Algorithms

Algorithms aren't just a thing of computers – they are everywhere, and learning to see and use patterns to your advantage is one of the best "life hacks".

Time management algorithms

1. **Earliest Due Date**. With this, you sort all of your tasks by deadlines and start with the one that's due next. This way, you'll make sure you won't run into any time issues.
2. **Moore's Algorithm**. If it's too late for Earliest Due Date, because you already *know* you won't make it all in time, skip the task that takes the longest to free a big chunk of time and have a shot at getting everything else done.
3. **Shortest Processing Time**. With lots of small tasks, it makes sense to sort them by how long they're going to take and knock out the shortest ones first.

Tidying algorithms

1. **Bubble sort**. You only ever compare two items at a time and put them in the right order, going through all pairs of items one by one and swap them if their order is wrong. Once you're through the list, start over until you don't have to swap anything anymore. Perfect for sorting books!
2. **Insertion sort**. This is less incremental, making you take out *all* items to be organized and then inserting them in the right order. This is ideal for organizing your wardrobe.
3. **Merge sort**. It works by dividing all of your collections into multiple piles, sorting the piles (for example by room), and then re-assembling the sorted piles to get a full solution.

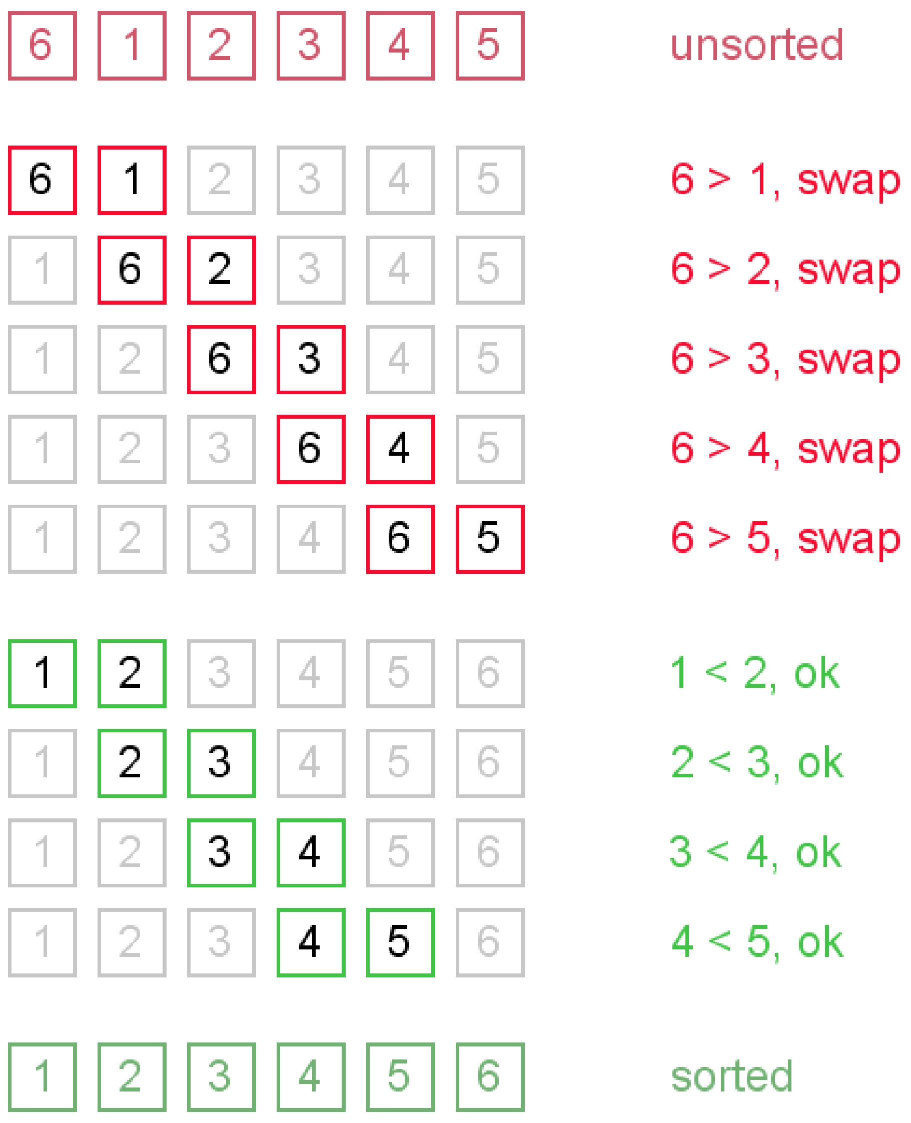
Bubble sort

The bubble sort algorithm works by repeatedly swapping adjacent elements that are not in order until the whole list of items is in sequence. In this way, items can be seen as bubbling up the list according to their key values.

```
// This is the bubble sort algorithm
for (i = N - 1; i > 0; i--){
    for (j = 0; j < i; j++){
        if (input[j] > input[j + 1]){
            // swap here
            int swapValue = input[j];
            input[j] = input[j + 1];
            input[j + 1] = swapValue;
            // Bubble-sort optimization
            // Add a flag in case a swap has occured, then check that flag to figure out
            // if you still need to do subsequent swaps
            // swapped = true
        }
    }
}
```

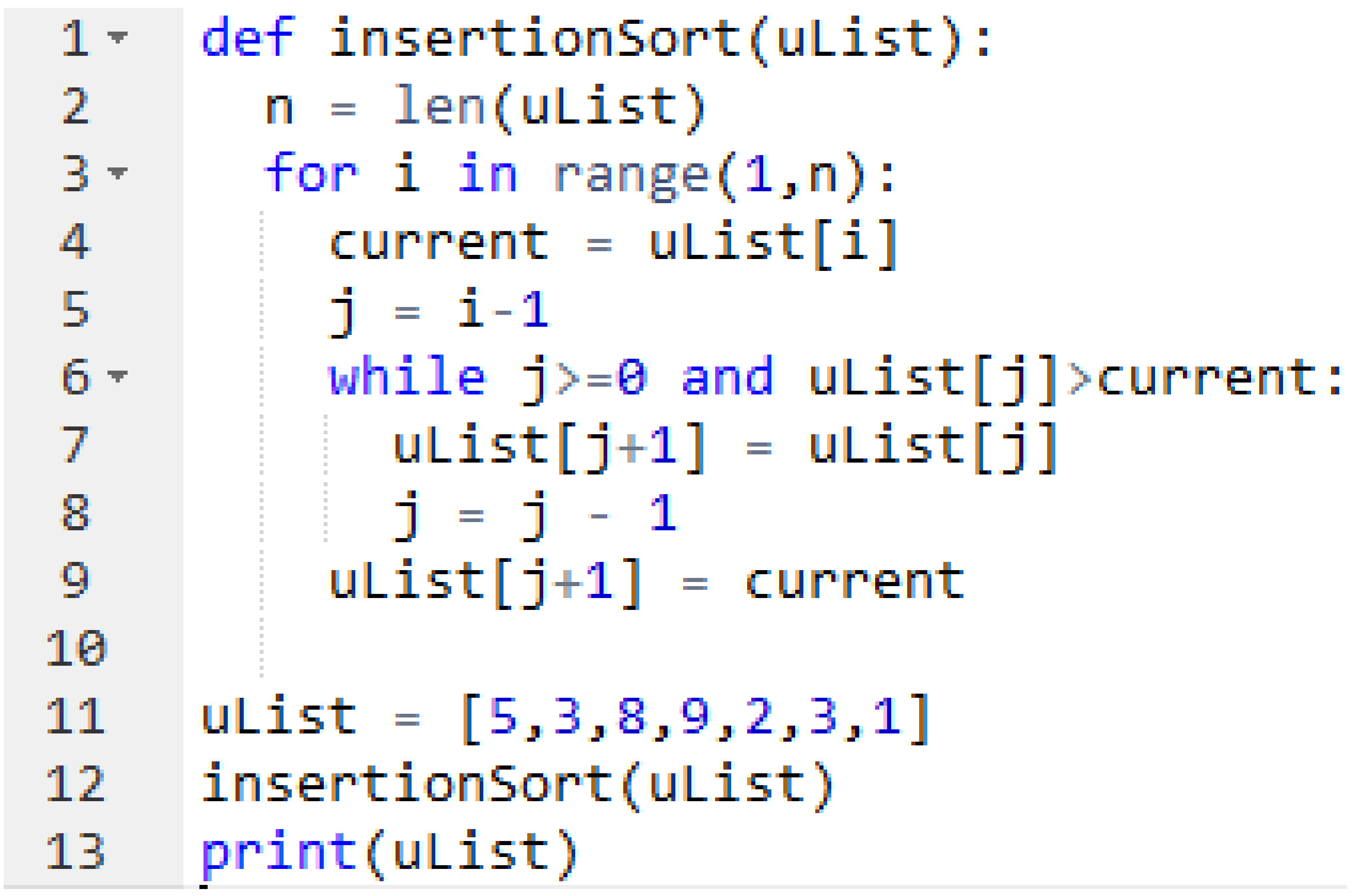
The primary advantage of the bubble sort is that it is popular and easy to implement. Furthermore, in the bubble sort, elements are swapped in place without using additional temporary storage, so the space requirement is at a minimum.

The main disadvantage of the bubble sort is the fact that it does not deal well with a list containing a huge number of items. This is because the bubble sort requires n-squared processing steps for every n number of elements to be sorted. As such, the bubble sort is mostly suitable for academic teaching but not for real-life applications.



Insertion Sort

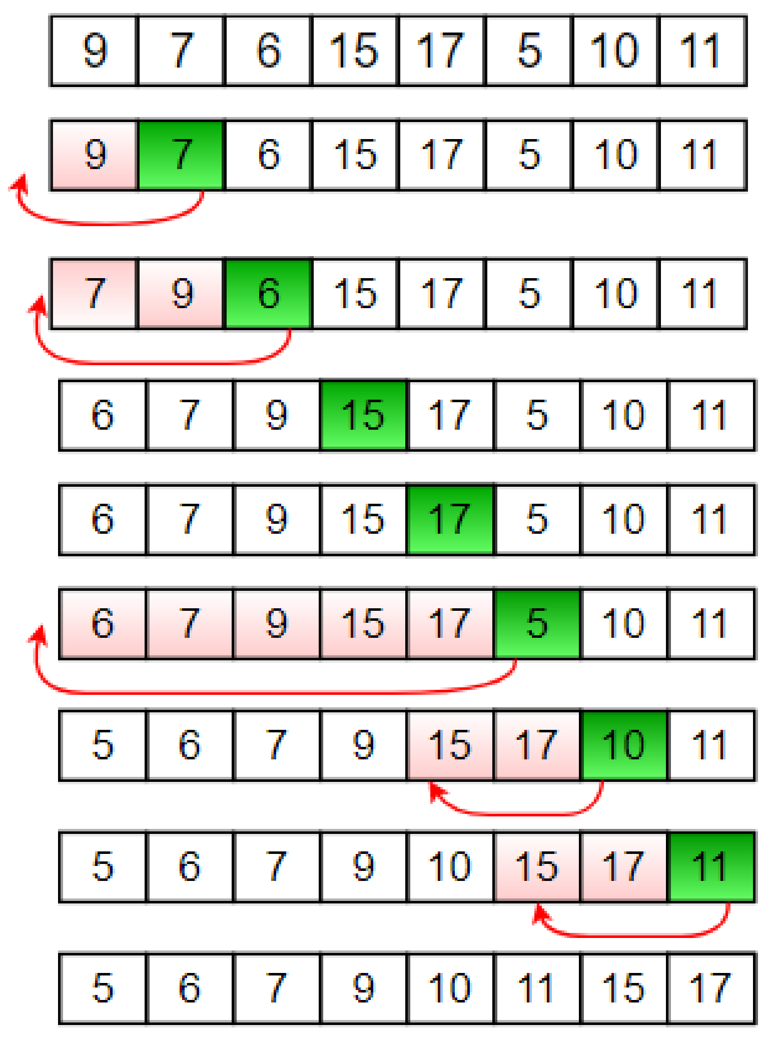
The insertion sort repeatedly scans the list of items, each time inserting the item in the unordered sequence into its correct position.



```python
def insertionSort(uList):
  n = len(uList)
  for i in range(1,n):
    current = uList[i]
    j = i-1
    while j>=0 and uList[j]>current:
      uList[j+1] = uList[j]
      j = j - 1
    uList[j+1] = current

uList = [5,3,8,9,2,3,1]
insertionSort(uList)
print(uList)
```

The main advantage of the insertion sort is its simplicity. It also exhibits a good performance when dealing with a small list. The insertion sort is an in-place sorting algorithm so the space requirement is minimal.

The disadvantage of the insertion sort is that it does not perform as well as other, better sorting algorithms. With n-squared steps required for every n element to be sorted, the insertion sort does not deal well with a huge list. Therefore, the insertion sort is particularly useful only when sorting a list of few items.

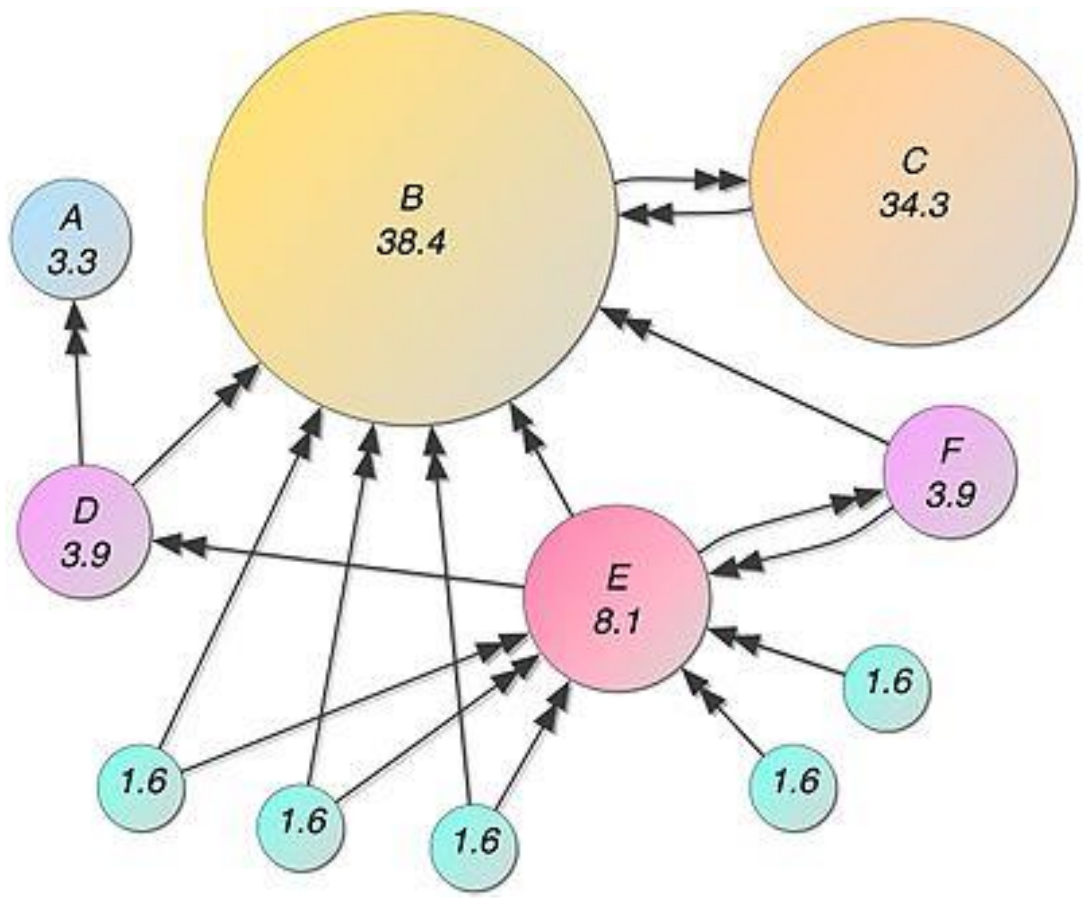


PageRank Algorithm

This is an algorithm used by Google Search to rank websites in their search engine results. PageRank was named after Larry Page, one of the founders of Google. PageRank is a way of measuring the importance of website pages. According to Google:

PageRank works by counting the number and quality of links to a page to determine a rough estimate of how important the website is. The underlying assumption is that more important websites are likely to receive more links from other websites.

It is not the only algorithm used by Google to order search engine results, but it is the first algorithm that was used by the company, and it is the best-known



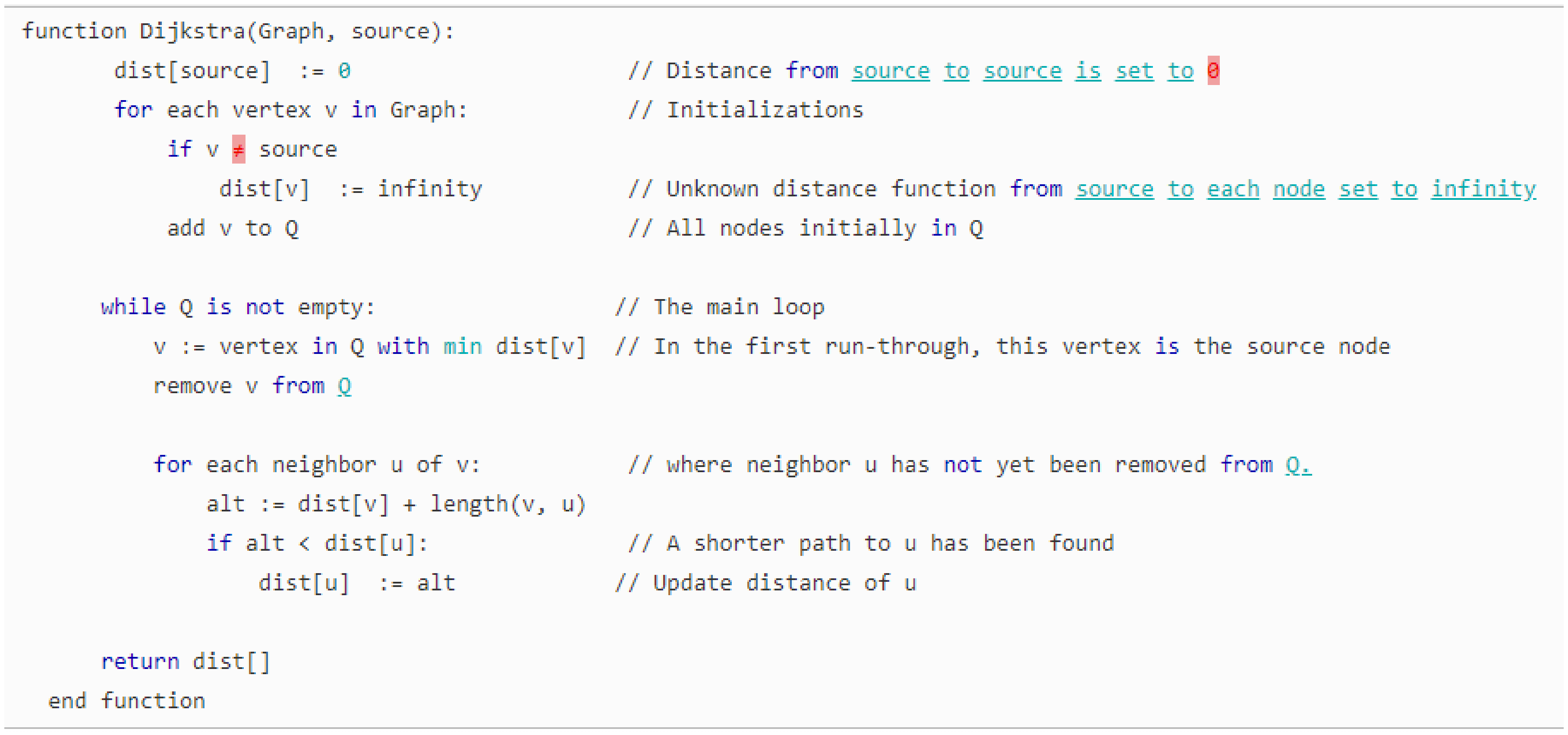
After making an initial guess at the importance of each site in a network, the relative importance is calculated by looking at the number of pages that link to a given site and how important those pages are.

$$PR(A) = \frac{1-d}{N} + d\left(\frac{PR(B)}{L(B)} + \frac{PR(C)}{L(C)} + \frac{PR(D)}{L(D)} + \cdots\right).$$

Eventually the importance values stabalise.

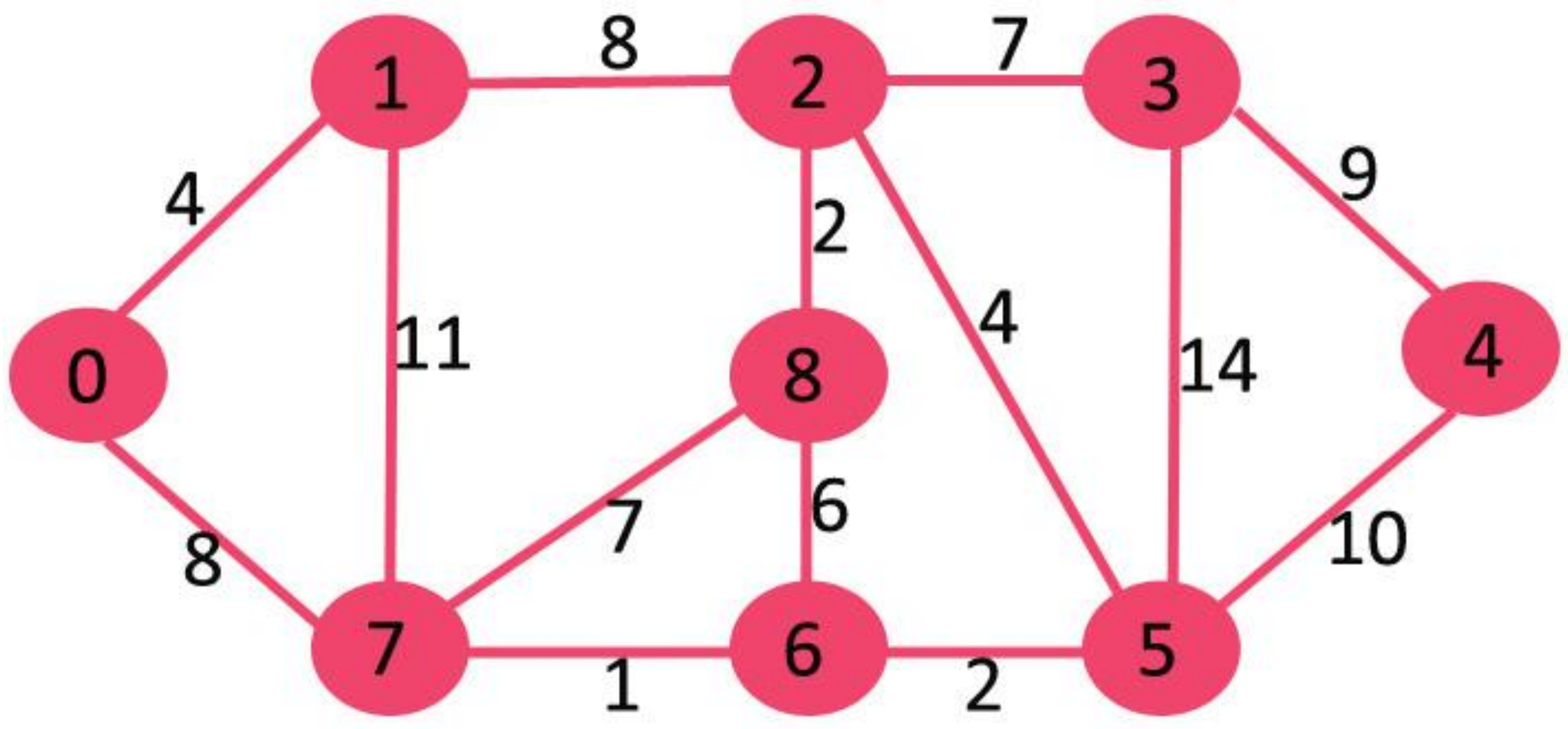
Dijkstra’s shortest path algorithm

One algorithm for finding the shortest path from a starting node to a target node in a weighted graph is Dijkstra's algorithm. The algorithm creates a tree of shortest paths from the starting vertex, the source, to all other points in the graph.



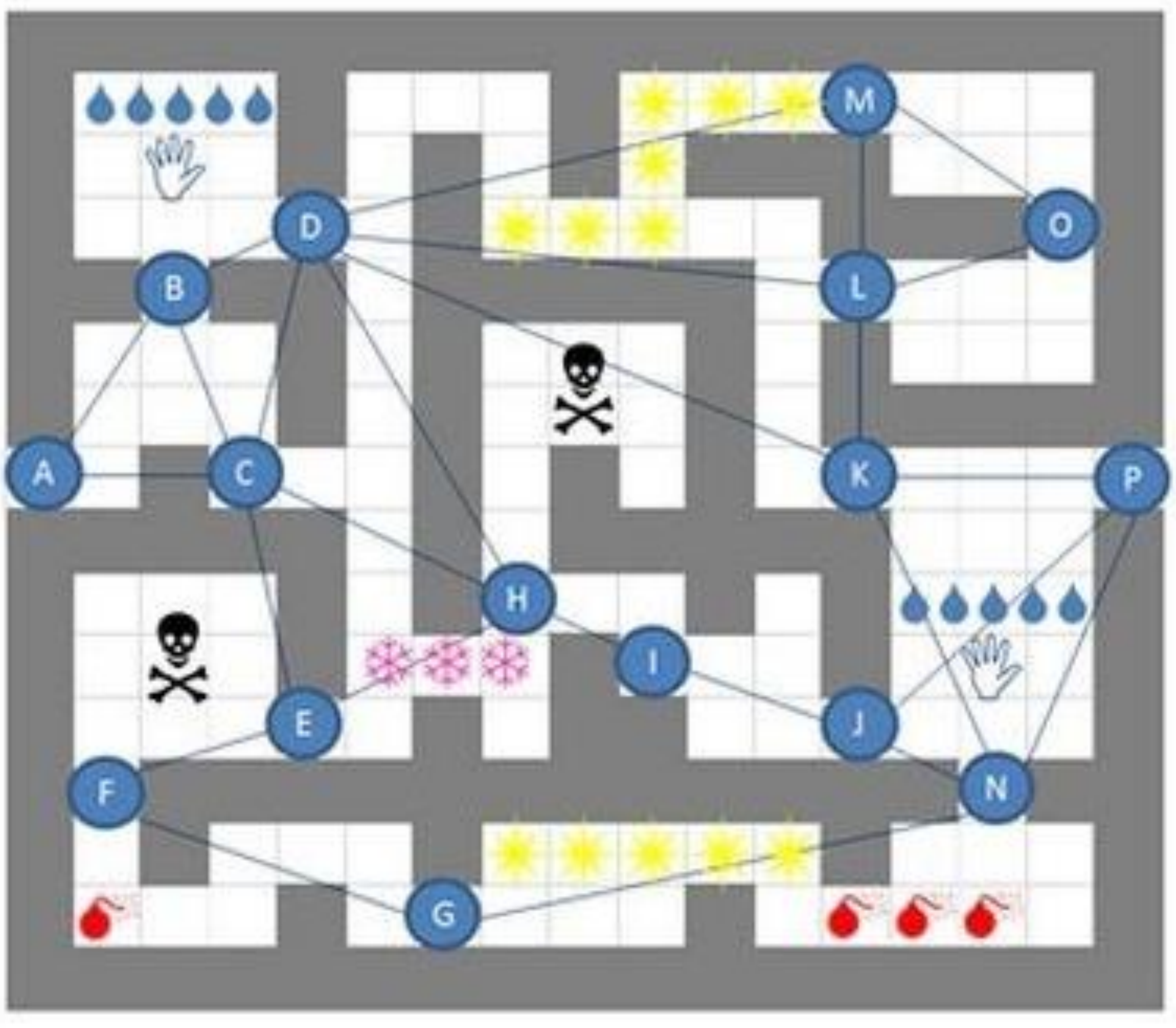
```
function Dijkstra(Graph, source):
       dist[source]  := 0                     // Distance from source to source is set to 0
       for each vertex v in Graph:            // Initializations
           if v ≠ source
               dist[v]  := infinity           // Unknown distance function from source to each node set to infinity
           add v to Q                         // All nodes initially in Q

      while Q is not empty:                  // The main loop
          v := vertex in Q with min dist[v]  // In the first run-through, this vertex is the source node
          remove v from Q

          for each neighbor u of v:           // where neighbor u has not yet been removed from Q.
              alt := dist[v] + length(v, u)
              if alt < dist[u]:               // A shorter path to u has been found
                  dist[u]  := alt            // Update distance of u

      return dist[]
  end function
```

(Source: https://brilliant.org/wiki/dijkstras-short-path-finder/)



Data structures – trees and graphs

$$\begin{tikzcd}
 & & & & & & 50 \arrow[dlll] \arrow[drrr] & & & & \\
 & & & 17 \arrow[dll] \arrow[drr] & & & & & & 72 \arrow[dll] \arrow[dr] & \\
 & 12 \arrow[dl] \arrow[dr] & & & & 23 \arrow[dl] & & 54 \arrow[dr] & & & 76 \\
9 & & 14 & & 19 & & & & 67 & &
\end{tikzcd}$$



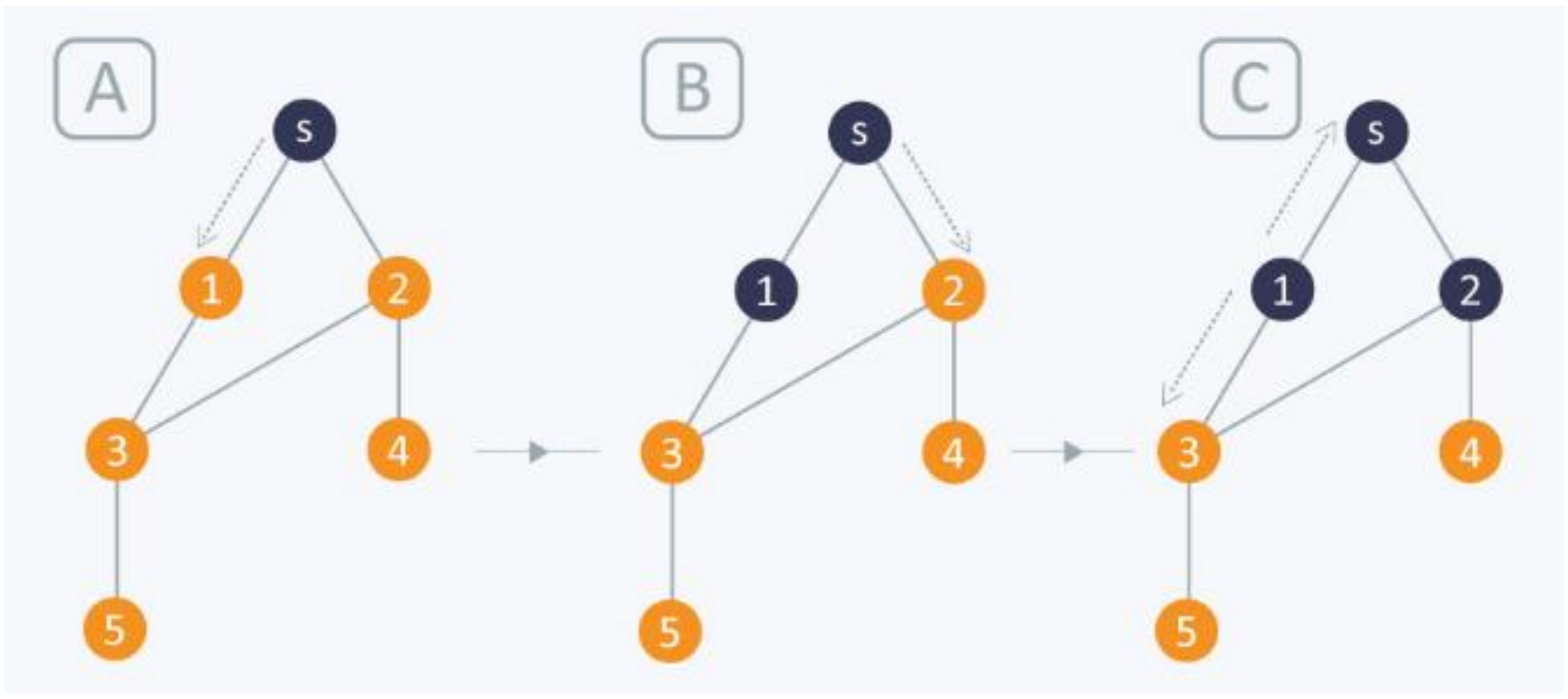
Breadth first traversal algorithm

BFS is a traversing algorithm where you should start traversing from a selected node (source or starting node) and traverse the graph layerwise thus exploring the neighbour nodes (nodes which are directly connected to source node). You must then move towards the next-level neighbour nodes.

As the name BFS suggests, you are required to traverse the graph breadthwise as follows:

- First move horizontally and visit all the nodes of the current layer
- Move to the next layer

```
BFS (G, s)                   //Where G is the graph and s is the source node
      let Q be queue.
      Q.enqueue( s ) //Inserting s in queue until all its neighbour vertices are marked.

      mark s as visited.
      while ( Q is not empty)
           //Removing that vertex from queue,whose neighbour will be visited now
           v  =  Q.dequeue( )

          //processing all the neighbours of v
          for all neighbours w of v in Graph G
               if w is not visited
                        Q.enqueue( w )             //Stores w in Q to further visit its
neighbour
                        mark w as visited.
```



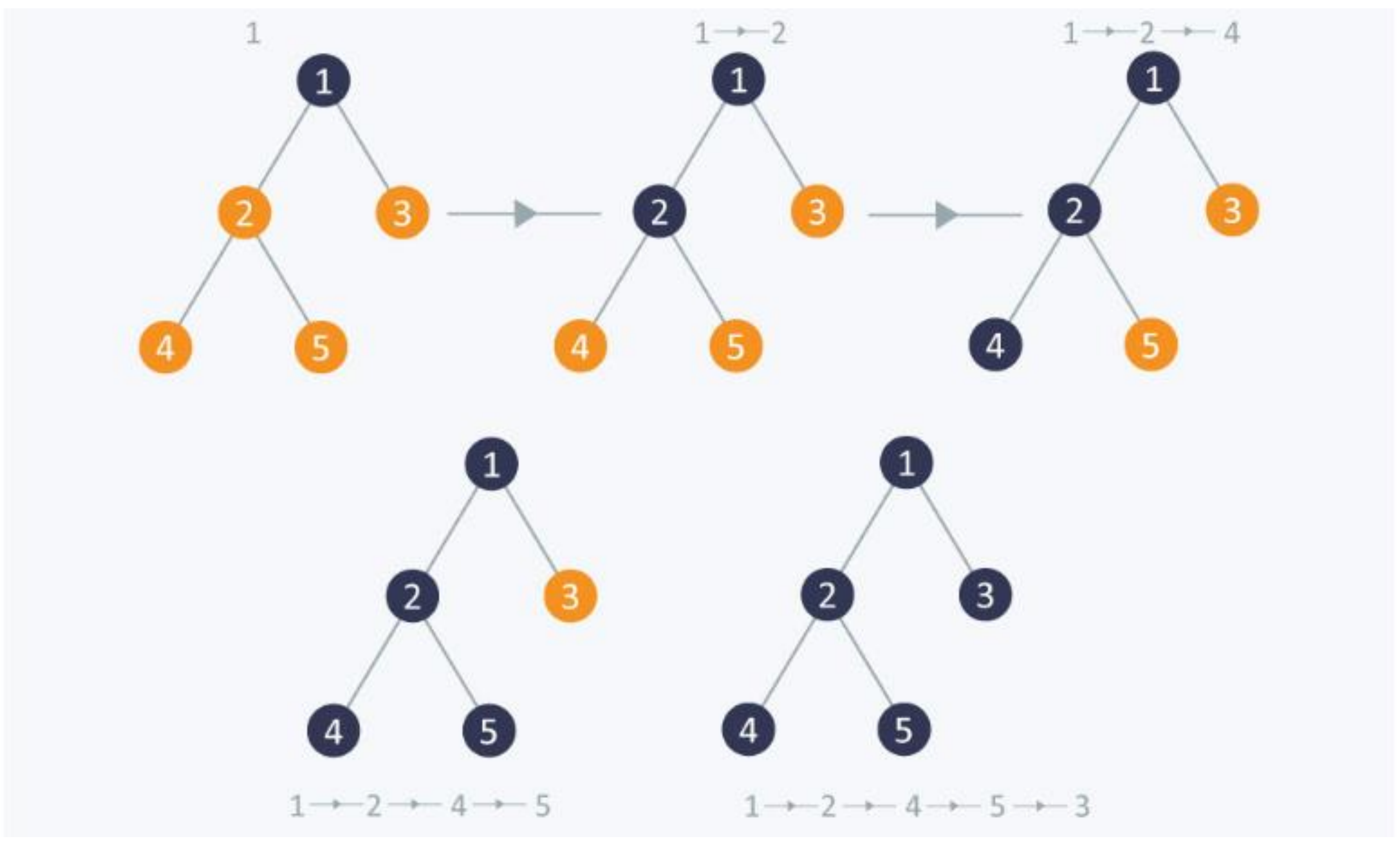
Depth first traversal algorithm

The DFS algorithm is a recursive algorithm that uses the idea of backtracking. It involves exhaustive searches of all the nodes by going ahead, if possible, else by backtracking.

This recursive nature of DFS can be implemented using stacks. The basic idea is as follows:

- Pick a starting node and push all its adjacent nodes into a stack.
- Pop a node from stack to select the next node to visit and push all its adjacent nodes into a stack.
- Repeat this process until the stack is empty. However, ensure that the nodes that are visited are marked. This will prevent you from visiting the same node more than once.

```
    DFS-iterative (G, s):                                   //Where G is graph and s is
source vertex
      let S be stack
      S.push( s )            //Inserting s in stack
      mark s as visited.
      while ( S is not empty):
          //Pop a vertex from stack to visit next
          v  =  S.top( )
         S.pop( )
         //Push all the neighbours of v in stack that are not visited
        for all neighbours w of v in Graph G:
            if w is not visited :
                     S.push( w )
                    mark w as visited


    DFS-recursive(G, s):
        mark s as visited
        for all neighbours w of s in Graph G:
            if w is not visited:
                DFS-recursive(G, w)
```


1
1→2
1→2→4
1→2→4→5
1→2→4→5→3