Dynamic Programming

Week 11

Required Activities

- Check Announcements regularly (every 2-3 days)
- Read FA book: Chapter 3
- Continue working on **assignment 4 due Jan 25**

Dynamic Programming

- Similar to the divide-and-conquer approach where an instance of a problem is divided into smaller instances
- Solve the small instances first, store the results, and look them up when we need them instead of re-computing them (remember Fibonacci Sequence Improved algorithm)
- Bottom-up approach as the solution is constructed from the bottom up in the array
- Two steps:
  - establish the recursive property that gives the solution
  - solve an instance of the problem in a bottom-up fashion by solving smaller instances first

Divide & Conquer vs Dynamic Programming

Divide and Conquer	**Dynamic Programming**
Top-down approach to problem solving	Bottom-up approach to problem solving
Blindly divide problem into smaller instances and solve the smaller instances	Instance of problem divided into smaller instances
Technique works efficiently for problems where smaller instances are unrelated	Smaller instances solved first and stored for later use by solution to solve larger instances
Inefficient solution to problems where smaller instances are related	Look it up instead of re-compute
e.g. recursive solution to the Fibonacci sequence	e.g. iterative solution to the Fibonacci Sequence

When to use Dynamic Programming

- Often used for optimization
- Two main properties of a problem that suggest it can be solved using Dynamic Programming:
  1. Overlapping sub problems – recursive solution contains small and distinct sup problems repeated many times (in Fibonacci Sequence keep calculating same sequences over and over)
  2. Optimal substructure – optimal solution to the problem contains optimal solution to sub problems
- Typical algorithm examples:
  - String algorithms such as longest common sequence, longest increasing subsequence, etc.
  - Algorithms on graphs for efficiency such as shortest distance
  - Traveling salesman
  - Chained matrix multiplication

Factorial

- n! = product of integers between 1 and n
- Recursive definition: n! = n * (n-1)! where 1!=1 and 0!=1

```
int f(int n) {
    if (n==1)      return 1;
    else if (n==0) return 1;
    else // recursive
                   return n * f(n-1);
}
```

  - Recurrence: T(n) = n * T(n-1) So time complexity = O(n) and space complexity O(n) because recursion needs stack of size n
- Using Dynamic Programming using array to save calculations fact[n]:

```
int f(int n) {
    if (n==1)      return 1;
    else if (n==0) return 1;
    else if (fact[n] != 0) // already calculated
                   return fact[n];
    else           return fact[n] = n * f(n-1);
}
```

  - Eliminate overhead of repeated calls so space complexity O(1) but time complexity while improved is still O(n)

Floyd's Algorithm for Shortest Path

- **Optimization problem** - finding the best solution from all feasible solutions because there can be more than one solution
- Problem is represented as weighted graph and looking for **shortest path** from each vertex to all the other vertices
- The shortest path must be a **simple path** - a path that does not pass through the same vertex twice
- As there may be more than one simple path select the one with the **minimum value** based on length of path
- **All pairs** approach – shortest path between any pair of nodes in the graph
- e.g. Finding shortest air connections when must fly from one city to another when a direct flight does not exist

Floyd's Algorithm

- Given V={1, 2, …, n}, C[i][j] represent weight (cost) between node i and j (edge weight). C[i][j]=-1 or INF means no path between the two nodes. Then we compute matrix A[1..n][1..n] which gives the shortest path between two nodes.
  - Initialize the solution matrix same as input graph matrix (not consider intermediate vertex)

    Loop to initialize => A[1..n][1..n] = C[1..n][1..n]
  - Then adds intermediate vertex, shortest path (i, j, k) for all (i, j) pairs for k=1, k=2, until k=n and we have found the shortest path

    Loop for all k and => if (A[i][k] + A[k][j] < A[i][j]) // shorter

    A[i][j] = A[i][k] + A[k][j];

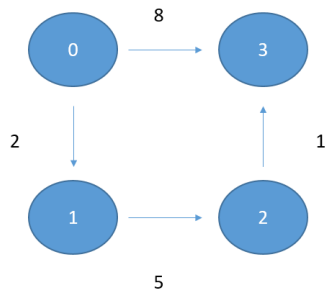
Floyd Pseudo Code

```
floydAlgorithm(int C[][], int A[][], int V)
{
    for (i = 0; i < V; i++)
        for (j = 0; j < V; j++)
            A[i][j] = C[i][j];

    for (k = 0; k < V; k++) {
        for (i = 0; i < V; i++)  {
            for (j = 0; j < V; j++)   {
                if (A[i][k] + A[k][j] < A[i][j])
                        A[i][j] = A[i][k] + A[k][j];
            }
        }
    }
}
```

Floyd Execution

- Given directed graph with 4 vertices:




```
C[][] = { {0,   2,  INF, 8},
          {INF,  0,  5,  INF},
          {INF, INF, 0,   1},
          {INF, INF, INF, 0} }
```

k=0

```
0   2   INF 8
INF 0   5   INF
INF INF 0   1
INF INF INF 0
```

k=1

```
0   2   7   8
INF 0   5   INF
INF INF 0   1
INF INF INF 0
```

k=2

```
0   2   7   8
INF 0   5   6
INF INF 0   1
INF INF INF 0
```

k=3

```
0   2   7   8
INF 0   5   6
INF INF 0   1
INF INF INF 0
```

So from vertex 0 to vertex 1 shortest path is 2 (0 **2** 7 8)
from vertex 0 to vertex 2 shortest path is 7 (0 2 **7** 8)
from vertex 0 to vertex 3 shortest path is 8 (0 2 7 **8**)
from vertex 1 to vertex 2 shortest path is 5 (INF 0 **5** 6)
etc.

Optimal Binary Search Tree

- **binary search tree –** keys in left sub-tree are less than or equal and in right sub-tree are greater
- **search time** – number of comparisons done by procedure search to locate a key
- **optimal binary search tree:**
  - keys are organized to **minimize time** to locate
  - all the keys have the **same probability** of being the search key
- Our goal is to determine a tree for which the average search time is minimal
- Can be implemented as recursive and using DP (storing cost of sub-problems in array)

Questions ?

- Post in the discussions
- Send email to RMcFadden@HarrisburgU.edu
- Respond usually within 48hours