Coding Interview Preparation

Kaiyu Zheng

January 2017

1 Ground

People say that interviews at Google will cover as much ground as possible. As a new college graduate, the ground that I must capture are the following. Part of the list is borrowed from a reddit post: `https://www.reddit.com/r/cscareerquestions/comments/206ajq/my_onsite_interview_experience_at_google/#bottom-comments`.

1. Data structures
2. Trees and Graph algorithms
3. Dynamic Programming
4. Recursive algorithms
5. Scheduling algorithms (Greedy)
6. Caching

7. Sorting
8. Files
9. Computability
10. Bitwise operators
11. System design

As a anticipated Machine Learning TA, I would think that they might ask me several questions about **machine learning**. I should also prepare something about that. **Operating systems** is another area where you'll get asked, or you may answer questions from that aspect which can potentially impress your interviewer.

Google likes to ask follow-up questions that scale up the size of input. So it may be helpful to prepare some knowledge on Big Data, distributed systems, and databases.

The context of any algorithm problems can involve *sets, arrays, hashtables, strings*, etc. It will be useful to know several well-known algorithms for those contexts, such the KMP algorithm for substrings.

In this document, I will also summarize my past projects, the most difficult bugs, etc., things that might get asked. When I fly to Mountain View, this is the only document I will bring with me. I believe that it is powerful enough.

Contents

2 Knowledge Review

2.1 Data structures

2.1.1 Array

An array is used to describe a collection of elements, where each element is identified by an index that can be computed at run time by the program. I am familiar with this, so no need for more information.

Bit array A bit array is an array where each element is either 0 or 1. People use bit array to leverage parallelism in hardware. Implementation of bit array typically use an array of integers, where all the bits of an integer are used to be elements in the bit array. With such implementation, if we want to retrieve bit with index k in the array, it is the bit with index $k\%32$ in the int with index $k/32$.

An interesting use case of bit array is the *bitmap* in the file system. Each bit in the bitmap maps to a block. To retrieve the address of the block, we just do `BLKSTART`+$k/32$/`BLKSIZE`.

Circular buffer A circular buffer is a single, fixed-size buffer used as if it is connected end-to-end. It is useful as a FIFO buffer, because we do not need to shift every element back when the first-inserted one is consumed. Non-circular buffer is suited as a LIFO buffer.

2.1.2 Tuple

A tuple is a finite ordered list of elements. In mathematics, *n-tuple* is an ordered list of n elements, where n is non-negative integer. A tuple may contain multiple instances of the same element. Two tuples are equal if and only if every element in one tuple equalst to the element at the corresponding index in the other tuple.

2.1.3 Union

In computer science, a *union* is a value that may have any of several representations or formats within the same position in memory; or it is a *data structure* that consists of a variable that may hold such a value. Think about the `union` data type in C, where essentially it allows you to store different data types in the same memory location.

2.1.4 Tagged union

A tagged union, also called a disjoint union, is a data structure used to hold a value that could take on several different, but fixed, types. Only one of the types can be in use at any one time, and a `tag` field explicitly indicates which one is in use. It can be thought of as a type that has several "cases," each of which should be handled correctly when that type is manipulated. Like ordinary unions, tagged unions can save storage by overlapping storage areas for each type, since only one is in use at a time.

Mathematically, tagged unions correspond to disjoint or discriminated unions, usually written using +. Given an element of a disjoint union A + B, it is possible to determine whether it came from A or B. If an element lies in both, there will be two effectively distinct copies of the value in A + B, one from A and one from B.

This data structure is not even covered in my computer science education. I don't expect any problems about it. Good to know.

2.1.5 Dictionary

A dictionary, also called a map or associative array, is a collection of **key, value pairs**, such that each possible key appears at most once in the collection.

There are numerous ways to implement a dictionary. This is basically a review of CSE 332.

Hash table At the bottom level, a hash table is a list T of buckets. We want the size of this list, $|T|$, to be a prime number, to fix sparseness of the list, which can lower the number of collisions.

In order to store a key-value pair into T, we need a *hash function* to map the likely non-integer key to an integer. This hash function should ideally have these properties:

1. Uniform distribution of outputs.
2. Low computational cost.

As more elements are inserted into the hash table, there will likely be collisions. A collision is when two distinct keys map to the same bucket in the list T. Here are several common collision resolution strategies:

1. Separate Chaining: If we hash multiple items to the same bucket, store a `LinkedList` of those items at that bucket. Worst case insert and delete is $O(n)$. Average is $O(1)$. Separate Chaining is easy to understand and implement, but requires a lot more memory allocation.
2. Open Addressing: Choose a different bucket when the natural choice (one computed by hash function) is full. Techniques include linear probing, quadratic probing, and double hashing. The optimal open addressing technique allows (1) duplication of the path we took, (2) coverage of all spaces in the table, (3) avoid putting lots of keys close together. The reasons to use open addressing could be less memory allocation and easier data representation.

I found that open addressing seems to be the preferred way, used by major languages such as Python. So it is worthy to understand how those probing techniques work.

Linear probing: This method is a naive one. It finds the very next free bucket relative to the natural choice bucket. Formula: `(h(key) + i) % |T|`, where `h` is the hash function. When we delete an element

from T, we must use lazy deletion, i.e., mark that element as deleted, but not actually removing it. Otherwise, we won't be able to retrace the insertion path. Linear probing can cause *primary clustering*, which happens when different keys collide to form one big group[1].

Quadratic probing: Similar to linear probing, except that we use a different formula to deal with collision: `(h(key) + i^2) % |T|`. Theory shows that if $\lambda < 1/2$, quadratic probing will find an empty slot in at most $|T|/2$ probes (no failure of insertion)[2]. Quadratic probing causes *secondary clustering*, which happens when different keys hash to the same place and follow the same probing sequence[3].

Double hashing: When there is a collision, simply apply a second, independent hash function g to the key: `(h(key) + i*g(key)) % |T|`. With careful choice of g, we can avoid the infinite loop problem similar to quadratic probing. An example is `h(key) = key % p`, `g(x) = q - (key % p)` for primes p, q with $2 < q < p$.

It is also important to know how *rehashing* works. We need to maintain λ to a reasonable level. Rehashing is done by iterating over old table ($O(n)$), then do n inserts to the new table ($O(n)$). So rehashing is $O(n)$, but the amortized run time is $O(1)$. Amortized analysis considers both the costly and less costly operations together over the *whole* series of operations of the algorithm (so you divide by the big-O of the number of operations performed.)

Variations As we can see, the way keys are inserted into a hash table is dependent on the hash function, which can compute in $O(1)$ time. We don't have to use hashing to store these keys, because this way we will not have any reasonable ordering of the keys in T. So we may use a *binary search tree* to hold the keys if we want keys to be sorted automatically.

In Java, besides HashMap and TreeMap, there is another popular implementation – LinkedHashMap (so there's even LinkedHashSet.)The LinkedHashMap differs from the HashMap in that it maintains a separate doubly-linked list running through all of its entries, which defines the order of keys iteration. The order is normally *insertion-order*. Insertion

[1] See page 6 of `https://courses.cs.washington.edu/courses/cse332/15au/lectures/hashing-2/hashing-2.pdf`.

[2] λ is the load factor, defined as $\lambda = N/|T|$, where N is the number of elements in the hash table.

[3] See page 11 of `https://courses.cs.washington.edu/courses/cse332/15au/lectures/hashing-2/hashing-2.pdf`.

order is not affected if a key is re-inserted into the map. This information is from the Oracle Java Documentation. So essentially, LinkedHashMap has no difference w.r.t. normal hash table at the data-storage level.

Bitmap Same structure as bit array. The only thing is that each bit is mapped to some other stuff of meaning.

2.1.6 Multimap

A multimap is a generalization of a associative array in which more than one value may be associated with and returned for a given key.

2.1.7 Set

A set is a collection of certain values without any particular order, and no repeated values. It is basically a finite set in mathematics. *Set Theory* is useful to understand what you can do with sets. The most basic operations are `union(S, T)`, `intersection(S, T)`, `difference(S, T)`, `subset(S, T)`,

2.1.8 Bag

A bag, also called a multiset, is a set that allows repeated values (duplicates).

2.1.9 Stack

You should be pretty familar with this already. To implement stack, we can use an array, and keep track of the top of the stack.

2.1.10 Queue

You should be pretty familar with this already. To implement queue, we can use an array, and keep track of the front, rear element of the queue. Or we can just remember the front element, and keep a count of elements.

2.1.11 Priority queue

It has similar operations as stack or queue (remove, add, size), but each element has a priority value associated with it. Standard priority queue serves high priority value before one with low priority, and if two elements have equal priority, they are served according to their order in the queue.

ADT Priority Queue ADT should at least support the following functions. Some times we also like to have the `decrease_key(`v, p`)` function.

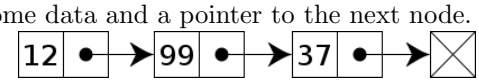
`insert(`v, p`)` `find_min()` `delete_min()`

Implementation Common implementation of priority queue uses a *heap*. Refer to the section for heap for more details.

We can also use *doubly linked list*.

2.1.12 List

Linked list A linked list consists of a group of nodes that together represent a sequence (ordered list). In the basic form, each node stores some data and a pointer to the next node.



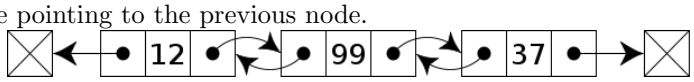
It is important to understand the trade-offs of linked list:

1. Indexing: $\Theta(n)$
2. Insert/delete at both ends: $\Theta(1)$.
3. Insert/delete in middle: search time + $\Theta(1)$. (No need to shift elements)

Linked list has wasted space of $\Theta(n)$, because of the extra storage of the references.

The advantage of singly linked list over others is that for some operations, such as merging two lists, or enumerating elements in reverse order, have very simple recursive algorithms, compared to iterative ones. For other lists, these algorithm have to include extra arguments and base cases. Additionally, linear singly linked lists allow *tail-sharing*, which is using a common terminal sublist for two different lists. This is not okay for doubly linked list or circular linked list, because a node cannot belong to more than one list in those cases.

Doubly linked list A doubly linked list differs from singly linked list in that each node has two references, one pointing to the next node, and one pointing to the previous node.



The convenience of doubly linked list is that it allows traversal of the list in either direction. In operating systems, doubly linked lists are used to maintain active processes, threads, etc.

There is a classic problem: Convert a given binary tree to doubly linked list (or the other way around). This problem will be discussed later.

Unrolled linked list An unrolled linked list differs from linked list in that each node stores multiple elements. It is useful for increasing cache performance while decreasing the memory overhead assoicated with storing list metadata (e.g. references). Related to the B-tree. A typical node looks like this:

```
record node {
    node next
    int numElements // number of elements in this node, up to
        some max limit
    array elements
}
```

XOR linked list An XOR linked list takes advantage of the bitwise XOR operation, to decrease storage requirements for doubly linked lists. An ordinary doubly linked list requires two references in a node, one for the previous node, one for the next node. An XOR linked list uses one address field to compress the two references, by storing the bitwise XOR between the address of the previous node and the address of the next node.

Example: We have XOR linked list nodes A, B, C, D, in order. So for node B, it has a field A$\oplus$C; for node C it has a field B$\oplus$D, etc. When we traverse the list, if we are at C, we can obtain the address of D by XORing the address of B with the reference field of C, i.e. B$\oplus$(B$\oplus$D)=(B$\oplus$B)$\oplus$D=0$\oplus$D=D.

For the traversal to work, we can store B's address alone in A's field, and store C's address alone in D's field, and we have to mark A as start and D as end. This is because given an arbitrary middle node in XOR linked list, one cannot tell the next or previous addresses of that node.

Advantages: Obviously it saves a lot of space.

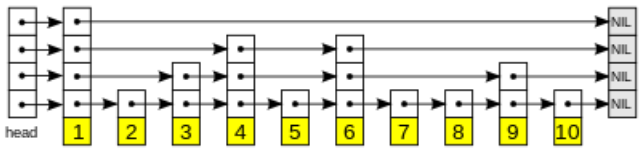
Disadvantages: General-purpose debugging tools cannot follow XOR chain. Most garbage collection schemes do not work with data structures that do not contain literal pointers. Besides, while traversing the list, you have to remember the address of the previous node in order to figure out the next one. Also, XOR linked list does not have all features of doubly linked list, e.g. the ability to delete a node knowing only its address.

Self-organizing list From Wikipedia:A self-organizing list is a list that reorders its elements based on some self-organizing heuristic to improve average access time. The aim of a self-organizing list is to improve efficiency of linear search by moving more frequently accessed items towards the head of the list. A self-organizing list achieves near constant time for element access in the best case. A self-organizing list uses a reorganizing algorithm to adapt to various query distributions at runtime. Self-organizing list can be used in compilers (even for code on embedded systems) to maintain symbol tables during compilation of program source code. Some techniques for rearranging nodes:

Move to Front (MTF) Method: Moves the accessed element to the front of the list. Pros: easy to implement, no extra memory. Cons: may prioritize infrequently used nodes.

Count Method: Keep a count of the number of times each node is accessed. Then, nodes are rearranged according to decreasing count. Pros: realistic in representing the actual access pattern. Cons: extra memory; unable to quickly adapt to rapid changes in access pattern.

Skip list A skip list is a probabilistic data structure that allows fast search within an ordered sequence of elements. Fast search is made possible by maintaining a linked hierarchy of subsequences, where each subsequence skips over fewer elements than the previous one.



A skip list is built in layers. The bottom layer is the ordinary linked list. Each layer higher is an "express lane" for the lists below, where an element in layer i appears in layer $i+1$ with some *fixed probability* p. This seems fancy. How are these express lanes used in searching? How are skip lists used?

A search for a target starts at the head element of the top layer list, and it proceeds horizontally until the current element is *greater than or equal* to the target. If equal, target is found. If greater, the search returns to the previous element, and drops down vertically to the list at the lower layer. The expected run time is $O(logn)$. Skip lists can be used to maintain some, e.g. key-value, structure in databases.

People compare skip lists with balanced trees. Skip lists have the same asymptotic expected time bounds as balanced trees, and they are

simpler to implement, and use less space. The average time for search, insert and delete are all $O(logn)$. Worst case $O(n)$.

2.1.13 Heap

Minimum-heap property: All children are larger.

Binary heap One more property of binary heap is that the tree has no gaps. Implementation using an array: `parent(n) = (n-1) / 2`; `leftChild(n) = 2n + 1`; `rightChild(n) = 2n + 2`. Floyd's build heap algorithm takes $O(n)$. There are several variations of binary heap. Insert, deleteMin, decreaseKey operations take $\Theta(logn)$ time. Merge takes $\Theta(n)$ time.

Fibonacci heap This kind of heap is not a single binary-tree shaped structure for (conventional) binary heaps. Instead, it is a collection of trees satisfying the minimum-heap property. This implies that the minimum key is always at the root of one of the trees. The trees structures can be more flexible - they can have gaps. Insert, decrease-key, and merge all have amortized constant run time. Delete-min is $O(logn)$. Implementation is kind of complex. Visualization: `https://www.cs.usfca.edu/~galles/visualization/FibonacciHeap.html`

2.1.14 Graph

A graph consists of a set of vertices and a set of pairs of these vertices as edges. If these pairs are unordered, then the graph is an undirected graph. If these pairs are ordered pairs, then the graph is a directed graph.

Paths A path is called *simple* if all its vertices are distinct from one another. A *cycle* is a path $\{v_1, v_2, \cdots, v_{k-1}, v_k\}$ in which for $k > 2$, the first $k-1$ nodes are distinct, and $v_1 = v_k$. The *distance* between two nodes u and v is the minimum number of edges in a u-v path.

Connectivity In an undirected graph, the graph is *connected* if. for every pair of nodes u and v, there is a path from u to v.

In an directed graph, the graph is *strongly connected* if, for every two nodes u and v, there is a path from u to v and a path from v to u.

ADT The following is the typical operations for a graph abstract data type (ADT). During the interview, if you need to use a graph library, you can expect it to have these functions. In Python, there are graph libraries such as `python-graph`.

`add_vertex`(G, v)	`add_edge`(G, u, v)	`neighbors`(G, v)
`remove_vertex`(G, v)	`remove_edge`(G, u, v)	`adjacent`(G, v, w)

Common representations of a graph are *adjacency list* or *adjacency matrix*. Wikipedia has a nice explanation and comparison of different representations of a graph ADT. Check it out below.

Adjacency list: Vertices are stored as records or objects, and every vertex stores a list of adjacent vertices. This data structure allows the storage of additional data on the vertices. Additional data can be stored if edges are also stored as objects, in which case each vertex stores its incident edges and each edge stores its incident vertices.

Adjacency matrix: A two-dimensional matrix, in which the rows represent source vertices and columns represent destination vertices. Data on edges and vertices must be stored externally. Only the cost for one edge can be stored between each pair of vertices.

	Adjacency list	Adjacency matrix
Store graph	$O(\|V\| + \|E\|)$	$O(\|V\|^2)$
Add vertex	$O(1)$	$O(\|V\|^2)$
Add edge	$O(1)$	$O(1)$
Remove vertex	$O(\|E\|)$	$O(\|V\|^2)$
Remove edge	$O(\|E\|)$	$O(1)$
`adjacent`(G, v, w)	$O(\|V\|)$	$O(1)$

2.1.15 Tree

An undirected grah is a *tree* if it is connected and does not contain a cycle (acyclic). *Descendant & ancestor:* We say that w is a descendent of v if v lies on the path from root to w. In this case, v is an ancestor of w. There are so many different kinds of trees. Won't discuss them here.

Trie A trie is also called a prefix tree. Each edge in a trie represents a character, and the value in each node represents the current prefix by collecting all characters on the edges when traversing from the root (an empty string) to that node. All the descendants of a node have the same prefix as that node. See 5.1 for my Java implementation of Trie.

A *compressed trie* is a trie where non-branching paths are compressed into a single edge. See the figure in 2.8.3 as an example.

B-Tree In computer science, a B-tree is a self-balancing tree data structure that keeps data sorted and allows searches, sequential access, insertions, and deletions in logarithmic time. The B-tree is a generalization of a binary search tree in that a node can have more than two children (Comer 1979, p. 123). For an interview, I doubt that we need to know implementation details for a B-Tree. Know its motivation through.

Motivation: Self-balanced binary search tree (e.g. AVL tree) is slow when the height of the tree reaches a certain limit such that manipulating nodes require disk access. In fact, for a large dictionary, most of the data is stored on disk. So we want a self-balancing tree that is even shallower than AVL tree, to minimize the number of disk accesses, and exploit disk block size.

Binary Indexed Tree A binary indexed tree, also called a *Fenwick tree*, is a data structure that can efficiently update elements and calculate prefix sums in a table of numbers. I used it for the 2D Range sum problem (3.8). See my implementation of 2D binary indexed tree there.

Motivation: Suppose we have an array *arr* with length n, we want to (1) find the sum of first k elements, and (2) update the value of the element *arr[i]*, both in $O(logn)$ time.

How it works[4]: The core idea behind BIT is that every integer can be written as the sum of power 2's. Each node in a BIT stores the sum of a range $[i, j]$, and with all nodes combined, the BIT will cover the entire range of the array. There needs to be a dummy root node, so the size of BIT is $n + 1$. Here is how we **build up** the BIT for this array.

First, we initialize an array `BIT` with size $n + 1$ and all elements set to 0. Then, we iterate through `arr`. For element i, we do an update for the `BIT` array as follows:

1. We look at the node `BIT[i+1]`. We add `arr[i]` to it so

   `BIT[i+1]=BIT[i+1]+arr[i].`

2. Now since we changed the range sum of a node, we have to update the value of some other nodes. We can obtain the index `n` of the *next node* with respect to node `j` for updating using the following formula:

   `n=j+j&(-j).`

[4]Watch YouTube video for decent explanation, by Tushar Roy: `https://www.youtube.com/watch?v=CWDQJGaN1gY&t=13s`

We add the value `arr[i]` to each of the affected node, until the computed index `n` is out of bound. The run time here is $O(logn)$.

Here is how we use a BIT to compute the **prefix sum** of the first k elements. Just like before, we find the `BIT` node with index `k+1`. We add the value of that node to our `sum`. Then, traverse from the node `BIT[i+1]` back to the root. Each time we go to the parent node of current node, suppose `BIT[j]`, we compute the index of that parent node `p`, by

```
p=j-j&(-j)
```

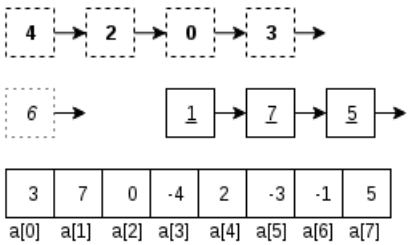
Then we add the value of the parent node to `sum`, until we reach the root. Return `sum` as the result. This process is also $O(logn)$ time. The space of BIT is $O(n)$.

2.1.16 Union-Find

A union-find data structure, also called a disjoin-set data structure, is a data structure that maintains a set of disjoint subsets (e.g. components of a graph). It supports two operations:

1. *Find(u):* Given element u, it will return the name of the set that u is in. This can be used for checking if u and v are in the same set. Optimal run time: $O(logn)$.
2. *Union(N_1, N_2):* Given disjoint sets N_1, N_2, this operation will merge the two components into one set. Optimal run time $O(1)$ if we use pointers; if not, it is $O(logn)$.

First, we will discuss an implementation using implicit list. Assume that all objects can be labeled by numbers $1, 2, 3, \cdots$. Suppose we have three disjoint sets as shown in the upper part of the following image. Notice that the above representation is called an *explicit list*, because it explicitly connects objects within a set together.



As shown in the lower part of the image above, we can use a single *implicit list* to represent the disjoint sets that can remember (1) pointers to the canonical element (i.e. name) for a disjoint set, (2) the size of each disjoint set. See appendix 5.3 for my Python implementation of Union-Find, using implicit list.

When we `union` two sets, it is conceptually like joining two trees together, and the root of the tree is the canonical element of the set after union. *Path compression* is basically the idea that we always join the tree with smaller height into the one with greater height, i.e. the root of the taller tree will be the root of the new tree after union. In my implementation, I used *union-by-size* instead of height, which can produce the same result in run time analysis[5]. The run time of `union` is determined by the run time of `find` in this implementation. Analysis shows that the upper bound of run time of `find` is the *inverse Ackermann function*, which is even better than $O(logn)$.

There is another implementation that uses tree that is also optimal for `union`. In this case, the union-find data structure is a collection of trees (forest), where each tree is a subset. The root of the tree is the canonical element (i.e. name) of the disjoint set. It is essentially the same idea as implicit list.

2.2 Trees and Graph algorithms

2.2.1 BFS and DFS

Pseudo-code BFS and DFS needs no introduction. Here is the pseudo-code. The only difference here between BFS and DFS is that, for BFS, we use *queue* as `worklist`, and for DFS, we use *stack* as `worklist`.

```
BFS/DFS(G=(V,E), s) {
    worklist = [s]
    seen = {s}
    while worklist is not empty:
        node = worklist.remove
        {visit node}
        for each neighbor u of node:
            if u is not in visited:
                queue.add(u)
                seen.add(u)
}
```

[5]Discussed in CSE 332 slides, by Adam Blank: `https://courses.cs.washington.edu/courses/cse332/15au/lectures/union-find/union-find.pdf`.

There is another way to implement DFS using recursion (From MIT 6.006 lecture):

```
DFS(V, Adj):
    parent = {}
    for s in V:
        if s is not in parent:
            parent[s] = None
            DFS-Visit(Adj, s, parent)

DFS-Visit(Adj, s, parent):
    for v in Adj[s]:
        if v is not in parent:
            parent[v] = s
            DFS-Visit(Adj, v, parent)
```

Obtain BFS/DFS layers BFS/DFS results in BFS/DFS-tree, which has layers $L_1, L_2, \cdots$. Each layer is a set of vertices. BFS layers are really useful for problems such as determining if the graph is two-colorable. DFS layers are useful too (application?)

```
BFS/DFS(G=(V,E), s) {
    worklist = [s]
    seen = {s}
    layers = {s:0}
    while worklist is not empty:
        node = worklist.remove
        {visit node}
        for each neighbor u of node:
            if u is not in visited:
                queue.add(u)
                seen.add(u)
                layers.put(u, layers[node]+1)
    Go through keys in layers and obtain the set of nodes for
        each layer.
}
```

Now we will look at some BFS/DFS Tree Theorems.

Theorem 2.1 (BFS)**.** *For each $j \geq 1$, layer L_j produced by BFS starting from node s consists of all nodes at distance exactly j from s.*

Theorem 2.2 (BFS)**.** *There is a path from s to t if and only if t appears*

in some BFS layer.

Theorem 2.3 (BFS)**.** *For BFS tree T, if there is an edge (x, y) in G such that node x belongs to layer L_i, and node y belongs to layer L_j, then i and j differ at most 1.*

Theorem 2.4 (DFS)**.** *Let T be a DFS tree. Let x, y be nodes in T. Let (x, y) be an edge of G that is NOT an edge of T. Then, one of x or y is an ancestor of the other.*

Theorem 2.5 (DFS)**.** *For a given recursive call DFS(u), all nodes that are marked visited (e.g. put into the `parent` map) between the invocation and end of this recursive call are descendents of u in the DFS tree T.*

Now, let us look at how BFS layers is used for the two-colorable graph problem. A graph is two-colorable if and only if it is *bipartite*. A *bipartite graph* is a graph whose vertices can be divided into disjoint sets U and V (i.e. U and V are independent sets), such that *every edge* connects a vertex in U to one in V.

Theorem 2.6 (No Odd Cycle)**.** *If a graph G is bipartite, then it cannot contain an odd cycle, i.e. a cycle with odd number of edges (or nodes).*

Theorem 2.7 (BFS and Bipartite)**.** *Let G be a connected graph. Let $L_1, L_2, \cdots$ be the layers of teh BFS tree starting at node s. Then,*

1. *Either there exists an edge that joins two nodes of the same layer, which implies that there exists odd cycle in G, so G isn't bipartite.*
2. *Or, there is no edge that joins two nodes of the same layer. So G is bipartite.*

Edge classification for DFS tree We can classify edges in G after DFS into four categories:

1. *tree edge:* We visit *new* vertex via such edge in DFS.
2. *forward edge:* edge from node to its descendant in DFS tree.
3. *backward edge:* edge from node to its ancestor in DFS tree.
4. *cross edges:* edges between two non-ancestor related subtrees.

Note that DFS tree for a directed graph can have all four types of edges, but DFS tree for an undirected graph cannot have forward edges and cross edges.

Cycle detection

Theorem 2.8. *A graph G has a cycle if and only if the DFS has a backward edge.*

Besides using this theorem, Kahn's algorithm (discussed in Topological Sort section) can be used to detect if there is a cycle.

2.2.2 Topological Sort

For a directed graph G, we say that a *topological ordering* of G is an ordering of its nodes as $v_1, v_2, \cdots, v_n$, such that for every edge (v_i, v_j), we have $i < j$. (Edges point forward in the ordering.)

Theorem 2.9. *G has a topological ordering if and only if G is a DAG (directed acyclic graph).*

Theorem 2.10. *In every DAG G, there is a node v with no incoming edges.*

Here we discuss Kahn's algorithm for topological sort; run time is $O(|V|+|E|)$. First, start with a list of nodes that have no incoming edges, and insert them into a set S. Then, we remove all nodes from G in S. While removing a node, we remove the outgoing edges of that node. We will repeat these two steps until S is empty. The order that we remove nodes from G is the topological order.

Kahn's algorithm can be used to test if a graph is a DAG. After S is empty, if the graph is a DAG, all edges should have been removed from G. If there is still edges, then G is not a DAG.

Another, more gentle (no removal of nodes and edges) algorithm uses DFS. The idea behind is that when we do DFS, we *start visiting* a node before its descendants in the DFS tree, and *finish visiting* those descendants before we actually finish visiting the node itself.

The topological sort thus can be obtained by doing DFS, and output the reverse of the finishing times of verticies[6]. So if node v is finished visiting after node u, then v is topologically sorted before u; v must be the ancestor of u in the tree. We must check if there is a back edge in the graph, before we output the final topological order, because if so, the graph has a cycle.

[6] From MIT 0.006 class, Fall 2011

2.2.3 Paths

Obtain BFS/DFS path from s to t When we have an unweighted graph, we can find a path from s to t simply by BFS or DFS. BFS gives the shortest path in this case. It is straightforward if we have a map that can tell the parent of each node in the tree.

```
BFS/DFS(G=(V,E), s, t):
    worklist = [s]
    seen = {s}
    parent = {s:None}
    while worklist is not empty:
        node = worklist.remove
        if node == t:
            return Get-Path(s, t, parent)
        for each neighbor u of node:
            if u is not in visited:
                queue.add(u)
                seen.add(u)
                parent[u] = node
    return NO PATH

Get-Path(s, t, parent):
    v = t
    path = {t}
    while v != s:
        v = parent[v]
        path.add(v)
    return path
```

Dijkstra's Algorithm This algorithm is for finding the shortest path from s to t on a graph with nonnegative weights. When choosing the node v, if we use Fibonacci Heap to store the edges, then the run time can be $O(|E| + |V|log|V|)$.

```
Dijkstra's-Algorithm(G=(V,E), s, t):
    S = {}; d[s] = 0; d[v] = infinity for v != s
    prev[s] = None
    While S != V
        Choose v in V-S with minimum d[v]
        Add v to S
        For each w in the neighborhood of v
            if d[v] + c(v,w) < d[w]:
```

```
            d[w] = d[v] + c(v,w)
            prev[w] = v
    return d, prev
```

This algorithm works by expanding a set S starting from s, within which we have nodes such that the shortest paths from s to these nodes are known. The step that chooses `v` from the set difference between V and S with minumum `d[v]` is equivalent as choosing the edge with minimum weight that goes from S to $V - S$.

Bellman-Ford Algorithm This algorithm works for graph that has negative edge weights, but no negative cycles. Its run time is $O(|V||E|)$.

```
Bellman-Ford(G=(V,E), s, t):
    d[s] = 0; d[v] = infinity for v != s
    prev[s] = None
    for i = 1 to |V|-1:
        for each edge (u, v) with weight w in E:
            if d[u] + w < d[v]:
                d[v] = d[u] + w
                prev[v] = u
    return d, prev
```

A* Algorithm Proposed by P.Hart et. al., this algorithm is an extension of Dijkstra's algorithm. It achieves better performance by using *heuristics* to guide its search. A* algorithm finds the path that minimizes

$$f(n) = g(n) + h(n)$$

where n is the last node on the path, $g(n)$ is the cost of the path from the start node to n, and $h(n)$ is a heuristic that estimates the cost of the cheapest path from n to the goal. The heuristic must be *admissible*, i.e. it never overestimates the actual cost to get to the goal. Below is my implementation with Python, when I took the CSE 473 (Artificial Intelligence) class:

```
def aStarSearch(problem, heuristic=nullHeuristic):
    """Search the node that has the lowest combined cost and
        heuristic first."""
    startState = problem.getStartState()
    visitedStates = set({})
    worklist = util.PriorityQueue()
```

```
# First tuple means (state, path_to_state, cost_of_path)
worklist.push((startState, [], 0),
              0 + heuristic(startState, problem))
while not worklist.isEmpty():
    state, actions, pathCost = worklist.pop()
    if state in visitedStates:
        continue
    if problem.isGoalState(state):
        return actions

    successors = problem.getSuccessors(state)
    for stepInfo in successors:
        sucState, action, stepCost = stepInfo
        sucPathCost = pathCost + stepCost
        worklist.push((sucState, actions + [action],
            sucPathCost),
                    sucPathCost + heuristic(sucState,
                        problem))

    # mark the current state as visited
    visitedStates.add(state)

# Not able to get there
return None
```

All-pairs shortest paths This problem concerns finding all paths between every pair of vertices. A well-known algorithm for this is *the Floyd-Warshall's algorithm*, which runs in $O(|V|^3)$ time. It works for graphs with positive or negative weights, with no negative cycles. Its run time is impressive, considering the fact that there may be up to $\Omega(|V|^2)$ edges in a graph. This is a dynamic programming algorithm.

The algorithm considers a function `shortestPath(i, j, k)` which finds the shortest path from `i` to `j` with only vertices $\{v_1, v_2, \cdots, v_k\} \subset V$. Given that for every pair of nodes i and j, we know the output of `shortestPath(i, j, k)`, our goal is to figure out the output of `shortestPath(i, j, k+1)` for every such pair. When we have a new vertex, v_{k+1}, either the path from v_i to v_j goes through v_{k+1}, or not. This brings us to the core of Floyd-Warshall's algorithm:

`shortestPath(i, j, k+1)` = min(`shortestPath(i, j, k)`, `shortestPath(i, k+1, k)` + `shortestPath(k+1, j, k)`)

Hamiltonian path In the mathematical field of graph theory, a Hamiltonian path (or traceable path) is a path in an undirected or directed graph that visits each vertex exactly once. A Hamiltonian cycle (or Hamiltonian circuit) is a Hamiltonian path that is a cycle. Determining whether such paths and cycles exist in graphs is the Hamiltonian path problem, which is NP-complete.

Dynamic Programming algorithm for finding a Hamiltonian path: Bellman, Held, and Karp proposed an algorithm to find Hamiltonian path in time $O(n^2 2^n)$. In this method, one determines, for each set S of vertices and each vertex v in S, whether there is a path that covers exactly the vertices in S and ends at v. For each choice of S and$v$, a path exists for (S, v) if and only if v has a neighbor w such that a path exists for $(S - v, w)$, which can be looked up from already-computed information in the dynamic program[7].

2.2.4 Minimum Spanning Tree

Kruskal's Algorithm Builds a spanning tree by successively inserting edges from E in order of increasing cost, using a union-find data structure. Run time: $O(|E|log|E|)$.

Prim's Algorithm Start with a root node s and try to greedily grow a tree from s outward. At each step, we simply add the node that can be attached as cheaply as possible to the partial tree we already have. If we use adjacency list to represent the graph, and use Fibonacci heap to retrieve minimum cost edge, then the run time is $O(|E| + |V|log|V|)$.

Reverse-Delete Algorithm We start with the full graph (V, E) and begin deleting edges in order of decreasing cost. As we get to each e, we would not delete it if doing so disconnects the graph we currently have. To check if the graph is connected, we can do BFS or DFS, and see if the reachable component has the same size as $|V|$. The run time could be $O(|E|log|V|(loglog|V|)^3)$, if we use Thorup's algorithm to check connectivity of a graph, which has run time $O(log|V|(loglog|V|)^3)$[8].

[7] These two paragraphs are from Wikipedia.

[8] `http://www.cs.princeton.edu/courses/archive/spr10/cos423/handouts/NearOpt.pdf`

2.2.5 Max-flow Min-cut

Given a directed weighted graph, with one source node s and one sink node t, we can find a partition of the nodes A, B such that the sum of the cost of edges that go from component A to B is minimum, compared to all other possible partition (*Min-cut*). This sum equals to the maximum *flow value* from the s to t[9].

To find Max-flow, we can use the *Ford-Fulkerson Algorithm*. Pseudo-code is as follows.

```
Ford-Fulkerson Max-Flow(G=(V, E), s, t):
    Initially f(e) = 0 for all e in E
    While there is an s-t path in the residual graph G_f:
        Let P be a simple s-t path in G_f
        f' = augment(f, P)
        update f to be f'
        update the residual graph G_f to be G_f'.
    return f

augment(f, P):
    b = bottleneck(f, P)
    for e = (u, v) in P:
        if e is forward edge, then
            increase f(e) in G by b
        else e is a backward edge
            decrease f(e) in G by b
    return f
```

2.3 Dynamic Programming

What kinds of problems can be solved using Dynamic Programming? One property these problems have is that if the optimal solution involves solving a subproblem, then it uses the optimal solution to that subproblem. For instance, say we want to find the shortest path from A to B in a graph, and say this shortest path goes through C. Then it must be using the shortest path from C to B. Or, in the knapsack example, if the optimal solution does not use item n, then it is the optimal solution for the problem in which item n does not exist. The other key property is that there should be only a polynomial number of different subproblems. These two properties together allow us to build the optimal solution to

[9] Refer to resources for Network Flow for what this means.

the final problem from optimal solutions to subproblems[10].

2.3.1 One example problem involving 2-D table

Problem: Given a string x consisting of 0s and 1s, we write x^k to denote k copies of x concatenated together. We say that a string x' is a *repetition* of x if it is a *prefix* of x^k for some number k. So $x' = 10110110110$ is a repetition of $x = 101$.

We say that a string s is an *interleaving* of x and y if its symbols can be partitioned into two (not necessarily contiguous) subsequences s' and s'', so that s' is a repetition of x, and s'' is a repitition of y. For example, if $x = 101$, and $y = 00$, then $s = 100010101$ is an interleaving of x and y, since characters 1,2,5,7,8,9 form 101101 – a repetition of x – and the remaining characters 3,4,6 form 000, a repitition of y.

Give an efficient algorithm that takes strings s, x, y, and decide if s is an interleaving of x and y[11].

My Solution Our Opt table will look like this.(Consider / as an empty character, which means including / in a substring is as if we included nothing). Assume that the length of string s is l.

	/	x'_1	$\cdots$	x'_l
/	True			
y'_1				
$\cdots$				
y'_l				

The value of each cell is either True or False. Strictly,

Opt[x_i, y_j] = True **if and only if**

Opt[x_i, y_{j-1}] = True AND $s[i+j] = y'[j]$ OR,

Opt[x_{i-1}, y_j] = True AND $s[i+j] = x'[i]$.

(If $x'[i] =$ / or $y'[j] =$ /, then we treat $s[i+j] = x'[i$ and $s[i+j] = y'[j]$ to be True.

We can think of a cell being True as there is an interleaving of i characters of x' and j characters of y' that makes up the first $i + j$ characters of string s.

If we filled out this table, we can return True if for some i and j such that i is a multiple of $|x|$ AND j is a multiple of $|j|$ AND $i + j = l$ AND

[10]Cited from CMU class lecture note: `https://www.cs.cmu.edu/~avrim/451f09/lectures/lect1001.pdf`

[11]This problem is from *Algorithm Design*, by Kleinberg, Tardos, pp.329 19.

Opt[i, j] = True. This is precisely saying that we return True if s can be composed by interleaving some repetition of x and y.

So first, our job now is to fill up this table. We can traverse j from 1 to l (traverse each row). And inside each iteration, we traverse i from 1 to l. Inside each iteration of this nested loop, we set the value of Opt[i, j] according to the rule described above.

Then, we can come up with i and j by fixing i and increment j by $|y|$ until $i + j > l$ (*). Then, we increment i by $|x|$. Then, we repeat the previous step (*), and stop when $i + j > l$. We check if $i + j = l$ inside each iteration, and if so, we check if Opt[i,j]=True. If yes, we return True. If we don't return True, we return False at the end.

2.3.2 Well-known problems solved by DP

Longest Common Subsequence Find the longest subsequence common to all sequences in a set S of sequences. Unlike substrings, subsequences are not required to occupy consecutive positions within the original sequences.

Let us look at the case where there are only two sequences x, y in S. For example, x is 14426, and y is 2134. The longest common subsequence of x and y is then 14. **Define** Opt[i, j] to be the longest common subsequence for subsring $x[0 : i]$ and $y[0 : j]$. We have the following update formula:

$$\text{Opt} = \begin{cases} \emptyset & \text{if } i = 0, j = 0 \\ \text{Opt}[i-1, j-1] \cup x_i & \text{if } x[i] = y[j] \\ \max(\text{Opt}[i-1, j], \text{Opt}[i, j-1]) & \text{if } x[i] \neq y[j] \end{cases}$$

Similar problems: longest common substring, longest increasing subsequence).

Now, let us discuss the **Levenshtein distance problem**. The *Levenshtein distance* measures the difference between two sequences, i.e. the fewest number of operations (edit, delete, add) to change from one sequence to another. The definition of Levenshtein distance is itself a dynamic programming solution to find the edit distance, as follows (from Wikipedia):

Definition 2.1. The Levenshtein distance between two strings a, b (of length $|a|$ and $|b|$ respectively) is given by $\text{lev}_{a,b}(|a|, |b|)$ where

$$\text{lev}_{a,b}(i,j) = \begin{cases} \max(i,j) & \text{if} \min(i,j) = 0, \\ \min \begin{cases} \text{lev}_{a,b}(i-1,j) + 1 \\ \text{lev}_{a,b}(i,j-1) + 1 \\ \text{lev}_{a,b}(i-1,j-1) + 1_{(a_i \neq b_j)} \end{cases} & \text{otherwise.} \end{cases}$$

where $1_{(a_i \neq b_j)}$ is the indicator function that equals to 1 if $a_i \neq b_j$, and $\text{lev}_{a,b}(i,j)$ is the distance between the first i characters of a and the first j characters of b.

The Knapsack Problem Given a set of items S, with size n, each item a_i with a weight w_i and a value v_i, determine the number of each item to include in a collection so that the total weight is less than or equal to a given limit K and the total value is as large as possible.

Define $\text{Opt}[i,k]$ to be the optimal subset of items from $a_1, \cdots, a_i$ such that the total weight does not exceed k. Our final result will then be given by $\text{Opt}[n,K]$. For an item a_i, either it is included into the subset, or not. If not, that means the total weight of the subset with a_i added exceeds k. Therefore we have:

$$\text{Opt}[i,k] = \begin{cases} \emptyset & \text{if k=0,} \\ \text{Opt}[i-1,k] & \text{if adding } w_i \text{ exceeds } k, \\ \max(\text{Opt}[i-1,k], & \\ \quad \text{Opt}[i-1,k-w_i]+v_i) & \text{otherwise} \end{cases}$$

Similar problems: subset sum.

Matrix Chain Multiplication Given a sequence of matrices $A_1 A_2 \cdots A_n$, the goal is to find the most efficient way to multiply these matrices. The problem is not actually to perform the multiplications, but merely to decide the sequence of the matrix multiplications involved.

Here are many options because matrix multiplication is associative. In other words, no matter how the product is parenthesized, the result obtained will remain the same. For example, for four matrices A, B, C, and D, we would have:

$$((AB)C)D = ((A(BC))D) = (AB)(CD) = A((BC)D) = A(B(CD))$$

However, the order in which the product is parenthesized affects the number of simple arithmetic operations needed to compute the product, or the efficiency. For example, if A is a 10×30 matrix, B is a 30×5 matrix, and C is a 5×60 matrix, then computing $(AB)C$ needs

$$(10 \times 30 \times 5) + (10 \times 5 \times 60) = 1500 + 3000 = 4500$$

operations, *while* computing $A(BC)$ needs

$$(30 \times 5 \times 60) + (10 \times 30 \times 60) = 9000 + 18000 = 27000$$

operations. The first parenthesization is obviously more preferable. Given n matrices, the total number of ways to parenthesize them is $P(n) = \Omega(4^n/n^{3/2})$, so brute force is impractical[12].

We use dynamic programming. First, we characterize the structure of an optimal solution. We claim that one possible structure is the following:

$$((A_{1:i})(A_{i+1:n})) \tag{1}$$

where $A_{i:j}$ means matrix multiplication of $A_i A_{i+1} \cdots A_j$. In order for the above to be optimal, the parenthesization for $A_{1:i}$ and $A_{i+1:n}$ must also be optimal. Therefore, we can recursively break down the problem, till we only have one matrix. A subproblem is of the form $A_{i:j}$, with $1 \leq i, j \leq n$, which means there are $O(n^2)$ unique subproblems (counting).

Let $\text{Opt}[i, j]$ be the cost of computing $A_{i:j}$. If the final multiplication of $A_{i:j}$ is $A_{i:j} = A_{i:k} A_{k+1,j}$, assuming that $A_{i:k}$ is $p_{i-1} \times p_k$, and $A_{k+1:j}$ is $p_k \times p_j$, then for $i < j$,

$$\text{Opt}[i, j] = \text{Opt}[i, k] + \text{Opt}[k + 1, j] + p_{i-1} p_k p_j$$

This is because by definition of Opt, we need $\text{Opt}[i, k]$ to compute $A_{i:k}$, and $\text{Opt}[k + 1 : j]$ to compute $A_{k+1:j}$, and $p_{i-1} p_k p_j$ to compute the multiplication of $A_{1:k}$ and $A_{k+1:j}$. For $i = j$, $\text{Opt}[i, j] = 0$. Since we need to check all values of i, j pair, i.e. the parenthesization shown in (1), the run time is $O(n^3)$.

2.3.3 Top-down dynamic programming

So far, we are able to come up with an equation for the Opt table/array in the example problems above. This is called bottom-up approach. However, for some problems, it is not easy to determine such equation. In this case, we can use *memoization* and top-down approach, usually involving recursion. The top-down approach basically leads to the same algorithm as bottom-up, but the perspective is different. According to a lecture note of CMU algorithms class:

> **Basic Idea:** Suppose you have a recursive algorithm for some problem that gives you a really bad recurrence like $T(n) = 2T(n - 1) + n$. However, suppose that many of the subproblems you reach as you go down the recursion tree are the same. Then you can hope to get a big savings if you store your computations so that you only compute each different subproblem

[12] Columbia class slides: `http://www.columbia.edu/~cs2035/courses/csor4231.F11/matrix-chain.pdf`.

once. You can store these solutions in an array or hash table.

This view of Dynamic Programming is often called *memoizing*.

For example, the longest common subsequence (LCS) problem can be solved with this top-down approach. Here is the pseudo-code, from the CMU lecture note.

```
LCS(S,n,T,m)
{
    if (n==0 || m==0)
        return 0;
    if (arr[n][m] != unknown)
        return arr[n][m]; // memoization (use)
    if (S[n] == T[m])
        result = 1 + LCS(S,n-1,T,m-1);
    else
        result = max( LCS(S,n-1,T,m), LCS(S,n,T,m-1) );
    arr[n][m] = result; // memoization (store)
    return result;
}
```

If we compare the above code with the bottom-up formula for LCS, we realize that they are just using the same algorithm, with same cases. The idea that both approaches share is that, we only care about computing the value for a particular subproblem.

2.4 Recursive algorithms

2.4.1 Divide and Conquer

The matrix chain multiplication problem discussed previously can be solved, using top-down approach, with recursion, and the idea there is basically divide and conquer – break up the big chain into smaller chains, until $i = j$ (Opt$[i, j]$=0). *Divide and conquer* (D&C) works by recursively breaking down a problem into two or more sub-problems of the same or related type, until these problems are simple enough to be solved directly[13]. For some problems, we can use memoization technique to optimize the run time.

Now, let us look at two well-known problems solvable by divide-and-conquer algorithms.

[13] From Wikipedia.

Binary Search This search algorithm runs in $O(logn)$ time. It works by comparing the target element with the middle element of the array, and narrow the search to half of the array, until the middle element is exactly the target element, or until the remaining array has only one element. Binary search is naturally a divide-and-conquer algorithm.

```
Binary-Search-Recursive(arr, target, lo, hi):
    # lo inclusive, hi exclusive.
    if hi <= lo:
        return NOT FOUND
    mid = lo + (hi-lo)/2
    if arr[mid] == target:
        return mid
    else if arr[mid] > target
        return Binary-Search-Recursive(arr, target, mid+1, hi)
    else:
        return Binary-Search-Recursive(arr, target, lo, mid)

Binary-Search-Iterative(arr, target):
    lo = 0
    hi = arr.length
    while lo < hi:
        mid = lo + (hi-lo)/2
        if arr[mid] == target:
            return mid
        else if arr[mid] > target:
            lo = mid + 1
        else:
            hi = mid
    return NOT FOUND
```

Implement the square root function: To implement the square root function programmatically, with integer return value, we can use binary search. Given number n, we know that the square root of n must lie between 0 and $n/2$. Then, we can basically treat all integers in $[0, n/2]$ as an array, and do binary search. We terminate only when `lo` equals to `hi`.

Closest Pair of Points Given n points in metric space, e.g. plane, find a pair of points with the smallest distance between them. A divide-and-conquer algorithm is as follows (from Wikipedia):

1. Sort points according to their x-coordinates.

2. Split the set of points into two equal-sized subsets by a vertical line $x = x_{split}$.
3. Solve the problem recursively in the left and right subsets. This yields the left-side and right-side minimum distances d_{Lmin} and d_{Rmin}, respectively.
4. Find the minimal distance d_{LRmin} among the set of pairs of points in which one point lies on the left of the dividing vertical and the other point lies to the right.
5. The final answer is the minimum among d_{Lmin}, d_{Rmin}, and d_{LRmin}.

The recurrence of this algorithm is $T(n) = 2T(n/2) + O(n)$, where $O(n)$ is the time needed for step 4. This recurrence to $O(nlogn)$. Why can step 4 be completed in linear time? How? Suppose from step 3, we know the current minimum distance is δ. For step 4, we first pick the points with x-coordinates that are within $[x_{split} - \delta, x_{split} + \delta]$, call this the *boundary zone*. Suppose we have $p_1, \cdots, p_m$ inside the boundary zone. Then, we have the following magical theorem.

Theorem 2.11. *If dist(p_i, p_j) $< \delta$, then $j - i \leq 15$.*

With this, we can write the pseudo-code for this algorithm[14]:

```
Closest-Pair(P):
    if |P| == 2:
        return dist(P[0], P[1])

    L, R = SplitPointsByHalf(P)
    dL = Closest-Pair(L)
    dR = Closest-Pair(R)
    dLR = min(dL, dR)

    S = BoundaryZonePoints(L, R, dLR)
    for i = 1, ..., |S|:
        for j = 1, ..., 15:
            dLR = min(dist(S[i], S[j]), d)
    return dLR
```

Obviously, there are other classic divide-and-conquer algorithms to solve problems such as the convex hull (two-finger algorithm), and the median of medians algorithm (groups of five). As the writer, I have read those

[14] Cited from CMU lecture slides, with modification `https://www.cs.cmu.edu/~ckingsf/bioinfo-lectures/closepoints.pdf`

algorithms and understood them, but I will save my time and not discuss them here.

2.4.2 Backtracking

Backtracking is a general algorithm for finding all (or some) solutions to some computational problems, notably constraint satisfaction problems, that incrementally builds candidates to the solutions, and abandons each partial candidate c ("backtracks") as soon as it determines that c cannot possibly be completed to a valid solution[15].

Solving Sudoku *Sudoku* is a puzzle that is played on a grid of 9 by 9 cells, such that when all cells are filled up with numbers, each row and column have an enumeration of $1, 2, \cdots, 9$, and so does each "big cell" (*subregion*). It needs no more introduction than that. Here are the constraints for a Sudoku puzzle:

1. Each cell can contain one number in $\{1, 2, 3, 4, 5, 6, 7, 8, 9\}$
2. Each row, column, and subregion all have an enumeration of the numbers 1 through 9, with no repeat.

Pseudo-code[16]. The idea of backtracking is illustrated in the `undo & try again` step.

```
bool SolveSudoku(Grid<int> &grid)
{
 int row, col;
 if (!FindUnassignedLocation(grid, row, col))
    return true; // all locations successfully assigned!

 for (int num = 1; num <= 9; num++) { // options are 1-9
    if (NoConflicts(grid, row, col, num)) { // if # looks ok
        grid(row, col) = num; // try assign #
        if (SolveSudoku(grid))
            return true; // recur if succeed stop
        grid(row, col) = UNASSIGNED; // undo & try again
    }
 }
 return false; // this triggers backtracking from early decisions
}
```

[15] Description of backtracking is from Wikipedia.
[16] From `https://see.stanford.edu/materials/icspacs106b/Lecture11.pdf`

Relation between backtracking and DFS: DFS is a specific form of backtracking related to searching tree structures. Backtracking is more broad – it can be used for searching in non-tree structures, such as the Sudoku board.

2.5 Greedy algorithms

A greedy algorithm is an algorithm that makes locally optimal decisions, with the hope that these decisions would lead to globally optimal solution. It is easy to come up with these greedy rules, but most of them are wrong, and the right ones are typically hard to justify. So nothing is better than showing examples when discussing greedy algorithms.

Scheduling Problem [17] There is a computer system with numerous processes, some of which is marked *sensitive*. Each sensitive process has a designated start time and finish time, and it runs continuously between these times.There is a list of the planned start and finish times of all sensitive processes that will be run that day.

You are given a program called `status_check` that, when invoked records various pieces of logging information about all the sensitive processes running on the system at that moment. You should run `status_check` as few times as possible during the day, but enough that for each sensitive process P, `status_check` is invoked at least once during the execution of process P.

Give an efficient algorithm that, given the start and finish times of all the sensitive processes, finds as small a set of times as possible at which to invoke `status_check` , subject to the requirement that `status_check` is invoked at least once during each sensitive process P.

Algorithm: We start by sorting the processes by descending start time, using a heap (i.e. the process with highest start time will have the minimum value in the heap). Then, we keep removing the root process from the heap, and use the start time of this process as the time to call `status_check`. And we mark processes that are running at that time as checked, and remove them from the heap as well. Here is a pseudo-code to illustrate this algorithm

```
CountCall(processes):
    Heap h = a heap of processes ordered by descending start time.
    count_calls = 0
    While h is not empty:
```

[17] This problem is from Kleinberg Algorithm Design, pp.194 14.

```
    p = h.RemoveMin()
    call_time = start time of p
    count_calls += 1
    For each process q in h:
        If q is running at call_time:
            h.Remove(q)
Return count_calls
```

Justification for correctness: The above algorithm **will terminate** because there is finite number of sensitive processes, and thus the heap will eventually be empty when all processes are checked.

We can use *induction* to show that the above algorithm produces correct result. Suppose we have n sensitive processes in total. Processes are labeled $P_0, P_1, P_2, ..., P_n$. Let W_n be defined as the number of calls to `status_check` when there are n processes.

Proof. **Base Case:** The base case when n=0 is trivial. The algorithm will simply return 0 since the heap is empty. This is a correct behavior. So, the algorithm works for the base case.

Induction Hypothesis: Assume that for $0 <= j <= k$, our algorithm produces minimum possible value of W_j.

Inductive Step: Now show that for $n = k+1$, our algorithm still produces minimum possible value for W_{k+1}. For P_{k+1}, there are two cases to consider:

1. We need to call `status_check` once more in order to check P_{k+1}, because P_{k+1} is not checked when we handle the other $P_0, ..., P_k$ processes.
2. We do not need to call `status_check` any more, because P_{k+1} is checked when we handle the other $P_0, ..., P_k$ processes.

For case (a), since our algorithm will only terminate when the heap is empty, so when P_{k+1} is not checked, it is still in the heap. Therefore, the algorithm will do one extra `RemoveMin()` and remove P_{k+1} from the heap. By the induction hypothesis, the algorithm produces optimal result for $0 <= j <= k$. Thus, the result produced by the algorithm for $n = k+1$ matches the optimal in case (a), which requires one extra call to the `status_check` function.

For case (b), since P_{k+1} is checked when we handle $P_0, ..., P_k$, our algorithm will have already removed P_{k+1} when it is done dealing with $P_0, ..., P_k$. By the induction hypothesis, the algo-

rithm produces optimal result for $0 <= j <= k$. Thus, the result produced by the algorithm for $n = k + 1$ matches the optimal in case (b), which is to NOT call `status_check` any more.

Conclusion From the above proof of base case and induction step, by Strong Induction, we have shown that our algorithm works for integer $n >= 0$. □

Indeed, induction is how you formally prove that a greedy rule works correctly.

Justification for run time: The above algorithm is efficient. We first construct a heap of processes, which takes $O(nlogn)$ time. Then we loop until we remove all items inside the heap, which is $O(nlogn)$ time. Since we do not add any process back into the heap after we removed it, the algorithm will terminate when the heap is empty. Besides, any other operations in the algorithm is $O(1)$. Therefore, combined, our algorithm has an efficient runtime of $O(nlogn) + O(nlogn) = O(nlogn)$.

2.6 Sorting

2.6.1 Merge sort

Merge sort is a divide-and-conquer, stable sorting algorithm. Worst case $O(nlogn)$; Worst space $O(n)$. Here is a pseudo-code for non-in-place merge sort. An in-place merge sort is possible.

```
Mergesort(arr):
    if arr.length == 1:
        return arr
    l = Mergesort(arr[0:mid])
    r = Mergesort(arr[mid:length])
    return merge(l, r)
```

2.6.2 Quicksort

Quicksort is a divide-and-conquer, unstable sorting algorithm. Average run time $O(nlogn)$; Worst case run time $O(n^2)$; Worst case auxiliary space[18] $O(logn)$ with good implementation. (Naive implementation is $O(n)$ space still.) Quicksort is fast if all comparisons are done with constant-time memory access (assumption). People have argued which sort is the best. Here is an answer from Stackoverflow, by `user11318`:

[18] Auxiliary Space is the extra space or temporary space used by an algorithm. From GeeksForGeeks.

> ... However if your data structure is big enough to live on disk, then quicksort gets killed by the fact that your average disk does something like 200 random seeks per second. But that same disk has no trouble reading or writing megabytes per second of data sequentially. Which is exactly what merge sort does.
>
> **Therefore if data has to be sorted on disk, you really, really want to use some variation on merge sort.** (Generally you quicksort sublists, then start merging them together above some size threshold.) ...

Here is the pseudo-code:

```
Quicksort(arr, lo, hi):
    # lo inclusive, hi exclusive
    if hi <= lo:
        return
    pivot = ChoosePivot(arr, lo, hi)
    p = Partition(arr, lo, hi, pivot)
    Quicksort(arr, lo, p)
    Quicksort(arr, p, hi)

Partition(arr, lo, hi, pivot):
    # lo inclusive, hi exclusive
    i = lo, j = hi
    while i < j:
        if arr[i] < pivot:
            i += 1
        else:
            swap(arr, i, j-1)
            j -= 1
    return i # sorted position of pivot
```

A nice strategy for choosing pivot is to just choose randomly. Another good way is to choose the median value from the first, last and middle element of the array.

2.6.3 Bucket sort

Bucket sort in some cases can achieve $O(n)$ run time. But it is unstable (worst case $O(n^2)$. It works by distributing the elements of an array into a number of buckets. Each bucket is then sorted individually, either using a different sorting algorithm, or by recursively applying the bucket sorting algorithm.

2.6.4 Radix sort

Radix sort is a non-comparison sort, where the array must only contain elements that are integers. Suppose the array arr has size n, and each value $arr_i \in \{1, \cdots, k\}$, $k = n^{O(1)}$ and arr_i has base b. Then radix sort can complete this sorting task in time $O(cn)$, where $c = log_n k$.

More specifically, radix sort basically sorts the array of integers by by each digit, and uses counting sort for each digit sorting.

> *Counting sort* works when the given array is of integers, and each ranges from p to q; it creates an array, say M, of $p - q$ elements, counts the number of occurrence each element in the given array, and records it into M, and then the sorted order can be produced by traversing M and repeating each element[19] with the occurrence recorded. The run time of this sort is $O(n + (p - q))$.

Suppose the number in arr has base b. So the time needed to sort by each digit is $O(n + b)$ using counting sort. Then, the number of digits of arr_i is maximum $d = log_b k$. Thus, the run time of radix sort can be derived:

$$O((n + b)d) = O((n + b)log_b k) = O(nc) \quad \text{when we choose } b = n$$

Implementation by Github user `yeison`: `https://gist.github.com/yeison/5606963`. In this implementation, the coder assumes that each integer fits in a word of size 32 bits.

2.7 Searching

Graph search algorithms have been discussed already. Binary search has been discussed in the divide-and-conquer section. We will only look at quickselect here. Searching is a topic where interviewers like to ask questions, for example, search for an element in a sorted an *rotated*[20] array.

2.7.1 Quickselect

Quickselect is a method to select the element with rank k in an unsorted array. Here is the pseudocode:

```
Quickselect(arr, k, lo, hi):
```

---

[19] Need appropriate shifting (adding a), since we have p as lower bound.

[20] A sorted rotated array is one that has a pivot, and if we swap the whole region of element on one side of the pivot with the other side, then we obtain a sorted array.

```
    if hi <= lo:
        return arr[lo]
    pivot = ChoosePivot(arr, lo, hi)
    p = Partition(arr, lo, hi, pivot)
    if p == k:
        return arr[k]
    else if p > k:
        return Quickselect(arr, k, lo, p)
    else:
        return Quickselect(arr, k-p, p, hi)
```

The (expected) recurrence for the above psuedo-code is $T(n) = T(n/2) + O(n)$. When solved, it gives $O(n)$ run time. For the worst case, which is also due to bad pivot selection, the run time is $O(n^2)$.

2.8 String

2.8.1 Regular expressions

Regular expression needs no introduction. In interviews, the interviewer may ask you to implement a regular expression matcher for a subset of regular expression symbols. Similar problems could be asking you to implement a program that can recognize a particular string pattern.

Here is a regex matcher written by Brian Kernighan and Rob Pike in their book *The Practice of Programming*[21]

```
/* match: search for regexp anywhere in text */
int match(char *regexp, char *text)
{
    if (regexp[0] == '^')
        return matchhere(regexp+1, text);
    do {    /* must look even if string is empty */
        if (matchhere(regexp, text))
            return 1;
    } while (*text++ != '\0');
    return 0;
}

/* matchhere: search for regexp at beginning of text */
int matchhere(char *regexp, char *text)
{
```

[21] Discussed here http://www.cs.princeton.edu/courses/archive/spr09/cos333/beautiful.html

```
    if (regexp[0] == '\0')
        return 1;
    if (regexp[1] == '*')
        return matchstar(regexp[0], regexp+2, text);
    if (regexp[0] == '#' \&\& regexp[1] == '\0')
        /* # means dollar sign here! */
        return *text == '\0';
    if (*text!='\0' && (regexp[0]=='.' || regexp[0]==*text))
        return matchhere(regexp+1, text+1);
    return 0;
}

/* matchstar: search for c*regexp at beginning of text */
int matchstar(int c, char *regexp, char *text)
{
    do {    /* a * matches zero or more instances */
        if (matchhere(regexp, text))
            return 1;
    } while (*text != '\0' \&\& (*text++ == c || c == '.'));
    return 0;
}
```

2.8.2 Knuth-Morris-Pratt (KMP) Algorithm

The KMP algorithm is used for the string matching problem.

> Find the index that a pattern P with length m occurs (if ever) in a string W with length n.

The naive algorithm to solve this problem takes $O(nm)$ time, which does not utilize any information when a matching failed. The key of KMP is that it uses it and achieves run time of $O(n+m)$. It is a complicated algorithm, and let me explain it now. See the appendix 5.2 for my Python implementation, based on the ideas below.

Building prefix table (π table) The first thing that KMP does is to preprocess the pattern P and create a π table. $\pi[i]$ is the largest integer *smaller than* i such that $P_0 \cdots P_{\pi[i]}$ is a suffix of $P_0 \cdots P_i$. Consider the following example:

i	0	1	2	3	4	5	6	7
P_i	a	b	a	b	c	a	b	a
$\pi[i]$	-1	-1	0	1	-1	0	1	2

When we are filling $\pi[i]$, we focus on the substring $P_0 \cdots P_i$, and see if there is a prefix equal to the suffix in that substring. $\pi[0], \pi[1], \pi[4]$ are -1, meaning that there is no prefix equal to suffix in the corresponding substring. For example, for $\pi[4]$, the substring of concern is *ababc*, and there is no valid index value for $\pi[4]$ to be set. $\pi[7] = 2$, because the substring $P_0 \cdots P_7$ is *ababcaba*, and the prefix $P_0 \cdots P_2$, *aba*, is a suffix of that substring.

Below is a pseudo-code for constructing a π table. The idea behind the pseudo-code is captured by two observations:

1. If $P_0 \cdots P_{\pi[i]}$ is a suffix for $P_0 \cdots P_i$, then $P_0 \cdots P_{\pi[i]-1}$ is a suffix for $P_0 \cdots P_{i-1}$ as well.
2. If $P_0 \cdots P_{\pi[i]}$ is a suffix for $P_0 \cdots P_i$, then so does $P_0 \cdots P_{\pi[\pi[i]]}$, and so does $P_0 \cdots P_{\pi[\pi[\pi[i]]]}$, etc., a recursion of the π values.

So we can use two pointers i, j, and we are always looking at if the prefix $P_0 \cdots P_{j-1}$ is a suffix for the substring $P_0 \cdots P_{i-1}$. So pointer i moves quicker than pointer j. In fact i moves up by 1 every time we are done with a comparison between P_i and P_j, and j moves up by 1 when $P_i = P_j$ (observation 1). At this time ($P_i = P_j$), we set $\pi[i] = j$. If $P_i \neq P_j$, we will move j back to $\pi[j-1]+1$ (+1 because $\pi[i]$ is -1 when there is no matching prefix.) This guarantees that the prefix $P_0 \cdots P_{j-1}$ is the longest suffix for the substring $P_0 \cdots P_{i-1}$. We need to initialize $\pi[-1] = -1$ and $\pi[0] = -1$.

```
Construct-π-Table(P):
    j = 0, i = 1
    while i < |P|:
        if P_i = P_j:
            π[i] = j
            i += 1, j += 1
        else:
            if j > 0:
                j = max(0, π[j-1]+1)
            else:
                π[i] = -1
                i += 1
```

Pattern Matching Once we have the π table, we can skip characters when comparing the pattern P and the string W. Consider P and W to

be the following, as an example. (P is the same as the above example.)

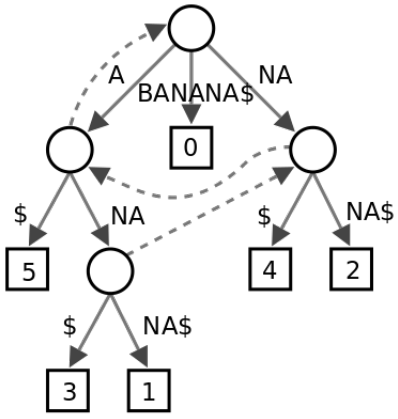
$$W = abccababababca\boldsymbol{ababcaba}, \quad P = ababcaba$$

Based on the way we construct the π table above, we have the following rules when doing the matching. Assume the matched substring (i.e. the substring of P before the first mismatch, starting at $W[k]$, has length d.

1. If $d = |P|$, we found a match. Return k.
2. Else, if $d > 0$, and $\pi[d-1] = -1$, then the next comparison starts at $W[k+d]$.
3. Else, if $d > 0$, and $\pi[d-1] \neq -1$, then the next comparison starts at. $W[k+d-\pi[d-1]-1]$.Note: we don't need the -1 here if π table is 1-based index. See Stanford slides.
4. Else, if the matched substring, starting at index k, has length 0, then the next match starts at $k+1$.

2.8.3 Suffix/Prefix Tree

See Trie 2.1.15. If we have a text of size T, and a small pattern of size P, and we are interested to know if P occurs in T, then we can achieve $O(P)$ time and $O(T)$ space by building a suffix tree of the text T. A suffix tree is a *compressed trie* containing all the suffixes of the given text as their keys and positions in the text as their values. Suffix trees allow particularly fast implementations of many important string operations (Wikipedia). The construction of such a tree for the string T takes time and space linear in the length of T. Here is an example of a suffix tree, for $T =$ "banana$". (The $ sign is for marking the end of a string.)



2.8.4 Permutation

String permutation is another topic that interviewers may like to ask. One generates permutations typically by depth-first search (i.e. a form of backtracking); we can imagine that all possible permutations are the leafs of a tree, and the paths from root to them represent the characters chosen.

2.9 Caching

In general a *cache* is a location to store a small amount of data for more convenient access. It is everywhere. Here are some common examples, described at a high level.

CPU cache is used by the CPU to quickly access data in the main memory. Depending on the type of memory data, e.g. regular data, instruction, virtual address (translation lookaside buffer used by MMU), etc., there may be different types of caches, such as data cache and instruction cache.

Cache server (web cache) basically saves web data (e.g. web page, requests) and serve them when the user request the same thing again, in order to reduce the amount of data transmitted over the web.

2.9.1 Cache Concepts Review

Definition 2.2. *Cache hit* is when we request something that is in the cache. *Cache miss* is when the requested item does not exist in the cache.

Definition 2.3. A *cache block*, or cache line, is a section of continuous bytes on a cache. It is the lowest I/O level for a cache.

Locality refers to the fact that programs tend to use data and instructions close to (spatial) some of those recently used (temporal).

Memory Cache *Direct mapped cache* is a cache where each memory address can be mapped to exactly one block in the cache. Each block is divided into three sections: tag, data, and valid bit. The tag stores the first few bits of an address. The data stores the cached memory data, which can consist of several blocks of typically 32 or 64 bytes. The valid bit indicates whether the cached data can be trusted by the CPU for computation. When CPU needs to refer to some data with an address,

it will figure out the tag on that address (e.g. by dividing page size), and check if the *mapped block* in the cache has the same tag, and if the valid bit is set. If so, then the cached data is usable.

Fully-Associative cache is one that allows any memory page to be cached in any cache block, opposite to direct mapped cache. The advantage is that it avoids the possibly constantly empty entries in a direct mapped cache, so that the cache miss rate is reduced. The drawback is that such cache requires hardware sophistication to support parallel look-up of tags, because in this case, the only way to identify if an address is cached is to compare the tag on that address with all tags in the cache (in parallel).

In CSE 351, we adopted the *set-associative cache* as the reasonable compromise between complicated hardware and the direct mapped cache. Here, we divide the cache into sets, where each set contains several entries. This way, we can reduce the cache miss rate compared to direct mapped cache, but also check the tags efficiently enough in parallel, because there are only a few entries in a set. We say a cache is n-way, if in each set there are n cache blocks.

For the sake of interviews, we will discuss LRU cache and LFU cache (in software). Both of them fall under the topic of *cache replacement policies*, which means that when a cache is full, how do we evict cached data. There are numerous policies, including FIFO, LIFO, LRU (Least Recently Used), LFU (Least Frequently Used), etc. In hardware, these caches are usually implemented by manipulating some bits (e.g. LRU counter for each cache block) in the block to keep track of some property such as age. Concepts are similar.

2.9.2 LRU Cache

LRU cache policy is to evict the least recently used data first. In software, we can use a doubly linked list plus a hash table to implement the LRU cache. *Each node in the list corresponds to a key-value pair*[22] *in the hash table.* When we insert new key-value pair into the cache, we also add a node to the front of the list. When the cache is full and we still need to add data to it, we basically remove the last element in the list, because it is least recently used. When we actually have a cache hit, we can simply bring the corresponding node to the front of the list,

[22] In the context of a memory cache, we can think of the key as the address, and the value as the cache block associated with the tag of that address.

by removing it (constant time for doubly linked list) then prepending it to the list.

2.10 Game Theory

Game theory is basically the theory of modeling intelligent rational decision making process in different kinds of situations. It is not just used in computer science, but also in economics and political science. This is not a very hot topic in coding interviews, but knowing these algorithms may help in some cases – they are essentially various sorts of searching algorithms with different model of the world.

Some concepts In economics, game theory, decision theory, and artificial intelligence, *a rational agent* is an agent that has clear preferences, models uncertainty via expected values of variables or functions of variables, and always chooses to perform the action with the optimal expected outcome for itself from among all feasible actions. A rational agent can be anything that makes decisions, typically a person, firm, machine, or software[23]. A measure of preferences is called *utility*. A game where agents have opposite utilities is a *zero-sum game*, e.g. chess; agents go against each other in a adversarial, pure competition. A *general game* is one where agents have independent utilities; Cooperation, indifference, competition, etc. are all possible in this kind of game.

2.10.1 Minimax and Alpha-beta

Minimax is an intuitive adversarial search algorithm for deterministic games. There is an agent A and an agent (opponent) Z. Minimax follows the following two equations[24]:

$$V_A(s_A) = \max_{s_Z \in \text{successors}(s_A)} V_A(s_Z)$$
$$V_Z(s_Z) = \min_{s_A \in \text{successors}(s_Z)} V_Z(s_A)$$

where $V_K(s)$ means the utility function of agent K for a state s. The parameter s_K is the state that agent K takes control – agent K makes the decision of how to change from s_K to some successor state. So, agent A tries to maximize the utility, and Z tries to minimize its utility.

[23] From Wikipedia
[24] Modified from CSE 473 lecture slides, 2016 spring, by Prof. L. Zettlemoyer.

Alpha-beta is a pruning method for Minimax tree. The output of Alpha-beta is the same as the output of Minimax. The α value represents the assured maximum score that the agent A can get, and the β value is the assured minimum score that agent Z can get.

Below is my Python implementation of alpha-beta, when doing the pacman assignment. Only the agent with index 0 is a maximizing agent.

```
def _alphabeta(self, gameState, agentIndex, depth, alpha, beta):
       if gameState.isWin() or gameState.isLose() or depth == 0:
           score = self.evaluationFunction(gameState)
           return score

       curAgent = agentIndex % gameState.getNumAgents()
       legalActions = gameState.getLegalActions(curAgent)

       score = -float("inf")
       if agentIndex != 0:
           score = -score
       nextAgent = curAgent + 1
       if nextAgent >= gameState.getNumAgents():
           nextAgent = 0
           depth -= 1

       for action in legalActions:
           successorState = gameState.generateSuccessor(curAgent,
               action)
           if curAgent == 0:
               score = max(score, self._alphabeta(successorState,
                   nextAgent, depth, alpha, beta))
               if score > beta:
                   return score
               alpha = max(alpha, score)
           else:
               score = min(score, self._alphabeta(successorState,
                   nextAgent, depth, alpha, beta))
               if score < alpha:
                   return score
               beta = min(beta, score)
       return score
```

2.10.2 Markov Decision Process

A Markov Decision Process (MDP) is defined by:

- A set of states $s \in S$,
- A set of actions $a \in A$,
- A transition function $T(s, as')$ for the probability of transition from s to s' with action a,
- A reward function $R(s, as')$,
- A start state,
- (maybe) a terminal state.

The world for MDP is usually a grid world, where some grids have positive reward, and some have negative reward. The goal of solving an MDP is to figure out an optimal policy $\pi^*(s)$, the optimal action for state s, so that the agent can take actions according to the policy in order to gain the highest amount of reward possible. There are two ways to solve MDP discussed in the undergraduate level AI class: *value (utility) iteration* and *policy iteration*. I will just put some formulas here. For most of the time, understanding them is straightforward and sufficient.

Definition 2.4. The utility of a state is $V^*(s)$, the expected utility starting in s and acting optimally.

Definition 2.5. The utility of a q-state[25] is $Q^*(s, a)$, the expected utility starting by taking action a in state s, and act optimally afterwards.

Using the above definition, we have the following recursive definition of utilities. The γ value is a *discount*, from 0 to 1, which can let the model prefer sooner reward, and help the algorithm converge. Note max is different from argmax.

$$\begin{aligned} V^*(s) &= \max_a Q^*(s, a) \\ Q^*(s, a) &= \sum_{s'} T(s, a, s')[R(s, a, s') + \gamma V^*(s')] \\ V^*(s) &= \max_a \sum_{s'} T(s, a, s')[R(s, a, s') + \gamma V^*(s')] \end{aligned}$$

Value Iteration: Start with $V_0(s) = 0$ for all $s \in S$. Then, we update V_{k+1} iteratively using the following (almost trivial) update rule, until convergence. Complexity of each iteration: $O(S^2 A)$.

$$V_{k+1}(s) \leftarrow \max_a \sum_{s'} T(s, a, s')[R(s, a, s') + \gamma V_k(s')]$$

[25] The naming, q-state, from my understanding, means quasi-state, which is seemingly a state, but not really.

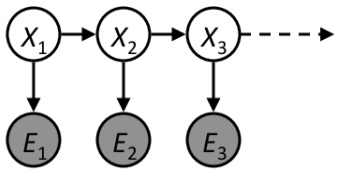
Policy Iteration: Start with an arbitrary policy π_0, then iteratively *evaluate* and *improve* the current policy until policy converges. This is better than value iteration in that policy usually converges long before value converges, and the run time for value iteration is not desirable.

$$V_{k+1}^{\pi_i} \leftarrow \sum_{s'} T(s, \pi_i(s), s')[R(s, \pi_i(s), s') + \gamma V_k(s')^{\pi_i}]$$

$$\pi_{i+1}(s) = \operatorname*{argmax}_a \sum_{s'} T(s, a, s')[R(s, a, s'] + \gamma V^{\pi_i}(s')]$$

2.10.3 Hidden Markov Models

A Hidden Markov Model looks like this:



A state is a value of a variable X_i. For example, if X_i is a random variable meaning "it rains on day i", then the value of X_i can be True or False. If $X_1 = t$, then we have a state X_1 which means it rains on day 1.

An HMM is defined by an initial distribution $P(X_1)$, transitions $P(X_t|X_{t-1})$, and emissions $P(E_t|X_t)$. The value of an emission variable represents an observation, e.g. sensor readings.

We are interested to know $P(X_t|e_{1:t})$, which is the distribution of X_t given all of the observations to date. We can obtain the joint distribution of $x_t \in X_t$ and all current observations.

$$P(x_t, e_1, \cdots, e_t) = P(e_t|x_t) \sum_{x_{t-1}} P(x_t|x_{t-1}) P(x_{t-1}, e_1, \cdots, e_{t-1})$$

Then, we normalize all entries in $P(X_t, e_1, \cdots, e_t)$ to the desired current belief, $B(X_t)$, by the definition of conditional probability.

$$B(X_t) = P(X_t|e_{1:t}) = P(X_t, e_1, \cdots, e_t) / \sum_{x_t} P(x_t, e_1, \cdots, e_t)$$

This is called *the forward algorithm*. From this, we can derive the formula for the belief at the next time frame, given current evidence:

$$\begin{aligned} B'(X_{t+1}) = P(X_{t+1}|e_{1:t}) &= \sum_{x_t} P(X_{t+1}|x_t)P(x_t|e_{1:t}) \\ &= \sum_{x_t} P(X_{t+1}|x_t)B(x_t) \end{aligned}$$

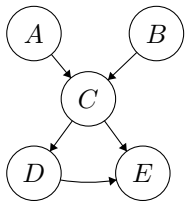
The above equation allows us to perform online belief updates.

2.10.4 Baysian Models

Baysian Network is based on the familiar Bayes's Theorem:

$$P(A|B) = \frac{P(B|A)P(A)}{P(B)}$$

A Baysian Network can be represented by a DAG. A diagram of an example network is as below.



Each node is a random variable. The edges encode conditional independence; nodes that are not connected represent variables that are conditionally independent of each other. For example, in the diagram below, A and B are conditionally independent given the other variables. Recall that if X, Y are conditionally independent given Z, then $P(XY|Z) = P(X|Z)P(Y|Z)$.

Besides, a Baysian Network implicitly encode a joint distribution. For the above diagram, we have:

$$P(A, B, C, D, E) = P(E|C, D)P(C|A, B)P(D|C)P(A)P(B)$$

In general,

$$P(x_1, x_2, \cdots, x_n) = \prod_{i=1}^{n} P(x_i|\text{parents}(X_i))$$

2.11 Computability

This section is about some fundamental theory of computer science. I am not sure if any interview will ask questions directly related to them, but knowing them definitely helps.

2.11.1 Countability

A set S is countable if and only if the elements in S can be mapped on-to $\mathbb{N}$. The union of countably many countable sets is countable. (Given axiom of choice)

2.11.2 The Halting Problem

A set is called *decidable* if there is an algorithm which terminates after a finite amount of time and correctly decides whether a given element belongs to the set.

Here is my proof of a 311 problem related to the Halting Problem. The problem is the following:

Prove that the set $\{$<CODE(P)>$, x, y) : P$ is a program and $P(x) \neq P(y)\}$ *is undecidable.*

1. Assume that there exists an algorithm A that satisfies the above. Note that the argument for A is an element in the set, i.e. (CODE(P), x ,y). A has return value of true or false. True when the element is in, and false otherwise.

2. Suppose we have the following program D, inside which there is an arbitrary program H. We claim that we can use A to show if H halts:

```
D(x):
    if x \% 2 == 0:
        while(1)
    else:
        return H()  # H is an arbitrary function
```

3. As you can see, the output of this program is either "loop forever", or what H returns.

4. If A(CODE(D), 1, 2) returns true, then $D(1) \neq D(2)$. So D(1) halts, which means H halts.

5. If $A(\text{CODE}(D), 1, 2)$ returns false, then $D(1) = D(2)$. So $D(1)$ does not halt. (Bascially, when two outputs D(1)= D(2), the only chance that happens is that both when into infinite loop, which does not return a value (abstractly, this "infinite loop" is the returned value).

6. Suppose H is a program that actually plays the same role as A but for the halting set, a set defined like this:

$$\{ (i, x) \text{ — program } i \text{ halts when run on input } x \}$$

7. Because of Halting Problem, we know that this set is undecidable. So here is a contradiction.

8. Thus, the given set is undecidable.

2.11.3 Turing Machine

A Turing Machine is basically a conceptual machine, imagined using the materials in Alan Turing's time (1930s), that can be used to do computation with algorithmic logic. A Turing Machine can only solve problems that are decidable, which means that there exists a single program that always correctly outputs Yes/No.

limitations A limitation of Turing machines is that they do not model the strengths of a particular arrangement well. Another limitation of Turing machines is that they do not model concurrency well (Wikipedia).

2.11.4 P-NP

The following definitions and theorems are provided by Algorithm Design, book by Kleinberg et. al., and the CSE 332 lecture slides on P-NP, by Adam Blank.

Definition 2.6. A *complexity class* is a set of problems limited by some resource constraint (e.g. time, space).

Definition 2.7. A *decision problem* is a set of strings ($L \in \Sigma*$). An algorithm (from $\Sigma*$ to boolean) solves a decision problem when it outputs True if and only if the input is in the set.

Definition 2.8. P is the set of decision problems with a polynomial time (in terms of the input) algorithm

Definition 2.9. NP (1) NP is the set of decision problems with a non-deterministic polynomial time algorithm.

Definition 2.10. A *certifier* for problem X is an algorithm that takes as input: (1) a string s which is an instance of X; (2) A string w which is a "certificate" or "witness" that $s \in X$, and returns False if $s \in X$ regardless of w, and returns True otherwise, i.e. there exists some w to let $s \in X$.

Definition 2.11. NP (2) NP is the set of decision problems with a polynomial time certificate.

Definition 2.12. Suppose X, Y are two different problems. We say X is *at least as hard as* Y if there is a "black box" capable of solving X, and if we can use polynomial operations plus polynomial number of calls to X in order to solve Y. This also means that $Y \leq_P X$.

In otherwords, X is powerful enough for us to solve Y.

Theorem 2.12. *Suppose $Y \leq_P X$. If X can be solved in polynomial time, then Y can be as well.*

Theorem 2.13. *Suppose $Y \leq_P X$. If Y cannot be solved in polynomial time, then X cannot either.*

Theorem 2.14. $P \subseteq NP$.

Definition 2.13. Problem X is NP-hard if for all $Y \in NP$, $Y \leq_P X$.

Definition 2.14. Problem X is NP-Complete if and only if $X \in NP$ and for all $Y \in NP$, $Y \leq_P X$ (X is NP-hard).

Theorem 2.15. *Suppose X is an NP-Complete problem. Then X is solvable in polynomial time if and only if $P = NP$.*

Example P-NP reduction. Multiple Interval Scheduling Problem (Algorithm Design, pp.512 14): you're given a set of n jobs, each specified by a set of time intervals, and you want to answer the following question: For a given number k, is it possible to accept at least k of the jobs so that no two of the accepted jobs have any overlap in time?

Multiple Interval Scheduling Problem is in NP We can find a polynomial time certifier, which takes as inputs an instance of Multiple Interval Scheduling Problem: (1) n jobs, (2) number k, and a certificate a scheduling L of time intervals. We can implement such certifier by checking if L has overlap in O($|L|$) time, and also count the number of jobs accepted by L in O($|L|$) time. Combining these two operations, we have a polynomial time certifier.

Multiple Interval Scheduling Problem is NP-hard Now we show that we can reduce an instance of the Multiple Interval Scheduling Problem (MISP) to Independent Set Problem (ISP). Suppose we are given an instance of ISP: a graph $G = (V, E)$, and a number k. We can create an instance of MISP in polynomial time by: First, for each node $v_i \in V$, we create a job l_i. For each edge $e \in E$ with end nodes v_i and v_j , we create an interval such that jobs l_i and l_j require to work in that interval. Then, we use the same number k in MISP. Now we claim that there is k non-overlapping jobs scheduled if and only if the corresponding Independent Set Problem has independent set of size k. Suppose we have k non-overlapping jobs scheduled. Then, because if two jobs overlap, they must require to work in some identical interval. Therefore, the k nodes corresponding to the k jobs must have no edges connecting each other, which means that it is an independent set of size k. Suppose we have an independent set of size k. Then because of the way we construct the MISP, the corresponding k jobs must have no overlapping intervals. Therefore, we proved our claim.

2.12 Bitwise operators

Basic operators include NOT, AND, OR and XOR.

NOT	$\sim$	$\sim$(0111) = 1000
AND	&	011 & 110 = 010
OR	|	011 | 110 = 111
XOR	$\veebar$	011$\veebar$110 = 101

Bit shifts include *arithmetic shift* and *logical shift*. In arithmetic right shift, the vacant bit-positions are filled with the value of the leftmost bit of the original value, i.e. sign extension. In logical right shift, however, zeros are filled into the vacant bit-positions. In left shift, both arithmetic shift and logical shift fill zeros at the end of the bit value.

Bit rotation is basically circular shift of bits. In other words, while doing bit rotation, the two ends of a bit value seem to be joined. So when we perform left shift, the rightmost vacant bit will be filled with the bit that was the leftmost bit of the original value. For example, suppose we have binary number

$$N = 01101$$

If we perform right rotate, then we get

$$N' = 10110$$

because the rightmost 1 in N was dropped, and circulated onto the leftmost bit of N'. Left rotate works in similar way. Sometimes the circulated bit is stored also stored in a carry flag (another single bit), which is called *rotate through carry*.

2.12.1 Facts and Tricks

Properties of XOR Here are several properties of XOR that we should all be familar with:

1. Identity: $X \veebar 0 = X$
2. Bit negation: $X \veebar 1 = \ X$
3. Self-zero: $X \veebar X = 0$
4. Associativity: $(X \veebar Y) \veebar Z = X \veebar (Y \veebar Z)$
5. Communitivity: $X \veebar Y = Y \veebar X$

Swapping two numbers with XOR Usually when we swap two items, we need to use a temporary variable. But if those two items are integers, say `x` and `y`, you do not have to. You can use XOR as follows.

```
x = x ^ y
y = x ^ y
x = x ^ y
```

This is based on a simple fact:

$$(A \veebar B) \veebar A = (A \veebar A) \veebar B = 0 \veebar B = B$$

However, the compiler cannot tell what you are doing with these lines of code. So sometimes it may be better to just leave the optimization work to the compiler.

Find Odd in Evens Given a set of numbers where all elements occur even number of times, except for one number, find that number that only occur odd number of times.

Because of XOR's properties, including communitivity, associativity and self-zero, if we xor all numbers in the given set, the result will be exactly the odd occuring number!

Power of 2 Left shift is equivalent as multiplying by 2, as long as your number does not overflow. Right shift is equivalent as dividing by 2.

Max Without Comparison With bit operations, we can implement the max operation between two integers a, b without comparisons.

```
int max(int x, int y)
{
    int c = a - b;
    int k = (c >> 31) & 0x1;
    int max = a - k * c;
    return max;
}
```

The purpose of `k` in the `max` function is to check if the difference is negative. Run it manually to see how it works.

Other bit manipulation code snippets Some of these are from my solutions to CSE 351 lab1.

```
/* invert - Return x with the n bits that begin at position p
     inverted (i.e., turn 0 into 1 and vice versa) and the rest
     left unchanged. Consider the indices of x to begin with the
     low-order bit numbered as 0. Can assume that 0 <= n <= 31
     and 0 <= p <= 31
 *   Example: invert(0x80000000, 0, 1) = 0x80000001,
 *            invert(0x0000008e, 3, 3) = 0x000000b6,
 */
int invert(int x, int p, int n) {
  int mask = (1 << n) + ~0; // have n 1s at the end
  mask <<= p;
  return x ^ mask;
}

/*
 * sign - return 1 if positive, 0 if zero, and -1 if negative
 *  Examples: sign(130) = 1
 *            sign(-23) = -1
 */
int sign(int x) {
  // If x is negative, x >> 31 should give all 1s
  // which is essentailly -1
  int neg = x >> 31;
  // need to make x ^ 0 only 0 or 1
  int pos_or_zero = !(x >> 31) & !!(x ^ 0);
  return neg + pos_or_zero;
}
```

```
Another way to compute sign:
int flip(int bits):
    return 1 ^ bits

int sign(int a):
    """Return 1 if a is nonnegative, return 0 if otherwise.
    return flip((a >> 31) & 0x1);
```

2.13 Math

2.13.1 GCDs and Modulo

Theorem 2.16. *Euclid's Algorithm To compute gcd(n_1, n_2), produce a new pair of number that consists of* $\min(n_1, n_2)$ *and the difference* $|n_1 - n_2|$*; Keep doing this until the numbers in the pair are the same.*

My implementation of gcd:

```
int gcd(n1, n2):
  if (n1 != n2):
      int smaller = n1 < n2 ? n1 : n2
      int diff = abs(n2 - n1)
      return gcd(smaller, diff) # keep doing
  else:
      return n1
```

Modulo Arithmetic Let us look at some basic modulo arithmetic.

Definition 2.15 (Division Theorem)**.** For $a \in \mathbb{Z}, d \in \mathbb{Z}$ with $d > 0$, there exist unique integers q, r with $0 \leq r < d$, such that $a = dq + r$. We say $q = a$ div d, and $r = a \bmod d$.

Theorem 2.17 (Modulo Congruence)**.** *For $a, b, m \in \mathbb{Z}$ with $m > 0$, we have*

$$a \equiv b \ (mod\ m) \leftrightarrow m \mid (a - b)$$

Theorem 2.18 (Modulo Congruence Properties)**.** *For $a, b, c, dm \in \mathbb{Z}$ with $m > 0$, we have*

$$a \equiv b \ (mod\ m) \leftrightarrow a \ mod\ m = b \ mod\ m$$

$$(a \ mod\ m)(b \ mod\ m) \equiv ab \ (mod\ m)$$

$$(a \ mod \ m) + (b \ mod \ m) \equiv a + b \ (mod \ m)$$

Modulo can be used to obtain the single digit of a number. My Python code for solving a Google online assessment problem is as follows.

```
def nthdigit(n, x):
    # We use zero-based index for digits, where 0 is the
    # least significant digit of the given number x.
    return x / 10**n % 10

def replace2for1(x, d, N, i):
    # Replace the digit at i-th + (i+1)-th position in the number
        x with
    # the given digit d. N is the number of digits.
    firstPart = 0
    if i+2 < N:
        firstPart = x / 10**(i+2)
    secondPart = x % 10**(i)
    return firstPart * 10**(i+1) + d * 10**i + secondPart
```

2.13.2 Prime numbers

My code for checking if a number is prime:

```
//1. Loop from 2 to sqrt(n)
//2. skip all numbers that are even, and
//  numbers that are not multiples of
//  6+-1
bool isPrime(long n)
{
  if (n < 2) return false;
  if (n < 4) return true;
  if (n % 2 == 0) return false;
  if (n % 6 != 1 && n % 6 != 5) return false;

  for (int i = 2; i <= sqrt(n); i++)
    {
      if (n % i == 0) return false;
    }
  return true;
}
```

2.13.3 Palindromes

Although palindromic numbers are most often considered in the decimal system, the concept of palindromicity can be applied to the natural numbers in any numeral system. Formal definition of Palindromicity:

Definition 2.16. Consider a number $n > 0$ in base $b \geq 2$, where it is written in standard notation with k digits a_i as:

$$n = \sum_{i=0}^{k-1} (a_i b^i)$$

with $0 \leq a_i < b$ for all i and $a_k \neq 0$. Then n is palindromic if and only if $a_i = a_{k-i}$ for all i. Zero is written 0 in any base and is also palindromic by definition.

My C++ code for checking if a base-10 number is palindrome:

```
 vector<int> get_digits(int n)
{
  vector<int> result;
  int d = n % 10;
  n /= 10;
  result.push_back(d);
  while (n != 0)
    {
      d = n % 10;
      n /= 10;
      result.push_back(d);
    }
  return result;
}

// base = 10
bool is_palindrome(int n)
{
  vector<int> digits = get_digits(n);
  int size = digits.size();
  for (int i = 0; i < size / 2; i++)
    {
      if (digits.at(i) != digits.at(size-1-i))
        {
          return false;
        }
```

```
    }
  return true;
}
```

2.13.4 Combination and Permutation

Combination n choose k is defined as follows:

$$C(n,k) = \binom{n}{k} = \frac{n(n-1)\cdots(n-k+1)}{k(k-1)\cdots 1}$$

One property of combination:

$$\binom{n}{k} = \binom{n}{n-k}$$

Permutation n choose k where order matters can be expressed as follows:

$$P(n,k) = \underbrace{n\cdot(n-1)\cdot(n-2)\cdots(n-k+1)}_{k \text{ factors}} = \frac{n!}{(n-k)!}$$

$$C(n,k) = \frac{P(n,k)}{P(k,k)} = \frac{P(n,k)}{k!} = \frac{n!}{(n-k)!k!}$$

2.13.5 Series

The following description is from Wikipedia.

Arithmetic Series If the initial term of an *arithmetic progression* is a_1 and the common difference of successive members is d, then the nth term of the sequence a_n is given by:

$$a_n = a_1 + (n-1)d$$

and in general

$$a_n = a_m + (n-m)d$$

A finite portion of an arithmetic progression is called a finite arithmetic progression and sometimes just called an arithmetic progression. The sum of a finite arithmetic progression is called an *arithmetic series*,

$$S_n = \frac{n(a_1+a_n)}{2} = \frac{n}{2}[2a_1 + (n-1)d]$$

Geometric Series The n-th term of a geometric sequence with initial value a and common ratio r is given by

$$a_n = ar^{n-1}$$

Such a geometric sequence also follows the recursive relation

$$a_n = ra_{n-1}$$

for every integer $n \geq 1$. Generally, to check whether a given sequence is geometric, one simply checks whether successive entries in the sequence all have the same ratio. The sum of a finite geometric progression is called an *geometric series*. If one were to begin the sum not from $k = 1$, but from a different value, say m, then

$$\sum_{k=m}^{n} ar^k = \frac{a(r^m - r^{n+1})}{1-r}$$

.

Power Series In mathematics, a power series (in one variable) is an infinite series of the form

$$\sum_{n=0}^{\infty} a_n (x-c)^n = a_0 + a_1(x-c)^1 + a_2(x-c)^2 + \dots$$

where an represents the coefficient of the nth term and c is a constant. This series usually arises as the Taylor series of some known function. The geometric series formula

$$\frac{1}{1-x} = \sum_{n=0}^{\infty} x^n = 1 + x + x^2 + x^3 + \cdots,$$

which is valid for $|x| < 1$, is one of the most important examples of a power series, as are the exponential function formula

$$e^x = \sum_{n=0}^{\infty} \frac{x^n}{n!} = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \cdots,$$

and the sine formula

$$\sin(x) = \sum_{n=0}^{\infty} \frac{(-1)^n x^{2n+1}}{(2n+1)!} = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \cdots,$$

valid for all real x.

These power series are also examples of *Taylor series*. The Taylor series of a real or complex-valued function $f(x)$ that is infinitely differentiable at a real or complex number a is the power series

$$f(a) + \frac{f'(a)}{1!}(x-a) + \frac{f''(a)}{2!}(x-a)^2 + \frac{f'''(a)}{3!}(x-a)^3 + \cdots .$$

which can be written in the more compact sigma notation as

$$\sum_{n=0}^{\infty} \frac{f^{(n)}(a)}{n!}\,(x-a)^n$$

where $n!$ denotes the factorial of n and $f^{(n)}(a)$ denotes the nth derivative of f evaluated at the point a. The derivative of order zero of f is defined to be f itself and $(x-a)^0$ and 0! are both defined to be 1.

2.14 Concurrency

2.14.1 Threads & Processes

Typically, threads share the same address space, and processes have independent and private address spaces. Below is the context switch assembly code in xv6.

```
/* Switch from current_thread to next_thread. Make next_thread
 * the current_thread, and set next_thread to 0.
 * Use eax as a temporary register; it is caller saved.
 * (Note, # means dollar sign.)
 */
   .globl thread_switch
thread_switch:
   /* YOUR CODE HERE */
   pushal       /* Push all registers */

   /* save current sp */
   movl  current_thread, %eax
   movl  %esp, (%eax)

   /* SAVE CURRENT SP TO SOMEWHERE POINTED BY current_thread */
   movl  next_thread, %eax
   /* Set current_thread to next_thread */
   movl  %eax, current_thread
   movl  #0, next_thread
```

```
/* load saved sp for next_thread*/
movl  (%eax), %esp
/* LOAD next_thread's SP */
popal
/* pop return address from stack */
ret
```

What is done here is pretty straightforward. The current thread stores all of its registers to the stack for this thread, and then loads the stack pointer of the next thread, and then pop all of next thread's registers. So a thread is basically a struct that stores a set of register values, and when a thread is running, its register values are loaded into the real registers.

For processes, when a process is initialized, it is given a private address space, including its own page directory and page tables, where each entries' permissions are configured appropriately. Depending on the multitasking scheme, e.g. preemptive multitasking, a process either chooses to give up the CPU autonomously, or the kernel has a time tick interrupt that traps the current process every 10ms, for example, and then the kernel can give the CPU to other processes. Switching between processes requires a *scheduler*. Sharing or communication between processes is achievable through inter-process communication (IPC).

2.14.2 Locks

Locks help us write code that uses concurrency by avoiding multiple processors modifying the same resource at the same time. A lock has two operations, acquire and release. When one process acquire a lock, other processes have to wait until the lock is released. Wherever code accesses a section of code which is shared, always lock.[26]

One problem with locking is *deadlock*. It happens when two processes acquire the locks for two independent resources, and wait for each other to release their lock in order to grab the other lock and proceed. This problem can be avoided by one simple rule: Always acquire the locks in a predetermined order.

Lock implementation with atomic instruction:

```
struct lock { int locked; };
```

[26] From Stackoverflow: http://stackoverflow.com/questions/8720735/when-to-use-the-lock-thread-in-c, user: `OmegaMan`

```
void acquire(struct lock *l) {
  for (;;) {
    if (atomic_exchange(&l->locked, 1) == 0)
      return;
  }
}
void release(struct lock *l) {
  l->locked = 0;
}
```

2.15 System design

2.15.1 Specification

Abstraction Functions An abstraction function AF maps the concrete representation of an abstract data type to the abstract value that the ADT represents. Formally,

$$AF : R \Rightarrow A$$

where R is the set of rep (representation) values, and A is the set of abstract values.

Representation Invariant A representation invariant RI is a condition that must be true over all valid concrete representations of a class. It maps the concrete representation to a Boolean (true or false). Formally,

$$RI : R \Rightarrow \text{boolean}$$

where R is the set of representation values. The representation invariant describes whether a representation value is a well-formed instance of the type.

Below is an example AF and RI that I wrote, from the RatTerm (rational term in a polynomial) class in CSE 311.

```
// Abstraction Function:
For a given RatTerm t, "coefficient of t" is synonymous with
    t.coeff, and, likewise, "exponent of t" is synonymous with
    t.expt. All RatTerms with a zero coefficient are represented
    by the zero RatTerm, z, which has zero for its coefficient
    AND exponent.
```

```
// Representation Invariant:
coeff != null
coeff.equals(RatNum.ZERO) ==> expt == 0
```

Weak vs. Strong precondition *Preconditions* are properties that must be true when the method is called. *Postconditions* are properties that a method guarantees will hold when the method exits. If precondition P_1 implies precondition P_2, then:

- P_1 is stronger than P_2.
- P_2 is weaker than P_1.

This means: Whenever P_1 (stronger) holds, P_2 (weaker) *must* hold. It is *more difficult* to satisfy the stronger precondition P_1. A strong precondition is a subset of the weak, i.e. $P_1 \subseteq P_2$.

The *weakest precondition* is the most lenient assumptions. For every statement S and post-condition Q, there exists a unique weakest precondition $WP(S, Q)$. We can produce this weakest precondition using backwards reasoning.

Weak vs. Strong specification Similar to weak/strong precondition, weak specification gives more freedom (requires less). If S_1 is weaker than S_2, then for any implementation M,

$$M \text{ satisfies } S_2 \Rightarrow M \text{ satisfies } S_1$$

Some specifications may be incomparable.

2.15.2 Subtyping and Subclasses

Definition 2.17 (Subtyping)**.** if B is a *subtype* of A, then the specification of A works correctly even if given a B.

Subtypes are substitutable for supertypes. If B is a true subtype of A, then B's specification must not be weaker than A's.

On the other hand, *subclass* is an implementation notion used for eliminating repeated code (used in inheritance).

Python Class Inheritance According to Python Documentation[27]:

[27] Found here `https://docs.python.org/2/tutorial/classes.html`

Python classes provide *all* the standard features of Object Oriented Programming: the class inheritance mechanism allows multiple base classes, a derived class can override any methods of its base class or classes, and a method can call the method of a base class with the same name.

Python has several special methods that may be overrided ("customizable").

```
class MyClass:
    def __init__(self):
        # The instantiation operation
    def __del__(self):
        # Called when the instance is about to be destroyed
    def __repr__(self):
        # Called by repr() built-in function. Official string
            representation.
    def __str__(self):
        # Called by str() built-in function and print(). Informal
            string representation.
    def __eq__(self, other)
    def __ne__(self, other)
    def __lt__(self, other)
    def __le__(self, other)
    def __gt__(self, other)
    def __ge__(self, other)
        # These are comparison methods, called for comparison
            operators. Note: x==y is True does not imply that
            x!=y is False (no dependence).
    def __cmp__(self, other)
        # Called by comparison operations. Returns a negative
            integer if self < other, zero if self == other, and a
            positive integer if self > other.
    def __hash__(self, other)
        # Called by comparison operations.
```

The way inheirtance works in Python is just like in Java. Subclasses can override superclass methods and fields. In the override version, the subclass can call the superclass's function by simply referring that function as an attribute with a dot.

According to the documentation, Python supports class-private members in a limited way, to avoid name clashes with subclasses' names, by using *name mangling*. Any identifier of the form `__spam` (at least two underscores, at most one trailing underscore) is textually replaced with

`_classname__spam`.

2.15.3 Design Patterns

Software design patterns are solutions to common problems in software design, in the forma of a template. One who knows design patterns can save more time, and be able to compare and contrast different design options, and use the most suitable one.

Design patterns can be grouped into four categories, depending on the kind of problems that they solve.

Creational Patterns These patterns deal with object creation in different situations.

Abstract factory: Provide an interface for creating families of related or dependent objects without specifying their concrete classes.

Builder: Separate the construction of a complex object from its representation, allowing the same construction process to create various representations.

Factory method: Define an interface for creating a single object, but let subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses

Lazy initialization: Tactic of delaying the creation of an object, the calculation of a value, or some other expensive process until the first time it is needed.

Object pool: Avoid expensive acquisition and release of resources by recycling objects that are no longer in use. Can be considered a generalisation of connection pool and thread pool patterns.

Prototype: Specify the kinds of objects to create using a prototypical instance, and create new objects from the 'skeleton' of an existing object, thus boosting performance and keeping memory footprints to a minimum.

Singleton: Ensure a class has only one instance, and provide a global point of access to it. In Python, since there is not really private constructors, the role of *modules* can be considered to be similar as singleton.

Structural Patterns These patterns deal with realizing relationships between entities.

Adapter/Wrapper: Convert the interface of a class into another interface clients expect. An adapter lets classes work together that could not otherwise because of incompatible interfaces. The enterprise integration pattern equivalent is the translator.

Bridge: Decouple an abstraction from its implementation allowing the two to vary independently.

Composite: Compose objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly.

Decorator: Attach additional responsibilities to an object dynamically keeping the same interface. Decorators provide a flexible alternative to subclassing for extending functionality.

Flyweight: Use sharing to support large numbers of similar objects efficiently.

Behavioral Patterns These patterns deal with *communication* patterns between objects.

Chain of responsibility: Avoid coupling the sender of a request to its receiver by giving more than one object a chance to handle the request. Chain the receiving objects and pass the request along the chain until an object handles it.

Interpreter: Given a language, define a representation for its grammar along with an interpreter that uses the representation to interpret sentences in the language.

Command: Encapsulate a request as an object, thereby allowing for the parameterization of clients with different requests, and the queuing or logging of requests. It also allows for the support of undoable operations.

Iterator: Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation.

Observer or Publish/Subscribe: Define a one-to-many dependency between objects where a state change in one object results in all its dependents being notified and updated automatically.

Template method: Define the skeleton of an algorithm in an operation, deferring some steps to subclasses. Template method lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure.

Visitor: Represent an operation to be performed on the elements of an object structure. Visitor lets a new operation be defined without changing the classes of the elements on which it operates.

Concurrency Patterns These patterns deal with the multi-threaded programming paradigm.

Active Object: Decouples method execution from method invocation that reside in their own thread of control. The goal is to introduce concurrency, by using asynchronous method invocation and a scheduler for handling requests.

Balking: Only execute an action on an object when the object is in a particular state.

Double-checked locking: Reduce the overhead of acquiring a lock by first testing the locking criterion (the 'lock hint') in an unsafe manner; only if that succeeds does the actual locking logic proceed. Can be unsafe when implemented in some language/hardware combinations. It can therefore sometimes be considered an anti-pattern.

Monitor object: An object whose methods are subject to mutual exclusion, thus preventing multiple objects from erroneously trying to use it at the same time.

Reactor: A reactor object provides an asynchronous interface to resources that must be handled synchronously.

Scheduler: Explicitly control when threads may execute single-threaded code.

Thread-specific storage: Static or "global" memory local to a thread.

Lock: One thread puts a "lock" on a resource, preventing other threads from accessing or modifying it.

Read-write lock: Allows concurrent read access to an object, but requires exclusive access for write operations.

Other Patterns There are other useful patterns, such as the MVC pattern.

Model-view-controller (MVC): Model–view–controller (MVC) is a software design pattern for implementing user interfaces on computers. The **model** directly manages the data, logic, and rules of the application. A **view** can be any output representation of information, such as a chart or a diagram. Multiple views of the same information are possible, such as a bar chart for management and a tabular view for accountants. The third part, the **controller**, accepts input and converts it to commands for the model or view.

Active Record: The active record pattern is an approach to accessing data in a database. A database table or view is wrapped into a class. Thus, an object instance is tied to a single row in the table. After creation of an object, a new row is added to the table upon save. Any object loaded gets its information from the database. When an object is updated, the corresponding row in the table is also updated. The wrap-

per class implements accessor methods or properties for each column in the table or view. Think about Rails.

Data access object: a data access object (DAO) is an object that provides an abstract interface to some type of database or other persistence mechanism. By mapping application calls to the persistence layer, the DAO provides some specific data operations without exposing details of the database.

2.15.4 Architecture

Software architecture is the fundamental structures of a software system, the discipline of creating such structures, and the documentation of these structures[28].

Agile Development This is a architecture design method. Agile software development describes a set of principles for software development under which requirements and solutions evolve through the collaborative effort of self-organizing cross-functional teams. It advocates adaptive planning, evolutionary development, early delivery, and continuous improvement, and it encourages rapid and flexible response to change. hese principles support the definition and continuing evolution of many software development methods.

Most agile development methods break product development work into small increments that minimize the amount of up-front planning and design. Iterations are short time frames (timeboxes) that typically last from one to four weeks. Each iteration involves a cross-functional team working in all functions: planning, analysis, design, coding, unit testing, and acceptance testing. At the end of the iteration a working product is demonstrated to stakeholders. This minimizes overall risk and allows the product to adapt to changes quickly.

2.15.5 Testing

Defects and Failures Software faults occur through the following processes. A programmer makes an error (mistake), which results in a defect (fault, bug) in the software source code. If this defect is executed, in certain situations the system will produce wrong results, causing a failure.

[28] From Wikipedia.

Testing Methods There are several well-known testing methods, as discussed below.

- *Static vs. Dynamic testing:* There are many approaches available in software testing. Reviews, walkthroughs, or inspections are referred to as *static* testing, whereas actually executing programmed code with a given set of test cases is referred to as *dynamic* testing.
- *Black-box testing & White-box testing* **Black-box** testing treats the software as a "black box", examining functionality without any knowledge of internal implementation, without seeing the source code. The testers are only aware of what the software is supposed to do, not how it does it. **White-box** testing (also known as clear box testing, glass box testing, transparent box testing and structural testing, by seeing the source code) tests internal structures or workings of a program, as opposed to the functionality exposed to the end-user.

Testing Levels There are generally four recognized levels of tests: *unit testing, integration testing, component interface testing, and system testing.* Tests are frequently grouped by where they are added in the software development process, or by the level of specificity of the test.

- *Unit testing:* These types of tests are usually written by developers as they work on code (white-box style), to ensure that the specific function is working as expected. One function might have multiple tests, to catch corner cases or other branches in the code. Unit testing alone cannot verify the functionality of a piece of software, but rather is used to ensure that the building blocks of the software work independently from each other.
- *Integration testing:* Integration testing is any type of software testing that seeks to verify the interfaces between components against a software design. Integration testing works to expose defects in the interfaces and interaction between integrated components (modules). Progressively larger groups of tested software components corresponding to elements of the architectural design are integrated and tested until the software works as a system.
- *Component interface testing:* The practice of component interface testing can be used to check the handling of data passed between various units, or subsystem components, beyond full integration testing between those units.

- *System testing:* System testing, or end-to-end testing, tests a completely integrated system to verify that the system meets its requirements. For example, a system test might involve testing a logon interface, then creating and editing an entry, plus sending or printing results, followed by summary processing or deletion (or archiving) of entries, then logoff.

Regression Testing A type of software testing that verifies that software *previously* developed and tested *still performs correctly* even after it was changed or interfaced with other software. Changes may include software enhancements, patches, configuration changes, etc. During regression testing, new software bugs or regressions may be uncovered.

Common methods of regression testing include re-running previously completed tests and checking whether program behavior has changed and whether previously fixed faults have re-emerged. Regression testing can be performed to test a system efficiently by systematically selecting the appropriate minimum set of tests needed to adequately cover a particular change.

Contrast with non-regression testing (usually validation-test for a new issue), which aims to verify whether, after introducing or updating a given software application, the change has had the intended effect.

3 Flagship Problems

It turns out that I do not have a lot of time to complete many problems and record the solutions here. I will include the description several important problems, and solve them after I print this out. I will update the solutions hopefully eventually. Refer to 3.9 for these unsolved problems.

3.1 Arrays

Missing Ranges (Source. Leetcode 163) Given a sorted integer array where the range of elements are in the inclusive range `[lower, upper]`, return its missing ranges.

For example, given `[0, 1, 3, 50, 75]`, `lower = 0` and `upper = 99`, return `["2", "4->49", "51->74", "76->99"]`.

My code:

```
class Solution(object):
    def __getRange(self, a, b):
        if b - a > 1:
            if b - a == 2:
                return str(a+1)
            else:
                return str(a+1) + "->" + str(b-1)
        else:
            return None

    def findMissingRanges(self, nums, lower, upper):
        result = []
        upper += 1
        lower -= 1
        lastOne = lower
        if len(nums) > 0:
            lastOne = nums[len(nums)-1]

        for i, n in enumerate(nums):
            rg = None
            if i == 0:
                # First number
                rg = self.__getRange(lower, nums[0])
            else:
                rg = self.__getRange(nums[i-1], nums[i])
            if rg is not None:
```

```
            result.append(rg)
        # Last number
        rg = self.__getRange(lastOne, upper)
        if rg is not None:
            result.append(rg)
        return result
```

Merge Intervals Given a collection of intervals, merge all overlapping intervals.

For example,

```
Given [1,3],[2,6],[8,10],[15,18],
return [1,6],[8,10],[15,18].
```

My code:

```
class Solution(object):
    def merge(self, intervals):
        if len(intervals) <= 1:
            return intervals

        intervals = sorted(intervals, key=lambda x: x.start)
        result = [intervals[0]]
        i = 0
        j = 1
        while j < len(intervals):
            if result[i].start <= intervals[j].start and
                intervals[j].start <= result[i].end:
                result[i] = Interval(result[i].start,
                    max(result[i].end, intervals[j].end))
            else:
                result.append(intervals[j])
                i += 1
            j += 1
        return result
```

Summary Ranges (Source. Leetcode 228) Given a sorted integer array without duplicates, return the summary of its ranges. For example,

```
Given [0,1,2,4,5,7],
Return ["0->2","4->5","7"].
```

My code:

```
class Solution(object):
    def summaryRanges(self, nums):
        if len(nums) == 0:
            return []

        ranges = []
        start = nums[0]
        prev = start
        j = 1
        while j <= len(nums):
            cur = nums[-1] + 2
            if j < len(nums):
                cur = nums[j]
            if cur - prev > 1:
                if start == prev:
                    ranges.append(str(start))
                else:
                    ranges.append(str(start) + "->" + str(prev))

                start = cur
            prev = cur
            j += 1
        return ranges
```

3.2 Strings

Longest Absolute Image File Path (Source. Leetcode 388) Suppose we abstract our file system by a string in the following manner. The string

```
dir\n\tsubdir1\n\t\tfile1.ext\n\t\tsubsubdir1\n\tsubdir2\n\t\tsubsubdir2\n\t\t\tfile2.ext
```

represents:

```
dir
    subdir1
        file1.ext
        subsubdir1
    subdir2
        subsubdir2
            file2.png
```

We are interested in finding the longest (number of characters) absolute path to a file within our file system. For example, in the example above, the longest absolute path to an image file is

`dir/subdir2/subsubdir2/file2.png`

and its length is 32 (not including the double quotes).

Given a string S representing the file system in the above format, return the length of the longest absolute path to file with image extension (`png, jpg, bmp`) in the abstracted file system. If there is no file in the system, return 0.

Idea: Use a hash table to keep track of each level and the most recent absolute path size at this level, as well as the maximum absolute path size at this level. See code next page.

```
def solution(S):
    S += '\n'
    # Key: level
    # Value: a tuple (most recent absolute path size at this
        level, maximum absolute path size at this level)
    dict = {}
    dict[-1] = (0,0)
    curLevelCount = 0
    curFname = ""
    for ch in S:
        if ch != '\n' and ch != ' ':
            # Append new character if it is not a special character
            curFname += ch
        elif ch == '\n':
            curFnamePathSize = dict[curLevelCount-1][0] +
                len(curFname)
            if curLevelCount != 0:
                # For the slash
                curFnamePathSize += 1

            pathSizeWeCare = curFnamePathSize - len(curFname)
            if not (curFname.endswith(".jpeg") or
                curFname.endswith(".png") or
                curFname.endswith(".gif")):
                pathSizeWeCare = 0

            if curLevelCount in dict:
                prevMax = dict[curLevelCount][1]
                dict[curLevelCount] = (curFnamePathSize,
                    max(prevMax, pathSizeWeCare))
            else:
                dict[curLevelCount] = (curFnamePathSize,
                    pathSizeWeCare)
            curFname = ""
            curLevelCount = 0
        else:
            curLevelCount +=1
    maxPathSize = 0
    for level in dict:
        if level >= 0:
            maxPathSize = max(maxPathSize, dict[level][1])

    return maxPathSize
```

Repeated Substring Pattern (Source. Leetcode 459) Given a non-empty string check if it can be constructed by *taking a substring of it* and appending multiple copies of the substring together. You may assume the given string consists of lowercase English letters only and its length will not exceed 10000. **Difficulty: Easy.** Examples:

```
Input: "abab"
Output: True
--
Input: "abcabcabc"
Output: False
```

Idea 1: There is a Greedy way to solve this by using the π table in the KMP algorithm (refer to 2.8.2 for more description.) Once we have the π table of the given string P, then P is a repetition of its substring if:

- $\pi[|P|-1] \geq (|P|-1)/2$. Basically the longest prefix equal to suffix must end beyond half of the string.
- The length of the given string, $|P|$, must be divisible by the pattern, given by $P_0 \cdots P_\pi[|P|-1]$

```
class Solution(object):
    def kmpTable(self, p):
        ... Check this code in appendix (5.2).

    def repeatedSubstringPattern(self, s):
        table = self.kmpTable(s)
        if table[len(s)-1] < (len(s)-1)/2:
            return False
        pattern = s[table[len(s)-1]+1:]
        if len(s) % len(pattern) != 0:
            return False
        return True
```

Idea 2: There is a trick. Make a new string Q equal to the concatenation of two given string P, so $Q = P+P$. Then, check if P is a substring of the substring of Q, removing the front and last character, i.e. $Q_1 \cdots Q_{|Q|-1}$. Code:

```
def repeatedSubstringPattern(s):
    q = s + s
    return q[1:len(q)-1].find(s) != -1
```

Valid Parenthesis Given a string containing just the characters '(', ')', '', '', '[' and ']', determine if the input string is valid.

The brackets must close in the correct order, "()" and "()[]" are all valid but "(]" and "([)]" are not.

My code, using stack.

```
class Solution(object):
    def isValid(self, s):
        """
        :type s: str
        :rtype: bool
        """
        if len(s) % 2 != 0:
            return False
        pstart = {'(': ')', '{': '}', '[': ']'}
        if s[0] not in pstart:
            return False

        recorder = []
        for ch in s:
            if ch in pstart:
                recorder.append(ch)
            else:
                rch = recorder.pop()
                if pstart[rch] != ch:
                    return False
        return len(recorder) == 0
```

3.3 Permutation

There are several classic problems related to permutations. Some involve strings, and some are just array of numbers.

Generate Parenthesis (Source. Leetcode 22) Given n pairs of parentheses, write a function to generate all combinations of well-formed parentheses.

For example, given n = 3, a solution set is:

```
[
  "((()))",
  "(()())",
```

```
  "(())()",
  "()(())",
  "()()()"
]
```

A bit strangely, I used iterative method. It was more intuitive for me when I was doing this problem. My code:

```
class Solution(object):
    def generateParenthesis(self, n):
        S = {}
        solution = []
        S['('] = (1,0)
        while True:
            if len(S.keys()) == 0:
                break
            str, tup = S.popitem()
            o, c = tup
            if o == n:
                if c == n:
                    solution.append(str)
                    continue
                else:
                    S[str+')'] = (o, c+1)
            elif o == c:
                S[str+'('] = (o+1, c)
            else:
                S[str+'('] = (o+1, c)
                S[str+')'] = (o, c+1)
        return solution
```

Palindrome Permutations Given a string, determine if a permutation of the string could form a palindrome. For example, `"code" -> False, "aab" -> True, "carerac" -> True.`.

This problem is relatively easy. Count the frequency of each character. My code:

```
class Solution(object):
    def canPermutePalindrome(self, s):
        freq = {}
        for c in s:
```

```
            if c in freq:
                freq[c] += 1
            else:
                freq[c] = 1
        if len(s) % 2 == 0:
            for c in freq:
                if freq[c] % 2 != 0:
                    return False
            return True
        else:
            count = 0
            for c in freq:
                if freq[c] % 2 != 0:
                    count += 1
                    if count > 1:
                        return False
            return True
```

Next Permutation (Source. Leetcode 31) Implement next permutation, which rearranges numbers into the lexicographically next greater permutation of numbers.

If such arrangement is not possible, it must rearrange it as the lowest possible order (i.e., sorted in ascending order).

The replacement must be in-place, do not allocate extra memory.

Here are some examples. Inputs are in the left-hand column and its corresponding outputs are in the right-hand column.

```
1,2,3 -> 1,3,2
3,2,1 -> 1,2,3
1,1,5 -> 1,5,1
```

My code:

```
class Solution(object):
    def _swap(self, p, a, b):
        t = p[a]
        p[a] = p[b]
        p[b] = t

    def nextPermutation(self, p):
        if len(p) < 2:
            return
```

```
if len(p) == 2:
    self._swap(p, 0, 1)
    return

i = len(p)-1
while i > 0 and p[i-1] >= p[i]:
    i -= 1
# Want to increase the number at p[i-1]. That number
# should be the smallest one (but >= p[i] in the range
# i to len(p)-1
if i > 0:
    smallest = p[i]
    smallestIndex = i
    for j in range(i, len(p)):
        if p[j] > p[i-1] and p[j] <= smallest:
            smallest = p[j]
            smallestIndex = j
    self._swap(p, i-1, smallestIndex)
# Reverse [i to len(p)-1)].
for j in range(i, i+(len(p)-i)/2):
    self._swap(p, j, len(p)-1-(j-i))
```

3.4 Trees

Binary Tree Longest Consecutive Sequence (Source. Leetcode 298) Given a binary tree, find the length of the longest consecutive sequence path.

The path refers to any sequence of nodes from some starting node to any node in the tree along the parent-child connections. The longest consecutive path need to be from parent to child (cannot be the reverse). Example:

```
  1
   \
    3
   / \
  2   4
       \
        5

Longest consecutive sequence path is 3-4-5, so return 3
```

My code:

```
# Definition for a binary tree node.
# class TreeNode(object):
#     def __init__(self, x):
#         self.val = x
#         self.left = None
#         self.right = None
class Solution(object):
    def longestConsecutive(self, root):
        if root is None:
            return 0
        return self.__longest(root, None, 0) + 1

    def __longest(self, root, parent, count):
        if parent is not None:
            if root.val - parent.val == 1:
                count += 1
            else:
                count = 0
        countLeft = 0
        countRight = 0
        if root.left is not None:
            countLeft = self.__longest(root.left, root, count)
        if root.right is not None:
            countRight = self.__longest(root.right, root, count)
        return max(count, countLeft, countRight)
```

3.5 Graphs

Number of Islands (Source. 200) Given a 2d grid map of '1's (land) and '0's (water), count the number of islands. An island is surrounded by water and is formed by connecting adjacent lands horizontally or vertically. You may assume all four edges of the grid are all surrounded by water. Example:

```
11000
11000
00100
00011

Answer: 3
```

My code:

```
class Solution(object):
    def __getRange(self, a, b):
        if b - a > 1:
            if b - a == 2:
                return str(a+1)
            else:
                return str(a+1) + "->" + str(b-1)
        else:
            return None

    def findMissingRanges(self, nums, lower, upper):
        result = []
        upper += 1
        lower -= 1
        lastOne = lower
        if len(nums) > 0:
            lastOne = nums[len(nums)-1]

        for i, n in enumerate(nums):
            rg = None
            if i == 0:
                # First number
                rg = self.__getRange(lower, nums[0])
            else:
                rg = self.__getRange(nums[i-1], nums[i])
            if rg is not None:
                result.append(rg)
        # Last number
        rg = self.__getRange(lastOne, upper)
        if rg is not None:
            result.append(rg)
        return result
```

3.6 Divide and Conquer

Median of Two Sorted Arrays (Source. Leetcode 4) There are two sorted arrays `nums1` and `nums2` of size m and n respectively.

Find the median of the two sorted arrays. The overall run time complexity should be $O(log(m+n))$.

This is a hard problem. I have two blog posts about two different approaches to *find k-th smallest elements in two sorted arrays*:

- Recursive $O(log(mn))$:

  http://zkytony.blogspot.com/2016/09/find-kth-smallest-element-in-two-sorted.html

- Recursive $O(logk)$):

  http://zkytony.blogspot.com/2016/09/find-kth-smallest-element-in-two-sorted_19.html

My code:

```
class Solution(object):
    def findMedianSortedArrays(self, nums1, nums2):
        n = len(nums1)
        m = len(nums2)
        if (n+m) % 2 == 0:
            m0 = self.kth(nums1, nums2, (n+m)/2-1)
            m1 = self.kth(nums1, nums2, (n+m)/2)
            return (m0 + m1) / 2.0
        else:
            return self.kth(nums1, nums2, (n+m)/2)

    def kth(self, A, B, k):
        if len(A) > len(B):
            A, B = (B, A)
        if not A:
            return B[k]
        if k == len(A) + len(B) - 1:
            return max(A[-1], B[-1])

        i = min(len(A)-1, k/2)
        j = min(len(B)-1, k-i)

        if A[i] > B[j]:
            return self.kth(A[:i], B[j:], i)
        else:
            return self.kth(A[i:], B[:j], j)
```

3.7 Dynamic Programming

Paint Fence There is a fence with n posts, each post can be painted with one of the k colors.

You have to paint all the posts such that no more than two adjacent fence posts have the same color.

Return the total number of ways you can paint the fence. Note that n and k are non-negative integers.

My code[29]:

```
class Solution(object):
    def numWays(self, n, k):
        if n == 0:
            return 0
        if n == 1:
            return k
        # Now n >= 2.
        # Initialize same and diff as if n == 2
        same = k
        diff = k*(k-1)
        for i in range(3,n+1):
            r_prev = same + diff # r(i-1)
            same = diff         # same(i)=diff(i-1)
            diff = r_prev*(k-1)
        return same + diff
```

3.8 Miscellaneous

Range Sum Query 2D - Mutable (Source. Leetcode 308) Given a 2D matrix matrix, find the sum of the elements inside the rectangle defined by its upper left corner (row1, col1) and lower right corner (row2, col2). **Difficulty: Hard.** Example:

```
Given matrix = [
  [3, 0, 1, 4, 2],
  [5, 6, 3, 2, 1],
  [1, 2, 0, 1, 5],
  [4, 1, 0, 1, 7],
  [1, 0, 3, 0, 5]
]

sumRegion(2, 1, 4, 3) -> 8
```

[29] I have a blog post about this problem: `http://zkytony.blogspot.com/2016/09/paint-fence.html`

```
//The above rectangle (with the red border) is defined by (row1,
    col1) = (2, 1) and (row2, col2) = (4, 3), which contains sum
    = 8.

update(3, 2, 2)
sumRegion(2, 1, 4, 3) -> 10
```

Idea: Use binary indexed tree. This kind of tree is built to solve problems like this. See 2.1.15 for more detailed explanation of how it works. Code is below. The formula to compute parent index of an index `i`, `parent(i) = i+i&(-i)`, not only works for 1D array, but also for the row and column index for 2D array.

```
class BinaryIndexTree():
    def __init__(self, matrix):
        if not matrix:
            return
        self.num_rows = len(matrix)+1
        self.num_cols = len(matrix[0])+1 if len(matrix) > 0 else 0
        self.matrix = [[0 for x in range(self.num_cols-1)] for y
            in range(self.num_rows-1)]
        self.tree = [[0 for x in range(self.num_cols)] for y in
            range(self.num_rows)]
        for r in range(self.num_rows-1):
            for c in range(self.num_cols-1):
                self.update(r, c, matrix[r][c])

    def update(self, row, col, val):
        i = row + 1
        while i < self.num_rows:
            j = col + 1
            while j < self.num_cols:
                self.tree[i][j] += val - self.matrix[row][col]
                j += ((~j+1) & j)
            i += ((~i+1) & i)
        self.matrix[row][col] = val

    def sum(self, row, col):
        result = 0
        i = row + 1
        while i > 0:
            j = col + 1
            while j > 0:
                result += self.tree[i][j]
```

```
                j -= ((~j+1) & j)
            i -= ((~i+1) & i)
        return result

class NumMatrix(object):
    def __init__(self, matrix):
        self.BIT = BinaryIndexTree(matrix)

    def update(self, row, col, val):
        self.BIT.update(row, col, val)

    def sumRegion(self, row1, col1, row2, col2):
        return self.BIT.sum(row2, col2) \
            - self.BIT.sum(row2, col1-1) \
            - self.BIT.sum(row1-1, col2) \
            + self.BIT.sum(row1-1, col1-1)
```

3.9 Unsolved

Longest Substring With At Most k Distinct Characters (Source. Leetcode 340) Given a string, find the length of the longest substring T that contains at most k distinct characters.

For example, given `s = 'eceba'` and `k = 2`, T is `'ece'` which its length is 3.

I solved this problem, but my code is very hard to understand. So not included here. Treat this problem as unsolved.

Sentence Screen Fitting Given a `rows x cols` screen and a sentence represented by a list of non-empty words, find how many times the given sentence can be fitted on the screen. Rules:

1. A word cannot be split into two lines.
2. The order of words in the sentence must remain unchanged.
3. Two consecutive words in a line must be separated by a single space.
4. Total words in the sentence won't exceed 100.
5. Length of each word is greater than 0 and won't exceed 10. $1 \leq$ rows, cols $\leq$ 20,000.

Example:

```
Input:
rows = 3, cols = 6, sentence = ["a", "bcd", "e"]

Output:
2

Explanation:
a-bcd-
e-a---
bcd-e-

The character '-' signifies an empty space on the screen.
```

Trapping Rain Water (Source. Leetcode 42) Given n non-negative integers representing an elevation map where the width of each bar is 1, compute how much water it is able to trap after raining.

For example, Given `[0,1,0,2,1,0,1,3,2,1,2,1]`, return 6. Visualize it yourself.

Trapping Rain Water 2D (Source. Leetcode 407) Given an m x n matrix of positive integers representing the height of each unit cell in a 2D elevation map, compute the volume of water it is able to trap after raining. Example:

```
Given the following 3x6 height map:
[
  [1,4,3,1,3,2],
  [3,2,1,3,2,4],
  [2,3,3,2,3,1]
]

Return 4.
```

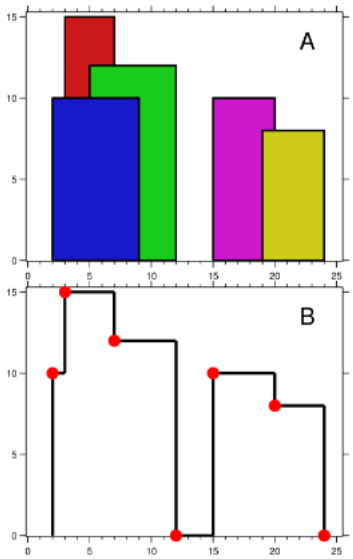
Visualize this matrix by drawing a 3D image based off of a 2D grid base. Each grid extends upwards by height specified in the corresponding cell in the matrix.

Implement `pow(x, n)` Implement the power frunction.

Find Minimum in sorted rotated array Suppose an array sorted in ascending order is rotated at some pivot unknown to you beforehand. (i.e., `0 1 2 4 5 6 7` might become `4 5 6 7 0 1 2`). Find the minimum element. You may assume no duplicate exists in the array.

Follow-up: What if duplicates are allowed?

The Skyline Problem (Source. Leetcode 218) A city's skyline is the outer contour of the silhouette formed by all the buildings in that city when viewed from a distance. Now suppose you are **given the locations and height of all the buildings** as shown on a cityscape photo (Figure A), write a program to **output the skyline** formed by these buildings collectively (Figure B).



The geometric information of each building (**input**) is represented by a triplet of integers `[Li, Ri, Hi]`, where `Li` and `Ri` are the `x` coordinates of the left and right edge of the ith building, respectively, and Hi is its height.

For instance, the dimensions of all buildings in Figure A are recorded as: `[ [2 9 10], [3 7 15], [5 12 12], [15 20 10], [19 24 8] ]` .

The **output** is a list of "key points" (red dots in Figure B) in the format of `[ [x1,y1], [x2, y2], [x3, y3], ... ]` that uniquely defines a skyline. A key point is the left endpoint of a horizontal line

segment. Note that the last key point, where the rightmost building ends, is merely used to mark the termination of the skyline, and always has zero height. Also, the ground in between any two adjacent buildings should be considered part of the skyline contour.

For instance, the skyline in Figure B should be represented as:`[ [2 10], [3 15], [7 12], [12 0], [15 10], [20 8], [24, 0] ]`.

Notes:

1. The input list is already sorted in ascending order by the left x position Li.
2. The output list must be sorted by the x position.
3. There must be no consecutive horizontal lines of equal height in the output skyline. For instance, `[...[2 3], [4 5], [7 5], [11 5], [12 7]...]` is not acceptable; the three lines of height 5 should be merged into one in the final output as such: `[...[2 3], [4 5], [12 7]`, ...]

Minimum Path Sum (Source. Leetcode 64) Given a $m \times n$ grid filled with non-negative numbers, find a path from top left to bottom right which minimizes the sum of all numbers along its path. *Note:* You can only move either down or right at any point in time.

Minimum Height Trees (Source. Leetcode 310) For a undirected graph with tree characteristics, we can choose any node as the root. The result graph is then a rooted tree. Among all possible rooted trees, those with minimum height are called minimum height trees (MHTs). Given such a graph, write a function to find all the MHTs and return a list of their root labels.

Format: The graph contains n nodes which are labeled from 0 to n - 1. You will be given the number n and a list of undirected edges (each edge is a pair of labels).

You can assume that no duplicate edges will appear in edges. Since all edges are undirected, [0, 1] is the same as [1, 0] and thus will not appear together in edges.

Closest Binary Search Tree Value (Source. Leetcode 270) Given a non-empty binary search tree and a target value, find the value in the BST that is closest to the target.

Wiggle Sort (Source. Leetcode 324) Given an unsorted array nums, reorder it in-place such that `nums[0] <= nums[1] >= nums[2] <= nums[3]....`

For example, given nums = `[3, 5, 2, 1, 6, 4]`, one possible answer is `[1, 6, 2, 5, 3, 4]`.

Wiggle Sort II (Source. Leetcode 324) Given an unsorted array nums, reorder it such that `nums[0] < nums[1] > nums[2] < nums[3]....`

For example, given `nums = [1, 3, 2, 2, 3, 1]`, one possible answer is `[2, 3, 1, 3, 1, 2]`.

Number of Islands II A 2d grid map of m rows and n columns is *initially filled with water*. We may perform an *addLand* operation which turns the water at position `(row, col)` into a land. Given a list of positions to operate, count the number of islands after each *addLand* operation. An island is surrounded by water and is formed by connecting adjacent lands horizontally or vertically. You may assume all four edges of the grid are all surrounded by water.

See 3.5 for the first version of *Number of Islands* problem.

Word Squares Given a set of words (without duplicates), find all word squares you can build from them.

A sequence of words forms a valid word square if the kth row and column read the exact same string, where `0` $\leq$ `k` $\leq$ `max(numRows, numColumns)`.

For example, the word sequence `["ball","area","lead","lady"]` forms a word square because each word reads the same both horizontally and vertically.

```
b a l l
a r e a
l e a d
l a d y
```

All words will have the exact same length. Word length is at least 1 and at most 5. Each word contains only lowercase English alphabet a-z.

Go to Leetcode for more questions.

4 Behavioral

4.1 Standard

4.1.1 introduce yourself

This question is an ice breaker. For this kind of question, the most important points to hit are (1) What is your interest in software engineering, (2 Very briefly say your background; don't be too detailed because that will take up too much time in the interview. Be yourself. Be natural. I will probably say something as follows.

> I major in Computer Science. I am expected to graduate in June, 2017. I am interested in backend or fullstack software development. I also hope to do work that involves some flavor of research, because I love doing research. I am also interested in using machine learning to solve some of problems I will work on. I am currently working at the Robotics State-Estimation Lab at UW. I can talk more about that project later. [(but) The big parts that I've contributed is that I've improved the navigation system for our robot, and also made a pipeline for data collection of the deep learning model.] *This part can be omitted* Besides robotics, I am also leading a group of 4 to work on the Koolio project, a website for people to share flippable content. In the last summer, I worked at CME Group for basically full stack development of a web application for helping my PM creating JIRA subtickets for the upcoming sprint (2 weeks). I think I am well prepared to be able to work at Google.

4.1.2 talk about your last internship

Here is what I may say:

> I interned at CME Group. The goal of the project that I worked on, solo, was to develop a web app to help replace my project manager's heavy Excel sheet work flow. Part of the Excel work flow involved creating "work types" which are basically cartesian product of several sets of short names such as "Front", "Back" as a set, "Dev", "Test" as a set, etc. So part of the functionality I implemented was to let the user configure the work types, save them into a database, and then use those to assign people tasks in another interface I made. This project used JavaScript, Java and Groovy on Grails, which is a Spring MVC under the hood. I also learned some knowledge in finance, and saw the closing of the

Chicago Mercantile Exchange physical place, and the transition to high-frequency online trading.

4.1.3 talk about your current research

Here is what I may say. Depending on the level of detail expected, I will vary my answer.

I started working at the lab in April, 2016, supervised by Postdoc Andrzej Pronobis and Professor Rajesh Rao. The goal of our project is to develop a novel probabilistic framework that enables robots to learn a unified deep generative model that captures multiple layers of abstraction, which is essentially one model that does the job of several independent ones such as place recognition, object detection, or action modeling. The motivation is that although those single models work well, they each require huge computation power, and they exchange information in a limited way.

The two main components that I have been in charge of are (1) mobile robot navigation, (2) design and development of a pipeline for representation learning of the deep learning model, which can collect virtual scans of the environment from the robot's sensor readings. I worked on both components independently. I was acknowledged for my help in collecting the data in the ICRA 2017 paper. I also have video demo of the navigation on the robot, and also contributed a navigation tuning guide to ROS community. I'm expected to work on simulated world generation soon.

4.1.4 talk about your projects

I will talk about Koolio, for sure.

The biggest side project I have been working on is the Koolio.io. Koolio.io is a website where users can share and view two-sided flippable cards. You may be tempted to know what's on the other side of a card, or you may just flip around as you enjoy. So the central core of this site is entertainment, in a unique form. [A piece of information has value as long as it is entertaining to people.]Can be omitted

When creating a card, user can decide what type of content to put on each side of it (we support text, image, and video for now). A deck groups multiple cards together, indicating their topic.

This project is implemented with Ruby on Rails. I chose this because I saw Github and Twitter was using this framwork, and it is suitable for our purpose, and has a big enough community behind. I chose to use PostgresSQl. It's not a big difference from MySQL, but since I have used MySQL before, I wanted to use something different. The production sever is NginX plus Unicorn. This is a quite popular combo for Rails projects. So I chose it.

(*Back story. Can be omitted if you see fit.*) I had this idea in sophomore year, and there were some stories in assembling the team. But the important piece is that, when I was doing internship in Chicago, my landlord is a student at SAIC. I basically persuaded her to be the designer for this project, and she has done a fantastic job. Being busy recently, I have got several new people in my team, with background in computer science and informatics. I think this project may one day be explosive, so I have been persisting on it. It's fun and satisfying to do.

4.1.5 why Google?

Google has great people, great projects, and great culture.

4.2 Favorites

4.2.1 project?

Koolio.io. See 4.1.4 for how to describe it.

4.2.2 class?

Major class: Machine learning, CSE 332, data abstraction, and Computer Graphics. Intriguing. Non-major class: JSIS 202 and OCEAN 250. Broaden my vision.

4.2.3 language?

Python. Python is like math. It is my go-to language if I just want to code something up. It is slow, but it has nice integration with C.

4.2.4 thing about Google?

People.

4.2.5 machine learning technique?

Support Vector Machine (SVM), and boosting. SVM optimizes the decision boundary by maximizing the margin (it is offline). Math shows that maximizing the margin is the same as minimizing the norm for the weight vector. Boosting is an ensemble algorithm that combines a set of weak learners into a strong learner. Another amazing thing about Boosting is that: it does not overfit.

4.3 Most difficult

4.3.1 bug?

In OS class, we had 5 labs. When doing lab4, multitasking and interprocess communication, I found a bug in lab2, about initializing the virtual memory. I started chasing that bug because my JOS runs out of envirionments too quickly. I finally found that I had an off-by-one error when I initialized the array to store all of the page metadata. It was very tough to find out.

4.3.2 design decision in your project?

For the Koolio project, the most difficult design decision is actually the UI, up to this point. It is the design of the card editor. It's flexibility vs. ease-to-use.

There are some other tough decision in the backend, about how the user activity works, and how the notification system works. The functionality is there, but these components are not yet done optimally and completely.

4.3.3 teamwork issue?

In winter last year, I took an entrepreneurship class. 80 percent of people in that class are not undergraduate like myself; they either already have family, or are pursuing post-graduate degrees such as MBA, or CSE P. We did pitches and divided into groups. For my group (diverse five-person group), our first idea was an website that works like Airbnb, but for languages; basically, people in non-native countries pay money to have lessons, taught by people with free time in native countries. But there was a great divergence of ideas and uncertainty if we could do it, because it has been done. There was a meeting where we discussed if we should break-up and join other teams in the class. I thought it was a

desperate situation, and I stood out and said something like, "Greg (our professor) said every group in this class ends up doing fantastic work. I trust that. But why don't we go ahead and see if that is true?" Then we began brainstorming new ideas, and eventually went for one that is related to helping small business owners choosing ideal location. We did great in the end.

4.3.4 failure?

Interviewed 10 companies last year for internship, and all rejected. I think I didn't do enough preparation for the coding interview. But I ended up working at the UW RSE lab, which is a great thing.

4.3.5 interview problem you prepared?

Just find on from the *Flagship Problems* section.

5 Appendix

5.1 Java Implementation of Trie

Below is my Java implementation of Trie; I wrote this when working on project 1 in CSE 332.

```
// Only for String keys
public class Trie {

   private TrieNode root;

   public Trie() {
      root = new TrieNode(); // start with empty string
   }

   public void insert(String word) {
      TrieNode current = root;
      for (int i = 0; i < word.length(); i++) {
         char c = word.charAt(i);

         if (current.children.containsKey(c)) {
            current = current.children.get(c);
         } else {
            TrieNode newNode = new TrieNode(c);
            current.children.put(c, newNode);
            current = newNode;
         }
         if (i == word.length() - 1) {
            current.isWord = true;
         }
      }
   }

   public boolean contains(String word) {
      TrieNode current = root;
      for (int i = 0; i < word.length(); i++) {
         char c = word.charAt(i);

         if (current.children.containsKey(c)) {
            current = current.children.get(c);
         } else {
            return false;
         }
```

```
            if (current.isWord && i == word.length() - 1) {
               return true;
            }
         }
         return false;
      }

      public boolean hasPrefix(String prefix) {
         TrieNode current = root;
         for (int i = 0; i < prefix.length(); i++) {
            char c = prefix.charAt(i);

            if (current.children.containsKey(c)) {
               current = current.children.get(c);
            } else {
               return false;
            }
         }
         return true;
      }

      private class TrieNode {
         public char key;
         public Map<Character, TrieNode> children;
         public boolean isWord;

         public TrieNode(char key) {
            this.key = key;
            this.children = new HashMap<>();
            this.isWord = false;
         }

         public TrieNode() {
            this('\0');
         }
      }
}
```

5.2 Python Implementation of the KMP algorithm

Based on ideas discussed in 2.8.2, I implemented the KMP algorithm and tested it against the Python's own string `find` function.

```
def kmpTable(p):
    i, j = 1, 0
    table = {-1: -1, 0: -1}
    while i < len(p):
        if p[i] == p[j]:
            table[i] = j
            j += 1
            i += 1
        else:
            if j > 0:
                j = max(0, table[j-1] + 1)
            else:
                table[i] = -1
                i += 1
    return table


def kmp(W, P):
    table = kmpTable(P)
    k = 0
    while k < len(W):
        d = 0
        j = k
        # Check if the remaining string is long enough
        if len(W) - k < len(P):
            return -1
        for i in range(0, len(P)):
            if P[i] == W[j]:
                d += 1
                j += 1
            else:
                break # mismatch

        # KMP rules
        if d == len(P):
            return k
        elif d > 0 and table[d-1] == -1:
            k = k + d
        elif d > 0 and table[d-1] != -1:
            k = k + d - table[d-1] - 1
        else: # d == 0
            k = k + 1
    return -1
```

5.3 Python Implementation of Union-Find

Based on details described in 2.1.16, I implemented the Union-Find data structure in python as follows.

```
class UnionFind:
    """Caveat: This implementation does not support adding
        additional elements other than ones given initially."""
    def __init__(self, elems):
        """
        Constructs a union find data structure. Assumes that all
        elements in elems are hashable.
        """
        self.elems = list(elems)
        self.idxmap = {}
        self.impl = []
        for i in range(len(elems)):
            self.idxmap[self.elems[i]] = i
            self.impl.append(-1)

    def find(self, x):
        """return the canonical name of the set that element x
            belongs to"""
        if self.__implVal(x) < 0:
            return x
        return self.find(self.elems[self.__implVal(x)])

    def union(self, x, y):
        """union the two sets that each of x and y is in."""
        # We want |s(c1)| <= |s(c2)|. Here, s(N) means the set
            represented by the canonical element N.
        c1, c2 = self.find(x), self.find(y)
        if c1 == c2:
            return c1 # already unioned
        s1, s2 = abs(self.__implVal(c1)), abs(self.__implVal(c2))
        if s1 > s2:
            c1, c2 = c2, c1
        self.impl[self.idxmap[c1]] = self.idxmap[c2] # Connect.
        self.impl[self.idxmap[c2]] = -(s1 + s2) # Update the size.

        return c2 # Return the canonical element of the new set.

    def __implVal(self, x):
        return self.impl[self.idxmap[x]]
```