lec15

August 12, 2016

1 More memoization

2 Example: number of ways to sum up to an integer

Let `numWays(n)` be the number of ways to write a nonnegative integer `n` as the sum of positive integers. For example, there are `8` ways of writing `4`: `1 + 1 + 1 + 1`, `2 + 1 + 1`, `1 + 2 + 1`, `1 + 1 + 2`, `2 + 2`, `1 + 3`, `3 + 1`, and `4`. One can show by induction that `numWays(n)` $= 2^{n-1}$, but let's see how to calculate it using recursion and memoization.

2.0.1 Recursive implementation without memoization

```
In [8]: def numWays(n):
            if n==0:
                return 1
            ans = 0
            for i in range(1, n+1):
                # try the first number in sum being i, then the remaining part must
                ans += numWays(n-i)
            return ans
```

2.0.2 With memoization

```
In [9]: def memNumWays(n, seen, mem):
            if n==0:
                return 1
            elif seen[n]:
                return mem[n]
            seen[n] = True
            mem[n] = 0
            for i in range(1, n+1):
                mem[n] += memNumWays(n-i, seen, mem)
            return mem[n]

        def numWaysFast(n):
            seen = [False]*(n+1)
            mem = [0]*(n+1)
            return memNumWays(n, seen, mem)
```

```
In [10]: numWays(4)
```

```
Out[10]: 8
```

```
In [11]: numWays(14)
```

```
Out[11]: 8192
```

```
In [15]: numWays(24)
```

```
Out[15]: 8388608
```

```
In [16]: # now using the memoized version
         numWaysFast(24)
```

```
Out[16]: 8388608
```

2.0.3 Another example

What if we want to compute a function distinctNumWays(n) which doesn't differentiate between different orderings of the same sum? For example, it treats `1 + 1 + 2` and `2 + 1 + 1` as the same sum. So, there would only be `5` ways to sum up to the number `4`: `1 + 1 + 1 + 1, 1 + 2 + 2, 2 + 2, 1 + 3, 4`.

We can calculate `distinctNumWays(n)` recursively as well, by generating all ways of forming `n` where the integers in the sum are generating in nondecreasing order. That is, we would not generate `2 + 1 + 1` or `1 + 2 + 1` since the integers do not appear in nondecreasing order; we would only generate `1 + 1 + 2`. That way, we never count each sum exactly once.

```
In [19]: # how many ways are there to sum up to n, not counting different
         # orderings of the sum, when the smallest number must be at least
         # atLeast
         def recurse(n, atLeast):
             if n==0:
                 return 1
             ans = 0
             for i in range(atLeast, n+1):
                 ans += recurse(n-i, i)
             return ans

         def distinctNumWays(n):
             return recurse(n, 1)
```

2.0.4 Now with memoization

```
In [35]: def recurseMem(n, atLeast, seen, mem):
             if n==0:
                 return 1
             elif seen[n][atLeast]:
                 return mem[n][atLeast]
             seen[n][atLeast] = True
```

```
            mem[n][atLeast] = 0
            for i in range(atLeast, n+1):
                mem[n][atLeast] += recurseMem(n-i, i, seen, mem)
            return mem[n][atLeast]

        def distinctNumWaysFast(n):
            mem = [[-1 for x in range(n+1)] for y in range(n+1)]
            seen = [[False for x in range(n+1)] for y in range(n+1)]
            return recurseMem(n, 1, seen, mem)
```

```
In [21]: distinctNumWays(4)
```

```
Out[21]: 5
```

```
In [31]: distinctNumWays(60)
```

```
Out[31]: 966467
```

```
In [36]: distinctNumWaysFast(60)
```

```
Out[36]: 966467
```

2.0.5 Last example: figuring out how to have the most fun at parties

As asked yesterday: we saw how to figure out how to optimize a function using recursion/memoization, but what if we want to remember the decisions we made to obtain the optimal value?

As discussed yesterday, suppose you have a budget of D dollars. You are given a list L of parties $[[c_0, f_0], \ldots, [c_{n-1}, f_{n-1}]]$ where the i^{th} party costs c_i to attend and will give you f_i units of fun. Yesterday we asked: is the maximum amount of fun you can have with your budget?

Today we will ask the question: what if you want to know specifically which parties to attend to have the most fun while respecting the budget constraint?

```
In [48]: def maximum_fun_recur2(D,L,seen,mem,choices):
             """returns the maximum amount of fun we can have with D dollars attend
             where L is a tuple/list containing pairs (c,f) of cost/fun for every p
             if seen[D][len(L)]:
                 return (mem[D][len(L)],choices)
             if len(L)==0:
                 # if L is empty then we can't have any fun
                 seen[D][0] = True
                 mem[D][0]=0
                 return (0,choices)
             seen[D][len(L)] = True
             fun_if_skip_first_party,c = maximum_fun_recur2(D,L[1:],seen,mem,choice
             if D<L[0][0]: # if we can't afford to attend the first party then we h
                 mem[D][len(L)] = fun_if_skip_first_party
                 choices[D][len(L)] = 'D' # 'D' means "don't go" to first party
                 return (fun_if_skip_first_party,choices)
```

```
        # otherwise we will check both options and see what's the maximum fun
        fun_if_attend_first_party,c = maximum_fun_recur2(D-L[0][0], L[1:],seen
        if fun_if_skip_first_party > L[0][1]+fun_if_attend_first_party:
            mem[D][len(L)] = fun_if_skip_first_party
            choices[D][len(L)] = 'D'
        else:
            mem[D][len(L)] = L[0][1]+fun_if_attend_first_party
            choices[D][len(L)] = 'G' # 'G' means "go" to first party
        return (mem[D][len(L)], choices)

    def memoized_maximum_fun2(D,L):
        seen = [[False]*(len(L)+1) for i in range(D+1)]
        mem = [[-1]*(len(L)+1) for i in range(D+1)]
        choices = [[-1]*(len(L)+1) for i in range(D+1)]
        return maximum_fun_recur2(D,L,seen,mem,choices)

    def whichParties(D, L):
        best,choices = memoized_maximum_fun2(D, L)
        parties = []
        while len(L) > 0:
            c = choices[D][len(L)]
            if c == 'G':
                parties += [L[0]]
                D -= L[0][0]
            L = L[1:]
        return parties
```

```
In [49]: whichParties(4,((1,6),(2,5),(3,6),(2,10)))
```

```
Out[49]: [(1, 6), (2, 10)]
```

2.0.6 Exercise 1

Remember in yesterday's exercises, you were given an arithmetic expression with digits separated by $*$ and $+$ and were asked: how can you parenthesize the expression so as to maximize its value? For example, with the expression $1+2*3+4*5$ the best way of parenthesizing it is $(1+2)*((3+4)*5)$, giving 105. For example, parenthesizing it as $1+((2*3)+(4*5))$ would only give 27.

In today's lab, implement a function `withParens(s)` which outputs the string `s` parenthesized in a way that achieves the maximum value. For example, `withParens('1+2*3+4*5')` should return `'(1+2)*((3+4)*5)'`

```
In [15]: def withParens(s):
             # write your code here
             pass
```

2.0.7 Exercise 2

Consider the function `makeChange(n,L)` from yesterday's lab, which returned the minimum number of coins needed from the list of denominations `L` to make change for `n` cents. Write a function `whichCoins(n, L)` which actually returns a list of coins used to make change in this optimal way.

```
In [ ]: def whichCoins(n,L):
            # write your code here
            pass
```