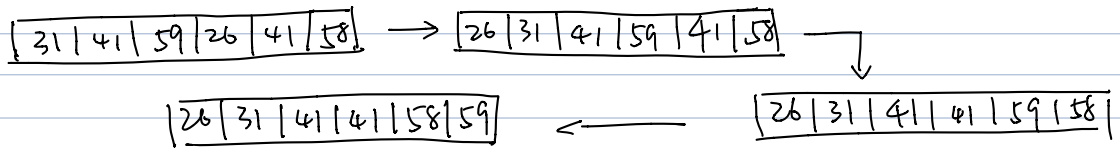
2.1-1

$A = \langle 31, 41, 59, 26, 41, 58 \rangle$



2.1-2

Change line 5 to while i > 0 and A[i] < key

2.1-3

```
LINEAR-SEARCH (A, v)
    for i = 1 to A.length
        if A[i] = v then return i
    return NIL
```

Loop invariant: for a certain i, no k < i s.t. A[k] = v

Initialization: i = 1, no k exist, ✓

Maintenance: Suppose no k < i, s.t. A[k] = v.
if A[i] = k, we directly return, ✓
o.w. enter next loop, we know A[k], A[i] ≠ v
which keeps loop invariant. ✓

Termination: would terminate if found.
o.w. By Li. at last i = n+1, and no k ∈ [1,n] s.t. A[k] = v, we return NIL ✓

2.1-4

Input: two arrays A, B of length n. (binary array)
Output: one array C of length n+1. (binary array)

```
ADD-TWO-NUMBERS (A, B)
    Carry = 0
    for i = A.length to 1
        tmp = A[i] + B[i] + Carry
        C[i+1] = tmp % 2
        carry = tmp // 2
    C[1] = Carry
    return C
```

2.2-1

$O(n^3)$

2.2-2

```
SELECTION-SORT (A)
    for i = 1 to A.length - 1
        minimum = A[i], min_index = i
        for j = i+1 to A.length
            if A[j] < minimum
                minimum = A[j], min_index = j
        A[i], A[min_index] = A[min_index], A[i]
```

Loop variant:
i th round, we have A[1 .. i-1] sorted and are exactly the i-1 st smallest elements

Because the last element would have to be there (naturally the largest)

Best: $O(n^2)$, Worst: $O(n^2)$

2.2-3

In average, $\frac{n}{2}$ elements need to be checked. $O(n)$

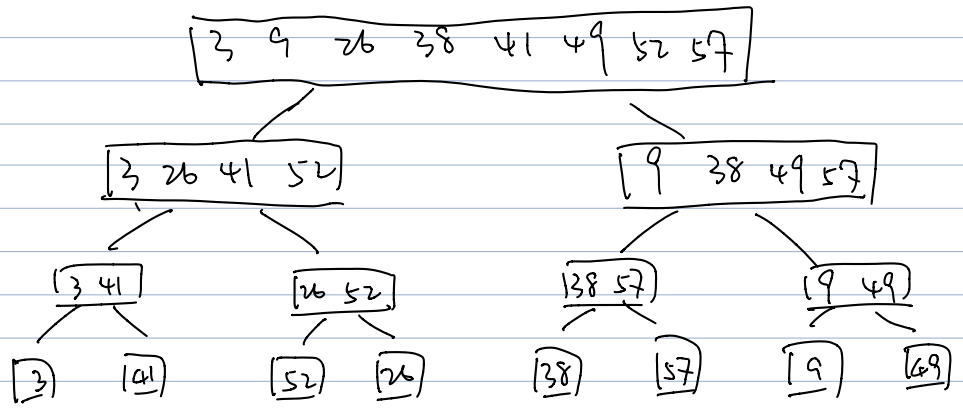
Worst case, n elements $O(n)$

2.2-4

Given a best-case, we modify the algorithm as, first check if the input is in the best case, if so, directly output the result. O/w, compute it from the very scratch.

2.3-1

$A = \langle 3, 41, 52, 26, 38, 57, 9, 49 \rangle$



2.3-2

```
YET-ANOTHER-MERGE(A, p, q, r)
  line 1-7 unchanged
  i = 1, j = 1, k = p
  while i ≤ n1 and j ≤ n2
    if L[i] ≤ R[j]
      A[k] = L[i]
      i++
    else
      A[k] = R[j]
      j++
    k++
  if i > n1
    while j ≤ n2
      A[k] = R[j], k++, j++
  else
    while i ≤ n1
      A[k] = L[i], k++, i++
```

2.3-3

Base: $n = 2$, $T(n) = T(2) = 2 \lg 2 = 2$ ✓

Induction: Suppose $T(2^k) = 2^k \log 2^k = k \cdot 2^k$ for some k, then

$$T(2^{k+1}) = 2T(2^k) + 2^{k+1}$$
$$= k \cdot 2^{k+1} + 2^{k+1}$$
$$= (k+1)2^{k+1} = 2^{k+1} \log 2^{k+1}$$ ✓

2.3-4

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

2.3-5

```
BINARY-SEARCH (A, k)   // A should already be sorted
  lo, hi = 1, A.length
  while lo ≤ hi
     mid = lo + (hi - lo)/2
     if A[mid] = k
        return mid
     else if A[mid] < k
        lo = mid + 1
     else
        hi = mid - 1
  return NIL  // not found
```

Worst Case: k not in the array A.
each time we eliminate half of the array.
until we only have a singleton element
$O(\log n)$

2.3-6

NOPE. You can certainly binary search the position for inserting. But since you're doing in-place sort, counting the cost of moving elements backwards you'll end up with $O(n^2)$ still.

2.3-7

Sort the array. Keep two pointers initially pointing to A[1] and A[n]
If the sum is larger than x, move the second pointer forward. ←
. . .. Smaller first backward →
is x, we succeed.

(You can do it in O(n) if you use a hash table.)

2-1

a. For each of the list of length k, it takes $O(k^2)$. There are $\frac{n}{k}$ of them, so $\frac{n}{k} O(k^2) = O(nk)$

b. Adopt the normal merge procedure, and merge them pairwise.
The cost: $O(k) \cdot \frac{n}{2k} + O(2k) \cdot \frac{n}{4k} + \cdots$ (Solve $k \cdot 2^i = n$ give us $i = O(\lg \frac{n}{k})$, which is the # of terms we have)
$= O(\frac{n}{2}) \cdot O(\lg \frac{n}{k})$
$= O(n \lg \frac{n}{k})$

c. k can be $O(\log n)$. In this case $\Theta(nk + n \lg \frac{n}{k}) = \Theta(n \log n + n \log n - n \log(\log n)) = \Theta(n \log n)$

d. measure the time it takes to sort k-length array by merge sort and select sort.
If select sort is faster, use this k.

2-2

a. We need to prove A' has the same elements as A.
Iow, A' is a permutation of A.

b. loop variant: A[j] is the smallest element in A[j..n] and A'[j..n] is a permutation of A[j..n]
Initialization: $j = n$, thus a singleton elem, thus ✓
Maintenance: Assume we have A[j] is the smallest element in A[j..n] then based on the algorithm we know the new smallest element would be swapped to A[j].
Also, since it's just comparisons and swaps we know it's a valid permutation.
Termination: After done, $j = i$, and we know A[i] is the smallest elem in A[i..n], also A'[i..n] is a valid permutation of A[i..n]

c. loop invariant : A[1..i] is sorted and A[1..n] is a valid permutation.
Initialization: i=1, singleton element, correct anyways
Maintanance: Suppose A[1..i-1] is sorted and A[1..n] is a valid permutation.
Based on part b, A[i] is the smallest among A[i..n], then A[1..i] is sorted.
Also a permutation of a permutation is also a valid permutation.
Termination : At last, i=n. we know A[1..n] is sorted, and is a valid permutation.

d. Worst-case : $O(n^2)$, Same as insertion sort.

2-3

a. $\theta(n)$

b. NAIVE-POLY-EVAL() $\theta(n^2)$

```
Sum = 0
for i = n downto 0
    term = A[i]
    for j = 0 to i
        Term *= x
    Sum += term
```

It's slower compared with Horner's rule.

c. Initialization: As stated, no term equals to 0 , $y=0$, equals

maintance: Suppose we have $y=\sum_{k=0}^{n-(i+1)} a_{k+i+1}x^k$ already, after one round

$$x\cdot y + a_i = \sum_{k=0}^{n-(i+1)} a_{k+i+1}x^{k+1} + a_i$$
$$= \sum_{k=1}^{n-i} a_{k+i}x^k + a_{k+0}x^0$$
$$= \sum_{k=0}^{n-i} a_{k+i}x^k \text{, which is the case when } i-1$$

Termination: At last, $i=-1$. we have $y=\sum_{k=0}^{n} a_k x^k$

d. By part c, the final form we get is exactly what we want about P(x).

2-4

a. $\langle 2,1\rangle, \langle 3,1\rangle, \langle 8,6\rangle, \langle 8,1\rangle, \langle 6,1\rangle$

b. $n, n-1, \cdots, 2, 1,$ It has $(n-1)+(n-2)+\cdots+1 = \frac{n(n-1)}{2}$ inversions

c. The number of the inversions is the same as the number of swaps in insertion sort.

d. Use mergesort, but we need to return # of inversions as well.
In the merge step, MERGE(A, p, q, r), each time when $L[i] > R[j]$, we have $q-i+1$ inversions.
The final # of inversions are # left inversion + # right inversion + # merge inversion.
Since the addition of inversions are constant, the running time is still $O(n\log n)$

3.1-1

WOLOG, we assume $f(n) > g(n)$, then $\max(f(n), g(n)) = f(n)$
We know $f(n) < f(n)+g(n)$, also since $0.5f(n) > 0.5g(n)$, we have
$0.5(f(n)+g(n)) < 0.5f(n)+0.5f(n) = f(n)$
Thus $\max(f(n), g(n)) = \theta(f(n)+g(n))$

3.1-2

$(n+a)^b = n^b + p_1 n^{b-1} + p_2 n^{b-2} + \cdots + p_b < (1+\sum_{k=1}^{b} p_k)\, n^b$
$(n+a)^b = n^b + \cdots > n^b$
So $(n+a)^b = \theta(n^b)$

3.1-3

The big-O notation itself indicates inequality, and it means 'less than', informally speaking. Which is not compatible with 'at least'.

3.1-4

Yes. No.

3.1-5

$\Rightarrow$ $f(n) = \Theta(g(n))$. So according to the definition, there $\exists\ C_1, C_2 > 0$, and n_0 s.t.

$$C_1 g(n) \le f(n) \le C_2 g(n) \text{ for all } n \ge n_0.$$

Thus, the right part indicates $f(n) = O(g(n))$, the left part indicates $f(n) = \Omega(g(n))$

$\Leftarrow$ Based on definition, we have n_1, C_1 s.t. $f(n) \le C_1 g(n)$ for $n \ge n_1$

we have n_2, C_2, s.t. $C_2 g(n) \le f(n)$ for $n \ge n_2$

pick $n_0 = \max(n_1, n_2)$, we have $C_2 g(n) \le f(n) \le C_1 g(n)$ for $n \ge n_0$, which indicates $f(n) = \Theta(g(n))$

3.1-6

An algo is $\Theta(g(n)) \Rightarrow \exists\ C_1, C_2, n_0$ s.t. $C_1 g(n) \le f(n) \le C_2 g(n)$ for $n \ge n_0$.

then $C_1 g(n) \le f_b(n) \le f_w(n) \le C_2 g(n)$, which promises the conclusion.

Also, if worst-case is $O(g(n))$, best-case is $\Omega g(n)$, we have

$C_2 g(n) \le f_b(n)$ and $f_w(n) \le C_1 g(n)$.

And all $f(n)$ showed have $f_b(n) \le f(n) \le f_w(n)$, then

$C_2 g(n) \le f(n) \le C_1 g(n) \Rightarrow f(n) = \Theta(g(n))$

3.1-7

If a function $f(n) \in o(g(n))$, then $\forall C > 0$, $\exists n_0 > 0$, s.t. $0 \le f(n) < C g(n)$ for all $n \ge n_0$,

then there's no such n_0', s.t. $C g(n) < f(n)$ for all $n \ge n_0'$. (o/w violates the o-notation definition)

So If $f(n) \in o(g(n))$, then $f(n) \notin \omega(g(n))$

Similarly for the opposite direction, thus $o(g(n)) \cap \omega(g(n)) = \phi$.

3.1-8

$\Omega(g(n,m))$: $\exists\ C, n_0, m_0 > 0$, s.t. $C g(n,m) \le f(n,m)$ for $\forall\ n \ge n_0$ or $m \ge m_0$.

$\Theta(g(n,m))$: if both $\Omega(g(n,m))$ and $O(g(n,m))$

3.2-1

Trivial high-school math...

3.2-2

Prove $a^{\log_b c} = c^{\log_b a}$

Denote $x = a^{\log_b c}$, then $\log_b x = \log_b(a^{\log_b c}) = \log_b c \log_b a = \log_b(c^{\log_b a})$, then $x = c^{\log_b a}$.

3.2-3

$\lg(n!) = \lg 1 + \lg 2 + \cdots + \lg(n-1) + \lg n \le n \lg n$, also

$\lg(n!) = \lg 1 + \lg 2 + \cdots + \lg(n-1) + \lg n \ge \lg\frac{n}{2} + \lg(\frac{n}{2}+1) + \cdots + \lg n \ge \frac{n}{2}\lg\frac{n}{2}$

So we can say $\lg n! = \Theta(n \lg n)$

To prove $n! = \omega(2^n)$, we know $n! = 1 \cdot 2 \cdot 3 \cdots n \ge 2 \cdot 2 \cdots 2 = 2^{n-1} = \frac{1}{2} \cdot 2^n$

To prove $n! = o(n^n)$, we know $n! = 1 \cdot 2 \cdot 3 \cdots n \le n \cdot n \cdots n = n^n$

3.2-4

No. Since $\lg(\lceil \lg n \rceil!) = O(\lceil \lg n \rceil \lg \lceil \lg n \rceil) = O(\lg n (\lg \lg n)) > O(\lg n)$, then $\lg n! > O(n)$

Yes. Since $\lg(\lceil \lg \lg n \rceil!) = O(\lg \lg n \cdot (\lg \lg \lg n)) = o(\lg n)$, thus $\lceil \lg \lg n \rceil! = O(n)$

3.2-5

$\lg^*(\lg n) = \lg^* n - 1$ based on definition.

Thus $\lg(\lg^* n) < \lg^* n - 1 = \lg^*(\lg n)$ larger.

3.2-6

Show $\phi^2 - \phi - 1 = 0$ and $\hat{\phi}^2 - \hat{\phi}^2 - 1 = 0$

3.2-7

Base: $F_0 = \frac{1-1}{\sqrt{5}} = 0$. $F_1 = \frac{\frac{1+\sqrt{5}}{2} - \frac{1-\sqrt{5}}{2}}{\sqrt{5}} = \frac{2\sqrt{5}}{2\sqrt{5}} = 1$.

Induction: $F_{i+2} = F_{i+1} + F_i = \frac{\phi^{i+1} - \hat{\phi}^{i+1}}{\sqrt{5}} + \frac{\phi^i - \hat{\phi}^i}{\sqrt{5}} = \frac{\phi^{i+2} - \hat{\phi}^{i+2}}{\sqrt{5}}$, since $\phi^2 - \phi - 1 = 0$ and $\hat{\phi}^2 - \hat{\phi}^2 - 1 = 0$

3.2-8

$k \ln k = \Theta(n)$, we can know $\exists n_0, c_1, c_2$, s.t. $c_1 n \le k \ln k \le c_2 n$ for $n \ge n_0$

then $c_1' \ln n \le \lg k + \lg \lg k \le c_2' \ln n$, which means $c_1'' \ln n \le \lg k \le c_2'' \ln n$

thus $c_1'' \frac{n}{\ln n} \le k \le c_2'' \frac{n}{\ln n}$, thus $k = \Theta(n / \ln n)$.

(easy to know, all the constants can be found.)

3.1

a. $p(n) = \sum_{i=0}^{d} a_i n^i \le (\sum_{i=0}^{d} a_i) n^k = O(n^k)$

b. $p(n) = \sum_{i=0}^{d} a_i n^i \ge n^k = \Omega(n^k)$

c. $p(n) \le (\sum a_i) n^k$, and $p(n) \ge a_k n^k$, thus $p(n) = \Theta(n^k)$

3.2

	A	B	O	o	Ω	ω	Θ
a.	$\lg^k n$	n^ϵ	✓	✓			
b.	n^k	c^n	✓	✓			
c.	$\sqrt{n}$	$n^{\sin n}$					
d.	2^n	$2^{n/2}$			✓	✓	
e.	$n^{\lg c}$	$c^{\lg n}$	✓		✓		✓
f.	$\lg(n!)$	$\lg(n^n)$	✓		✓		✓

3.3

(a) I'll skip ...

(b) Some piecewise function, like $f(n) = \begin{cases} 2^n & n \text{ even} \\ 0 & n \text{ odd} \end{cases}$

3.4

(a) NO

(b) NO

(c) YES, Prove by definition

(d) NO.

(e) YES

(f). YES

(g). NO

(h) YES

3-5

(a). Based on definition, if $f(n) = O(g(n))$, then $\exists C, n_0$, s.t. $0 \le f(n) \le cg(n)$ for all $n \ge n_0$.
If this is not the case, then $\forall C, n_0$, s.t. $f(n) \ge cg(n)$ for some arbitrarily large n_0.
Thus for every C we can find such a $n_0(c)$. Thus we have $f(n) = \tilde{\Omega} g(n)$.

(b) Advantages: nice notation to preserve completeness
Disadvantages: lose intuition

(c). now only $\Theta(g(n)) \Rightarrow O'(g(n))$ and $\Omega(g(n))$, not the other way.
An example would be $f(n) = (-1)^n 2^n$. $g(n) = 2^n$

(d) $f(n) = \tilde{\Omega}(g(n))$ $\exists c, k > 0, n_0 > 0$, s.t. $0 \le cg(n) \lg^k(n) \le f(n)$ for $\forall n \ge n_0$.
$f(n) = \tilde{\Theta}(g(n))$ $\exists c_1, c_2, k_1, k_2 > 0, n_0 > 0$, s.t. $0 \le c_1 g(n) \lg^{k_1}(n) \le f(n) \le c_2 g(n) \lg^{k_2}(n)$ for $\forall n \ge n_0$
No difference when proving analog of Thm 3.1

3-6

	$f(n)$	c	$f_c^*(n)$
a.	$n-1$	0	$\Theta(n)$
b.	$\lg n$	1	$\Theta(\lg^* n)$
c.	$n/2$	1	$\Theta(\lg n)$
d.	$n/2$	2	$\Theta(\lg n)$
e.	$\sqrt{n}$	2	$\Theta(\log\log n)$
f.	$\sqrt{n}$	1	∞
g.	$n^{1/3}$	2	$\Theta(\log\log n)$
h.	$n/\lg n$	2	$\Omega(\lg n/\lg\lg n)$

we have $(n^{\frac{1}{2^x}}) = 2$ then $\frac{1}{2^x} = \log n \Rightarrow x = \log\log n$
It would approximate 1 but never reach 1.

$$\frac{n}{\lg n} \to \frac{n(\lg n}{\lg(n(\lg n)} = \frac{n(\lg n}{\lg n - \lg\lg n} > \frac{n}{\lg^2 n}$$

$$\therefore \lg^x n > n \to x > \lg_{\lg n} n = \frac{\lg n}{\lg\lg n}$$