a) $T(n) = T(n-1) + c$

$T(n-1) = T(n-2) + c$

$T(n) = T(n-2) + c + c$

Constant amount of work with n iterations so it makes sense that its a linear amount of work total

So for iteration k

$T(n) = T(n-k) + kc$

When $n = 1$ $T(1) = c$

So recurrence stops at $k = n-1 \Rightarrow T(n) = T(n-(n-1)) + (n-1)c$

$$= T(1) + cn - c$$
$$= cn + c - c = cn$$
$$= O(n)$$

b) $T(n) = T(n-1) + n$, work per iteration linear in input size, with linear number of iterations as n decreases by 1 each time. So we probably guess its $O(n^2)$

$T(n-1) = T(n-2) + (n-1)$

$T(n-2) = T(n-3) + (n-2)$

So $T(n) = n + T(n-1)$

$$= n + (n-1) + T(n-2)$$
$$= n + (n-1) + (n-2) + T(n-3)$$
$$= n + (n-1) + \cdots + (n-(k-1)) + T(n-k) \text{ at iter } k$$

recurrence stops at $(n=1)$ or rather $k = n-1$

We conclude $T(n) = n + (n-1) + \cdots + 2 + T(1)$

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

So $T(n) = O(n^2)$ as expected

c) $T(n) = T(n/2) + c$, Constant amount of work per iteration and input is halfed each iteration so we expect a logarithmic number of iterations.

(to see why ask how many times we can half n before we reach the base case.)

$$T(n/2) = T(n/4) + c = T(n/2^2) + c$$
$$T(n/4) = T(n/8) + c = T(n/2^3) + c$$

So $T(n) = T(n/2) + c$

$$= T(n/2^2) + 2c$$
$$= T(n/2^3) + 3c$$
$$= T(n/2^k) + kc$$

Recursion stops when $n/2^k = 1$

$$\Rightarrow n = 2^k$$

or $\log n = k$

We conclude $T(n) = T\left(n/2^{\log n}\right) + \log n \cdot c = T(1) + c\log n$

$$= c + c\log n$$
$$= O(\log n)$$

So $T(n) = O(\log n)$

$\not{d}$) $T(n) = 2T(n/2) + C$, Constant work for each call but 2 recursive calls. What do we think? logarithmic or linearithmic.

$$T(n/2) = 2T(n/4) + C$$
$$T(n/4) = 2T(n/8) + C$$
$$T(n/2^k) = 2T(n/2^{k}) + C$$

$$T(n) = 2T(n/2) + C$$
$$= 2[2T(n/4) + C] + C$$
$$= 2[2(2T(n/8) + C) + C] + C$$
$$= 8T(n/8) + 6C + C$$
$$= 2^k T(n/2^k) + (2^k - 1)C$$

Recursion stops at $k = \log n$ so we conclude

$$T(n) = 2^{\log n} T(1) + (2^{\log n} - 1)C$$
$$= nT(1) + (n-1)C$$
$$= nC + nC - C = O(n)$$

d.2) $T(n) = 2T(n/2) + n$ merge sort recurrence.

$$= 2[2T(n/4) + (n/2)] + n$$
$$= 4T(n/4) + n + n$$
$$= 4[2T(n/8) + n/4] + 2n$$
$$= 8T(n/8) + n + 2n$$
$$= 2^k T(n/2^k) + kn$$

Recursion stops when $k = \log n$. we conclude

$$T(n) = 2^{\log n} T(1) + n\log n$$
$$= nT(1) + n\log n$$
$$= cn + n\log n = O(n \log n)$$ as expected.

e) $T(n) = 2T(n-1) + C$, This could be very bad.

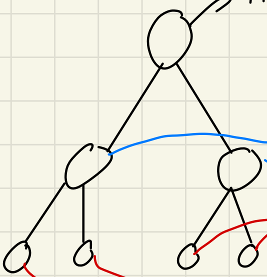
$$T(n) = 2T(n-1) + C \qquad , T(n-1) = 2T(n-2) + C$$
$$= 4T(n-2) + 3C$$
$$= 8T(n-3) + 7C$$
$$= 2^k T(n-k) + (2^k - 1)C$$

Recurrence stops when $k = n-1$ we conclude

$$T(n) = 2^{n-1} T(1) + (2^{n-1} - 1)C$$
$$= 2^{n-1}C + 2^{n-1}C - C = O(2^n).$$ Exponential runtime very bad!

d* revisited) Lets reconsider the recurrence $T(n) = 2T(n/2) + c$ from a different perspective. Why is it linear? Lets consider its recursion tree.

This represents the root call $T(n)$



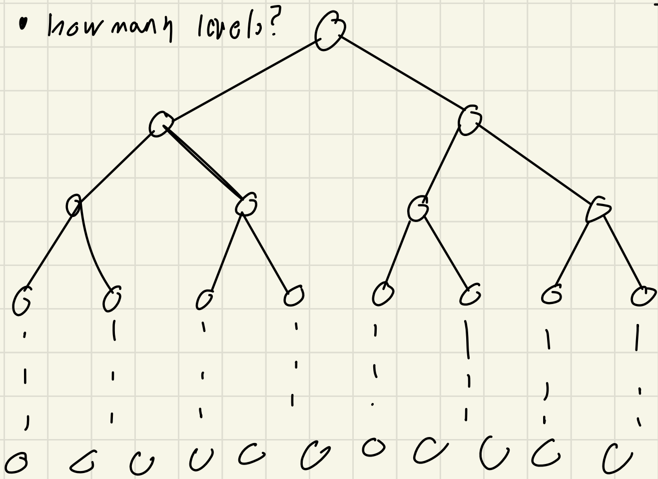
By the recurrence relation it makes 2 more child calls.

$$T(n) = T(n/2) + T(n/2) + c$$

$$T(n) = [T(n/4) + T(n/4) + c] + [T(n/4) + T(n/4) + c] + c$$

And the process continues. Lets draw the full tree. Each node in the tree represents a recursive call with cost c. So if we can count how manys nodes are in our tree we can determine how much work our algorithm does for an input of size n.

We need to answer
- how many nodes at each level?
- how many levels?



Level	Cost
1	c
2	$2c$
3	$4c$
4	$8c$
⋮	
$\log n$	$2^{\log n} \cdot c$

By the analysis of our recurrence we observed that recursion stops when $k = \log n$. So our tree has $\log n$ levels. Since every node at the previous has 2 children at the next level we always have twice the number of nodes at the next level as we do in the previous.

Level	1	2	3	4	5	6	. . . .	i
# Nodes	1	2	4	8	16	32	. . . . .	2^{i-1}

For every node in our tree we have a constant c amount of work. So summing c(# nodes at level i) over i from $1, \ldots, \log n$ we obtain to work or runtime of our algorithm.

$$\sum_{i=1}^{\log n} c \cdot 2^{i-1} = c \sum_{i=0}^{\log n - 1} 2^i = c\left(2^{\log n} - 1\right) = c(n-1) = O(n)$$

Hence our algorithm is linear.