Exercise 1.

(i) For any comparison-based algorithm, it's clear that each comparisom can produce a smaller one and a larger one, and the larger one have no chance to be the smallest number. For each number except the smallest one, it needs exactly one comparison to be excluded. Therefore, there are exactly n-1 comparisons in such algorithm.

(ii) similar to (i), we can first find the smallest number in n-1 comparisons.

Then for the second smallest number, we know that this number must have been compared with the smallest number in the above comparisons, so the possible second smallest number is among $\lceil \log_2 n \rceil$ numbers, which equals to how many times the smallest number had been at least compaired.

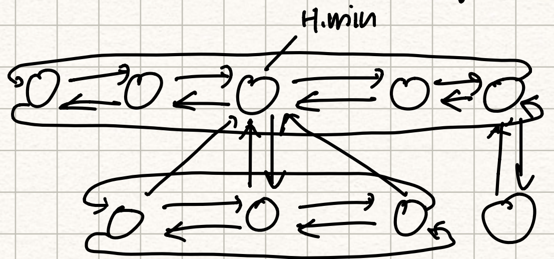
Therefore, we can have a new list of $\lceil \log_2 n \rceil$ numbers, and need $\lceil \log_2 n \rceil - 1$ comparisons to find the smallest one which is the second smallest of the original list.

So totally we need $n-1+\lceil \log_2 n \rceil - 1 = n-1+\lfloor \log_2 n \rfloor$ comparisons.

(iii) Details are in the code

Exercise 3.

(i) we can use fibonacci heap to realise this.



For each item, it contain fields

next	former	father	son	key	degree
pointer to next	pointer to former	pointer to father	pointer to son	the value	the number of sons

The handle is just the address of a specific item.

for basic operations:

insert(k): just add another item to the root list.

delete_min(): remove the minimun item, and loop to find another min

decrease(h,k): Use h to point to the target item and decrease. If the order is violated, cut out the subtree and insert into root list

delete(h): first apply decrease(h, $-\infty$), then delete_min()

(ii) For sorted list:

insert(k): O(1) As it is appended to the root list directly

delete_min(): O(1) As the root list is sorted, need constant time to find the next min

decrease(h,k): O(1) with handle h, only need constant time

delete(h): O(n) need to call delete-min and decrease.

For unsorted list:

insert(k): O(1) As it is appended to the root list directly

delete_min(): O(n) As the root list is unsorted, need to loop to find the next min

decrease(h,k): O(1) with handle h, only need constant time

delete(h): O(n). need to call delete-min and decrease.