Recitation 4: Recursion and Dynamic Programming

1 Recursion

1. In class you saw how to implement binary search via recursion. Now try to implement it as an iterative algorithm instead.

   Solution. Suppose the input is a sorted array $A[1 \dots n]$ and a target value x. An iterative version of binary search maintains the current search interval $[\ell, r]$ and repeatedly halves it:

Algorithm 1 IterativeBinarySearch(A, x)

```
1: ℓ ← 1, r ← n
2: while ℓ ≤ r do
3:     m ← ⌊(ℓ + r)/2⌋
4:     if A[m] = x then
5:         return m                          ▷ found at index m
6:     else if A[m] < x then
7:         ℓ ← m + 1
8:     else
9:         r ← m − 1
10:    end if
11: end while
12: return "not found"
```

   At each iteration the size of the interval $[\,\ell, r\,]$ is at least halved, so the number of iterations (and thus the running time) is $O(\log n)$.

2. In class you saw a recursive algorithm to compute the n-th power of real number x. This algorithm ran in $\Omega(n)$ time. Is it possible to design a recursive algorithm that runs in $O(\log(n))$ time? (You may assume that n is an integer.) If so, provide the algorithm. If not, prove that it's impossible.

   Solution. Yes. We can use exponentiation by squaring.

   - If $n = 0$, return 1.
   - If n is even, write $n = 2k$ and note $x^n = x^{2k} = (x^k)^2$.
   - If n is odd and positive, write $n = 2k + 1$ and note $x^n = x^{2k+1} = x \cdot x^{2k}$.

   This leads to the recursive algorithm:

Algorithm 2 FastPower(x, n)

```
if n = 0 then
    return 1
else if n is even then
    y ← FastPower(x, n/2)
    return y · y
else
    return x · FastPower(x, n − 1)
end if
```

Let $T(n)$ be the running time (number of recursive calls). For even n, we have

$$T(n) = T(n/2) + O(1),$$

and for odd n, $T(n) = T(n-1) + O(1)$, followed by the even case. Overall, the argument n is reduced to at most $\lfloor n/2 \rfloor$ after at most two steps, so the recursion depth is $O(\log n)$, and thus the running time is $O(\log n)$.

3. Design a recursive algorithm which generates all the subsets of $\{1, \ldots, n\}$. It should take n as input.

   Solution. A natural recursive approach is to decide, for each $i \in \{1, \ldots, n\}$, whether i is included in the subset or not. We can implement a helper procedure that builds subsets incrementally.

   Let the main function call a recursive helper:

Algorithm 3 GenerateSubsets(n)

```
Backtrack(1, ∅)
```

The helper procedure:

Algorithm 4 Backtrack(i, S)

```
if i > n then
    print S                          ▷ S is a complete subset
    return
end if
Backtrack(i + 1, S)                  ▷ Case 1: do not include i
Backtrack(i + 1, S ∪ {i})            ▷ Case 2: include i
```

At each level i we branch into two choices (include or exclude i), so this generates all 2^n subsets of $\{1, \ldots, n\}$.

2 Dynamic Programming

1. Suppose you can climb stairs either one or two steps at a time. Design an algorithm to compute the total number of distinct ways to climb a staircase with n stairs.

Solution. Let ways[k] be the number of distinct ways to climb to the top of a staircase with k steps.

- To reach step k, either the last move was from step $k-1$ (a 1-step) or from step $k-2$ (a 2-step).
- Thus every way to reach step k either
  (a) is a way to reach step $k-1$ followed by a 1-step, or
  (b) is a way to reach step $k-2$ followed by a 2-step.
- These two sets are disjoint, so

$$\text{ways}[k] = \text{ways}[k-1] + \text{ways}[k-2].$$

We also need base cases. There is exactly one way to be at step 0 (do nothing), and one way to reach step 1 (take a single 1-step), so:

$$\text{ways}[0] = 1, \qquad \text{ways}[1] = 1.$$

This leads to the following DP:

Algorithm 5 CountWays(n)

```
1: if n = 0 or n = 1 then
2:     return 1
3: end if
4: create array ways[0 ... n]
5: ways[0] ← 1, ways[1] ← 1
6: for k = 2 to n do
7:     ways[k] ← ways[k − 1] + ways[k − 2]
8: end for
9: return ways[n]
```

The running time and space are both $O(n)$. (We can reduce the space to $O(1)$ by just keeping the last two values.)

2. You are given an unlimited supply of coins with denominations 1, 5, 10, 25, and 100 cents. Given a non-negative integer S (in cents), design an algorithm that returns the number of distinct combinations of coins that sum to S. Two combinations that differ only in the order of the same coins are considered the same.

   Solution. We will use a 2D dynamic programming table. Let the coin denominations be stored in an array $c[1], \ldots, c[m]$, where here $m = 5$ and $c = [1, 5, 10, 25, 100]$.

   Define

$$\text{dp}[i][s]$$

   to be the number of distinct combinations to make sum s using only the first i coin types $\{c[1], \ldots, c[i]\}$. Our goal is to compute dp$[m][S]$.

   Base cases.

- There is exactly one way to make sum 0 using any subset of coins: take no coins. So for all i we set
$$\mathrm{dp}[i][0] = 1.$$
- If we use 0 coin types, we cannot make any positive sum:
$$\mathrm{dp}[0][s] = 0 \quad \text{for all } s > 0.$$

Recurrence. To compute $\mathrm{dp}[i][s]$ for $i \geq 1$ and $s \geq 1$, consider coin $c[i]$:

- We *do not* use coin $c[i]$: then we must make s using only the first $i-1$ coins, which can be done in $\mathrm{dp}[i-1][s]$ ways.
- We *do* use coin $c[i]$ at least once: remove one $c[i]$, leaving sum $s - c[i]$ still to be formed using coins $\{c[1], \ldots, c[i]\}$ (we can still use coin i again). There are $\mathrm{dp}[i][s-c[i]]$ such combinations, provided $s \geq c[i]$.

These two cases are disjoint and cover all possibilities, so
$$\mathrm{dp}[i][s] = \mathrm{dp}[i-1][s] + \begin{cases} \mathrm{dp}[i][s-c[i]], & \text{if } s \geq c[i], \\ 0, & \text{otherwise.} \end{cases}$$

Algorithm.

Algorithm 6 CountCoinCombinations(S)

```
1: c ← [1, 5, 10, 25, 100]
2: m ← 5
3: create table dp[0 ... m][0 ... S] and set all entries to 0
4: for i = 0 to m do
5:     dp[i][0] ← 1                                   ▷ one way to make sum 0
6: end for
7: for i = 1 to m do
8:     for s = 1 to S do
9:         dp[i][s] ← dp[i − 1][s]                    ▷ do not use coin c[i]
10:        if s ≥ c[i] then
11:            dp[i][s] ← dp[i][s] + dp[i][s − c[i]]  ▷ use coin c[i]
12:        end if
13:    end for
14: end for
15: return dp[m][S]
```

The table has $(m+1)(S+1)$ entries, and each is computed in $O(1)$ time, so the running time and memory usage are both $O(mS)$. In our case $m = 5$, so this is $O(S)$ in practice. The row index i ensures that combinations are counted without regard to the order in which coins appear.

3. Given an array of integers A, design an algorithm to compute the longest increasing subsequence (LIS) in A. The numbers in the subsequence do not have to be adjacent in A. For example: If $A = [3, -2, 5, -4]$ the answer is 2. If $A = [12, 1, 2, -1, 10, 5, 7, -3, 12, 6]$ the LIS is $1, 2, 5, 7, 12$ so you should output 5.

Solution. Let $A[1 \dots n]$ be the array. Define:

$$\mathrm{dp}[i] = \text{length of the longest increasing subsequence that ends at position } i.$$

Then:

- Every LIS ending at position i includes $A[i]$ as its last element.
- Any such subsequence can be obtained by taking an LIS ending at some position $j < i$ with $A[j] < A[i]$, and then appending $A[i]$.
- So for each i,

$$\mathrm{dp}[i] = 1 + \max_{\substack{1 \le j < i \\ A[j] < A[i]}} \mathrm{dp}[j],$$

  and if there is no $j < i$ with $A[j] < A[i]$, we just have $\mathrm{dp}[i] = 1$ (the subsequence consisting only of $A[i]$).

After computing all $\mathrm{dp}[i]$, the length of the overall LIS is

$$\max_{1 \le i \le n} \mathrm{dp}[i].$$

Algorithm:

Algorithm 7 LISLength($A[1 \dots n]$)

```
for i = 1 to n do
    dp[i] ← 1
end for
for i = 1 to n do
    for j = 1 to i − 1 do
        if A[j] < A[i] then
            dp[i] ← max(dp[i], dp[j] + 1)
        end if
    end for
end for
return max_{1≤i≤n} dp[i]
```

The outer loop runs n times and the inner loop up to $i-1$ times, so the running time is $O(n^2)$ and the space is $O(n)$.