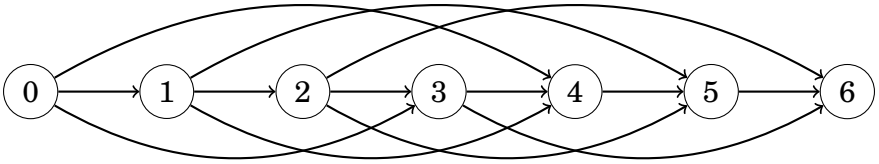


Using this representation, the shortest path from node 0 to node n corresponds to a solution with the minimum number of coins, and the total number of paths from node 0 to node n equals the total number of solutions.

Successor paths

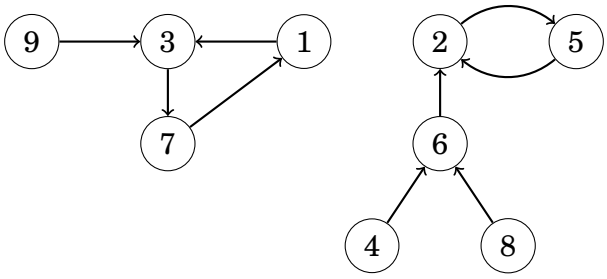
For the rest of the chapter, we will focus on **successor graphs**. In those graphs, the outdegree of each node is 1, i.e., exactly one edge starts at each node. A successor graph consists of one or more components, each of which contains one cycle and some paths that lead to it.

Successor graphs are sometimes called **functional graphs**. The reason for this is that any successor graph corresponds to a function that defines the edges of the graph. The parameter for the function is a node of the graph, and the function gives the successor of that node.

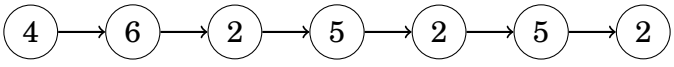
For example, the function

x	1	2	3	4	5	6	7	8	9
succ(x)	3	5	7	6	2	2	1	6	3

defines the following graph:



Since each node of a successor graph has a unique successor, we can also define a function succ(x,k) that gives the node that we will reach if we begin at node x and walk k steps forward. For example, in the above graph succ$(4,6) = 2$, because we will reach node 2 by walking 6 steps from node 4:



A straightforward way to calculate a value of succ(x,k) is to start at node x and walk k steps forward, which takes $O(k)$ time. However, using preprocessing, any value of succ(x,k) can be calculated in only $O(\log k)$ time.

The idea is to precalculate all values of succ(x,k) where k is a power of two and at most u, where u is the maximum number of steps we will ever walk. This can be efficiently done, because we can use the following recursion:

$$\text{succ}(x,k) = \begin{cases} \text{succ}(x) & k = 1 \\ \text{succ}(\text{succ}(x,k/2),k/2) & k > 1 \end{cases}$$

Precalculating the values takes $O(n \log u)$ time, because $O(\log u)$ values are calculated for each node. In the above graph, the first values are as follows:

x	1	2	3	4	5	6	7	8	9
succ$(x,1)$	3	5	7	6	2	2	1	6	3
succ$(x,2)$	7	2	1	2	5	5	3	2	7
succ$(x,4)$	3	2	7	2	5	5	1	2	3
succ$(x,8)$	7	2	1	2	5	5	3	2	7
$\cdots$									

After this, any value of $\text{succ}(x,k)$ can be calculated by presenting the number of steps k as a sum of powers of two. For example, if we want to calculate the value of $\text{succ}(x,11)$, we first form the representation $11 = 8+2+1$. Using that,

$$\text{succ}(x,11) = \text{succ}(\text{succ}(\text{succ}(x,8),2),1).$$

For example, in the previous graph

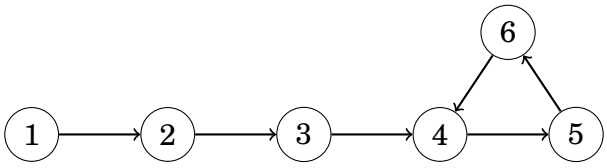
$$\text{succ}(4,11) = \text{succ}(\text{succ}(\text{succ}(4,8),2),1) = 5.$$

Such a representation always consists of $O(\log k)$ parts, so calculating a value of $\text{succ}(x,k)$ takes $O(\log k)$ time.

Cycle detection

Consider a successor graph that only contains a path that ends in a cycle. We may ask the following questions: if we begin our walk at the starting node, what is the first node in the cycle and how many nodes does the cycle contain?

For example, in the graph



we begin our walk at node 1, the first node that belongs to the cycle is node 4, and the cycle consists of three nodes (4, 5 and 6).

A simple way to detect the cycle is to walk in the graph and keep track of all nodes that have been visited. Once a node is visited for the second time, we can conclude that the node is the first node in the cycle. This method works in $O(n)$ time and also uses $O(n)$ memory.

However, there are better algorithms for cycle detection. The time complexity of such algorithms is still $O(n)$, but they only use $O(1)$ memory. This is an important improvement if n is large. Next we will discuss Floyd's algorithm that achieves these properties.

Floyd's algorithm

Floyd's algorithm[2] walks forward in the graph using two pointers a and b. Both pointers begin at a node x that is the starting node of the graph. Then, on each turn, the pointer a walks one step forward and the pointer b walks two steps forward. The process continues until the pointers meet each other:

```
a = succ(x);
b = succ(succ(x));
while (a != b) {
    a = succ(a);
    b = succ(succ(b));
}
```

At this point, the pointer a has walked k steps and the pointer b has walked $2k$ steps, so the length of the cycle divides k. Thus, the first node that belongs to the cycle can be found by moving the pointer a to node x and advancing the pointers step by step until they meet again.

```
a = x;
while (a != b) {
    a = succ(a);
    b = succ(b);
}
first = a;
```

After this, the length of the cycle can be calculated as follows:

```
b = succ(a);
length = 1;
while (a != b) {
    b = succ(b);
    length++;
}
```

[2] The idea of the algorithm is mentioned in [46] and attributed to R. W. Floyd; however, it is not known if Floyd actually discovered the algorithm.