1. a. i 앞의 리스트는 (A[:i]) 정렬되어있다.

b. j는 i 뒤의 값 중 가장 작은 값의 index 이다.

c. i. outer loop invariant 가 i iteration 끝까지 유지된다

→ A[:i] 가 정렬됨

ii. i=0 일때 A[0] 이고, 1개의 값이므로 성립한다.

iii. A[:i] 가 정렬되어 있을때, i+1 iteration에서 뒤 list의 최솟값을 i+1로 swap 하므로

A[:i+1] 또한 정렬된 리스트이다.

iv. i~iii 에 의해 n iteration 에서 A[:n]은 정렬된다.

그러므로 알고리즘은 잘 동작한다.

d. i Inner Invariant (j가 i 뒤의 값중 가장작은 값의 index 선택)가

i iteration 끝까지 유지된다.

ii i=0일때 전체 리스트에서 기존 A의 모든 원소를 비교하여 최솟값을 선택한다.

iii. i iteration에서 A[i:] 중 최솟값을 비교하고

i+1 iteration에서는 A[i+1:] 중 최솟값을 찾는다

계속 유지된다. 보기가

iv. i~iii 에 의해, n iteration 까지 최솟값을 찾는 알고리즘은 잘 작동한다.

2. 블록의 크기를 놓지고 볼때, 예를들어 0~9의 블럭이 있는 경우,

09 18 27 36 54 와 같이 크기의 합이 같은 두 블럭이 묶이고

주개의 상 관계로

배열된 경우가 worst case 이다.

sorting 과정은 가장 큰값을 맨위로 올린 후, 다시 뒤집어 순서에 맞게

정렬하는 방식이다. 동일한 과정이 recursive 하게 일어나므로,

$$T(n) \le 2 + T(n-1) \quad \text{이때, } T(n-1) = 2 + T(n-2)$$

$$= 4 + T(n-2)$$

$$T(1) = 0.$$

$$T(2) \le 1$$

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

$$k = n$$

$$T(n) = 2n$$

$n \ge 2$ 일때, $T(n) = \underline{2n-3}$

$\underline{2n-3}$ 이 필요하다.

3. (a) $T(n) = 3T(\frac{n}{2}) + 1 \quad \leftarrow T(\frac{n}{2}) = 3T(\frac{n}{4}) + 1$

$= 3 \cdot (3T(\frac{n}{4}) + 1) + 1 = 3^2 T(\frac{n}{2^2}) + 3 + 1 \quad \leftarrow T(\frac{n}{4}) = 3T(\frac{n}{8}) + 1$

$= 3^2(T(\frac{n}{8}) + 1) + 3 + 1 = 3^3 T(\frac{n}{2^3}) + 3^2 + 3 + 1$

$\vdots$

$= 3^k T(\frac{n}{2^k}) + \underline{3^{k-1} + \cdots + 3 + 1}$

$\hookrightarrow \frac{(3^k - 1)}{3-1} = \frac{1}{2}(3^k - 1)$

$\frac{n}{2^k} = 1 \quad k = \log_2 n$

$= 3^{\log_2 n} T(1) + \frac{1}{2}(3^{\log_2 n} - 1)$

$= n^{\log_2 3} T(1) + \frac{1}{2} \times n^{\log_2 3} - \frac{1}{2}$

$= C \cdot n^{\log_2 3} - \frac{1}{2} \geq C \cdot n \qquad \underline{\Theta(n^{\log_2 3})}$

3-b. $T(n) = 3T\left(\frac{n}{2}\right) + n$

$= 3^2 T\left(\frac{n}{4}\right) + 3n + n$

$= 3^3 T\left(\frac{n}{8}\right) + 3^2 n + 3n + n = \cdots = 3^k T\left(\frac{n}{2^k}\right) + \underline{n(3^{k-1} + 3^{k-2} \cdots + 3 + 1)}$

$n \frac{3^k - 1}{3 - 1}$

$2^k = n \quad k = \log_2 n$

$= 3^{\log_2 n} T(1) + \frac{n}{2}(3^k - 1)$

$= C_1 \cdot n^{\log_2 3} + C_2 n$ $\quad 2 > \log_2 3 > 1$ 이므로

$\underline{\Theta(n^{\log_2 3})}$

3-c. $T(n) = 3T\left(\frac{n}{2}\right) + 3^n$

$= 3^2 T\left(\frac{n}{4}\right) + 3^{n+1} + 3^n = 3^3 T\left(\frac{n}{8}\right) + 3^{n+2} + 3^{n+1} + 3^n$

$= \cdots = 3^k T\left(\frac{n}{2^k}\right) + 3^n(\underline{3^{k-1} + \cdots + 3 + 1})$

$k = \log_2 n \quad \hookrightarrow \frac{3^k - 1}{2}$

$= \underline{3^{\log_2 n}}_{n^{\log_2 3}} T(1) + 3^n \cdot \frac{3^k - 1}{2}$

$= C_1 n^{\log_2 3} + C_2 \cdot 3^n$ $\quad \underline{\Theta(3^n)}$

$n^{\log_2 3} < 3^n$

3-d. $T(n) = 8T\left(\frac{n}{2}\right) + n^3$

$= 8\left(8T\left(\frac{n}{4}\right) + \left(\frac{n}{2}\right)^3\right) + n^3 = 8^2\left(8T\left(\frac{n}{8}\right) + \left(\frac{n}{4}\right)^3\right) + 8 \cdot \left(\frac{n}{2}\right)^3 + n^3$

$= \cdots = 8^k T\left(\frac{n}{2^k}\right) + \underline{8^{k-1}\left(\frac{n}{2^{k-1}}\right)^3 + \cdots + n^3}$

$\hookrightarrow n^3 \{ \underbrace{\cancel{8^{k-1}} \cdot \frac{1}{\cancel{8^{k-1}}} + \cdots + 1}_{k} \}$

$2^k = n \qquad k = \log_2 n$

$= \underbrace{8^{\log_2 n}}_{n^{\log_2 8} = n^3} T(1) + n^3 \cdot \log_2 n \quad = n^3 \cdot T(1) + n^3 \cdot \log_2 n$

$= C \cdot n^3$

$\underline{\Theta(n^3)}$

3-e. $T(n) = 3T(n-1) + 1 \qquad T(n-1) = 3T(n-2) + 1$

$= 3^2 T(n-2) + 3 + 1 \qquad T(n-2) = 3T(n-3) + 1$

$= 3^3 T(n-3) + 3^2 + 3 + 1$

$\vdots$

$= 3^k T(n-k) + \frac{3^k - 1}{3-1} \qquad n-k=1$

$k = n-1$

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

$= \underbrace{\left(3T(1) + \frac{1}{6}\right)}_{C_1} 3^n - \underbrace{\frac{1}{2}}_{C_2} \qquad \Theta(3^n)$

4.

$$T(n) = T\left(\frac{n}{2}\right) + T\left(\frac{n}{4}\right) + T\left(\frac{n}{8}\right) + T\left(\frac{n}{16}\right) + n \qquad T(1) = 1$$

$$- \quad T\left(\frac{n}{2}\right) = T\left(\frac{n}{4}\right) + T\left(\frac{n}{8}\right) + T\left(\frac{n}{16}\right) + T\left(\frac{n}{32}\right) + n$$

$$T(n) = 2T\left(\frac{n}{2}\right) - T\left(\frac{n}{32}\right)$$

Guess $\Rightarrow$ $O(n)$

$T(k) \le Ck$ for all $1 \le k < n$

$$\frac{7}{10}c + 3 \le c$$
$$\frac{3}{10}c \ge 3$$
$$c \ge 10$$

Base $\rightarrow$ $T(k) \le Ck$ $\quad k=1 \rightarrow \quad 1 \le C$

Inductive Step $T(n) = 2T\left(\frac{n}{2}\right) - T\left(\frac{n}{32}\right) \le \cancel{2}\frac{Cn}{\cancel{2}} - \frac{Cn}{32}$

$$\frac{31}{32}Cn \le Cn.$$ 만족시키는 $C=1$ 이 존재.

모든 $n \ge 1$ 에서 $T(n) \le Cn$ 을 만족시키는 C가 존재하므로

$T(n) = O(n)$ 이다.

6. **(8 points)** Radix sort 는 대표적인 선형시간 정렬알고리즘(Linear Time Sorting) 중 하나이다. Radix sort 의 subroutine 인 bucket sort 에 대해, 배열 A = [byte, pits, bits, pins] 가 주어졌을 때 아래의 질문에 답하시오. (a to z ascending order, 풀이과정 필요 X)

a. Bucket Sort 의 첫번째 호출 후 배열의 상태는?

[byte, pits, bits, pins]

b. Bucket Sort 의 두번째 호출 후 배열의 상태는?

[pins, byte, pits, bits]

c. Bucket Sort 의 세번째 호출 후 배열의 상태는?

[pins, pits, bits, byte]

d. Bucket Sort 의 네번째 호출 후 배열의 상태는?

[bits, byte, pins, pits]

5.

a.

```
Find-Inversion-Pairs (A, low, high)                        T(n)
        inversion_count = 0
        if len(A) <=1:
                return A, inversion_count
                                                                 T(n/2)
        left, l_count = Find-Inversion-Pairs (A[:n/2])
        right, r_count = Find-Inversion-Pairs (A[n/2:])
        inversion_count += l_count                                T(n/2)
        inversion_count += r_count

        result = []
        l_index, r_index = 0, 0

        while l_index < len(left) and r_index<len(right) :
                if left[l_index] < right[r_index]:
                        result.append(left[l_index])
                        l_index++
                elif left[l_index] == right[r_index]:                Θ(n)
                        result.append(right[r_index])
                        r_index++
                else :
                        inversion_count ++
                        result.append(R[r_index])
                        r_index++

        result.extend(L[l_index:len(left)])
        result.extend(R[r_index:len(right)])

        return result, inversion_count
```

b.

$T(n) = T(n/2) + T(n/2) + \theta(n)$

6.

a.

```
def mergesort3(A, n):
    if n <= 1:
        return A

    unit = n//3

    # part1. You need to handle additional edge cases (Hint: see above)
    if unit == 0:
        if A[0]> A[1]:
            a = A[0]
            A[0] = A[1]
            A[1] = a
            return A

    # part2. reculsively do something
    L = mergesort3(A[:unit],unit)
    M = mergesort3(A[unit:unit*2],unit)
    R = mergesort3(A[unit*2:],n-(unit*2))

    # part3. merge something and return it
    L_M = merge(L,M)
    return merge(L_M,R)
```

$T(\frac{n}{3})$

$O(n)$

$$T(n) = 3 \cdot T\left(\frac{n}{3}\right) + \Theta(n)$$

subproblem size가 동일하므로, master theorom 적용가능

$a=3 \quad b=3 \quad f(n)=n$

$$\therefore T(n) = O(f(n) \times \log_3 n)$$

$$= O(n \log_3 n)$$

6.

b.

```python
def mergesortK(A, n, k):
    if n <= 1:
        return A
    A = A.copy()

    unit = n//k
    # part1. You need to handle additional edge cases
    if unit == 0:
        return mergesortK(A, n, k-1)

    # part2. reculsively do something
    i = k
    united_list = []
    while i > 1:
        united_list.append(mergesortK(A[:unit], unit, k))
        del A[:unit]
        i = i - 1
    united_list.append(mergesortK(A, len(A), k))

    # part3. merge something and return it
    m = len(united_list)-1
    while m > 0:
        united_list[m-1] = merge(united_list[m-1], united_list[m])
        m = m -1
    return united_list[0]
```

$T(\frac{n}{k}) \times k$

↳ $\Theta(n)$

$$T(n) = k \times T\left(\frac{n}{k}\right) + \Theta(n)$$

by master theorom

$a = k \quad b = k$

$$\therefore T(n) = O(n \log_k n)$$

2way　　3way　　kway

$n \geq 1$에서 $n\log_2 n \geq n\log_3 n \geq n\log_k n$

정렬연산을 병렬적으로 많이 하기 때문에 k가 클수록 시간복잡도는 줄어든다.