TUT 1

```
1: function STARS(n)
2:     for i ← 1; i ≤ n; i++ do
3:         Print '*' i times
```

b) Assume for simplicity that n is even. To lowerbound the running time, we consider only the number of stars printed during the last $\frac{n}{2}$ iterations. Since this is part of the full execution, analyzing only this part gives a lower bound on the total running time. The main observation we need is that for each of the considered iterations, we print at least $n/2$ stars, allowing us to lower bound the total number of stars printed:

$$\sum_{j=1}^{n} j \geq \sum_{j=n/2+1}^{n} \frac{n}{2} = \frac{n^2}{4} = \Omega(n^2).$$

这里只用看 partial running of original，并且说每个 loop 最少打印多少个星星。

TUT 2

Consider the problem of given an integer n, generating all possible permutations of the numbers {1,2,...,n}. Provide a recursive algorithm for this problem.

```
1: function PERMUTATIONS-RECURSIVE(n)
2:     # input: integer n
3:     # do: print all permutations of order n
4:     A ← [1, 2, ..., n]
5:     HELPER(A, 0)
```

分别

```
1: function HELPER(A, i)
2:     if i = size(A) then
3:         Print A
4:     for j ← i; j < n; j++ do
5:         Swap A[i] and A[j]
```

确保 0,1,2,3 开头


```
6:         HELPER(A, i + 1)
7:         Swap A[i] and A[j]
```

复位作用

TUT 2

Problem 3. Design a linear time algorithm that given a tree T computes for every node u in T the size of the subtree rooted at u.

Solution 3. We use a recursive helper function. When the function is called at some node u in T, we recursively compute the size of the left and right subtrees of u, add 1, set the result to be the size of the subree at u, and return the value to be reused further up in the recursion.

```
1: function SUBTREE-SIZE(T)
2:     SIZE-HELPER(T.root)
```

```
1: function SIZE-HELPER(u)
2:     u.sub_size ← 1
3:     for w ∈ u.children() do
```



```
4:         u.sub_size ← u.sub_size+ SIZE-HELPER(w)
5:     return u.sub_size
```

这里求子树的大小

Problem 8. The balance factor of a node in a binary tree is the absolute difference in height between its left and right subtrees (if the left/right subtree is empty we consider its height to be -1). Design an algorithm for computing the balance factor of **every** node in the tree in $O(n)$ time.

Solution 8. We use a recursive helper function that takes as input a node u and computes the balance factor at u and returns the height of the subtree rooted at u.

```
1: function BALANCE(T)
2:     BALANCE-HELPER(T.root)
```

```
1: function BALANCE-HELPER(u)
2:     left_height ← −1
3:     right_height ← −1
4:     if u.left ≠ nil then
5:         left_height ← BALANCE-HELPER(u.left)
6:     if u.right ≠ nil then
7:         right_height ← BALANCE-HELPER(u.right)
8:     u.balance ← |left_height − right_height)|
9:     return 1 + max(left_height, right_height)
```

We claim that BALANCE-HELPER(u) sets $u.balance$ to the balance factor at u and returns the height of u. To prove the correctness of this claim we use the inductive assumption that the algorithm returns the correct height for the left and right subtrees. If both are empty then BALANCE-HELPER returns 0 and sets $u.balance$ to 0, which is the correct thing to do. If one of them is non-empty we return its height plus 1 and set $u.balance$ to its height, which again is correct. Finally, if both are non-empty, we return $1 + \max(left\_height, right\_height)$ and set $u.balance$ to $|left\_height - right\_height|$, which again is correct.

Each call to BALANCE-HELPER(u) takes $O(1)$ time, not counting the work done in recursive calls. Therefore, the total running time is $O(n)$ time.

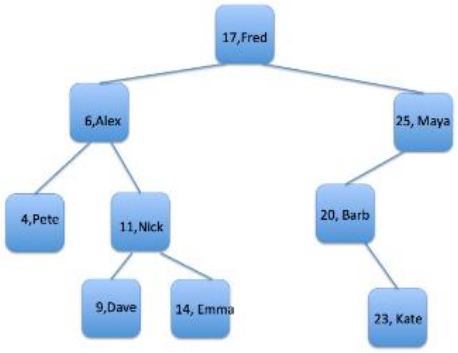
这里 left_height 是左树的高度，right_height 是右树的高度
此外，每个 node 海增加了一个新的 variable 叫做 balance
单节点（无 child）的 balance 是 0

QUIZ 3

Question 7

0 / 1 pts

Consider the binary search tree shown below. After we perform remove(25), which entry will be the right child of the node with key 17?



Correct Answer

- (20, Barb)
- The node with key 17 has no right child.

You Answered

- (23, Kate)
- (14, Emma)

这里没有 right child，所以直接 promote left child

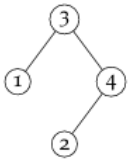
TUT 4

Problem 4. Consider the following algorithm for testing if a given binary tree has the binary search tree property.

```
1: function TEST-BST(T)
2:     for u ∈ T do
3:         if u.left ≠ nil and u.key < u.left.key then
4:             return False
5:         if u.right ≠ nil and u.key > u.right.key then
6:             return False
7:     return True
```

Either prove the correctness of this algorithm or provide a counter example where it fails to return the correct answer.

Solution 4. The algorithm is not correct as it returns True on the following tree lacking the binary search tree property.



all elements of right subtree should large than root node.

Problem 7. Consider the following operation on a binary search tree: MEDIAN() that returns the median element of the tree (the element at position $\lfloor n/2 \rfloor$ when considering all elements in sorted order).

Give an implementation that runs in $O(h)$, where h is the height of the tree. You are allowed to augment the insertion and delete routines to keep additional data at each node.

Solution 7. We can augment the nodes in our binary search trees with a size attribute. For a node u, $u.size$ is the number of nodes in the subtree rooted at u. When inserting a new key at some node w, we only need to increase by 1 the size of ancestors of w, which takes $O(h)$ time. Similarly, when we delete a node w with at least one external child, we only need to decrease by 1 the size of ancestors of w. Hence, the insertion and deletion times remain the same.

We implement a more powerful operation called POSITION(k), which returns the kth key (in sorted order) stored in the tree. When searching for the kth element at some node u, we check if $u.left.size \geq k$ in which case we know that the kth element is in the left subtree and we search for the kth key in $u.left$. Otherwise, if $u.left.size = k - 1$, the kth key is at the root u. Finally, if $u.left.size < k - 1$, then we know that the kth element is in the right subtree, so we search for the $(k - u.left.size - 1)$th key there, i.e., k minus the number of nodes smaller than the root of $u.right.key$.

The complexity is $O(h)$ because we do $O(1)$ work at each node and we only traverse one path from the root to the element we are searching for.

一种在 node 添加新的 variable 的思路，只要确保题目问的是只用 O（logn），那么这种思

路就比较有效

TUT 5

Problem 5. Given an array A with n distinct integers, design an $O(n \log k)$ time algorithm for finding the kth value in sorted order.

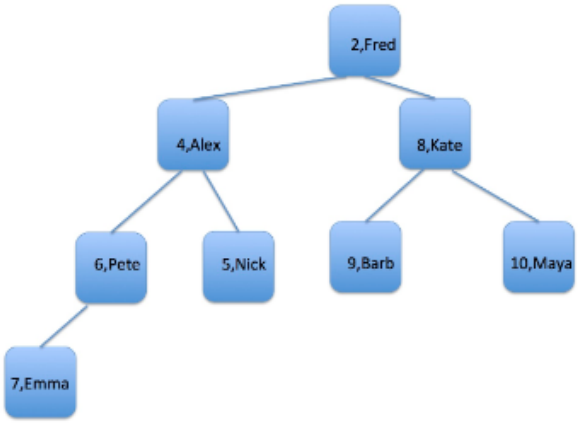
Solution 5. We use a priority queue that supports MAX and REMOVE-MAX operations. We start by inserting the first k elements from A into the priority queue. Then we scan the rest of array comparing the current entry $A[i]$ to the maximum value in the priority queue, if $A[i]$ is greater than the current maximum, we can safely ignore it as it cannot be the kth value due to the fact that the priority queue holds k values smaller than $A[i]$. On the other hand, if $A[i]$ is less than the current maximum, we remove the maximum from the queue and insert $A[i]$. After we are done processing all the entries in A, we return the maximum value in the priority queue.

To argue the correctness of the algorithm we can use induction to prove that after processing $A[i]$, the k smallest elements in $A[0, i]$ are in the queue. Since we return the largest element in the queue after processing the whole array A (which we just argued holds the k smaller elements in A), we are guaranteed to return the kth element of A.

The time complexity is dominated by $O(n)$ plus the time it takes to perform the REMOVE-MAX and INSERT operations in the priority queue. In the worst case we perform n such operations each one taking $O(\log k)$ time, when we implement the priority queue as a heap. Thus, the overall time complexity is $O(n \log k)$.

QUIZ 4

Consider the min-heap (structured as a tree) that is shown in the figure below. The entries have integer key and String value. Suppose that we now perform a removeMin() operation. After the removal has been done, what are the keys in the first two rows of the tree (shown in level order, that is we list the root, then root.left, then root.right)?



Correct Answer

- 4, 5, 8
- 4, 7, 8
- 7, 4, 8

You Answered

- 4, 6, 8

DownHeap 要和最小的 element 交换

QUIZ 5

Consider the hash table of size 11 that uses linear probing shown below, where the hash function is h(k) = (k mod 11). Thus h(14)=3, h(25)=3.

What indices are examined if we now perform get(5)?

0	1	2	3	4	5	6	7	8	9	10
			14	defunct	25	4		19		

Correct!

- (●) 5,6,7
- ( ) 3,4,5,6
- ( ) 4,5,6,7
- ( ) 5,6

1. start at cell h(k)
2. Probe consecutive locations until one of the following occurs
    - An item with key k is found, or
    - An empty cell is found, or
    - N cells have been probe

Question 4

1 / 1 pts

Consider the hash table of size 11 that uses linear probing shown below, where the hash function is h(k) = (k mod 11). Thus h(14)=3, h(25)=3.

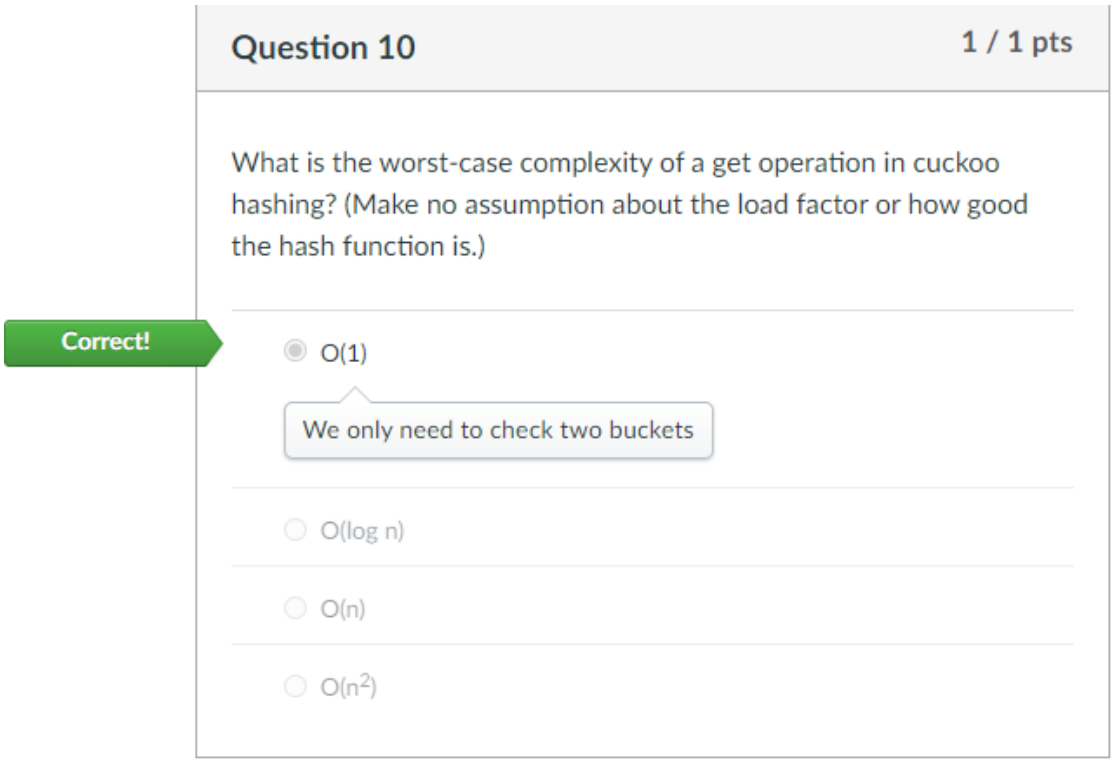
What indices are examined if we now perform get(3)?

0	1	2	3	4	5	6	7	8	9	10
			14	defunct	25	4		19		

Correct!

- (●) 3,4,5,6,7
- ( ) 3,4
- ( ) 3
- ( ) 3,4,5,6

get(k): must pass over cells with DEFUNCT and keep probing until the element is found, or until reaching an empty cell


Question 10
1 / 1 pts
What is the worst-case complexity of a get operation in cuckoo hashing? (Make no assumption about the load factor or how good the hash function is.)
Correct!
O(1)
We only need to check two buckets
O(log n)
O(n)
O(n^2)

TUT 6

Problem 4. Work out the details of implementing cycle detection in cuckoo hashing based on keeping a flag for each entry.

Solution 4. We augment the hash table entries with a boolean attribute called flag. Assume that at the start of the put routine all entries are unflagged (i.e., all flags are set to false). Suppose we are trying to put a new element x into the hash table and that the element could store in hash entries i or j. To test if there exists an eviction path starting at, say, i, we design a recursive auxiliary routine TEST-CHAIN(x, i).

This auxiliary routine first checks if i is empty; if that is the case, then the test is successful. If the entry is not empty, it checks if the entry is flagged; if that's the case the test is

unsuccessful. Finally, if the entry is not empty and not flagged, let y be the item currently stored at i and k be the alternative entry for y. We flag entry i, and call TEST-CHAIN(y, k). Regardless of the outcome (successful or unsuccessful) before returning we unflag the entry previously flagged so that all entries are unflagged at the start of the next put call.

If both TEST-CHAIN(x, i) and TEST-CHAIN(x, j) are unsuccessful, then it is not possible to put x into the has table. The running time is proportional to the length of the eviction chain tests.

Note that the unsuccessful eviction chain test could be much longer than the successful one. We could modify the algorithm to interleave the search for a chain so that we do work proportional to the length of the shortest successful chain (if one exists) but this would complicate the algorithm a great deal and would require us to keep two flags. Not really worth the effort.

Problem 5. Design a sorted hash table data structure that performs the usual operations of a hash table with the additional requirement that when we iterate over the items, we do so in the order in which they were inserted into the hash table. Iterating over the items should take $O(n)$ time where n is the number of items stored in the hash table. Your data structure should only add $O(1)$ time to the standard put, get, and delete operations.

Solution 5. In addition to the hash table, we keep a doubly linked list of the entries and augment each entry in the hash table to also have a pointer to its position in the list.

When we need to put a new item, after inserting it into the hash table in the usual way, we add it to the end of the list and set the pointer of the entry in the hash table accordingly in $O(1)$ time.

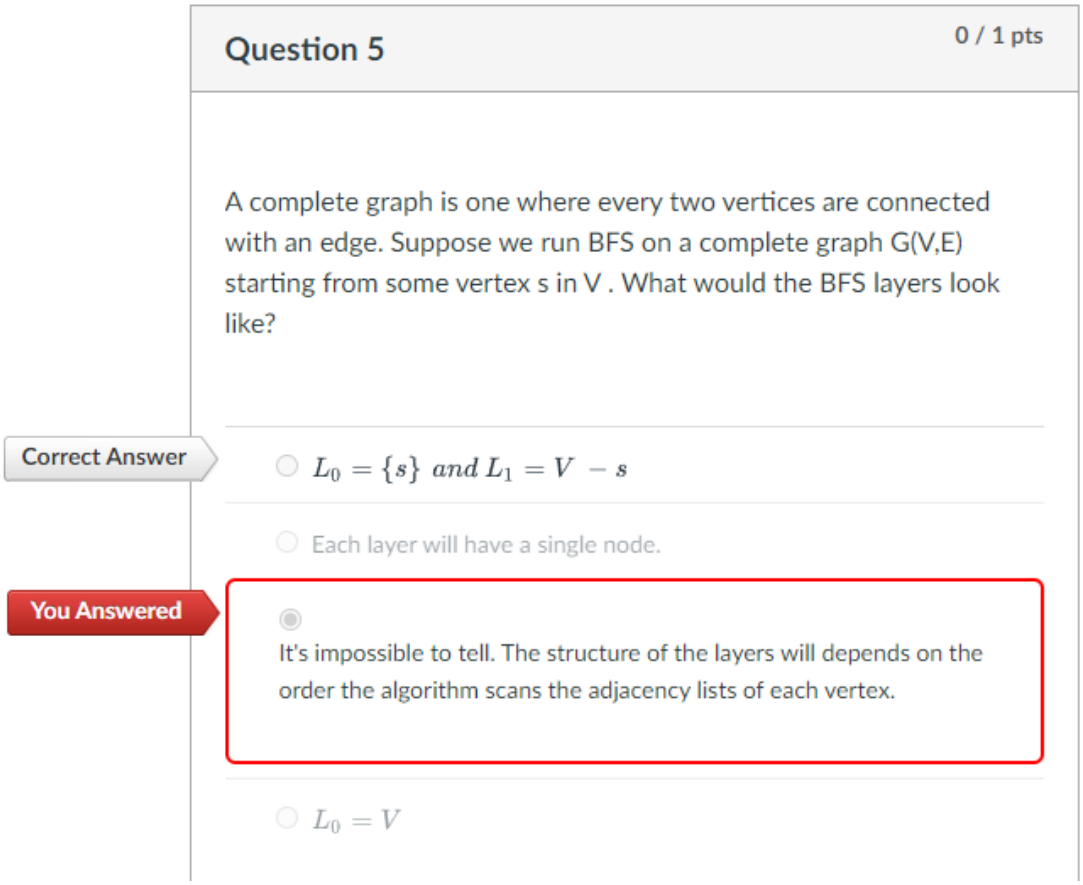
When we remove an item, after removing it from the hash table in the usual way, we use the pointer to the doubly linked list to remove it from there as well in $O(1)$ time.

When we update an existing item, after updating it in the usual way, we move its position in the list to the end in $O(1)$ time or we leave it in place depending on how we want to interpret this case; i.e., we care about the order of when the key of the item arrived or when the value of the item arrived.

Whenever we need to iterate over the entries, we use the linked list. Since iterating through all elements of a doubly linked list takes $O(n)$ time, this satisfies our requirements.

Map 没有 iteration 的功能。

QUIZ 6



Complete graph is fully connected.

Question 10

0 / 1 pts

Consider an undirected graph G that is connected. Suppose we perform a depth-first search starting from vertex v. Which of the following is true?

Correct Answer

- Every vertex (other than v itself) is reached by exactly one DFS edge.
- Every vertex (other than v itself) is reached by exactly one DFS edge and by exactly one back edge.
- Vertex v is reached by at least one back edge.

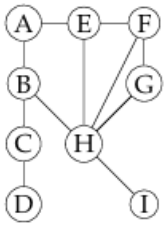
You Answered

- Vertex v is reached by at least one DFS edge.

参考 DFS edge 的定义

TUT 7

Problem 1. Consider the following undirected graph.



a) Starting from A, give the layers the breadth-first search algorithm finds.

b) Starting from A, give the order in which the depth-first search algorithm visits the vertices.

Solution 1.

a) $L_0 = \{A\}$
$L_1 = \{B, E\}$
$L_2 = \{C, F, H\}$
$L_3 = \{D, G, I\}$

b) There are a number of different solution possible, when the algorithm can choose multiple nodes to recurse on. This example picks the lexicographically first node, but other choices are just as valid.
$A, B, C, D, H, E, F, G, I$

Problem 3. Give an $O(n)$ time algorithm to detect whether a given undirected graph contains a cycle. If the answer is yes, the algorithm should produce a cycle. (Assume adjacency list representation.)

Solution 3. We run DFS with a minor modification. Every time we scan the neighborhood of a vertex u, we check if the neighbor v has been discovered before and whether it is different than u's parent. If we can find such a vertex then we have our cycle: $v, u, \text{parent}[u], \text{parent}[\text{parent}[u]], \ldots, v$.

We only need to argue that this algorithm runs in $O(n)$ time. Consider the execution of the algorithm up the point when we discovered the cycle. After the $O(n)$ time spent initializing the arrays needed to run DFS, each call to DFS-VISIT takes time that is proportional to the edges discovered. However, up until the time we find the cycle we have only discovered tree edges. So the total number of edges is upper bounded by $n - 1$. Thus, the overall running time in $O(n)$.

在这个特定算法中，DFS 在发现第一个环之前**只会遍历树边（DFS edges）**，这些边的数量在无环的连通图中最多为 n - 1 条。因此，直到发现环之前，DFS 只会遍历 n−1 条边，时间复杂度近似为 O(n)。

Problem 7. In a connected undirected graph $G = (V, E)$, a vertex $u \in V$ is said to be a cut vertex if its removal disconnects G; namely, $G[V - u]$ is not connected.

The aim of this problem is to adapt the algorithm for cut edges from the lecture, to handle cut vertices.

a) Derive a criterion for identifying cut vertices that is based on the `down-and-up[·]` values defined in the lecture.

b) Use this criterion to develop an $O(n + m)$ time algorithm for identifying all cut vertices.

Solution 7. Recall that the main idea for the cut edges algorithm, was to run DFS on G and for every $v \in V$ set level(v) of the vertex v in the DFS tree. Then we defined down-and-up(u) be the highest level (closer to the root) we can reach by taking any number of DFS tree edges down and one single back edge up.

a) First note that the leaves of the DFS tree cannot be cut vertices, since the rest of the DFS tree keeps the graph together.
Secondly, note that if the root has two or more children, then it must be a cut vertex since there are no edges connecting its children subtrees (we only have DFS tree edges or back edges connecting a node to one of its ancestors).
Finally, consider an internal node u other than the root. Note that after we remove u, it may be the case that the subtree rooted at one of its children gets disconnected from the rest of the graph. Let v be one of u's children. If DOWN-AND-UP$(v) <$ LEVEL(u) then after we remove u the subtree rooted at v remains connected by the edge that connects some descendant of v to some ancestor of u. Otherwise, if DOWN-AND-UP$(v) \geq$ LEVEL(u) then u is a cut vertex.

b) The algorithm first runs DFS and computes level(u) and DOWN-AND-UP(u) for all $u \in V$. If the root has two or more children, it is declared a cut vertex. For any other internal node u, we check if it has a child v such that DOWN-AND-UP$(v) \geq$ LEVEL(u). If this is the case, we declare u to be a cut vertex.
The correctness of the algorithm rests on the observations made in the previous point. The running time is dominated by running DFS and computing LEVEL and DOWN-AND-UP, which can be done in $O(n + m)$ time.

这里描述了具体怎么用 DFS 判断 cut vertex, D&U(v) >= Level(u)

Problem 9. Let G be a connected undirected graph. Design a linear time algorithm for finding all cut edges by using the following guide:

a) Derive a criterion for identifying cut edges that is based on the `down-and-up[·]` values defined in the lecture.

b) Use this criterion to develop an $O(n+m)$ time algorithm for identifying all cut edges.

Solution 9.

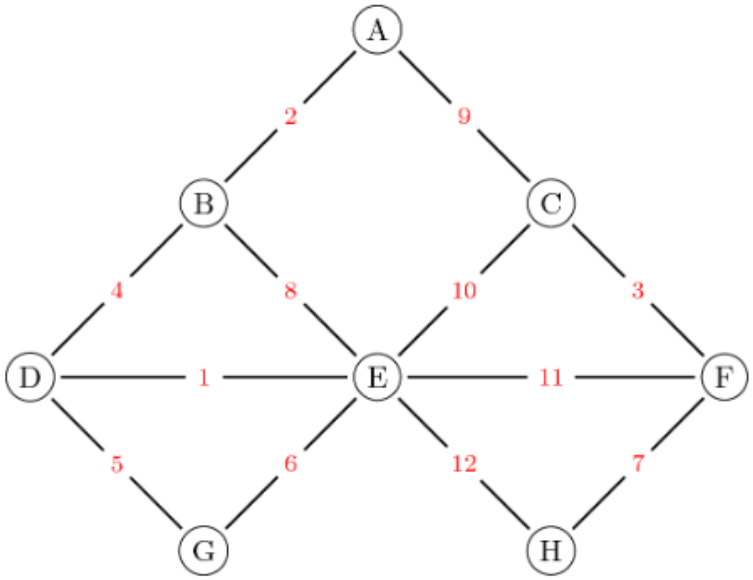
a) First note that only DFS-tree edges can be cut edges because other edges can never disconnect the graph (the DFS tree keeps the graph in one piece).
Let (u,v) be a DFS-tree edge. Assume that u is v's parent in the DFS tree. If (u,v) is not a cut edge then some vertex in T_v (the subtree rooted at v) must have a back edge to u or a higher vertex; i.e., if $\text{DOWN-AND-UP}(v) \leq \text{LEVEL}(u)$.

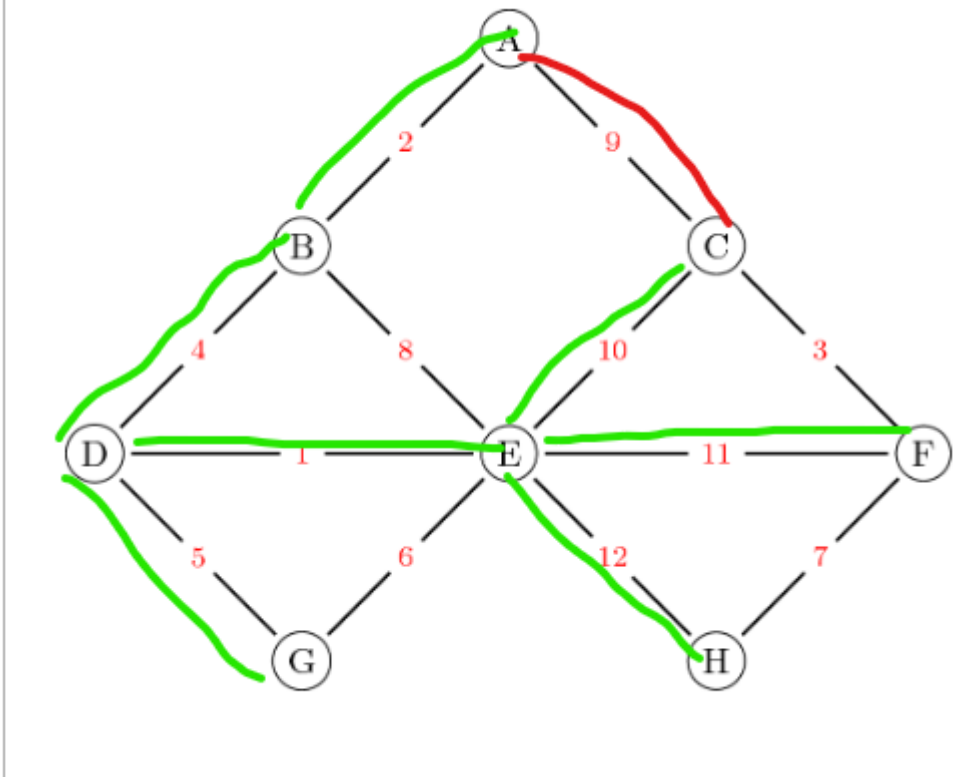
b) The algorithm first runs DFS and computes $\text{UP}[u]$ at the highest level up the tree that we can reach from u by taking a back edge. Then we compute $\text{DOWN-AND-UP}(u) = \min_{v \in T_u} \text{UP}(v)$ for all $u \in V$ (see previous problem). Then for each DFS-tree edge (u,v), where v is a child of u, the algorithm declares (u,v) to be a cut edge if $\text{DOWN-AND-UP}(v) > \text{LEVEL}(u)$.
The correctness of the algorithm rests on observations made in the previous point.
The running time is dominated by running DFS and computing LEVEL, UP and DOWN-AND-UP, which can be done in $O(n+m)$ time.

具体讲述了怎么判断 cut edge，D&U(v) > Level(u)

QUIZ 7

Select **all** edges that make up the Shortest Path Tree rooted at D. The edge weights are shown in red.

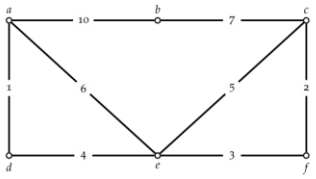




这里在画 shortest path tree 的时候, 红色代表我画错的, 我应该注意一下到底如何画 shortest tree

TUT 8

Problem 2. Consider the following weighted undirected graph G:



Your task is to compute a minimum weight spanning tree T of G:

a) Which edges are part of the MST?

b) In which order does Kruskal's algorithm add these edges to the solution.

c) In which order does Prim's algorithm (starting from a) add these edges to the solution.

Solution 2.

a) $(a,d),(b,c),(c,f),(d,e),(e,f)$

b) The edges are considered in order of their weight, so they are added in the order: $(a,d),(c,f),(e,f),(d,e),(b,c)$. Note that when (c,e) is considered (c,f) and (e,f) are already present, so it isn't added. Similarly, (a,e) and (a,b) aren't added.

c) Prim's algorithm grows the MST from the starting vertex, each time adding the cheapest edge that connects the MST to a new vertex, so the edges are added in the order: $(a,d),(d,e),(e,f),(c,f),(b,c)$.

MST

Problem 3. Let $G = (V, E)$ be an undirected graph with edge weights $w : E \to R^+$. For all $e \in E$, define $w_1(e) = \alpha w(e)$ for some $\alpha > 0$, $w_2(e) = w(e) + \beta$ for some $\beta > 0$, and $w_3(e) = w(e)^2$.

a) Suppose p is a shortest s-t path for the weights w. Is p still optimal under w_1? What about under w_2? What about under w_3?

b) Suppose T is minimum weight spanning tree for the weights w. Is T still optimal under w_1? What about under w_2? What about under w_3?

Solution 3.

a) For w_1 we note that for any two paths p and p' if $w(p) \leq w(p')$ then

$$w_1(p) = \alpha w(p) \leq \alpha w(p') = w_1(p').$$

so if a path is shortest under w it remains shortest under w_1.
This is no longer the case with w_2 and w_3. Consider a graph with three vertices and three edges, basically a cycle. Suppose one edge has weight 1 and the other edges have weight $1/2$. Consider the shortest path between the endpoints of the edge with weight one. The two possible paths have total weight 1, so both of them are shortest. Now imagine adding 1 to all edge weights; that is, consider w_2 with $\beta = 1$. The shortest path in w_2 is unique—the edge with weight 1 in w. Now imagine squaring the edge weights; that is, consider w_3. The shortest path in w_3 is unique—the path with two edges of weight $1/2$, which under w_3 have weight $1/4$ each.

b) We claim that the optimal spanning tree does not change. This is because the relative order of the edge weights is the same under w, w_1, w_2, and w_3. Therefore, if we run Kruskal's algorithm the edges will be sorted in the same way and the output will be the same in all cases.

主要看第二问

Problem 6. Consider the IMPROVING-MST algorithm for the MST problem.

```
1: function IMPROVING-MST(G, w)
2:     T ← some spanning tree of G
3:     for e ∈ E \ T do
4:         T ← T + e
5:         C ← unique cycle in T
6:         f ← heaviest edge in C
7:         T ← T − f
```



```
8:     return T
```

Prove its correctness and analyze its time complexity. To simplify things, you can assume the edge weights are distinct.

Solution 6. Let $T_0, T_1, T_2, \ldots, T_m$ be the trees kept by the algorithm in each of its m iterations. Consider some edge $(u, v) \in E$ that did not make it to the final tree. It could be that (u, v) was rejected right away by the algorithm, or it was in T for some time and then it was removed. Suppose this rejection took place on the ith iteration. Let p be the u-v in path in T_i. Since (u, v) was rejected, all edges in p have weight less than $w(u, v)$. It is easy to show using induction that this is true not only in T_i but in all subsequent trees T_j for $j \geq i$.

Let (x, y) be an edge in the final tree T_m. Remove (x, y) from T_m to get two connected components X and Y. The Cut Property states that if (x, y) is the lightest edge in the cut (X, Y) then every MST contains (x, y). Suppose for the sake of contradiction that there is some rejected edge $(u, v) \in \text{cut}(X, Y)$ such that $w(u, v) < w(x, y)$. This means that (u, v) is not the heaviest edge in the unique u-v path in T_m, which contradicts the conclusion of the previous paragraph. Thus, (x, y) belongs to every MST. Since this holds for every edge in T_m, it follows that T_m itself is an MST.

Regarding the time complexity, finding an initial spanning tree can be done in $O(n + m)$ time by running for example DFS and returning the DFS tree. Next, we note that finding the cycle in $T + e$ can be done $O(n)$ time using DFS. Similarly finding the heaviest edge and removing it takes $O(|C|) = O(n)$ time. There are m iterations, so the overall time complexity is $O(nm)$.

SPT 一定包含所有 n 个 vertex，以及 n-1 个 edges

Cycle property

每一次去除最大边

Problem 7. Consider the REVERSE-MST algorithm for the MST problem.

```
1: function REVERSE-MST(G, w)
2:     Sort edges in decreasing weight w
3:     T ← E
4:     for e ∈ E in decreasing weight do
5:         if T − e is connected then
6:             T ← T − e
7:     return T
```

因为 vertex 和 Edge 有限 遍历

O(m)

DFS check, O(n+m)

O(m²)

Prove its correctness and analyze its time complexity. To simplify things, you can assume the edge weights are distinct.

- 还是 delete 最大 Edge
- Works

Problem 8. A computer network can be modeled as an undirected graph $G = (V, E)$ where vertices represent computers and edges represent physical links between computers. The maximum transmission rate or *bandwidth* varies from link to link; let $b(e)$ be the bandwidth of edge $e \in E$. The bandwidth of a path P is defined as the minimum bandwidth of edges in the path, i.e., $b(P) = \min_{e \in P} b(e)$.

Suppose we number the vertices in $V = \{v_1, v_2, \ldots, v_n\}$. Let $B \in R^{n \times n}$ be a matrix where B_{ij} is the maximum bandwidth between v_i and v_j. Give an algorithm for computing B. Your algorithm should run in $O(n^3)$ time.

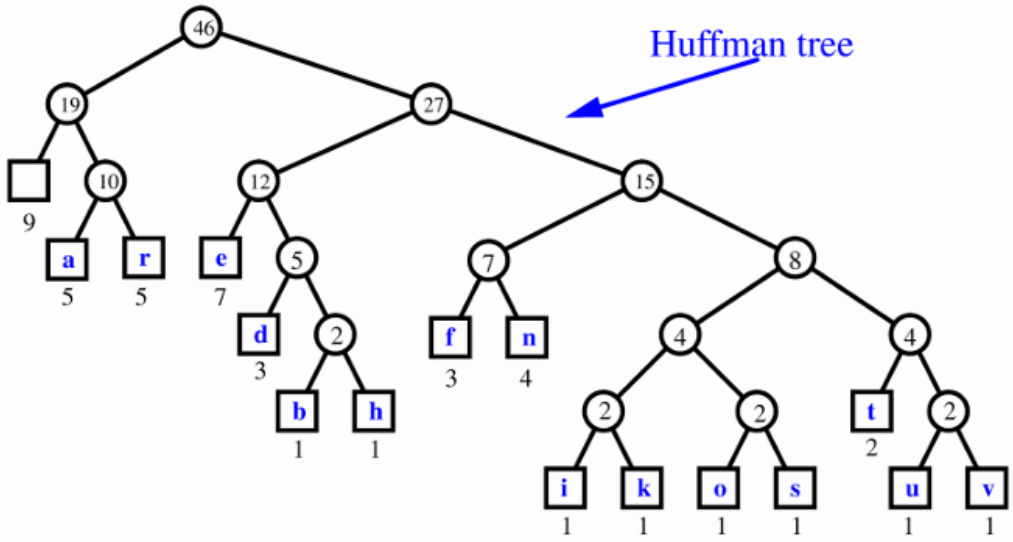
- Since we want to find max bandwidth
- we can just find the max weighted edges instead of min weighted edge in Prim's algorithm
- 我们有 n^2 个 vertices pair (V_i, V_j)
  每个 vertices pair 找到 max weighted path take $O(n)$
- In total $O(n^3)$

→ 不用考虑 (x, y) 和 (y, x) 重复的问题

Lecture 9

String: **a fast runner need never be afraid of the dark**

Character		a	b	d	e	f	h	i	k	n	o	r	s	t	u	v
Frequency	9	5	1	3	7	3	1	1	1	4	1	5	1	2	1	1



TUT 9

Problem 1. Construct the Huffman tree of the following phrase: "data structure"

Solution 1. We first determine the frequency of each character in the phrase (including the space):

character	frequency
a	2
c	1
d	1
e	1
r	2
s	1
t	3
u	2
space	1

After creating a single-node tree for each of these, the Huffman tree algorithm repeatedly merges the two trees with smallest total frequency. Assuming ties are broken lexicographically, it performs the following merges:

1. c and d form a new tree cd with frequency 2
2. e and s form a new tree es with frequency 2
3. space and a form a new tree $a\_$ with frequency 3
4. cd and es form a new tree $cdes$ with frequency 4
5. r and u form a new tree ru with frequency 4
6. $a\_$ and t form a new tree $at\_$ with frequency 6
7. $cdes$ and ru form a new tree $cdersu$ with frequency 8
8. and finally $cdersu$ and $at\_$ form a new tree with frequency 14

The resulting tree can be found below:

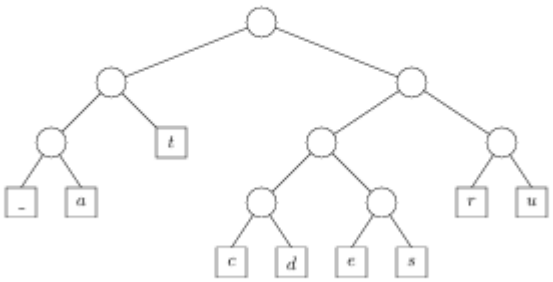


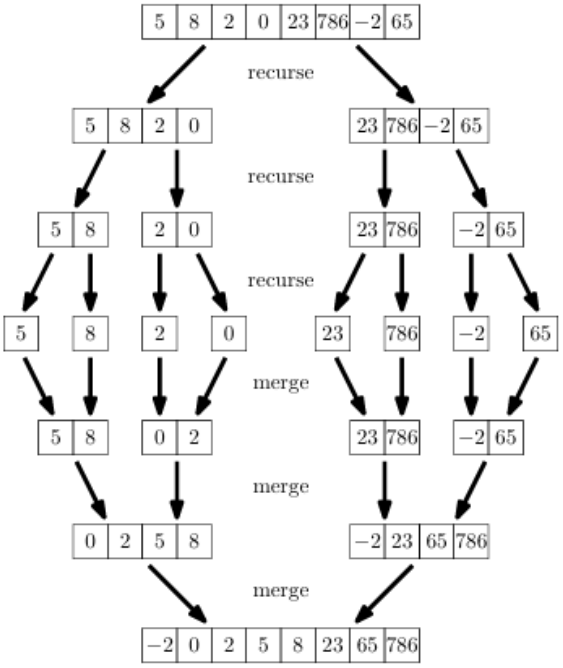
Figure 1: The resulting Huffman tree.

空格也算一个 char

TUT 10

Problem 1. Sort the following array using merge-sort: $A = [5, 8, 2, 0, 23, 786, -2, 65]$. Give all arrays on which recursive calls are made and show how they are merged back together.

Solution 1. Below all arrays are shown. Until all arrays have size 1, all splits represent recursive calls. Since arrays of size 1 are sorted, after this the arrays are merged back together in order to create the final sorted array.



Problem 2. Consider the following algorithm.

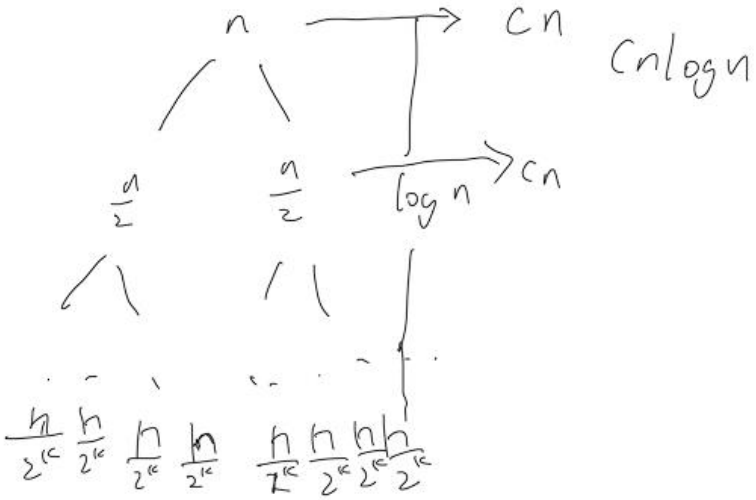
```
1: function REVERSE(A)
2:     if |A| = 1 then
3:         return A
4:     else
5:         B ← first half of A
6:         C ← second half of A
7:         return concatenate REVERSE(C) with REVERSE(B)
```

2 Recu

Let $T(n)$ be the running time of the algorithm on an instance of size n. Write down the recurrence relation for $T(n)$ and solve it by unrolling it.

$$T(n) = \begin{cases} 2T(\frac{n}{2}) + O(n) & n > 1 \\ O(1) & n = 1 \end{cases}$$

Concatenate takes O(n) because we need to create new Array with size n



Problem 3. Given an array A holding n objects, we want to test whether there is a *majority* element; that is, we want to know whether there is an object that appears in more than $n/2$ positions of A.

Assume we can test equality of two objects in $O(1)$ time, but we cannot use a dictionary indexed by the objects. Your task is to design an $O(n \log n)$ time algorithm for solving the majority problem.

a) Show that if x is a majority element in the array then x is a majority element in the first half of the array or the second half of the array

b) Show how to check in $O(n)$ time if a candidate element x is indeed a majority element.

c) Put these observation together to design a divide an conquer algorithm whose running time obeys the recurrence $T(n) = 2T(n/2) + O(n)$

d) Solve the recurrence by unrolling it.

c) If the array has a single element, we return that element as the majority element. Otherwise, we break the input array A into two halves, recursively call the algorithm on each half, and then test in A if either of the elements returned by the recursive calls is indeed a majority element of A. If that is the case we return the majority element, otherwise, we report that there is "no majority element".
To argue the correctness, we see that if there is no majority element of A then the algorithm must return "no majority element" since no matter what the recursive call returns, we always check if an element is indeed a majority element. Otherwise, if there is a majority element x in A we are guaranteed that one of our recursive calls will identify x (by part a)) and the subsequent check will lead the algorithm to return x.
Regarding time complexity, breaking the main problem into the two subproblems and testing the two candidate majority elements can be done in $O(n)$ time. Thus we get the following recurrence

$$T(n) = \begin{cases} T(n) = 2T(n/2) + O(n) & \text{for } n > 1 \\ O(1) & \text{for } n = 1 \end{cases}$$

Majority 可以在 O（n）就算出来，只要在每个 layer 抛弃 half element 就行，即相同的保留一个，不相同的都 discard

Problem 4. Let A be an array with n distinct numbers. We say that two indices $0 \le i < j < n$ form an inversion if $A[i] > A[j]$. Modify merge sort so that it computes the number of inversions of A.

A = [0, 2, 1, 3] A[i] > A[j]
       i j

(1) 每次 merge 的时候，每当 put right array element into sorted list 的 current level inversion += # of elements currently in left side

L(1) > R(1)

不需要，只有在 $R[r] < L[l]$ 的情况下才需要增加逆序对的计数。因为：

1. **已排序的性质**：在合并排序的过程中，左右两个子数组 L 和 R 是各自有序的（分别递归排序得到），所以当 $L[l] \leq R[r]$ 时，不会有新的逆序对产生。
2. **逆序对的定义**：逆序对定义为 $i < j$ 且 $A[i] > A[j]$。因此，当 $R[r] < L[l]$ 时，左边未合并的所有元素 $L[l]$ 及其后的元素都会大于 $R[r]$，因此这些元素都与 $R[r]$ 构成逆序对。
3. **一次性计数**：由于 L 是排序的，所以在 $R[r] < L[l]$ 时，剩余的 L 中元素的数量 $|L| - l$ 正是与 $R[r]$ 构成逆序对的数量。因此，只需在这种情况下累加 $|L| - l$ 到逆序对计数即可。

总结来说，只有当 $R[r] < L[l]$ 时，我们才会增加逆序对计数，而这种情况已经涵盖了所有可能的逆序对，所以不需要处理其他情况。

TUT 11

Problem 1. The product of two $n \times n$ matrices X and Y is a third $n \times n$ matrix $Z = XY$, where the (i, j) entry of Z is $Z_{ij} = \sum_{k=1}^{n} X_{ik} Y_{kj}$. Suppose that X and Y are divided into four $n/2 \times n/2$ blocks each:

$$X = \begin{bmatrix} A & B \\ C & D \end{bmatrix} \text{ and } Y = \begin{bmatrix} E & F \\ G & H \end{bmatrix}.$$

Using this block notation we can express the product of X and Y as follows

$$XY = \begin{bmatrix} AE + BG & AF + BH \\ CE + DG & CF + DH \end{bmatrix}.$$

In this way, one multiplication of $n \times n$ matrices can be expressed in terms of 8 multiplications and 4 additions that involve $n/2 \times n/2$ matrices. Let $T(n)$ be the time complexity of multiplying two $n \times n$ matrices using this recursive algorithm.

a) Derive the recurrence for $T(n)$. (Assume adding two $k \times k$ matrices takes $O(k^2)$ time.)

b) Solve the recurrence by unrolling it.

(a) $T(n) = 8T\left(\frac{n}{2}\right) + 4O(k^2)$ i.e. $k = n$

$a = 8 \quad b = 2 \quad \log_b a = 3$

$f(n) = 4O(k^2) = n^2$

Problem 4. Suppose we are given an array A with n distinct numbers. We say an index i is locally optimal if $A[i] < A[i-1]$ and $A[i] < A[i+1]$ for $0 < i < n-1$, or $A[i] < A[i+1]$ for if $i = 0$, or $A[i] < A[i-1]$ for $i = n-1$.

Design an algorithm for finding a locally optimal index using divide and conquer. Your algorithm should run in $O(\log n)$ time.

Solution 4. First we test whether $i = 0$ or $i = n-1$ are locally optimal entries. Otherwise, we know that $A[0] > A[1]$ and $A[n-2] < A[n-1]$. If $n \leq 4$, it is easy to see that either $i = 1$ or $i = 2$ is locally optimal and we can check that in $O(1)$ time.

Otherwise, pick the middle position in the array (for example $i = \lfloor n/2 \rfloor$) and test whether i is locally optimal. If it is we are done, otherwise $A[i-1] < A[i]$ or $A[i] > A[i+1]$; in the former case we recur on $A[0, \ldots, i]$ and in the latter we recur on $A[i, \ldots, n-1]$. Either way, we reduce the size of the array by half and we maintain that the half that we recurse on contains a locally optimal index (this can be argued similarly to how we argued that binary search recurses on the half that contains the element that we're searching for, if it exists).

The above invariant immediately implies the correctness of our algorithm, since if the half we recurse on contains the index we're searching for and our array gets smaller with each recursive call, we eventually end up in the base case, which is easily checked. Note that our recursion could end earlier, if the middle element happens to be locally optimal, in which case we also found what we were looking for.

Rather than creating a new array each time we recur (which would take $O(n)$ time) we can keep the working subarray implicitly by maintaining a pair of indices (b, e) telling us

that we are working with the array $A[b, \ldots, e]$. Thus, each call demands $O(1)$ work plus the work done by recursive calls. This leads to the following recurrence,

$$T(n) = \begin{cases} T(n/2) + O(1) & \text{for } n > 1 \\ O(1) & \text{for } n = 1 \end{cases}$$

which solves to $T(n) = O(\log n)$.

Solution 6. Recall that the Master Theorem applies to recurrences of the following form: $T(n) = aT(n/b) + f(n)$ where $a \geq 1$ and $b > 1$ are constants and $f(n)$ is an asymptotically positive function. There are three cases:

1. If $f(n) = O(n^{\log_b a - \varepsilon})$ for some constant $\varepsilon > 0$, then $T(n) = \Theta(n^{\log_b a})$.
2. If $f(n) = \Theta(n^{\log_b a} \log^k n)$ with $k \geq 0$, then $T(n) = \Theta(n^{\log_b a} \log^{k+1} n)$.
3. If $f(n) = \Omega(n^{\log_b a + \varepsilon})$ with $\varepsilon > 0$, and $f(n)$ satisfies the regularity condition, then $T(n) = \Theta(f(n))$. Regularity condition: $af(n/b) \leq cf(n)$ for some constant $c < 1$ and all sufficiently large n (this condition will be satisfied for all recurrences you'll encounter during this unit and we'll ignore it).

Also remember that if $f(n)$ is given in big-O notation, we can only use the above to conclude an upper bound on $T(n)$ in big-O notation and not the Θ used in the theorem. This is because Θ implies that we have both an upper and lower bound on the running time and big-O only gives us an upper bound on $f(n)$.

a) $T(n) = 4T(n/2) + O(n^2) \rightarrow T(n) = O(n^2 \log n)$ (Case 2 with $a = 4$, $b = 2$ and $f(n) = n^2$)

b) $T(n) = T(n/2) + O(2^n) \rightarrow O(2^n)$ (Case 3 with $a = 1$, $b = 2$ and $f(n) = 2^n$)

c) $T(n) = 16T(n/4) + O(n) \rightarrow T(n) = O(n^2)$ (Case 1 with $a = 16$, $b = 4$ and $f(n) = n$)

d) $T(n) = 2T(n/2) + O(n \log n) \rightarrow T(n) = O(n \log^2 n)$ (Case 2 with $a = 2$, $b = 2$ and $f(n) = n \log n$)

e) $T(n) = \sqrt{2}T(n/2) + O(\log n) \rightarrow T(n) = O(\sqrt{n})$ (Case 1 with $a = \sqrt{2}$, $b = 2$ and $f(n) = \log n$)

f) $T(n) = 3T(n/2) + O(n) \rightarrow T(n) = O(n^{\log 3})$ (Case 1 with $a = 3$, $b = 2$ and $f(n) = n$)

g) $T(n) = 3T(n/3) + O(\sqrt{n}) \rightarrow T(n) = O(n)$ (Case 1 with $a = 3$, $b = 3$ and $f(n) = \sqrt{n}$)