1.2-2) insertion sort $= \Theta(8n^2)$
merge sort $= \Theta(64n \lg(n))$

$$\frac{8n^2}{8n} < \frac{64n \lg n}{8n}$$

$$n < 8 \lg n$$

$$n - 8 \lg n < 0$$

$$\boxed{1 < n < 43}$$

1.2-3) $\Theta(100n^2)$ vs $\Theta(2^n)$

$$100n^2 < 2^n$$

$$\lg 100 + 2\lg(n) < n$$

$$\lg(100) + 2\lg n - n < 0$$

$$\boxed{n \geq 15}$$

$n = 15$ is when $100n^2$ becomes faster than 2^n

PROBLEM 1-1

formula: # of instruction = time (in secs) · 10^6 instructions per second

$2^{\lg n} = 1 \cdot 10^6$

$n = 2^{10^6}$

$\sqrt{n^2} = (1 \cdot 10^6)^2$

$n = 10^{12}$

$n = 10^6$

$n \lg n = 10^6$

$n_0 = \frac{10^6}{\lg 1000000} \approx 50172$

$n_1 = \frac{10^6}{\lg 50171} \approx 64043$

$n_2 = \frac{10^6}{\lg 64043} \approx 62630$

$n_3 = \frac{10^6}{\lg 62630} \approx 62756$

$n_4 = \frac{10^6}{\lg 62756} \approx 62745$

$n_5 = \frac{10^6}{\lg 62745} \approx 62746$

$n_5 = \frac{10^6}{\lg 62746} \approx \boxed{62,746}$

$\sqrt{n^2} = \sqrt{10^6}$

$n = 10^3 = 1000$

$\sqrt[3]{n^3} = \sqrt[3]{10^6}$

$n = 10^2 = 100$

$2^n = 10^6$

$n = \lg 1000000$

$n \approx 19$

$n! = 10^6$

$9! = 362880$

$n \approx 9$

$\frac{1,000,000 \text{ instructions}}{1 \text{ second}} \times \frac{60 \text{ sec}}{1 \text{ min}} \times \frac{60 \text{ min}}{1 \text{ hr}} \times \frac{24 \text{ hours}}{1 \text{ day}} \times \frac{30 \text{ days}}{1 \text{ month}} \times \frac{12 \text{ mon}}{1 \text{ year}} \times \frac{100 \text{ year}}{1 \text{ century}}$

	1 second	1 minute	1 hour	1 day	1 month	1 year	1 century
$\lg n$	2^{10^6}	$2^{6 \cdot 10^7}$	$2^{3.6 \cdot 10^9}$	$2^{8.64 \cdot 10^{10}}$	$2^{2.592 \cdot 10^{12}}$	$2^{3.1536 \cdot 10^{13}}$	$2^{3.1536 \cdot 10^{15}}$
$\sqrt{n}$	10^{12}	3.6×10^{15}	1.296×10^{19}	7.465×10^{20}	6.718×10^{24}	9.945×10^{26}	9.945×10^{30}
n	10^6	6×10^7	3.6×10^9	8.64×10^{10}	2.592×10^{12}	3.1536×10^{13}	3.1536×10^{15}
$n \lg n$	62,746	2.8×10^6	1.334×10^8	2.755×10^9	7.187×10^{10}	7.976×10^{11}	6.861×10^{13}
n^2	1000	7745	66600	293938	1609968	5615692	56156922
n^3	100	391	1532	4420	13736	31593	146645
2^n	19	25	31	36	41	44	51
$n!$	9	11	12	13	15	16	17

2.1-1)

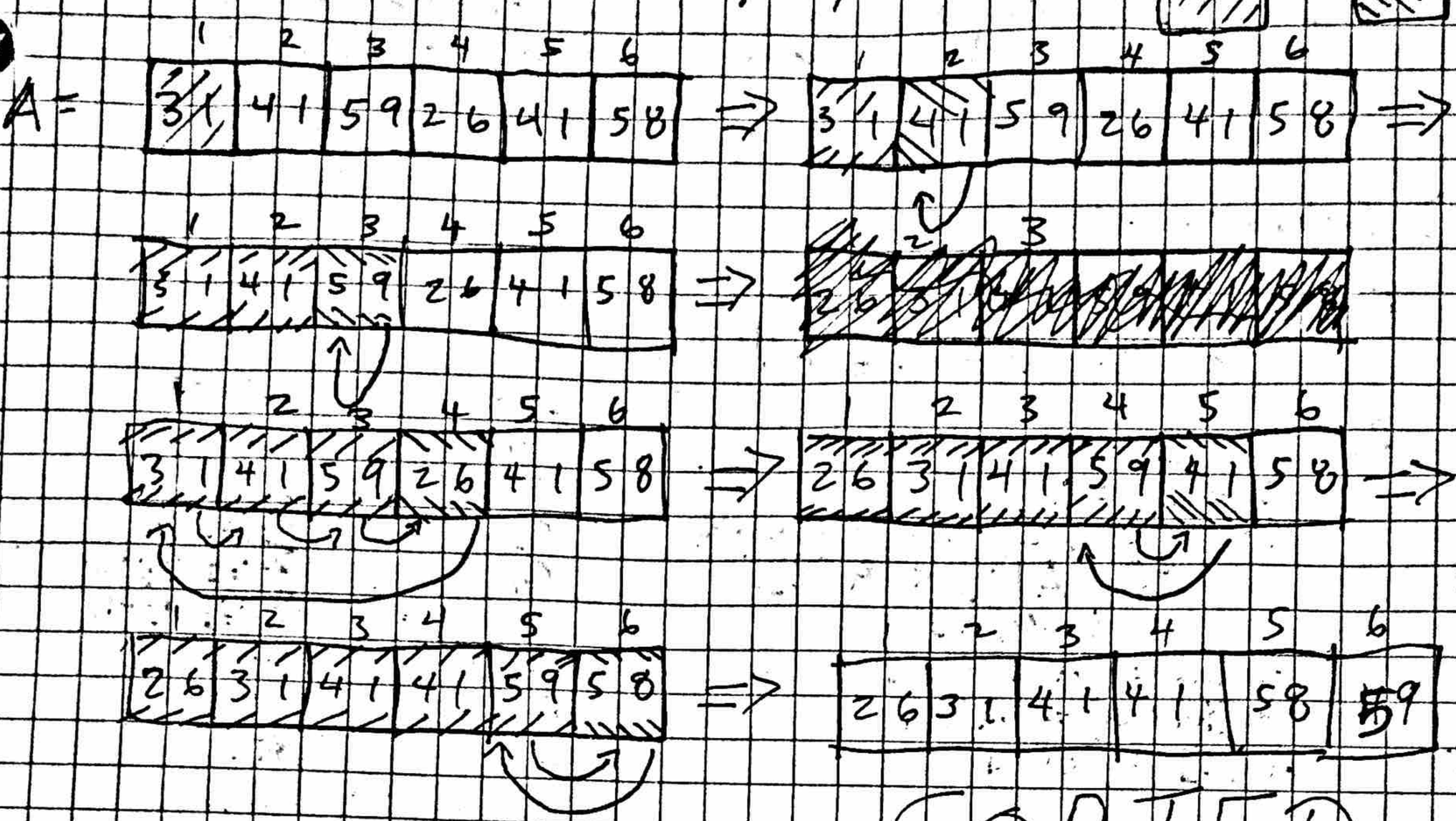
Insertion sort

A = ⟨3, 41, 59, 26, 41, 58⟩

Key

already sorted

where key needs to be placed



2.1-3)

```
linear search (A, v)
1  for i = 2 to A.length
2      if (A[i] = v)
3          return i
4  return nil
```

Loop Invariant

Initialization: from the start of the loop when i = 1, if A[1] was equal to v, then it would return i, (the index of v), this holds true prior to the first iteration

Maintence: maintains the loop invarient because if A[i] ≠ v then the loop will increment otherwise if A[i] = v then we will exit the loop and return the index (line 3)

Termination: The condition causing the for loop to terminate is when i > A.length. The loop terminates when v is not found in that case we will return nil (line 4). We can conclude that if v is found in A then we will return the index otherwise return nil. Hence, the algorithm is correct

2.)

2.2-1) $T(n) = \frac{n^3}{1000} - 100n^2 + 3$

$\boxed{\Theta(n^3)}$ remove all lower order terms and remove leading coefficients.

2.2-2) Selection Sort

		cost	times
1	for $i = 1$ to $A.length - 1$	c_1	n
2	index = i	c_2	$n-1$
3	for $j = i+1$ to $A.length$	c_3	$\sum_{j=2}^{n} t_j$
4	if $(A[j] < A[index])$	c_4	$\sum_{j=2}^{n} (t_j - 1)$
5	index = j	c_5	$\sum_{j=2}^{n} (t_j - 1)$
6	temp = A[index]	c_6	$n-1$
7	A[index] = A[i]	c_7	$n-1$
8	A[i] = temp	c_8	$n-1$

Loop Invariant

Initialization: When $i = 1$ at the start of the main loop if nothing were to change then temp would be equa to A[i] in which nothing would be swapped if A[i] was the smallest this holds true prior to first iteration.

Maintain: i is used to figure out which data needs to be stored (next lowest in Array). The inner loop (Lines 3-5) traverses each index after the index variable to the end of the array to check for the smallest variable to switch with A[i]. Lines (6-8) is used to switch variables and properly places one variable in the correct location maintaining the loop through iterations.

Termination: The loop terminates when i reaches $A.length - 1$ ($n-1$) because if the last location doesn't needs to be sorted because if it was switched with the second to last location then the last location is sorted. At this point we can conclude that the list is now sort from smallest to largest and the algorithm is correct.

$$T(n) = c_1 n + c_2 (n-1) + c_3 \sum_{j=1}^{n} t_j + c_4 \sum_{j=2}^{n} (t_j - 1) + c_5 \sum_{j=2}^{n} (t_j - 1) + c_6 (n-1) + c_7 (n-1) + c_8 (n-1)$$

$$T(n) = c_1 n + (c_2 + c_6 + c_7 + c_8)(n-1) + c_3 \frac{n(n+1)}{2} - 1 + (c_4 + c_5)\left(\frac{n(n-1)}{2}\right)$$

$$T(n) = an^2 + bn + c$$

Best Case: $\Theta(n^2)$ Worst Case: $\Theta(n^2)$

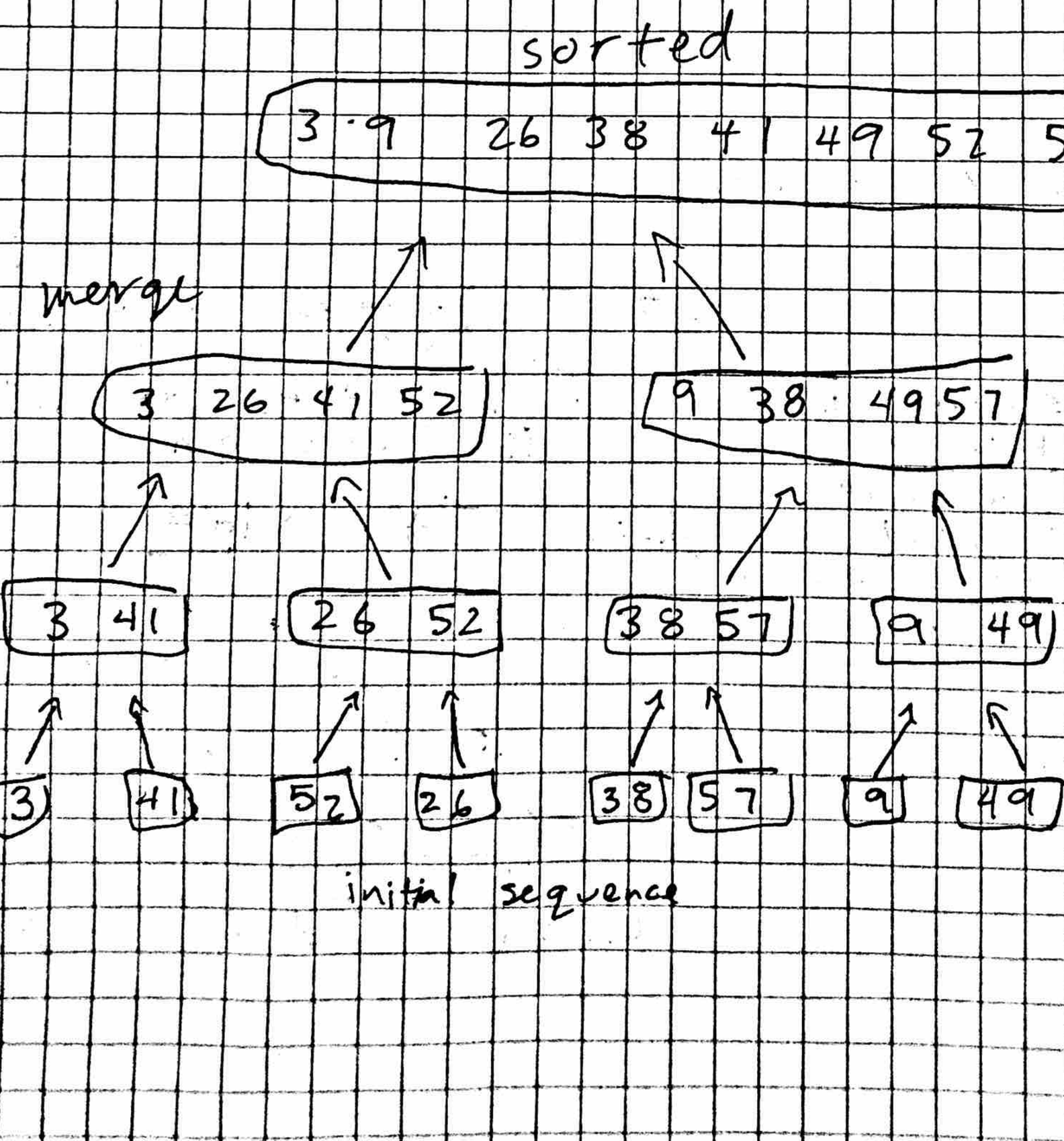
2.2-3)

```
linear search (A, v)                    cost        times
1 for i = 1 → A.length                  c1          n
2     if v = A[i]                       c2          n-1
3         return i                      c3          1
4 return nil                            c4          1
```

One average about half of the array has to be traversed through so $\Theta = (\frac{n}{2})$ but we must remove leading coefficients so average case is $\Theta(n)$.

The worst case results from traversing through the entire list and not finding v or v being in the last position of the array. The worst case therefore is also $\Theta(n)$.

2.3-1)



2.3-3)

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

is $T(n) = n \lg n$ Prove $T(n) \le c \lg n$ for some $c > 0$

Base Case: $T(2) = 2 \lg 2 = 2$

Inductive Hypothesis: $T(\lfloor n/2 \rfloor) \le c \lfloor n/2 \rfloor \lg(\lfloor n/2 \rfloor)$

$$T(n) = 2T(\lfloor n/2 \rfloor) + n$$
$$\le 2(c \lfloor n/2 \rfloor \lg(\lfloor n/2 \rfloor)) + n$$
$$\le cn \lg(n/2) + n$$
$$= cn \lg n - cn \lg 2 + n$$
$$= cn \lg n - cn + n$$

$T(n) \le cn \lg n$ if $c \ge 1$

Q.E.D Inductive step

2.3-5)

Binary Search(A, min, max, value)

```
low = min
high = max
while (low ≤ high) {
    mid = (low+high)/2
    if (A[mid] > value)
        high = mid - 1
    else if (A[mid] < value)
        low = mid + 1
    else
        return mid
return low
```

Let $T(n)$ = # of comparisons

total # of levels of recursion tree

$$T(n) = \begin{cases} 0 & \text{if } n=1 \\ T(\frac{n}{2}) + 1 & \text{for } n > 1 \end{cases}$$

is $T(n) = \lg n$

I.H.: $T(\frac{n}{2}) \le c \lg(\frac{n}{2})$

$$T(n) = T(\tfrac{n}{2}) + 1$$
$$\le c \lg n - c \lg 2 + 1$$
$$\le c \lg n - c + 1$$

$T(n) < c \lg n$

$\Theta(\lg n)$

Q.E.D Inductive step

2.3-6

Binary Insertion Sort (A)

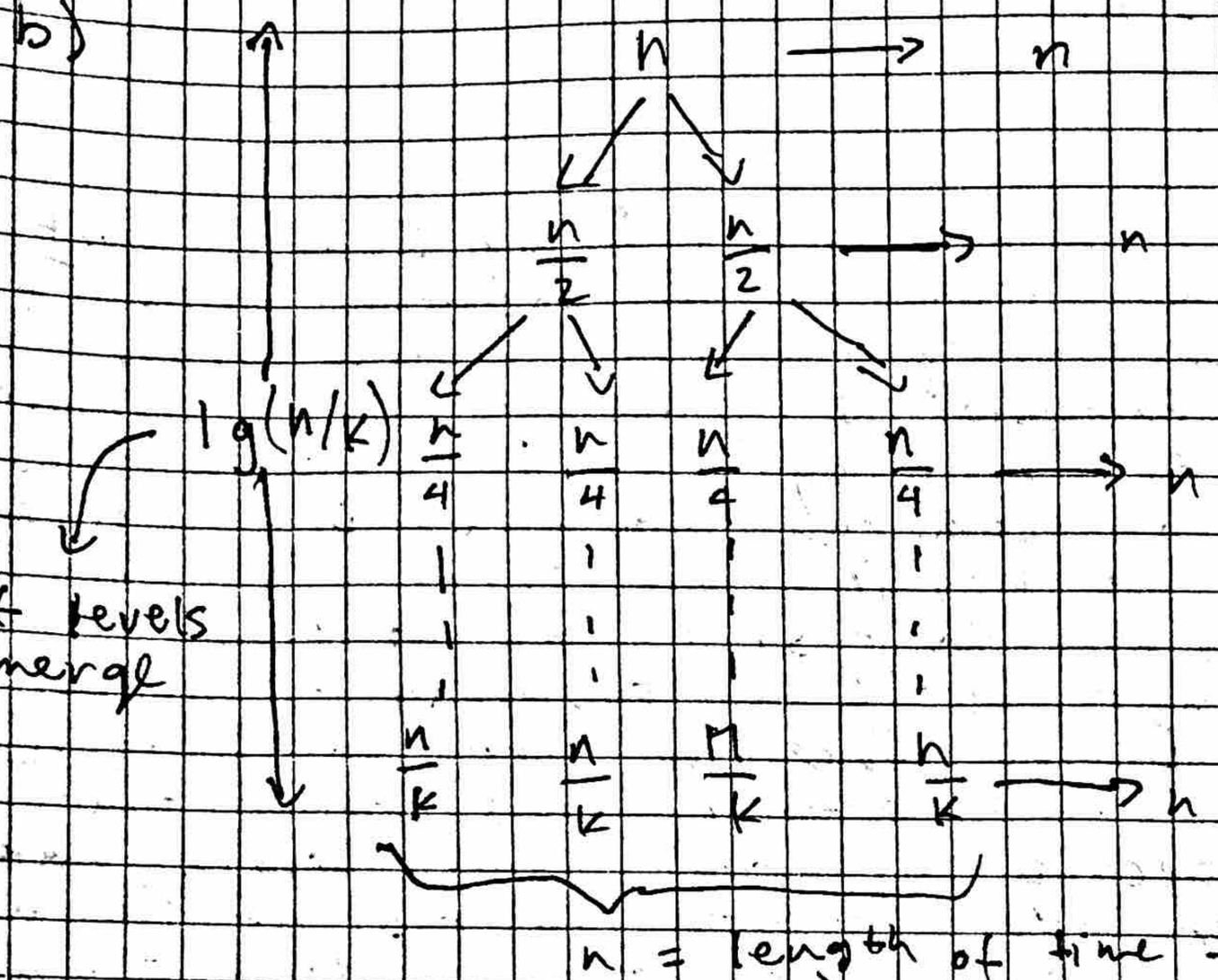
```
1  for j = 2 + A.length
2      key = A[j]
3      i = j - 1
4      index = binarySearch (A, 1, i, key) ← method 2.3-5
5      while (i >= index)
6          A[i+1] = A[i]
7          i = i - 1
8      A[i+1] = key
```

The worst-case running time of Binary Insertion Sort does not improve from regular Insertion Sort. If the list we are trying to sort is in reverse order then each time the key is placed at the beginning of the sorted subarray A[1..j-1] ~~meaning~~ lines (5-7) has to shift the rest of the array down by one in order to properly insert the key in the correct location. Since the Binary Insertion Sort still has to traverse the entire array and shift the subarray to properly place the key, Binary Insertion Sort takes $\Theta(n^2)$. Binary Insertion Sort actully takes longer than regular insertion sort because although Binary Search makes finding the proper index faster, the rest of the array to the right has to still be shifted one down the array. The Binary Search is just an extra step that makes insertion sort slightly slower.

Problem 2-1

a) The length of sublists K will take $\Theta(k^2)$ to use Insertion Sort on one sublist.
If n/k sublists need to be sorted
then $\Theta(k^2) * (n/k) = \Theta(nk)$ in the worst case.

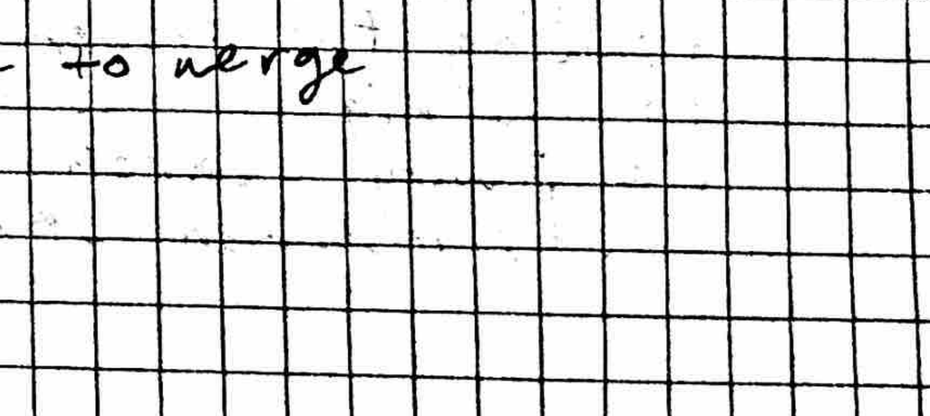
b)



n = length of time to merge
$\Theta(n \lg(n/k))$

c) same running time as merge sort.
$\Theta(nk + n \lg(n/k)) = \Theta(n \lg n)$

Use the nk equation
because if k → $nk = \Theta(n \lg n)$
is a constant
then $n > n \lg n$ $k = \Theta(\lg n)$
for large n. k can't be larger lg n



d) for some cost c, being a scalar factor multiplied to each Θ equation that is an added cost when writing the program. Then you'll make your program the most efficient for $c(nk + n \lg(n/k)) < < (n \lg n)$