CSC373 Week 3 Notes

Dynamic Programming (DP):

1. Introduction:
   - Break the problem down into simpler subproblems, solve each subproblem just once and store their solns. Then, the next time the same subproblem occurs, we can just use the result we got instead of re-computing it. This is called memoization.

   - With DP, you save a lot of computation at the expense of a modest increase in storage space.

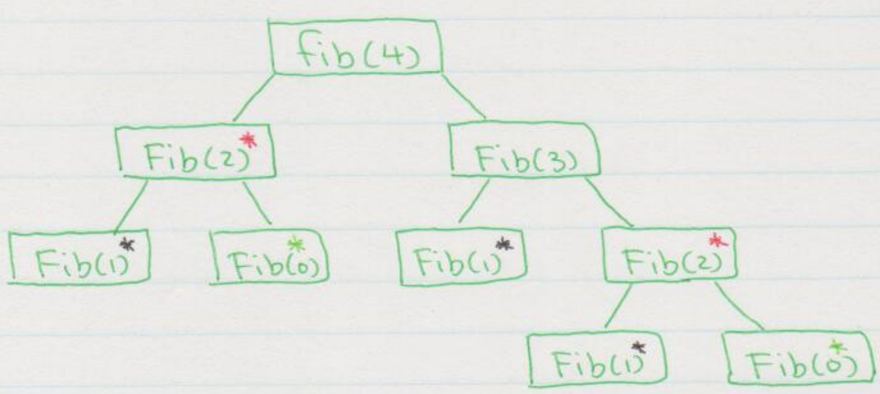
2. Examples:

   a) Fibonacci Numbers:

Here is a soln that doesn't use DP.

```
def fib(n):
    if (n ≤ 1):
        return n
    return fib(n-1) + fib(n-2)
```

Consider the steps of solving for n=4 with this code.



Notice how some terms are repeated many times.

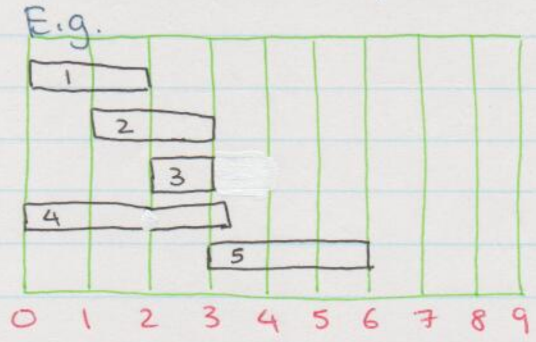
Here's a soln that use DP:

```
def fib(n):
    vals = [0,0]

    for i in range(2, n+1):
        vals.append(vals[i-1] + vals[i-2])

    return vals[n]
```

Here, we're reusing the prev computations we did to find new ones.

b) Weighted Interval Scheduling:
- Problem: Job j starts at time s_j and finishes at time f_j and has a weight of w_j. 2 jobs are compatible if they don't overlap. We want to find a set S of mutually compatible jobs with the highest total weight.

- DP Soln:
  - We know that the jobs are sorted by their finish time: $f_1 \le f_2 \le \ldots \le f_n$.
  - Let $p[j]$ be the i, $i<j$, s.t. job i is compatible with job j. I.e. $f_i < s_j$.
  - $p[j]$ can be computed via binary search.
  - E.g.



$P[5] = 3$
$P[4] = 0$
$P[3] = 1$
$P[2] = 0$
$P[1] = 0$

- Let Opt be an optimal soln.
  Then, we have 2 cases regarding job n:
  1. Job n is in Opt.
     Then, we know that jobs
     $p[n]+1, \ldots, n-1$ are incompatible.
     Hence we must select from jobs
     $1, \ldots, p[n]$.

  2. Job n is not in opt.
     Then, we must select the optimal
     subset of jobs from $\{1, \ldots, n-1\}$.

- Let $OPT(j)$ = max total weight of
  all compatible jobs from 1 to j.

- Base Case: $OPT(0) = 0$
  Note: You can also do $OPT(1) = w_1$

- Consider job j:
  1. Job j is selected. optimal weight $= OPT(p[j]) + w_j$
  2. Job j is not selected. optimal weight $= OPT(j-1)$

- Hence, we can write this:

$$OPT(j) = \begin{cases} 0 & , \text{if } j=0 \\ \max(OPT(j-1), w_j + OPT(p[j])), & \text{if } j>0 \end{cases}$$

- Brute Force Soln:

```
def brute-force(n, s_1, ..., s_n,
                f_1, ..., f_n,
                w_1, ..., w_n):

    Sort the jobs by their finish time.
    Compute p[1], ..., p[n] via binary search.

    return compute-opt (n)

    def compute-opt(n):

        if n=0:
            return 0
        return max {compute-opt(j-1),
                    w_j + compute-opt
                          (p[j])
                   }
```

The runtime complexity of this is $O(2^n)$. This is bc we have to repeat calculations many times.

- DP Soln:

```
def dp-soln(n, s_1, ..., s_n, f_1, ..., f_n, w_1, ..., w_n):
    Sort the jobs by finish time
    Compute p[1], ... p[n] via binary search
    M[0]=0
    return compute_opt_new(n)

    def compute_opt_new(n):
        if(!M[n]):
            M[n]= max (...)
        return M[n]
```

This new algo takes $O(n \lg n)$.
- Sorting by fin time: $O(n \lg n)$
- Computing $p[j]$'s : $O(n \lg n)$
- At most n calls are made to compute_opt_new but each call takes $O(1)$, so together it takes: $O(n)$

c) Knapsack Problem:
- Problem: There are n items. Item i provides value $v_i > 0$ and has weight $w_i > 0$. We have a knapsack with capacity W.

  We want to pack the knapsack with a subset of the items with the highest total value s.t. their combined weight does not exceed W.

- DP Approach:
  - We'll create a table to keep track of total weights and values.

$v \backslash w$	0						W
0							
⋮							
⋮							
n							

Table T

The entry $T[i][w]$ will store the max combined values of any subset of items $\{1, \ldots, i\}$ of combined size at most w.

- Consider item i:
  1. If $w_i > w$, then we can't choose it. We have to use $T[i-1][w]$.

  2. If $w_i \le w$, then:
     a) If we choose i, then we have $v_i + T[i-1][w-w_i]$

     b) If we don't choose i, then we have $T[i-1][w]$.

  Therefore, we get this:

$$T[i][w] = \begin{cases} 0 & \text{, if } i=0 \\ T[i-1][w] & \text{, if } w_i > w \\ \max(T[i-1][w],\ v_i + T[i-1][w-w_i]) & \text{, if } w_i \le w \end{cases}$$

- Time Complexity:
  - The total running time is $O(n \cdot W)$.
  - This is pseudo-polynomial.

- A similar problem: We'll do a similar problem. This time, we want to find the min capacity needed to pack a total value of at least V.

  We can set up a similar table like we did in the prev question. Let's call this table T as well.

- Consider item i:
  1. If we choose it, we need capacity $w_i + T[i-1][v-v_i]$
  2. If we don't choose it, we need capacity $T[i-1, v]$.

$$T[i][v] = \begin{cases} 0, & \text{if } v \leq 0 \\ \infty, & \text{if } v > 0 \text{ and } i \leq 0 \\ \min(w_i + T[i-1][v-v_i], \; T[i-1, v]), & \text{if } v > 0 \text{ and } i > 0 \end{cases}$$

d) Shortest Path:
- Terminology:
  - Graph: A set of nodes/vertices and edges.
  - Directed Graph: A graph where the direction of flow on edges is specified.
  - Cycle: A path that takes you to a previously seen node.
  - Path: A consecutive set of edges going from node A to node B.
- Problem: Given a directed graph $G = (V, E)$ with edge lengths l_{vw} on each edge (v, w) and a starting node s, compute the length of the shortest paths from s to every vertex t.

Note: When all edges have an edge length that is non-negative (≥ 0), we can use Dijkstra's Algorithm.

Note: In our case, we can have negative lengths, but we have to ensure that no cycles can be negative. Otherwise, you can loop in the infinitely to decrease the length.

- DP Soln:
  - Claim: Suppose $s \to u \to t$ is the shortest path to t from s. This implies that $s \to u$ is the shortest path from s to u.
  - $d(s,t) = \min(d(s,u) + \ell_{ut})$

    I.e. The shortest length from s to t is equal to the shortest length from s to u plus the length from u to t.

e) All-pairs Shortest Path:
- Problem: Given a directed graph $G = (V, E)$ with edge lengths ℓ_{vw} on each edge (v, w) and no negative cycles, compute the lengths of the shortest path from all vertices s to all other vertices t.

- DP Solution:
  - Consider this function shortestPath(i,j,k) where i is the start node, j is the end node and k is some other node s.t. we return the shortest path btwn i and j using vertices only from {1 ... k}.
  - Given this function, we know that

```
shortestPath(i, j, k) =
min
(
   shortestPath(i, j, k-1),      1
   ShortestPath(i, k, k-1) +   } 2
   shortestPath(k, j, k-1)
)
```

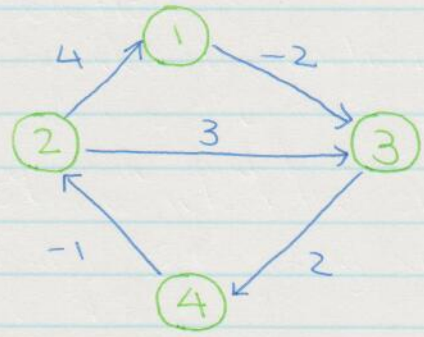
1: Here, the path btwn i and j doesn't go through vertex k.

2: Here, we go from i to k first, and then k to j.
Both times, the intermediate nodes are in {1, ..., k-1}.

- We'll apply this algo for vertices 1 to N as the k values.
  I.e. We'll find the shortest paths s.t. the node 1 is used. Then, we'll find the shortest paths that include 1 or 2. And so on.

Note:
This algo is called Floyd-Warshall.

- E.g.



k=0 (At the beginning)

j

i	1	2	3	4
1	0	∞	-2	∞
2	4	0	3	∞
3	∞	∞	0	2
4	∞	-1	∞	0

k=1 (Using only node 1 as intermediate node(s) in your paths)

	1	2	3	4
1	0	∞	-2	∞
2	4	0	2	∞
3	∞	∞	0	2
4	∞	-1	∞	0

Since the only path that only goes through node 1 is 2→1→3, we see if 2→1→3 is shorter than 2→3. It is, so we update the length.

k=2 (Using only 1 or 2 as intermediate nodes)

	1	2	3	4
1	0	∞	-2	∞
2	4	0	2	∞
3	∞	∞	0	2
4	3	-1	1	0

k=3

	1	2	3	4
1	0	∞	-2	0
2	4	0	2	4
3	∞	∞	0	2
4	3	-1	1	0

k = 4

	1	2	3	4
1	0	-1	-2	0
2	4	0	2	4
3	5	1	0	2
4	3	-1	1	0

← Final result

e) Chain Matrix Product:

- Problem: Given matrices $M_1, M_2, \ldots, M_n$ where the dimension of M_i is $d_{i-1} \times d_i$, compute $M_1 \cdot M_2 \cdot \ldots \cdot M_n$.

- DP Soln:
  - Recall: Matrix multiplication is associative.
    $(A \cdot B) \cdot C = A \cdot (B \cdot C)$
  - Suppose we're doing $A \cdot B \cdot C \cdot D$. We have to figure out which of the following is the cheapest:
    (A)(BCD), (AB)(CD), (ABC)(D)
  - Can use DP to store these info and build up.
  - $O(n^3)$ time complexity.

f) Edit Distance:

- Problem: Given 2 strings $X = x_1, \ldots, x_m$ and $y = y_1, \ldots, y_n$, and suppose we can delete or replace symbols in either string, how many deletions and replacements does it take to match the 2 strings?

Suppose the cost to delete symbol a is $d(a)$
Suppose the cost to replace symbol a with symbol b is $r(a,b)$ and that $\forall a,b \; r(a,b) = r(b,a)$ and $r(a,a) = 0$.

- DP Solution:
  - Consider the last symbol of each string, x_m and y_n. There are 3 options:
    1. Delete x_m and optimally match $x_1, \ldots, x_{m-1}$ and $y_1, \ldots, y_n$.
    2. Delete y_n and optimally match $x_1, \ldots, x_m$ and $y_1, \ldots, y_{n-1}$.
    3. Match x_m and y_n and optimally match $x_1, \ldots, x_{m-1}$ and $y_1, \ldots, y_{n-1}$.
  - Let $E[i,j]$ be the edit distance between $x_1, \ldots, x_i$ and $y_1, \ldots, y_j$

$$E[i,j] = \begin{cases} 0 & \text{, if } i=j=0 \\ B & \text{, if } i=0 \wedge j>0 \\ A & \text{, if } i>0 \wedge j=0 \\ \min(A,B,C), & \text{otherwise} \end{cases}$$

$A = d(x_i) + E[i-1,j]$
$B = d(y_j) + E[i,j-1]$ $\quad C = r(x_i, y_j) + E[i-1,j-1]$

- $O(n \cdot m)$ time complexity
  $O(n \cdot m)$ space complexity

g) Travelling Salesman Problem:

- Problem: Given a directed graph $G = (V, E)$ where $d_{i,j}$ is the distance from node i to node j, find the min dist which needs to be travelled starting from node v, visiting every other node exactly once and coming back to v.

- DP Solution:
  - Suppose we start at vertex 1 and end on vertex c.

  - Our soln is:

    $$\min_{c \in \{2, \dots, n\}} \left(OPT(S, c) + d_{c,1} \right)$$

    where $S = \{2, \dots, n\}$ and $OPT(S, c)$ is the min total dist starting at node 1, visiting each node in S exactly once and ending at node $c \in S$.

  - $O(n \cdot 2^n)$ calls $\times$ $O(n)$ time per call, so the time complexity is $O(n^2 \cdot 2^n)$