of weights along each of the steps
  - E.g. 1+6+…
- Find the total weight of all the paths through the lattice
• Knowing the sum of paths through the first N-1 steps is easier and seems helpful.
Consider the weight of all paths of n steps ending on ith row.  

- Call it partialWeight(n,i)

The total sum of path weights of n steps long is  

- weight(n) = partialWeight(n,1) + partialWeight(n,2) + ... + partialWeight(n,M)

partialWeight(1,3) = 1 + 3 + 0 = 4
• Call the edge on the $n$th step from node $i$ to node $j$ edge($n,i,j$)
  – $n$ ranges from 1 to $N$
  – $i$ and $j$ range from 1 to $M$
Recursion step

- Assume you know $\text{partialWeight}(n-1,i)$ for all $i=1\ldots M$
- $\text{partialWeight}(n,i) =$
  - $\text{partialWeight}(n-1,1) + \text{edge}(n,1,i) +$
  - $\text{partialWeight}(n-1,2) + \text{edge}(n,2,i) +$
  - $\ldots +$
  - $\text{partialWeight}(n-1,2) + \text{edge}(n,M,i)$
Recursion step

- Assume you know $\text{partialWeight}(n-1,i)$ for all $i=1\ldots M$

\[
\text{partialWeight}(n,i) = \\
\text{partialWeight}(n-1,1) + \text{edge}(n,1,i) + \\
\text{partialWeight}(n-1,2) + \text{edge}(n,2,i) + \\
\ldots + \\
\text{partialWeight}(n-1,M) + \text{edge}(n,M,i)
\]
Base case $N=0$

- $\text{partialWeight}(0,1)=0$
- $\text{partialWeight}(0,2)=0$
- $\ldots$
- $\text{partialWeight}(0,M)=0$
Implementation of partialWeight ()

- int partialWeight(int[][][]] edges, int n, int row)
  - edges: the array with all the edges. So edges[n][i][j] is an edge at step n from row i to row j
  - n: step n
  - row: the row of the lattice
Implementation of partialWeight ()

```c
int partialWeight(int[][][] edges, int n, int row){
    int weight=0;
    for(int i =0; i<edges[0][0].length;i++)
        weight= weight + partialWeight(edges,n-1,i) +
        edges[n][i][row];

    return weight;
}
```
Implementation of weight ()

```c
int partialWeight(int[][][] edges, int n, int row){
    int weight=0;
    for(int i =0; i<edges[0][0].length;i++)
        weight= weight + partialWeight(edges,n-1,i) +
               edges[n][i][row];

    return weight;
}

int weight(int[][][] edges, int n){
    int weight=0;
    for(int i =0; i<edges[0][0].length;i++)
        weight= weight + partialWeight(edges,n,i);

    return weight;
}
```