Lecture 8

Solving Dynamic Programming Problems

For the common dynamic programming problems, there is a systematic way of searching for a solution that consists of repeating the following steps:

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 such a solution and the memory requirements (based on number of states, number of dependencies, work per state). By using memoization (top-down), or dynamic programming (bottom-up) we will not have to compute the value of a given state twice.
4. Implement the solution. Determine if you just need a single result, or the subset/subsequence/whatever that gives the result as well.

Finding a DP solution consists of repeating the above steps until you succeed or give-up on dynamic programming. To help you execute the above procedure, we will discuss and practice each of the above steps several times.

Finding Subproblems

Each problem may have a distinct method for solving it, but the methods for finding subproblems often repeat themselves. Some common ones are:

1. When working on a list:
   (a) Work on a proper prefix or suffix: Remove elements from the head or tail.
   (b) Work on a subrange: Pick a division point, and divide the list into subranges.
2. When working on a set you can work on a proper subset: Remove elements.
3. When working on a DAG (directed acyclic graph) you can work on a subgraph: Choose a neighbor, and work on nodes reachable from neighbor.
4. When working on a number $n > 0$, you can work on smaller numbers.
5. When working on a 2d array, you can work on a rectangular subarray (often the subarray will have top left corner in the first row and first column).

6. Combinations of the types of states above: Effectively a cartesian product of the states. For instance, if you have to choose a k element subsequence from a list, your state space may be
$$\{\text{Possible suffixes}\} \times \{\text{Number of elements to be chosen}\}.$$

Exercises:

For each of the following exercises, solve the problem by precisely stating the subproblem, and show how the subproblems can be used to compute the full solution. Give the runtime of your solution.

1. Compute the sum of a given array **int**[] arr.
2. Compute the longest increasing subsequence of **int**[] arr in $O(n^2)$ time.
3. 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.
4. Suppose you know that $x_0 = 1$, $x_1 = 1$ and that $x_n = x_{\lfloor n/2 \rfloor} + x_{\lfloor n/3 \rfloor} + x_{\lfloor n/4 \rfloor}$ for $n \geq 2$. Given k compute x_k mod 1000003.
5. 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.
6. 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.

Solutions:

1. Subproblem: Assume the array has length L. Compute the sum over the indices $[k, L-1]$. Runtime is linear (memoization unnecessary).

```
int sum(int[] arr, int k)
{
  return k == arr.length ? 0 : arr[k] + sum(arr,k+1);
}
int sum(int[] arr) { return sum(arr,0); }
```

2. Subproblem: Assume the array has length L. Compute the longest increasing subsequence over the indices $[k, L-1]$ such that every entry is greater than arr[p] for some $p < k$. There are $O(n^2)$ states and constant work is done per state giving a runtime of $O(n^2)$.

```
//p = -1 denotes no constraint on the values
int LIS(int[] arr, int k, int p)
{
  if (k == arr.length) return 0;
  if (cache[k][p+1] > -1) return cache[k][p+1];
  int ret = LIS(arr,k+1,p);
  if (p == -1 || arr[k] > arr[p])
      ret = Math.max(ret, LIS(arr,k+1,k)+1);
  return cache[k][p+1] = ret;
}
int LIS(int[] arr) { return LIS(arr,0,-1); }
```

We will see a faster, and trickier DP solution later.

3. Subproblem: Compute the largest/smallest possible subtraction value over the indices $[a, b]$. There are $O(n^2)$ states and we perform $O(n)$ work per state giving $O(n^3)$ runtime.

```
static int sub(int[] arr, int a, int b, int isMax)
{
  if (a == b) return arr[a];
  if (cache[a][b][isMax] > Integer.MIN_VALUE) return
     cache[a][b][isMax];
  int best = isMax == 1 ? Integer.MIN_VALUE :
     Integer.MAX_VALUE;
  for (int c = a; c < b; ++c)
  {
      if (isMax == 1) best = Math.max(best,
         sub(arr,a,c,1)-sub(arr,c+1,b,0));
      else best = Math.min(best,
         sub(arr,a,c,0)-sub(arr,c+1,b,1));
  }
  return cache[a][b][isMax] = best;
}
static int sub(int[] arr) { return sub(arr,0,arr.length-1,1); }
```

Here is a DP solution.

```
static int sub(int[] arr)
{
  int n = arr.length;
  int[][][] dp = new int[n][n][2];
  for (int i = 0; i < n; ++i) for (int j = 0; j < 2; ++j)
     dp[i][i][j] = arr[i];
  for (int L = 2; L <= n; ++L)
  {
    for (int a = 0; a + L <= n; ++a)
    {
            for (int j = 0; j < 2; ++j)
            {
                int b = a+L-1;
                dp[a][b][j] = j == 1 ? Integer.MIN_VALUE :
                    Integer.MAX_VALUE;
                for (int c = a; c < b; ++c)
                {
                    if (j == 1) dp[a][b][j] =
                        Math.max(dp[a][b][j],dp[a][c][1]-dp[c+1][b][0]);
                    else dp[a][b][j] =
                        Math.min(dp[a][b][j],dp[a][c][0]-dp[c+1][b][1]);
                }
            }
        }
    }
    return dp[0][arr.length-1][1];
}
```

4. Subproblem: Compute x_j for a given value $0 \leq j \leq k$. With a naive cache or DP the runtime will be $O(k)$. With a hash map for the cache, the runtime should be $O((\lg n)^2)$ (haven't tried to prove this yet). The function decreases so quickly that no cache is faster than any cache, but this is designed to show you the space saving efficiency.

```
static HashMap<Integer,Integer> cache = new
    HashMap<Integer,Integer>();
int xval(int j)
{
    if (j <= 1) return 1;
    Integer val = cache.get(j);
    if (val != null) return val;
    int ret = (xval(j/2)+xval(j/3)+xval(j/4)) % 1000003;
    cache.put(j,ret);
    return ret;
}
```

5. Subproblem: Compute the maximum matching on a subset S of $\{1, \ldots, 20\}$. For $n = 20$ there are $O(2^n)$ states and we perform $O(n)$ work per state giving a runtime of $O(n2^n)$.

```
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); }
```

6. Subproblem: Assume the length of the array is L. Compute the number of subsets with indices in $[j, L-1]$ that sum to x. If the range is $[0, m]$ there are $O(L^2m)$ states each taking constant time. Thus the runtime is $O(L^2m)$.

```
  static int count(int[] arr, int j, int x)
  {
    if (j == arr.length) return x == 0 ? 1 : 0;
    if (x < 0) return 0;
    if (cache[j][x] > -1) return cache[j][x];
    return cache[j][x] = count(arr,j+1,x) +
       count(arr,j+1,x-arr[j]);
  }
  static int count(int[] arr, int x) { return count(arr,0,x); }
```

Here is a space-saving DP solution.

```
  static int count(int[] arr, int x)
  {
    int[] dp = new int[100*100+1];
    dp[0] = 1;
    for (int j = 0; j < arr.length; ++j)
        for (int k = dp.length-arr[j]-1; k >= 0; --k)
            dp[k+arr[j]]+=dp[k];
    return dp[x];
  }
```

Here we iterate in reverse to prevent us from reusing an element.

Maximum Sum Range Queries

Consider the list

$$L = [1, 3, -2, 1, -2, 1, -3, 2, 4, 1]$$

when considering the following approaches. After practicing on the exercises, let's solve a problem together. Suppose you are given an array **int**[] arr and you want to compute the maximum sum of any contiguous subsequence (i.e., max range sum). One method would be to loop through all possible ranges $[a, b]$, and then sum over each. This would require $O(n^2)$.

If instead we used DP to precompute the sums of the ranges $[0, a]$, we could then compute the sum of any range in $O(1)$ time as the sum over $[a, b]$ is the sum over $[0, b]$ minus the sum over $[0, a-1]$ (this is the static version of what the Fenwick tree does for a dynamic list). This also gives $O(n^2)$. For an even better DP solution, we could build a DP array **int**[] dp such that dp[i] contains the sum of the largest subrange with highest index i. That is, the index i has to be used. Note that we have:

```
dp[n+1] = arr[n+1] + max(0,dp[n]).
```

The maximum size subrange is then the maximum element of dp. Thus we have a linear algorithm that requires linear space. Interestingly, we only ever need the immediately previous element of the dp array at each step of the algorithm. Thus we can do the following:

```
int max = arr[0], curr = 0;
for (int i = 0; i < arr.length; ++i)
{
  curr += arr[i];
  max = Math.max(max, curr);
  curr = Math.max(curr,0);
}
```

Here curr serves as the max of the previous element of the dp array with 0, and max stores the maximum of all previous dp entries. This gives us a linear algorithm (called Kadane's algorithm) with $O(1)$ number of variables (technically $O(\lg n)$ space). This also illustrates that some DP algorithms (typically not memoization) can be optimized to use less space by recognizing redundancy.

Exercises:

1. 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.

2. Show how to compute the maximum subrectangle sum of the $n \times n$ **int**[][] mat in $O(n^4)$ time.

3. Show how to compute the maximum subrectangle sum of the $n \times n$ **int**[][] mat in $O(n^3)$ time.

4. Given an array **int**[][] mat of zeros and ones, compute the size of the largest rectangle (in area) that is entirely composed of ones.

Solutions:

1. The following code populates sums in $O(n^2)$ time:

```
int m = mat.length, n = mat[0].length;
int[][] sums = new int[m][n];
for (int i = 0; i < m; ++i) for (int j = 0; j < n; ++j)
{
    sums[i][j] = mat[i][j];
    if (i > 0) sums[i][j] += sums[i-1][j];
    if (j > 0) sums[i][j] += sums[i][j-1];
    if (i > 0 && j > 0) sums[i][j] -= sums[i-1][j-1];
}
```

The following code gets the sum over a rectangle with upper left corner $(r1, c1)$ and lower right corner $(r2, c2)$:

```
int getRectSum(int[][] sums, int r1, int c1, int r2, int c2)
{
  int ret = sums[r2][c2];
  if (r1 > 0) ret -= sum[r1-1][c2];
  if (c1 > 0) ret -= sums[r2][c1-1];
  if (r1 > 0 && c1 > 0) ret += sums[r1-1][c1-1];
  return ret;
}
```

The code can be made easier if we offset everything by 1 (i.e., assume the first row and column of mat are ignored):

```
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];
}
```

By zero-padding the arrays, all of the if-statements disappear. Zero-padding is a common technique.

2. Simply loop over all possible upper-left and lower-right rectangle corners, and call getRectSum on each.

3. Here we mix Kadane's algorithm with the precomputation of row sums. First, for each row separately, we precompute the sums of all prefixes. This allows us to compute the sum of a subrange of any row in constant time. Next, we loop over all column ranges $[c1, c2]$. Once we fix a column range, we have converted the problem into one that

can be solved by Kadane's algorithm. There are $O(n^2)$ column ranges, and Kadane requires $O(n)$ time giving $O(n^3)$.

4. This is the max rectangle sum where 0's are treated as $-\infty$.