Basic question 1: Quicksort partition

Assume the following 9-element array:

0	1	2	3	4	5	6	7	8
19	25	23	31	20	17	34	12	22

Perform **one single Quicksort partition**, using the middle value (*not* the median) as the pivot. You must use a common partition algorithm, such as the one from the course book.

a) What swaps did you perform? (Don't forget the pivot swaps.)

Show the swaps in order, and write them as `swap(i,j)` where `i` and `j` refer to the array cells to be swapped. For example, to swap the elements 31 and 34 in the original array, write `swap(3,6)`.

swap pivot first: `swap(4,0)`

partition the array: `swap(1,7), swap(2,5), swap(3,4)`

swap pivot back in place: `swap(0,3)`

b) How does the array look after partitioning? (But before sorting the parts.)

0	1	2	3	4	5	6	7	8
19	12	17	20	31	23	34	25	22

Basic question 2: Hash tables

Insert the following elements into an empty linear probing hash table with modular compression, in the order specified:

241, 607, 313, 143, 890

(a) Assume that the size of the table is 10, and the hash function is:
$h_1(x)$ = the sum of the digits of x

0	1	2	3	4	5	6	7	8	9
890			607				241	313	143

(b) Assume that the size of the hash table is 8, and the hash function is:
$h_2(x)$ = the product of the digits of x

0	1	2	3	4	5	6	7
241	607	313	890	143			

Note: here are some examples showing how h_1 and h_2 work:

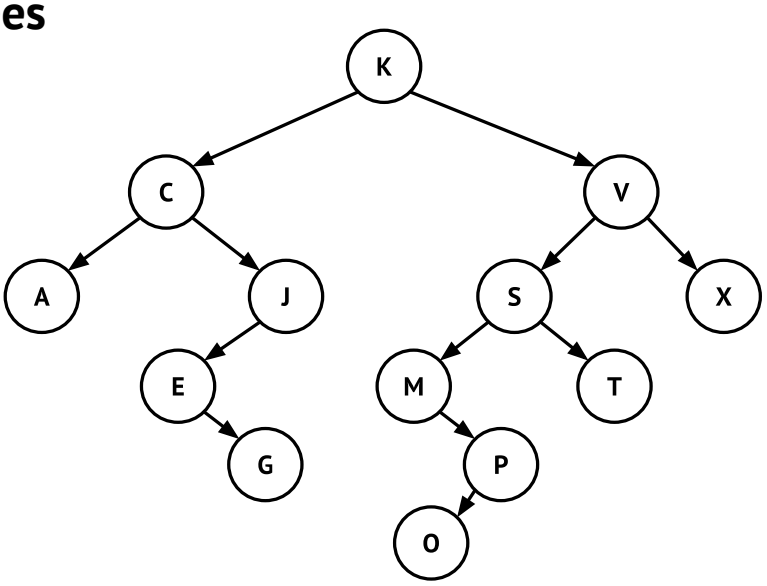
- $h_1(356) = 3+5+6 = 14$, $h_1(719) = 7+1+9 = 17$
- $h_2(356) = 3\times5\times6 = 90$, $h_2(719) = 7\times1\times9 = 63$

$h_1(241) = 7$, $h_1(607) = 13$, $h_1(313) = 7$, $h_1(143) = 8$, $h_1(890) = 17$

$h_2(241) = 8$, $h_2(607) = 0$, $h_2(313) = 9$, $h_2(143) = 12$, $h_2(890) = 0$

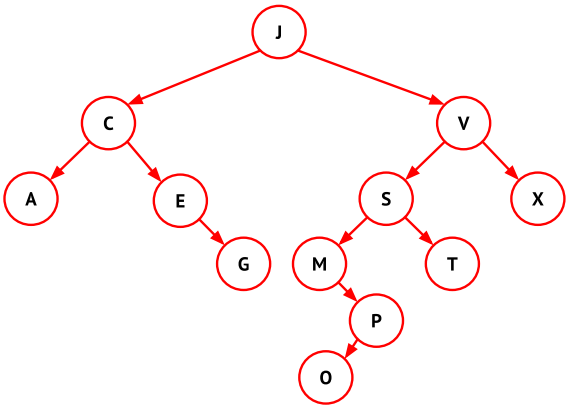
Basic question 3: Search trees

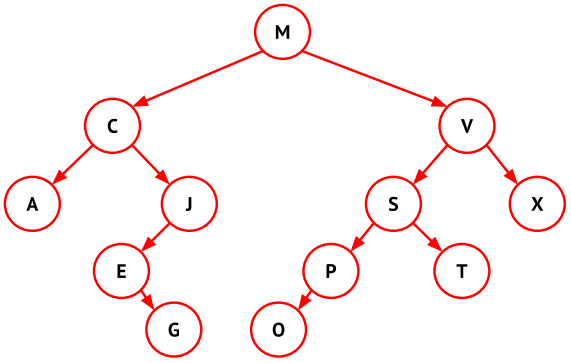
Delete the root value from the following binary search tree, using the standard deletion procedure. You can use any of the two possible variants.



a) Draw the resulting tree:

Here are both possible solutions:





b) Briefly explain how you did this:

1. Find the largest (smallest) node in the left (right) subtree, this is J (M).
2. Replace the root value with J (M).
3. Delete the old J (M) node using the normal procedure: since it has one child, we can replace the node by its child E (P).

Basic question 4: Stacks and queues

Here is some trivial code that partitions an array according to a pivot value:

```
function stupid_partition(array, pivot):
    stack = new empty Stack
    queue = new empty Queue

    for i in 0 .. len(array)-1:
        if array[i] < pivot:
            stack.push(array[i])
        else:
            queue.enqueue(array[i])

    for i in 0 .. len(array)-1:
        if stack is not empty:
            array[i] = stack.pop()
        else:
            array[i] = queue.dequeue()
```

Assume we call the function with the array `[4,9,3,5,1,2,7,8]` and the pivot value `6`.

a) How will the array look after the function is finished?

`array =` `[2, 1, 5, 3, 4, 9, 7, 8]`

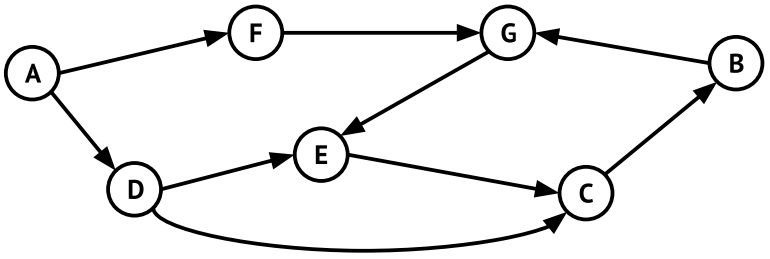
b) How will the stack and the queue look *in between* the two loops? Show each as a list of elements, and indicate the top/bottom of the stack and the front/back of the queue.

`stack =` `2 (top), 1, 5, 3, 4 (bottom)`

`queue =` `9 (front), 7, 8 (back)`

Basic question 5: Graphs

Perform depth-first and breadth-first search on the following directed graph.



- Start at vertex A and show the order in which the vertices are visited.
- Only write the first time a vertex is visited.
- You choose the order of outgoing edges for each node. So there are many possible visitation orders. Show two alternative ones for each algorithm.

(a) Depth-first search (DFS):

A						

(b) Another possible DFS traversal order:

A						

(c) Breadth-first search (BFS):

A						

(d) Another possible BFS traversal order:

A						

All possible DFS traversals are: ADCBGEF, ADECBGF, AFGECBD

All possible BFS traversals are: ADFECGB, ADFCEGB, AFDGECB, AFDGCEB

Basic question 6: Complexity

Here is a two-dimensional sorting algorithm. It takes an $n \times n$ square matrix of numbers as input, and rearranges the numbers so that the following holds:

```
m[i][j] ⩽ m[i'][j']   whenever   i+j < i'+j'
```

The intermediate storage pq is a minimum priority queue, implemented as a binary heap.

```
function sort_matrix(m):
    n = the length of each side of m
    pq = new empty binary heap

    for i in 0 .. n-1:                          O(n) ×
        for j in 0 .. n-1:                        O(n) ×
            add m[i][j] to pq                       O(log(n²)) = O(log(n))

    i = 0; j = 0
    while pq is not empty:                      O(n²) ×
        m[i][j] = remove minimum from pq          O(log(n²)) = O(log(n))
        if i = n-1 or j = 0:                      O(1)
            swap i and j                          O(1)
        else:                                     O(1)
            j = j - 1                             O(1)
        i = i + 1                                 O(1)
```

What is the asymptotic time complexity of the algorithm in terms of n, the length of each side of m?

- Assume that all arithmetic operations and array indexing take constant time.
- Answer in O-notation, and as tight and simple as possible.
- State how you concluded this (you may do this by annotating the program).

Both loops have complexity $O(n^2 \log(n^2)) = O(n^2 \log(n))$, see the annotations above.

Note: There was a bug in the algorithm, so it didn't correctly sort the matrix (it skips writing to the cells (0, j) for j > 0). But this doesn't change the complexity analysis, only the correctness. Here is a corrected version of the 2nd loop:

```
while pq is not empty:
    m[i][j] = remove minimum from pq
    if i = n - 1:   i = j + 1; j = n - 1
    else if j = 0:  j = i + 1; i = 0
    else:           i = i + 1; j = j - 1
```

Advanced question 7: Mergeable heap

A (max) heap is a binary tree that satisfies the heap property:

- The priority of every parent is at least as large as those of its children.

If we further demand that the tree is complete, we have a binary heap (which as you know can be implemented efficiently as an array). Binary heaps are efficient for many purposes, but there is one quite common operation that is difficult to implement for them: *merging*.

To be able to efficiently implement merging, we drop the completeness invariant and work with heaps as binary trees satisfying only the heap property. We add a height attribute to binary nodes so that we can calculate the height of a heap in constant time.

```
class Heap:
    left, right : Heap
    priority : Integer
    height   : Integer   // 1 + the maximum height of left or right
```

The empty heap is represented by **null**, so here is a helper function for the height of any heap:

```
height(heap) = 0 if heap is null, else heap.height
```

a) Implement an algorithm for merging two heaps:

```
function merge(heap1, heap2 : Heap) -> Heap
```

It should return the merged heap, built from the nodes of the given heaps.

b) Your function must run in time O(log(*n*)) in terms of the total number *n* of elements of the heaps. Justify why this is the case.

***Hint*:** In a binary tree of size *n*, the shortest branch has length O(log(*n*)).
You can use this fact without proof.

```
function merge(heap1, heap2 : HeapNode) -> HeapNode
    if heap1 is null: return heap2
    if heap2 is null: return heap1
    if heap1.priority < heap2.priority:
        swap heap1 and heap2
    if height(heap1.left) < height(heap1.right):
        heap1.left = merge(heap1.left, heap2)
    else:
        heap1.right = merge(heap1.right, heap2)
    heap1.height = 1 + max(height(heap1.left), height(heap1.right))
    return heap1
```

Everything in this function except for the recursive call to merge is constant time. So the run time is linear in the depth of the recursion. In each recursive call, we descend one step along a shortest branch in one of the two heaps. So the recursion depth is bounded by the sum of the shortest branch lengths of `heap1` and `heap2`. Using the hint, both of these are O(log(*n*)).

Advanced question 8: Rolling hashes

A hash function for strings supports *rolling hashing* if we have a function

```
shift(h: Hash, k: Integer, old: Char, new: Char) -> Hash
```

that computes how the hash value *h* of a string of length *k* changes when we remove ("roll out") the first character *old* and add ("roll in") the last character *new*. For example:

```
shift(hash("word"), 4, 'w', 's') = hash("ords")
```

This allows us to "slide a window" over a string while calculating hashes. For example, for the string "datastructures", we could start with the hash for "data" and then use `shift` to calculate the hashes for "atas", then for "tast", and so on.

a) Here is a standard hash function for strings (where k = length(s)):

$$\texttt{hash(s)} = \texttt{s[0]}\cdot 37^{k-1} + \texttt{s[1]}\cdot 37^{k-2} + \ldots + \texttt{s[k-1]}\cdot 37^{0}$$

This runs in time linear in k, so is expensive to call repeatedly – therefore we want it to support rolling hashing.

Implement a constant-time `shift` function for this hash function.

***Note*:** Assume that all arithmetic operations, including powers of 37, take constant time. (In practice, hashes are computed modulo 2^{32} and powers of 37 are precomputed.)

b) Use `hash` and `shift` to implement a function

```
search(str: String, k: Integer, hash_key: Hash) -> Integer
```

that finds the starting position of the first substring of *str* with length *k* and hash *hash_key*, or **null** if no such substring exists.

Your function must run in time linear in the length of *str*. You can assume $k \leqslant$ length(*str*).

Example: `search`("datastructures", `length`("tast"), `hash`("tast")) should return 2, assuming no hash collisions.

a) Many possible solutions:

shift(h, k, old, new):
    rolled_out = h − old · 37^{k-1}
    rolled_in = rolled_out · 37 + new
    return rolled_in

shift(h, k, old, new) =
    (h − old · 37^{k-1}) · 37 + new

shift(h, k, old, new) =
    h · 37 − old · 37^{k} + new

b) Also several solutions:

```
search(str, k, hash_key):
    h = hash(str[0:k])
    for start from 0 to length(str)−k:
        if h == hash_key:
            return start
        end = start + k
        if end < length(str):
            h = shift(h, k, str[start],
                      str[end])
    return null
```

An alternative that starts the sliding window before the start of the string:

```
search(str, k, hash_key):
    h = 0  // hash of fake prefix
    for end from 0 to length(str):
        start = end − k
        chr = str[start] if start⩾0 else 0
        h = shift(h, k, chr, str[end])
        if start ⩾ 0 and h == hash_key:
            return start
    return null
```

Advanced question 9: Range queries

Recall that a binary search tree is either **null** or a (binary) **node** with *left* and *right* subtrees.
For this question, we assume that nodes have an additional *size* attribute:

```
class Node:
    left, right : Node
    value : Integer
    size  : Integer     // the number of values in this tree
```

a) Write an efficient function that takes a binary search tree and a range, and counts the number of values in the tree that are within the given range.

```
function countRange(tree : Node; low, high : Integer) -> Integer
```

In other words, `countRange` should return the number of values x such that $low \leqslant x < high$.

Note: You can assume that there are no duplicate values in the tree.

Example: If the tree `t` contains the values [2, 5, 8, 11, 22, 28] then `countRange(t, 5, 22)` should return 3, because there are three values (5, 8, and 11) that are ⩾5 and <22.

b) What is the time complexity of your function, in terms of n, the size of the tree? Explain your reasoning.

Note: Assume that the tree is balanced.

Here is a solution based on the fact that we can count the number of values less than a given limit in O(log(n)) time. Then we can get the number in the interval by subtracting the number of values below lower from the number of values below upper.

```
function countRange(node, low, high):
    return countLesser(node, low) - countLesser(node, high)


function countLesser(node, limit):
    if node is null:
        return 0
    else if node.value < limit:
        return 1 + size(node.left) + countLesser(node.right, limit)
    else:
        return countLesser(node.left, limit)


function size(node):
    if node is null:
        return 0
    else:
        return node.size
```