Homework #1

1

a. At the beginning of iteration i (the iteration where we try to select the element to occupy A[i]), A[: i] contains the i smallest elements from the original list A, in sorted order.

b. At the beginning of iteration j, min_idx contains the index of the minimum element in the sublist A[i : j].

c. i) The loop invariant holds at the beginning of iteration i of the inner loop. i.e. A[: i] contains the i smallest elements from the original list A, in sorted order.

ii) The loop invariant holds before the algorithm starts when i=0 i.e. A[: 0] contains 0 elements, which are trivially the smallest elements in sorted order

iii) Suppose the loop invariant holds at the beginning of iteration i. We prove it holds at the beginning of iteration i+1. If the inner loop invariant holds at the end of the inner loop, then min_idx contains the index of the smallest element in A[i : n]. Calling swap swaps this smallest element into position A[i]. According to the inductive hypothesis, A[0] ≤ A[1] ≤ ... ≤ A[i-1]; also, A[i] is at least as large as A[i-1] since A[: i] contained the i smallest elements from the original list A. Therefor after the swap, A[: i+1] contains the i+1 smallest elements from the original list A, in sorted order, completing the induction

iv) At the beginning of iteration n-1 of the outer loop, A[: n-1] contains the n-1 smallest elements from the original list A, in sorted order. If A[: n-1] contains the n-1 smallest elements from the original list A, then A[n-1] must be at least as large as all elements in A[: n-1]; therefore A[: n] is sorted. Since A[: n] is the whole list A, selection sort is correct

d. i) The loop invariant holds at the beginning of iteration j of the inner loop i.e. min_idx contains the index of the minimum element in the sublist A[i : j]

e.

ii) The loop invariant holds before the loop starts when j=i+1 i.e. min_idx is set to i and A[i : i+1] contains only one element, element i.

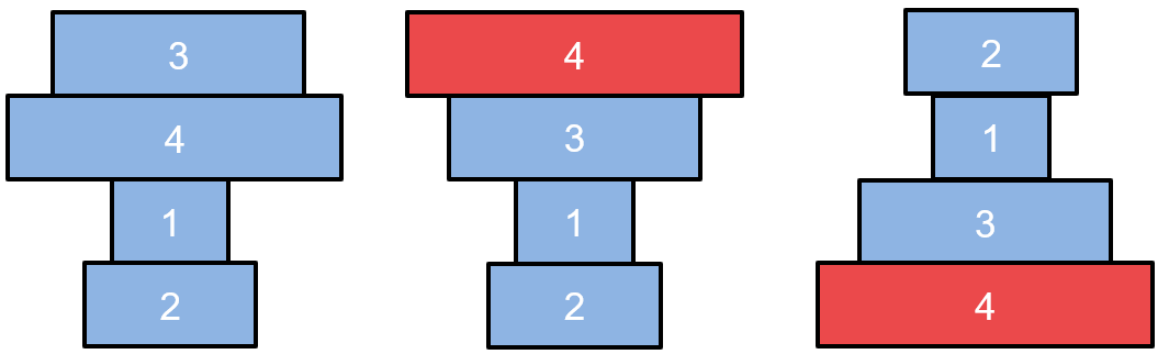
iii) Suppose the loop invariant holds at the beginning of iteration j. We prove it holds at the beginning of iteration j+1. According to the inductive hypothesis, min_idx contains the index of the minimum element in the sublist A[i : j]. If A[j] < A[min_idx], then A[j] is the minimum element in A[i : j+1], so setting min_idx = j is correct. Otherwise, A[min_idx] is the minimum element in A[i : j+1],. so leaving it along is correct. this completes the induction.

iv) At the beginning of iteration n of the inner loop, min_idx contains the index of the minimum element in the sublist A[i:n]. This is the condition required by the inductive step of outer loop.

2. 정답 2n-3 (n>2)

2 번문제는 펜케이크 정렬문제로 임의로 블럭이 정렬된 형태에서 밑판이 가장 넓은 블럭이 가장 아래에 오는 정렬된 형태로 만드는 최소한의 뒤집기 횟수를 bound 하는 것입니다.

아래 맨왼쪽 그림을 initial sate 라 할때 생각할 수 있는 알고리즘은 가장 큰 원판을 가장 위로 올려서 가장 아래와 뒤집는 것입니다.



다음 차례에서는 똑같이 4 를 제외한 3 개의 원판에서 가장큰 블럭(3)을 선택하여 맨위로 올려주고, 아래와 뒤집는 연산을 수행해 주면 됩니다.

이를 활용하면 각 사이클당 2 번의 연산이 필요함으로 n 개의 블럭에대해 2(n-1) 번을 수행해 주면 된다고 생각할 수 있습니다.

하지만 마지막 사이클에서는 2 번의 연산이 필요 없습니다. (왜냐하면 블럭 두개만 남을 경우, 2 회의 연산이 아닌 1 회의 연산만으로 완성할 수 있습니다.) 따라서 n-2 번까지는 2 회의연산을, 마지막 n-1 사이클에서는 1 번의 연산만을 해주면 되므로 2(n-2) +1 = 2n-3 의 뒤집기로 bound 할 수 있습니다.

3.

a**. (2points) T(n) = O(n^{log3}**) by master method

b. (**2points) T(n) = O(n^{log3})** by master method

c. (**2points) T(n) = O(3^n)** by master method (d = $\log_n 3^n$, n>1)

d. **(2points)T(n) = O(n^3logn)** by master method

e. **(4points)T(n) = O(3^n**)

T(n) = 3T(n-1) + 1

= 3*(3T(n-2) +1) + 1

= 3*3*(3T(n-3) +1) + 1 + 3

after n iteration : $3^n T(0) + 1+3+3^2+...+3^{n-1} = 3^n T(0) + (3^n-1)/(3-1) \sim= O(3^n)$

4.

1. Guess what the answer is

T(n) ≤4T(n/2) + n , T(n) ≤ $O(n^2)$ and n ≤ T(n)

we can guess **T(n) = O(n)**

2. Formally prove that's what the answer is

if we assume T(n) = Kn, K = 16

inductive hypothesis T(n) ≤ 16n for all 1≤n

base case T(1) = 1 < 16

Inductive step T(n)= T(n/2)+ T(n/4)+ T(n/8) T(n/16)+ n
≤ 16n

conclusion By induction, we conclude that T(n) ≤ 16n for all 1≤n which establishes the inductive hypothesis . So T(n) = O(n)

풀이 없을시 0 점

5.

a. **(10points)** pseudo code (별도 첨부 5.cpp 참고)

```
Function Find-Inversion-Pairs(A,low,high)
{
    if(low<high) then
        mid = (low+high)/2
        c1 = Find-Inversion-Pairs(A,low,mid)
        c2 = Find-Inversion-Pairs(A,low,mid)
        c3 = Merge-Inversion-Pairs(A,low,mid,high)
        return c1+c2+c3
    else return 0
}
```

Divide and Conquer Logic 이 위의 코드와 유사하면 정답처리

b. **(5points)** T(n) = 2T(n/2) + O(n) (merge sort 와 동일)

6.

a. **(2points)** [byte, pits, bits, pins]

b. **(2points)** [pins, byte, pits, bits]

c. **(2points)** [pins, pits, bits, byte]

d. **(2points)** [bits, byte, pins, pits]

7. 코드는 별도 첨부의 예시 코드를 확인 (7_3.py, 7_k.py)

(b) 이론적으로, 3 가지 경우 모두 O(nlogn)이다. 하지만, 실제 실행시간은 way 의 수가 증가함에 따라 merge 할때 비교 연산이 더욱 많아져서 오래 걸릴 수 있다.

세 경우 모두 O(nlogn)으로 표현한 경우 정답.

예외적으로, O(knlogn)과 같이, 실행시간이 비교적 증가함을 나타내는 표현을 사용해도 정답.