• Heap & heapsort
• Binary Search Tree
• AVL-tree
• Graph: BFS & DFS
• Dijkstra’s Algorithm
• Bellman-Ford Algorithm
Review

• Heap & heapsort
  • Priority Queue is a data structure maintaining a set of elements, each element has an index and a key value. One only can remove the first element and insert a new element to the end.
  • Heap is a special priority queue, MaxHeap: \( A[i] \geq A[2i], A[2i + 1] \); \( A[1] \) is the largest key value. MinHeap: “\( \leq \)”. Can be visualized by a “nearly complete binary tree”.
  • Build MaxHeap: from \( n/2 \) back to 1, call maxheapify each time. Time complexity is \( O(n \log n) \).
Review

• Heap sort
  • Build a MaxHeap.
  • Repeat the following steps n times:
    • Swap the key values of the first and final elements.
    • Reduce the size of the heap by one.
    • Run MaxHeapify(A, 1).
**Review**

```plaintext
BuildMaxHeap(A)
1:  For i = n/2 to 1
2:     MaxHeapify(A, i)
3:  End For

MaxHeapify(A, i)
1:  if A[i] \geq \max\{A[2i], A[2i + 1]\} then
2:      Do nothing
3:  else
4:      if A[2i] = \max\{A[2i], A[2i + 1]\} then
5:         Swap the key values of A[i] and A[2i]
6:         MaxHeapify(A, 2i)
7:      else
8:         Swap the key values of A[i] and A[2i + 1]
9:         MaxHeapify(A, 2i+1)
10:  end if
11: end if
```
Heap & Heapsort

• Q1:

Heap & Heapsort

• A1:

Heap & Heapsort

• A1:
Heap & Heapsort

• A1:
Heap & Heapsort

• A1:
Heap & Heapsort

• A1:
Heap & Heapsort

• A1:
Heap & Heapsort

• A1:
Heap & Heapsort

• A1:
Heap & Heapsort

• A1:

25, 13, 20, 8, 7, 17, 2, 5, 4
Heap & Heapsort

• A1:
  • Swap the key values of the first and final elements.
  • Reduce the size of the heap by one.
  • Run MaxHeapify(A, 1).

25, 13, 20, 8, 7, 17, 2, 5, 4
Heap & Heapsort

• A1:
  • Swap the key values of the first and final elements.
  • Reduce the size of the heap by one.
  • Run MaxHeapify(A, 1).
Heap & Heapsort

• A1:
  • Swap the key values of the first and final elements.
  • Reduce the size of the heap by one.
  • Run MaxHeapify(A, 1).

4, 13, 20, 8, 7, 17, 2, 5, 25
Heap & Heapsort

• A1:
  • Swap the key values of the first and final elements.
  • Reduce the size of the heap by one.
  • Run MaxHeapify(A, 1).

20, 13, 17, 8, 7, 4, 2, 5, 25
Heap & Heapsort

• A1:
  • Swap the key values of the first and final elements.
  • Reduce the size of the heap by one.
  • Run MaxHeapify(A, 1).

5, 13, 17, 8, 7, 4, 2, 20, 25
Heap & Heapsort

• A1:
  • Swap the key values of the first and final elements.
  • Reduce the size of the heap by one.
  • Run MaxHeapify(A, 1).

5, 13, 17, 8, 7, 4, 2, 20, 25
Heap & Heapsort

- A1:
  - Swap the key values of the first and final elements.
  - Reduce the size of the heap by one.
  - Run MaxHeapify(A, 1).
Heap & Heapsort

- A1:
  - Swap the key values of the first and final elements.
  - Reduce the size of the heap by one.
  - Run MaxHeapify(A, 1).
Heap & Heapsort

• A1:
  • Swap the key values of the first and final elements.
  • Reduce the size of the heap by one.
  • Run MaxHeapify(A, 1).
Heap & Heapsort

A1:

- Swap the key values of the first and final elements.
- Reduce the size of the heap by one.
- Run MaxHeapify(A, 1).
Heap & Heapsort

• A1:
  • Swap the key values of the first and final elements.
  • Reduce the size of the heap by one.
  • Run MaxHeapify(A, 1).
Heap & Heapsort

- A1:
  - Swap the key values of the first and final elements.
  - Reduce the size of the heap by one.
  - Run `MaxHeapify(A, 1)`.
Heap & Heapsort

- A1:
  - Swap the key values of the first and final elements.
  - Reduce the size of the heap by one.
  - Run MaxHeapify(A, 1).

2, 4, 5, 7, 8, 13, 17, 20, 25
Review

• Binary Search Tree (BST)
  • A binary tree (children <=2);
  • Each node x;
  • Key value key(x);
  • Pointers Left(x), right(x), and parent(x);
  • For each node x, the key values in its left sub-tree \(\leq \text{key}(x)\) \(\leq\) the key values in its right sub-tree.
Binary Search Tree

• Q3:

Let $T$ be a binary search tree, and let $x$ be a key. Give an efficient algorithm for finding the smallest key $y$ in $T$ such that $y > x$. Note that $x$ may or may not be in $T$. Explain why your algorithm has the running time it does.
Binary Search Tree

• A3:

if \( v \leq x \) then \( y \) is the smallest thing larger than \( x \) in \( v \)’s right subtree (if \( v = x \) then \( y \) is the smallest thing in the right subtree, but this is just a special case of \( y \) being in the right subtree and does not need to be treated separately)
Binary Search Tree

• A3:

if $v > x$ then $y$ is the smaller of $v$ and the smallest thing greater than $x$ in $v$’s left subtree (it is not necessary to consider $v$’s right subtree because while everything there is larger than $x$, it is also larger than $v$)
Binary Search Tree

- **A3:**

  Algorithm `findFloor(T, x)`:  
  
  *Input:* a binary search tree $T$ and a key $x$  
  
  *Output:* the smallest $y$ in $T$ such that $y > x$ if such a $y$ exists, and $\infty$ otherwise  

  $current \leftarrow T.root()$  
  $best \leftarrow null$  

  **while** $(T.isInternal(current))$ **do**  
  
  **if** $(current.key() \leq x)$ **then**  
  
  $current \leftarrow T.rightChild(current)$  
  \{ $y$ is in right subtree \}  
  
  **else**  
  
  $best \leftarrow current.key()$  
  \{ current node is a candidate for $y$ \}  
  
  $current \leftarrow T.leftChild(current)$  
  \{ $y$ is in left subtree \}  

  $best \leftarrow \min(best, current.key())$  

  **return** $best$  

  The running time is $O(h)$, where $h$ is the height of the tree, because in each iteration of the **while** loop, $current$ advances one level down the tree. Note that in the worst case this is $O(n)$ and in the best case it is $O(\log n)$. 
Binary Search Tree

- A3:

**Algorithm** `findFloor(T, x)`:

  *Input*: a binary search tree $T$ and a key $x$
  
  *Output*: the smallest $y$ in $T$ such that $y > x$ if such a $y$ exists, and $\infty$ otherwise

  
  ```
  current ← $T$.root()
  best ← null
  
  while ($T$.isInternal(current)) do
    if (current.key() \leq x) then
      current ← $T$.rightChild(current)
      \{ $y$ is in right subtree \}
    else
      best ← current.key()
      current ← $T$.leftChild(current)
      \{ current node is a candidate for $y$ \}
      \{ $y$ is in left subtree \}
    
  return best
  ```
Binary Search Tree

- A3:

Algorithm findFloor(T, x):

Input: a binary search tree T and a key x

Output: the smallest y in T such that y > x if such a y exists, and ∞ otherwise

current ← T.root()
best ← null

while (T.isInternal(current)) do

if (current.key() ≤ x) then

    current ← T.rightChild(current)  

else

    best ← current.key()  

    current ← T.leftChild(current)  

end

best ← min(best, current.key())  

return best

Key: x = 24
Best = null

Current node

30

23

26

11

40
Binary Search Tree

- A3:

**Algorithm** findFloor($T, x$):

*Input:* a binary search tree $T$ and a key $x$

*Output:* the smallest $y$ in $T$ such that $y > x$ if such a $y$ exists, and $\infty$ otherwise

1. $current \leftarrow T.root()$
2. $best \leftarrow null$
3. while ($T.isInternal(current)$) do
   1. if ($current.key() \leq x$) then
      1. $current \leftarrow T.rightChild(current)$ \{ $y$ is in right subtree \}
   2. else
      1. $best \leftarrow current.key()$ \{ current node is a candidate for $y$ \}
      2. $current \leftarrow T.leftChild(current)$ \{ $y$ is in left subtree \}
4. $best \leftarrow \min(best, current.key())$
5. return $best$
Binary Search Tree

• A3:

Algorithm $\text{findFloor}(T, x)$:

Input: a binary search tree $T$ and a key $x$

Output: the smallest $y$ in $T$ such that $y > x$ if such a $y$ exists, and $\infty$ otherwise

1. $\text{current} \leftarrow T.\text{root}()$
2. $\text{best} \leftarrow \text{null}$
3. while ($T.\text{isInternal}(\text{current})$) do
   4. if ($\text{current}.\text{key}() \leq x$) then
      5. $\text{current} \leftarrow T.\text{rightChild}(\text{current})$
         \{ $y$ is in right subtree \}
   6. else
      7. $\text{best} \leftarrow \text{current}.\text{key}()$
         \{ current node is a candidate for $y$ \}
      8. $\text{current} \leftarrow T.\text{leftChild}(\text{current})$
         \{ $y$ is in left subtree \}
8. $\text{best} \leftarrow \text{min}(\text{best}, \text{current}.\text{key}())$
9. return $\text{best}$
Binary Search Tree

• A3:

**Algorithm** findFloor(T, x):

*Input:* a binary search tree T and a key x

*Output:* the smallest y in T such that y > x if such a y exists, and \( \infty \) otherwise

1. current ← T.root()
2. best ← null
3. while (T.isInternal(current)) do
   4. if (current.key() \leq x) then
      5. current ← T.rightChild(current)  \{ y is in right subtree \}
   6. else
      7. best ← current.key()  \{ current node is a candidate for y \}
      8. current ← T.leftChild(current)  \{ y is in left subtree \}
8. best ← min(best, current.key())
9. return best

Key: \( x = 24 \)

Best = 30
Binary Search Tree

- **A3:**

  **Algorithm** findFloor\(T, x\):
  
  **Input:** a binary search tree \(T\) and a key \(x\)
  
  **Output:** the smallest \(y\) in \(T\) such that \(y > x\) if such a \(y\) exists, and \(\infty\) otherwise
  
  \[
  \text{current} \leftarrow T.\text{root}\()
  \]
  
  \[
  \text{best} \leftarrow \text{null}
  \]
  
  while \((T.\text{isInternal}\text{(current))})\) do
  
  if \((\text{current.key()} \leq x)\) then
    \[
    \text{current} \leftarrow T.\text{rightChild}\text{(current)}
    \]
    \[
    \text{current} \leftarrow T.\text{leftChild}\text{(current)}
    \]
    \[
    \text{best} \leftarrow \text{min}\text{(best, current.key()})
    \]
  else
    \[
    \text{best} \leftarrow \text{current.key()}
    \]
    \[
    \text{current} \leftarrow T.\text{leftChild}\text{(current)}
    \]
    \[
    \text{current} \leftarrow T.\text{rightChild}\text{(current)}
    \]
  return \text{best}

Key: \(x = 24\)

\(\text{Best} = 30\)
Binary Search Tree

- A3:

Algorithm findFloor(T, x):

Input: a binary search tree T and a key x
Output: the smallest y in T such that y > x if such a y exists, and \( \infty \) otherwise

\[\begin{align*}
current &\leftarrow T.\text{root()} \\
best &\leftarrow \text{null}
\end{align*}\]

while (\( T.\text{isInternal(current)} \)) do

if (\( current.key() \leq x \)) then

\[\begin{align*}
current &\leftarrow T.\text{rightChild(current)} \\
\end{align*}\]

else

\[\begin{align*}
best &\leftarrow current.key() \\
current &\leftarrow T.\text{leftChild(current)}
\end{align*}\]

\[\begin{align*}
best &\leftarrow \min(best, current.key())
\end{align*}\]

return best

Key: \( x = 24 \)
Best = 30
Binary Search Tree

- **A3:**

  **Algorithm** findFloor($T, x$):

  *Input*: a binary search tree $T$ and a key $x$

  *Output*: the smallest $y$ in $T$ such that $y > x$ if such a $y$ exists, and $\infty$ otherwise

  $current \leftarrow T.\text{root}()$

  $best \leftarrow \text{null}$

  while ($T.\text{isInternal}(current)$) do

    if ($current.\text{key}() \leq x$) then

      $current \leftarrow T.\text{rightChild}(current)$  \hspace{1cm} \{ $y$ is in right subtree \}

    else

      $best \leftarrow current.\text{key}()$  \hspace{1cm} \{ current node is a candidate for $y$ \}

      $current \leftarrow T.\text{leftChild}(current)$  \hspace{1cm} \{ $y$ is in left subtree \}

    $best \leftarrow \text{min(best, current.\text{key}())}$

  return $best$

---

Key: $x = 24$

Best = 30
Binary Search Tree

• A3:

Algorithm findFloor(T, x):

Input: a binary search tree T and a key x
Output: the smallest y in T such that y > x if such a y exists, and \( \infty \) otherwise

```plaintext
current ← T.root()
best ← null
while (T.isInternal(current)) do
    if (current.key() \leq x) then
        current ← T.rightChild(current)  \{ y is in right subtree \}
    else
        best ← current.key()  \{ current node is a candidate for y \}
        current ← T.leftChild(current)  \{ y is in left subtree \}

best ← \min(best, current.key())
return best
```

Key: \( x = 24 \)
Best = 30
Binary Search Tree

• A3:

**Algorithm** `findFloor(T, x)`: 

*Input*: a binary search tree $T$ and a key $x$  
*Output*: the smallest $y$ in $T$ such that $y > x$ if such a $y$ exists, and $\infty$ otherwise

1. $current \leftarrow T.root()$
2. $best \leftarrow \text{null}$
3. While ($T.isInternal(current)$) do
   1. If ($current.key() \leq x$) then
      1. $current \leftarrow T.rightChild(current)$  
         \{ $y$ is in right subtree \}
   2. Else
      1. $best \leftarrow current.key()$  
         \{ current node is a candidate for $y$ \}
      2. $current \leftarrow T.leftChild(current)$  
         \{ $y$ is in left subtree \}
5. $best \leftarrow \min(best, current.key())$
6. Return $best$

*Output*: $\infty$
Review

• AVL-tree
  • A balanced binary search tree \((Height = \Theta(\log n))\)
  • For each node, the difference between the heights of its left and right sub-trees is at most 1
  • Height(leaf) = 1, Height(null) = 0, Height(x) = \text{max}\{Height(left(x)), Height(right(x))\}+1

• Build AVL-tree
  • First insert as in binary search tree (BST).
  • Then restoring AVL property by modification (LR, RR, RRLR, LRRR).
• Modification (LR)
Review

- Modification (RR+LR)

- Right - rotate (z)

- Left - rotate (x)

- B and C at least one
  - k - 1
Review

• Modification (RR)
Review

• Modification (LR+RR)

Left - rotate(z)

Right - rotate(x)

B and C at least one $k - 1$
AVL-tree

• Q1: insert 23 and 55?
AVL-tree

• A1: insert 23
AVL-tree

- A1: insert 23
AVL-tree

• A1: insert 23
AVL-tree

• A1: insert 23
AVL-tree

• A1: insert 23
AVL-tree

• A1: insert 55
AVL-tree

- A1: insert 55
AVL-tree

• A1: insert 55
A graph $G = (V, E)$ consists of a vertex set $V$ and an edge set $E$. $|V| = n$ and $|E| = m$.

Each edge $e = (u, v) \in E$ is an unordered pair of vertices.

If $(u, v) \in E$, we say $v$ is a neighbor of $u$.

The degree $\text{deg}(u)$ of a vertex $u$ is the number of edges incident to $u$. 
Review

Definition
- If the two end vertices of $e$ are ordered, the edge is directed, and we write $e = u \rightarrow v$.
- If all edges are directed, then $G$ is a directed graph.
- The in-degree $\deg_{in}(u)$ of a vertex $u$ is the number of edges that are directed into $u$.
- The out-degree $\deg_{out}(u)$ of a vertex $u$ is the number of edges that are directed from $u$. 
Review

Adjacency List Representation

- For each vertex \( v \in V \), there’s a **linked list** \( \text{Adj}[v] \). Each entry of \( \text{Adj}[v] \) is a vertex \( w \) such that \((v, w) \in E\).

- If there are other information associated with the edges (such as edge weight), they can be stored in the entries of the adjacency list.

- For undirected graphs, each edge \( e = (u, v) \) has **two entries in this representation**, one in \( \text{Adj}[u] \) and one in \( \text{Adj}[v] \).

- The Adjacency List Representation for **directed graphs** is similar. For each edge \( e = u \rightarrow v \), there is an entry in \( \text{Adj}[u] \).

- For directed graphs, each edge has only one entry in the representation.
Review

\[ \text{BFS}(G, s) \]

1. \( Q \leftarrow \emptyset \)
2. for each \( u \in V - \{s\} \) do
3. \( \pi[u] = \text{NIL}; \ d[u] = \infty; \color[u] = \text{white} \)
4. \( d[s] = 0; \color[s] = \text{grey}; \pi[s] = \text{NIL} \)
5. Enqueue\((Q, s)\)
6. while \( Q \neq \emptyset \) do
7. \( u \leftarrow \text{Dequeue}(Q) \)
8. for each \( v \in \text{Adj}[u] \) do
9. \( \text{if } \color[v] = \text{white} \)
10. \( \text{then } \color[v] = \text{grey}; \ d[v] \leftarrow d[u] + 1; \pi[v] \leftarrow u; \text{Enqueue}(Q, v) \)
11. \( \color[u] = \text{black} \)
Review

• DFS: Recursive algorithm

**DFS**\((G)\)

1. for each vertex \( u \in V \) do
2. \hspace{1em} color[\( u \)] \leftarrow \text{white}; \quad \pi[\( u \)] = \text{NIL}
3. time \leftarrow 0
4. for each vertex \( u \in V \) do
5. \hspace{1em} if color[\( u \)] = \text{white} then DFS-Visit(\( u \))

**DFS-Visit**\((u)\)

1. color[\( u \)] \leftarrow \text{grey}; \quad time \leftarrow time + 1; \quad d[\( u \)] \leftarrow time
2. for each vertex \( v \in \text{Adj}[u] \) do
3. \hspace{1em} if color[\( v \)] = \text{white}
4. \hspace{2em} then \pi[\( v \)] \leftarrow u; \quad \text{DFS-Visit}(\( v \))
5. color[\( u \)] \leftarrow \text{black}
6. \hspace{1em} f[\( u \)] \leftarrow \text{time} \leftarrow \text{time} + 1
Depth First Search

• Q2:
Prove or give a counter example:

(a) In the textbook, we have two routines for graph traversal - DFS\( (G) \) and BFS\( (G, s) \) - where \( G \) is a graph and \( s \) is any node in \( G \). These two procedures will create a DFS tree and a BFS tree respectively. If \( G = (V, E) \) is a connected, undirected graph then the height of DFS\( (G) \) tree is always larger than or equal to the height of any of the BFS trees created by BFS\( (G, s) \).
Depth First Search

- A2:

DFS ($s = 1$):

BFS ($s = 2$):
Depth First Search

• Q3:

Prove or give a counter example:

(b) BFS and DFS algorithms on a undirected, connected graph $G = (V, E)$, produce the same tree if and only if $G$ is a tree.
Depth First Search

• A3:

(b) BFS and DFS algorithms on a undirected, connected graph $G = (V, E)$, produce the same tree if and only if $G$ is a tree.

Answer: True

Prove: DFS tree = BFS tree = $T$ (of $G$) $\Rightarrow$ $G$ is a tree
Depth First Search

• A3:

(b) BFS and DFS algorithms on a undirected, connected graph $G = (V, E)$, produce the same tree if and only if $G$ is a tree.

Prove: $DFS\ tree = BFS\ tree = T\ (of\ G) \Rightarrow G\ is\ a\ tree$

$G = T \Rightarrow G\ is\ a\ tree$

$G \neq T \Rightarrow e(u, v) \in G\ and\ e(u, v) \notin T$
Depth First Search

• A3:

\[ T \neq G \Rightarrow e(u, v) \in G \quad \text{and} \quad e(u, v) \notin T \]

**Theorem**

Let \( G \) be an undirected graph, and \( T \) the DFS tree of \( G \) constructed by DFS algorithm. Then there are no cross edges. 

\[ e(u, v) \text{ is a back edge or a forward edge} \Rightarrow \text{one of } u \text{ or } v, \text{ is ancestor of the other} \]
Depth First Search

• A3:

\[ T \neq G \Rightarrow e(u, v) \in G \text{ and } e(u, v) \notin T \]

**Theorem**

Let \( G = (V, E) \) be a graph. Let \( d[u] \) be the value computed by BFS algorithm. Then for any \((u, v) \in E\), \(|d[u] - d[v]| \leq 1\).

\[ u \text{ and } v \text{ differ in one level} \]
Depth First Search

• A3:

One of $u$ or $v$ ($e(u, v)$), is ancestor of the other

$u$ and $v$ differ in one level

$e(u, v) \in T$

Contradict with $e(u, v) \notin T$

$G = T$, hence $G$ is a tree
Depth First Search

• A3:

Prove: $G$ is a tree $\Rightarrow$ DFS tree = BFS tree
Depth First Search

• A3:

\[ G \text{ is a tree } \Leftrightarrow \text{DFS tree } = \text{BFS tree} \]
Dijkstra’s Algorithm

Main Data structures:

- $G$: Adjacency List Representation.
- For Each vertex $u \in V$:
  - $\text{Adj}[u]$: the adjacency list for $u$
  - $d[u]$: An upper bound for $\delta(s, u)$
  - $\pi[u]$: indicates the first vertex after $u$ in the shortest $s \rightarrow u$ path
- $S$: A set that holds the finished vertices.
- $Q$: A priority queue that holds the vertices not in $S$. 
Priority Queue

A **Priority Queue** is a data structure \( Q \). It consists of a set of **items**. Each item has a **key**. The data structure supports the following operations.

- \( \text{Insert}(Q, x) \): insert an item \( x \) into \( Q \).
- \( \text{Extract-Min}(Q) \): remove and return the item with minimum key value.
- \( \text{Min}(Q) \): return the item with minimum key value.
- \( \text{Decrease-Key}(Q, x, k) \): decrease the key value of an item \( x \) to \( k \).

By using a **Heap** data structure, priority queue can be implemented so that:

- \( \text{Min}(Q) \) takes \( O(1) \) time.
- All other three operations take \( O(\log n) \) time (\( n \) is the number of items in \( Q \).)
Dijkstra\((G, s, w(\ast))\)

1. Initialize\((G, s)\)
2. \(S \leftarrow \emptyset\)
3. \(Q \leftarrow V\)
4. while \(Q \neq \emptyset\) do
5. \(u \leftarrow \text{Extract-Min}(Q)\)
6. \(S \leftarrow S \cup \{u\}\)
7. for each \(v \in Adj[u]\) do
8. \(\text{Relax}(u, v, w(\ast))\)
9. end for
10. end while

Initialize\((G, s)\)

1. for each \(u \in V\) do
2. \(d[u] = \infty; \pi[u] = \text{NIL};\)
3. \(d[s] = 0\)

Relax\((u, v, w(\ast))\)

1. if \(d[v] > d[u] + w(u \rightarrow v)\) do
2. \(d[v] = d[u] + w(u \rightarrow v)\)
3. \(\pi[v] = u\)
• Bellman-Ford Algorithm

\[
\begin{align*}
&\text{Initialize}(G, s) \\
&\text{for } i = 1 \text{ to } n \text{ do} \\
&\quad \text{for each } e = (u, v) \in E \text{ do} \\
&\quad\quad \text{Relax}(u, v, w(\ast)) \\
&\text{for each } e = (u, v) \in E \text{ do} \\
&\quad \quad \text{if } d[v] > d[u] + w(u \rightarrow v) \text{ do} \\
&\quad\quad\quad \text{output } \text{"G has a negative cycle"} \\
&d[u] \text{ is the length of the shortest } s \rightarrow u \text{ path for each } u \in V
\end{align*}
\]

\[
\begin{align*}
&\text{Initialize}(G, s) \\
&\quad \text{for each } u \in V \text{ do} \\
&\quad\quad d[u] = \infty; \pi[u] = \text{NIL}; \\
&\quad\quad d[s] = 0 \\
&\text{Relax}(u, v, w(\ast)) \\
&\quad \text{if } d[v] > d[u] + w(u \rightarrow v) \text{ do} \\
&\quad\quad d[v] = d[u] + w(u \rightarrow v) \\
&\quad\quad \pi[v] = u
\end{align*}
\]
Q1:
• Q1:
• Q1:
• Q1:
• Q1:
Q1:
• Q1:
Q1:
Thank you!