Algorithms example questions

Sorting

Exercise 1

Assume that each `swap(x, y)` means three assignments (namely `tmp = x; x = y; y = tmp`). Improve the `insertSort` algorithm pseudocode shown in the handout to reduce the number of assignments performed in the inner loop.

```
def insertSort(a):
    for i in range(1, len(a)):
        j = i - 1
        tmp = a[j+1]
        while j >= 0 and a[j] > tmp:
            a[j+1] = a[j]
            j = j - 1
        a[j+1] = tmp
```

We now only do 1 assignment in the inner loop instead of 3.

Exercise 2

Provide a useful invariant for the inner loop of insertion sort, in the form of an assertion to be inserted between the `while` line and the `swap` line.

```
for k in range(i-1):
    assert(a[k] <= a[k+1])
```

Exercise 3

$$|sin(n)| = O(1)$$
$$|sin(n)| \neq \Theta(1)$$
$$200 + sin(n) = \Theta(1)$$
$$123456n + 654321 = \Theta(n)$$
$$2n - 7 = O(17n^2)$$
$$lg(n) = O(n)$$
$$lg(n) \neq \Theta(n)$$
$$n^{100} = O(2^n)$$
$$1 + 100/n = \Theta(1)$$

For each of the above “$=$” lines, identify the constants k, k_1, k_2, N as appropriate. For each of the “$\neq$” lines, show they can’t possibly exist.

1. $k = 1, N = 0$
2. Since $k_1 > 0$, and $|sin(n)|$ falls to zero infinitely many times, there is no N such that $|sin(n)| > k_1$ for $n > N$.
3. $k_1 = 199, k_2 = 201, N = 0$
4. $k_1 = 123456, k_2 = 777777, N = 1$
5. $k = 1, N = 0$
6. $k = 1, N = 0$
7. $k_1 n = ln(n) \implies k_1 = ln(n)/n$, which has solutions for all $0 < k_1 \leq 1/e$, so no $k_1 > 0$ is small enough to provide a lower bound.
8. $k = 1, N = 1000$
9. $k_1 = 1, k_2 = 101, N = 1$

Exercise 4

What is the asymptotic complexity of the variant of insertSort that does fewer swaps?

The outer loop still runs $n - 1$ times, the inner loop still runs i times on the ith loop, the only difference is some constant factors in each of the loops, so the complexity remains as $O(n^2)$.

Exercise 5

The proof of Assertion 1 (lower bound on exchanges) convinces us that $\Theta(n)$ exchanges are always sufficient. But why isn’t that argument good enough to prove that they are also *necessary*?

If the items are already in order, then no exchanges are needed, so n exchanges are not *necessary*.

Exercise 6

When looking for the minimum of m items, every time one of the $m - 1$ comparisons fails the best-so-far minimum must be updated. Give a permutation of the numbers from 1 to 7 that, if fed to the Selection sort algorithm, maximizes the number of times that the above-mentioned comparison fails.

$(7, 6, 5, 4, 3, 2, 1)$ gives the maximum of 12 failures.

Exercise 7

Code up the details of the binary partitioning portion of the binary insertion sort algorithm.

```python
def binaryInsertionSort(a):
    for k in range(1, len(a)):
        # find the right location for a[k] in the sorted sub-array
        i_0 = 0
        i_1 = k
        i = int(k/2)

        while i_0 < i:
            if a[k] < a[i]:
                i_1 = i
            else:
                i_0 = i
            i = int((i_0+i_1)/2)

        if a[k] >= a[i]:
            i += 1

        # shuffle items so it's in the right place.
        if i != k:
            tmp = a[k]
            for j in range(k-1, i-1, -1):
                a[j+1] = a[j]
            a[i] = tmp
```

Exercise 8

Prove that Bubble sort will never have to perform more than n passes of the outer loop.

With each pass of the outer loop, one more item is placed correctly, starting from the top of the list. This means that after n passes, all of the items will have been placed correctly, and the list is sorted, so not more than n passes are required.

Exercise 9

Can you spot any problems with the suggestion of replacing the somewhat mysterious line `a3[i3] = smallest(a1, i1, a2,i2)` with the more explicit and obvious `a3[i3] = min(a1[i1],a2[i2])` ? What would be your preferred way of solving such problems? If you prefer to leave that line as it is, how would you implement the procedure `smallest` it calls? What are the trade-offs between your chosen method and any alternatives?

If we use `min` , we don’t know which of the arrays the item came from, so we can’t say which of the indexes (`a1` or `a2`) we should increment. Additionally, we don’t check whether each array index is in bounds, so we may get an error.

To get around the bounds problem, you should make index checks. To find out which array the smallest is a part of, you can either put additional if statements to see which array contains the smaller element, and then take from that array, or you can check the element which results from `min` , and compare it against the first element in each of the arrays to see which it came from.

Exercise 10

In one line we return the same array we received from the caller, while in another we return a new array created within the mergesort subroutine. This asymmetry is suspicious. Discuss potential problems.

If `mergeSort(a)` is expected to return a copy of `a[]` which has been sorted, as in the second case, issues may arise when editing the sorted array, since then the original array will be edited as well.

Exercise 11

Never mind the theoretical computer scientists, but how do you mergesort in $n/2$ space?

Store the left and right sub-arrays in one array together, then only copy the left array to temporary space. Then merge the two sub-arrays back into the original array.

Exercise 12

Justify that the merging procedure just described will not overwrite any of the elements in the second half.

The amount of spare space in `a[]` is always the same as the number of elements in the `left` sub-array, since when an item from `left` is selected, the amount of each decreases by one, and when an item from `right` is selected, the amounts stay the same.

Exercise 13

Write pseudocode for the bottom-up mergesort.

```
def bottomUpMergeSort(a):
    # for each partition width
    for (width = 1; width < n; width *= 2):
        # for each pair of partitions
        for (i = 0; i < n; i += 2*width):
            # merge the partitions
            merge(a[i:i+width], a[i+width:i+2*width])
```

Exercise 14

Can picking the pivot at random *really* make any difference to the expected performance? How will it affect the average case? The worst case? Discuss.

In the average case, random selection of the pivot will maintain a performance of $O(n \lg n)$, since the pivot will tend to be roughly the median value of the array. There isn't really a worst case when using random pivot selection, since the program is no longer deterministic, however, the worst running cost remains at $O(n^2)$ for any array.

Exercise 15

Justify why running Insertion sort over the messy array produced by the truncated Quicksort might not be as stupid as it may sound at first. How should the threshold be chosen?

A truncated quicksort will return an array where each element is *close* to where it should be, but maybe not *exactly*. This kind of array can be quickly fixed using insertion sort, because the insertion process is likely to only need to go back a few elements to find the right place.

The threshold in effect is the maximum distance that the insertion process would then have to search to place an element. The cost of inserting an element at that depth can be compared with the cost of dividing and sorting an array of that length.

Exercise 16

What is the smallest number of pairwise comparisons you need to perform to find the smallest of n items?

Only $n-1$ are needed. A binary tree can be constructed by comparison of pairs of items, like a tournament. This will take $n-1$ comparisons.

Exercise 17

(More challenging.) And to find the *second* smallest?

Using the binary tree we constructed earlier, we can traverse back through it, to find the smallest of the items which the smallest was compared to. This will be the second smallest. This process will take $\lceil \lg n \rceil$ steps to complete, for a total of $n + \lceil \lg n \rceil - 1$ comparisons.

Exercise 18

What are the minimum and maximum number of elements in a heap of height h?

$$2^h \leq n < 2^{h+1} \implies h \leq \lg n < h+1 \implies h = \lfloor \lg n \rfloor$$

Exercise 19

For each of the sorting algorithms seen in this course, establish whether it is stable or not.

Algorithm	Stable?
Insertion	Yes
Selection	Yes
Binary insertion	Yes
Bubble	Yes
Merge	Yes
Quicksort	No

Algorithm	Stable?
Heapsort	No

Exercise 20

Give detailed pseudocode for the counting sort algorithm (particularly the second phase), ensuring that the overall cost stays linear. Do you need to perform any kind of precomputation of auxiliary values?

```
def countingSort(a):
    # array of counts recorded for each integer
    counts = [0 for i in range(max(a)+1)]

    for i in range(len(a)):
        counts[a[i]] += 1

    positions = [0 for i in range(max(a)+1)]
    total = 0

    for i in range(max(a)+1):
        positions[i] = total
        total += counts[i]

    output = [0 for i in a]
    for i in range(len(a)):
        output[positions[a[i]]] = a[i]
        positions[a[i]] += 1

    return output
```

Exercise 21

Why couldn't we simply use counting sort in the first place, since the keys are integers in a known range?

We technically could, but when the numbers are so large, the space complexity of counting sort becomes an inconvenience, which is much better dealt with using radix.

Exercise 22

Leaving aside for brevity Fibonacci's original 1202 problem on the sexual activities of a pair of rabbits, the Fibonacci sequence maybe more abstractly defined as follows:

$$\begin{cases} F_0 = 1 \\ F_1 = 1 \\ F_n = F_{n-2} + F_{n-1} \quad n \geq 2 \end{cases}$$

(This yields $1, 1, 2, 3, 5, 8, 13, 21, \ldots$) In a couple of lines in your favourite programming language, write a recursive program to compute F_n given n, using the definition above. And now, finally, the question: how many function calls will your recursive program perform to

compute F_{10}, F_{20} and F_{30}? First, guess; then instrument your program to tell you the actual answer.

```python
def F(n):
    if n <= 1:
        return 1
    else:
        return F(n-2) + F(n-1)
```

Guess: The number of function calls $C(n)$ to compute F_n is equal to F_n itself.

Having computed the actual answer, I found that $C(n) = C(n-2) + C(n-1) + 1$

Exercise 23

Prove (an example is sufficient) that the order in which the matrix multiplications are performed may dramatically affect the total number of scalar multiplications — despite the fact that, since matrix multiplication is associative, the final matrix stays the same.

$$\begin{bmatrix}1\end{bmatrix} \times \begin{bmatrix}1 & 1\end{bmatrix} \times \begin{bmatrix}1 \\ 1\end{bmatrix}$$

Multiplying the right matrices first gives 2, and then 1, for a total of 3 multiplications.

Multiplying the left matrices first gives 2 and then 2, for a total of 4 multiplications.

Exercise 24

There could be multiple distinct longest common subsequences, all of the same length. How is that reflected in the above algorithm? And how could we generate them all?

When backtracing through the table, we occasionally come across a choice between reducing i and reducing j, if they produce subsequences of the same length. In this case, we could produce both options at any stage, which would produce all longest subsequences.

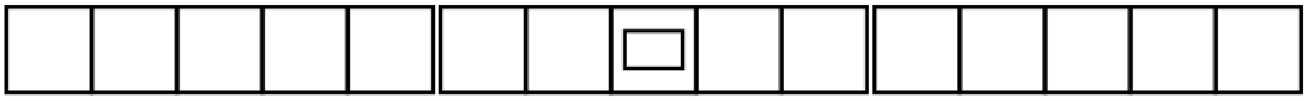
Exercise 25

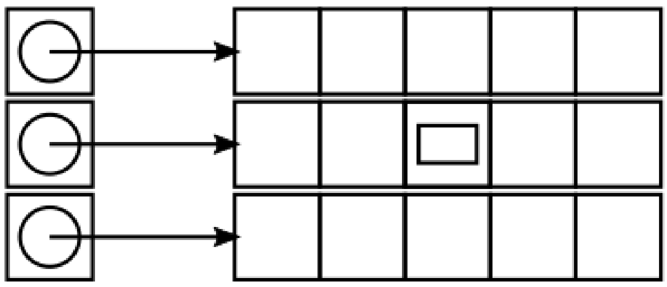
Provide a small counterexample that proves that the greedy strategy of choosing the item with the highest £/kg ratio is not guaranteed to yield the optimal solution.

$W = 2kg, \{(v = 2, w = 1), (v = 3, w = 2)\}$

Exercise 26

Draw the memory layout of these two representations for a 3×5matrix, pointing out where element (1,2) would be in each case.





Exercise 27

Show how to declare a variable of type list in the C case and then in the Java case. Show how to represent the empty list in the Java case. Check that this value (empty list) can be assigned to the variable you declared earlier.

C:

```
ListWagon l = new ListWagon();
```

Java:

```
ListWagon l = new ListWagon();
ListWagon emptyList = null;
```

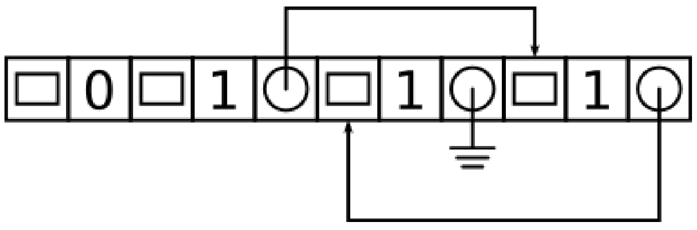
Exercise 28

As a programmer, do you notice any uncomfortable issues with your Java definition of a list? *(Requires some thought and O-O flair.)*

The implementation only allows lists of integers. We would ideally want to be able to create a list of any type, maybe using generics.

Exercise 29

Draw a picture of the compact representation of a list described in the notes.



Exercise 30

Invent (or should I say “rediscover”?) a linear-time algorithm to convert an infix expression such as `( 3 + 12 ) * 4 - 2` into a postfix one without parentheses such as `3 12 + 4 * 2 -`. By the way, would the reverse exercise have been easier or harder?

Assuming the expressions have been nicely arranged into a list of integers, brackets, and operators, e.g. $e = [(, 3, +, 12,), *, 4, -, 2]$, we use the following algorithm:

```
def convert(e):
    result = []

    s = stack()
    for e_i in e:
        if e_i is an integer:
            result.append(e_i)
        elif e_i == '(':
            s.push('(')
        elif e_i == ')':
            while s.top() != '(':
                result.append(s.pop())
            s.pop()
        else:
            while not s.isEmpty() and s.top() != '(':
                result.append(s.pop())
            s.push(e_i)

    while not s.isEmpty():
        result.append(s.pop())

    return result
```

Exercise 31

How would you deal efficiently with the case in which the keys are English words? *(There are several possible schemes of various complexity that would all make acceptable answers provided you justified your solution.)*

Using a binary tree as seen in the FoCS course, where the $k_L < k_P < k_R$ for parent node P and children L and R.

Exercise 32

Should the new key-value pair added by `set()` be added at the start or the end of the list? Or elsewhere?

The new entry should be added in a place that maintains an ordering on the keys in the list, so that when searching for a key, you know it is not present once you have passed its expected location.

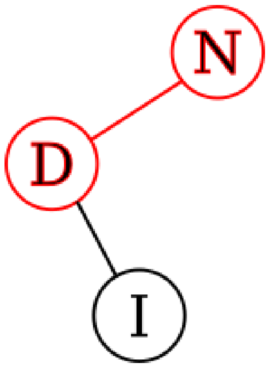
Exercise 33

Solve the $f(n) = f(n/2) + k$ recurrence, again with the trick of setting $n = 2^m$.

$$\begin{aligned} f(n) &= f(n/2) + k \\ &= f(2^{m-1}) + k \\ &= f(2^{m-2}) + 2k \\ &= f(2^{m-m}) + mk \\ &= f(1) + mk \\ &= f(1) + k \lg n \\ &= \Theta(\lg n) \end{aligned}$$

Exercise 34

Professor Bunyan thinks he has discovered a remarkable property of binary search trees. Suppose that the search for key k in a binary search tree ends up in a leaf. Consider three sets: A, the keys to the left of the search path; B, the keys on the search path; and C, the keys to the right of the search path. Professor Bunyan claims that any three keys $a \in A$, $b \in B$, and $c \in C$ must satisfy $a \leq b \leq c$. Give a smallest possible counterexample to the professor's claim.



Search for "A". $``I" \in C \leq ``N" \in B$.

Exercise 35

Why, in BSTs, does this up-and-right business find the successor? Can you sketch a proof?

The successor of N is the node with the smallest $k > k_N$. If N has no right sub-tree, then we check through parents, grandparents, etc. If we take a right turn in our traversal of parents, that means that we have found a node P with $k_N < k_P$. Also, all nodes R in P's right sub-tree have values $k_R > k_P$, so P must be the smallest node such that $k > k_N$.

Exercise 36

Prove that, in a binary search tree, if node n has two children, then its successor has no left child.

Since n has two children, its successor must be in its right sub-tree. All nodes in the right sub-tree have $k > k_n$, and we want to find the smallest of those. Suppose we have found the successor, node s, i.e. k_s is the smallest $k > k_n$. If node s had a left child, then k for the child would be less than k_s, so s would not be the successor. Therefore, s must have no left child.

Exercise 37

Prove that this deletion procedure, when applied to a valid binary search tree, always returns a valid binary search tree.

Deletion of a leaf node will not change the ordering of the rest of the tree, so all the properties of the BST are conserved.

Deletion of a node with one child by replacing it with that child will maintain the properties of the BST, since any nodes in the sub-tree will be suitably bounded by the parent nodes of the node to be deleted.

Deletion of a node with two children by replacement with its successor will work, since the successor is greater than all of the nodes in the left sub-tree, and less than all the nodes in the right sub-tree, again, conserving the BST properties.

Exercise 38

What are the smallest and largest possible number of nodes of a red-black tree of height h, where the height is the length in edges of the longest path from root to leaf?

The largest number is obviously $2^h - 1$, in accordance with any other binary tree.

The minimum number is achieved by first contructing a chain of height h, made from alternating black-red nodes. This chain contains $h/2$ black nodes, so any other path from root to leaf must contain the same amount. This is achieved in the minimum number of nodes by adding a full binary tree of black nodes of the necessary height to each node in the chain. This produces the following sum:

$$\begin{aligned} \min(h) &= 2\sum_{i=0}^{\frac{h}{2}-1}(2^i - 1) + h \\ &= 2\left(2^{\frac{h}{2}} - 1 - \frac{h}{2}\right) + h \\ &= 2^{\frac{h}{2}+1} - 2 \end{aligned}$$

Exercise 39

With reference to the rotation diagram in the handout, and to the stupid way of referring to rotations that we don't like, what would a *left* rotation of the D node be instead? (Hint: it would *not* be the one marked as "Left rotation" in the diagram.)

A left rotation of the D node would be a left rotation of the DE edge.

Exercise 40

During RBT insertion, if p is red and g is black, how could u ever possibly be black? How could p and u ever be of different colours? Would that not be an immediate violation of invariant 5?

u can be black if p has another black child. In general p and u can be different colours as long as the red child has further black children.

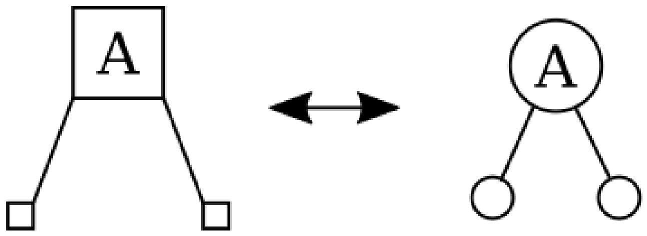
Exercise 41

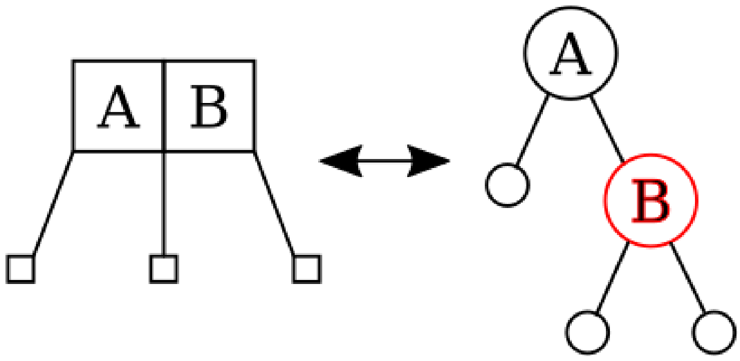
Draw the three cases by yourself and recreate, without reading, the correct procedure to fix each of them. Then apply it to figure 13.4.(a) of CLRS3, without looking at the rest of the figure.

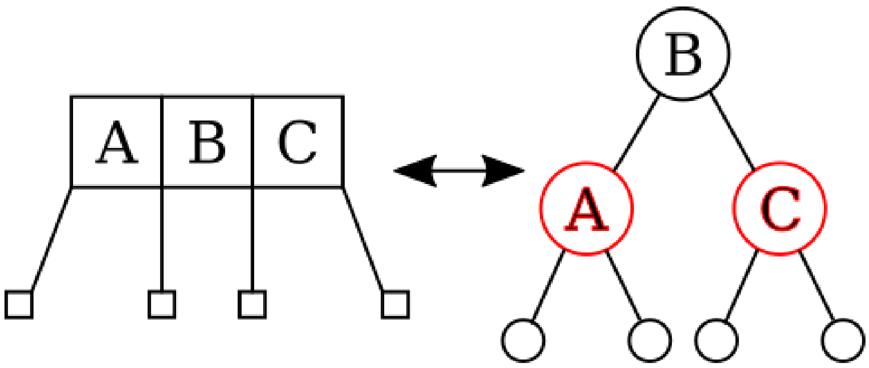
Exercise 42

For each of the three possible types of 2-3-4 nodes, draw an isomorphic "node cluster" made of 1, 2 or 3 red-black nodes. The node clusters you produce must:

- **Have the same number of keys, incoming links and outgoing links as the corresponding 2-3-4 nodes.**
- **Respect all the red-black rules when composed with other node clusters.**







Exercise 43

Using a soft pencil, a large piece of paper and an eraser, draw a B-tree with $t = 2$, initially empty, and insert into it the following values in order: $63, 16, 51, 77, 61, 43, 57, 12, 44, 72, 45, 34, 20, 7, 93, 29$. How many times did you insert into a node that still had room? How many node splits did you perform? What is the depth of the final tree? What is the ratio of free space to total space in the final tree?

10 insertion into nodes with room. 6 splits. Final depth is 2. 11/27 spaces are free.

Exercise 44

Prove that, if a key is not in a bottom node, its successor, if it exists, must be.

The successor is the smallest $k > k_n$. The sub-tree to the right of the current key will contain the successor. Suppose we have found the successor, k_s. k_s must therefore have no left sub-tree, since then the successor would be in that tree instead. Since k_s has no left sub-tree, then it must be in the bottom of the tree, because that is the only case when a key has no sub-tree.

Exercise 45

Make a hash table with 8 slots and insert into it the following values: $15, 23, 12, 20, 19, 8, 7, 17, 10, 11$. Use the hash function $h(k) = (k \mod 10) \mod 8$ and, of course, resolve collisions by chaining.

```
0: -> 20 -> 8 -> 10
1: -> 19 -> 11
2: -> 12
3: -> 23
4:
5: -> 15
6:
7: -> 7 -> 17
```

Exercise 46

Imagine redoing the exercise above but resolving collisions by open addressing. When you go back to the table to retrieve a certain element, if you land on a non-empty location, how can you tell whether you arrived at the location for the desired key or on one occupied by the overspill from another one? (Hint: describe precisely the low level structure of each entry in the table.)

Each entry in the table consists of a key-value pair. The key found at the non-empty location can be compared to the one we were looking up.

Exercise 47

How can you handle deletions from an open addressing table? What are the problems of the obvious naïve approach?

You can mark the item as deleted, without actually removing it from the list, or you can try and rearrange the remaining items in some clever way. If you just remove the item, then searches for items which have been displaced are disrupted.

Exercise 48

Why do we claim that keeping the sorted-array priority queue sorted using bubble sort has linear costs? Wasn't bubble sort quadratic?

Since the rest of the array was sorted already, only one pass of bubble sort is required, to get the out-of-place item to its correct position.

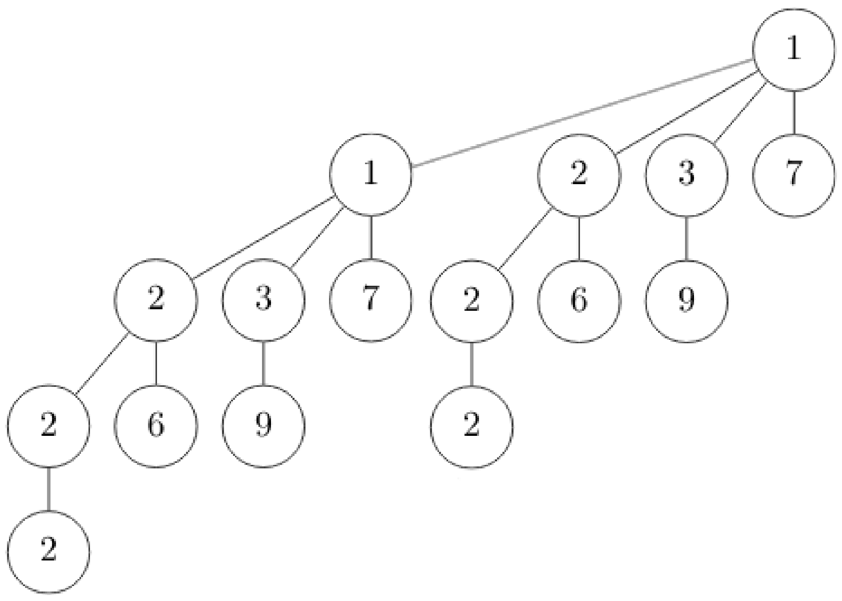
Exercise 49

Before reading ahead: what is the most efficient algorithm you can think of to merge two binary heaps? What is its complexity?

Heapify the combination of both heaps.

Exercise 50

Draw a binomial tree of order 4.



Exercise 51

Give proofs of each of the stated properties of binomial trees (trivial) and heaps (harder until you read the next paragraph - try before doing so).

Assume for tree of order k, it contains 2^k nodes, number of child subtrees of the root is k, and the height is k.

For tree of order $k+1$, it contains $2 \cdot 2^k = 2^{k+1}$ nodes. The number of child nodes of the root is the number for order k, plus one for the new subtree appended to it $= k+1$. The height is the height of a tree of order k, plus one for the root node $= k + 1$.

Since trees have 2^k nodes, and there is at most one of each order in a binomial heap, we can see that the number of trees required to house n nodes only increases when n reaches a power of 2, so the number of trees grows as $O(\lg n)$. For the same reason, the kth tree has degree k, so the degree of the largest tree grows identically to the number of trees; again $O(\lg n)$.

Exercise 52

Prove that the sequence of trees in a binomial heap exactly matches the bits of the binary representation of the number of elements in the heap.

Each tree in the binomial heap holds 2^k nodes, and there can be 1 or 0 of them. For any number of nodes n, there is only one representation of n using 1 or 0 of successive powers of two, namely the binary representation.