1)
a)

```
LAZY-SAMPLE(m, n)
1       RANDOMIZE-IN-PLACE(A, n)
2       return A[1...m]

RANDOMIZE-IN-PLACE(A, n)
1       for i = 1 to n
2               swap A[i] with A[RANDOM(i, n)]
```

LAZY-SAMPLE would make a call to RANDOMIZE-IN-PLACE which creates a permutation of the array A in-place. RANDOMIZE-IN-PLACE in turn makes n calls to RANDOM. LAZY-SAMPLE then return first m elements of permuted array A.

Following the similar arguments for RANDOMIZE-IN-PLACE as discussed in class:
For RANDOMIZE-IN-PLACE, Loop invariant:
Before the i-th iteration, for each possible (i-1)-combination of the n elements, the subarray A[1...i-1] contains this (i-1)-combination with probability $1/{}^{n}C_{i-1}$ = [(n-i+1)!*(i-1)!]/n!

Initialization: i = 1, array is of size 0, A[1...0] contains any 0-combination with probability 1

Maintenance: Assume A[1...i-1] satisfies the loop invariant. Each possible (i-1)-combination appears in A[1...i-1] with probability [(n-i+1)!*(i-1)!]/n!

Add A[i] from A[i...n]

Consider a particular i-combination $\{x_1, x_2, \ldots x_i\}$
E1: event that any combination of $\{x_1, \ldots x_{i-1}\}$ is in A[1...i-1]. Note that we are not concerned with the ordering of elements.
E2: event that x_i becomes a part of this combination.
Both E1 and E2 need to occur for the i-combination $\{x_1, x_2, \ldots, x_i\}$ to be in A[1...i]

$Pr\{E2 \cap E1\} = Pr\{E2|E1\} * Pr\{E1\}$
Pr{E2|E1} = i/(n-i+1) because i-th element can be placed at any of the i slots in between or start/end of i-1 numbers.

=> $Pr\{E2 \cap E1\} = Pr\{E2|E1\} * Pr\{E1\}$
= [i/(n-i+1)] * [(n-i+1)!*(i-1)!/n!]
= (n-i)!*i!/n!

Termination: As we are just concerned with first m numbers, we calculate the probability corresponding to those numbers. Hence, i=m for termination.
=> Pr = [(n-m)!*m!]/n! = $1/({}^{n}C_{m})$
b)

Consider the invariant:
QUICK-SAMPLE(m, n) returns a uniformly distributed combination.

Proof by induction:
For m=1, it holds because QUICK-SAMPLE would make one recursive call and add a random element to the set.
Let's assume that it holds for m-1 for any value of m, and consider the case for m:
There are two possible cases:

1) n is included in the set S:
   E1: n is part of m-subset
   E2: Select remaining m-1 from n-1 elements
   Pr{E1}: One place in m-slots is fixed with n and we need to select remaining m-1 from n-1 elements = ${}^{n-1}C_{m-1}/{}^{n}C_{m} = m/n$
   Pr{E2}: Now, for the remaining positions select any one of ${}^{n-1}C_{m-1}$ combinations and each combination is equally likely = $1/{}^{n-1}C_{m-1}$
   Probability of getting the combination (using chain rule) = $P(E1 \cap E2)$ = $Pr\{E1\}*Pr\{E2|E1\} = (m/n)*(1/{}^{n-1}C_{m-1}) = 1/{}^{n}C_{m}$
2) n is not included in the set S:
   E1: n is not part of m-subset
   E2: Selecting all m from n-1 elements
   Pr{E1}: 1 - Pr(E1 from part 1) = $1 - {}^{n-1}C_{m-1}/{}^{n}C_{m} = 1-m/n$
   Pr{E2}: Now, for the remaining positions select any one of ${}^{n-1}C_{m}$ combinations and each combination is equally likely = $1/{}^{n-1}C_{m}$
   Probability of getting the combination (using chain rule) = $P(E1 \cap E2)$ = $Pr\{E1\}*Pr\{E2|E1\} = (1-{}^{n-1}C_{m-1}/{}^{n}C_{m})*(1/{}^{n-1}C_{m}) = (1-m/n)*(1/{}^{n-1}C_{m}) = 1/{}^{n}C_{m}$

As probability of getting a m-subset is $1/{}^{n}C_{m}$ and there are in all ${}^{n}C_{m}$ m-subsets, therefore each of them are equally likely.

2)
a)
Assuming root index is 1.
k-th child of i-th node would be at index $d*(i-1) + k + 1$ [k varies from 1 to d]
Parent of i-th node would be at index $\lfloor (i-2)/d \rfloor + 1$

b)
Counting all nodes in a tree of height h with total number of nodes as n:
$d^0 + d^1 + d^2 + \ldots + d^h = (d^{h+1}-1)/(d-1)$
$(d^{h+1}-1)/(d-1) = n$
$\Rightarrow h = \log_d(nd-n+1)$
$\Rightarrow h = O(\log_d n)$

c)
EXTRACT-MAX would remain the same as for binary heaps.

```
EXTRACT-MAX(A, n)
1       if n < 1
2               error “heap underflow”
3       max = A[1]
4       A[1] = A[n]
5       MAX-HEAPIFY(A, 1, n-1)
6       return max
```

The subroutine MAX-HEAPIFY would change slightly and is as follows:

```
MAX-HEAPIFY(A, i, n)
1       largest = i
2       start = d*(i-1) + 2
3       end = d*i + 1
4       for j = start to end
5               if A[j] > largest
6                       largest = j
7       if largest ≠ i
8               exchange A[i] with A[largest]
9               MAX-HEAPIFY(A, largest, n)
```

Instead of just comparing the node with its left and right child, we compare it with all its d children to check for the largest value. If the largest value is one of the children, we swap the node with that child and then call MAX-HEAPIFY on the node again to let the node sink down.

MAX-HEAPIFY’s complexity is $O(d*\log_d n)$. Worst case occurs when the node sinks down from root to a leaf which takes time equivalent to the height of the heap. d comes in complexity because we have to traverse over all the children to find the largest value.
EXTRACT-MAX takes some constant time and a call to MAX-HEAPIFY, which makes its time complexity as $O(d*\log_d n)$.

d)
INCREASE-KEY would remain same as for binary heaps.

```
INCREASE-KEY(A, i, k)
1       if k < A[i]
2               error “new key is smaller than current key”
3       A[i] = k
4       while i > 1 and A[Parent(i)] < A[i]
5               exchange A[i] with A[Parent(i)]
6               i = Parent(i)
```

INCREASE-KEY's complexity is $O(\log_d n)$. Worst case occurs when we assign a leaf a value largest than all the values in the heap. In that case, this node would float up from leaf to the root position and hence time complexity becomes equivalent to the height of the heap.

3 . a)

```
MAX-HEAP-EXTRACT-MIN(A)
1 for i = A.length downto A.length/2  \\ extract leaf nodes
2        X1[i] = A[i]
3        index[i] = i  \\ store the index of each leaf node
4 end for
5 for j = 1 to X1.length  \\ linear search for the minimum element
6        if min < X1[j]
7                min = X1[j]
8        min-index = index[j]  \\ index of the leaf node with minimum value
9 end for
10 exchange A[min-index] with A[A.length]
11 A.heapsize = A.heapsize-1
12 for i = min-index; i >1; i = Parent(i) // Compare with parent node for each node
13       if A[Parent(i)] < A[i]  // if parent is smaller , exchange
14               exchange A[Parent(i) with A[i]
15       Else break \\ heap property already maintained
16 end for
15 return min

PARENT(i)
{
        Return i/2 (ceiling);
}
```

Time complexity - O(n)

Proof of Correctness:
We proof the correctness by assuming that lines 5-9 identifies the min from the leaf nodes and the following loop invariant holds -
Loop Invariant : At the start of each iteration of the **for** loop of lines 12-16, the subarray A[1…A.heap-size] satisfies the max-heap property, except that there may be one violation: A[i] may be larger than A[PARENT(i)].

Initialization: A is a heap except that A[i] might be larger than its parent, because it has been modified. A[i] is larger than its children, because otherwise the guard clause would fail and the loop will not be entered (the new value is larger than the old value and the old value is larger than the children).

Maintenance: When we exchange A[i] with its parent, the max-heap property is satisfied except that now A[PARENT(i)] might be larger than its parent. Changing i to its parent maintains the invariant.
Termination: The loop terminates whenever the heap is exhausted or the max-heap property for A[i] and its parent is preserved. At the loop termination, A is a max-heap.

b) MY-HEAP-SORT(A)
1 BUILD-MAX-HEAP(A) \\ built MAX heap
2 for i = A.length downto 2
3 min = MAX-HEAP-EXTRACT-MIN(A) // return smallest elements
4 end for

Time complexity: O(n^2)

3. b) Proof of Correctness
Correctness can be proved by using the following loop invariant:
Loop Invariant : At the start of each iteration of the for loop of lines 2-5, the subarray A[1..i] is a max-heap containing the i largest elements of A[1..n], and the subarray A[i+1..n] contains the n-i smallest elements of A[1..n], sorted.

Initialization: Prior to the first iteration of the loop, i = n. The subarray A[1..i] is a max-heap due to BUILD-MAX-HEAP. The subarray A[i + 1..n] is empty, hence the claim about it being sorted is vacuously true.

Maintenance: For each iteration, By the loop invariant, A[1..i] is a max-heap containing the i largest elements of A[1..n] because of the MAX-HEAP-EXTRACT-MIN that extracts the smallest element from A[1…n]. The function call MAX-HEAP-EXTRACT-MIN makes the subarray A[i..n] contain the (n−i+1) smallest elements of A[1..n], sorted. The heap size of A is decreased in to heap-size(A) = i − 1. Though at first, the subarray A[1..i−1] violates a max-heap after the extracting the min, the max heap property is restored by calling the above function which leaves a max-heap in A[1..i−1]. Consequently, the loop invariant is reestablished for the next iteration.

Termination: At termination, i = 1. By the loop invariant, the subarray A[2.. n] contains the n − 1 smallest elements of A[1..n], sorted, and thus, A[1] contains the largest element of A[1..n]. A[1..n] is a sorted array.

4. a) At the end of the loop the variables has the following values (answers will be different for different values of p and r)

Array : 13 19 9 5 12 8 7 4 11 2 6 21

Other partition p =0 r = 11

Iteration 0 : 13 19 9 5 12 8 7 4 11 2 6 21
Iteration 1 : 6 19 9 5 12 8 7 4 11 2 13 21
Iteration 2 : 6 2 9 5 12 8 7 4 11 19 13 21

x =13 i = 9 j = 8

b) When we check for i<j, we get i=p and j≥p since A[p]=x. If i=j, the algorithm will terminate. If i<j, the next loop will also have indices i and j within the array, (because i≤r and j≥p). Note that if one of the indices gets to the end of the array, then i won't be less or equal to j any more.

c) As for the return value, it will be at least one less than j. At the first iteration, either (1) A[p] is the maximum element and then i=p and j=p< r or (2) it is not and A[p] gets swapped with A[j] where j≤r. The loop will not terminate and on the next iteration, j gets decremented (before eventually getting returned). Combining those two cases we get p≤j<r.

4 d. We prove the following loop invariant: We claim that the following statement is true at every execution of line 11: Every entry of A[p...(i-1)] is less than equal to x , every entry of A[(j+1)...r] is greater than or equal to x

Initialization: This is maintained by the first two repeat blocks. The first time we reach line 11, the test A[i'] > x was successful for every p <= i' < i, and the test A[j'] < x was successful for every j < j' < r, but the tests A[i] > x and A[j] < x were unsuccessful.

Maintenance: By exchanging A[i] and A[j] we make the A[p…i] ≤ x and A[j...r] ≥ x. Incrementing i and decrementing j maintain this invariant. Consider reaching line 11 after the first time. Let's denote i1 < j1 and we perform the swap on line 12. Then all entries of A[1...(j1-1)] are less than or equal to x and all entries of A[(j1+1)...r] are greater than or equal to x; but in fact, since we swapped A[i1] and A[j1], even all entries of A[1...i1] are less than or equal to x and all entries of A[j1...r] are greater than or equal to x. Thus we have maintained the invariant.

Termination: For the program to terminate, we must have i >= j. Then the loop invariant says that every entry of A[p...(i-1)] is less than or equal to x and every entry of A[(j+1)...r] is greater than equal to x, while A[i] >= x and A[j] <= x. If i=j, this means that A[j] = x, so every entry of A[p...j] is either an entry of A[p...(i-1)] or A[j], hence is less than or equal to x, which is in turn less than or equal to every entry of A[(j+1)...r]. If i > j, then every entry of A[p...j] is an entry of A[p...(i-1)], hence is less than or equal to x, which is in turn less than or equal to every entry of A[(j+1)...r]. Thus the invariant still holds at termination.

4. e)
OTHER-PARTITION(A,p,r)
1 x = A[p]
2 i = p-1

```
3 j=r
4 while(true)
5       while(!(A[j] <= x))  j--
6       while(!(A[i] >= x))  i++
7       if i < j
8               swap (A[i], A[j])
9       else
10              return j
11 end while
```

```
QUICKSORT(A, p, r)
1 if p < r
2       q = OTHER-PARTITION(A,p,r)
3       QUICKSORT(A, p, q)
4       QUICKSORT(A, q+1, r)
```

This algorithm is commonly known as HOARE Partition.

References:
Q4.b,c - http://clrs.skanev.com/07/problems/01.html