1)
(a) Write a pseudo-code for Random-Search.
=>
RandomSearch(int[] A, int x) //A[1....n], x is the target
Int n = size of A;
For int i = 0 to n-1 do
int random = random int from i to n-1;
if A[random] == x then return random;
else swap A[i] and A[random];
Return –1;

(b) Assume there is exactly one *i* such that $A[i] = x$. What is the expected number of checks of $A[i] = x$ Random-Search performs before terminates.
=>
Best case: the first random int is the x.
B(n) = O(1)

Worst case: the last random int is the x.
W(n) = O(n)

Average case:
Because the random int is uniformly distributed
P(random int is the x) = 1/n
P(random int is not the x) = 1- P(random int is the x) = (n-1)/n
X = expected number of checks of $A[i] = x$ Random-Search performs before terminates
Pr[X=1] = 1/n
Pr[X=2] = (n-1)/n * 1/(n-1) = 1/n
…….
Pr[X = i] = 1/n
E[X] = Sigma(i = 1 to n) { i* 1/n }
=(n(n+1)/2)/n
=(n+1)/2
2).
Min-Heapify(A, i)
mini= i; n = s(A); l = 2*i+1; r = 2+i+2;
If l <= n and A[l] <A[i] then mini = l else mini = i;
If r <= n and A[r] <A[mini] then mini = r;
If mini is not equal to i then swap A[i] and A[mini];
Min-Heapify(A, mini);

```
Heap-minimum(A)
Return A[0];

Heap-Extract-Min(A)
If s(A)<1 then output error message and return;
Mini = A[1]; A[1] =A[s(A)]; Heap size --;
Min-Heapify(A,1);
    Return mini;

Heap-Decrease-Key(A,i,key)
If key > A[i] then output error message and return;
A[i] = key;
    For i>1 and A[(i-1)/2]>A[i] do
        swap A[i] and A[(i-1)/2];
    I = (i-1)/2;

Min-Heap-Insert (A,key)
S(A)++;
A[s(A)] = infinity;
Heap-Decrease-Key(A,s(A),key)
```

3).

(a) How would you represent a *d*-ary heap in an array *A*[1..*n*]?

Current index= i

Child of A[i]: i*d-d+2, ….., i*d, i*d+1

Parent of A[i] = (i-1)/d taking upper limit(e.g. 2.3 has upper limit of 3)

(b) What is the height of a *d*-ary heap of *n* elements in terms of *n* and *d*?

(c) Give an effiffifficient pseudo code of EXTRACT-MAX in a *d*-ary max-heap. Analyze its running time in terms of *d* and *n*.

```
EXTRACT-MAX (A,n)
Max = A[1]; A[1] = A[s(A)];Heap size --;
Max-Heapify(A,1,n,d);
```

```
Max-Heapify(A,i,n,d)
Max = i;
    For j = 0 to d-1 do
if i*d+j <= n and A[d*i+j]>A[Max] then max = d*i+j;
If max is not equal to i then swap A[i] and A[max];
Max-Heapify(A,max,n,d);
```

The heapify have time t(n) =O(d*log(d,n)), thus EXTRACT-MAX has O(d*log(d,n))

(d) Give an efficient pseudo code of INSERT in a *d*-ary max-heap. Analyze its running time in terms of *d* and *n*.

```
INSERT(A,key,n,d)
S(A)++;
A[s(A)] = -infinity;
INCREASE-KEY(A,s(A),key,n)

INCREASE-KEY(A,i,key)
A[i] = key;
    For i>1 and A[(i-1)/d]<A[i] do
        swap A[i] and A[(i-1)/d];
    I = upper limit( (i-1)/d);
```

The heap-increase-key has t(n) = O(log(d,n)), INSERT calls it once so INSERT has t(n)= O(log(d,n))

(e) Give an efficient pseudo code of INCREASE-KEY(*A, i, k*), which flags an error if $k < A[i]$, but otherwise sets $A[i] = k$ and then updates the *d*-ary max-heap structure appropriately. Analyze its running time in terms of *d* and *n*.

```
INCREASE-KEY(A,i,key)
If key < A[i] then output error message and return.
A[i] = key;
    For i>1 and A[(i-1)/d]<A[i] do
        swap A[i] and A[(i-1)/d];
    I = upper limit( (i-1)/d);
```

The INCREASE-KEY increase the value of target node and then maintain the max-dary-heap in O(log(d,n)).

4).

	N = 1000	N = 2000	N = 4000	N = 8000	N = 16000	N = 32000
Randomized	4576000n	8019800ns	9105000ns	13537200ns	22345200ns	46890500ns

quicksort	s					
New quicksort	2494000ns	592000ns	1747300ns	5706900ns	17120700ns	51362100ns

The recursion stops when n/2^(depth) = k, so depth = log(n/k). Thus T(quicksort part) = O(nlog(n/k)). Because the returned subarray have n/k subarrays of size k, and the average big O of insertion sort is O(n^2) in this case is O(k^2), so T(insertion sort in this case) = O((n/k)*k^2)=O(kn). Thus the new quicksort has O($nk + n \log(n/k)$). For the value of k, because the insertiuon sort needs to be faster than quicksort(O(nlogn)), so kn <= nlogn, k<=logn. So the k should be selected in the domain of [1, logn]. In this question, the smalled size n = 1000, so the k should be 10 in this question.

```
class Main
{
    public static void insertionSort(int[] A, int l, int n)
    {
        for (int i = l + 1; i <= n; i++)
        {
            int value = A[i];
            int j = i;

            while (j > l && A[j - 1] > value)
            {
                A[j] = A[j - 1];
                j--;
            }
            A[j] = value;
        }
    }

    public static int partition (int[] A, int l, int r)
    {

        int pivot = A[r];

        int index = l;
        for (int i = l; i < r; i++)
        {
            if (A[i] <= pivot)
            {
                int temp = A[i];
                A[i] = A[index];
                A[index] = temp;
```

```
            index++;
        }
    }

    int temp = A[r];
    A[r] = A[index];
    A[index] = temp;

    return index;
}

public static int randomizedPartition(int[] A, int l, int r){
    Random random = new Random();
    int i = random.nextInt(r-l+1) + l; // random.nextInt(max - min + 1) + min
    int temp = A[i];
    A[i] = A[r];
    A[r] = temp;
    return partition(A,l,r);
}

public static void QuickSort(int[] A, int l, int r)
{
    if (l >= r)
        return;


    int pivot = randomizedPartition(A, l, r);

    QuickSort(A, l, pivot - 1);
    QuickSort(A, pivot + 1, r);
}

public static void newQuickSort(int[] A, int l, int r)
{
    while (l < r)
    {
        if (r - l < 10)
        {
            insertionSort(A, l, r);
            break;
        }
        else
        {
            int pivot = randomizedPartition(A, l, r);
```

```
            if (pivot - l < r - pivot) {
                newQuickSort(A, l, pivot - 1);
                l = pivot + 1;
            } else {
                newQuickSort(A, pivot + 1, r);
                r = pivot - 1;
            }
        }
    }
}

public static void main(String[] args)
{
    int[] A1 = new int[1000];
    int[] A2 = new int[2000];
    int[] A3 = new int[4000];
    int[] A4 = new int[8000];
    int[] A5 = new int[16000];
    int[] A6 = new int[32000];

    Arrays.fill(A1, new Random().nextInt());
    Arrays.fill(A2, new Random().nextInt());
    Arrays.fill(A3, new Random().nextInt());
    Arrays.fill(A4, new Random().nextInt());
    Arrays.fill(A5, new Random().nextInt());
    Arrays.fill(A6, new Random().nextInt());

    int[] temp = Arrays.copyOf(A1, 1000);
    long begin, end;

    begin = System.nanoTime();
    QuickSort(A1, 0, 1000-1);
    end = System.nanoTime();

    System.out.println("Time for randomized quickSort: " + (end-begin) + "ns"+"\n");

    begin = System.nanoTime();
    newQuickSort(temp, 0, 1000-1);
    end = System.nanoTime();

    System.out.println("Time for new quickSort: " + (end-begin) + "ns"+"\n");

}
```

}

5).
insertion sort and merge sort are stable, because they only change the position of elements which need to change, such as 2,1 to 1,2. If A[1,1,1,1] is sorted with insertion sort or merge sort, the positions of the elements are not changed.

To make a sorting algorthm stable, we can create an array of indexs which each element i is pointing to ith element of the unsorted array and store its original position. By doing this, every time the algorthm compares the elements that have same value, if they have same value then skip the comparation to maintain the original position is in increasing order. The running time is not changed since it does not affect the algorthm itself. But the space effiviency is O(n) since it requires creating an array.

6).
(a).input A[0....n], return an sorted array B[0....n]

1.traverse the array to find the count1 which is the num of 0's

2.creat an array B and have a pointer i pointing to the first element of A, and index j pointing to the A[count1].

3. traverse A and insert the 0 to B[i], 1 to B[j], then i++, j++

4. return B

(b). input A[0....n], change A in place

1.let i pointing to A[0], j pointing to A[n].

2.if A[i] = 1 and A[j] = 1 then decreasing j untill A[j] = 0 and change A[i] = 0, A[j] = 1, i ++, j--

if A[i] = 0 and A[j] = 1 then i++, j--

if A[i] = 1 and A[j] = 0 then change A[i] = 0, A[j] = 1, i++, j--

if A[i] = 0 and A[j] = 0 then increasing i untill A[i] = 1 and change A[i] = 0, A[j] = 1, i++, j--

Keep compare and stops if i meets j

(c). Insertion Sort

7).
(a). $x_i < x_k$ if $w_i < w_k$ or $w_i = w_k$ and i < k. Because wi are all same, the xi<xk if only i < k. Thus weighted median of $x_1, .., x_n$ with weight $w_i = 1/n$ for $1 \le i \le n$ is the median

(b).using mergesort to sort x1,….,xn in increasing order by i. Thn traverse the x and count the weights, stops if wi causes the count exceeds ½, then return xi.

(c)If n<=2 then return the brute force solution. Else find the median of A and apply partition. If the median of A, m(A), has all left weight count < ½ and all right weight count<½ then return m(A), else search the part which has weight count >½ untill find the weighted median and return.

Because the SELECT devide the subarray at every iteration thus T(n) = Θ(*n*) at worst-case time.