1:

The algorithm uses heap sort: first build a heap using the given list where the heap has the smallest number as the root, then remove the root for k times. The removed parts is the partially sorted list.

//request one more space after A[n-1]

//shift the whole array one place to the right.

//took time O(n)

for (int i = 0; i < n; ++n){

A[i] = A[i-1];

}

//recursive function that keeps a non-root node in place by swaping with its parrent.

//will take time O(log n) each time calling it.

check\_valid(int i){

if (i = 1){

return;

} else {

if (A[i] > A[i/2]){

swap (A[i], A[i/2])

} else {

check\_valid(i/2);

}

}

}

//rebuild the original array so that it is in heap format.

//took time O(0 + 2 \* 1 + 4 \* 2 + 8 \* 3 + ... + n/2 \* log n) = O(nlog n) //in worst case

for (int i = 1; i <= n; ++n){

check\_valid(i);

}

//after this step, the array is a heap with the smallest number being the //root. for any index i, i/2 is its parent, i\*2 is its left child, i\*2+1 //is its right child. reminder: the array starts at index 1, ends at index //n. by removing the smallest node k times, the k smallest numbers are //picked out.

//took time: O(klog n)

int N = n;

for (int i = 1; i <= k; ++i){

// A[N] is the smallest number by property of heap.

swap (A[1], A[N]);

// format rebuilds the array by placing A[the first argument] in the

// right position.

format (1, N);

// the second argument states the heap is from A[1] to A[N], since

//the heap shrinks each time we remove the root from the heap.

// record the current array length;

--N;

}

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

// helper function: format

// format rebuilds the array by placing A[the first argument] in the right

// position. the second argument states the heap is from A[1] to A[N],

// since the heap shrinks each time we remove the root from the heap.

// took time O(log n)

format (int index, int last){

int smallest\_child;

if (index \* 2 > last){

// index is a leaf node.

return;

} else if (index \* 2 = last ){

// index has only one child.

smallest\_child = A[index \* 2];

} else {

// index has two children.

smallest\_child = (A[index \* 2] < A[index \* 2 + 1])? index \* 2, index \* 2 + 1;

}

if (A[index] > A[smallest\_child]){ // out of order

swap(A[index], A[smallest\_child]);

format (smallest\_child, length);

}

}

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

//shift the whole array back, and swap the head and tails.

//took time O(n)

for (int i = 1; i <= n; ++n){

A[i-1] = A[i];

}

for (int i = 0; i < n/2; ++n){

A[i] = A[n-1-i];

}

The total run time is except the initialization part.

2:

a)

Let , then .

Therefore , .

At the same time, for large , , we have

Therefore , .

Hence .

b)

void deleteMax(){

temp = P\_00; // record content;

swap (P\_00, P\_XY); // swap content; XY is the index of the last element

if (P\_XY->leftParent){

P\_XY->leftParent->rightChild = null;

}

if (P\_XY->rightParent){

P\_XY->rightParent->leftChild = null;

}

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

// helper function: format

// format rebuilds the pyramid by placing P\_ij in the right position.

// took time O(log n), n is the height of the pyramid.

format (int i, int j){

//X,Y is used to denote the largest child's position.

int X;

int Y;

if (!(P\_ij->leftChild)){

// P\_ij is a leaf node.

return;

} else if (!(P\_ij->RightChild)){

// P\_ij is has only one child on its left.

X = i + 1;

Y = j;

} else {

// P\_ij has two children.

X = i + 1;

Y = (P\_i+1,j < P\_i+1,j+1)? j, j + 1;

}

if (P\_ij < P\_XY){ // out of order

swap(P\_ij < P\_XY); // content

format (X, Y);

}

}

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

format (0,0);

return temp;

}

c)

void insert(TYPE content){

//assume XY is the index of the last element

// i,j denotes the index of the added element

// initialization

// took time O(1)

int i,j;

if (X == Y){

p\_X+1,0 = new node (content);

i = X + 1;

j = 0;

} else {

auto temp = new node (content);

P\_XY->rightParent->rightChild = temp;

i = X;

j = Y + 1;

if (i != j){

P\_i-1,j ->leftChild = temp;

}

}

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

// helper function: formatUp

// formatUp rebuilds the pyramid by placing P\_ij in the right position.

// took time O(n), n is the height of the pyramid.(after each recursive

// call of function, i decreases by one, and stops at 0)

formatUp (int i, int j){

//X,Y is used to denote the smallest parrent's position.

int X;

int Y;

if (i == 0){

//the current node is the root.

return;

} else if (j == 0){

// P\_ij is the left most node.

X = i - 1;

Y = j;

} else if (j == i){

// P\_ij is the right most node.

X = i - 1;

Y = j - 1;

} else {

// P\_ij has two parents.

X = i - 1;

Y = (P\_i-1,j-1 < P\_i-1,j)? j - 1, j;

}

if (P\_ij > P\_XY){ // out of order

swap(P\_ij < P\_XY); // content

formatUp (X, Y);

}

}

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

formatUp (i, j);

}