All-Pairs Shortest Paths

Lots of apps let you plan your trips using real-time bus, train, and traffic data. They try to predict which route will be faster, but aren't always right.

Instead of just planning, my new app lets you send "ghost" versions of you along different routes, simulating their travel using the real-time data. That way, you can see which route turned out to be faster in practice.

That way, you can also race your past selves.

Soon...

Ugh, lost to the bike ghost again.

Hey, my key won't work. I'm sorry, but we've decided to replace you. This floaty guy is much more punctual.

Our new dad never misses our games!

Noo!
TL;DR dynamic programming

What are two ways you can compute the Fibonacci numbers?

\[ F_n = F_{n-1} + F_{n-2} \]

with \( F_0 = 0, \ F_1 = 1 \)

Which way is better?
TL;DR dynamic programming

One way, simply use the definition

Recursive:

F(n):
    if(n==1 or n==0)
        return n
    else return F(n-1)+F(n-2)
TL;DR dynamic programming

Another way, compute $F(2)$, then $F(3)$ ...
... until you get to $F(n)$

Bottom up:

$$A[0] = 0$$
$$A[1] = 1$$

for $i = 2$ to $n$

TL;DR dynamic programming

This second way is much faster

It turns out you can take pretty much any recursion and solve it this way (called “dynamic programming”)

It can use a bit more memory, but much faster
How many multiplication operations does it take to compute:

\[ x^4? \]

\[ x^{10}? \]
How many multiplication operations does it take to compute:

$x^4$? Answer: 2

$x^{10}$? Answer: 4
Can compute $x^4$ with 2 operations:

$x^2 = x \times x$ (store this value)

$x^4 = x^2 \times x^2$

Save CPU by using more memory!

Can compute $x^n$ using $O(\lg n)$ ops

Also true if $x$ is a matrix
Shortest paths using matrices

Any sub-path \((p_{x,y})\) of a shortest path \((p_{u,v})\) is also a shortest path.

Thus we can recursively define a shortest path \(p_{0,k} = <v_0, ..., v_k>\), as:

\[
w(p_{0,k}) = \min_{k-1}(w(p_{0,k-1}) + w("k-1", k))
\]
Shortest paths using matrices

Thus a shortest path (using less than \( m \) edges) can be defined as:

\[
L^m_{i,j} = \min_k (L^{m-1}_{i,k} + L^1_{k,j}),
\]

where \( L^1 \) is the edge weights matrix.

Can use dynamic programming to find an efficient solution.
Shortest paths using matrices

$L^m$ is not the $m^{th}$ power of $L$, but the operations are very similar:

$$L^m = l^m_{i,j} = \min_k (l^{m-1}_{i,k} + l^1_{k,j}) \quad \text{// ours}$$

$$L^m = l^m_{i,j} = \sum_k (l^{m-1}_{i,k} \ast l^1_{k,j}) \quad \text{// real times}$$

Thus we can use our multiplication saving technique here too!

(see: MatrixAPSPmult.java)
Shortest paths using matrices

All-pairs-shortest-paths($W$)
$L(1) = W$, $n = W\.rows$, $m = 1$
while $m < n$
    $L(2m) = ESP(L(m), L(m))$
    $m = 2m$
return $L(m)$

(ESP is L min op on previous slide)
Shortest paths using matrices

Runtime:
$|V|^3 \log |V|$

Correctness:
By definition (brute force with some computation savers)
Floyd-Warshall

The Floyd-Warshall is similar but uses another shortest path property.

Suppose we have a graph $G$, if we add a single vertex $k$ to get $G'$.

We now need to recompute all shortest paths.
Eeither the path goes through $k$, or remains unchanged

\[
d^k_{i,j} = \min (d^{k-1}_{i,j}, d^{k-1}_{i,k} + d^{k-1}_{k,j})
\]
Floyd-Warshall

Floyd-Warshall(W) // dynamic prog

d^0_{i,j} = W_{i,j}, n = W.rows

for k = 1 to n
    for i = 1 to n
        for j = 1 to n
            d^k_{i,j} = \min (d^{k-1}_{i,j}, d^{k-1}_{i,k} + d^{k-1}_{k,j})
Floyd-Warshall

Runtime:
$O(|V|^3)$

Correctness:
Again, by definition of shortest path