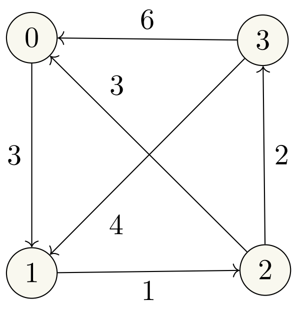
6.3 All-pairs shortest paths

A single-source shortest path algorithm finds the shortest path from a given source vertex to every vertex in the graph. Sometimes we are interested in the shortest path between every pair of vertices in a graph (i.e. from vertex 0 to every other vertex, from vertex 1 to every other vertex, and so on). One way of achieving all-pairs shortest paths is to run a single-source shortest path algorithm n times, each time starting from a different vertex. Using Dijkstra's algorithm, which has a time complexity of $O(n^2)$, the all-pairs shortest paths problem can be solved in $O(n^3)$ time. Figure 16 shows the result of running such an algorithm on a weighted graph.

```
D 4 W
0 1 3
1 2 1
2 3 2
2 0 3
3 0 6
3 1 4
```

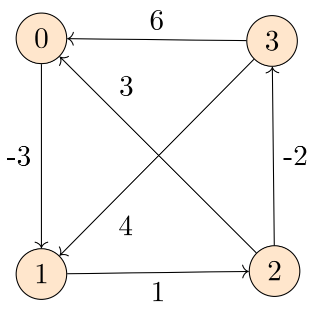


Root	Distance array	Parent array
0	[0, 3, 4, 6]	[-, 0, 1, 2]
1	[4, 0, 1, 3]	[2, -, 1, 2]
2	[3, 6, 0, 2]	[2, 3, -, 2]
3	[6, 4, 5, 0]	[3, 3, 1, -]

Figure 16: Top left: the textual description of a weighted directed graph. Top right: the drawing of the graph. Bottom: the parent and distance arrays obtained by Dijkstra's algorithm. Each distance array shows the shortest distance from the root to each vertex. Each parent array represents the structure of the shortest path tree. The stacking of the distance or parent arrays can be viewed as a square matrix. The element $Distance[i][j]$ is the shortest distance from vertex i to vertex j. The element $Parent[i][j]$ is the parent of vertex j in the shortest path tree that is rooted at vertex i.

One limitation of Dijkstra's algorithm is that, in the presence of negative weights, it may produce non-optimal solutions. This is because the algorithm is greedy; it finds a vertex that is closest to the root and adds it to the shortest path tree. This works fine with non-negative edges because any other path to that vertex has to be at least as long as the first one. With negative edges, however, a greedy algorithm can miss better future options by committing to a locally optimal choice. Figure 17 shows the result of Dijkstra's algorithm applied to a graph with negative edges. It can be seen that, for some pairs of vertices, the algorithm has found non-optimal paths.

D	4	W
0	1	−3
1	2	1
2	3	−2
2	0	3
3	0	6
3	1	4



Root	Distance by Dijkstra	Correct distance
0	[0, -3, -2, -4]	[0, -3, -2, -4]
1	[4, 0, 1, -1]	[4, 0, 1, -1]
2	[3, 2, 0, -2]	[3, 0, 0, -2]
3	[6, 4, 5, 0]	[6, 3, 4, 0]

Figure 17: Top left: the textual description of a weighted directed graph that contains some negative weights. Top right: the drawing of the graph. Bottom: the result of applying Dijkstra's algorithm on all the vertices and the true shortest distance between vertices. It can be seen that, in some cases, Dijkstra's algorithm produces non-optimal solutions. For example, starting from vertex 3, Dijkstra sees the edge to 1 with a weight of 4 as the best choice, while a worse local choice (the edge from 3 to 0 with a weight of 6) can later lead to a shorter path to vertex 1.

6.3.1 Developing a recursive solution

In this section, we develop a recursive function for the shortest path problem which will lead to a DP solution for the all-pairs shortest paths problem, called the Floyd-Warshall algorithm. While we will address the problem of producing optional solutions for graphs with edges that have negative weights, we limit the solution to graphs that do not have *negative cycles* which are defined as cycles along which the sum of weights is a negative value.

6.3.1.1 Intermediate vertices

Consider a path $p = (v_s, v_1, v_2, ..., v_l, v_d)$ which is from the source vertex v_s to the destination vertex v_d. We call the vertices in between the source and destination *intermediate vertices.* The path p has l intermediate vertices denoted by v_1, v_2, $\ldots$, v_l.

6.3.1.2 Properties of shortest paths

The following properties hold for shortest paths:

- Shortest paths are simple paths; that is, no vertex is repeated along the path. Why? Because repeating a vertex (in the absence of a negative cycle) cannot make the sum of the weights smaller.
- Any subpath (consecutive subsequence of vertices) of a shortest path is also a shortest path. Why? If this was not the case, then there would be a shorter path connecting the vertex at the beginning of the subpath to the vertex at the end of the subpath. This would mean that the original path is not the shortest path, which is a contradiction.

6.3.1.3 A new distance function

Consider a graph with n vertices identified by numbers from 0 to $n-1$. We define the function $d(i, j, k)$ to be the shortest distance from vertex i to vertex j if we only consider paths whose intermediate vertices are in $\{0, 1, \ldots, k\}$. For example, $d(i, j, 1)$ is the shortest distance from i to j if we only consider the following paths (if they exit):

- a path with no intermediate vertex, that is, an edge from i to j;
- a path from i to 0 and then to j;
- a path from i to 1 and then to j;
- a path from i to 0, to 1, and then to j; and
- a path from i to 1, to 0, and then to j.

The parameters i and j are integers between 0 and $n-1$ corresponding to the vertices of the graph. The parameter k can take values between -1 and $n-1$. The value of -1 means that no intermediate vertices can be used. The value of $n-1$ means that intermediate vertices can be any vertex in the graph (i.e. all paths are considered). Figure 18 shows a few examples of the value of this function on a graph.

i	j	k	intermediates allowed	distance	path
3	1	-1	$\{\}$	4	$(3,1)$
2	1	-1	$\{\}$	∞	-
3	1	0	$\{0\}$	3	$(3,0,1)$
2	1	0	$\{0\}$	0	$(2,0,1)$
3	2	0	$\{0\}$	∞	-
3	2	1	$\{0,1\}$	4	$(3,0,1,2)$
1	0	1	$\{0,1\}$	∞	-
1	0	2	$\{0,1,2\}$	4	$(1,2,0)$
1	0	3	$\{0,1,2,3\}$	4	$(1,2,0)$

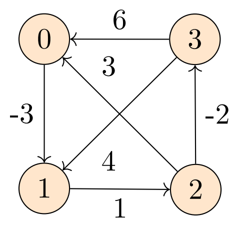


Figure 18: The table on the left presents some examples of the value of the function $d(i,j,k)$ for the graph on the right.

The value of $d(,,k)$ can be recursively expressed based on the values of the form $d(,,k-1)$. When evaluating $d(i,j,k)$, two cases can arise:

- k is an intermediate vertex in the shortest path from i to j. In this case, the path can be seen as the concatenation of the shortest path from i to k and the shortest path from k to j. The intermediate vertices of these two paths are in $\{0,1,\ldots,k-1\}$. Note that k is not an intermediate vertex, because in one of the paths, it is the destination, and in the other one, it is the source. Therefore, the lengths of these two paths are $d(i,k,k-1)$ and $d(k,j,k-1)$ respectively.
- k is not an intermediate vertex in the shortest path from i to j. In this case, the intermediate vertices are in $\{0,1,\ldots,k-1\}$. Therefore, the length of the path is the same as $d(i,j,k-1)$.

This leads to the following recurrence equation:

$$d(i,j,k) = \begin{cases} 0, & \text{if } k=-1 \wedge i=j \\ \infty, & \text{if } k=-1 \wedge (i,j) \notin E \\ w((i,j)), & \text{if } k=-1 \wedge (i,j) \in E \\ \min(d(i,k,k-1)+d(k,j,k-1), d(i,j,k-1)), & \text{if } k \geq 0 \ . \end{cases}$$

In this equation:

- The first three cases are the base cases, where $k=-1$; that is, no intermediate vertex is used.
- The first base case is when the source and destination are the same.
- The second base case is when there is no edge directly connecting i to j and no intermediate vertices are allowed. In this case the distance is defined to be infinity.

- The third case is when there is a direct edge connecting i to j. In this case, the distance is the weight of the edge.

- The last case is the recursive case. The shortest distance from i to j is the smaller of the two cases mentioned above: one has k as the intermediate vertex, the other does not.

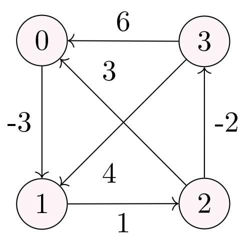
6.3.2 The Floyd-Warshall algorithm

The Floyd-Warshall algorithm takes a weighted (directed or undirected) graph with no negative cycle and computes the shortest distance between every pair of vertices.

The algorithm can be implemented recursively according to the recursive definition of $d(i, j, k)$. Since there will be repeated recursive calls, memoisation must be used. This leads to a top-down DP solution. Since each of the parameters i, j, and k can take on n distinct values, the size of the cache will be in $O(n^3)$. The body of the function itself is very simple and runs in constant time. Thus the running time of the algorithm is proportional to the number of unique recursive calls which is in $O(n^3)$.

6.3.2.1 Bottom-up implementation

A bottom-up version of the algorithm can be implemented by starting from a distance matrix which contains the values of $d(i, j, k)$ for $k = -1$. Figure 19 shows an example of such a matrix. Then the matrix can be updated iteratively by considering more intermediate vertices in each iteration. This leads to the following algorithm.



```
                 j

i |    0     1     2     3
--------------------------
0 |    0,   -3,  inf,  inf
1 |  inf,    0,    1,  inf
2 |    3,  inf,    0,   -2
3 |    6,    4,  inf,    0
```

Figure 19: Left: a weighted directed graph. Right: the values of the function $d(i, j, -1)$ which represents the shortest distance from vertex i to vertex j when no intermediate vertex is used. The matrix on the right is similar to an adjacency matrix. The only differences are: (a) when there is no edge from i to j, the value is set to infinity (instead of NULL); and (b) when $i = j$, the distance is considered to be zero. This matrix is the starting state of the (bottom-up) Floyd-Warshall algorithm.

```
procedure FLOYD($Distance$)
  $n \leftarrow$ number of rows (or columns) of the matrix $Distance$
  for $k$ from 0 to $n-1$:
    for $i$ from 0 to $n-1$:
      for $j$ from 0 to $n-1$:
        if $Distance[i][j] > Distance[i][k] + Distance[k][j]$
          $Distance[i][j] \leftarrow Distance[i][k] + Distance[k][j]$
  return $Distance$
```

In the algorithm:

- The matrix *Distance* initially contains all the single edge (without intermediate vertices) distances. This information is directly available from the graph.
- In each iteration of the outer loop, a new vertex k is considered as an intermediate vertex in computing the shortest path between every pair of vertices.
- At the end of the algorithm, $Distance[i][j]$ has the shortest path from i to j.
- The runtime complexity of the algorithm is in $\Theta(n^3)$.
- The algorithm takes advantage of the fact that the value of $d(i, j, k)$ only depends on the values of $d(i, k, k-1)$ and $d(k, j, k-1)$. Thus, it only keeps one distance matrix in memory and incrementally updates the matrix. This means that the space complexity of the algorithm is in $\Theta(n^2)$.
- The algorithm can be extended to update a parent matrix. At the beginning, $parent[i][j]$ is set to i if there is an edge (i, j) in the graph. Later, every time the distance matrix is updated, $parent[i][j]$ is set to $parent[k][j]$.

Trace of Floyd-Warshall algorithm

```
initial values (k=-1)

         distance                                 parent
       0     1     2     3                 0     1     2     3
---------------------------             --------------------------
0 |    0,   -3,  inf,  inf              0 | None,    0, None, None
1 |  inf,    0,    1,  inf              1 | None, None,    1, None
2 |    3,  inf,    0,   -2              2 |    2, None, None,    2
3 |    6,    4,  inf,    0              3 |    3,    3, None, None


k = 0

         distance                                 parent
       0     1     2     3                 0     1     2     3
---------------------------             --------------------------
0 |    0,   -3,  inf,  inf              0 | None,    0, None, None
1 |  inf,    0,    1,  inf              1 | None, None,    1, None
2 |    3,    0,    0,   -2              2 |    2,    0, None,    2
3 |    6,    3,  inf,    0              3 |    3,    0, None, None


k = 1

         distance                                 parent
       0     1     2     3                 0     1     2     3
---------------------------             --------------------------
0 |    0,   -3,   -2,  inf              0 | None,    0,    1, None
1 |  inf,    0,    1,  inf              1 | None, None,    1, None
2 |    3,    0,    0,   -2              2 |    2,    0, None,    2
3 |    6,    3,    4,    0              3 |    3,    0,    1, None


k = 2

         distance                                 parent
       0     1     2     3                 0     1     2     3
---------------------------             --------------------------
0 |    0,   -3,   -2,   -4              0 | None,    0,    1,    2
1 |    4,    0,    1,   -1              1 |    2, None,    1,    2
2 |    3,    0,    0,   -2              2 |    2,    0, None,    2
3 |    6,    3,    4,    0              3 |    3,    0,    1, None


k = 3

         distance                                 parent
       0     1     2     3                 0     1     2     3
---------------------------             --------------------------
0 |    0,   -3,   -2,   -4              0 | None,    0,    1,    2
1 |    4,    0,    1,   -1              1 |    2, None,    1,    2
2 |    3,    0,    0,   -2              2 |    2,    0, None,    2
3 |    6,    3,    4,    0              3 |    3,    0,    1, None
```

* Values have been printed after the completion of each iteration of the outer loop.

7 Backtracking

Backtracking is an algorithmic technique for generating all or some solutions of a computational problem. Examples include:

- generating permutations of a set of items;
- generating subsets of a set;
- completing a Sudoku puzzle; and
- placing 8 queens on a chess board so that they do not attack each other.

The idea is that candidate solutions can be constructed incrementally.

Example 7.1. When constructing a string representing a binary number of certain length, one can start with an empty string. Then, recursively, the next candidates are constructed by appending 0 or 1 to a copy of a candidate from the previous stage. This process is repeated until the desired length is reached.

The following program prints all binary numbers of a given length.

```python
def print_binary_numbers(desired_length, string=""):
    if len(string) == desired_length:
        print(string)
    else:
        print_binary_numbers(desired_length, string + "0")
        print_binary_numbers(desired_length, string + "1")
```

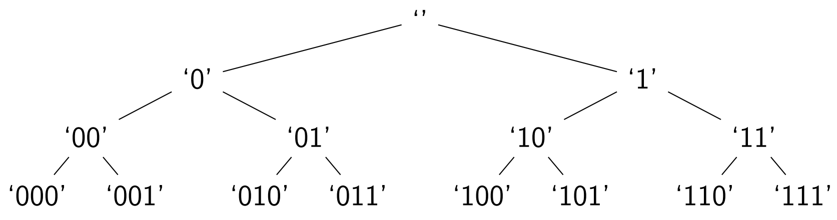
Sample output:

```
> print_binary_numbers(3)

000
001
010
011
100
101
110
111
```

The incremental generation of candidates can be seen as a tree. There is a vertex for each candidate. There is an edge from a candidate to another candidate if and only if the latter can be constructed by adding an element to the former.

Example 7.2. Consider the problem of generating all binary numbers of length n. Draw the tree of candidates for $n = 3$.



In most problems these trees grow very rapidly with respect to the size of the problem.

7.1 Backtracking algorithm

A traversal (BFS or DFS) can be used to search the tree for solutions.

Typically there are regularities in the graph of candidates. Each problem has its own recursive pattern. Therefore, the graph can be implicitly constructed on the fly. Since the graph is a tree, given its root it can be fully described with a Children function.

The backtracking algorithm is a DFS on this implicit tree.

```
procedure DFS-Backtrack(candidate, input, output)
  if Is-Solution(candidate, input)
    AddToOutput(candidate, output)
  else
    for child-candidate in Children(candidate, input)
      DFS-Backtrack(child-candidate, input, output)
```

Elements of the algorithm are:

- *candidate* is a vertex of the implicit tree.
- *input* contains information about the instance of the problem. For example, when generating binary numbers, the input is the desired length.
- *output* is where the solutions are collected. For example, it can be a list that at the end of the algorithm will contain all binary numbers.

- IS-SOLUTION is a predicate (a Boolean-valued function) that returns true if the candidate is a solution (a leaf vertex). For example, when generating binary strings, it returns true if the candidate has the desired length.

- CHILDREN is a function that takes a candidate and returns a (possibly empty) sequence of candidates that can be constructed by augmenting the given candidate. Augmentation usually involves adding another element to a copy of the given candidate. The function CHILDREN often needs to refer to the *input* in order to construct the new candidates. For example, when generating binary numbers of a certain length, the children of a candidate are candidates constructed by appending 0 or 1 to the end of the given candidate binary string.

7.1.1 Example implementation

```python
def dfs_backtrack(candidate, input, output):
    if is_solution(candidate, input):
        add_to_output(candidate, output)
    else:
        for child_candidate in children(candidate):
            dfs_backtrack(child_candidate, input, output)


def is_solution(candidate, desired_length):
    return len(candidate) == desired_length


def children(candidate):
    return [candidate + "0", candidate + "1"]


def add_to_output(candidate, output):
    output.append(candidate)


def binary_numbers(desired_length):
    solutions = []
    dfs_backtrack("", desired_length, solutions)
    return solutions
```

7.1.2 Sample output

```
> binary_numbers(4)

['0000',
 '0001',
 '0010',
 '0011',
 '0100',
 '0101',
 '0110',
 '0111',
 '1000',
 '1001',
 '1010',
 '1011',
 '1100',
 '1101',
 '1110',
 '1111']
```

7.2 Backtracking for search

Backtracking can be used to search for a solution. For example in Sudoku the algorithm tries various placements of numbers in empty cells until it finds a solution (where all cells are filled and all criteria of Sudoku are met). If only one solution is desired, the algorithm can be modified to set a flag as soon as it finds a solution so that the search can be stopped.

The algorithm is called backtracking because it makes some choices when branching down but it can "backtrack" by returning from recursive calls and try other branches.

7.2.1 Pruning

The performance of the algorithm can be improved by *pruning*, which means some nodes are not expanded because they cannot lead to solution. For example, when searching for a solution of a Sudoku problem, if the candidate already breaches some of the criteria for being a valid solution, there is no point in expanding it any further. At this point the (recursive) call can return.

To implement pruning the following statement is added to the beginning of DFS-BACKTRACK:

```
if SHOULD-PRUNE(candidate)
    return
```

The function SHOULD-PRUNE takes a candidate and returns true if it can be determined that no leaf candidate in the subtree rooted at the given candidate is a solution.

For example, when searching for binary numbers of a given length, if we are only interested in binary numbers that have up to four '1's in them, the function can be implemented as the following:

```python
def should_prune(candidate):
    return candidate.count("1") > 4
```