1. (a) (5)^{i} 4 7 3 10 2 6 8 (1)^{j} [2 comparisons]

[1 comparison to check for break] | swap ↓

1 4 (7)^{i} 3 10 (2)^{j} 6 8 5 [7 comparisons]

[1 comparison] | swap ↓

1 4 2 (3)^{j} (10)^{i} 7 6 8 5 [6 comparisons]

[1 comparison]

(b) 18 comparisons, 2 swaps

(c) 4 − 1 = 3

(d) After recursive calls have returned:

x

(3) 4 5 7 10 | y

(1) 2 6 8

|

Final merge:

1	2	3	4	5	6	7	8	10
(1<3)	(2<3)	(3<6)	(4<6)	(5<6)	(6<7)	(7<8)	(8<10)	

Total comparisons = 8

Checks on x and y cursors in main while loop: 9 × 2 = 18

Checks on x and y in final two while loops: 2 + 1 = 3

⇒ Sum total comparisons are 29.

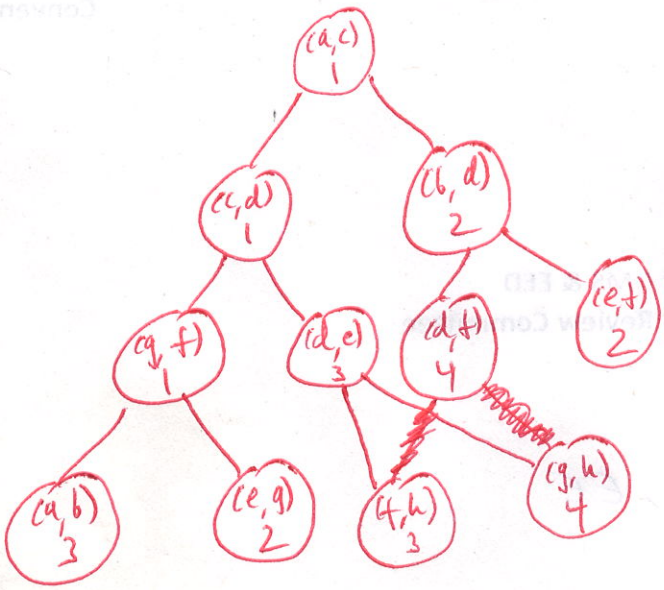
If we include the for loop to copy values back to main array: 10 more

⇒ Total comparisons in merge() fn. = 39.

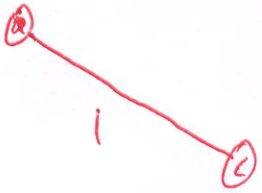
(e) Quicksort does appear more efficient in terms of no. of comparisons, as expected. Quicksort has 2 additional swap operations, but probably still faster. And note that mergesort may also have greater memory access costs: due to merging into a new array, and then moving elements back to original array. So overall, quicksort definitely seems faster.

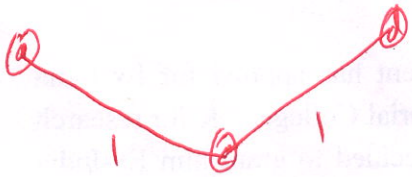
2. (a) Consider, e.g., $U = \{a, b, c\}$. Then (c, d) is lowest-cost edge from U to $V - U$. So must be in an MST.

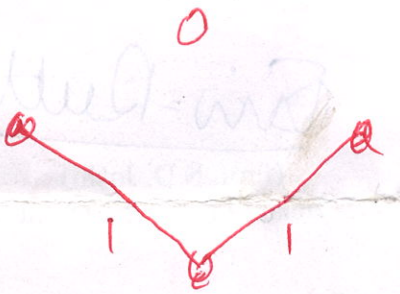
(b)



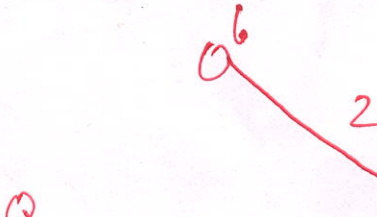
[any valid min-heap with all edges is OK]


a
1
c

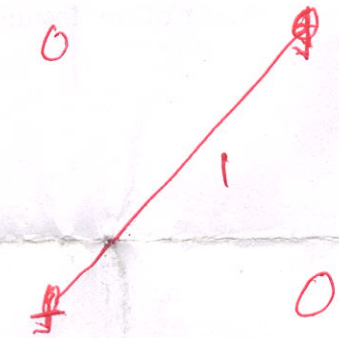

a
1
c
1
d

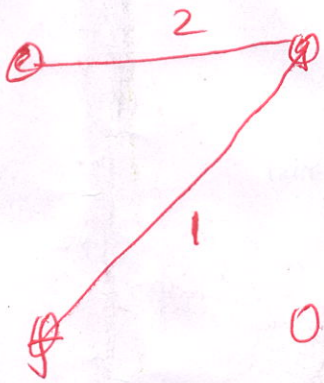

1
1
1

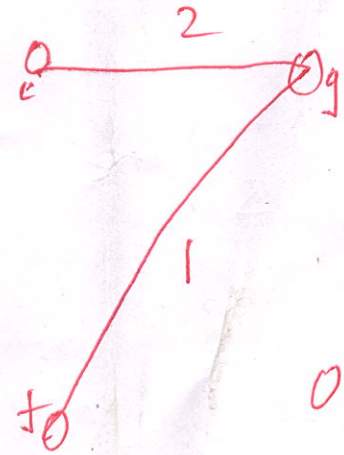

a
1
c
1
d

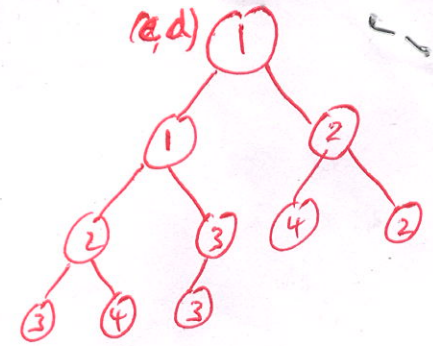

a
1
c
1
d

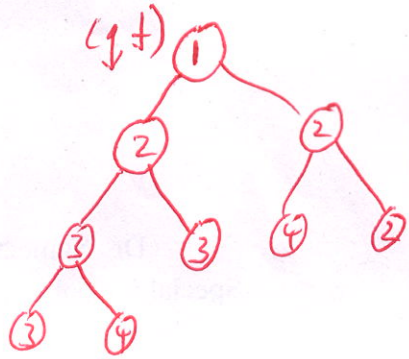
b
2
a
1
c
1
d

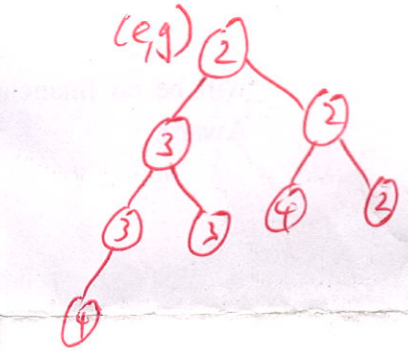

1
f

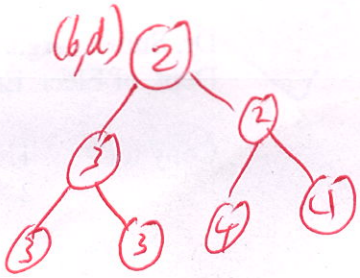

2
e
1
g

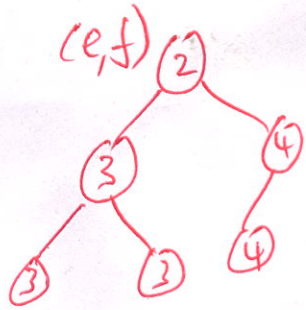

2
e
g
1
f

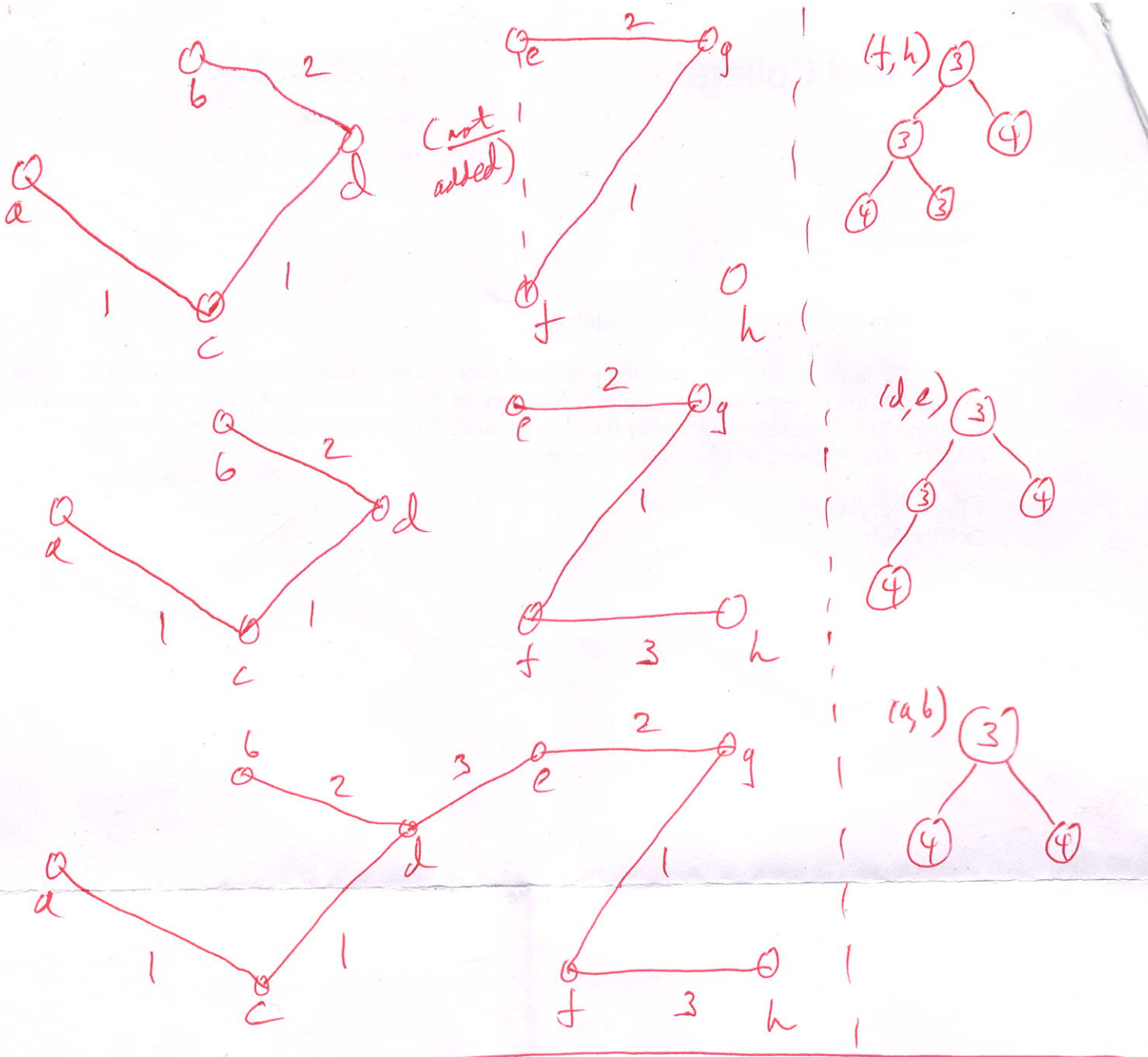

(a,d)


(b,f)


(e,g)


(b,d)


(e,f)



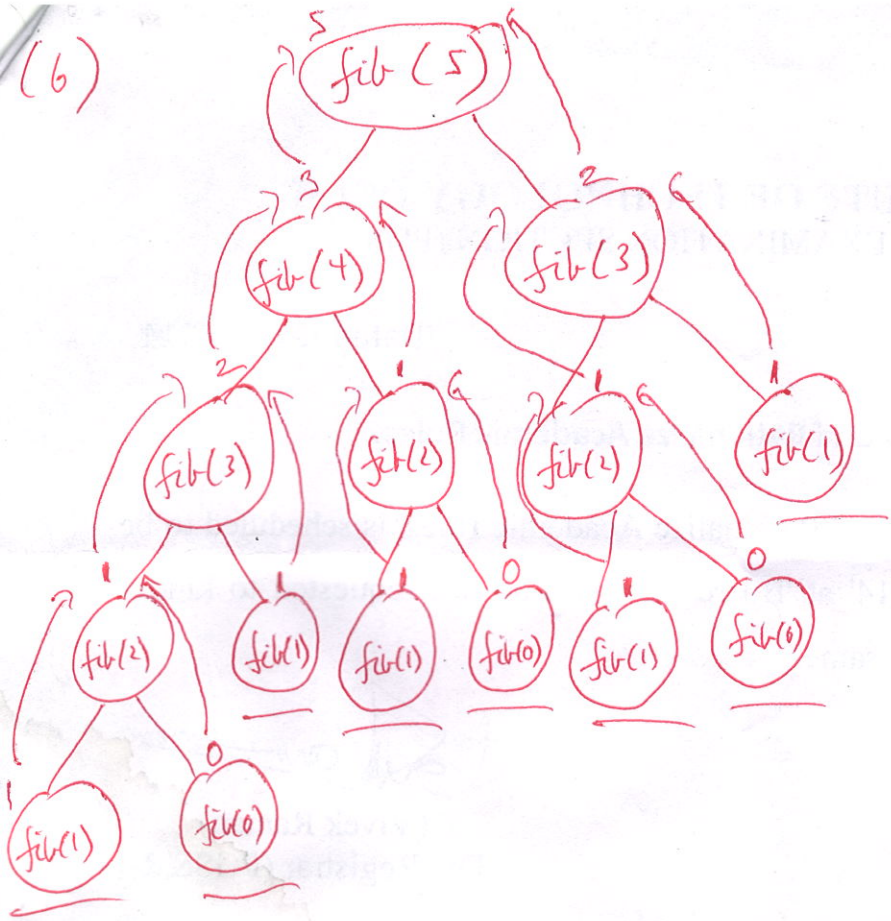
3. (a)

```
int fib (int n) {
    if (n == 0) return 0;
    if (n == 1) return 1;
    return fib(n-1) + fib(n-2);
}
```

$$T(n) = \begin{cases} T(n-1) + T(n-2) + d & ; \quad n \geq 2 \\ c & ; \quad n < 2 \end{cases}$$

Since prob. size is reducing linearly, recurrence tree has $O(n)$ depth, and hence $O(2^n)$ leaves. So $T(n)$ is $O(2^n)$.

(b)



(c) NO; repeated computation of subproblems.

(d) Dynamic Programming:

```
int fib (int n) {
    O(1)    int *f = malloc((n+1)* sizeof (int));
    O(n)    for ( int i=0; i<=n; i++) {
    O(1)        if (i<2) f[i] = i;
    O(1)        else f[i] = f[i-1]+ f[i-2];
            }
    O(1)    return f[n];
}
```

O(n)

fib(5):