• Homework 2
  • Due Friday at 2pm
• Homework 3
  • Will go live Friday at 8pm
• Midterm: Tuesday, March 8, 2016
Exercise: You run a widget factory, where a widget is made from parts A, B, and C. Each widget needs A, B, C in quantities na, nb, nc, respectively. If pa, pb, pc are the prices for purchasing a single A, B, C, you start with sa, sb, sc of each part, and you have d dollars at your disposal, compute the maximum number of widgets that can be built.
To approach the common dynamic programming problems:

1. Suggest a subproblem structure (state space) and state it precisely
2. Look for a relationship that allows you to solve “more complex” subproblems in terms of “simpler” subproblems
3. Compute the runtime of solution and memory requirements
   - Use memoization or dynamic programming we will not have to compute the value of a given state twice
4. Implement solution

Primarily used to solve optimization or counting problems
• Each problem may require a distinct approach, but some problems repeat themselves
  • When working on a list:
    • Work on a proper prefix or suffix
    • Work on a subrange
  • When working on a set, work on a proper subset
  • When working on a DAG, work on a subgraph
  • When working on a number \( n > 0 \), start with smaller numbers
  • When working on a 2D array, start with a rectangular subarray
    • Often with the same upper left corner
  • Combinations of the above. E.g., if you need to choose a \( k \) element
• Exercise: Recursively compute the sum of a given array int[] arr

• Exercise: Compute the longest increasing non-contiguous subsequence of int[] arr in O(n^2) time
• Exercise: Consider a list of numbers given in int [] arr and assume we place minus symbols between all of them. Allowing for any possible parenthesizing, compute the largest possible value for the subtraction.
• What is the state space? Compute the largest and smallest possible subtraction value over the range \([a, b]\)
• Sketch of solution (for list arr length N)
  1. Store largest and smallest value for each subrange in \(dp[N][N][2]\)
     • Initialize diagonal \(dp[i][i][\ast]\) (for both min/max) to be \(arr[i]\)
  2. Compute min and max value at each subrange \([a, b]\):
     1. Initialize \(dp[a][b][\text{min}] = \text{inf}, \ dp[a][b][\text{max}] = -\text{inf}\)
     2. Iterate over possible “split” position \(c\) (i.e., where parentheses could be added on both sides.
        o \(dp[a][b][\text{max}] = \max(dp[a][b][\text{max}], \ dp[a][c][\text{max}] – dp[c+1][b][\text{min}]\))
        o \(dp[a][b][\text{min}] = \min(dp[a][b][\text{min}], \ dp[a][c][\text{min}] – dp[c+1][b][\text{max}]\))
• Running time?
Exercise: Suppose you know that $x_0 = 1$, $x_1 = 1$ and that
$$x_n = x_{n/2} + x_{n/3} + x_{n/4} \text{ for } n \geq 2 \text{ (take the floor)}$$
Given $k$ compute $x_k \mod 1000003$. 
Exercise: Given a list of pairs \((i, j)\) stating that players \(i\) and \(j\) \((1 \leq i, j \leq 20)\) can be put on a two person team together, and assuming someone can only be on one team, compute the maximum number of teams that can be formed. You may assume you are given a symmetric adjacency matrix boolean\[\text{[]\text{[]\text{[]}}}\] \(\text{adj}\) giving the pairs.
• What is the state space? Compute maximum match on a subset of all 20 players
• Sketch of solution:
  • For each of the $2^n$ possible subset of players, determine the maximum number of teams that can be formed
    • Can be done recursively:
      • Choose a player
      • While another player is left
        • Remove both players from set $S$ to form new subset $S'$
        • $\text{teams}(S) = \text{teams}(S') + 1$
      • Return max $\text{teams}(S)$
    • Memoize
```java
int match(boolean[][] adj, int S)
{
    if (S == 0) return 0;
    if (cache[S] > -1) return cache[S];
    int smallest = 0;
    while ( ((1<<smallest) & S) == 0 ) ++smallest;
    int ret = match(adj, S^(1<<smallest));
    for (int i = 0; i < adj.length; ++i)
    {
        if (!adj[smallest][i] || ((1<<i)&S) == 0) continue;
        ret = Math.max(ret, match(adj, S^(1<<smallest)^(1<<i))+1);
    }
    return cache[S] = ret;
}
int match(boolean[][] adj) { return match(adj,(1<<adj.length)-1); }
```
Exercise: Compute the number of distinct subsets of int[] arr that sum to k. Here the length of arr is at most 100, and each element is in the interval [0, 100], and each element is distinct.
What is the state space? If |arr| = L, compute number of subsets with indices in [j, L-1] that sum to k.

Recall that each element is in [0, 100] and L <= 100, so max sum is 10,000.

Sketch of solution:

- Basis case: if j == L then return x == 0 ? 1 : 0
- Recurrence: count(j, x) = count(j+1, x) + count(j+1, x – arr[j])
Suppose you are given an array int[] arr and you want to find the maximum sum of any contiguous subsequence

Different approaches:
- Loop over all ranges [a, b] and sum each over: O(n^2)
- Compute all ranges [0, a] and then compute all [a, b] as [0, b] – [0, a-1]: O(n^2)
  - This is sort of a dynamic programming approach

Better approach?
• Build a DP array int[] dp where dp[i] contains the sum of the largest subrange that ends at index i, i.e., index i has to be used
  • dp[n+1] = arr[n+1] + max(0, dp[n])
    • i.e., if the sum of the largest subrange ending at the previous position is negative, then start a new subrange
    • Maximum subrange sum is then the maximum value in dp

• O(n) running time and O(n) space
  • Actually, you only need the previous element, so O(1) space
Exercise: You are given a rectangular array in int [][] mat. Compute the sum of each subrectangle with corners mat[0][0] and mat[r][c] and store them at sums[r][c] in another array int [][] sums. Show how you can populate the sums array quickly, and how you can use the array to compute the sum over any subrectangle of mat in constant time.
```java
int m = mat.length, n = mat[0].length;
int[][] sums = new int[m][n];
for (int i = 1; i < m; ++i) for (int j = 1; j < n; ++j)
    sums[i][j] = mat[i][j] + sums[i-1][j] + sums[i][j-1] -
                 sums[i-1][j-1];

int getRectSum(int[][] sums, int r1, int c1, int r2, int c2) {
    return sums[r2][c2] - sums[r2][c1-1] - sums[r1-1][c2] +
         sums[r1-1][c1-1];
}
```
Competitive Programming 3.5