HW#3.

Dong Boyuan

1547489

Problem 1.

(a). Since n is very large and k is fairly small, elements in array A are mostly sorted already. So its runtime is almost the best case of the algorithem. So, the overall runtime will take $O(n)$ time., as PutInPlace will take constant. time overall.

For MergeSort, it will still take $O(n\log n)$ time, even though it is a almost sorted. array. Because, mergesort will still continue dividing array into parts. which will take $O(\log n)$ time and then combine them ~~too~~ together which will take $O(n\log n)$ time overall.

For HeapSort, it will still take $O(n\log n)$ time, although A is almost sorted. For each subtree, it will call the Max-Heap ify. to make the subtree a heap. which will take $O(\log n)$ time, There will be $O(n)$ trees., so it will take $O(n\log n)$ time overall.

For QuickSort, its runtime will be $O(n^2)$, for an almost sorted array A. Each time it will chose the pivot and the partition will be 1 and $O(n)$. Since it will take $O(n)$ time for comparing $O(n)$ keys to pivot, So, it will take $O(n^2)$ time overall.

clearly, we can see InsertSort will take the least time $O(n)$.

(b)

For. InsertSort, k elements were exchanged their positions arbitrarily. That means, it's possible that first ten element are happened to be the largest elements of the whole array, which will take $O(n)$ time in PutInplace. Then it will take $O(n^2)$ time overall in such case.

For MergeSort, it will still take $O(n\log n)$ time. Because, MergeSort will still take $O(\log n)$ time to divide array into parts then put them back together. Overall it will take $O(n\log n)$ time.

For HeapSort, it will still take $O(n\log n)$ time. For each position, Max-Heapify takes $O(\log n)$ time, so, in total this is $O(n\log n)$.

For QuickSort, it will still take $O(n^2)$ time.. For each pivot it chosen, it will take $O(n)$ time to get its correct position. So, in total, this is $O(n^2)$.

But, we can know from the example, when comparing bad and good elements PutInplace will take $O(n)$ time. While comparing good elements it will take $O(1)$ time. Overall, InsertSort's runtime still depends on the positions that k elements stays.
However, k is fairly smaller than n, so even though when compare bad and good elements, it will take a constant time.
Overall, it will take InsertSort $O(n)$ time.
InsertSort will take least time $O(n)$.

Dong Boyuan
1547489

problem 2. Dong Boyuan 1547489

(a). Claim: for each $n \geq 1$ $M^n = \begin{bmatrix} a_{n-1} & a_n \\ a_n & a_{n+1} \end{bmatrix}$, $M = \begin{bmatrix} 0 & 1 \\ 1 & 2 \end{bmatrix}$, and $a_0 = 0$ $a_1 = 1$

for $n \geq 2$ $a_n = 2a_{n-1} + a_{n-2}$

Proof: by induction:

Base case: $a_0 = 0$ $a_1 = 1$ $a_2 = 2a_1 + a_0 = 2 \times 1 + 0 = 2$. and $M = \begin{bmatrix} 0 & 1 \\ 1 & 2 \end{bmatrix} = \begin{bmatrix} a_0 & a_1 \\ a_1 & a_2 \end{bmatrix}$

which is true when $n=1$ $\rightarrow$ true.

Inductive step: Fix $n > 1$. Assuming that ~~for~~ $M^n = \begin{bmatrix} a_{n-1} & a_n \\ a_n & a_{n+1} \end{bmatrix}$ holds the claim,

we need to show $n+1$ also holds the claim: $M^{n+1} = \begin{bmatrix} a_n & a_{n+1} \\ a_{n+1} & a_{n+2} \end{bmatrix}$

$$M^{n+1} = M^n \cdot M = \begin{bmatrix} a_{n-1} & a_n \\ a_n & a_{n+1} \end{bmatrix} \begin{bmatrix} 0 & 1 \\ 1 & 2 \end{bmatrix}$$

$$= \begin{bmatrix} 0 \cdot a_{n-1} + a_n & a_n \cdot 0 + a_{n+1} \\ a_{n-1} + 2a_n & a_n + 2a_{n+1} \end{bmatrix}$$

$$= \begin{bmatrix} a_n & a_{n+1} \\ a_{n+1} & a_{n+2} \end{bmatrix} \text{ since } \begin{array}{l} a_{n+1} = 2a_n + a_{n-1} \\ a_{n+2} = 2a_{n+1} + a_n. \end{array}$$

So for $n+1$ the claim also holds. $\rightarrow$ true.

(b).

```
procedure getA(n)
F[1] ← 0
F[2] ← 1
F[3] ← 1
F[4] ← 2
if(n= 0) then
    return 0
if(n=1) then
    return 1
power(F, n)
return F[2];
```

```
procedure power(F, n)
if(n>1) then
    power(F, n/2);
    multiply(F, F).
    if(n%2 != 0) then
        M[1] ← 0
        M[2] ← 1
        M[3] ← 1
        M[4] ← 2
        multiply(F, M)
```

Dong Boyuan
1547489

procedure multiply (F , M)

$x \leftarrow F[1] * M[1] + F[2] * M[3]$

$y \leftarrow F[3] * M[1] + F[4] * M[3]$

$z \leftarrow F[1] * M[2] + F[2] * M[4]$

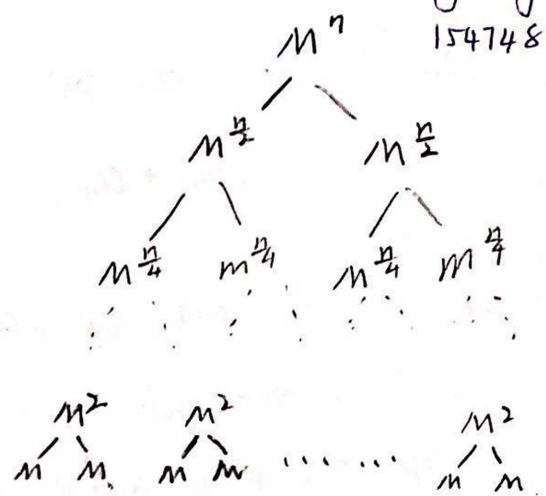
$w \leftarrow F[3] * M[2] + F[4] * M[4]$

$F[1] \leftarrow x$

$F[2] \leftarrow y$

$F[3] \leftarrow z$

$F[4] \leftarrow w$.



$$M^n = \underbrace{M \cdot M \cdot M \cdots M}_{n} = \prod_{1}^{n} M^1 \qquad M = \begin{bmatrix} 0 & 1 \\ 1 & 2 \end{bmatrix}$$

So the height of tree is $\log_2 n$ when n is power of 2. $> \in \Theta(\log n)$.

the height of tree is $\log_2 n + 1$ when n is not power of 2

$$T(n) = \begin{cases} O(1) & 0 \leq n \leq 1 \\ T(\frac{n}{2}) + O(1) & n \geq 2 \end{cases}$$

$$f(n) = O(1) \in \Theta(n^{\log_2 1} \cdot \log^k n) = \Theta(\log^k n) \in \Theta(\log n)$$

for some $k \geq 0$. by appling Master Theorem Method case II

so. the runtime is $O(\log n)$

problem 3. Dong Boyuan 1547489

(a). procedure multiply (M, v)

$**$ M – is a n-dimential array. n×n

$**$ V, W are n-cell array

for(i from 1 to n)

  sum ← 0

  for(j from 1 to n)

    sum ← sum + M[i][j] * V[j]

  W[i] ← sum.

return W.

Runtime analysis – big O.

The for-loop iterates n times. → O(n) times

In each iteration we do constant amount of instructions + the for-loop.

The for-loop iterates at most O(n) times.

Overall runtime: $O(n)(O(1)+O(n)) = O(n^2)$

Runtime analysis – big Ω

for every i the for-loop iterates n times. → O(n) times.

This means that for each $i \in [\frac{n}{2}, \frac{n}{2}+1, \cdots n]$ the for-loop iterates at least $i > \frac{n}{2}$ times, and each iteration we take at least one action.

Overal runtime: $\frac{n}{2} \cdot \frac{n}{2} \in \Omega(n^2)$

Conclusion: runtime is $\Theta(n^2)$.

(b). procedure product (M, V, S, n)

W[1] ← M[1][1] * V[S] + M[1][2] * V[S+1]

W[2] ← M[2][1] * V[S] + M[2][2] * V[S+1]

if(n = 2) then

  return W.

Mk1 ← product (M, V, S, $\frac{n}{2}$)

Mk2 ← product (M, V, $\frac{n}{2}$, $\frac{n}{2}$)

W ← add (Mk1, Mk2, n)

return W.

procedure add (Mk1, Mk2, n)

for(i from 1 to $\frac{n}{2}$)

  W[i] ← Mk1[i] + Mk2[i]

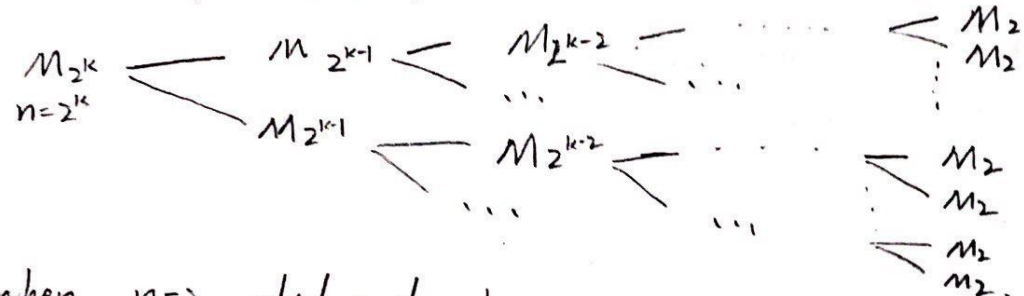
for(j from 1 to $\frac{n}{2}$)

  W[j + $\frac{n}{2}$] ← Mk1[j] - 2 * Mk2[j]

return W.

claim: For any $n \times n$ matrix M, n-dimentional vector V, product( ) returns the product of M with V.

Dong Boyuan
1547489.



when $n=2$ which is the base case, it will return a 2-dimentional vector W which is the product of $M_{2\times2}$ with $V\begin{bmatrix} x \\ y \end{bmatrix}$

when $n>2$, divide the matrix into 4 parts, $M_{2^k \times 2^k} \to M_{2^{k-1} \times 2^{k-1}}$ each smaller part M_K then continue dividing into parts, until reach the base case. Then it will get the product of first small $M_{2\times2}$ with $V\begin{bmatrix} 1 \\ 2 \end{bmatrix}$ and the product of first small $M_{2\times2}$ with $V\begin{bmatrix} 3 \\ 4 \end{bmatrix}$, then add them together to get the product of $M_{2^2 \times 2^2}$ matrix with $V\begin{bmatrix} 1 \\ 2 \\ 3 \\ 4 \end{bmatrix}$ and then use the product to get the new product of $M_{2^3 \times 2^3}$ matrix with $V\begin{bmatrix} 1 \\ \vdots \\ 8 \end{bmatrix}$ until get the product of $M_{n \times n} = M_{2^k \times 2^k}$ with $V\begin{bmatrix} 1 \\ \vdots \\ n \end{bmatrix}$ which is $W\begin{bmatrix} 1 \\ \vdots \\ n \end{bmatrix}$.

Since the height of the tree is $k = \log n$, so we need $k = \log n$ times n-dimensional vectors additions. Each two n-dimensional vectors takes $O(n)$ time. So, it will take $((\log n)\, n) \in O(n \log n)$ time.

problem 4

Dong Boyuan
1547489

(a)

```
procedure mergeHeap(A, B, n)
  for(i from 1 to n)
      merged[i] ← A[i]
  for(j from 1 to n)
      merged[j+n] ← B[j]
  Build-Max-Heap(merged).
```

Since the runtime of Build-Max-Heap(A) is $O(n)$ for building an array A with n elements

So, the runtime of mergeHeap(A, B, n) is $O(2n)+O(2n) \in \Theta(n)$
(runtime coping 2n elements to an new array + runtime of Build-Max-Heap() $\in \Theta(n)$)

(b)

```
procedure getMedian(A, S, B, n)
```

** A, B are two sorted arrays with n elements
** S is the start index of A, and S=1 when 'getMedian()' first called.

```
if(n=1) then
    return (A[1] + B[1]) / 2
if(n=2) then
    return (max(A[1], B[1]) + min(A[2], B[2])) / 2
m1 ← median(A, S, n)
m2 ← median(B, 1, n)
if(m1 = m2) then
    return (m1+m2) / 2
if(m1<m2) then
    if(n%2=0) then
        return getMedian(A, n/2, B, n/2)
    return getMedian(A, n/2+1, B, n/2)

if(n%2=0) then
    return getMedian(B, n/2, A, n/2)
return getMedian(B, n/2+1, A, n/2)
```

procedure Median (A, S, n)

Dong Boyuan

1547489

if ($n \% 2 = 0$) then

return $\dfrac{A[S+\frac{n}{2}-1] + A[S+\frac{n}{2}]}{2}$

return $A[S+\frac{n}{2}]$

(C). procedure printHeap (A, i, x)

```
if( A[i] >= x) then
        print( A[i])
lc ← leftchild (i)
rc ← rightchild( i)
if( lc ≤ heapsize(A) ) then
        printHeap ( A, lc, x)
if( rc ≤ heapsize (A)) then
        printHeap( A, rc, x)
```

claim: For any heap A with numbers, printHeap( ) will print all elements in A that are greater than or equal to x.

proof. By induction.

Base case: when A[i] is a leaf (which is a node without children),

case I: $A[i] >= x$ then it will print A[i] itself.

case II: $A[i] < x$ then it will do nothing

Inductive step: Assuming printHeap(A, i, x) and printHeap (A, j, x) holds the claim, we need to show the

parent of A[i] and A[j] → A[k] also holds the claim.

Case I: If $A[k] >= x$ then print A[k].

case II: If $A[k] < x$ then do not print A[k]

Then lc ← leftchild (k) ⇒ i → call the function printHeap(A, i, x)

rc ← rightchild (k) ⇒ j → call the function printHeap (A, j, x)

Then it will print all children (including) of A[k] that are greater than or equal to x.

So it will print all elements in heap A that are greater or equal to x.

$T(k) = T(lc) + T(rc) + O(1)$ lc, rc prints the elements greater or equal to x in heap(lc) and heap(rc).

$\because K_{lc} + K_{rc} + 1 \le K_{total}$.

$\therefore T(k) = T(lc) + T(rc) + O(1) \le 2\max\{K_{lc} + K_{rc}\} + O(1)$

$\le 2k + O(1) \rightarrow T(k) \in O(k)$

Dong Boyuan
1547489

Fix some $c = 10$ $n_0 = 10$ $\forall n \geq 10$, we have

$T(k) = T(lc) + T(rc) + O(1) \le 2k + O(1) \le 10k$.

So. $T(k) \in O(k)$

(d). procedure smallest (A, n, k)

```
Build-Min-Heap(A; n)
for(i from 1 to k)
    get min (A)
return A[1]
```

procedure Build-Min-Heap(A; n)

```
heapsize(A) ← n
for( i ← ⌊n/2⌋ down to 1)
    do Min-Heapify(A, i)
```

procedure Min-Heapify (A, i)

```
lc ← leftchild(i)
rc ← rightchild(i)
min ← i
if( lc ≤ heapsize(A) and A[lc] < A[min]) then
    min ← lc
if( rc ≤ heapsize(A) and A[rc] < A[min]) then
    min ← rc
if( min ≠ i) then
    exchange A[i] ↔ A[min]
    Min-Heapify(A, min)
```

procedure get min (A)

```
min ← A[1]
if( heapsize(A) > 1) then
    A[1] ← A[heapsize(A)]
    Min-Heapify(1)
heapsize(A) ← heapsize(A) - 1
```

1. 1st build up a min-Heap of n elements which take $O(n)$ time.
2. 2nd. continue getting the smallest elements of the heap for k times. Each time call getmin() to get the smallest element with decreasing the size of the heap A by 1.
   ① get the root of the heap which is the smallest element of size n.
   ② put the last element at the 1st position to replace the root
   ③ Use min-Heap(1) to get the new smallest element.
   ④ decrease the size of heap. → new root is the new smallest element.
   It will take $O(\log n)$ time

3. So, after getmin( ) was called for k times, the ~~newest~~ latest root will be the k-th smallest element in the array.

Overall; it will take $O(n) + O(k)O(\log(n)) = O(n + k\log n)$ time.

The algorithm is still linear in n only when $k\log n \in O(n)$

$\exists c > 0,\ n_0 = 1\ \forall n \geq 1.$ we have

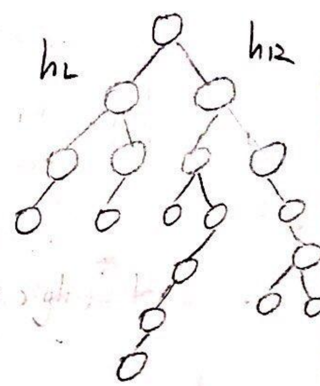
$k\log n \leq c \cdot n \rightarrow k \leq \frac{c \cdot n}{\log n}$ for some $c > 0$.

Dong Boyuan
1547489

So. k is upperbounded by $\frac{c \cdot n}{\log n}$. for some $c > 0$.

1e) procedure getSize (T)

```
if (T = nil) then
    return nil.
if (T.root.left = nil and T.root.right = nil) then
    T.size ← 1
if (T.root.left ≠ nil and T.root.right = nil) then
    T.size ← T.root.left.size + 1
if (T.root.right ≠ nil and T.root.left = nil) then
    T.size ← T.root.right.size + 1
if (T.root.right ≠ nil and T.root.left ≠ nil) then
    T.size ← T.root.left.size + T.root.right.size + 1
```



h_L – the height of left subtree
h_R – the height of right subtree

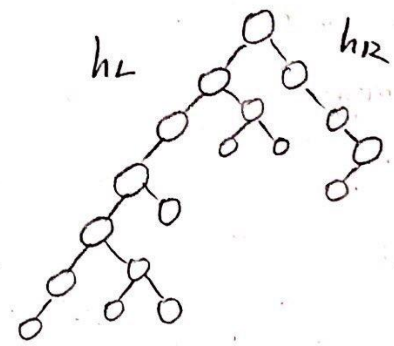
So, it will take $T(h_L) + T(h_R)$ time

$T(h_L) + T(h_R) \leq 2\max\{h_L, h_R\} \in O(2h)$ time $\in O(h)$

Overall, it will take $O(h)$ time in total.

② procedure quantile (T)

```
if (T = nil) then
    return nil
if (T.root.left = nil) then
    T.quantile ← 0
T.quantile ← T.root.left.quantile + 1
```



The runtime takes $O(h_L) \in O(h)$ time. h_L – the height of left subtree

Since, it goes directly to the left, only.

Dong Boyuan
1547489