• Homework 2
  • Due Saturday at 2am (so, basically Friday night)
• Homework 3
  • Will be posted tomorrow afternoon-ish
• Recitation
  • Eric will talk about testing (generating test cases; profiling)
• Midterm: Tuesday, March 10, 2015
• Next week’s office hours:
  • Monday morning from 9am to 10am
  • Wednesday office hour canceled
Given a list of \( n \leq 30 \) signed integers and a 64-bit integer \( L \), determine if some sublist (not necessarily contiguous) sums up to the \( L \).

- Split list in half, compute all possible sums that can occur in each half
- For one subset, sort (or store in hashtable)
- For each element \( v \) in the other subset, search (in the sorted[hashed subset]) for value equal to \( L - v \)
- \( O(2^{n/2} \log(2^{n/2})) = O(n2^{n/2}) \)
Exercise: Suppose you have access to a function `double f(double x)` taking inputs from the interval `[0, 100]`. It is given that `f` (which may be discontinuous) is increasing on `[0, c]` and decreasing on `[c, 100]` for some unknown `c \in [0, 100]`. Find `c` accurate to 6 digits after the decimal point.
**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 \(dp[a][b][\min] = \infty\), \(dp[a][b][\max] = -\infty\)
     2. Iterate over possible “split” position \(c\) (i.e., where parentheses could be added on both sides.
        o \(dp[a][b][\max] = \max(dp[a][b][\max], dp[a][c][\max] - dp[c+1][b][\min])\)
        o \(dp[a][b][\min] = \min(dp[a][b][\min], dp[a][c][\min] - dp[c+1][b][\max])\)

• **Running time?**
Exercise: Suppose you know that $x_0 = 1$, $x_1 = 1$ and that
\[ x_n = \lfloor x_{n/2} \rfloor + \lfloor x_{n/3} \rfloor + \lfloor x_{n/4} \rfloor \text{ for } n \geq 2 \text{ (take the floor)} \]
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\[][]\ 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;
}
```

```java
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
Competitive Programming 3.5

We’ll talk more about dynamic programming next class