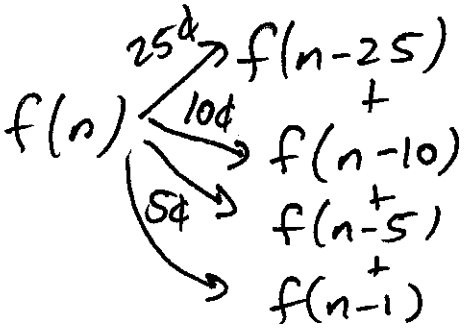
Dynamic Programming

(1) Change Problem

(2) Floyd Warshall's ALG

→ 1¢, 5¢, 10¢, 25¢

how many combinations are there to make change for n cents?



b.c.

$f(0) = 1$

$f(x) = 0$ if $x < 0$.

$n = 6$ [6 Ps, 1N 1P] (2 ways)

$f(6) \to f(1) \to f(0) \to 1$ (5¢, 1¢)

$f(6) \to f(5) \to f(4) \to f(3) \to f(2) \to f(1) \to f(0)$ (1¢, 1¢ .. 1¢)

$f(5) \to f(0) \to 1$ (1¢, 5¢)

Problem - this is counting <u>permutations</u> of the ways to provide change.

Idea 1 - ~~divide~~ final ans by something

Idea 2 - reformulate our problem recursively

→ each combo isn't counted the same # of times!

Can we reformulate our problem to "force" one ordering - highest to lowest denomination?

$f(n) \xrightarrow[10]{} \boxed{f(n-10)} \xrightarrow[25]{} f(n-35)$

has no idea what coin was previously given!

In general, if a function needs some info but doesn't have it, often times we <u>ADD</u> a parameter to the function.

$f(n, coin)$ = # of ways to make change for n cents using coin c of a smaller coin.

Coins are #s 0, 1, 2, c-1
Values of coins value[0], value[1], ... value[c-1]
$<$ $<$ $<$

$f(n, k)$ { prev coin given / max coin allowed to give.

```
int res = 0;
for (int i = 0; i <= k; i++)
    res += f(n - value[i], i)

return res;
}
```

Recursively - too slow, must memoize or use DP.

Alternate recursive sol

```
f(n, k) {
    int res = 0;
    res += f(n, k-1) // No coins den k
    res += f(n - value[k], k) // All combos using 1 or more coin of den k.
    return res;
}
```

To store use an array

Size $(n+1) \times C$ ← # coins

Den: 1, 3, 5, 9, 13

n = 15

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
3	1	1	1	2	2	2	3	3	3	4	4	4	5	5	5	6
5	1	1	1	2	2	3	4	4	5	6	7	8	9	10	11	13
9	1	1	1	2	2	3	4	4	5	7	8	9	11	12	14	17
13	1	1	1	2	2	3	4	4	5	7	8	9	11	13	15	18

$dp[n][k] = dp[n][k-1] + dp[n - value[k]][k]$

↳ Watch out for base cases and array out of bounds.

Floyd-Warshall's

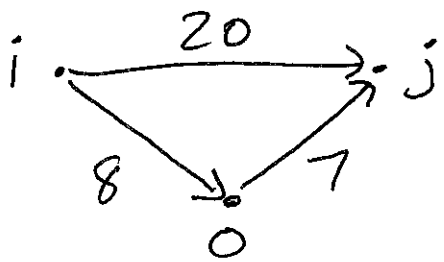
All Pairs Shortest Path Problem on a directed weighted graph.

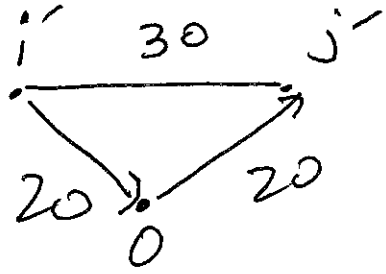
Works w/ neg. edge weights.

(1) Initially all paths are direct

(2) Main Algorithm

- iteratively allow new intersections (vertices) to be traversed as stopping pts/intermediate pts on longer path





Does it help me to use intersection 0?

```
adj[i][j] = Math.min(adj[i][j],
                     adj[i][0]+
                     adj[0][j]);
```

Does it help to use intersection 1?

We want to iterate through all possible intersections, 1 by 1, allowing them to be intermediate intersections in paths.

In code

k is intermediate

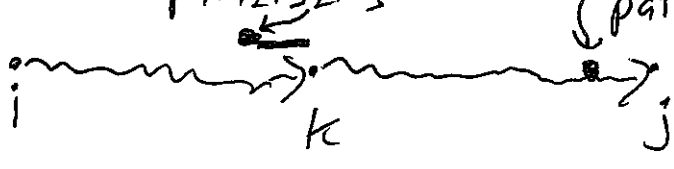
```
for (k=0; k<n; k++) {
    for (i=0; i<n; i++) {
        for (j=0; j<n; j++) {
            if (adj[i][k]+adj[k][j]<adj[i][j])
                adj[i][j] = adj[i][k]+adj[k][j];
                path[i][j] = path[k][j];
        }
    }
}
```

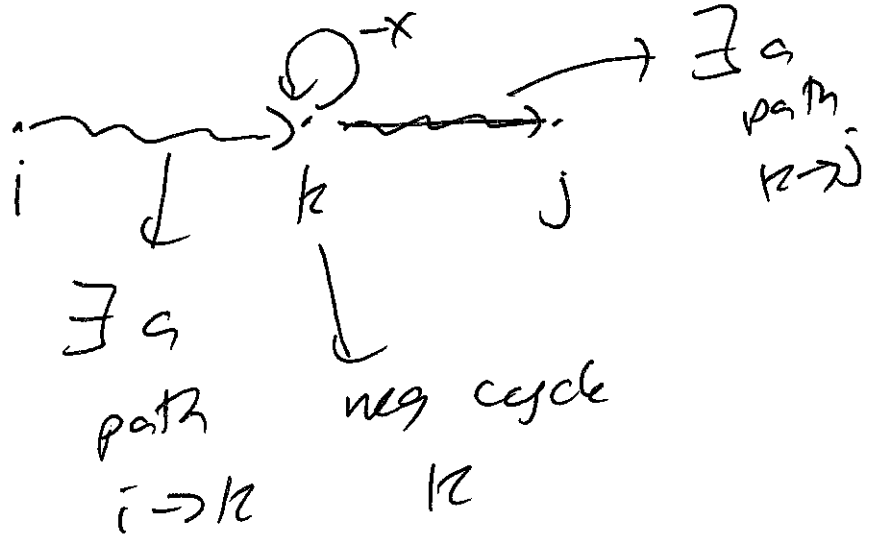
↳ the last vertex visited on the shortest path from i to j.

---

adj[i][k] = shortest path from i to k going through some subset of vertices {0, 1, 2, .. k-1}.

adj[k][j] = shortest path from k to j going through some subset of vertices {0, 1, .. k-1}.





There is no shortest path from i to j if there is some vertex k such that a path from i to k exists AND adj[k][k] < 0 AND a path from k to j exists.