1. In linear runtime, for every element in an array, a monotonic stack can find the prior/next, greater/less element. It is extensively used for subarray problems where min/max are of interest.

2.  In linear runtime, for every element x[i] in an array, a monotonic stack can find the greatest element in (j, i) that is less than x[i], where x[j] is the most recent element that is no less than x[i].

3. For thinking about graphs, first construct a rectangle graph on paper: a-->b, a-->d, b-->c, c-->d.

4. Topological sorting can use DFS! Keep thinking about that!

5. Using one monotonic stack, for every element in an array, we can find which previous element is less equal (greater), and which next element is less (greater equal). This can be done in just one pass. The key is to produce the arrays of nextLess and previousLessEqual, then based on which we construct previousGreater and nextGreaterEqual.

def previousAndNext(x):  
  nextLess = [len(x)] \* len(x)  
  previousLessEqual = [-1] \* len(x)  
  sk = []  
  for i in range(len(x)):  
    while sk and sk[-1][0] > x[i]:  
      val, ind = sk.pop()  
      nextLess[ind] = i  
    previousLessEqual[i] = sk[-1][1]  
    sk.append((x[i], i))  
  previousGreater = [-1] \* len(x)  
  nextGreaterEqual = [len(x)] \* len(x)  
  for i in range(len(x)):  
    if nextLess[i] >= 0 and nextLess[i] < len(x):  
      previousGreater[nextLess[i]] = i  
    if previousLessEqual[i] >= 0 and previousLessEqual[i] < len(x):  
      nextGreaterEqual[previousLessEqual[i]] = i  
  return previousLessEqual, previousGreater, nextLess, nextGreaterEqual

6.  template that can solve most 'substring' problems: <https://leetcode.com/problems/minimum-window-substring/discuss/26808/Here-is-a-10-line-template-that-can-solve-most-'substring'-problems>

General idea is to use two pointers assisted by a hash table that can disqualify substrings within the pointers.

7. GCD, just like the min/max operator, is **associative** and **commutative**! Meaning, gcd(x1, x2, x3) = gcd(gcd(x1, x2), x3). The function is also non-increasing.

8. A better way of generating C(N, k) using DFS: consider the binary mapping of the subset. Traverse the binary tree instead of the multinary tree. The same idea applies to traverse all subsets of all sizes. For problem solving, always thinking in binary trees. E.g.

*S = []  
x = [0, 1, 2, 4, 8, 16, 32, 64, 128, 256]  
subseqLen = 5  
y = []  
def f(position, remainSlots):  
  if remainSlots <= 0: S.append([u for u in y]); return  
  if position >= len(x): return  
  y.append(x[position])  
  f(position + 1, remainSlots - 1)  
  y.pop()  
  f(position + 1, remainSlots)  
f(0, subseqLen)  
print(S)*

9. Dijkstra's algorithm's worst case time complexity is O(E + Vlog(V)). For the BFS version of finding shortest path listed *here:*[*https://cs.stackexchange.com/questions/154983/whats-the-average-and-worst-case-time-complexities-of-the-following-bfs-for-fin*](https://cs.stackexchange.com/questions/154983/whats-the-average-and-worst-case-time-complexities-of-the-following-bfs-for-fin)which I often code, the worst time complexity is probably O(E + V^2).

For A\* search, the key is to estimate the potential of neighboring nodes. It is similar to Dijkstra's but the priority queue compares the node potential. In a grid world/real world, the potential is often the current path length plus the L1/L2 distance between the current point and the sink point. The potential MUST be no more than the actual path length.

For multi-objective shortest path such as finding the shortest path with a constraint of k-hops: [problem](https://leetcode.com/problems/shortest-path-in-a-grid-with-obstacles-elimination/solution/) or [cheapest-flights-within-k-stops](https://leetcode.com/problems/cheapest-flights-within-k-stops/) , a general approach is to store all potential multi-objective function values. See my own submissions for those problems. The key is to update children's best current multi-objectives instead of a single value, i.e., the distance.

10: Deeper understanding about the shortest path and BFS. Some problems require considering **multi-dimensional state space**. When the states (the full graph) are unknown beforehand and need to be computed on the fly during propagation, the "if updated, push into new frontier" tactic for shortest path will not work, because you'll never know when to stop. In this case, only the priority-queue, Dijkstra approach can work.  Review [race-car](https://leetcode.com/problems/race-car/).

11. Doubly linked hash table is intriguing. It can find an element in O(1) on average and also maintains the order of elements when they were added.

12. Given n labeled tree leaves where only neighboring leaves can be aggregated, count the number of rooted full binary trees (every node has either 0 or 2 children): s(n) = s(1) \* s(n - 1) + ... + s(k) \* s(n - k) + ... + s(n - 1) \* s(1).

Given n unlabeled tree leaves, count the number of non-isomorphic rooted full binary trees:

if n is odd:  t(n) = t(1) \* t(n - 1) + ... + t(n // 2) \* t(n - n // 2)

if n is even:  t(n) = t(1) \* t(n - 1) + ... + t(n // 2 - 1) \* t(n // 2 + 1) + Choose{ t(n//2), 2 } + t(n//2)

When n is even, we need to first select two different structures from t(n//2), and every structure in t(n//2) paring itself is also a distinctive member.

Given n labeled tree leaves where every two leaves can be aggregated, count the number of rooted full binary trees:

u(n) = t(n) \* A{n, n}

On-Line Encyclopedia of Integer Sequences!

13. You finally get a taste of the power of std::queue! Now, to maintain order while constantly changing element values, or erasing/inserting elements, the speedup against std::vector is about 6~7x. Try std::queue first! std::deque has a bucket size of **at least** 512 bytes! This has been checked by myself.