# Dynamic Programming

Ethan Arnold, Kevin Chen

CS 104C

Spring 2019

# Recursion review

- Fibonacci sequence
  - $F_0 = 0$
  - $F_1 = 1$
  - $F_n = F_{n-2} + F_{n-1}$, for $n \geq 2$
- How do we code this?

# Recursive Fibonacci

```
int fib(int n) {
    if (n == 0) return 0;
    if (n == 1) return 1;
    return fib(n - 2) + fib(n - 1);
}
```

- Problems?
  - Work is repeated
- How do we fix it?

# Recursive Fibonacci with memoization

```
int fib(int n,
        HashMap<Integer, Integer> memo) {

    if (n == 0) return 0;
    if (n == 1) return 1;
    if (memo.containsKey(n))
        return memo.get(n);

    int answer = fib(n - 2, memo) +
                 fib(n - 1, memo);
    memo.put(n, answer);
    return answer;
}
```

- Time complexity? $O(N)$
- Space complexity? $O(N)$

# Memoization

- This technique is called **memoization**
  - Also called **top-down** dynamic programming
- We just save the "smaller" solutions as we go
- Problems?
  - Stack overflow risk
- How do we fix it?

# Iterative Fibonacci

```
int fib(int n) {
    if (n == 0) return 0;
    if (n == 1) return 1;
    int[] dp = new int[n + 1];
    dp[0] = 0;
    dp[1] = 1;
    for (int i = 2; i <= n; i++)
        dp[i] = dp[i - 2] + dp[i - 1];

    return dp[n];
}
```

- Time complexity? $O(N)$
- Space complexity? $O(N)$

# Iterative Fibonacci

```
int fib(int n) {
    if (n == 0) return 0;
    int a = 0;
    int b = 1;
    for (int i = 0; i < n - 1; i++) {
        int newB = a + b;
        a = b;
        b = newB;
    }

    return b;
}
```

- Time complexity? $O(N)$
- Space complexity? $O(1)$

# Bottom-up dynamic programming

- This technique is called **bottom-up** dynamic programming
- No stack overflow risk
- Often a significant constant factor faster than top-down
- Less natural to write at first, but easier to optimize memory
- Must think about iteration order

# Coin change, revisited

- **Problem:** Given a set of $K$ coin values $\{C_1, C_2, \ldots, C_K\}$ and a target value $N$, can we choose an integer quantity $Q_i \geq 0$ for each coin such that $N = Q_1 \cdot C_1 + \cdots + Q_K \cdot C_K$? What is the minimum number of coins ($\sum_{i \leq K} Q_i$) needed to do so?
- As we saw last week, greedy doesn't always work
- Let $DP_i$ = the minimum number of coins to make a target value of $i$
  - $DP_N$ will be our answer
- Can we write $DP_i$ as a recurrence?
  - Base case: $DP_0 = 0$
  - Recursive case: $DP_i = \min_{j \leq K}(DP_{i-Q_j} + 1)$
- How do we code this?

# Coin change solution

```java
int minCoins(int n, int[] coinValues) {
    int[] dp = new int[n + 1];
    dp[0] = 0;
    for (int i = 1; i <= n; i++) {
        dp[i] = INF;
        for (int coinValue : coinValues)
            if (i - coinValue >= 0 &&
                dp[i - coinValue] + 1 < dp[i])
                dp[i] = dp[i - coinValue] + 1;
    }

    return dp[n] >= INF ? -1 : dp[n];
}
```

- Time complexity? $O(KN)$
- Space complexity? $O(N)$ (could be reduced to $O(\max_{i \le K} C_i)$)

# Dynamic programming approach

1. Write the problem as a recurrence (including one or more base cases)
   - This is the hard part!
2. Code the recurrence in a bottom-up (iterative) or top-down (recursive/memoized) manner

# Longest increasing subarray

- **Problem:** Given an array $A$ of length $N$, what is the length of the longest increasing subarray?
  - A subarray $A_{i:j}$ is a *contiguous* segment of the array $A$, identified by a start and end index $i$ and $j$ where $1 \leq i \leq j \leq N$
  - A subarray $A_{i:j}$ is *increasing* if, for every $i \leq k_1 \leq k_2 \leq j$, $A_{k_1} \leq A_{k_2}$
- Can we write a recurrence for this problem?
  - Let $DP_i =$ the length of the longest increasing subarray *ending with element $i$*
  - Our answer will be $\max_{1 \leq i \leq N} DP_i$
  - Base case: $DP_1 = 1$
  - Recursive case:
    - If $A_i \geq A_{i-1}$, then $DP_i = DP_{i-1} + 1$
    - If $A_i < A_{i-1}$, then $DP_i = 1$
- How do we code this?

# Longest increasing subarray solution

```
int longestIncreasingSubarray(int[] arr) {
    int answer = 0;
    int[] dp = new int[arr.length];
    dp[0] = 1;
    for (int i = 1; i < arr.length; i++) {
        if (arr[i] >= arr[i - 1])
            dp[i] = dp[i - 1] + 1;
        else
            dp[i] = 1;
        if (dp[i] > answer)
            answer = dp[i];
    }
    return answer;
}
```

- Time complexity? $O(N)$
- Space complexity? $O(N)$ (could be reduced to $O(1)$)

# Longest increasing subsequence

- **Problem:** Given an array $A$ of length $N$, what is the length of the longest increasing subsequence?
  - A subsequence $A_{i_1, i_2, \ldots, i_K}$ is a sequence of elements obtained by removing some number of elements (possibly zero) from $A$, but keeping the rest in order, identified by a set of included indices $i_1, i_2, \ldots, i_K$ where $1 \leq i_1 < i_2 < \cdots < i_K \leq N$
  - A subsequence $A_{i_1, i_2, \ldots, i_K}$ is *increasing* if, for every $k_1, k_2 \in \{i_1, i_2, \ldots, i_K\}$, $k_1 < k_2 \to A_{k_2} \geq A_{k_1}$.
- Can we write a recurrence for this problem?
  - Let $DP_i =$ the length of the longest increasing subsequence *ending with element $i$*
  - Our answer will be $\max_{1 \leq i \leq N} DP_i$
  - Base case: $DP_1 = 1$
  - Recursive case: $DP_i = \max_{1 \leq j < i, A_j \leq A_i}(DP_j + 1)$
- How do we code this?

# Longest increasing subsequence solution

```java
int longestIncreasingSubsequence(int[] arr) {
    int answer = 0;
    int[] dp = new int[arr.length];
    dp[0] = 1;
    for (int i = 1; i < arr.length; i++) {
        dp[i] = 0;
        for (int j = 0; j < i; j++)
            if (arr[j] <= arr[i] &&
                dp[j] + 1 > dp[i])
                dp[i] = dp[j] + 1;
        if (dp[i] > answer)
            answer = dp[i];
    }
    return answer;
}
```

- Time complexity? $O(N^2)$ (could be reduced to $O(N \log N)$)
- Space complexity? $O(N)$

# Maximum path sum

- **Problem:** Given an $M \times N$ matrix $A$ of integers, what is the maximum sum on a path from the top left of the matrix to the bottom right, where each move is either down or right?
- Can we write a recurrence for this problem?
  - Let $DP_{i,j} =$ the maximum sum on a path from the top left of the matrix to the element $A_{i,j}$
  - Our answer will be $DP_{M,N}$
  - Base case: $DP_{1,1} = A_{i,j}$
  - Recursive case:
    - If $i = 1$, then $DP_{i,j} = DP_{i,j-1} + A_{i,j}$
    - If $j = 1$, then $DP_{i,j} = DP_{i-1,j} + A_{i,j}$
    - If $i \neq 1$ and $j \neq 1$, then $DP_{i,j} = \max\{DP_{i-1,j} + 1, DP_{i,j-1} + 1\}$
- How do we code this?

# Maximum path sum solution

```
int maxPathSum(int[][] mat) {
    int M = mat.length, N = mat[0].length;
    int[][] dp = new int[M][N];
    for (int i = 0; i < M; i++)
        for (int j = 0; j < N; j++) {
            dp[i][j] = Integer.MIN_VALUE;
            if (i == 0 && j == 0)
                dp[0][0] = mat[0][0];
            if (i > 0 && dp[i - 1][j] +
                mat[i][j] > dp[i][j])
                dp[i][j] = dp[i - 1][j] + mat[i][j];
            if (j > 0 && dp[i][j - 1] +
                mat[i][j] > dp[i][j])
                dp[i][j] = dp[i][j - 1] + mat[i][j];
        }
    return dp[M - 1][N - 1];
}
```

# Maximum path sum solution

- Time complexity? $O(MN)$
- Space complexity? $O(MN)$ (could reduce to $O(\min\{M, N\})$)