Ryan, these 21 questions are for practice. A 45 minute technical interview might have three such questions. Questions marked with ⋆ are (inspired by) actual ones I've gotten. Questions marked with ! are especially challenging; I wouldn't expect more than one in an interview. Each question has a main version as well as a followup twist that an interviewer might ask upon satisfactory solution of the main version. It might be good to avoid reading these yourself, instead asking Chloe to draw randomly from these questions during a practice interview.

Questions of the form *determine xyz given abc* stand for *construct a program that determines xyz given abc*. Also implicit is the expectation that you explain what assumptions you make and that you offer an informal complexity analysis for each solution. For example, you may want to remark on your assumptions about the types and probability distribution of input data and on which parameters you regard as asymptotically large.

0. ⋆ **legal parentheses.** check whether a given string of `(`s and `)`s is well-formed. —— TWIST: what if we permit both round and square brackets? we regard `([)]` as ill-formed.

1. popularity contest. from a non-empty list of names identify the name(s) of greatest frequency. —— TWIST: identify the 10^4 most frequently occuring names (assumed unique) from a list of length 10^6.

2. justify text. a paragraph is a list $(x_i : 0 \leq i < N)$ of words with natural widths w_i. we segment a paragraph into lines and slightly stretch each word. precisely, we choose indices $(i_j : 0 \leq j < J)$ of the first word in each jth line as well as a stretch amount $(\rho_i \in [-0.1, +0.1] : 0 \leq i < N)$ for each word. we then render the ith word at width $(1 + \rho_i)w_i$ so that each line fits, i.e., so that the excess $e_j = W - \sum_{l_j \leq i < l_{j+1}} (1 + \rho_i)w_i$ is non-negative. determine ρ_i, J, and l_j that minimize the *distortion* $D = \sum_j e_j^3 + \lambda \sum_i \rho_i^2$. —— TWIST: in a greedy approach, we set l_1 and $(\rho_i : l_0 \leq i < l_1)$ to minimize the distortion $e_0^3 + \lambda \sum_{l_0 \leq i < l_1} \rho_i^2$ of the first line, then repeat. is this greedy approach suboptimal?

3. missing numbers. given a length-$(n-1)$ array L containing distinct integers in the range $R = [0, n)$, determine the element of R not in L. —— TWIST: what if L misses two elements of R?

4. viola jones. imagine a 2d $H \times W$ array of `floats`, some negative. find a contiguous sub-array (specified by a corner (r, c) a shape $h \times w$ with $r + h \leq H, c + w \leq W$) whose elements have greatest sum? —— TWIST: what if we wish to maximize the *product*?

5. ⋆ **merge sort.** implement merge sort. —— TWIST: compare merge sort with some alternatives.

6. ! **memory allocation.** implement a memory allocator (`char* malloc(int nb_bytes)`, `char* free(int nb_bytes)`). use a primitive `char* sbrk(int nb_kilobytes)` to request chunks of memory from the OS. —— TWIST: our game in each video frame allocates many small (4-byte to 8-byte) polygons to be freed together before the next frame. adapt your memory allocator to this setting. speed is key.

7. palindromic substring. find a longest contiguous palindromic substring of a given string. —— TWIST: count the length-$\geq k$ contiguous palindromic substrings of a given string and for a given k.

8. distinct elements. consider a type that supports a comparator with respect to which it is totally ordered. determine whether a list of elements of this type has all elements distinct. —— TWIST: does your solution use as few comparisons as possible (asymptotically)?

9. ⋆ **sparse vectors.** a sparse vector is a list of numbers most of which are zero. implement a space-saving sparse vector type along with a dot product. —— TWIST: a sparse matrix is defined accordingly. build on your vector code to represent and multiply large sparse matrices.

10. ! ⋆ **herding cows.** a farmer tries to herd musical cows all onto some single hill in a set H of hills. each cow starts on a hill in H. when the farmer blares an s in a set S of songs, all the cows on hill h move to the hill $f(s, h)$. given H, S, f determine whether the farmer may achieve her goal by blaring some sequence of songs. —— TWIST: what if the farmer seeks to clear a specific hill $h_\star \in H$ of cows?

11. super spreader. we represent the (reflexive, symmetric) 'physically meets' relation on people by giving for each person a list of contacts. *compute the relation's transitive closure.* choose an output representation that uses a reasonable amount of space. —— TWIST: a relation's *size* is the maximum among the cardinalities of equivalence classes of the induced equivalence relation. determine a node that, when removed from the graph, results in a relation of least size.

12. ! **bipartite graph.** the theory of markov chains references *k-periodic graphs*. an undirected graph is k-periodic if k divides the length of every cycle. determine whether a given graph is 2-periodic. —— TWIST: does there exist a $k \geq 2$ for which the given graph is k-periodic?
13. ⋆ **family tree.** implement a binary-tree-of-strings type. ancestor trees are of this type. we display ancestor trees in this style: `george(katy(mike,carole),will(charlie(liz,phil),diana))`. compute the length of such a display. —— TWIST: compute two people's genetic relation given their ancestor trees. the genetic relation of siblings is 1/2; of cousins is 1/8; of strangers is 0; and so forth. the two trees' nodes name people, and we assume people have distinct and unique names.
14. ⋆ **knight tours.** how many ways can a knight on an otherwise empty chessboard move in exactly n steps from one corner to the opposite corner? —— TWIST: consider a simplified game of chess with three black knights, one white king, and no other pieces. the knights start as close as possible to one corner; the king, to the opposite corner. this game has no rules about draws — no stalemate, no draw by repetition, no draw upon 50 reversible moves. *can black force a checkmate?*
15. **logical consistency.** fix a finite set of logical atoms. is a given list of assertions, each of the form p or $\mathsf{p} \to \mathsf{q}$, logically consistent? here, p, q stand for atoms or their negations. —— TWIST: what if we also permit a small number of assertions of the form $(\mathsf{p} \wedge \mathsf{q}) \to \mathsf{r}$?
16. ⋆ **an evil company.** a csv file is a text file representing a 2D grid of entries, with rows delimited by newlines and with elements of a row delimited by commas. each of our csv file's rows records some person's data with some store. its columns list: `a` the store and the person's `b` username at that store, `c` email address, `d` phone number, and `e` year-of-birth. *parse the file; do some two rows represent the same person?* use reasonable heuristics. —— TWIST: suppose now that some rows have missing entries (represented by the string `UNK`). heuristically impute what missing entries you can.
17. **city hall.** a new hall is to be built among a grid of streets. we label street intersections by (x, y) for $0 \leq x, y < 100$. it takes $|x - x'| + |y - y'|$ minutes to walk from (x, y) to (x', y'). at (x, y) live $p(x, y)$ many residents. place the hall at an intersection so as to minimize the walking time from residence to hall, averaged uniformly over residents. —— TWIST: a rectangular park has northwest corner (a, b) and southeast corner (A, B). we may not place city hall in the park. what then?
18. ! **task scheduling.** we employ three oxen on our farm: jesse, pat, and sandy. each t in a size list T of tasks requires the simultaneous effort of n_t many oxen over a contiguous period of $d_t \in \mathbb{N}$ hours. a schedule assigns each ox-hour to some task. schedule so as to complete all tasks by day's end, if possible. —— TWIST: remark on the computational hardness of this task as we modify its parameters.
19. ! **graph homology.** we're given a connected undirected graph G^0. an edge e is *crucial* if removing e disconnects the graph. count the crucial edges. —— TWIST: determine a sequence of edges to remove that results in a longest chain $G^0, \cdots, G^n$ of connected undirected graphs.
20. ! ⋆ **lru cache.** write an lru cache that maps small strings (say, urls) to large strings (say, html). that is, implement a data structure that at any time stores $\leq k$ many key-value pairs; that answers queries for whether a key is present and for its value if so; and that permits insertion of new key-value pair, with the least-recently inserted-or-queried pair removed if space is needed. —— TWIST: show us how you would increase your confidence that your code is correct.

Sample Solutions

0. legal parentheses.

```
def well_formed(s):
    depth = 0
    for c in s:
        if c=='('    depth += +1
        elif c==')': depth += -1
        else:        assert False, "didn't expect `{}`".format(c)
        if depth < 0: return False
    return depth==0
```

5. merge sort.

```
def zip_ordered_lists(l_a, l_b):
    l = []
    i = 0; j = 0
    while i != len(l_a):
        while j != len(l_b) and l_b[j] < l_a[i]:
            l.append(l_b[j]); j += 1
        l.append(l_a[i]); i += 1
    return l

def merge_sort(l):
    if len(l) <= 1: return l
    half = int(len(l)/2)
    return zip_ordered_lists(
                merge_sort(l[:half]),
                merge_sort(l[half:])
           )
```

10. herding cows.

insight: the farmer can unite *all* hills if and only if she can unite *any pair* of hills.

```
# assume H contains sortable, hashable elements

# consider only sorted pairs to cut work in half:
all_pairs = set([tuple(sorted([h_a, h_b])) for h_a in H for h_b in H)

def predecessor_pairs(f):
    predecessors = {p: set([]) for p in all_pairs}
    for s in S:
        for p in all_pairs:
            succ = tuple(sorted(f(s, h) for h in p))
            predecessors[succ].add(p)
    return predecessors

def can_herd(f):
    predecessors = predecessors_pairs(f)
    unfied = set([(h, h) for h in H])
    unifiable = reachable_pairs(unified, predecessors)
    return unifiable == all_pairs
```

here, `reachable_pairs` is just an alias for breadth-first-search. here's an implemnentation:

```
def reachable_pairs(starting_pairs, edges_from):
    # BFS:
    reachable = staring_pairs.copy()
    frontier = starting_pairs
    while True:
        old_frontier = frontier
        frontier = set([])
        for p in old_frontier: frontier.update(edges_from[p])
        if not frontier.difference(reachable): break
        reachable.update(frontier)
    return reachable
```

15. logical consistency.

```
# we represent each atom as a string in A and each judgement (of an atom
# as true or false) by a pair (string, bool).
# we're given P0, the set of simple assertions p, and P1, the set of
# asserted implications p --> q.  here, p, q stand for judgements.

def inference_rules(P1):
    rules = {(x,v):set([]) for x in A for v in (False, True)}
    for ((p, a), (q, b)) in P1:
        rules[(p, a)].add((q, b))       # ponens
        rules[(q, not b)].add((p, a))   # contrapositive
    return rules

def is_consistent(P0, P1):
    js = judgements_from(P0, inference_rules(P1))
    for (x, v) in js:
        if (x, not v) in js: return False
    return True
```

here, `judgements_from` is just an alias for breadth-first-search. here's an implemnentation:

```
def judgements_from(P0, rules):
    # BFS:
    judgements = P0.copy()
    frontier = P0
    while True:
        old_frontier = frontier
        frontier = set([])
        for p in old_frontier: frontier.update(rules[p])
        if not frontier.difference(judgements): break
        judgements.update(frontier)
    return judgements
```

20. lru cache.

insight: we keep the keys in order of last use; to permit fast deletion, we use a doubly linked list instead of an array; to permit fast access-by-key, we also maintain a hash table mapping keys to nodes of the list.

```
class LruCache:
    def __init__(self, k):
        self.k = k
        self.data = {} # key --> [value, DLL node for key]
        self.head = DLLNode(None     , None, None)
        self.tail = DLLNode(self.head, None, None)

    def query(self, key):
        self.touch(key)
        present = key in self.data
        return (present, self.data[key][0] if present else None)

    def update(self, key, value):
        if key not in self.data:
            self.data[key] = (value, DLLNode(self.head, self.head.succ, key))
            self.trim()
        self.touch(key)
        self.data[key][0] = value

    def touch(self, key):
        node = self.data[key][1]
        node.remove()
        node.insert(self.head, self.head.succ)

    def trim(self):
        while len(self.data) > self.k:
            last = self.tail.pred
            last.remove()
            del self.data[last.cargo]
```

for completeness, here's an implementation of `DLLNode`:

```
class DLLNode:
    def __init__(self, pred, succ, cargo):
        self.cargo = cargo
        self.insert(pred, succ)

    def insert(self, pred, succ):
        self.pred = pred
        self.succ = succ
        if self.pred is not None: self.pred.succ = self
        if self.succ is not None: self.succ.pred = self

    def remove(self):
        if self.pred is not None: self.pred.succ = self.succ
        if self.succ is not None: self.succ.pred = self.pred
```