Algorithms

for Coding Interviews
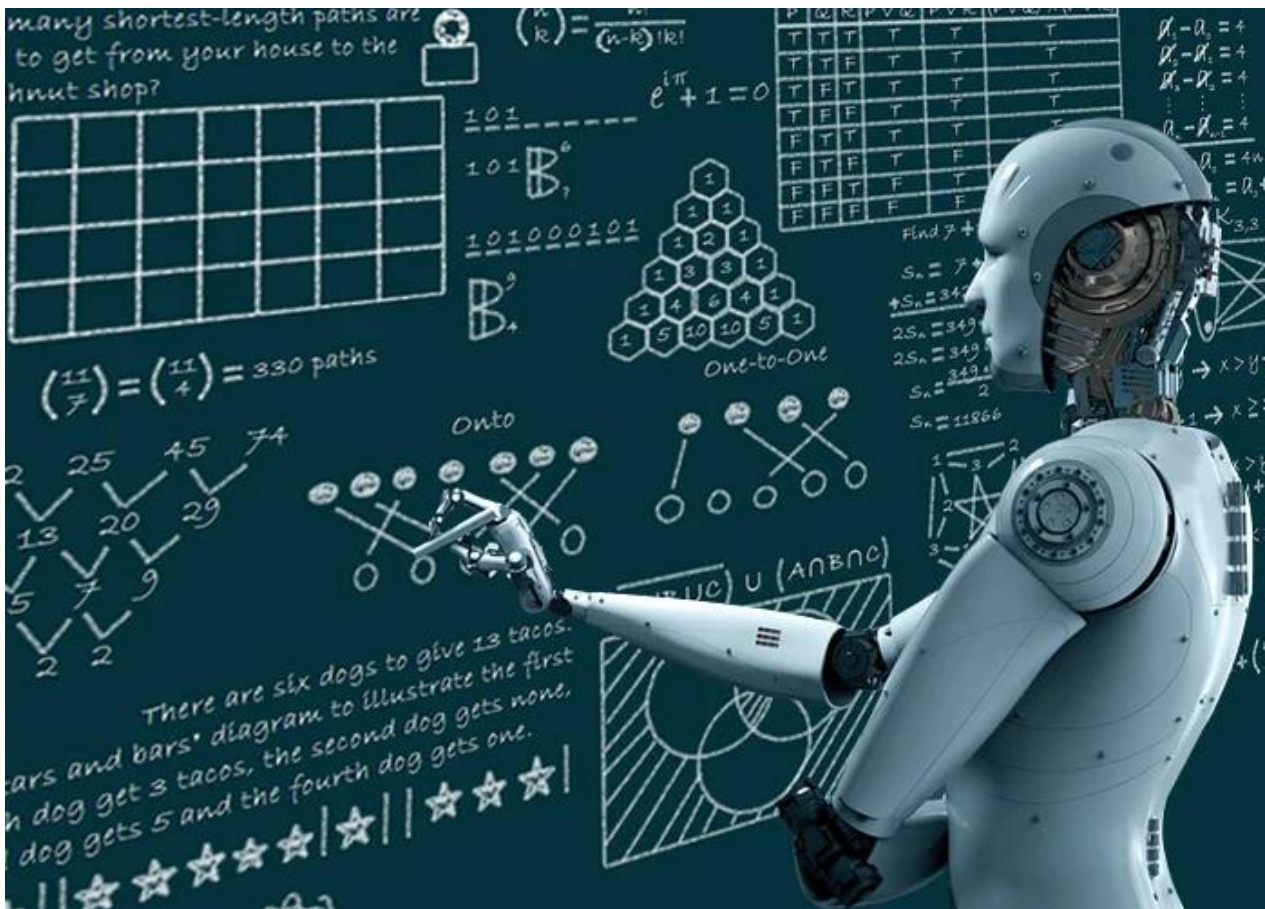
DRAFT VERSION

(Version 19/03/2023)

Curated By:

Ivancich Stefano

CONTENTS

Stuff to add:

- https://workflowy.com/s/study-guide/RD5kZ682pWX5oxiE
- Backtraking: https://leetcode.com/explore/learn/card/recursion-ii/472/backtracking/2793/ https://www.programiz.com/dsa/backtracking-algorithm
- https://leetcode.com/discuss/general-discussion/458695/dynamic-programming-patterns/680370
- Graph algorithms (Leetcode tutorial section, Ricerca operativa, Johnson's algorithm, Bellman, disjoint set, ford fulkerston)
- Union find Goodrich python book: tree implementation
- Pattern matching, aho corasik, Rabin-Karp Substring Search CTCI page 636, Z-algorithm
- Sliding windows
- 2 pointers: https://leetcode.com/discuss/study-guide/1688903/solved-all-two-pointers-problems-in-100-days
- Space complexity of sorting algos
- Paradigms: Branch and bound, Brute-force search
- Aggiungi le cose scritte nella sezione greedy
- Maps Goodrich: Tree Map, Sorted Map, Multiset, Multimap, SkipList
- https://towardsdatascience.com/self-balancing-binary-search-trees-101-fc4f51199e1d
- https://guides.codepath.com/compsci/UMPIRE-Interview-Strategy
- Read Competitive Programming Book
- Read Competitive Programming HandBook
- AVL trees CTCI page 637, https://www.programiz.com/dsa/avl-tree
- Red-Black Trees CTCI page 639
- Interval Trees
- Principle of Inclusion/exclusion
- Number theory: https://app.codesignal.com/interview-practice/topics/number-theory/tutorial
- Implementa un heap tua (copia da goodrich) in cui ogni nodo è linkato da un ht, e prova a eliminare un elemento e poi mantenere la heap order property. Oppure siccome viene utilizzato anche in heapq prova a capire come mantenere la heap order property. Heap deletion: Link, dr kattis problem.
- **OOP**:
  - https://www.educative.io/courses/grokking-the-object-oriented-design-interview
  - https://www.udemy.com/course/design-patterns-python/
  - https://tedweishiwang.github.io/journal/object-oriented-design-elevator.html

- o https://neetcode.io/courses/lessons/8-design-patterns
- Solid principles:
  - o https://www.educative.io/blog/solid-principles-oop-c-sharp
  - o www.freecodecamp.org/news/solid-principles-explained-in-plain-english/
  - o https://tedweishiwang.github.io/journal/object-oriented-design-elevator.html
- Unit Test
- Clean Code
- REST
- Ci cd
- SO   https://docs.google.com/document/d/191a-PHHLAH1I33-G3X_adOksNK0OKFiNPDMrnHhZdco/edit?usp=sharing
- DataBase https://drive.google.com/file/d/1DQ_eiQZuuF8hKbcGkhREjwdTh1MCrl-g/view https://files.jrebel.com/pdf/zt_sql_cheat_sheet.pdf
- …

1. HOW TO PREPARE FOR CODING INTERVIEWS

Use LEETCODE.
WHY LEETCODE IS BETTER than other (free) platforms (as 2023):
- Problems were almost all asked in real interviews. No competitive programming o theoretical problems.
- Most of the problems have fully explained solutions.
- For each problem there is a well-organized sub-section with a very active community.
- Hackerrank, codility, … have too long problem statements or don't have clear solutions.
- Codeforces, Codechef, kattis, … are for competitive programming.
- Most of the problems (probably all) on books like CTCI and EPI are on Leetcode. (Use those books just to learn the theory, don't waste time on doing problem on paper, search them on leetcode)

Leetcode alternatives:
- Hackerrank (used also for online assessment, contains also other programming topics)
- Codesignal (used also for online assessment)
- Codility (used also for online assessment)
- HireVue (used for online assessment)
- binarysearch.com (has shorter problem statements)
- CodeForces (Competitive programming)
- Kattis (Competitive programming)
- UsacoGuide (Competitive programming)

How to prepare:
- Read the first 86 pages of CTCI that explains how interviews works.
- Watch a couple of videos on YouTube "*Google coding interview*" to get familiar with the context of coding interviews. (Do not watch too old videos, the interviews slightly change over time). Example
- For each chapter of CTCI read the 2-3 theory pages (or theory here) then do exercises on the respective leetcode.com/explore/learn/ section. The optimal topic order for study should be this: Array, Hash table, Linked List, Stack & Queue, N-Trees, Binary Tree, Binary search trees, Trie, Heaps, Sorting, Searching, Recursion 1, Divide and Conquer (Recursion 2), Backtracking (Recursion 2), Dynamic Programming, Greedy, Graphs, Bit Manipulation, Math problems (combinations, permutation, prime numbers, probability, ...). You can also use the order proposed in neetcode.io/roadmap
- Then do problems on the problem sets (Grind75, Easy, Medium, Hard, Blind75, Google, …).
- Do mock interviews (pramp.com)
- (Optional) Learn more theory on books: Goodrich (in Python, Java or C++), Competitive Programming Handbook, Cormen, CP4.
- If you want to pay for the theory (not really needed), we suggest the following courses: Leetcode DS&A crash course, Neetcode DSA for beginners, Neetcode Advanced

General advices:
- Do problems on topics that you struggle at. Is not useful to do problems on topics you are already good.

- Do problems slightly above your level.
- Be constant: Solve at least ONE problem a day. (Do something like Leetcode Daily challenge)
- If a problem has too many dislikes, carefully read the problem statement and comments before solving it to avoid wasting time in useless or too harsh problem.
- Once you solved the problem, look at the discussion section and read other people solutions.
- Easy problems often are too easy. On the long run try to solve mostly mediums and hard.
- Use a timer. Leetcode has an embedded timer.
- Do problems on the interview lists of the company you want to join more than once.
- Participate in Leetcode contests to become comfortable solving problems under pressure.
- Do a mock interview AT LEAST once a week and recreate as much as possible the interview environment (clothing, light, papers, materials, whiteboard, …) to become comfortable.
- Use just 1 or 2 platforms to prepare, otherwise you will end up doing all easy problems in a platform, change to another platform, do all the easy problems, then switch again and so on. You will end up doing just easy problems. Do a couple of problems on other platforms just to become comfortable with the platform that is going to be used for online assessment (e.g. hackerrank, codility, Codesignal) and interviews (e.g. hirevue, coderpad).
- Write down the most interesting problems and techniques that you encounter. In general, medium problems are a composition of easy problems/techniques and hard problems are composition of medium problems/techniques.
- Do yourself a favor and take notes of the topics you are studying. During this time, you will be covering so many different subjects and tricks, and being a human being guarantees that you will forget the majority of them, so take notes and review them once in a while. Your notes are also a valuable resource for the next time that you are preparing for the interviews.

If you cannot solve a problem:
- Spend no more than 20-30 minutes without making progress. Just go look up the answer. Contrary to popular belief, most struggling past 30 minutes is pointless.
- Read just a little part of the solution and try to go on yourself. If you struggle, read a little bit more and so on, until you really cannot solve yourself.
- Always implement it yourself, don't just read the solution and copy and paste the code.
- Always mark it as something you need to try again. Wait at least a day and try to solve it fresh. If you fail, repeat at infinitum.

Aim to solve:
- 60 Easy problems (<10min)
- 100 Medium problems (<15min)
- 40 Hard problems (<25min)
- Ratio 3:5:2 (easy:medium:hard)

Important Problem Sets:
- https://www.techinterviewhandbook.org/grind75
- https://neetcode.io/roadmap
- Leetcode Explore -> Interview -> Easy Collection
- Leetcode Explore -> Interview -> Medium Collection
- Leetcode Explore -> Interview -> Hard Collection
- Leetcode BLIND 75 questions

Other problem sets:
- https://seanprashad.com/leetcode-patterns/
- Leetcode Curated Algo 170 (premium)
- Leetcode Explore -> Interview -> Specific company Collection (Premium)
- Leetcode Mock Interview online assessment, phone, … (Premium)
- Leetcode Discussion: Microsoft-Online-Assessment-Questions
- Leetcode Discussion: Google-Online-Assessment-Questions
- Leetcode Discussion: Amazon-Online-Assessment-Questions
- Dynamic Programing
- CSES problem set (Competitive programming)

Mock Interviews:
- pramp.com (completely free, also for behavioral, system design, frontend, …)
- interviewing.io (paid) https://www.youtube.com/c/interviewingio/videos

2. How to Solve Problems

2.1. Build an algorithm (L.E.BR.O.W.I.T.)

1) Listen (A.W.R.)
- Pay very close attention to any info in the problem description. You probably need it all for an optimal algorithm.
- **A**sk questions about anything you're unsure about. Always ask questions, they usually don't give you some information on purpose.
- **W**rite the pertinent information on the whiteboard.
- **R**epeat the problem statement to the interviewer to be sure that you understood it correctly.
- Ask if the input is correct or we should do some sanity check.

2) Draw an Example (La.NotSpec)
- Specific. It should use real numbers or strings (if applicable to the problem).
- Sufficiently **La**rge. Most examples are too small or are special cases.
- **Not** a **Spec**ial case.
- Ask to the interviewer if he can give you an example.

3) State a Brute Force
Get a brute-force solution as soon as possible. It's okay that this initial solution is terrible. Explain what the space and time complexity is, and then dive into improvements.
It's a starting point for optimizations, and it helps you wrap your head around the problem.
Ask to the interviewer if you can code, or if you can optimize.

4) Optimize
Walk through your brute force with **BUD** optimization or try some of these ideas:
- Look for any **unused info**. You usually need all the information in a problem.
- Solve it manually on an example, then **reverse engineer your thought process**. How did you solve it?
- Solve it "incorrectly" and then think about why the algorithm fails. Can you fix those issues?
- Make a time vs space tradeoff. Sometimes storing extra state about the problem can help you optimize the runtime.
- Precompute information. Is there a way that you can reorganize the data (sorting, etc.) or compute some values upfront that will help save time in the long run?
- Use a hash table.
- Think about the best conceivable runtime.

5) Walk Through
Now that you have an optimal solution, walk through your approach in detail. Make sure you understand each detail before you start coding. Know what the variables are and when they change. Test the solution on a few examples.

6) Implement
Write beautiful code. Modularize your code from the beginning and refactor to clean up anything that is not beautiful. **Unit test** critical lines of code before continue writing.

7) Test: in this order:
- **Conceptual test.** Read and analyze what each line of code does. Does the code do what you think it should do?
- **Unusual** or non-standard code.
- **Hot spots.** Base cases in recursive code. Integer division. Null nodes in binary trees. The start and end of iteration through a linked list. Double check that stuff.
- **Small test cases.** Use a 3 or 4 element array.
- **Normal test cases**
- **Special cases and edge cases.** Test your code against null or single element values, the extreme cases, and other special cases.

When you find bugs carefully analyze why the bug occurred and ensure that your fix is the best one.

2.2. Optimization techniques

BUD
Things that an algorithm can "waste" time doing.
When you find one of them, you can then focus on getting rid of it.
If it's still not optimal, you can repeat this approach on your current best algorithm.
Bottlenecks: Part of your algorithm that slows down the overall runtime in two common ways:
- One-time work that slows down your algorithm.
- A chunk of work that's done repeatedly, like searching. Perhaps you can reduce that from $O(n)$ to $O(\log n)$ or even $O(1)$.

Unnecessary Work, Duplicated Work

DIY (Do It Yourself)
When you get a question, try just working it through intuitively on a real example. Often a bigger example will be easier.
Use a nice, big example and intuitively-manually solve it for the specific example. Then, afterwards, think hard about how you solved it. Reverse engineer your own approach.
Be particularly aware of any "optimizations" you intuitively or automatically made.

Simplify and Generalize
- First, we simplify or tweak some constraint, such as the data type.
- Then, we solve this new simplified version of the problem.
- Finally, once we have an algorithm for the simplified problem, we try to adapt it for the more complex version.

Base Case and Build
Solve the problem first for a base case (e.g., n = 1) and then try to build up from there.
When we get to more complex/interesting cases (often n = 3 or n = 4), we try to build those using the prior solutions.
Often lead to natural recursive algorithms.

Data Structure Brainstorm
Run through a list of data structures and try to apply each one.
More problems you do, the more developed your instinct on which data structure to apply will be.

2.3. Other stuff

Best Conceivable Runtime (BCR)
Is the runtime you know you can't beat. For example, if asked to compute the intersection of two sets, you know you can't beat $O(|A| + |B|)$.
BCR is not necessarily achievable. It says only that you can't do better than it.
BCR can be useful. We can use the runtimes to get a "hint" for what we need to reduce.
Trick: if there is some information about the size of the input you can infer the maximum acceptable runtime, look at the **complexity limit table** at page 8. But don't tell this to the interviewer. (shhh)

Handling Incorrect Answers
Interview It's about how optimal your final solution was, how long it took you to get there, how much help you needed, and how clean was your code.
Your performance is evaluated in comparison to other candidates.
Many questions are too difficult to expect even a strong candidate to immediately spit out the optimal algorithm.

When You've Heard a Question Before
If you've heard a question before, admit this to your interviewer.
Your interviewer may find it highly dishonest if you don't reveal that you know the question.

What Good Coding Looks Like:
- **Correct:** The code should operate correctly on all expected and unexpected inputs.
- **Efficient:** both time and space. Both the asymptotic and the practical, real-life efficiency.
- **Simple:** 10 lines instead of 100
- **Readable**
- **Maintainable**
- **Modular:** separating isolated chunks of code out into their own methods. So it's easily testable because each component can be verified separately.
- **Flexible and Robust:** using variables instead of hard-coded values or using templates / generics to solve a problem. If we can write our code to solve a more general problem, we should. If the solution is much more complex for the general case, and it seems unnecessary at this point in time, it may be better just to implement the simple, expected case.
- **Error Checking:** validates that the input is what it should be, either through ASSERT statements or if-statements. Writing these can be tedious and can waste precious time in an interview. The important thing is to point out that you would write the checks. If the error checks are much more than a quick if-statement, it may be best to leave some space where the error checks would go and indicate to your interviewer that you'll fill them in when you're finished with the rest of the code.

Do not…
- Do not ignore information given. Info is there for a reason.
- Do not try to solve problems in your head. Use an example!
- Do not push through code when confused. Stop and think!
- Do not dive into code without interviewer "sign off."

If the optimal solution it's too long to write, tell them. Sometimes they make it on purpose.

2.4. Whiteboard Templates

Python Template

```
"""
#### RELEVANT INFO ####
...

#### EXAMPLES #########

Simple
....

Bigger
....

Edge Case
....


#### BRUTE FORCE ######
......


#### OPTIMIZE #########
....


#### TEST #############
Normal Cases:
    - ...

Edge Cases:
    - ...

---------------------------------------
APPROACH 1: Time: O(...), Space: O(...)
Idea: ...
1. .....    Time: O(...), Space: O(...)
2. .....    Time: O(...), Space: O(...)

APPROACH 2: Time: O(...), Space: O(...)
Idea: ...
1. .....

"""
```

3. Big O

3.1. Basics
Time Complexity
- O (big O): upper bound on the time.
- Ω (big omega): lower bound.
- Θ (big theta): when $O = \Omega$

In industry (and therefore in interviews), people seem to have merged O and Θ together, meaning O is the tightest runtime.

Best, worst, and expected cases describe the big O (or big theta) time for particular inputs or scenarios.
Big O, big omega, and big theta describe the upper, lower, and tight bounds for the runtime.

Space Complexity: Memory, array and Stack space used in recursive calls.
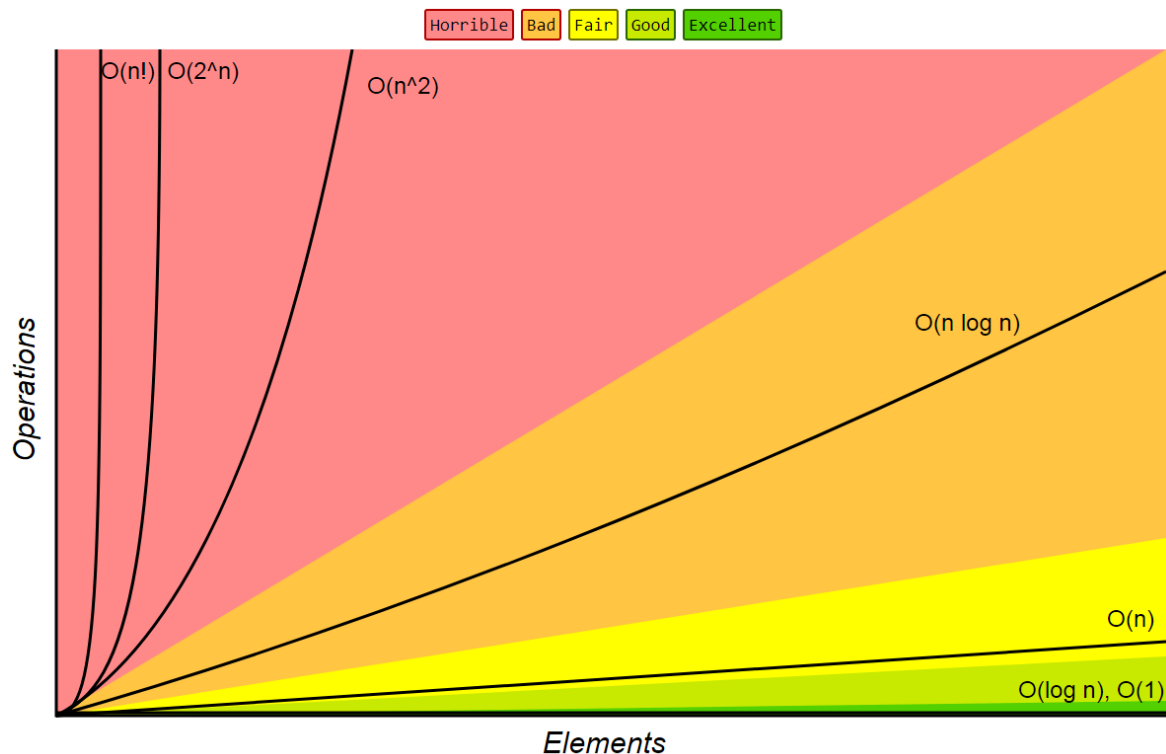
Drop the Constants
It is very possible for $O(N)$ code to run faster than $O(1)$ code for specific inputs. Big O just describes the rate of increase.
For this reason, we drop the constants in runtime. An algorithm that one might have described as $O(2N)$ is actually $O(N)$.

Drop the Non-Dominant Terms:

$$O(1) < O(\log n) < O(n) < O(n \log n) = O(\log n!) < O(n^2) < O(2^n) < O(n!)$$

3.2. Calculate Big O

Multi-Part Algorithms: if are Nested multiply. If are in sequence add.

Amortized Time: allows us to describe that the worst case happens every once in a while. But once it happens, it won't happen again for so long that the cost is "amortized'.

Log N Runtimes: When you see a problem where the number of elements in the problem space gets halved each time, that will likely be a $O(\log n)$ runtime.
The base of the log doesn't matter for the purposes of big O.

Recursive Runtimes: When you have a recursive function that makes multiple calls, the runtime will often (but not always) look like $\boldsymbol{O}\big(\text{branches}^{\text{depth}}\big)$, where branches is the number of times each recursive call branches.
For each invocation of the function, we only account for the number of operations that are performed within the body of that activation. Then taking the sum, over all activations, of the number of operations that take place during each individual activation.
Rely on the intuition afforded by a **recursion trace** in recognizing how many recursive activations occur, and how the parameterization of each activation can be used to estimate the number of primitive operations that occur within the body of that activation.
Linear recursion starts at most one other.
Binary recursion starts two others.
Multiple recursion starts three or more others.
  - function is being called recursively n times before reaching base case so its $O(n)$
  - for every time we divide by 5 before calling the function so its $O(\log n)$ (base 5)
  - $O(2^n)$ or exponential, since each function call calls itself twice unless it has been recursed n times.
  - $O(2^n)$ loop that starts by performing one operation and then doubles the number of operations performed with each iteration.

Memoization: is a very common technique to optimize exponential time recursive algorithms.

Think about how the runtime changes as n gets bigger.

$$1 + 2 + \cdots + n = \frac{n(n+1)}{2} = O(n^2)$$

$1 + a + .. + a^n = \frac{a^{n+1}-1}{a-1}$ loop for which each iteration takes a multiplicative factor longer than the previous one. If $a = 2$ then $= 2^n - 1$

$$1^2 + 2^2 + \cdots + n^2 = \frac{n(n+1)(2n+1)}{6}$$

Python typical **max recursion** value **is 1000**
```
import sys
old = sys.getrecursionlimit() # perhaps 1000 is typical
sys.setrecursionlimit(1000000) # change to allow 1 million nested calls
```

3.3. Common Data Structure Operations

Data Structure	Time Complexity								Space Complexity
	Average				Worst				Worst
	Access	Search	Insertion	Deletion	Access	Search	Insertion	Deletion	
Array	Θ(1)	Θ(n)	Θ(n)	Θ(n)	O(1)	O(n)	O(n)	O(n)	O(n)
Stack	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Queue	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Singly-Linked List	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Doubly-Linked List	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Skip List	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(n)	O(n)	O(n)	O(n)	O(n log(n))
Hash Table	N/A	Θ(1)	Θ(1)	Θ(1)	N/A	O(n)	O(n)	O(n)	O(n)
Binary Search Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(n)	O(n)	O(n)	O(n)	O(n)
Cartesian Tree	N/A	Θ(log(n))	Θ(log(n))	Θ(log(n))	N/A	O(n)	O(n)	O(n)	O(n)
B-Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
Red-Black Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
Splay Tree	N/A	Θ(log(n))	Θ(log(n))	Θ(log(n))	N/A	O(log(n))	O(log(n))	O(log(n))	O(n)
AVL Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
KD Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(n)	O(n)	O(n)	O(n)	O(n)

3.4. Array Sorting Algorithms

Algorithm	Time Complexity			Space Complexity
	Best	Average	Worst	Worst
Quicksort	Ω(n log(n))	Θ(n log(n))	O(n^2)	O(log(n))
Mergesort	Ω(n log(n))	Θ(n log(n))	O(n log(n))	O(n)
Timsort	Ω(n)	Θ(n log(n))	O(n log(n))	O(n)
Heapsort	Ω(n log(n))	Θ(n log(n))	O(n log(n))	O(1)
Bubble Sort	Ω(n)	Θ(n^2)	O(n^2)	O(1)
Insertion Sort	Ω(n)	Θ(n^2)	O(n^2)	O(1)
Selection Sort	Ω(n^2)	Θ(n^2)	O(n^2)	O(1)
Tree Sort	Ω(n log(n))	Θ(n log(n))	O(n^2)	O(n)
Shell Sort	Ω(n log(n))	Θ(n(log(n))^2)	O(n(log(n))^2)	O(1)
Bucket Sort	Ω(n+k)	Θ(n+k)	O(n^2)	O(n)
Radix Sort	Ω(nk)	Θ(nk)	O(nk)	O(n+k)
Counting Sort	Ω(n+k)	Θ(n+k)	O(n+k)	O(k)
Cubesort	Ω(n)	Θ(n log(n))	O(n log(n))	O(n)

Complexity Limits: A typical year 2020 CPU can process $100M = 10^8$ operations in 1 second.

n	Complexity	Example
$\leq 10..11$	$O(n!), O(n^6)$	Permutations
$\leq 15..18$	$O(2^n \times n^2)$	DP TSP
$\leq 18..22$	$O(2^n \times n)$	DP with bitmask
≤ 25	$O(2^n)$	Try 2^n possibilities with $O(1)$ check each
≤ 100	$O(n^4)$	Combinations with $k = 4$
≤ 400	$O(n^3)$	Floyd Warshall
≤ 1000	$O(n^2 \times \log_2 n^2)$	
≤ 2000	$O(n^2 \times \log_2 n)$	2 nested loops + tree DS or heap
≤ 5000	$O(n^2)$	2 nested loops
$\leq 10^5$	$O(n \times \log_2 n)$	Mergesort
$\leq 5 * 10^6$	$O(n)$	
$\leq 10^{12}$	$O(\sqrt{n})$	Primality check, Number of factors
$\leq 10^{18}$	$O(\log_2 n)$	Input bottleneck usually $n \leq 1M = 10^6$

Sometimes a constant is worse than a log: The Number of atoms in the universe is 10^{81}, $\log_2(10^{81}) \approx 269$, so a constant higher than 269 is worse than the worst logarithm possible. But remember that a logarithm code has its own constants, so the complexity it's actually higher. This brief reasoning just wants to tell you to be careful when using very high constants.

4. Arrays

Since array problems are quite easy this section only contains advanced techniques.

Array Edge cases: size=0, 1, 2, 3, even, odd

2 pointer techniques:
- one starting at head, one at tail
- sometimes need to sort the array before using the two-pointer technique.
- might need a greedy thought to determine your movement strategy.
- Usually can remove a n grade of complexity

Sliding window technique
- **Fixed window length k:** find something in the window such as the maximum sum of all windows, the maximum or median number of each window.
  Usually, we need kind of variables to maintain the state of the window, some are as simple as an integer or it could be as complicated as some advanced data structure such as list, queue or deque.
- **Two pointers + criteria:** the window size is not fixed, usually it asks you to find the subarray (size) that meets the criteria.
  1 pointer ahead.

The minimum Average of an array is on its sub-arrays:
- Given an array $A = [a_1, \ldots, a_n]$
- The average of A and its subarrays: $B = \frac{a_1, \ldots, a_n}{n}, C = \frac{a_1, \ldots, a_i}{i}, D = \frac{a_{i+1}, \ldots, a_n}{n-i}$
- It holds that $C \leq B$ or $D \leq B$

Range sum problem: given an array, repeatedly find the range sum between index $[i, j]$
Solution: $O(n)$ computing prefix array, $O(1)$ computing range sum
- Subarray sum (prefix): subarray[L..R]=pref[R]-pref[L-1]
- Subarray sum (suffix): sum from right to left

Range updates: we want update ranges and retrieve single values.
Solution: We build a difference array whose values indicate the differences between consecutive values in the original array. Thus, the original array is the prefix sum array of the difference array.

0	1	2	3	4	5	6	7
3	3	1	1	1	5	2	2

0	1	2	3	4	5	6	7
3	0	-2	0	0	4	-3	0

The advantage of the difference array is that we can update a range in the original array by changing just two elements in the difference array.
To increase the values in range $[a, b]$ by x, we increase the value at position a by x and decrease the value at position $b + 1$ by x.
Thus, it is only needed to update single values and process sum queries, so we can use a binary indexed tree or a segment tree.

For example, if we want to increase the original array values between positions 1 and 4 by 5, it suffices to increase the difference array value at position 1 by 5 and decrease the value at position 5 by 5. The result is as follows:

| 0	1	2	3	4	5	6	7
3	5	-2	0	0	-1	-3	0

Famous Problems

Count repeated elements in an array with [1,n]: change sign to the element at index=current element val

Max subarray sum (kadane)
```python
def maximumSum(self, arr: List[int]) -> int:
    n = len(arr)
    max_ending_here = n * [arr[0]]
    for i in range(1, n):
        max_ending_here[i] = max(max_ending_here[i-1] + arr[i], arr[i])
    return max(max_ending_here)
```

5. Hash Tables

Bucket array: each bucket may manage a collection of items that are sent to a specific index. We store the item (k, v) in the bucket $A[h(k)]$.
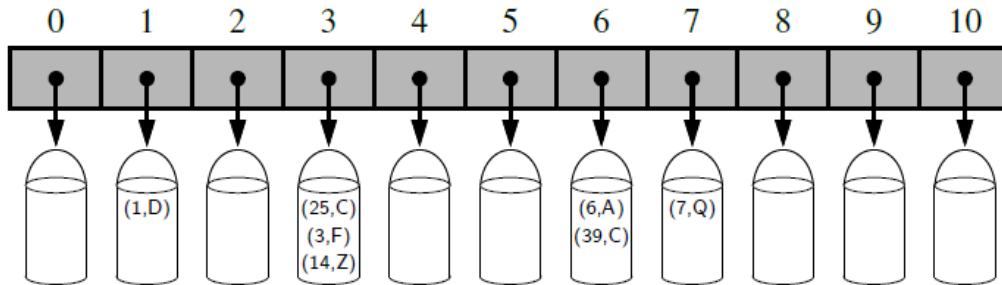

Hash Function: map general keys to corresponding indices in a table. Ideally, keys will be well distributed in the range from $[0, N - 1]$.

Collision: when there are two or more keys with the same hash value.

The hash function is composed by:
- **Hash code** that maps a key k to an integer
- **Compression function** that maps the hash code to an integer within a range of indices, $[0, N - 1]$, for a bucket array.
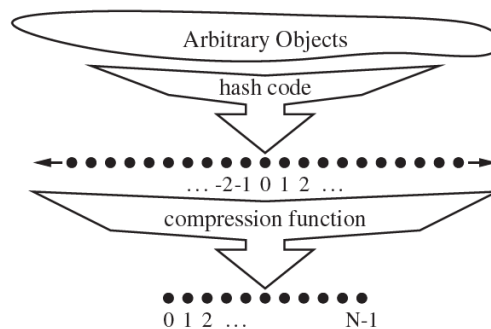
The advantage of separating the hash function into two such components is that the hash code portion of that computation is independent of a specific hash table size.

Hashing Algorithms:
- **Numeric keys**: $\text{hash}(key) = key \bmod n$
- **String keys** of length n: polynomial rolling hash function: $\text{hash}(s) = \left(\sum_{i=0}^{n-1} s[i] * p^i \right) \bmod m$
  where:
  - p a prime number roughly equal to the number of characters in the input alphabet.
  - m large prime number ($m = 10^9 + 9$ or $10^9 + 7$)

Compression Functions
- Division Method: $i \bmod N$ where N is prime and far from a power of 2
- MAD: $[(ai + b) \bmod p] \bmod N$ where p prime $> N$, a, b random integers $[0, p - 1], a > 0$

Separate chaining most commonly used collision resolution techniques. It is usually implemented using linked lists. Each element of the hash table is a linked list.

Load factor $\lambda = \frac{n}{N}$, where n is the number of objects to insert, N is the size of the bucket array.

Should maintain $\lambda < 0.9$

If an insertion causes the load factor of a hash table to go above the specified threshold, then it is common to resize the table and to reinsert all objects into this new table.

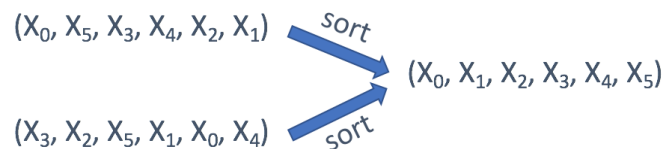| Operation	List	Hash Table	
		expected	worst case
__getitem__	$O(n)$	$O(1)$	$O(n)$
__setitem__	$O(n)$	$O(1)$	$O(n)$
__delitem__	$O(n)$	$O(1)$	$O(n)$
__len__	$O(1)$	$O(1)$	$O(1)$
__iter__	$O(n)$	$O(n)$	$O(n)$

Design the Key

Sometimes you have to think it over to design a suitable key when using a hash table.

Designing a key is to build a mapping relationship by yourself between the original information and the actual key used by hash map. You need to guarantee that:

- All values belong to the same group will be mapped in the same group.
- Values which needed to be separated into different groups will not be mapped into the same group.

Common:

- When the order of each element in the string/array doesn't matter, you can use the **sorted string/array** as the key.

$$(X_0, X_5, X_3, X_4, X_2, X_1) \xrightarrow{sort}$$
$$(X_3, X_2, X_5, X_1, X_0, X_4) \xrightarrow{sort} (X_0, X_1, X_2, X_3, X_4, X_5)$$

- If you only care about the offset of each value, usually the offset from the first value, you can use the **offset** as the key.

$$(X_0, X_1, X_2, X_3, X_4, X_5) \xrightarrow{offset} (0, X_1 - X_0, X_2 - X_0, X_3 - X_0, X_4 - X_0, X_5 - X_0)$$
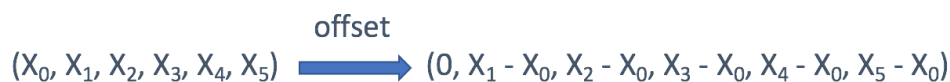
- In a tree, you might want to directly use the TreeNode as key sometimes. But in most cases, the **serialization of the subtree** might be a better idea.
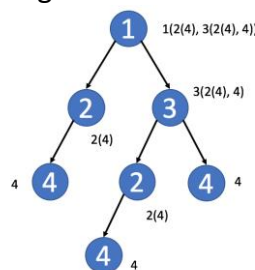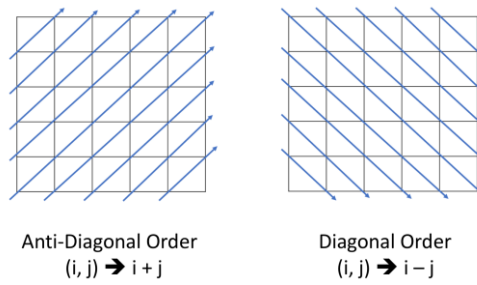

- In a matrix, you might want to use the row index or the column index as key.
- In a Sudoku, you can combine the row index and the column index to identify which block this element belongs to.

(i, j) ➔ (i / 3) * 3+ j / 3

15

- Sometimes, in a matrix, you might want to aggregate the values in the same diagonal line

Anti-Diagonal Order
(i, j) ➜ i + j

Diagonal Order
(i, j) ➜ i − j

Implementation with Python `dict` class:

Python's `dict` class is implemented with hashing.

Operation	Average Case	Amortized Worst Case
Copy[2]	O(n)	O(n)
Get Item	O(1)	O(n)
Set Item[1]	O(1)	O(n)
Delete Item	O(1)	O(n)
Iteration[2]	O(n)	O(n)

from collections import defaultdict
    self.d = defaultdict(int)
collection.Counter………………….

Implementation with Python:

Hash code: `hash(x)` returns an integer value, works for only immutable data types (int, float, str, tuple, and frozenset).

User-defined classes are treated as unhashable by default, to implement use __hash__

pag 422 Goodrich python………………..
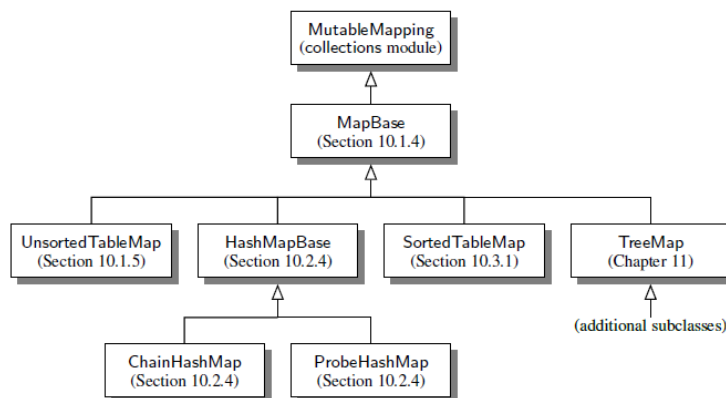
6. Linked Lists

Node maintains a reference to its element and one or more references to neighboring nodes in order to collectively represent the linear order of the sequence.

Elements of a linked list cannot be efficiently accessed by a numeric index k, and we cannot tell just by examining a node if it is the second, fifth, or twentieth node in the list.

Advantages of Array-Based Sequences
- $O(1)$-time access to an element based on an integer index. in a linked list requires $O(k)$
- $O(1)$ operations typically are more efficient.
- typically use proportionally less memory than linked structures.

Advantages of Link-Based Sequences:
- provide worst-case time bounds for their operations. in contrast to the amortized bounds associated with the expansion or contraction of a dynamic array.
- $O(1)$-time insertions and deletions at arbitrary positions with a hash table.

Solution techniques:
- In an interview, you must understand whether it is a singly linked list or a doubly linked list.
- **The "Runner" Technique:** iterate through the linked list with two pointers simultaneously, with one ahead of the other. The "fast" node might be ahead by a fixed amount, or it might be hopping multiple nodes for each one node that the "slow" node iterates through.
- Many LinkedList problems involve **recursion**.
- **Edge cases:** size=even, size=odd, size=0, size=1, size=2, size=3, head=tail
- Track the previous node.
- **Modify the structure** of the linked list while traversing it.

6.1. Singly Linked Lists

Each node stores a reference to an object that is an element of the sequence, as well as a reference to the next node of the list.

First and last node of a linked list are known as the **head** and **tail**
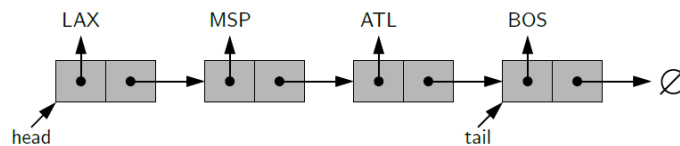
The linked list instance must keep a reference to the head of the list.

Storing an explicit reference to the tail node is a common convenience to avoid traversal.

Linked list instance to keep a count of the total number of nodes (size), to avoid the need to traverse the list to count the nodes.

Inserting an Element at the Head: create a new node, set its element to the new element, set its next link to refer to the current head, and then set the list's head to point to the new node.

Inserting an Element at the Tail: create a new node, assign its next reference to None, set the next reference of the tail to point to this new node, and then update the tail reference itself to this new node.

Deleting a node, given a node n, we find the previous node prey and set prev. next equal to `n.next`. If the list is doubly linked, we must also update `n.next` to set `n.next`. prey equal to `n.prev`. Check for the null pointer and to update the head or tail pointer as necessary.

6.2. Circularly Linked Lists
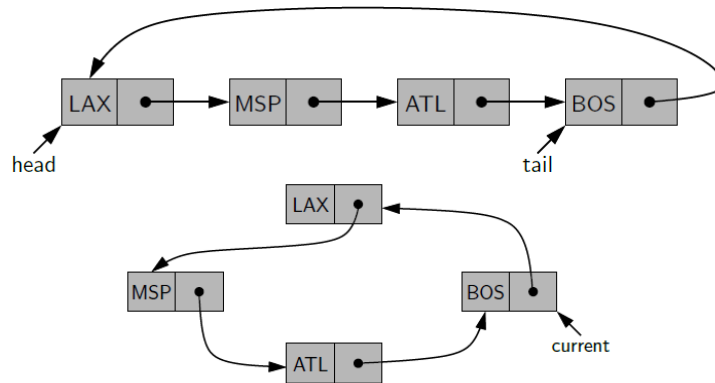
Do not have any particular notion of a beginning and end.
Current: reference to a particular node in order to make use of the list.
`current = current.next`, we can effectively advance through the nodes of the list.

6.3. Doubly Linked Lists

Each node keeps an explicit reference to the node before it and a reference to the node after it.
Allow a greater variety of $O(1)$-time update operations, including insertions and deletions at arbitrary positions within the list.

Special nodes at both ends of the list: a **header** node at the beginning of the list, and a **trailer** node at the end of the list. They do not store elements.
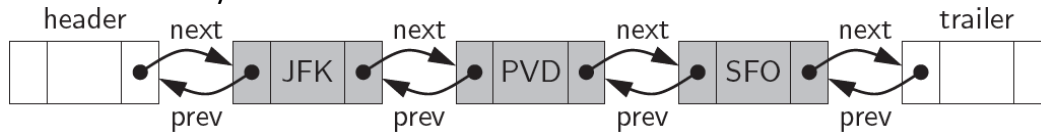
An empty list is initialized so that the `next` field of the `header` points to the `trailer`, and the `prev` field of the trailer points to the header.

Every insertion will take place between a pair of existing nodes.
The deletion, the two neighbors of the node to be deleted are linked directly to each other, thereby bypassing the original node.

Basic Implementation of a Doubly Linked List
Insertions and deletions in $O(1)$ worst-case time, but only if the location of an operation can be succinctly identified.

6.4. The Positional List ADT

Abstract data type that provides a user a way to refer to elements anywhere in a sequence, and to perform arbitrary insertions and deletions.

Position abstract data type to describe a location within a list. Is unaffected by changes elsewhere in a list; the only way in which a position becomes invalid is if an explicit command is issued to delete it.

Has just 1 method:

- **p.element( ):** Return the element stored at position p.

The advantage of receiving a position as a return value is that we can use that position to navigate the list.

Positional list methods:

- **L.first():** Return the position of the first element of L, or None if L is empty.
- **L.last():** Return the position of the last element of L, or None if L is empty.
- **L.before(p):** Return the position of L immediately before position p, or None if p is the first position.
- **L.after(p):** Return the position of L immediately after position p, or None if p is the last position.
- **L.is_empty():** Return True if list L does not contain any elements.
- **len(L):** Return the number of elements in the list.
- **iter(L):** Return a forward iterator for the elements of the list.
- **L.add_first(e):** Insert a new element e at the front of L, returning the position of the new element.
- **L.add_last(e):** Insert a new element e at the back of L, returning the position of the new element.
- **L.add_before(p, e):** Insert a new element e just before position p in L, returning the position of the new element.
- **L.add_after(p, e):** Insert a new element e just after position p in L, returning the position of the new element.
- **L.replace(p, e):** Replace the element at position p with element e, returning the element formerly at position p.
- **L.delete(p):** Remove and return the element at position p in L, invalidating the position.

Implementation with a Doubly Linked List

Each method of the positional list ADT runs in worst-case $O(1)$ time when implemented with a doubly linked list.

<mark>Code…Github</mark> PositionalList.py

Sorting a Positional List with insertion sort
<mark>Code…Github</mark> insertion_sort_positional.py

7.Stacks

Last-in, first-out (LIFO) principle.
Used to:

- reverse a data sequence.
- Recursive algorithms to store data and pick them up lately
- To implement a recursive algorithm iteratively
- In DFS

Methods:

- **S.push(e):** Add element e to the top of stack S.
- **S.pop():** Remove and return the top element from the stack S; an error occurs if the stack is empty.
- **S.top():** Return a reference to the top element of stack S, without removing it; an error occurs if the stack is empty.
- **S.is_empty():** Return True if stack S does not contain any elements.
- **len(S):** Return the number of elements in stack S; __len__

A newly created stack is empty.

Simple Array-Based Stack Implementation
Using a Python `list`, adding an element to the end with the `append` method, and removing the last element with the `pop` method. Storing the top element in the rightmost cell.

Stack Method	Realization with Python list
S.push(e)	L.append(e)
S.pop()	L.pop()
S.top()	L[−1]
S.is_empty()	len(L) == 0
len(S)	len(L)

Operation	Running Time
S.push(e)	$O(1)^*$
S.pop()	$O(1)^*$
S.top()	$O(1)$
S.is_empty()	$O(1)$
len(S)	$O(1)$

*amortized

There is occasionally an $O(n)$-time worst case, where n is the current number of elements in the stack, when an operation causes the list to resize its internal array.
The space usage $O(n)$.

Implementing a Stack with a Singly Linked List

Operation	Running Time
S.push(e)	$O(1)$
S.pop()	$O(1)$
S.top()	$O(1)$
len(S)	$O(1)$
S.is_empty()	$O(1)$

Space usage is $O(n)$

8. Queues

First-in, first-out (FIFO) principle.
Used in:
- breadth first search
- implementing a cache

Methods:
- **Q.enqueue(e):** Add element e to the back of queue Q.
- **Q.dequeue():** Remove and return the first element from queue Q; an error occurs if the queue is empty.
- **Q.first():** Return a reference to the element at the front of queue Q, without removing it; an error occurs if the queue is empty.
- **Q.isEmpty( ):** Return True if queue Q does not contain any elements.
- **len(Q):** Return the number of elements in queue Q; __len__

A newly created queue is empty.

Circular Array-Based Queue Implementation

Allow the front of the queue to drift rightward, and we allow the contents of the queue to "wrap around" the end of an underlying array.

We assume that our underlying array has fixed length N that is greater that the actual number of elements in the queue.

New elements are enqueued toward the "end" of the current queue, progressing from the front to index N −1 and continuing at index 0, then 1.
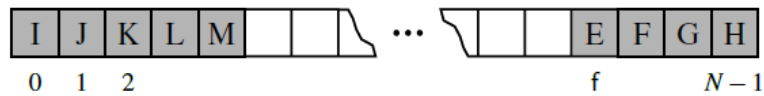
To advance the front index: `f=(f+1)%N`

_data: is a reference to a list instance with a fixed capacity.
_size: is an integer representing the current number of elements stored
_front: is an integer that represents the index within data of the first element of the queue

Operation	Running Time
Q.enqueue(e)	$O(1)^*$
Q.dequeue( )	$O(1)^*$
Q.first( )	$O(1)$
Q.is_empty( )	$O(1)$
len(Q)	$O(1)$

*amortized

Space usage is $O(n)$

Implementing a Queue with a Singly Linked List
Time complexity: everything $O(1)$ without amortization

Implementing a Queue with a Circularly Linked List

8.1. Double-Ended Queues

Supports insertion and deletion at both the front and the back of the queue.

- **D.add_first(e):** Add element e to the front of deque D.
- **D.add_last(e):** Add element e to the back of deque D.
- **D.delete_first():** Remove and return the first element from deque D; an error occurs if the deque is empty.
- **D.delete_last():** Remove and return the last element from deque D; an error occurs if the deque is empty.
- **D.first():** Return (but do not remove) the first element of deque D; an error occurs if the deque is empty.
- **D.last():** Return (but do not remove) the last element of deque D; an error occurs if the deque is empty.
- **D.is_empty():** Return True if deque D does not contain any elements.
- **len(D):** Return the number of elements in deque D; in Python, __len__

Doubly Linked List Implementation: all operation in worst-case $O(1)$ time.

Implementation with Python `collections.deque` class.

Our Deque ADT	collections.deque	Description
len(D)	len(D)	number of elements
D.add_first()	D.appendleft()	add to beginning
D.add_last()	D.append()	add to end
D.delete_first()	D.popleft()	remove from beginning
D.delete_last()	D.pop()	remove from end
D.first()	D[0]	access first element
D.last()	D[−1]	access last element
	D[j]	access arbitrary entry by index
	D[j] = val	modify arbitrary entry by index
	D.clear()	clear all contents
	D.rotate(k)	circularly shift rightward k steps
	D.remove(e)	remove first matching element
	D.count(e)	count number of matches for e

Operation	Average Case	Amortized Worst Case
Copy	O(n)	O(n)
append	O(1)	O(1)
appendleft	O(1)	O(1)
pop	O(1)	O(1)
popleft	O(1)	O(1)
extend	O(k)	O(k)
extendleft	O(k)	O(k)
rotate	O(k)	O(k)
remove	O(n)	O(n)

D=deque()

8.2. Monotonic Queue (MQ)

Data structure that keeps its elements either entirely in non-increasing, or entirely in non-decreasing order.

MQ is mostly used as a dynamic programming optimization technique and for some problems where it is applicable, we can reduce the reasoning to finding the nearest element. Sliding max/min window problem.

Operations:

- **push(item):** Usually removes elements from queue compared to value in {@param item} to preserve monotonicity. Then adds a new element.
- **getFirst():** the first value of the queue, which is usually a maximum or a minimum.
- **removeFirst():** removes min or max when it is no longer needed.

We maintain a monotonic array with index increasing and value decreasing, because smaller elements like A[i] on the left are useless.

A = [3, 1, 4, 3, 8] => monotonic queue is like [3], [3, 1], [4], [4, 3], [8]

when element 4 enters, we remove [3, 1] because they are on the left and smaller than 4, no chance being chosen as the max element.

Any DP problem where A[i] = min(A[j:k]) + C where $j < k \leq i$

…………………………………………………………………….. add stuff…….

9.Trees

```
class Node(object):
      def __init__(self, val, children):
            self.val = val
            self.children = children
```

In interviews typically the Tree class is not used, we just use the Node class.

9.1. General Trees
Tree T is a set of **nodes** storing elements such that the nodes have a parent-child relationship that satisfies the following properties:
- If T is nonempty, it has a special node, called the **root** of T, that has no parent.
- Each node v of T different from the root has a unique **parent** node w; every node with parent w is a **child** of w.

Edge: pair of nodes (u, v) such that u is the parent of v, or vice versa
Path: sequence of nodes such that any two consecutive nodes in the sequence form an edge.
Ancestor: x is ancestor of y if $x = y$ or x is ancestor of the father of y
Descendant: x is descendant of y if y is ancestor of x
Internal nodes: nodes with at least 1 child
External nodes (leaf): nodes without children

Ordered Tree: if there is a meaningful linear order among the children of each node; arranging siblings left to right, according to their order.

Depth of a node (2 ways):
- $\text{depth}_T(v) = |\text{ancestors}(v)| - 1$
- $\text{depth}_T(v) = \begin{cases} 0, & v = \text{root} \\ 1 + \text{depth}_T(\text{parent}(v)), & \text{otherwise} \end{cases}$ $O(n)$ worst case

Level i: set of nodes at depth i
Height of a node: $\text{height}_T(v) = \begin{cases} 0, & v = \text{leaf} \\ 1 + \max\limits_{\forall w: \text{child of } v} (\text{height}_T(w)), & \text{otherwise} \end{cases}$ $O(n)$ worst case
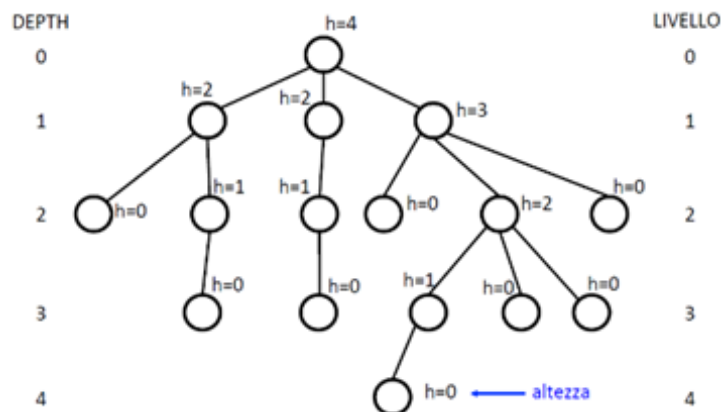
$\sum_v c_v = n - 1$ where $c_v = \#\text{children of } v$

Implementation with Linked Structure:
Each node stores a single container of references to its children.
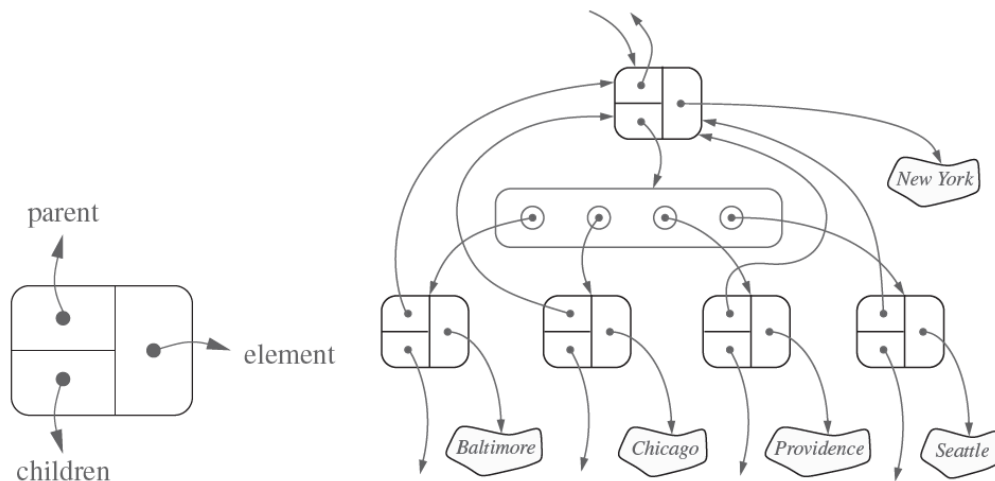
Operation	Running Time
len, is_empty	$O(1)$
root, parent, is_root, is_leaf	$O(1)$
children(p)	$O(c_p + 1)$
depth(p)	$O(d_p + 1)$
height	$O(n)$

c_p denotes the number of children of a node p.
The space usage is $O(n)$.

9.2. Binary Trees

Binary tree: ordered tree such that:

- **Every node has at most two children.**
- Each child node is labeled as being either a **left** child or a **right** child.
- A left child precedes a right child in the order of children of a node.

Recursive definition:

- A node r, called the root of T, that stores an element
- A binary tree (possibly empty), called the left subtree of T
- A binary tree (possibly empty), called the right subtree of T

Properties:

- $h + 1 \leq n \leq n^{h+1} - 1$ h: height
- $1 \leq n_E \leq 2^h$ n: # nodes
- $h \leq n_I \leq 2^h - 1$ n_E: leaves
- $\log(n + 1) - 1 \leq h \leq n - 1$ n_I: internal nodes

Proper(full) Binary tree: if each node has either 0 or 2 children

Proprieties:

- $\log(n + 1) - 1 \leq h \leq (n - 1)/2$
- $n_E = n_I + 1$

Complete Binary tree: every level is fully filled except perhaps the last level. (Filled left to right)
Perfect Binary tree: both full and complete. All levels have the maximum number of nodes.

Linked Structure Implementation:
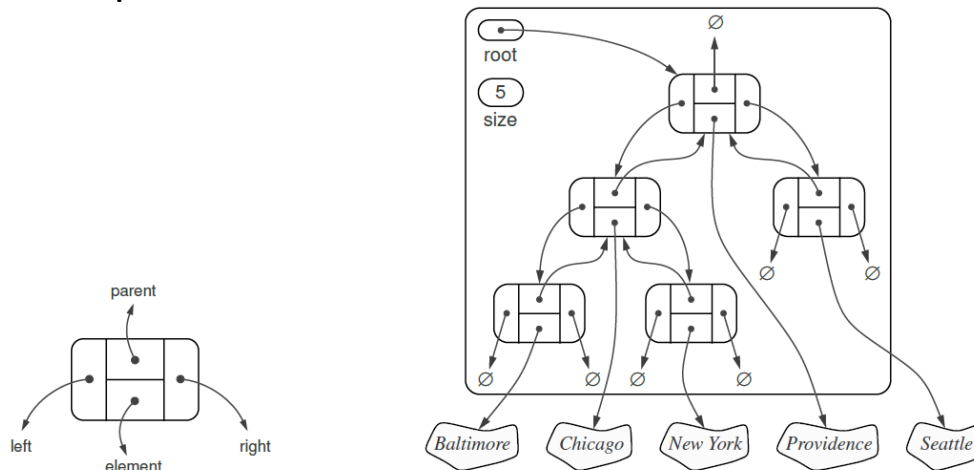
Space usage: $O(n)$

Operation	Running Time
len, is_empty	$O(1)$
root, parent, left, right, sibling, children, num_children	$O(1)$
is_root, is_leaf	$O(1)$
depth(p)	$O(d_p + 1)$
height	$O(n)$
add_root, add_left, add_right, replace, delete, attach	$O(1)$

Array based implementation:
Level numbering $f(v)$: index of the array where to store the node v
- If v is the root of T, then $f(v) = 0$.
- If v is the left child of position q, then $f(v) = 2f(q) + 1$.
- If v is the right child of position q, then $f(v) = 2f(q) + 2$.

Parent index = $\lfloor (f(v) - 1)/2 \rfloor$
Array length $N = 2^n - 1$ worst case (That is prohibitive if n is very large)

9.3. Tree Traversal Algorithms
Depth-first:
- **Preorder**: first visit the father then (recursively) his children. $\boldsymbol{O(n)}$
- **Postorder**: first visit (recursively) the children than the father. $\boldsymbol{O(n)}$
- **Inorder**: visit left child, visit father, visit right child. Applied to binary trees.

Breadth-first:
- **Level-order:** visit all nodes in a level before the next level. Uses a FIFO to save the order in which visits nodes. Time $\boldsymbol{O(n)}$ in order to n calls for enqueue and dequeue.

9.4. Solve Tree problems recursively
Top-down: visit the node first to come up with some values, and pass these values to its children when calling the function recursively. Kind of **preorder.**
top_down(root, params):
      1. return specific value for null node
      2. update the answer if needed   // answer <-- params
      3. left_ans = top_down(root.left, left_params)   // left_params <-- root.val, params
      4. right_ans = top_down(root.right, right_params) // right_params <-- root.val, params
      5. return the answer if needed   // answer <-- left_ans, right_ans

Bottom-Up: first call the functions recursively for all the children nodes and then come up with the answer according to the return values and the value of the root node itself. Kind of **postorder.**
bottom_up(root):
      1. return specific value for null node
      2. left_ans = bottom_up(root.left)   // call function recursively for left child
      3. right_ans = bottom_up(root.right)   // call function recursively for right child
      4. return answers   // answer <-- left_ans, right_ans, root.val

When you meet a tree problem, ask yourself two questions:
- can you determine some parameters to help the node know the answer of itself?
- Can you use these parameters and the value of the node itself to determine what should be the parameters parsing to its children?

If the answers are both yes, try to solve this problem using a **top-down** recursion solution.
Or you can think the problem in this way:
- for a node in a tree, if you know the answer of its children, can you calculate the answer of the node?

If the answer is yes, solving the problem recursively from **bottom up**.

Sometimes Tree problems can be solved with **DIVIDE AND CONQUER.**

9.5. Advanced Trees

9.5.1. Binary indexed tree (Fenwick tree)

Is a dynamic variant of a prefix sum array. It allows to efficiently update array values between sum queries. This would not be possible using a prefix sum array, because after each update, it would be necessary to build the whole prefix sum array again in $O(n)$ time.

Operations:
- add(i,x): add x to the value ad index i. $O(\log n)$
- sum(k): get the sum of the first k elements. $O(\log n)$

Structure: usually represented as an **array**. In this section we assume that all arrays are one-indexed, because it makes the implementation easier.
- $p(k) = k\&-k$ largest power of 2 that divides k.
- $\text{tree}[k] = \text{sum}_q(k - p(k) + 1, k)$
  each position k contains the sum of values in a range of the original array whose length is $p(k)$ and that ends at position k.
- $\text{sum}_q(a, b) = \text{sum}_q(1, b) - \text{sum}_q(1, a - 1)$ where $a > 1$
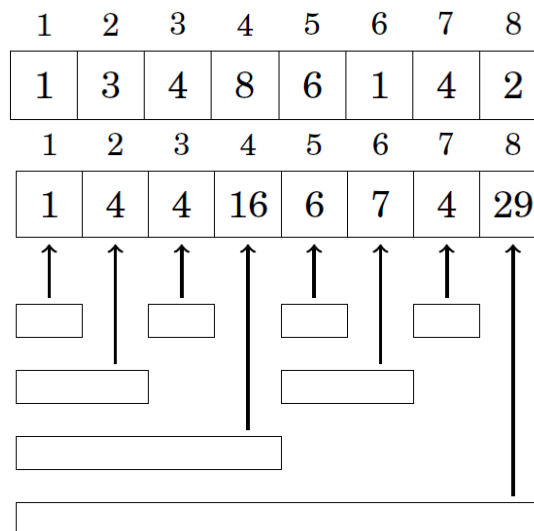

```
class BinaryIndexedTree:
  def __init__(self,arr):
    self.tree=[0]*(len(arr)+1)#1-indexed
    for i in range(len(arr)):
      self.add(i+1,arr[i])

  #Sum [1,k] included O(logn)
  def sum(self,k):
    s=0
    while k>=1:
      s+=self.tree[k]
      k-=k&-k
    return s

  #add x to the value ad index k O(logn)
  def add(self,k,x):
    while k<=len(self.tree):
      self.tree[k]+=x
      k+=k&-k
```

32

9.5.2. Segment Tree

More general than the binary indexed tree. Support more types of queries but requires more memory and is more difficult to implement. Operations supported:

- Processing a range query (sum, min, max, xor, …). $O(\log n)$
- updating an array value. $O(\log n)$

Structure: binary tree such that the nodes on the leaves correspond to the array elements, and the other nodes contain information needed for processing range queries.

Each internal tree node corresponds to an array range whose size is a power of two.

- Any range $[a, b]$ can be divided into $O(\log n)$ ranges whose values are stored in tree nodes.
- After an array update, we update all nodes whose value depends on the updated value. This can be done by traversing the path from the updated array element to the top node and updating the nodes along the path. $O(\log n)$
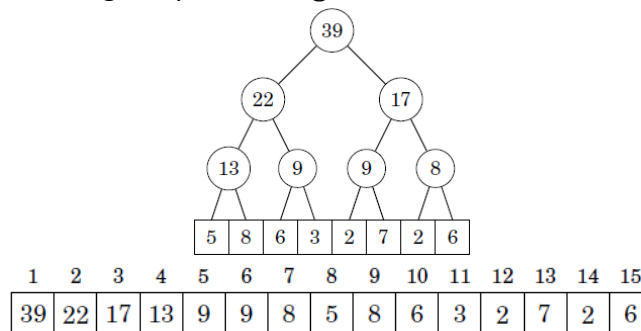
Implementation: array of $2n$ elements where n is the size of the original array and a power of two.

- parent of $\text{tree}[k]$ is $\text{tree}[\lfloor k/2 \rfloor]$
- children of $\text{tree}[k]$ are $\text{tree}[2k]$ and $\text{tree}[2k+1]$

Segment trees can support all range queries where it is possible to divide a range into two parts, calculate the answer separately for both parts and then efficiently combine the answers.

CHECK AGAIN THE CODE, SEEMS NOT WORKING

```python
#This SegmentedTree implement the sum
class SegmentedTree:
  #len(arr) is a power of 2
  def __init__(self,arr):
    self.N=len(arr)
    self.tree=[0]*2*self.N
    for i in range(len(arr)):self.add(i,arr[i])

  #Sum [a,b] included O(logn)
  def sum(self,a,b):
    a+=self.N
    b+=self.N
    s=0
    while a<=b:
      if a%2==1:
        s+=self.tree[a]
        a+=1
      if b%2==0:
        s+=self.tree[b]
        b-=1
      a//=2
      b//=2
    return s

  #add x to the value ad index k O(logn)
  def add(self,k,x):
    k+=self.N
    self.tree[k]+=x
    k//=2
    while k>=1:
      self.tree[k]=self.tree[2*k]+self.tree[2*k+1]
      k//=2
```

10. Search trees

If you want to store data in order and need several operations, such as search, insertion or deletion, at the same time, a BST might be a good choice.

10.1. Binary Search Trees

Binary Search Trees: binary tree with each node storing a key-value pair (k,v) such that:
- keys in the left subtree of n are $\leq k$
- keys in the right subtree of n are $> k$

$$n.left.val \leq n.val < n.right.val$$

Inorder visits nodes in increasing order of their keys. $O(n)$

Search(k): for each node:
- return the node if the target value is equal to the value of the node;
- continue searching in the left subtree if the target value is less than the value of the node;
- continue searching in the right subtree if the target value is larger than the value of the node.
  Time $O(h) = O(\log n)$ average, **worst** $O(n)$
  Space: $O(h) = O(\log n)$ average, worst $O(n)$
  The iterative: Time $O(h) = O(\log n)$, Space $O(1)$

Insert(v): for each node, we will:
- search the left or right subtrees according to the relation of the value of the node and the value of our target node;
- repeat STEP 1 until reaching an external node;
- add the new node as its left or right child depending on the relation of the value of the node and the value of our target node.
- **Time** $O(h) = O(\log n)$ average, **worst** $O(n)$
- Space: $O(h) = O(\log n)$ average, worst $O(n)$

Deletion(k): begin by calling TreeSearch(root, k) to find the node. If the search is successful, we distinguish between 3 cases:
- n has 0 child: delete node n

Case 1: No Child
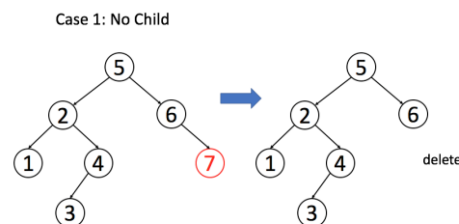
- n has 1 child: delete node n and replace it with its child

Case 2: One Child
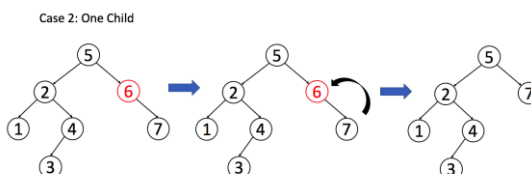
- n has 2 children: replace the node with its in-order successor or predecessor node and delete that node.

Case 3: Two Children

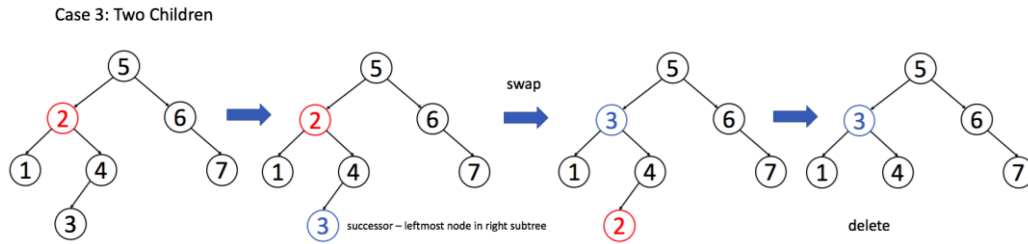| Operation	Running Time
k in T	$O(h)$
T[k], T[k] = v	$O(h)$
T.delete(p), del T[k]	$O(h)$
T.find_position(k)	$O(h)$
T.first(), T.last(), T.find_min(), T.find_max()	$O(h)$
T.before(p), T.after(p)	$O(h)$
T.find_lt(k), T.find_le(k), T.find_gt(k), T.find_ge(k)	$O(h)$
T.find_range(start, stop)	$O(s+h)$
iter(T), reversed(T)	$O(n)$

The space usage is $O(n)$

Binary search tree T is an efficient implementation of a map with n entries only if its height is small. On average, a binary search tree with n keys generated from a random series of insertions and removals of keys has expected height $O(\log n)$.
In applications where one cannot guarantee the random nature of updates, rely on variations of search trees, that guarantee a worst-case height of $O(\log n)$, and thus $O(\log n)$ worstcase time for searches, insertions, and deletions.

10.2. Balanced Search Trees
Ensure search, insert and delete in $O(\log n)$, height is always $O(\log n)$
Used especially in sets and maps.

Augment a standard binary search tree with occasional operations to reshape the tree and reduce its height.
**Rotation**: O(1). allows the shape of a tree to be modified while maintaining the search tree property.
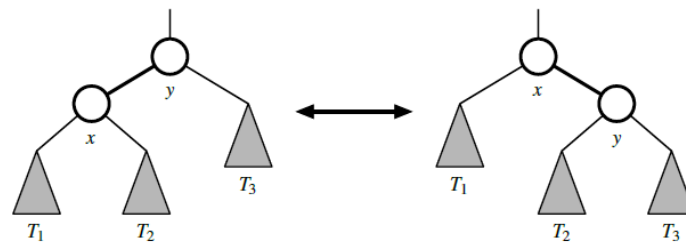
Trinode restructuring: One or more rotations combined to provide broader rebalancing. O(1)

10.3. AVL Trees

Balancing strategy that guarantees worst-case logarithmic running time for all the fundamental map operations.

Height-Balance Property: For every node n, the heights of the children differ by at most 1.
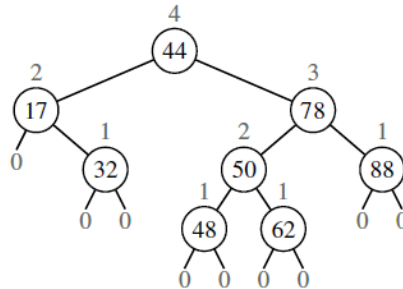
The height of an AVL tree storing n entries is $O(\log n)$.
Getitem: $O(\log n)$
Insertion: $O(\log n)$
Like a normal insertion in a binary search tree + …
Insert a new element results in a new node at a leaf position p. This may violate the height-balance property.

To restore the balance:
- let z be the first position we encounter in going up from p toward the root of T such that z is unbalanced.
- let y denote the child of z with higher height
- let x be the child of y with higher height
- trinode restructuring method

https://www.programiz.com/dsa/avl-tree (their code seems not working)
https://favtutor.com/blogs/avl-tree-python
Insertion: ……pag483 Goodrich Book python

Operation	Running Time
k in T	$O(\log n)$
T[k] = v	$O(\log n)$
T.delete(p), del T[k]	$O(\log n)$
T.find_position(k)	$O(\log n)$
T.first(), T.last(), T.find_min(), T.find_max()	$O(\log n)$
T.before(p), T.after(p)	$O(\log n)$
T.find_lt(k), T.find_le(k), T.find_gt(k), T.find_ge(k)	$O(\log n)$
T.find_range(start, stop)	$O(s + \log n)$
iter(T), reversed(T)	$O(n)$

10.4. Red-Black Trees

Is a type of self-balancing binary search tree where each node has an extra bit, and that bit is often interpreted as the color (red or black). These colors are used to ensure that the tree remains balanced during insertions and deletions. Although the balance of the tree is not perfect, it is good enough to reduce the searching time and maintain it around $O(\log n)$ time.

- Every node has a color either red or black.
- The root of tree is always black.
- There are no two adjacent red nodes (A red node cannot have a red parent or red child).
- Every path from a node (including root) to any of its descendant NULL node has the same number of black nodes.

The height of a Red-Black tree is always $O(\log n)$

Comparison with AVL Tree: The AVL trees are more balanced compared to Red-Black Trees, but they may cause more rotations during insertion and deletion. So, if your application involves frequent insertions and deletions, then Red-Black trees should be preferred. And if the insertions and deletions are less frequent and search is a more frequent operation, then AVL tree should be preferred over Red-Black Tree.

10.5. Tries

String searching algorithms that preprocess the text, for applications where a series of queries is performed on a fixed text, so that the initial cost of preprocessing the text is compensated by a speedup in each subsequent query.

Trie: tree-based data structure for storing strings, support fast pattern matching and prefix matching.

10.5.1. Standard Trie

Let S be a set of s strings from alphabet Σ such that no string in S is a prefix of another string.
Standard trie for S is an ordered tree T with the following properties:
*   Each node of T, except the root, is labeled with a character of Σ.
*   The children of an internal node of T have distinct labels.
* T has s leaves, each associated with a string of S, such that the concatenation of the labels of the nodes on the path from the root to a leaf v of T yields the string of S associated with v.

Trie T represents the strings of S with paths from the root to the leaves of T.
Each string of S is uniquely associated with a leaf of T.
An internal node can have anywhere between 1 and $|\Sigma|$ children.
If there are only two characters in the alphabet, then the trie is essentially a binary tree, with some internal nodes possibly having only one child.

A **standard trie** storing a collection S of s strings of total length n from an alphabet Σ has the following properties:
*   The height of T is equal to the length of the longest string in S.
*   Every internal node of T has at most $|\Sigma|$ children. Or +1 if it uses a Boolean flag.
*   T has s leaves
*   The number of nodes of T is at most n+1.

A trie can be used to implement a set or map whose keys are the strings of S.

Implementation with a `dict` and flag nodes:

```
#From array of words to Trie      #Search in trie
trie={}                           t=trie
for w in words:                   for c in word:
  t=trie                            if c in t: t=ht[c]
  for c in w:                     if "#" in t: return True
    if c not in t: t[c]={}        return False
    t=t[c]
  t["#"]=True
```
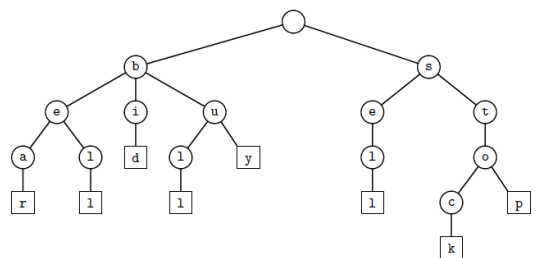

Time complexity: (using a hash table that stores the children)
*   Search: $O(m)$ where m is the length of the word to search
*   Insert: $O(m)$, to insert an entire set S. $O(n)$ where n is the total length of the string in S.
*   Delete: $O(m)$ if the last letter of the word has children we do not delete. When we traverse the trie we save the last node that has more than 1 children.

10.5.2. Compressed Trie

Each internal node has ≥ 2 children
By compressing chains of single-child nodes into individual edges.
Redundant node: if has one child and is not the root.
Redundant chain: of $k \geq 2$ edges $(v_0, v_1) \dots (v_{k-1}, v_k)$ if

- v_i is redundant for $i = 1, \dots, k-1$
- v_0 and v_k are not redundant
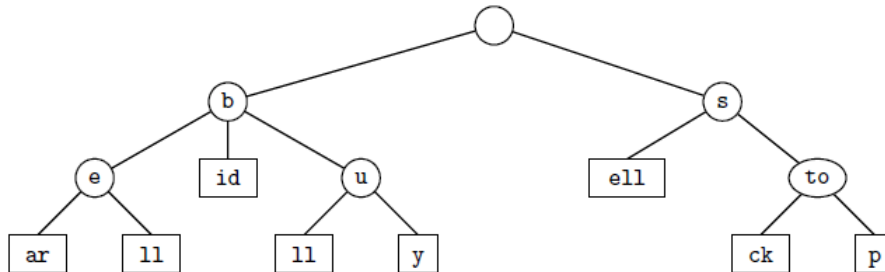
Proprieties: of a trie storing a collection S of s strings from an alphabet of size d

- Every internal node of T has at least two children and most d children.
- T has s leaves nodes.
- The number of nodes of T is $O(s)$.

10.5.3. Suffix Tries

Used in the case when the strings in the collection S are all the suffixes of a string X.
To satisfy the rule that no suffix of X is a prefix of another suffix, we can add a special character, denoted with $\$$, that is not in the original alphabet Σ at the end of X.
The compact representation of a suffix trie T for a string X of length n uses $O(n)$ time and space.

Generalized suffix tries……

11. Graphs

11.1. Basics
Graph $G = (V, E)$ where V=set of vertices, E=set of edges (pairs of vertices).
- directed: from u to v if the pair (u, v) is ordered, with u preceding v
- undirected: the pair (u, v) is not ordered

Undirected or mixed graph can be converted into a directed graph by replacing every undirected edge (u, v) by the pair of directed edges (u, v) and (v, u).

Adjacent edges: have a vertex in common.

Degree of a vertex $\deg(v) = $ # incident edges of v

$$\sum_{v \in V} \deg(v) = 2|E|$$

Simple graph: no parallel edges no self-loops. So E is a set of vertex pairs.
Path: sequence of edges.
Simple Path: don't use 2 times the same vertex.
Cycle: path that starts and ends at the same vertex, and that includes at least one edge
Directed Acyclic Graph: has no directed cycles.
Connected Graph: if, for any two vertices, there is a path between them.
Subgraph: graph $H = (V', E')$ where $V' \subseteq V, E' \subseteq E$
Spanning subgraph: subgraph $H = (V', E')$ where $V' = V, E' \subseteq E$

Forest: graph without cycles. (trees disconnected from each other)
Tree: connected forest, that is, a connected graph without cycles.
Spanning Tree of a graph: is a spanning subgraph that is a tree.

Properties of graphs: ($|V| = n$, $|E| = m$, not directed)
- $G\ simple \Longrightarrow m \leq \binom{n}{2} \Longrightarrow m \in O(n^2)$
- $G\ tree \Longrightarrow m = n - 1$
- $G\ connected \Longrightarrow m \geq n - 1$
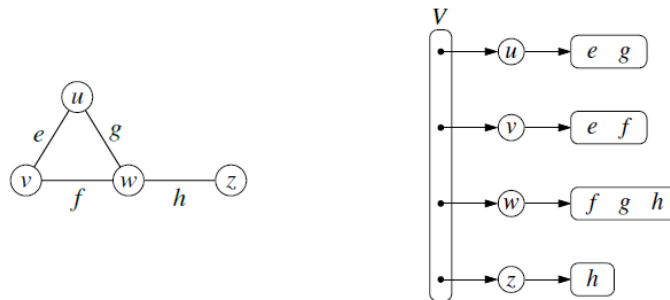- $G\ forest \Longrightarrow m \leq n - 1$

Eulerian Path: visits every edge exactly once.
Hamiltonian path: visits every vertex exactly once. NP-Complete.

Graph Editor Online: https://csacademy.com/app/graph_editor/

11.2. Graphs representations

Adjacency list: most common. Every vertex stores a list of adjacent vertices. In the form of array, linked list or hash table.

Operation	Running Time
vertex_count( ), edge_count( )	$O(1)$
vertices( )	$O(n)$
edges( )	$O(m)$
get_edge(u,v)	$O(\min(\deg(u), \deg(v)))$
degree(v)	$O(1)$
incident_edges(v)	$O(\deg(v))$
insert_vertex(x), insert_edge(u,v,x)	$O(1)$
remove_edge(e)	$O(1)$
remove_vertex(v)	$O(\deg(v))$

Space Complexity: $O(|V| + |E|)$

Adjacency Map: opposite endpoint of each incident edge serves as a key in the map, with the edge structure serving as the value.
get_edge(u,v) method can be implemented in expected $O(1)$ time
Space Complexity: $O(|V| + |E|)$

Adjacency Matrix: $|V| \times |V|$ Boolean matrix where a True value in `matrix[i][j]` indicates an edge from node i to node j.
Any edge (u, v) can be accessed in worst-case $O(1)$
A is symmetric if graph G is undirected.
Most real-world graphs are sparse. In such cases, use of an adjacency matrix is inefficient.

11.3. Graph Traversals

Traversal is considered efficient if is done in linear time.
Discovery edge: edge used to discover a new vertex.

Depth-First-Search (DFS): start at the root and explore each branch completely before moving to the next branch. Preferred if we want to visit each node in the graph.
The discovery edges form a spanning tree.
Can be used to solve the following problems in $O(n + m)$ time:
- Computing a path between two given vertices of G, if one exists.
- Testing whether G is connected.
- Computing a spanning tree of G, if G is connected.
- Computing the connected components of G.
- Computing a cycle in G, or reporting that G has no cycles.

Iterative Python Implementation:
```python
def dfs(graph, start):
    visited, stack = set(), [start]
    while stack:
        vertex = stack.pop()
        if vertex not in visited:
            visited.add(vertex)
            stack.extend(graph[vertex] - visited)
    return visited
```

Recursive Python Implementation:
```python
def dfs(graph, start, visited=None):
    if visited is None:
        visited = set()
    visited.add(start)
    for vertex in graph[start] - visited:
        dfs(graph, vertex, visited)
    return visited
```

Breadth-first-search (BFS): start at the root and explore each neighbor before going on to any of their children.
Takes $O(n + m)$ time
For each vertex v at level i, the path of the BFS tree T between starting vertex s and v has i edges, and any other path of G from s to v has at least i edges.

Iterative Python Implementation:
```python
def bfs(graph, start):
    visited, queue = set(), [start]
    while queue:
        vertex = queue.pop(0)
        if vertex not in visited:
            visited.add(vertex)
            queue.extend(graph[vertex] - visited)
    return visited
```

Bidirectional search: used to find the shortest path. Runs 2 simultaneous BFS, from 2 nodes, when their searches collide, we found a path.
Most of the problems can be dissected into:
- Finding the # of components
- Detecting/finding a cycle.
- Finding shortest path between nodes.
- Minimum spanning tree.
- Topological sort

11.4. Shortest Path Algos

11.4.1. Transitive Closure (Floyd-Warshall)

Transitive closure of a directed graph $\vec{G}$: is a directed graph $\vec{G}^*$ whose vertices are the same as the vertices of $\vec{G}$, and has an edge (u, v), whenever $\vec{G}$ has a directed path from u to v (including the case where (u, v) is an edge of the original $\vec{G}$).
Can be computed in $O\big(n(n + m)\big)$ by making n graph traversals, one from each starting vertex.

Floyd-Warshall $O(n^3)$: find shortest distances between every pair of vertices in a weighted graph with positive or negative edge weights. Can find negative circuits. Good for small graphs (hundreds of nodes).

Algorithm FloydWarshall($\vec{G}$):
    Input: A directed graph $\vec{G}$ with n vertices
    Output: The transitive closure $\vec{G}^*$ of $\vec{G}$
    let $v_1, v_2, \ldots, v_n$ be an arbitrary numbering of the vertices of $\vec{G}$
    $\vec{G}_0 = \vec{G}$
    for $k = 1$ to n **do**
      $\vec{G}_k = \vec{G}_{k-1}$
      for all i, j in $\{1, \ldots, n\}$ with $i \neq j$ and $i, j \neq k$ **do**
        if both edges (v_i, v_k) and (v_k, v_j) are in $\vec{G}_{k-1}$ **then**
          add edge (v_i, v_j) to $\vec{G}_k$ (if it is not already present)
    return $\vec{G}_n$

```
def floyd_warshall(self):
    #dist = {u: {v: float('inf') for v in self.V} for u in self.V}
    dist={}
    for u in self.V:
      d={}
      for v in self.V:
        d[v]=float('inf')
      dist[u]=d

    for u in self.V: dist[u][u] = 0
    for u in self.E:
      for v in self.E[u]:
        dist[u][v] = min(dist[u][v], self.E[u][v])
    for w, u, v in itertools.permutations(self.V, 3):
      dist[u][v] = min(dist[u][v], dist[u][w] + dist[w][v])
    return dist
```

11.4.2. Dijkstra

Finds shortest distances between s end every other vertex.
Requires weights ≥ 0 for each edge, because when a vertex join the cloud, the total cost till now can't decrease.

How it works: Perform a "weighted" breadth-first search starting at the source vertex s.
Iteratively grows a "cloud" of vertices out of s, with the vertices entering the cloud in order of their distances from s.
In each iteration, the next vertex chosen is the vertex outside the cloud that is closest to s.
The algorithm terminates when no more vertices are outside the cloud.
Edge Relaxation: put a label $D[v]$ for each vertex v in V, $D[v]$ will always store the length of the best path we have found so far from s to v. Initially, $D[s] = 0$ and $D[v] = \infty$.

$$\text{If } D[u] + w(u,v) < D[v] \text{ then } D[v] = D[u] + w(u,v)$$

Algorithm ShortestPath(G, s):
  Input: A weighted graph G with nonnegative edge weights, and a distinguished vertex s of G.
  Output: The length of a shortest path from s to v for each vertex v of G.
  Initialize $D[s] = 0$ and $D[v] = \infty$ for each vertex $v \neq s$.
  Let a priority queue Q contain all the vertices of G using the D labels as keys.
  while Q is not empty **do**
    {pull a new vertex u into the cloud}
    u = value returned by Q.remove_min()
    for each vertex v adjacent to u such that v is in Q **do**
      {perform the *relaxation* procedure on edge (u,v)}
      if $D[u] + w(u,v) < D[v]$ **then**
        $D[v] = D[u] + w(u,v)$
        Change to $D[v]$ the key of vertex v in Q.
  return the label $D[v]$ of each vertex v

Implementations:

- **Heap implementation** $O\big((n+m)\log n\big)$. Better when the number of edges in the graph is small $m < n^2/\log n$
- Unsorted sequence implementation $O(n^2)$. Better when the number of edges is large $m > n^2/\log n$.
- Fibonacci heap implementation $O(m + n\log n)$

Shortest-path tree rooted at source s can be reconstructed in $O(n+m)$ time, given the set of d[v].

```python
def dijkstra(self, source, trace=False):
  dist = {u: float('inf') for u in self.V}
  dist[source] = 0

  #Put all nodes in min heap
  Q = [(d, u) for u, d in dist.items()]   #heap (dist,node)
  heapq.heapify(Q)

  if trace:
    paths = {u: [[u]] for u in self.V}

  while Q:
    d, u = heapq.heappop(Q)
    for v, weight in self.E[u].items():
      alt = d + weight
      if alt <= dist[v]:
        if alt < dist[v]:
          dist[v] = alt
          heapq.heappush(Q, (alt, v))
        if trace:
          paths[v] = [x for x in paths[v] if x[0] == source] + [y + [v] for y in paths[u]]

  if trace:
    for u in paths:
      paths[u] = [x for x in paths[u] if x[0] == source]

  return paths if trace else dist
```

11.4.3. Bellman Ford

Finds shortest distances between s end every other vertex.
Work also with negative weights.

Negative weight can create negative weight cycles (a cycle that will reduce the total path distance by coming back to the same point).

Works by overestimating the length of the path from the starting vertex to all other vertices. Then it iteratively relaxes those estimates by finding new paths that are shorter than the previously overestimated paths. By doing this repeatedly for all vertices, we can guarantee that the result is optimized.

…………………CONTINUE…..

11.5. Minimum/Maximum Spanning Trees

Spanning tree: contains every vertex of a connected graph G

MST Problem: given an undirected, weighted graph G, find a tree T that contains all the vertices in G and minimizes/maximize the sum: $w(T) = \sum_{(u,v)\in T} w(u,v)$.

11.5.1. Prim-Dijkstra

Algorithm:
- Begin with some vertex s, defining the initial "cloud" of vertices C.
- In each iteration, choose a minimum(max)-weight edge e=(u,v), connecting a vertex u in the cloud C to a vertex v outside of C.
- The vertex v is then brought into the cloud C and the process is repeated until a spanning tree is formed.

Put a label $D[v]$ for each vertex v outside the cloud C. $D[v]$ stores the weight of the minimum observed edge for joining v to the cloud C. These labels serve as keys in a heap used to decide which vertex is next in line to join the cloud.

Implementations:
- **Heap** $O(m \log n)$
- Unsorted list as heap $O(n^2)$

Algorithm PrimJarnik(G):
> **Input:** An undirected, weighted, connected graph G with n vertices and m edges
> **Output:** A minimum spanning tree T for G
> Pick any vertex s of G
> $D[s] = 0$
> **for** each vertex $v \neq s$ **do**
>> $D[v] = \infty$
> Initialize $T = \emptyset$.
> Initialize a priority queue Q with an entry $(D[v], (v, \text{None}))$ for each vertex v,
> where $D[v]$ is the key in the priority queue, and (v, None) is the associated value.
> **while** Q is not empty **do**
>> $(u, e) = $ value returned by Q.remove_min()
>> Connect vertex u to T using edge e.
>> **for** each edge $e' = (u, v)$ such that v is in Q **do**
>>> {check if edge (u, v) better connects v to T}
>>> **if** $w(u, v) < D[v]$ **then**
>>>> $D[v] = w(u, v)$
>>>> Change the key of vertex v in Q to $D[v]$.
>>>> Change the value of vertex v in Q to (v, e').
> **return** the tree T

$T^* = \emptyset$
$S = \{1\}$
WHILE $|T^*| \neq m-1$:
    TROVA IL LATO $[v,h] \in \delta(S)$ DI COSTO MINIMO, CON $v \in S$ e $h \not\in S$
    $T^* = T^* \cup \{[v,h]\}$ MASSIMO SE IN
    $S = S \cup h$ CERCA IL COSTO MAX

```python
def prim_dijkstra(V): #O(n^2), V={vertex:->[(vertex,weight),…]}
    n=len(V)
    T=[]      #List of edges that form the tree
    S=set([0])  #Cloud of vertices

    while len(T)!=n-1:
        #find the minimum outgoing edge
        minW=float("inf")
        choosenEdge=None
        for v in S:
            for u,w in V[v]:
                if u in S: continue
                if w<minW:
                    choosenEdge=(v,u)
                    minW=w
        if not choosenEdge: break
        #Add the edge to the tree
        T.append(choosenEdge)
        #Add the vertex to the cloud
        S.add(choosenEdge[1])
    return T
```

11.5.2. Kruskal

Maintains a forest of clusters, repeatedly merge pairs of clusters until a single cluster spans the graph.

- Initially, each vertex is by itself in a singleton cluster.
- Considers each edge in turn, ordered by increasing/decreasing weight (min/max).
    - If an edge connects two different clusters, then it is added to the set of edges of the minimum spanning tree, and the two clusters connected by it are merged into a single cluster.
    - Otherwise, if it connects two vertices that are already in the same cluster, then it is discarded.

Implementation: $O(m \log n)$


```python
class Graph:
  def __init__(self, vertices):
    self.V = vertices
    self.graph = []

  def add_edge(self, u, v, w):
    self.graph.append([u, v, w])

  # Search function
  def find(self, parent, i):
      if parent[i] == i: return i
      return self.find(parent, parent[i])

  def apply_union(self, parent, rank, x, y):
    xroot = self.find(parent, x)
    yroot = self.find(parent, y)
    if rank[xroot] < rank[yroot]:
      parent[xroot] = yroot
    elif rank[xroot] > rank[yroot]:
      parent[yroot] = xroot
    else:
      parent[yroot] = xroot
      rank[xroot] += 1

  #  Applying Kruskal algorithm
  def kruskal_algo(self):
    result = []
    i, e = 0, 0
    self.graph = sorted(self.graph, key=lambda item: item[2])
    parent = []
    rank = []
    for node in range(self.V):
      parent.append(node)
      rank.append(0)
    while e < self.V - 1:
      u, v, w = self.graph[i]
      i = i + 1
      x = self.find(parent, u)
      y = self.find(parent, v)
      if x != y:
        e = e + 1
        result.append([u, v, w])
        self.apply_union(parent, rank, x, y)
    for u, v, weight in result:
      print("%d - %d: %d" % (u, v, weight))
```

……………Goodrich page 676

….

11.6. Other Algos

11.6.1. Topological Sort

Topological ordering of $\vec{G}$ is an ordering $v_1, \dots, v_n$ of the vertices s.t. for every edge (v_i, v_j) $i < j$.

Topological Sort: Computes a topological ordering of $\vec{G}$ or **fails** to include some vertices, which indicates that the subgraph of the vertices that have not been ordered **contain a directed cycle**.

Implementation uses a dictionary to map each vertex v to a counter that represents the current number of incoming edges to v, excluding those coming from vertices that have previously been added to the topological order.

Python Implementation: $\boldsymbol{O(n + m)}$ time, $O(n)$ space, n=#nodes, m=#edges

```python
def topological_sort(g):
    topo = [] # a list of vertices placed in topological order
    ready = [] # list of vertices that have no remaining constraints
    incount = {} # keep track of in-degree for each vertex
    for u in g.vertices():
        incount[u] = g.degree(u, False) # incoming degree
        if incount[u] == 0: # if u has no incoming edges,
            ready.append(u) # it is free of constraints
    while len(ready) > 0:
        u = ready.pop() # u is free of constraints
        topo.append(u) # add u to the topological order
        for e in g.incident_edges(u): # consider all outgoing neighbors of u
            v = e.opposite(u)
            incount[v] -= 1 # v has one less constraint without u
            if incount[v] == 0:
                ready.append(v)
    return topo
```

11.6.2. Bipartite graph

Its vertices can be divided into two disjoint and independent sets U and V such that every edge connects a vertex in U to one in V.

Does not contain any odd-length cycles.

Check if a graph is bipartite:

```python
def bipartite(self):
    color = {u: 0 for u in self.V}
    for u in color:
        if color[u] == 0:
            color[u] = 1
            stack = [u]
            while stack:
                u = stack.pop()
                for v in self.E[u]:
                    if self.E[u][v] < float('inf'):
                        if color[v] == 0:
                            color[v] = -color[u]
                            stack += [v]
                        elif color[v] == color[u]:
                            return False
    return True
```

11.6.3. Reachability

Tell which vertices are reachable from a starting vertex s. $O(n + m)$

```
#Given:
# - n vertices labeled from 0 to n-1
# - start vertex s
# - eventual endpoint t
#Initialize
pred=[0]*n   #predecessor of vertex i
pred[s]=s
Q=[s]      #vertices to process

#process vertices in Q
while Q:
  h=Q.pop()
  for j in outgoing_edges[h]:
    if pred[j]==0:
      pred[j]=h
      Q.append(j)
    #if pred[t]!=0: break    #If just interested to find a path between s and t
return pred
```

11.6.4. Kosaraju (Strongly Connected Components)

Strongly connected component: portion of a directed graph in which there is a path from each vertex to another vertex.

Adding an outgoing edge and an ingoing between 2 connected components they became a single connected component.

Kosaraju's Algorithm can find the strongly connected components of a graph. $O(V + E)$
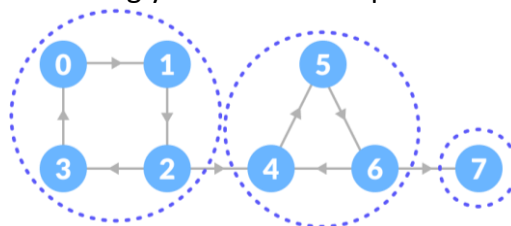
- Perform DFS:
  - Mark the visited vertices as done.
  - If a vertex leads to an already visited vertex, then push this vertex to the stack.
  - If there is nowhere to go from a vertex, push it into the stack.

Visited	0	1	2	3	4	5	6	7

Stack	3	7	6	5	4	2	1	0

- Reverse the edges of the original graph.
- Perform DFS on the reversed graph:
  - Start from the top vertex of the stack. Traverse through all of its child vertices. Once the already visited vertex is reached, one strongly connected component is formed.
  - Go to the stack and pop the top vertex if already visited. Otherwise, choose the top vertex from the stack and traverse through its child vertices as presented above.

SCC	0	1	2	3			
SCC	4	5	6				
SCC	7						

```python
def kosaraju(g):
    # 1. For each vertex u of the graph, mark u as unvisited. Let l be empty.
    size = len(g)

    vis = [False]*size # vertexes that have been visited
    l = [0]*size
    x = size
    t = [[]]*size    # transpose graph

    def visit(u):
        nonlocal x
        if not vis[u]:
            vis[u] = True
            for v in g[u]:
                visit(v)
                t[v] = t[v] + [u]
            x-=1
            l[x] = u

    # 2. For each vertex u of the graph do visit(u)
    for u in range(len(g)):
        visit(u)
    c = [0]*size

    def assign(u, root):
        if vis[u]:
            vis[u] = False
            c[u] = root
            for v in t[u]:
                assign(v, root)

    # 3: For each element u of l in order, do assign(u, u)
    for u in l:
        assign(u, u)
    return c

g = [[1], [2], [0], [1,2,4], [3,5], [2,6], [5], [4,6,7]]
g=[[1],[2],[3,4],[0],[5],[6],[4,7],[]]
print(kosaraju(g))
```

11.6.5. Ford-Fulkerson (Max-Flow)

Flow problems: distribute a product form one or more starting points to one or more ending points.
Flow network: ……

....

….

11.6.6. Disjoint subsets - Union Find

Data structure for managing a partition of elements into a collection of disjoint sets.
Operations:

- make_group(x): Create a singleton group containing new element x and return the position storing x.
- union(p, q): Merge the groups containing positions p and q.
- find(p): Return the position of the leader of the group containing position p.

Sequence Implementation

Collection of sequences, one for each group.

- **union(p,q)**: remove all the elements from the smaller group, and insert them in the larger. Update the group reference of those elements to now point to the larger group.
  $O\big(\min(n_p, n_q)\big)$, where n_p (resp. n_q) is the cardinality of the group.
- make_group(x) and find(p) operations in $O(1)$ time

Performing a series of k make group, union, and find operations on an initially empty partition involving at most n elements takes $O(k + n\log n)$ time.
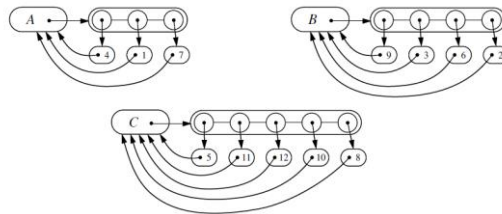
Figure 14.25: Sequence-based implementation of a partition consisting of three groups: $A = \{1,4,7\}$, $B = \{2,3,6,9\}$, and $C = \{5,8,10,11,12\}$.

CHECK AGAIN THE IMPLEMENTATION MAYBE ITS WRONG

```
elements={}    #MAP element id --> group id
groups={}      #MAP group id --> list of elements ids in that group
group_id=0     #group id to use to make a new group

def make_group(element):  #O(1)
  groups[group_id]=[element]
  elements[element]= group_id
  group_id+=1
  return group_id-1

#return group id that contains the element or None. O(1)
def find(element):
  if element in elements: return elements[element]
  return None

def union(a,b): #O(min(|group_A|,|group_B|))
  group_A=find(a)
  group_B=find(b)

  #if a and b are in the same group there is nothing to union
  if group_A==group_B: return

  #Identify which is the larger group
  smaller,larger=group_B,group_A
  if len(groups[group_A])>len(groups[group_B]):
    larger,smaller=group_B,group_A

  #Put all elements of the smaller group in the larger
  while groups[smaller]:
    e=groups[smaller].pop()
    groups[larger].append(e)
    elements[e]=larger
  del groups[smaller]
```

Tree-Based Partition Implementation
Collection of trees to store the n elements, where each tree is associated with a different group.

- Bellman-Ford - Network Delay Time, https://leetcode.com/problems/get-watched-videos-by-your-friends/
- Johnson's algorithm
  - All-pairs shortest paths - Johnson's algorithm for sparse graphs - GeeksforGeeks
  - Johnson's algorithm
- The Ford-Fulkerson method
  - Google | Onsite | Network flow for the matrix with given row and column sums
  - Ford-Fulkerson Algorithm for Maximum Flow Problem

Ford Fulkerston RO (https://www.programiz.com/dsa/ford-fulkerson-algorithm)
Bellman Ford's Algorithm (https://www.programiz.com/dsa/bellman-ford-algorithm)

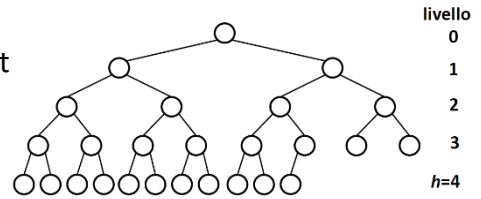https://github.com/AWice/cheat-sheet/blob/master/graph.md

12. Heaps

Priority Queue: collection of (key, value) pairs in which the keys represent the priority.
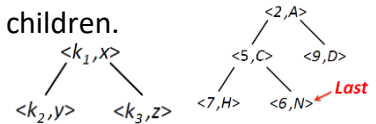
Complete binary tree:
- each level has the maximum number of nodes except maybe for the last
- the last level is filled from the left
- Height $h = \lfloor \log_2 n \rfloor$

Min-Heap: complete binary tree in which each node is smaller than its children.
Heap-order property: $k \leq \min(k.left, k.right)$
Last node: rightest node in the last level

Heap implemented by an array:
- index 0: root
- index $2i + 1$: Left Child
- index $2i + 2$: Right Child
- index $\lfloor (i-1)/2 \rfloor$: Father
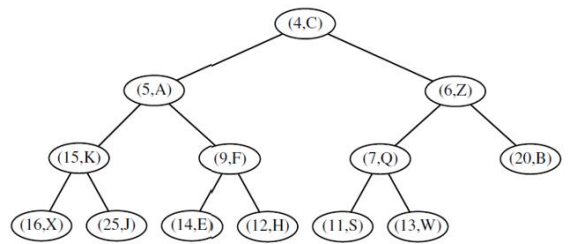
Level numbering property:
- $\forall i, 0 \leq i < h$: the 2^i nodes at level i taken from left to right are mapped in P[j], where $2^i - 1 \leq j \leq 2^{i+1} - 2$
- Nodes at level h taken from left to right are mapped in P[j], where $2^h - 1 \leq j \leq n - 1$
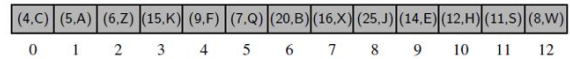
Operation	Running Time
len(P), P.is_empty()	$O(1)$
P.min()	$O(1)$
P.add()	$O(\log n)$*
P.remove_min()	$O(\log n)$*

*amortized, if array-based

Python implementation by `heapq` **Module:**
Provides functions that allow a standard Python list to be managed as a heap.
Does not separately manage associated values; elements serve as their own key.
Functions that presume existing list L satisfies the heap-order property:
- `heappush(L, e)`: $O(\log n)$ time. Insert the element at the bottom, at the rightmost spot. Fix the tree: swap the new element with its parent, until we find an appropriate spot.
- `heappop(L)`: $O(\log n)$ time. Remove the minimum and replace it with the last element in the heap (the bottommost, rightmost element). Then swap this element with the smaller of its children until the heap propriety is restored.
- `heappushpop(L, e)`: Push element e on list L and then pop and return the smallest item. $O(\log n)$ time, but it is slightly more efficient than separate calls to push and pop. If the newly pushed element becomes the smallest, it is immediately returned.
- `heapreplace(L, e)`: Similar to `heappushpop`, but the pop is performed before the push. $O(\log n)$ time, but it is more efficient that two separate operations.

Functions that operate on sequences that do not previously satisfy the heap-order property:
- `heapify(L)`: Transform unordered list to satisfy the heap-order property. $\boldsymbol{O(n)}$ time
- `nlargest(k, iterable)`: Produce a list of the k largest $O(n + k \log n)$ time.
- `nsmallest(k, iterable)`: $O(n + k \log n)$ time.

Implement **Max Heap:** multiply each key by -1.
To have key-value pairs: `heappush(h, (5, 'hello'))`

13. Maps and Dictionaries

Dictionary: unique keys are mapped to associated values. Implemented with Python's **dict** class.
Map: more general notion of the abstract data type.

Stuff to add:
- Tree Map
- Sorted Map
- Multiset
- Multimap
- SkipList

…. Goodrich python pag 403

14. Sorting and Searching

A good approach is therefore to run through the different sorting algorithms and see if one applies particularly well.

14.1. Sorting Algorithms

Given a collection rearrange the elements so that they are ordered from smallest to largest.
Python has built-in support for sorting data, `sort` method of the list class.
Tim-sort: bottom-up merge-sort that takes advantage of some initial runs in the data while using insertion-sort to build additional runs.

14.1.1. Merge Sort

$O(n \log n)$

- **Divide**: If S has zero or one element, return S immediately.
  Otherwise divide it in two sequences, S1 and S2. S1 contains the first $\lfloor n/2 \rfloor$ elements, S2 contains the remaining $\lceil n/2 \rceil$ elements
- **Conquer**: Recursively sort sequences S1 and S2.
- **Combine**: merge the sorted sequences S1 and S2 into a sorted sequence.
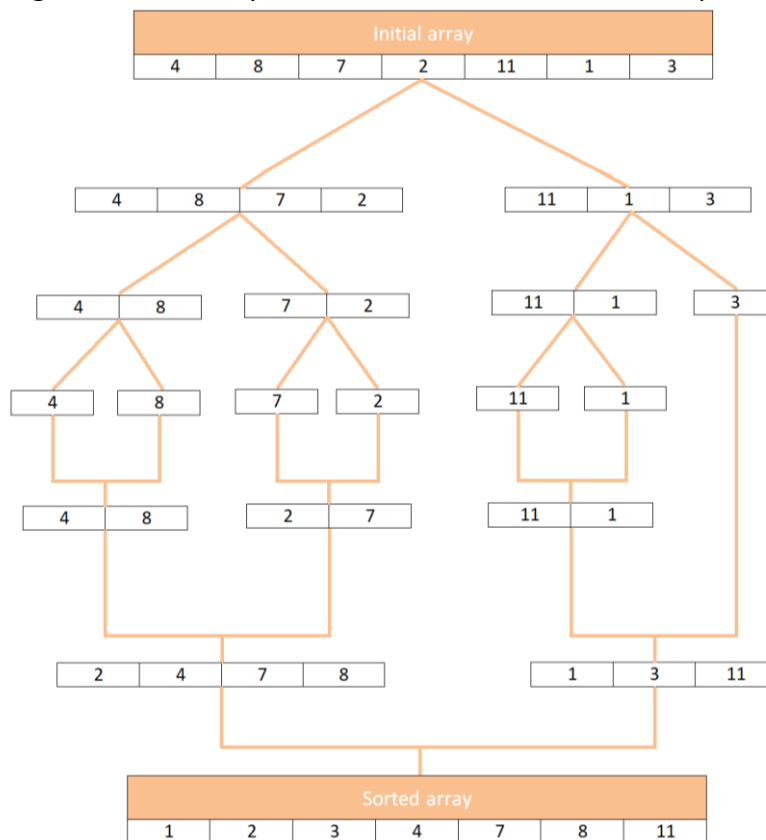
Merge-sort tree height $\lceil \log n \rceil$
Difficult to make in-place for arrays, and without that optimization the extra overhead of allocate a temporary array, and copying between the arrays is less attractive than in-place implementations of heap-sort and quick-sort for sequences that can fit entirely in a computer's main memory.
Is an excellent algorithm for situations where the input is stratified across various levels of the computer's memory hierarchy (e.g., cache, main memory, external memory).

```python
def merge_sort(nums):
    # bottom cases: empty or list of a single element.
    if len(nums) <= 1:
        return nums

    pivot = int(len(nums) / 2)
    left_list = merge_sort(nums[0:pivot])
    right_list = merge_sort(nums[pivot:])
    return merge(left_list, right_list)

def merge(left_list, right_list):
    left_cursor = right_cursor = 0
    ret = []
    while left_cursor < len(left_list) and right_cursor < len(right_list):
        if left_list[left_cursor] < right_list[right_cursor]:
            ret.append(left_list[left_cursor])
            left_cursor += 1
        else:
            ret.append(right_list[right_cursor])
            right_cursor += 1

    # append what is remained in either of the lists
    ret.extend(left_list[left_cursor:])
    ret.extend(right_list[right_cursor:])

    return ret
```

14.1.2. Quick-Sort

All the hard work is done before the recursive calls.

Divide S into subsequences, recur to sort each subsequence, and then combine the sorted subsequences by a simple concatenation.

- **Divide**: If S has at least two elements, select a specific element from S, the **pivot** (usually the last).
  Put element of S into three sequences
  L <-- elements < pivot
  E <-- elements = pivot
  G <-- elements > pivot
- **Conquer**: Recursively sort sequences L and G.
- **Combine**: Put back the elements into S in order by first inserting the elements of L, then those of E, and finally those of G.

1. Split using pivot x.
$E(=x)$
2. Recur.   2. Recur.
$L(<x)$ $G(>x)$
3. Concatenate.

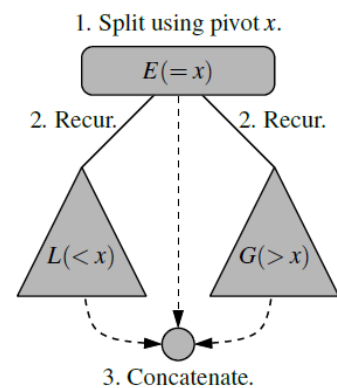**Randomized Quick-Sort:** pivot is random element of S

- **Time**: Average $O(n \log n)$, worst $O(n^2)$
- **Memory**: $O(\log n)$ (in place)

```python
def quicksort(lst):
    n = len(lst)
    qsort(lst, 0, n - 1)

def qsort(lst, lo, hi):
    """
    :param lst: the list to sort
    :param lo:  the index of the first element in the list
```

```
        :param hi:  the index of the last element in the list
        :return: the sorted list
        """
        if lo < hi:
            p = partition(lst, lo, hi)
            qsort(lst, lo, p - 1)
            qsort(lst, p + 1, hi)

    def partition(lst, lo, hi):
        """ Picks the last element hi as a pivot
         and returns the index of pivot value in the sorted array  """
        pivot = lst[hi]
        i = lo
        for j in range(lo, hi):
            if lst[j] < pivot:
                lst[i], lst[j] = lst[j], lst[i]
                i += 1
        lst[i], lst[hi] = lst[hi], lst[i]
        return i
```

14.1.3. Radix Sort

Is a sorting algorithm for **small integer keys**, **character strings**, or d-tuples of keys from a discrete range that takes advantage of the fact that integers have a finite number of bits. We iterate through each digit of the number, grouping numbers by each digit.

$O(kn)$, where n is the number of elements and k is the number of passes of the sorting algorithm.

```
import math
import itertools
def radix_sort(arr, w):
    # w is the number of buckets
    for i in range(int(round(math.log(max(map(abs, arr)), w)) + 1)):
        buckets = [[] for j in range(w)]
        for element in arr: buckets[element//w**i%w] += [element]
        arr = list(itertools.chain(*buckets))
    return [element for element in arr if element < 0] + [element for element in arr if element >= 0]
```

14.1.4. Bucket Sort

Time: $O(n + k)$ average, $O(n^2)$ worst
Space: $O(n)$

- Create n empty buckets (Or lists). $O(n)$
- For every element $A[i]$. $O(n)$
    - Insert $A[i]$ into $bucket[n*A[i]]$
- Sort individual buckets using insertion sort. $O(n)$ on average if numbers are uniformly distributed.
- Concatenate all sorted buckets. $O(n)$

14.1.5. Counting Sort

- **Time**: $O(n + k)$
- **Space**: $O(k)$

Elements are in range from 1 to k. It works by counting the number of objects having distinct key values (kind of hashing). Then doing some arithmetic to calculate the position of each object in the output sequence.

```
#Elements in range 0 to k (can be adapted to any range) - O(N+K) N>>K
def counting_sort(arr):
  k=max(arr)
  ht=[0]*(k+1)
  for e in arr: ht[e]+=1
```

```
idx=0
for n in range(k+1):
  count=ht[n]
  if count==0: continue
  for i in range(count): arr[idx+i]=n
  idx+=count
return arr
```

14.1.6. Selection Sort

- **Time**: $O(n^2)$
- **Space**: $O(1)$

Find the smallest element using a linear scan and move it to the front (swapping it with the front element). Continue doing this until all the elements are in place.

14.1.7. Bubble Sort

- **Time**: $O(n^2)$
- **Space**: $O(1)$

Start at the beginning of the array and swap the first two elements if the first is greater than the second. Then, we go to the next pair, and so on.

14.2. Searching Algorithms

14.2.1. Binary search
- First compare x to the midpoint of the array.
- If x is less than the midpoint, then we search the left half of the array.
- If x is greater than the midpoint, then we search the right half of the array.

repeat this process until we either find x or the subarray has size 0.

Iterative implementation Python
```python
def binarySearch(nums, target):
    if len(nums) == 0:
        return -1

    left, right = 0, len(nums) - 1
    while left <= right:
        mid = (left + right) // 2
        if nums[mid] == target: return mid
        elif nums[mid] < target: left = mid + 1
        else: right = mid - 1

    # End Condition: left > right
    return -1
```

When we can use binary search: if we can find an increasing/decreasing order in the solution space. Use binary search to guess the result, then check if this result it's correct. If its not we can find the solution in the left/right part, it its correct, we may find a better solution in the right/left part.

Bisect insertion (Python): way to do binary search in 1 line.
Module that provides support for maintaining a list in sorted order without having to sort the list after each insertion. Used also as binary search.
- `bisect.bisect_left(a, x, lo=0, hi=len(a))`: return the insertion point for x in a to maintain sorted order. The parameters lo and hi may be used to specify a subset of the list which should be considered; by default the entire list is used. If x is already present in a, the insertion point will be before (to the left of) any existing entries. The return value is suitable for use as the first parameter to `list.insert()` assuming that a is already sorted.
- `bisect.bisect_right(a, x, lo=0, hi=len(a))`
  `bisect.bisect(a, x, lo=0, hi=len(a))`
  Similar to bisect_left(), but returns an insertion point which comes after (to the right of) any existing entries of x in a.
- `bisect.insort_left(a, x, lo=0, hi=len(a))`
  Insert x in a in sorted order. This is equivalent to a.insert(bisect.bisect_left(a, x, lo, hi), x) assuming that a is already sorted. Keep in mind that the O(log n) search is dominated by the slow $O(n)$ insertion step.
- `bisect.insort_right(a, x, lo=0, hi=len(a))`
  `bisect.insort(a, x, lo=0, hi=len(a))`
  Similar to insort_left(), but inserting x in a after any existing entries of x

15. Paradigms

15.1. Divide and Conquer
Divide: If the input size is smaller than a certain threshold solves the problem directly and return the solution. Otherwise, divide the input data into two or more disjoint subsets.
Conquer: Recursively solve the subproblems
Combine: Take the solutions to the subproblems and merge them into a solution to the original problem.

Substructure property: property of the problem $\Pi \subseteq I \times S$ that allows to define the D&C.
It's realized by the functions:
- Divide $D: I \to I^*$
- Conquer $C: S^* \to S$

Downside of the D&C: it has no memory. When we go bottom up (conquer) the solution is not saved, so it could happen to re do some calculations. Solution: dynamic programming.

Complexity: $T(n) = \begin{cases} T_0 & , \ n \leq n_0 \\ T_D + T_R + T_C, & n > n_0 \end{cases}$

Recurrence: $T(n) = \begin{cases} T_0 & , \ n \leq n_0 \\ s(n) * T(f(n)) + w(n), & n > n_0 \end{cases}$
- $s(n) = $ # of sub instances (non base) generated
- $f(n) = $ size of sub instances (non base) generated $(f(n) < n \ \forall n > n_0)$
- $w(n) = $ work divide + conquer associated to an instance (non base) of size n

Iterate of a function: apply the function to itself. $f^{(l)}(n) = f(\dots f(n))$
Auxiliary function: $f^*(n, n_0) = \max\{k \geq 0: f^{(k)}(n) > n_0\}$ max index of iterate for which it return a value $> n_0$. Most commons:
- $n = b^k, b > 1, \ f(n) = n/b, \quad f^*(n, n_0) = \log_b(n/n_0) - 1$
- $n = a^{b^k}, \qquad f(n) = n^{1/b}, \quad f^*(n, n_0) = \log_b \log_a n - \log_b \log_a n_0 - 1$
- $n > 0, \qquad\quad f(n) = n - 1, \quad f^*(n, n_0) = n - n_0 - 1$

General Formula:

$$T(n) = \sum_{l=0}^{f^*(n, n_0)} \left(\left[\prod_{j=0}^{l-1} s\left(f^{(j)}(n)\right) \right] w\left(f^{(l)}(n)\right) \right) + \left[\prod_{j=0}^{f^*(n, n_0)} s\left(f^{(j)}(n)\right) \right] T_0$$

Livelli interni (# nodi livello l · Lavoro Nodo)   # foglie · lavoro foglie

Master Theorem: $T(n) = \begin{cases} T_0 & , \ n = 1 \\ aT(n/b) + w(n), & n > 1 \end{cases}$
- **Case 1:** $\Theta(n^{\log_b a})$ if $w(n) < n^{\log_b a}$
  Complexity is dominated by the # of leaves => reduce the # of sub-instances (Ex. reduce a)
- **Case 2:** $\Theta(n^{\log_b a} \log n)$ if $w(n) = n^{\log_b a}$
  Balanced strategy => find a new substructure property
- **Case 3:** $\Theta(w(n))$ if $w(n) > n^{\log_b a}$ and $a * w\left(\frac{n}{b}\right) < c * w(n)$
  Complexity is dominated by the work $w(n)$ at the root => reduce $w(n) = T_d(n) + T_c(n)$

15.2. Dynamic Programming

The problem may be solved using Dynamic programming if it has:

- **Overlapping Subproblems:** Those solutions to subproblems can be stored in a table so that these don't have to be recomputed.
- **Optimal Substructure:** optimal solution of the given problem can be obtained by using optimal solutions of its subproblems.

Identify if the problem can be solved with DP:

- Count something, often the number of ways, under certain conditions. (usually DFS + memoization)
- Minimize/max certain value (greedy)
- Yes/no (greedy)
- Maximum/longest, minimal/shortest value/cost/profit from doing operations on a sequence
- Partition a string/array into sub-sequences so that certain condition is met.
- Optimal way to play a game.
- Some probability problems.
- If greedy doesn't work, try DP
- All dynamic programming problems satisfy the overlapping subproblems, and most of them also the optimal substructure.

Decide the state:

- **State**: is the set of parameters that can uniquely identify a certain position or standing in the given problem. This set of parameters should be as small as possible to reduce state space.
- In a given position, what does matter so far?
- If only matter where we are, the state is just the position => dp[i].
- If also some other parameters matter => dp[i][k][…]

Formulate a relation among the states (Transition):

- how to calculate this state from the previous?
- List all the state(s) transitions with their respective conditions.
- Define the base case(s).
- Implement a naive recursive solution.

Adding memoization or tabulation for the state:

- Optimize the recursive solution to caching (memoization).
- Remove the overhead of recursion with a bottom-up approach (tabulation).
- Is a tabulation really needed can we use just a few variables?

Sometimes the bottom-up approach calculates unnecessary sub-instances.
Avoid double counting.

Memoization (Top-down) generic algorithm:

Init():

- Solve directly base cases
- Initialize the table with default values and base cases
- Call Rec()

Rec():

- Check if the instance is already solved
- If not compute it with the substructure property and save it to the table
- Return the solution of the instance

Patterns (Exercises for each one: [Leetcode](#))

Minimum (Maximum) Path to Reach a Target
- **Statement**: Given a target find minimum (maximum) cost / path / sum to reach the target.
- **Approach**: Choose minimum (maximum) path among all possible paths before the current state, then add value for the current state.
```
routes[i] = min(routes[i-1], routes[i-2], ... , routes[i-k]) + cost[i]
```

Distinct Ways
- **Statement**: Given a target find a number of distinct ways to reach the target.
- **Approach**: Sum all possible ways to reach the current state.
```
routes[i] = routes[i-1] + routes[i-2], ... , + routes[i-k]
```
Some questions point out the number of repetitions, in that case, add one more loop to simulate every repetition.

Merging Intervals
- **Statement**: Given a set of numbers find an optimal solution for a problem considering the current number and the best you can get from the left and right sides.
- **Approach**: Find all optimal solutions for every interval and return the best possible answer.
```
// from i to j
dp[i][j] = dp[i][k] + result[k] + dp[k+1][j]
```

DP on Strings
- **Statement**: Given two strings s1 and s2, return some result.
- **Approach**: Most of the problems requires a solution that can be accepted in O(n^2).

```
// i - indexing string s1
// j - indexing string s2
for (int i = 1; i <= n; ++i) {
    for (int j = 1; j <= m; ++j) {
        if (s1[i-1] == s2[j-1]) {
            dp[i][j] = /*code*/;
        } else {
            dp[i][j] = /*code*/;
        }
    }
}
```

```
With one string s the approach may vary
for (int l = 1; l < n; ++l) {
    for (int i = 0; i < n-l; ++i) {
        int j = i + l;
        if (s[i] == s[j]) {
            dp[i][j] = /*code*/;
        } else {
            dp[i][j] = /*code*/;
        }
    }
}
```

Decision Making
- **Statement:** Given a set of values find an answer with an option to choose or ignore the current value. (Use or not the current state)
- **Approach**: If you decide to choose the current value use the previous result where the value was ignored; vice-versa, if you decide to ignore the current value use previous result where value was used.
```
// i - indexing a set of values
// j - options to ignore j values
for (int i = 1; i < n; ++i) {
    for (int j = 1; j <= k; ++j) {
        dp[i][j] = max({dp[i][j], dp[i-1][j] + arr[i], dp[i-1][j-1]});
        dp[i][j-1] = max({dp[i][j-1], dp[i-1][j-1] + arr[i], arr[i]});
    }
}
```

Scansions of the 2D table

Row major	Column major	Principal Diagonal

`for i=0 to m:`
    `for j=0 to n:`
        `M[i][j]`
	`for i=0 to n:`
	    `for j=0 to m:`
	        `M[i][j]`
		`for i=0 to n: M[i][i]`

From Diagonal to upper right
`for c in range(0,C):`
  `for r in range(0,min(R,C-c)):`
    `col=r+c`
    `M[r][col]`
 	
`for l=0 to n:`
  `for i=0 to n-l+1: ???????`
    `j=i+l-1      ???????`
    `M[i][j]`
	From Diagonal to bottom left
	`for r in range(0,R):`
	  `for c in range(0,min(C,R-r)):`
	    `row=r+c`
	    `M[row][c]`
		From upper right to… ?????
		`for l=n-1 downto 1:`
		    `for i=1 to n-l+1:`
		        `j=i+l-1`
		        `M[i][j]`

Most common problems: https://algo.monster/problems/types_of_dynamic_programming

0-1 Knapsack Problem
Given two integer arrays `val[0..n-1]` and `wt[0..n-1]` which represent values and weights associated with n items. Also given an integer W which represents knapsack capacity, find the maximum value subset of `val[]` such that sum of the weights of this subset is smaller than or equal to W.
Substructure property:
- Maximum value obtained by $n-1$ items and W weight (excluding nth item).
- Value of nth item + maximum value obtained by $n-1$ items and W - the weight of the nth item (including nth item).

fractional knapsack: ???????

Unbounded Knapsack problem:
Given two integer arrays val[0..n-1] and wt[0..n-1] which represent values and weights associated with n items. Also given an integer W which represents knapsack capacity, calculate minimum amount that could make up this quantity exactly. We are allowed to use unlimited number of instances of an item.
    dp[i] = 0
    dp[i] = max(dp[i], dp[i-wt[j]] + val[j]
        where j varies from 0 to n-1 such that: wt[j] <= i
    result = d[W]

[LeetcodePoblems](#)

@functools.lru_cache(None)

15.3. Greedy

Problem of DP: we have a set of choices which final solution is obtained by composing those ones. But those choices can be done after having solved a numerous amount of instances.

How it works:
- Make a **Greedy choice**: we choose without computing sub-instances. Locally optimal.
- Find the remaining sub-instance: by **cleaning up the instance** by deleting incompatible elements.
- **Solve recursively** the sub-instance found.

Downsides of Greedy:
- Often does not produce an optimal solution => always search for a counter case
- Correctness analysis is difficult

Activity Selection Problem
Given:
- Set of activities $S = \{[s_0, f_0), \dots, [s_n, f_n)\}$
- Activities are sorted by ascending finish time.

Problem: determine the largest $A \subseteq S$ of compatible activities.
Greedy choice: select the activity that end first
Clean-Up: remove activities not compatibles with the greedy choice.

Huffman Code
Given: an alphabet C and a set of frequencies $f: C \to \mathbb{Q}^+$
Problem: determine the prefix code associated at the Trie T that minimizes $B(T) = \sum_{c \in C} d_T(c) f(c)$
Greedy choice: merge trees of smaller cumulative frequencies.
Clean-Up: substitute the 2 trees with the merge of them, with Cumulative Freq= sum of the 2.
$O(n \log n)$
The trie is optimal (full binary tree, every internal node has 2 children)
Uses an min heap.

….GOODRICH pag 601 ….

TO ADD:
- An activity-selection problem - Minimum Number of Arrows to Burst Balloons
- Elements of the greedy strategy
- Huffman codes - Construct Huffman Tree, Google | Onsite | Software Engineer | Huffman Coding Algorithm, Minimum Cost Tree From Leaf Values
- Matroids and greedy methods - Matroid intersection in simple words
- A task-scheduling problem as a matroid - Task Scheduler

15.4. Prune and Search

Selection Problem: selecting the kth smallest element from an unsorted collection of n elements. Can be achieved in $O(n)$

Prune-and-search:
- Prune away a fraction of the n objects
- Recursively solve the smaller problem.
- When we have finally reduced the problem to one defined on a constant-sized collection of objects, we then solve the problem using some brute-force method.

Randomized quick-select
- Pick a "pivot" element from S at random and use this to subdivide S into three subsequences L, E, and G, storing the elements of S less than, equal to, and greater than the pivot, respectively.
- In the prune step, we determine which of these subsets contains the desired element, based on the value of k and the sizes of those subsets.
- We then recur on the appropriate subset

```python
def quick_select(S,k):
  if len(S) == 1: return S[0]

  pivot=random.choice(S)

  L=[x for x in S if x < pivot]
  E=[x for x in S if x == pivot]
  G=[x for x in S if x > pivot]

  if k <= len(L): return quick_select(L, k)
  elif k <= len(L) + len(E): return pivot
  else:
    j=k - len(L) - len(E)
    return quick_select(G, j)
```

16. Other knowledge

16.1. Bit Manipulation

Base 2: $1001 = 1 * 2^3 + 1 * 2^0$
Base 10: $123 = 1 * 10^2 + 2 * 10^1 + 3 * 10^0$

Integer codification:
- Most significant bit is used as sign. (0=positive)
- 1-complement: transform 0 in 1 and 1 in 0.
- **2-complement**: 1-complement +1

Convert negative number in binary: convert the positive in binary and perform 2-complement.

Shifting:
- Right Shift: divides by 2
- Left Shift: multiply by 2
- **Arithmetic Shift:** divides by two. Shift values to the right but **fill in** the new bits with the value of the **sign bit**.
- **Logical Shift**: `>>1` `(<<1)`. Shift the bits and put a 0 in the most(least) significant bit.

Bitwise operations:

```
x ^ 0s = x        x & 0s = 0        x | 0s = x
x ^ 1s = ~x       x & 1s = x        x | 1s = 1s
x ^ x = 0         x & x = x         x | x = x
```

Xor is commutative and associative. In a sequence of xor we can remove duplicate numbers.

Common Bit Tasks: (bit i from the right LSB)
- **Get the i Bit:** `(num&(1<<i))!=0`
  - left shifts 1 over by i bits, creating a value that looks like 00010000.
  - Perform an AND with num.
  - Compare that to 0. If is not zero, then bit i must have a 1. Otherwise, bit i is a 0.
- **Set the i Bit:** `num=num|(1<<i)`
  - shifts 1 over by i bits, creating a value like 00010000.
  - perform an OR with num
- **Clear Bit**: `num=num&(~(1<<i))`
  - create a number like 11101111 by creating the reverse (00010000) and negate it.
  - Perform an AND with num.
- **Clear all bits from MSB to i (inclusive):** `num=num&((1<<i)-1)`
  - shifts 1 over by i bits, creating a value like 00010000.
  - Subtract 1 from it, giving us a sequence of 0s followed by i 1s. (00001111)
  - Perform an AND with num to leave just the last i bits
- **Clear all bits from i to LSB (inclusive)**: `num=num&((-1)<<(i+1))`
  - Create a sequence of all 1s (which is -1)
  - Shift it left by i + 1 bits.
  - Perform an AND with num
- **Count number of "1"**: `bin(n).count("1")`

70

- **Set union:** `A | B`
- **Set intersection:** `A & B`
- **Set subtraction:** `A & ~B`
- **Extract lowest set bit:** `A&(-A) or A&~(A-1) or x^(x&(x-1))`
- **Extract lowest unset bit:** `~s&(s+1)`
- **Set last bit to 0:** `A&(A-1)`
- **Get all 1-bits:** `~0`
- **Check if only one bit is set in a number:** `(num & (num-1) == 0)`
- **Toggle kth bit:** `s ^= (1<<k)`
- **Multiply by 2^n:** `s<<n`
- **Divide by 2^n:** `s>>n`
- **Swap Values:** `x^=y; y^=x; x^=y`
- **Check if odd (x%2):** `x&1==1`
- **Check if integers are equal:** `x^y==0`
- **Check if integers have same sign:** `(x ^ y) >= 0`
- **Flip the sign of an integer:** `x = ~x + 1`
- **Find the minimum:** `y^(x^y) & -(x<y)`
- **Find the maximum:** `x^(x^y) & -(x<y)`

16.2. Recursion

Good hint that a problem is recursive is that it can be built of subproblems.

When you hear a problem beginning with the following statements, it's often a good candidate for recursion: "Design an algorithm to compute the nth …", "Write code to list the first n …", "Implement a method to compute all …"

How to Approach:
- **Bottom-Up:** knowing how to solve the problem for a simple case, like a list with only one element. Then we figure out how to solve the problem for two elements, then for three elements, and so on.
- **Top-Down:** think about how we can divide the problem for case N into subproblems. Be careful of overlap between the cases.
- **Half-and-Half:** divide the data set in half. Like merge sort.

Recursive algorithms can be very space inefficient. Problem of depth n uses at least O(n) memory. All recursive algorithms can be implemented iteratively, although sometimes the code to do so is much more complex.

From Recursive to Iterative:
- push the parameters that would normally be passed to the recursive function onto a stack.

```
stack=[]
stack.append(first_object)
while stack:
  #Do something
  my_object = stack.pop()
  # Push other objects on the stack.
}
```
- If you have more than one recursive call inside and you want to preserve the order of the calls, you have to add them in the reverse order to the stack:

foo(first);	replaced by	stack.push(second);
foo(second);	$\Rightarrow$	stack.push(first);

16.3. Backtracking

Backtracking is a general algorithm for finding all (or some) solutions to some computational problems (notably Constraint satisfaction problems or CSPs), which incrementally builds candidates to the solution and abandons a candidate ("backtracks") as soon as it determines that the candidate cannot lead to a valid solution.

Because these problems usually involve permutations or combinations of choices they are also known as combinatorial search problems.

Backtracking is a technique used to build up to a solution to a problem incrementally. These "partial solutions" can be phrased in terms of a sequence of decisions. Once it is determined that a "partial solution" is not viable (i.e. no subsequent decision can convert it into a solution) then the backtracking algorithm retraces its step to the last viable "partial solution" and tries again.

Visualizing the decisions as a tree, backtracking has many of the same properties of depth-first search. The difference is that depth-first search will guarantee that it visits every node until it finds a solution, whereas backtracking doesn't visit branches that are not viable.

Because backtracking breaks problems into smaller subproblems, it is often combined with dynamic programming, or a divide-and-conquer approach.

Important concepts:
- State: A "partial solution" to the problem.
- Rejection criterion: A function that rejects a partial solution as "unrecoverable". i. e. no sequence of subsequent decisions can turn this state into a solution. Without rejection criteria, backtracking is the same as depth-first search.
- Viable: A state that may still lead to a solution. This reflects what is known "at this stage". All states that fail the rejection criteria are not viable. If we have a state that is initially viable, and find that all paths to leaves eventually terminate at a state that fail the rejection criteria, the state will become non-viable.
- Heuristic: Backtracking will (eventually) look at all the different viable nodes by recursively applying all the possible decisions. A heuristic is a quick way of ordering which decisions are likely to be the best ones at each stage, so that they get evaluated first. The goal is to find a solution early (if one exists).
- Pruning: The concept of determining that there are no nodes / states along a particular branch, so it is not worth visiting those nodes.

Related Concepts:
- Depth-first search: A backtracking algorithm is conceptually very similar to depth-first search (DFS). A DFS will "roll back" the current state to a node up the tree once it reaches a leaf node, and is guaranteed to find a solution (or search every node). Backtracking can be thought of as DFS with early pruning.
- Recursion: Backtracking is often implemented using recursion, so it helps to be familiar with how to write recursive functions.
- Stacks: If you need a lot of nested recursive calls, trying to use simple recursion might result in a stack overflow error. You can mimic recursion by using a stack data structure.

A recursive implementation of a backtracking algorithm takes the general form:

```
function doBacktrack( current ):
    if current is a solution:
        return current
    for each decision d from current:
        new_state <- state obtained from current by making decision d
        if new_state is viable:
            sol <- doBacktrack(new_state)
            if sol is not None:
                return sol
    # indicate there is no solution
    return None
```

Iterative implementation:

```
function doBacktrackIterative(start):
    stack <- initialize a stack
    stack.push(start)

    while (stack not empty):
        current = stack.pop()

        if current is a solution:
            return current

        for each decision d from current:
            new_state <- state obtained from current by making decision d
            if new_state is viable:
                stack.push(new_state)

    return None
```

…..

16.4. Math

Proof by induction:
Task: Prove statement $P(k)$ is true for all $k \geq b$.
- Base Case: Prove the statement is true for $P(b)$. This is usually just a matter of plugging in numbers.
- Assumption: Assume the statement is true for $P(n)$.
- Inductive Step: Prove that if the statement is true for $P(n)$, then it's true for $P(n + 1)$.

16.4.1. Number Theory
Number theory is a branch of mathematics that studies integers.

a is called a **factor** or a **divisor** of a number b if a divides b.
$n > 1$ is a **prime** if its only positive factors are 1 and n.
For every number $n > 1$, there is a unique **prime factorization** $n = p_1^{\alpha_1} * \dots * p_k^{\alpha_k}$
where $p_1, \dots, p_k$ are distinct primes and $\alpha_1, \dots, \alpha_k$ are positive numbers.

Number of factors: $\tau(n) = \prod_{i=1}^{k}(\alpha_i + 1) \; O(\sqrt{n})$

```
def divisors(n): #O(sqrt(N))
    i = 1
    result = 0
    while (i * i < n):
        if (n % i == 0): result += 2
        i += 1
    if (i * i == n):result += 1
    return result
```

Sum of factors $\sigma(n) = \prod_{i=1}^{k}(1 + p_i + \dots + p_i^{\alpha_i}) = \prod_{i=1}^{k} \frac{p_i^{\alpha_i+1}-1}{p_i-1}$
Product of factors $\mu(n) = n^{\tau(n)/2}$

Perfect number: if $n = \sigma(n) - n$

Greatest common divisor $\gcd(a, b)$: is the greatest number that divides both a and b.
Least common multiple $\mathrm{lcm}(a, b)$: is the smallest number that is divisible by both a and b.
$$\mathrm{lcm}(a, b) = \frac{ab}{\gcd(a,b)}$$
Euclidean algorithm: $O(\log n)$ where $n = \min(a, b)$. $\gcd(a, b) = \begin{cases} a, & b = 0 \\ \gcd(b, a \bmod(b)), & b \neq 0 \end{cases}$

```
gcd(int a, int b) {
    if b == 0: return a
    return gcd(b, a%b)
}
```

Number of common divisors $O(\log n + n^{1/2})$
Calculate the gcd of given two numbers, and then count divisors of that gcd.

```
n = 0
for i in range(1, min(a, b)+1):
    if a % i == b % i == 0:
        n+= 1
print(n)
```

16.4.2. Prime Numbers

Checking for Primality: $O(\sqrt{n})$

```python
def primality(n): #O(sqrt(N))
    i = 2
    while (i * i <= n):
        if (n % i == 0):return False
        i += 1
    return True
```

Generating a List of Primes: The **Sieve of Eratosthenes**. works by recognizing that all non-prime numbers are divisible by a prime number.

```python
if n<2: return 0
primes=[True]*(n+1)
primes[0]=False
primes[1]=False

for i in range(2, int(n**0.5)+1):
  if primes[i]:
     #all multiples of i are not primes
     for j in range(2*i, n+1, i):
       primes[j] = False
```

16.4.3. Discrete Calculus

Cartesian product between sets: $A_1 \times \dots \times A_n = \{(a_1, \dots, a_n), a_i \in A_i\}$

$I_n = \{1, \dots, n\}$

Labeling of a set $X: I_n \to X$. The first element of X is labeled with 1, second with 2, …

k-**sequence** of I_n: k-tuple **ordered** $(a_1, \dots a_k)$ of elements not necessarily unique.

k-**collection** of I_n: **non ordered** family of k elements of I_n not necessarily unique.
($k_1$copies of 1,…,k_n copies of n, with $k_1 + \dots + k_n = k$)

$$[\underbrace{1, \dots, 1}_{k_1}, \dots, \underbrace{n, \dots, n}_{k_n}]$$

Without repetitions:
- k-sequences: $S(n,k) = \dfrac{n!}{(n-k)!}, k \leq n$
- k-collections: $C(n,k) = \dfrac{S(n,k)}{k!} = \dfrac{n!}{k!(n-k)!} = \binom{n}{k}$ (n choose k)

With repetitions:
- k-sequences: $S\big((n,k)\big) = n^k$
- k-collections: $C\big((n,k)\big) = C(n-1+k, k) = \binom{n-1+k}{k}$

Collections with constraints:
- # of solution of $x_1 + \dots + x_n = k$ with $x_i \geq l_i$ => $C\Big(\big(n, k - (l_1 + \dots + l_n)\big)\Big)$
- # of solution of $x_1 + \dots + x_n \leq k$ with $x_i \geq l_i$ => $C\big((n+1, k)\big)$

Occupancy constrains

$k = k_1 + \dots + k_n$
- k-**sequence** of I_n with sequence of occupancy $(k_1, \dots, k_n)$: $k_1$repetitions of 1,…, k_n repetitions of n

  $S\big(n, k, (k_1, \dots, k_n)\big) = \dfrac{k!}{k_1! \dots k_n!} = P(a_1, \dots, a_k)$ **permutations**

  Elements repeats $k_1, \dots, k_n$ times knowing exactly which ones.

Itertools (python)
```
from itertools import permutations
# Get all permutations of length 2
perm = permutations([1, 2, 3], 2)
```
- It generates $\frac{n!}{(n-k)!}$ tuples
- Elements are treated as unique based on their position, not on their value. (It's a k-sequence)

```
from itertools import combinations
# Get all combinations of [1, 2, 3] and length 2
comb = combinations([1, 2, 3], 2)
```
- "*n choose k*"
- Elements are treated as unique based on their position, not on their value.
- Are emitted in lexicographic sort order of input. So, if the input list is sorted, the combination tuples will be produced in sorted order.
- It generates $\frac{n!}{k!(n-k)!}$ tuples

```
from itertools import combinations_with_replacement
# Get all combinations of [1, 2, 3] and length 2
comb = combinations_with_replacement([1, 2, 3], 2)
```
- n choose k with repetitions
- Elements are treated as unique based on their position, not on their value.
- It generates $\binom{n-1+k}{k}$

```
from itertools import product
product(A, B) returns the same as ((x,y) for x in A for y in B)
```

16.4.4. Probability

Probability is a real number between 0 and 1 that indicates how probable an event is. If an event is certain to happen, its probability is 1, and if an event is impossible, its probability is 0.
Calculation of a probability:
- $P = \frac{\text{number of desired outcomes}}{\text{total number of outcomes}}$
- Or simulate the process that generates the event.

Event: is a set $A \subset X$, where X contains all possible outcomes and A is a subset of outcomes.
Each outcome x is assigned a probability $p(x)$
Probability of an event $P(A) = \sum_{x \in A} p(x)$
- $P(X) = 1$

Complement $\bar{A}$ means "A does not happen", $P(\bar{A}) = 1 - P(A)$
Intersection $A \cap B$ means "A and B happen"
Union $A \cup B$ means "A or B happen", $P(A \cup B) = P(A) + P(B) - P(A \cap B)$
- If the events A and B are disjoint, i.e., $A \cap B = \emptyset$ then $P(A \cup B) = P(A) + P(B)$

Conditional probability $P(A|B) = \frac{P(A \cap B)}{P(B)}$ is the probability of A assuming that B happens.

Hence, when calculating the probability of A, we only consider the outcomes that also belong to B.

- $P(A \cap B) = P(B)P(A|B)$

Events A and B are **independent** if $P(A|B) = P(A)$ and $P(B|A) = P(B)$
which means that the fact that B happens does not change the probability of A, and vice versa.
In this case, the probability of the intersection is $P(A \cap B) = P(A)P(B)$

Reservoir Sampling

Choose k entries from n numbers. Make sure each number is selected with the probability of k/n.

- Choose $1, 2, 3, \ldots, k$ first and put them into the reservoir.
- For $k + 1$, pick it with a probability of $k/(k + 1)$, and randomly replace a number in the reservoir.
- For $k + i$, pick it with a probability of $k/(k + i)$, and randomly replace a number in the reservoir.
- Repeat until $k + i$ reaches n

Mutual exclusivity…
Random rejection….
…. Competitive Programming Handbook chapter 24 ….

16.5. Regex

```
import re
```

```
re.findall("ai", txt)
```
- Returns a list containing all matches
- The list contains the matches in the order they are found.
- If no matches are found, an empty list is returned

```
re.search("\s", txt)
```
- Searches the string for a match, and returns a Match object if there is a match.
- If there is more than one match, only the first occurrence of the match will be returned.
- If no matches are found, the value None is returned

```
re.split("\s", txt)
```
- Returns a list where the string has been split at each match
- You can control the number of occurrences by specifying the `maxsplit` parameter

```
re.sub("\s", "9", txt)
```
- replaces the matches with the text of your choice
- You can control the number of replacements by specifying the `count` parameter

Match Object
- is an object containing information about the search and the result.
- If there is no match, the value None will be returned, instead of the Match Object.
- `.span()` returns a tuple containing the start-, and end positions of the match.
- `.string` returns the string passed into the function
- `.group()` returns the part of the string where there was a match

Character classes		Quantifiers & Alternation	

`.`
`\w \d \s`
`\W \D \S`
`[abc]`
`[^abc]`
`[a-g]`
	any character except newline
	word, digit, whitespace
	not word, digit, whitespace
	any of a, b, or c
	not a, b, or c
	character between a & g
		`a* a+ a?`
		`a{5} a{2,}`
		`a{1,3}`
		`a+? a{2,}?`
		`ab|cd`
			0 or more, 1 or more, 0 or 1
			exactly five, two or more
			between one & three
			match as few as possible
			match ab or cd

Anchors		**Groups & Lookaround**	

`^abc$`
`\b \B`
	start / end of the string
	word, not-word boundary
		`(abc)`
		`\1`
		`(?:abc)`
		`(?=abc)`
		`(?!abc)`
			capture group
			backreference to group #1
			non-capturing group
			positive lookahead
			negative lookahead

Escaped characters			
`\. \* \\` `\t \n \r`	escaped special characters tab, linefeed, carriage return		

16.6. Pattern Matching

16.6.1. Single-pattern algorithms

Given a text string T of length n and a pattern string P of length m find whether P is a substring of T. Return the lowest index at which the pattern begins, or −1 if the pattern is not found.
OR find all the occurrences of the pattern in the text.

Brute Force: $O(mn)$

```
#Return the lowest index of T at which substring P begins (or else -1).
def match_bf(T, P):
  n, m = len(T), len(P)
  for i in range(n-m+1): # try every potential starting index within T
    k = 0 # an index into pattern P
    while k < m and T[i + k] == P[k]: # kth character of P matches
      k += 1
      if k == m: # if we reached the end of pattern,
        return i # substring T[i:i+m] matches P
  return -1
T="abacaabaccabacabaabb"
P="abacab"
print(match_bf(T,P))
```

Knuth-Morris-Pratt: $O(n + m)$

Precompute self-overlaps between portions of the pattern so that when a mismatch occurs at one location, we immediately know the maximum amount to shift the pattern before continuing the search.

Precompute a **failure function** f that indicates the proper shift of P upon a failed comparison. $O(m)$
$f(k) = $ length of the longest prefix of P that is a suffix of $P[1:k+1]$

k	0	1	2	3	4	5	6	7	8	9	10	11
$P[k]$	a	m	a	l	g	a	m	a	t	i	o	n
$f(k)$	0	0	1	0	0	1	2	3	0	0	0	0
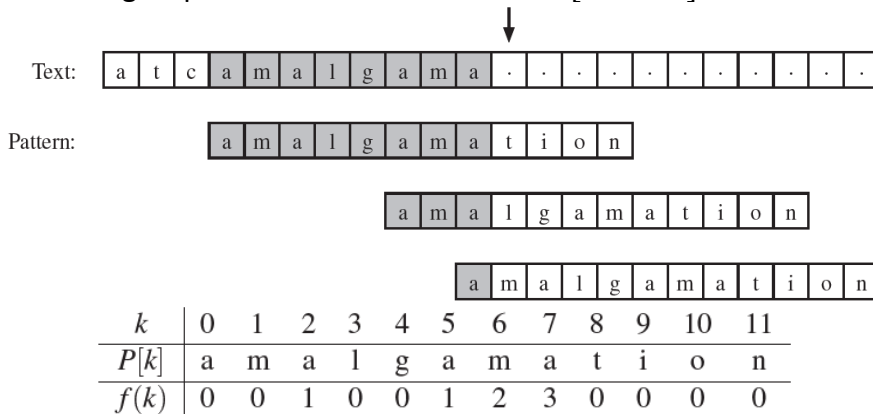
```
def kmp_table(P):
  m = len(P)
  table = [0]*m # by default, presume
overlap of 0 everywhere
  j = 1
  k = 0
  while j < m: # compute f(j) during
this pass, if nonzero
    if P[j] == P[k]: # k+1 characters
match thus far
      table[j] = k + 1
      j += 1
      k += 1
    elif k > 0: # k follows a matching
prefix
      k = table[k-1]
    else: # no match found starting at j
      j += 1
  return table
```

```
#Return the lowest index of T at which substring
P begins (or else -1).
def kmp(T, P):
  n, m = len(T), len(P)
  if m == 0: return 0 # trivial search for empty
string
  table = kmp_table(P) # rely on utility to
precompute
  j = 0 # index into text
  k = 0 # index into pattern
  while j < n:
    if T[j] == P[k]: # P[0:1+k] matched thus far
      if k == m - 1: # match is complete
        return j - m + 1
      j += 1 # try to extend match
      k += 1
    elif k > 0:
```

```
                  k = table[k-1] # reuse suffix of P[0:k]
              else:
                  j += 1
      return -1
```

Rabin-Karp: $O(n + m)$ expected. $O(nm)$ worst.

If two strings are the same, they must have the same hash value.

So, we precompute the hash of the pattern P and then for each substring of size m in T we compute the hash, and finally check if the hash(P)==hash(T[i:i+m)).

With a naive hash function, computing the hash(T[i:i+m)) require $O(m)$.

For speed, the hash must be computed in constant time. The trick is that with a good hash function the current hash value can be used to compute the next hash value in constant time.

Polynomial rolling hash function: $\text{hash}(s) = \left(\sum_{i=0}^{n-1} s[i] * p^i \right) \bmod k$ where:

- p a prime number roughly equal to the number of characters in the input alphabet. Precomputing the powers of p can boost performances.
- k large prime number ($k = 10^9 + 9$)

$h_0 = \left(\sum_{i=0}^{m-1} s[i] * p^i \right) \bmod k$, where m is the pattern length.

$h_i = \ldots$

<div style="background-color:orange">

SISTEMAAAAAAAAAAAAAAAAAAA

Using polynomial hashing, we can calculate the hash value of any substring of a string s in O(1) time after an O(n) time preprocessing. The idea is to construct an array h such that h[k] contains the hash value of the prefix s[0...k]. The array values can be recursively calculated as follows:

- h[0] = s[0]
- h[k] = (h[k-1]A+s[k]) mod B

In addition, we construct an array p where p[k] = A^k mod B:

- $p[0] = 1$
- $p[k] = (p[k-1]A) \bmod B$

Constructing these arrays takes O(n) time. After this, the hash value of any substring s[a...b] can be calculated in O(1) time using the formula: $(h[b] - h[a-1]p[b-a+1]) \bmod B$. Assuming that a>0. If a=0, the hash value is simply h[b].

</div>

Basic Idea	Implementation with polynomial hashing
```python\ndef RabinKarp(s, p):\n n, m = len(s), len(p)\n hp = hash(p) # O(m) only executed once\n for i in range(0, n-m+1):\n hs = hash(s[i:i+m]) # A good hash function can calculate this in O(1) knowing the previous hash value\n if hs == hp:\n if s[i:i+m] == p[0:m]: # O(m), but is expected to be executed very few times\n return i\n return -1 # not found\n```	```python\ndef RabinKarp(T,P):\n n, m = len(T), len(P)\n if m>n:return []\n\n #Lower case English letters (26)\n p = 31; #prime number close to alphabet length\n k = 1e9 + 9;#large prime number\n\n #Precompute powers of p\n p_pow=[0]*n\n p_pow[0]=1\n for i in range(1,n):p_pow[i]=(p_pow[i-1]*p)%k\n\n #compute polynomial hash of the entire text\n h=[0]*(n+1)\n for i in range(n):\n h[i+1]=(h[i]+(ord(T[i])-97+1)*p_pow[i])%k\n\n #compute polynomial hash of the pattern\n h_p=0\n for i in range(m):\n h_p=(h_p+(ord(P[i])-97+1)*p_pow[i])%k\n```

```
            occurences=[]
            for i in range(n-m+1):
              cur_h=(h[i+m] + k - h[i]) % k #compute current hash from the
            previous
              if (cur_h == h_p * p_pow[i] % k):
                occurences.append(i)
            return occurences
```

16.6.2. Multi-Pattern Algorithms

Brute force with Trie: $O(nm)$
- Put all patterns in a trie. $O(sum\ patterns\ length)$
- For each index of the Text: transverse the trie.

Rabin-Karp: $O(n + km)$
Main idea: (with patterns all the same length)
```
def RabinKarpMulti(s, subs):
    n, m = len(s),  min([len(p) for p in subs])
    hsubs = set()
    for sub in subs:
        hsubs.add(hash(sub[0:m]))
    for i in range(0, n-m+1):
        hs = hash(s[i:i+m])
        if hs in hsubs and s[i:i+m] in subs:
            return i
    return -1 # not found
```

Aho-Corasick $O(n + m + z)$ where m is the sum the patterns length and z is the count of matches.
Main idea: extension of Knuth-Morris-Pratt to multiple patterns. Link Link2
Construct a trie of the patterns and then adds a **suffix link** for each node.
Suffix (failure) link of a node v: connects v to the node p. Where p is the longest suffix of the string represented by v excluding itself. The longest suffix of the root is itself. It can be found following the father of v up till the root (this is the suffix), while at the same time from the root to the node p.
This special trie now represent a **finite deterministic automaton** that can scan through the text in linear time.
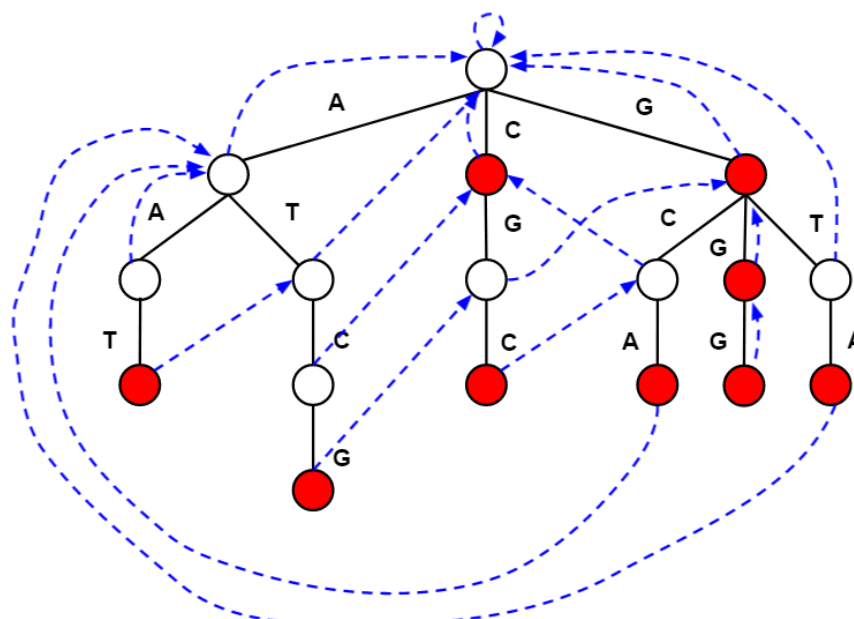
Dictionary links: allows to use patterns that are substring of other patterns. Otherwise in the linear scan some patters are not found.

Given a current node u, follow failure links from u to the root, and the first time you encounter a word node v along that failure link path, create a dictionary link from u to v, and stop for this node. If you never encounter a word node from the failure link path from u to the root, u does not have any dictionary links.

In the automata: You would keep following dictionary links until there are no more to follow. For example, consider the Aho-Corasick Automaton constructed from the following words: A, AA, AAA, AAAA, etc. There will be a dictionary link from AAAA to AAA, one from AAA to AA, and one from AA to A.

In the linear scan when a word node is found, if it has a dictionary link, follow it and mark also the other word node as visited and continue following the dictionary links if the node has one.
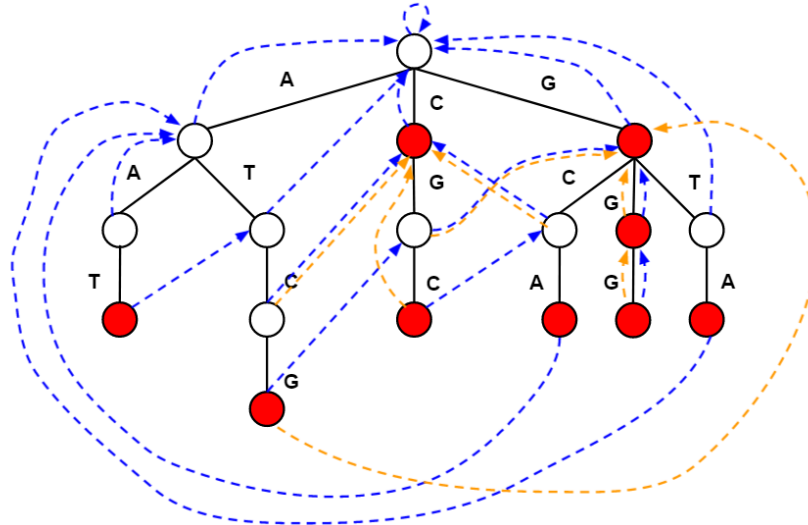
Aho-corasick summary:
- **Build a trie** with the patterns.
- **Add suffix links for each node:** BFS, for each node follow the father till reach the root to find the node that is the longest suffix.
- Linear scan trough the text: starting from the root, if the current letter is in the children of the current node, go to that child, otherwise follow the suffix link and retry the letter.
- If some patterns are substring of other ones, we have to **Add dictionary links**: for each node follow the failure links to the root, if there is a word node add a dictionary link between these two nodes, otherwise the node won't have a dictionary link, stop at the first word node you encounter.
- Linear scan trough the text: starting from the root, if the current letter is in the children of the current node, go to that child, otherwise follow the suffix link and retry the letter. If the current node is a word node, mark this word as found, if this node has a dictionary link follow the link and mark the word node, continue until you end up in node with no dictionary links, but for the next letter continue from the initial node.

Dynamic programming for generating failure links
We can do a preorder traversal of the automaton where for each :
If u is the root, create a failure link from u to itself
Otherwise, let p denote the parent of u and let f denote the target of the failure link from p:

- If f has a child labeled by the same letter as the MWT edge going into u, create a failure link from u to that child of f
- Else if f is not the root, recurse and set f to be the target of the failure link from f
- Else, create a failure link from u to the root

CODEEEEEEE
…

16.7. Game Theory

In this chapter, we will focus on **two-player games** that do **not** contain **random** elements.

Game States:
- **Winning state:** is a state where the player will win the game if they play optimally.
- **Losing state:** is a state where the player will lose the game if the opponent plays optimally.
- We can classify all states of a game as a winning state or a losing state.
- If there is a move that leads from the current state to a losing state, the current state is a winning state, and otherwise the current state is a losing state. Using this observation, we can classify all states of a game starting with losing states where there are no possible moves.

Base															
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
L	W	W	W	L	W	W	W	L	W	W	W	L	W	W	W
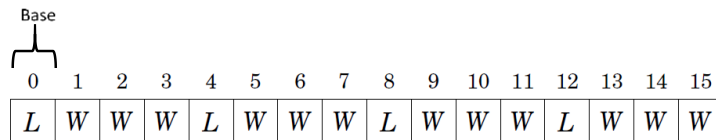
16.7.1. Nim game

Is a simple game that has an important role in game theory, because many other games can be played using the same strategy.
- *There are n heaps, each heap contains some number of sticks. The 2 players move alternately, and on each turn, the player chooses a heap that still contains sticks and removes any number of sticks from it. The winner is the player who removes the last stick.*
- States: $[x_1, x_2, \ldots, x_n]$, where x_k denotes the number of sticks in heap k.
- State [0,0,...,0] is a losing state, because it is not possible to remove any sticks, and this is always the final state.

Classify any nim state by calculating the **nim sum**: $s = x_1 \oplus \ldots \oplus x_n$ where $\oplus$ is XOR
- Nim sum = 0 is a losing state, otherwise is a winning state

16.7.2. Sprague-Grundy theorem

Generalizes the strategy used in nim to all games that fulfil the following requirements:
- There are two players who move alternately.
- The game consists of states, and the possible moves in a state do not depend on whose turn it is.
- The game ends when a player cannot make a move.
- The game surely ends sooner or later.
- The players have complete information about the states and allowed moves, and there is no randomness in the game.

The idea is to calculate for each game state a Grundy number that corresponds to the number of sticks in a nim heap. When we know the Grundy numbers of all states, we can play the game like the nim game.

Grundy number of a game state $= \text{mex}(\{g_1, \ldots, g_n\})$ where
- $g_1, \ldots, g_n$ are the Grundy numbers of the states to which we can move
- mex function returns the smallest nonnegative number that is not in the set. $\text{mex}(\emptyset) = 0$
- The Grundy number of a losing state is 0, and the Grundy number of a winning state is a positive number.
- If $= 0$: we can only move to states whose Grundy numbers are positive
- If $= x > 0$: we can move to states whose Grundy numbers include all numbers $0, 1, \ldots, x - 1$.

16.7.3. Minimax

Minimax is a kind of backtracking algorithm that is used in decision making and game theory to find the optimal move for a player, assuming that your opponent also plays optimally.

There are two players called maximizer and minimizer.
The maximizer tries to get the highest score possible while the minimizer tries to do the opposite and get the lowest score possible.
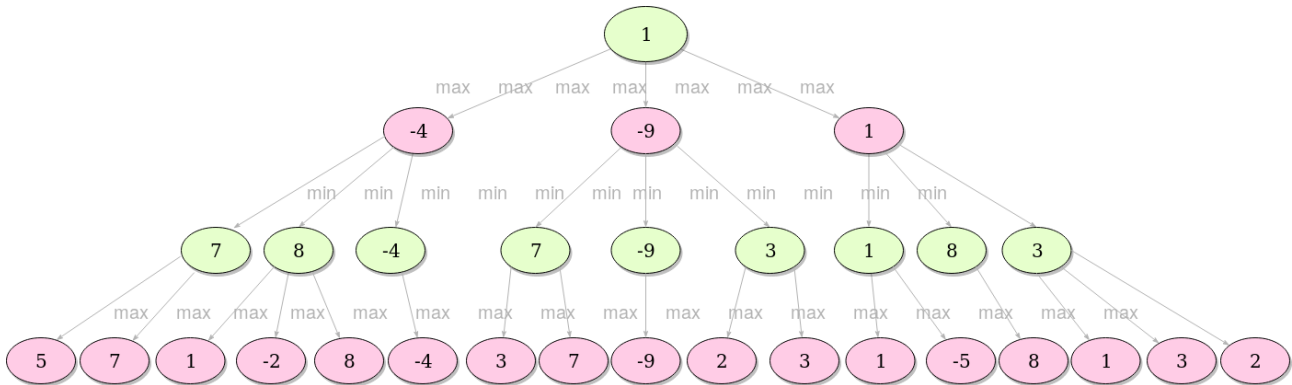
Every board state has a value associated with it. In a given state if the maximizer has upper hand then, the score of the board will tend to be some positive value. If the minimizer has the upper hand in that board state then it will tend to be some negative value. The values of the board are calculated by some heuristics which are unique for every type of game.

Heuristic Function (Evaluation Function): calculates the value of the board depending on the placement of pieces on the board. Is unique for every type of game. The basic idea behind the evaluation function is to give a high value for a board if maximizer's turn or a low value for the board if minimizer's turn.
Is a static number, that in accordance with the characteristics of the game itself, is being assigned to each node (position).
Usually, it maps the set of all possible positions into symmetrical segment $F: P \rightarrow [-M, M]$, M is being assigned only to leaves where the winner is the first player, and value -M to leaves where the winner is the second player.
In zero-sum games, the value of the evaluation function has an opposite meaning - what's better for the first player is worse for the second, and vice versa. Hence, the value for symmetric positions (if players switch roles) should be different only by sign.
A common practice is to modify evaluations of leaves by subtracting the depth of that exact leaf, so that out of all moves that lead to victory the algorithm can pick the one that does it in the smallest number of steps (or picks the move that postpones loss if it is inevitable).

The idea is to find the best possible move for a given node, depth, and evaluation function.

minimax() function consider all the possible ways the game can go and returns the best value for that move, assuming the opponent also plays optimally.

```
function minimax(board, depth, isMaximizingPlayer):
    if current board state is a terminal state:
        return value of the board
    if isMaximizingPlayer:
        bestVal = -INFINITY
```

```
            for each move in board :
                value = minimax(board, depth+1, false)
                bestVal = max(bestVal, value)
            return bestVal
        else :
            bestVal = +INFINITY
            for each move in board :
                value = minimax(board, depth+1, true)
                bestVal = min(bestVal, value)
            return bestVal
```

Alpha-Beta pruning is an optimization technique for minimax algorithm that can drastically improve the time taken to traverse a game tree. It cuts off branches in the game tree which need not be searched because there already exists a better move available. It passes 2 extra parameters in the minimax function, namely alpha and beta.

- **Alpha** is the best value that the maximizer currently can guarantee at that level or above.
- **Beta** is the best value that the minimizer currently can guarantee at that level or above.

```
minimax(0, 0, true, -INFINITY, +INFINITY) #Calling for the first time
function minimax(node, depth, isMaximizingPlayer, alpha, beta):
    if node is a leaf node :
        return value of the node
    if isMaximizingPlayer :
        bestVal = -INFINITY
        for each child node :
            value = minimax(node, depth+1, false, alpha, beta)
            bestVal = max( bestVal, value)
            alpha = max( alpha, bestVal)
            if beta <= alpha:
                break
        return bestVal
    else :
        bestVal = +INFINITY
        for each child node :
            value = minimax(node, depth+1, true, alpha, beta)
            bestVal = min( bestVal, value)
            beta = min( beta, bestVal)
            if beta <= alpha:
                break
        return bestVal
```

Transposition table: store the evaluated values of previous board states, so that if they are encountered again we simply retrieve the stored value from the transposition table.

Zobrist Hashing: hashing function widely used in 2 player board games. It is the most common hashing function used in transposition table.

For a given board state, if there is a piece on a given cell, we use the random number of that piece from the corresponding cell in the table.

If more bits are there in the random number the lesser chance of a hash collision. Therefore 64 bit numbers are commonly used as the standard and it is highly unlikely for a hash collision to occur with such large numbers. The table has to be initialized only once during the programs execution.

Also the reason why Zobrist Hashing is widely used in board games is because when a player makes a move, it is not necessary to recalculate the hash value from scratch. Due to the nature of XOR operation we can simply use few XOR operations to recalculate the hash value.

```
    // A matrix with random numbers initialized once
    Table[#ofBoardCells][#ofPieces]
    // Returns Zobrist hash function for current conf-
```

```
// iguration of board.
function findhash(board):
    hash = 0
    for each cell on the board :
        if cell is not empty :
            piece = board[cell]
            hash ^= table[cell][piece]
    return hash
```

16.8. Computational Geometry

Point
…..

2D Line
$ax + by + c = 0$ where $b = 1$ for non-vertical lines and $b = 0$ for vertical lines.

Given 2 points compute the line:
```
if abs(x1-x2)<EPS: a,b,c=1,0,-x1
else:
  a=-(y1-y2)/(x1-x2)
  b=1
  c=-(a*x1)-y1
```

Check if 2 lines are parallel: ….

Intersection point: …..

..

16.8.1. Convex Hull
Convex hull is the smallest convex polygon that contains all points of a given set.
Andrew's algorithm: $O(n \log n)$
- sort the points primarily according to x coordinates and secondarily according to y coordinates.
- go through the points and add each point to the hull. Always after adding a point to the hull, we make sure that the last line segment in the hull does not turn left. As long as it turns left, we repeatedly remove the second last point from the hull.

```
def convex_hull(points):
    # Sort the points lexicographically (tuples are compared lexicographically).
    # Remove duplicates to detect the case we have just one unique point.
    points = sorted(set(points))

    # Boring case: no points or a single point, possibly repeated multiple times.
    if len(points) <= 1:return points

    # 2D cross product of OA and OB vectors, i.e. z-component of their 3D cross product.
    # Returns a positive value, if OAB makes a counter-clockwise turn,
    # negative for clockwise turn, and zero if the points are collinear.
    def cross(o, a, b):
      return (a[0] - o[0]) * (b[1] - o[1]) - (a[1] - o[1]) * (b[0] - o[0])

    # Build lower hull
    lower = []
    for p in points:
      while len(lower) >= 2 and cross(lower[-2], lower[-1], p) <= 0:
        lower.pop()
      lower.append(p)
```

```python
      # Build upper hull
      upper = []
      for p in reversed(points):
        while len(upper) >= 2 and cross(upper[-2], upper[-1], p) <= 0:
          upper.pop()
        upper.append(p)

      # Concatenation of the lower and upper hulls gives the convex hull.
      # Last point of each list is omitted because it is repeated at the beginning of the other
list.
      return lower[:-1] + upper[:-1]

  # Example: convex hull of a 10-by-10 grid.
  assert convex_hull([(i//10, i%10) for i in range(100)]) == [(0, 0), (9, 0), (9, 9), (0, 9)]
```

16.9. Object-Oriented Design

Require a candidate to sketch out the classes and methods to implement technical problems or real-life objects.

Step 1: Handle Ambiguity

Questions are often intentionally vague in order to test whether you'll make assumptions or if you'll ask clarifying questions.

Who is going to use it? How they are going to use it?

Go through the 6 whys: who, what, where, when, how, why.

Ex. object-oriented design for a coffee maker: might be an industrial machine designed to be used in a massive restaurant servicing hundreds of customers per hour and making ten different kinds of coffee products. Or it might be a very simple machine, designed to be used by the elderly for just simple black coffee.

Step 2: Define the Core Objects

Ex. object-oriented design for a restaurant: Table, Guest, Party, Order, Meal, Employee, Server, and Host.

Step 3: Analyze Relationships

Which objects are members of which other objects? Do any objects inherit from any others? Are relationships many-to-many or one-to-many?

Step 4: Investigate Actions

The key actions that the objects will take and how they relate to each other. You may find that you have forgotten some objects, and you will need to update your design.

Ex. a Party walks into the Restaurant, and a Guest requests a Table from the Host. The Host looks up the Reservation and, if it exists, assigns the Party to a Table. Otherwise, the Party is added to the end of the list. When a Party leaves, the Table is freed and assigned to a new Party in the list.

Design Patterns: Singleton and Factory Method design patterns are widely used in interviews.

Don't fall into a trap of constantly trying to find the "right" design pattern for a particular problem. You should create the design that works for that problem. In some cases, it might be an established pattern, but in many other cases it is not.

Singleton Class: ensures that a class has only one instance and ensures access to the instance through the application. It can be useful in cases where you have a "global" object with exactly one instance. Many people dislike the Singleton design pattern because it can interfere with unit testing.

Factory Method: interface for creating an instance of a class, with its subclasses deciding which class to instantiate. ?????????????????????

Observer: ????

Composite: ???

State: ???

16.10. Testing

What the Interviewer Is Looking For:
- come up with a reasonable list of test cases.
- Big Picture Understanding
- Knowing How the Pieces Fit Together
- Organization: approach the problem in a structured manner
- Practicality: create reasonable testing plans

4 categories of Testing problems:
- Test a real-world object (like a pen);
- Test a piece of software;
- Write test code for a function;
- Troubleshoot an existing issue.

Testing a Real-World Object:
- **Step 1: Who will use it? And why?**
- **Step 2: What are the use cases?** Make a list of the use cases.
- **Step 3: What are the bounds of use?**
- **Step 4: What are the stress/failure conditions?** Discus when it's acceptable (or even necessary) for the product to fail, and what failure should mean.
- **Step 5: How would you perform the testing?** In some cases, it might also be relevant to discuss the details of performing the testing.

Testing a Piece of Software
Aspects**:**
- **Manual vs. Automated Testing:** Some things are simply much better with manual testing because some features are too qualitative for a computer to effectively examine.
- **Black Box Testing vs. White Box Testing:** degree of access we have into the software. In black box testing, we're just given the software as-is and need to test it. With white box testing, we have additional programmatic access to test individual functions. We can also automate some black box testing, although it's certainly much harder.

Steps:
- **Step 1: Are we doing Black Box Testing or White Box Testing?** or both.
- **Step 2: Who will use it? And why?** Software typically has one or more target users, and the features are designed with this in mind.
- **Step 3: What are the use cases?** it's not up to you to just magically decide the use cases. This is a conversation to have with your interviewer.
- **Step 4: What are the bounds of use?** Have the vague use cases defined, we need to figure out what exactly this means
- **Step 5: What are the stress conditions I failure conditions?** What should the failure look like?
- **Step 6: What are the test cases? How would you perform the testing?** Here is where the distinctions between manual and automated testing, and between black box and white box testing, really come into play.

Steps 3 and 4 should have roughly defined the use cases. In step 6, we further define them and discuss how to perform the testing. What exact situations are you testing? Which of these steps can be automated? Which require human intervention?

Remember that while automation allows you to do some very powerful testing, it also has some significant drawbacks. Manual testing should usually be part of your test procedures.

Approach this in a structured manner. Break down your testing into the main components, and go from there. Not only will you give a more complete list of test cases, but you'll also show that you're a structured, methodical person.

Testing a Function: Easiest type of testing. Usually limited to validating input and output. Discuss any assumptions with your interviewer, particularly with respect to how to handle specific situations.

- **Step 1: Define the test cases.**
  - The normal case: Does it generate the correct output for typical inputs?
  - The extremes: Eg. what happens if we pass in an empty array, a very small array, very large, …
  - Nulls and "illegal" input: think about how the code should behave when given illegal input.
  - Strange input: Eg. already sorted array, sorted in reverse order,…
- **Step 2: Define the expected result.** Often is the right output. in some cases, you might want to validate additional aspects. Eg. validate that the original array has not been touched.
- **Step 3: Write test code.** Once you have the test cases and results defined, writing the code to implement the test cases

Troubleshooting Questions
- **Step 1: Understand the Scenario.** ask questions to understand as much about the situation as possible.
  - How long has the user been experiencing this issue?
  - What version of the browser is it? What operating system?
  - Does the issue happen consistently, or how often does it happen? When does it happen?
  - Is there an error report that launches?
- **Step 2: Break Down the Problem.** Break down the problem into testable units. At some point in this process, something fails and it causes the crash.
- **Step 3: Create Specific, Manageable Tests.** Each of the above components should have realistic instructions-things that you can ask the user to do, or things that you can do yourself (such as replicating steps on your own machine). In the real world, you will be dealing with customers, and you can't give them instructions that they can't or won't do.

16.11. Databases

Denormalized vs. Normalized Databases
Normalized databases are designed to minimize redundancy, while denormalized databases are designed to optimize read time.

SQL Statements (Query)
… Riassunto DB

Small Database Design … Riassunto DB
- Step 1: Handle Ambiguity
- Step 2: Define the Core Objects
- Step 3: Analyze Relationships. How do these tables relate to each other? Are they many-to-many? One-to-many?
- Step 4: Investigate Actions

Large Database Design
joins are generally very slow. Thus, you must denormalize your data. Think carefully about how data will be used-you'll probably need to duplicate the data in multiple tables.

16.12. Threads and Locks

Threads in Java
created and controlled by a unique object of the java.lang.Thread class
Implementing the Runnable Interface
Extending the Thread Class
Extending the Thread Class vs Implementing the Runnable Interface:
- extending the Thread class means that the subclass cannot extend any other class.
- A class might only be interested in being runnable, and therefore, inheriting the full overhead of the Thread class would be excessive.

Synchronization and Locks
Threads share the same memory space. issues when two threads modify a resource at the same time.
The keyword synchronized and the lock form the basis for implementing synchronized execution of code.
Synchronized Methods: we restrict access to shared resources through the use of the synchronized keyword. applied to methods and code blocks
Synchronized Blocks: a block of code can be synchronized. only one thread per instance of MyObject can execute the code within the synchronized block.
Locks: is used to synchronize access to a shared resource by associating the resource with the lock.
A thread gets access to a shared resource by first acquiring the lock associated with the resource.
At any given time, at most one thread can hold the lock and, therefore, only one thread can access the shared resource.

Deadlocks and Deadlock Prevention
Situation where a thread is waiting for an object lock that another thread holds, and this second thread is waiting for an object lock that the first thread holds.
In order for a deadlock to occur, you must have all four of the following conditions met:
- Mutual Exclusion: Only one process can access a resource at a given time
- Hold and Wait: Processes already holding a resource can request additional resources, without relinquishing their current resources.
- No Preemption: One process cannot forcibly remove another process' resource.
- Circular Wait: Two or more processes form a circular chain where each process is waiting on another resource in the chain.

Most deadlock prevention algorithms focus on avoiding condition #4: circular wait.

16.13. Operative systems

virtual memory, segmentation, page faults, caching[L1 vs L2 vs L3], memory management algorithms…..

COPY STUFF from the OS course sum up

17. Competitive programming

17.1. Google HashCode

You have to read input from file and write the answer to a file that will be uploaded and simulated.

(20min) While other people are reading the problem, one person write in a file called `io_simulation.py`:
- function to read the file
- decide the solution object format (lists of lists of strings, …)
- function to print the solution

(10min) Then together decide the data structures format that represent the problem instance.

(30min) Then one person writes into a file called `solution.py`:
- data structures that represent the problem
- function that simulates the given solution on a given input

While others think about solutions.

Then together implements variants.

Analyze carefully each input set, some has one constraint very large and another one very small. So you can create an adhoc solution for each input given.
Sometime a solution that solve every input is not good. You should write an algorithm (or a small modification) based also on the input.

17.2. Google Kickstart

Template

```
FAST_IO = 0
if FAST_IO:
    import io, sys, atexit
    rr = iter(sys.stdin.read().splitlines()).ne
    sys.stdout = _OUTPUT_BUFFER = io.BytesIO()

    @atexit.register
    def write():
        sys.__stdout__.write(_OUTPUT_BUFFER.get
else:
    rr = raw_input
rri = lambda: int(rr())
rrm = lambda: map(int, rr().split())
rrmm = lambda n: [rrm() for _ in xrange(n)]


MOD = 10**9 + 7
YES, NO, IMP = "YES", "NO", "IMPOSSIBLE"
from collections import defaultdict as ddic
```

```
def solve(N, A):
    pass


T = rri()
for tc in xrange(1, T + 1):
    N = rri()
    A = rrm()
    ans = solve(N, A)
    print "Case #[]: []".format(tc, ans)
```

...