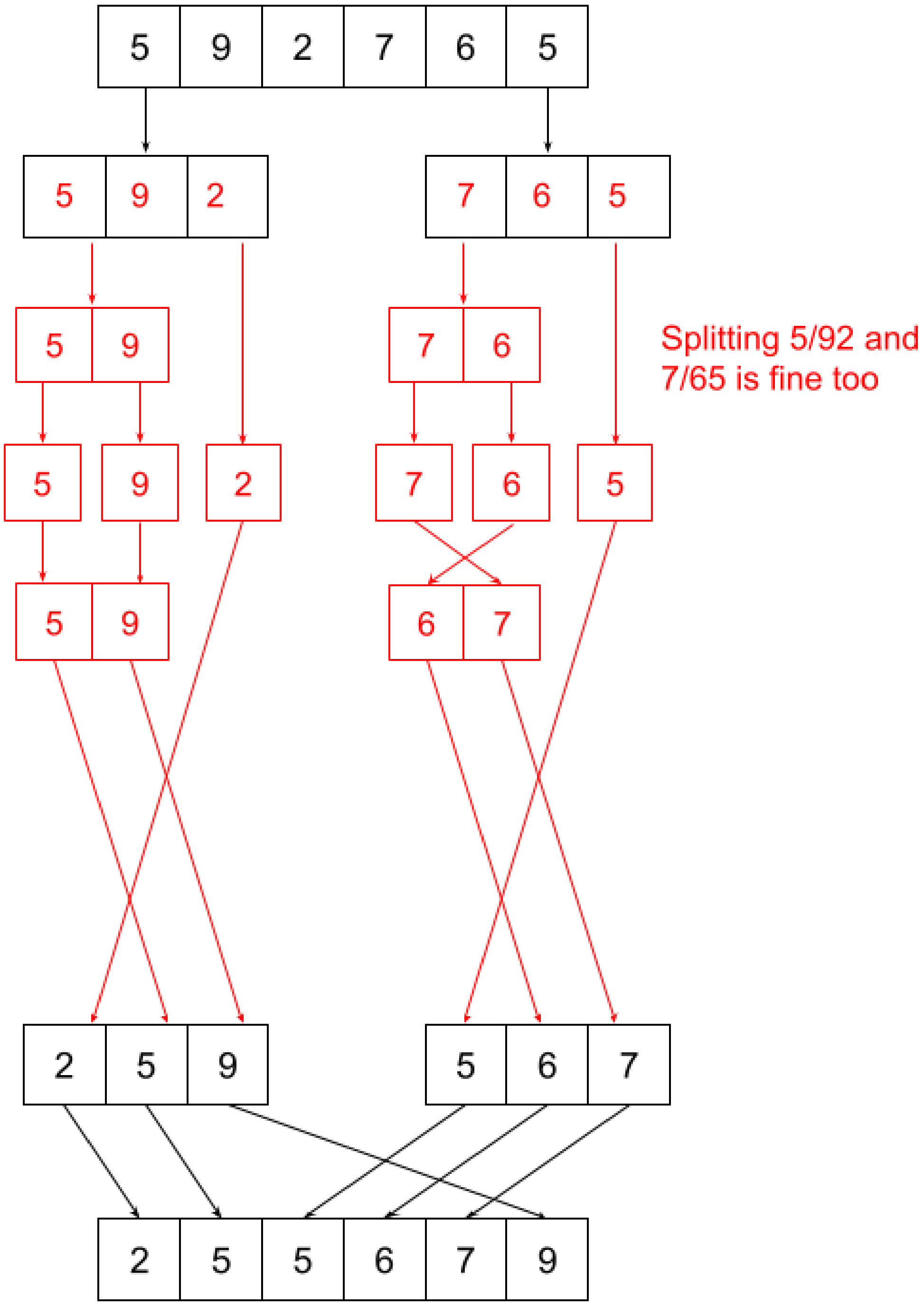
Basic question 1: Merge sort

Demonstrate Merge sort by completing this illustration (show every split/merge step).



Basic question 2: Hash tables

You have the following linear probing hash table using **modular hashing**:

0	1	2	3	4
A	C		B	D

Here are the hash values for each of the elements (before compression with %):

hash(A) = 9825498235
hash(B) = 2324213
hash(C) = 12214256
hash(D) = 3

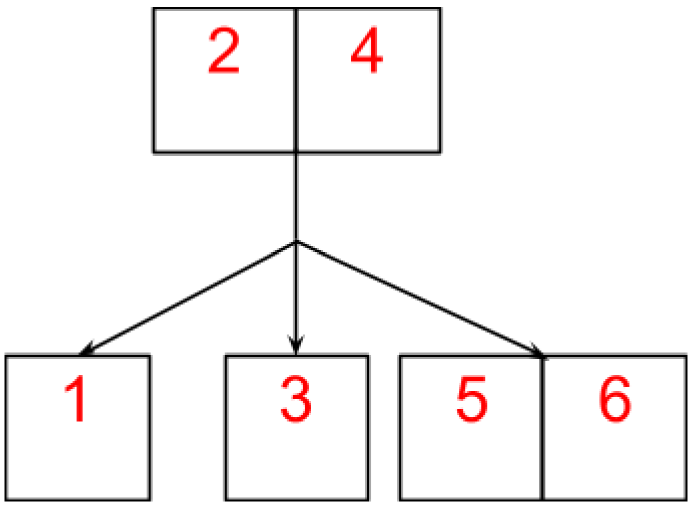
a) Double the size of the hash table.
How does the table look after resizing?

0	1	2	3	4	5	6	7	8	9
			B	D	A	C			

b) If searching the resized table for E such that hash(E) = 1234, which values are E compared to? (Answer as a list of nodes, e.g. "C, A, B" if your answer is that E is compared to C, A and B in that order)

D, A, C

Basic question 3: 2–3 trees



Consider this 2-3 tree structure.

The tree contains the numbers 1,2,3,4,5,6.

a) Write these numbers in the correct boxes in the tree structure.

b) In which of these orders can the numbers have been added if no deletions were done? (checkmark all correct orders)

- ☐ 1, 2, 3, 4, 5, 6
- ☐ 6, 5, 4, 3, 2, 1
- ☐ 2, 1, 4, 3, 6, 5
- ☐ 1, 2, 3, 6, 5, 4
- ☐ 6, 5, 4, 1, 2, 3

Basic question 4: Sets

A set is a collection of elements without duplicates. As an ADT, it is characterised primarily by two operations: Adding values (add) and checking if a value is a member (contains).

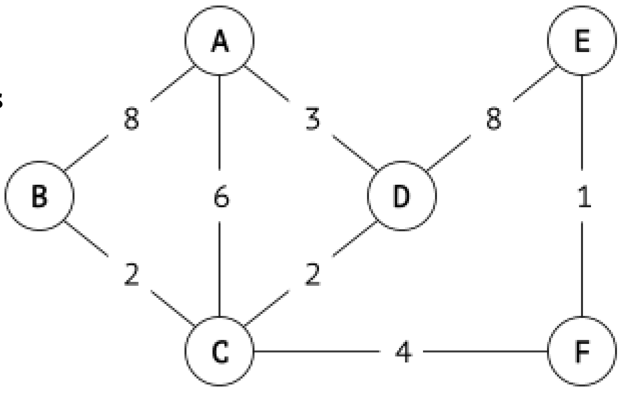
Here are three examples of data structures that can be used to implement sets. For each one, determine the worst case asymptotic complexity of the operations (assuming we use the fastest algorithms we have learned in the course).

	`add(value)`	`contains(value)`
(a) unsorted linked list	O(n)	O(n)
(b) sorted dynamic array	O(n)	O(log n)
(c) AVL trees	O(log n)	O(log n)

Hint: Don't forget the case of adding the same element multiple times.

Optional explanation (e.g. if you make any assumptions about the worst case, or what algorithms you use for (a) and (b)):

Basic question 5: UCS/Dijkstra's algorithm



Using the graph to the right, perform three iterations of UCS search starting in A (until you have discovered the shortest path to three nodes including A itself)

What nodes and total costs have you found?

A	:	0
D	:	3
C	:	5

What is the contents of the priority queue at this point (edges and total costs)?
This should be after visiting all three nodes listed above, just before you select a fourth node to include in the shortest path tree.
List items in ascending priority (total cost) order.

Using a lazy version:

From-node	**To-node**	**Total cost**
A	C	6
C	B	7
A	B	8
C	F	9
D	E	11

Using an eager version:

From-node	**To-node**	**Total cost**
C	B	7
C	F	9
D	E	11

Either of these are acceptable, other combinations are not.

Basic question 6: Complexity

What is the worst case time complexity of these two functions in the size **n** of the array xs, in O-notation?

```
hasPairSum(xs, k):
  for x1 in xs:
    for x2 in xs:
      if x1+x2 == k and x1!=0 and x2!=0:
        return true
  return false
```

O(n) iterations

O(n) iterations

$O(n^2)*O(1)$ total = $O(n^2)$

```
countPairSums(xs):
  count = 0
  for x in xs:
    if hasPairSum(xs, x):
      count = count+1
  return count
```

O(n) iterations

$O(n)*O(n^2)$ total = $O(n^3)$

hasPairSum: O(n^2)

countPairSums: O(n^3)

Short explanation (you should also write notes in the margins of the code):

Advanced question 7: Finding k smallest

Describe an algorithm for finding the k smallest elements (in ascending order) of an array xs with N elements in **worst case O(N log k) time**. You have access to a binary heap implementation that can be used for either a max- or min-priority queue, that you can use without implementing it.

Example: For an array like [3,5,1,4, 2] and k = 3, the algorithm should produce the array [1,2,3].

Hint: After 3,5,1, and 4 you have one too many elements - which one should you throw away?

(a) Justification for the O(N log k) worst case time complexity:

See notes in margins. Basically it does O(n) operations on a PQ of size O(k).

(b) Algorithm (pseudocode):

```
xs = your input array
result = new array of size k  (fill with k smallest elements of xs in ascending order)

pq = new max-priority queue
for x in xs:              //  O(N) iterations
  //  optionally, check if x<pq.getMax() here to improve best-case
  pq.add(x)               //  O(N)*O(log k) total, since pq has O(k) elements
  if(pq.size() > k):
    pq.removeMax()        //  O(N)*O(log k) since pq has k+1 elements

for i=k-1, k-2, ... 0:            //  O(k) iterations
  result[k] = pq.removeMax()      //  O(k) * O(log k) total time
return result
```

Roughly speaking, points are given for:

- Identifying the basic algorithm required, including using a max-PQ rather than a min-PQ (since we are removing the maximum element when the PQ has too many elements).
- Using other algorithms assuming they are fast enough and don't use other data structures without implementing them.
- Correctly explaining that k is an upper bound on the size of the PQ, hence O(log k) cost of operations.
- Using an O(n log n) algorithm and identifying it as such gives one point.
- Using the QuickSelect algorithm gives one point, since it's only average case O(N log K)

Advanced question 8: Building bridges

An archipelago of N islands wants to build bridges that connect all the islands. Your task is to sketch a program for finding the cheapest way to do this. A company offers these price estimates:

- Bridges less than 100 metres cost 1.5 million.
- Bridges between 100 and less than 500 metres cost 1 million (cheaper than the 100m ones!)
- Bridges between 500 and less than 1000 metres cost 2 million.
- Bridges 1000 metres or longer cannot be built.

Notes: Every bridge is a straight line between two islands. You don't need to consider bridges crossing other bridges or islands. Islands are tiny enough to be considered points (no area).

Task 1: Your program needs to find if it is possible to connect all islands with bridges, and return a set of bridges with the lowest possible cost to connect all islands (if possible). You can use any algorithms and data structures taught in the course without implementing them. You need to *explain how you encode the problem* using pseudocode or *detailed* text descriptions.

Task 2: What is the complexity of your solution for N islands? (include a short motivation)

Example: The islands are given with 2D-coordinates (in metres), something like:
islands = [Island1: (0,0), Island2: (0,90), Island3: (90, 0), Island4: (90, 90)]
In this case one cheapest set of bridges for Islands 1,2,3, and 4 is: (1-4), (2-3), (1-2) for 3.5 million.

Solution:

Represent the islands as a graph where edge weights are costs of bridges
Use Kruskal's or Prim's to find a MST (or discover that there is no MST)

Complexity: There are worst case $E=N^2$ possible bridges (edges), and MST calculations are $O(E \log E)$, so $O(N^2 \log N^2) = O(N^2 \log N)$

```
graph g = new empty undirected graph
for island1 in islands:  // this loop takes O(N²) time total
  g.addNode(island1)
  for island2 in g.nodes():
    d = distance(island1.pos, island2.pos) // Euclidean distance
    if d < 1000
      if d < 100: weight = 1.5
      elif d < 500: weight = 1
      else: weight = 2
      g.addEdge(island1, island2, weight)
mstEdges = kruskals(g) // Tries to find a minimal spanning tree, O(N² log N)
if len(mstEdges) < N-1: // Did we fail to find a MST?
  return None // no bridge network is possible
else
  return mstEdges
```

Points for identifying encoding the problem as an MST problem, and for determining the correct complexity.

Advanced question 9: Median finding set

Your task is to make a **set** data structure with **add**, **remove** and **contains** operations. It should also be extended with a **constant time median function**.

For full points, add, remove and contains must be **O(log n)**. For partial points, they can be any complexity. The median function must be **O(1)**.

Note: For even sized sets, the median function should give the lesser of the middle values. So the median of {1,2,3,4,5,6} is 3, the median of {5,7,9,11,13} is 9, etc.

You can use standard data structures without implementing them (AVL, hash-tables, sorted lists...)

You do not have to deal with empty sets. Assume the set is non-empty before and after operations.

1 point solution: Use a sorted list and insert/delete from it. Median is just indexing to the middle.

Slightly cheaty (in a good way) fast solution: Use an AVL tree and compute the median in add. Needs an explanation of how to index in AVL trees in O(log n) time.

Fast solution using AVL trees:

```
class MedianSet:
  median = None
  small = new AVL tree set // values smaller than median (at most n/2)
  large = new AVL tree set // values larger than median (at most n/2+1)

  median(): return median // O(1)

  rebalance(): // fixes balance assuming it's off by at most one. O(log n).
    if small.size() > large.size():
      large.add(median)          // O(log n) for AVL trees
      median = small.removeMax() // O(log n) for AVL trees ("go left")
    else if small.size < large.size()+1
      small.add(median)
      median = large.removeMin() // O(log n) for AVL trees ("go right")

  add(value):
    if(value < median)
      small.add(median)
    else if (value > median)
      large.add(median)
    rebalance()

  remove(): … // almost identical to add, just use .remove instead
  contains(value):
   // this could be made a bit faster by splitting into cases
   return value==median or small.contains(value) or large.contains(value)
```