Basic question 1: Complexity

Here is a simple function that finds the pair of elements in the input list *xs* whose sum is closest to *n*:

```
function find(n: int, xs: list of ints) -> pair of ints:
    pairs = new list of (pair of ints)

    for i from 0 to length(xs):
        for j from i+1 to length(xs):
            pairs.add((xs[i], xs[j]))

    closest = pairs[0]

    for each p in pairs:
        if abs(n - sum(p)) < abs(n - sum(closest)):
            closest = p

    return closest
```

What is the asymptotic complexity of the function `find` in the number of elements *N* of `xs`? You may assume that adding an element to a list and all arithmetic operations (including calculating the sum of a pair) are O(1).

Write your answer in O-notation. Be as exact and simple as possible. Justify why the complexity of the function has this order of growth.

Answer

Complexity: $O(n^2)$

Justification (you can also add notes directly in the code above):

The first interesting block has two nested loops, with each nesting having O(*n*) iterations. Creating a pair and adding it to `pairs` is O(1). So in total, this block has complexity O(n²). We also observe that the list `pairs` has O(n²) elements afterwards.

The second interesting block has O(n²) iterations. Each iteration is O(1). So this block also has complexity O(n²).

Basic question 2: Sorting

Perform a quicksort partitioning of the following array, using the median of the first, middle and last element as pivot:

0	1	2	3	4	5	6	7	8
63	65	35	5	30	10	47	75	4

Note: quicksorting the left and right parts of the partition is not part of this question.

Answer

Write down how the array looks after the partitioning:

0	1	2	3	4	5	6	7	8
5	4	10	30	63	35	47	75	65

What sequence of swaps did you make when partitioning the array?

The pivot is the middle element 30. We partition using the Hoare partitioning scheme from the course. We start by swapping it (position 4) with position 0. We initialise lo = 1 and hi = 8 and start moving lo to the right and hi to the left, swapping when both indices are stuck:

- We swap positions 1 and 8 (65 and 4)
- We swap positions 2 and 5 (35 and 10).

Finally, lo and hi have crossed. We swap the pivot into place (hi), i.e. swapping positions 0 and 3.

If you used a different algorithm from that of your course, explain it here:

Basic question 3: Bug disinfection

A dynamic array should have *amortised* constant complexity, O(1), for adding and removing elements at the end. Here is an attempt at implementing a dynamic array (of strings):

```
class DynamicArray<Elem>:        (There is an unintended type mismatch
    internal: array of strings    between Elem and string. Pointing out
    size: int                     this as a bug is also accepted.)

    const sizeIncrease = 100

    method addLast(x: Elem):
        if this.size >= length(this.internal):
            this.resize()
        this.internal[this.size] = x
        this.size += 1

    method resize():
        maxSize = this.size + this.sizeIncrease
        oldInternal = this.internal
        this.internal = new array with maxSize cells
        for k from 0 to (but not including) maxSize:
            this.internal[k] = oldInternal[k]
```

(Note that the loop for k includes the starting index 0, but not the ending index maxSize.)

Unfortunately, two bugs have sneaked into the code somewhere, so it doesn't behave as it should.
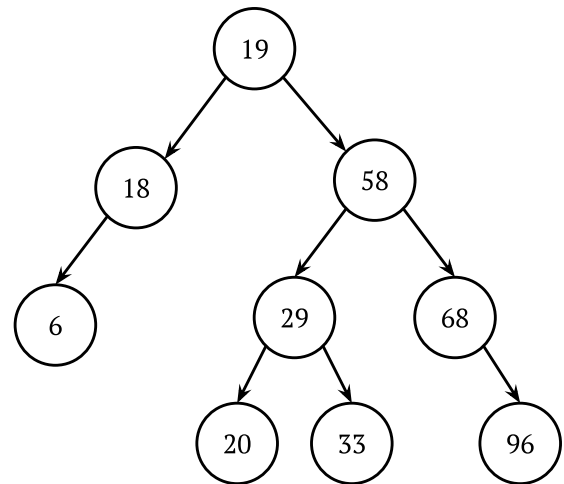
Answer

Mark the places in the code where the bugs are. Explain what the problem is and how to fix it:

Bug 1. When the dynamic array gets resized, a constant is added to its capacity (maximum size). Because of this, the amortised complexity of adding an element is not constant (instead, it is linear). To fix this, we can instead multiply the capacity by a constant factor greater than 1.

Bug 2. When the old array elements are copied over in `resize`, the loop goes over all indices from 0 to `maxSize`, which is `this.size` + `this.sizeIncrease`. But the old array only has `this.size` stored elements to copy over. (Indeed, the index may go beyond `this.internal.size`, which would lead to an exception or undefined behaviour.) To fix this, we iterate from 0 to `this.size` instead.

Basic question 4: Search trees

Consider the AVL tree to the right. In this question you are going to insert 25 into the tree using the AVL insertion algorithm. Additionally, you should annotate every node in the resulting tree with its AVL *balance factor* (height of the right subtree minus the height of the left subtree).
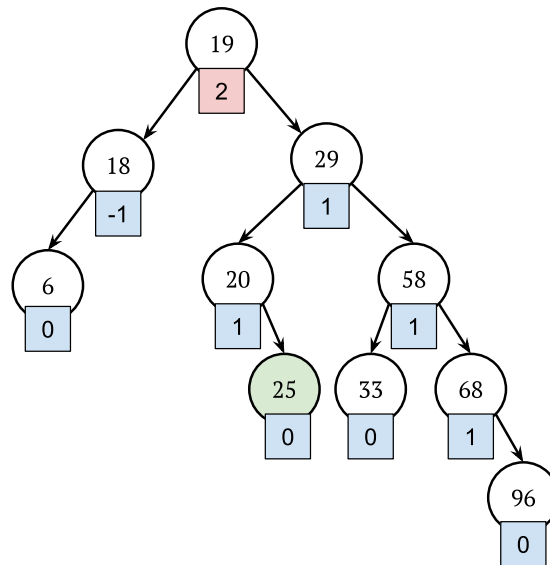

Answer

How does the tree look after you have inserted 25, but before rebalancing? Annotate the nodes in the resulting tree with their AVL balance factor.
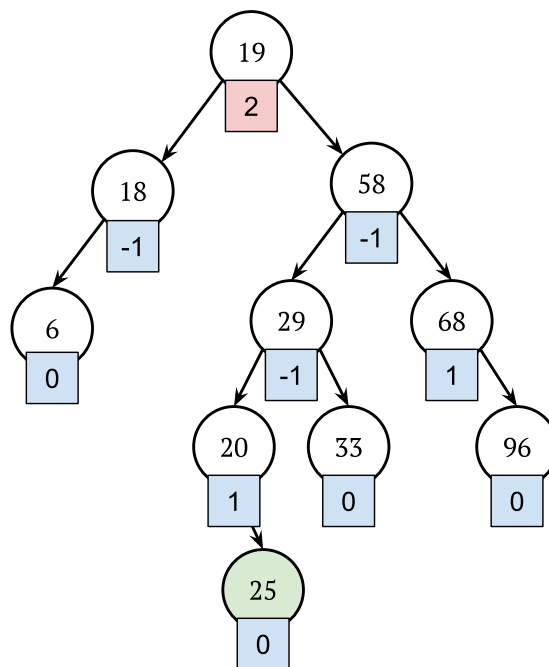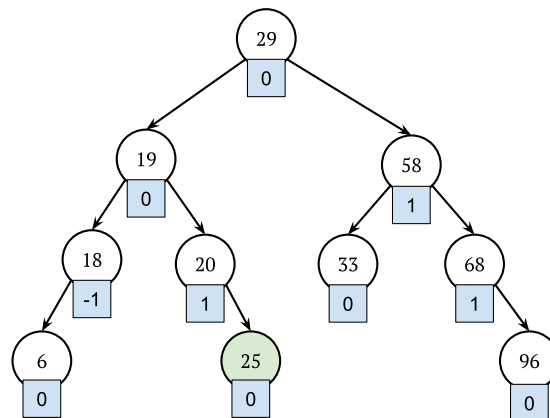
How does the tree look after rebalancing? Again, annotate the nodes with their balance factor.

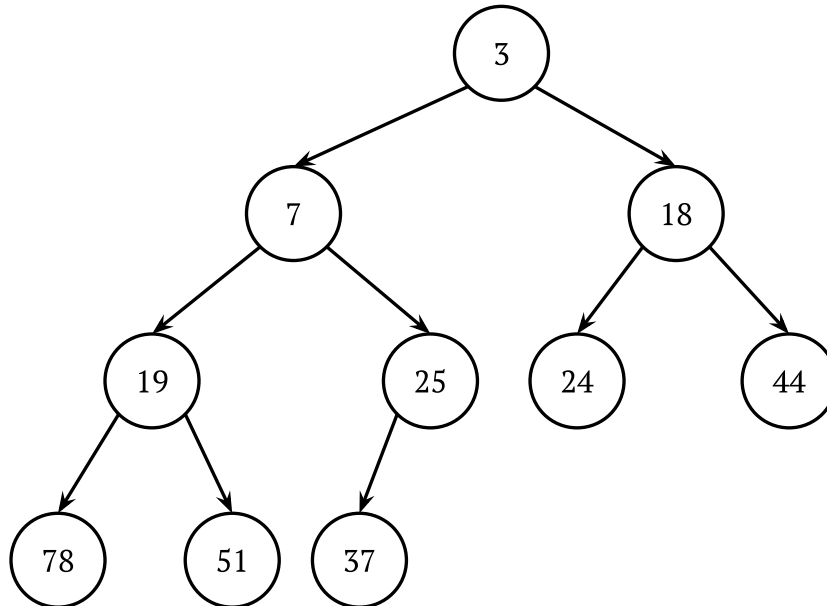It is a right-left case. Rotate the node 58 to the right:

Then rotate the node 19 (the root) to the left:

Basic question 5: Priority queues

You are given a minimum priority queue implemented as the following binary heap:

Answer

How does the above heap look when represented as an array?
You can choose whether to start at index 0 or index 1.

0	1	2	3	4	5	6	7	8	9	10
3	7	18	19	25	24	44	78	51	37	

Now remove the minimum element from the priority queue. How does the array look afterwards?

0	1	2	3	4	5	6	7	8	9	10
7	19	18	37	25	24	44	78	51		

Basic question 6: Hash tables

Suppose we have the following *open-addressing* hash table using *linear probing*. We are going to store strings (array of characters) consisting of the letters A, C, and T in the hash table. The hash code for the characters is: $h(A) = 1$, $h(C) = 3$, and $h(T) = 5$. The hash code for a string is the sum of the hash code of all characters in the string. For example, the hash code of ATT is:

$$h(\text{ATT}) = 1+5+5 = 11$$

The hash table uses the hash code to calculate an index by taking the remainder of the hash code divided by the size of the array (also called modular hashing):

$$index(\text{ATT}) = h(\text{ATT}) \% 10 = 11 \% 10 = 1$$

0	1	2	3	4	5	6	7	8	9
ACT	TT		C	CA	TTT	CC	AT		CCC

The hash table was created by adding the elements C, AT, CA, CC, TT, ACT, CCC, TTT **in an unknown order**. The array has never been resized, and no elements have been deleted. In what orders could the elements have been added?

    A) CC, CA, AT, C, CCC, TTT, ACT, TT

    B) TT, CCC, CC, TTT, AT, ACT, C, CA

    C) TT, CC, CA, TTT, AT, CCC, ACT, C

    D) C, ACT, TTT, TT, CC, CCC, AT, CA

    E) CA, CCC, ACT, C, TTT, CC, TT, AT

Determine which of these orders are possible (there may be several, or even none). For the others, explain why they are impossible.

Answer

Which orders are possible?   A and E

Explain why the others are impossible:

We have the following lookup indices: ACT:9, TT:0, C:3, CA:4, TTT:5, CC:6, AT:6, CCC:9.

- B and C are because TT would be inserted at position 0.
- D is impossible because ACT would be inserted at position 9.

Basic question 7: Graphs

Perform Dijkstra's algorithm on the graph to the right, starting in node **A**. Show each step of the algorithm: which node is removed from the priority queue, which node(s) are added to the priority queue, and what the priority queue looks like after each iteration.

Write the priority queue like this: "`X:4`, `Y:6`, `Z:8`", where the numbers are the priorities.
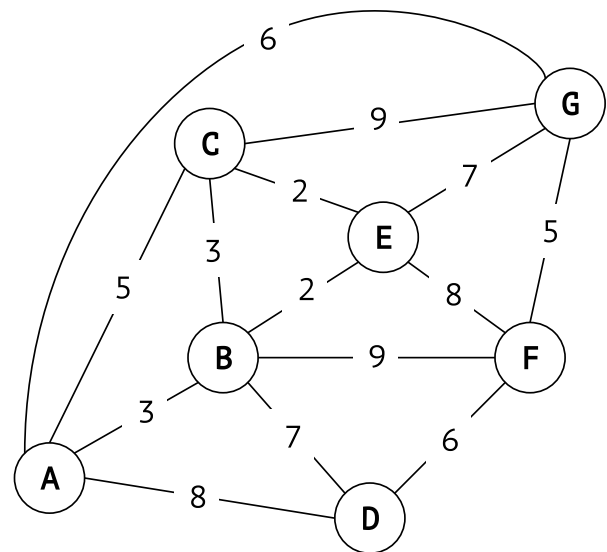
Answer

We save some work in the table by not adding entries for vertices we already visited. (Adding entries for all adjacent vertices is a simpler version of the algorithm, but will need extra rows.)

Removed node	Added node(s)	Priority queue after adding new nodes
—	A	A:0
A	B, C, D, G	B:3, C:5, G:6, D:8
B	C, D, E, F	C:5, E:5, C:6, G:6, D:8, D:10, F:12
C	E, G	E:5, C:6, G:6, E:7, D:8, D:10, F:12, G:14
E	F, G	C:6, G:6, E:7, D:8, D:10, F:12, G:12, F:13, G:14
C	none [visited before]	G:6, E:7, D:8, D:10, F:12, G:12, F:13, G:14
G	F	E:7, D:8, D:10, F:11, F:12, G:12, F:13, G:14
E	none [visited before]	D:8, D:10, F:11, F:12, G:12, F:13, G:14
D	F	D:10, F:11, F:12, G:12, F:13, F:14, G:14
D	none [visited before]	F:11, F:12, G:12, F:13, F:14, G:14
F	none	F:12, G:12, F:13, F:14, G:14
F	none [visited before]	G:12, F:13, F:14, G:14
G	none [visited before]	F:13, F:14, G:14
F	none [visited before]	F:14, G:14
F	none [visited before]	G:14
G	none [visited before]	

Basic question 8: Fill-in the blanks

Here is an iterative implementation of insertion into a set, represented as a binary search tree. However, someone has stolen parts of the code. Please repair it by filling in the blanks.

```
class BSTNode:
    value : number
    left  : BSTNode
    right : BSTNode

class BSTSet:
    root : BSTNode

    method add(value : number):
        parent = NULL
        node = this.root
        while node is not NULL:
            parent = node
            if value < node.value:
                node = node.left (or parent.left)
            else if value > node.value:
                node = node.right (or parent.right)
            else: // the value is already in the set
                return
        newnode = new BSTNode(value, NULL, NULL)
        if parent is NULL:
            this.root = newnode
        else if value < parent.value:
            parent.left = newnode
        else: // now we know that value > parent.value
            parent.right = newnode
```

Answer

Fill in all blanks so that the resulting code is correct.

Advanced question 9: Complexity

The following function `indexes` takes an integer *x* as input and returns a list of integers:

```
function indexes(x: int) -> list of ints:
    l = new list of ints; i = 0; n = 1

    while n <= x:
        if (x & n) > 0:        // bitwise and-operation to check
            l.addLast(i)       // if i-th bit is set, takes constant time
        i = i + 1
        n = n * 2

    return l
```

The function converts the decimal number x to a binary number (a binary number uses 2 as a base instead of 10, and consists of ones and zeros), and checks at which places (indexes) the binary digit is one. For example, the decimal number 13 is 1101 in the binary number system, and our function would then return the list [0,2,3], because the 0-th, 2-nd, and 3-rd places are one.

We use the `indexes` function in the following `powerset` function:

```
function powerset(xs: list of ints) -> list of list of ints:
    ps = new list of list of ints

    for i from 0 to 2^(length of xs):
        p = new list of ints

        for each j in indexes(i):
            p.add(xs[j])

        ps.add(p)

    return ps
```

that returns the 'power set' of the input list xs, that is all sub-lists including the empty list and the input list itself. For example, calling `powerset` on `[1,2,3]` gives:

```
[[],[1],[2],[1,2],[3],[1,3],[2,3],[1,2,3]]
```

Asymptotically, what are the **time complexities** of the `indexes` and `powerset` functions, related to the size of their input? Assume that adding to a list and indexing is O(1). The bitwise 'and' (&) operator is O(1) as well.

Write your answers in O-notation. Be as exact and simple as possible. Explain the key reasons why each complexity has the order of growth you give.

Answer

Note. The question asks for the asymptotic (time) complexities of `indices` and `powerset` in terms of the size of the input, but leaves it up to the reader to interpret what "size of the input" means in each case. Below, we make the following choices:

- complexity of `indices(x)` in terms of x,
- complexity of `powerset(xs)` in terms of the size of the list `xs`.

However, other interpretations are also accepted. For example, "size of the input x" could be taken to mean the number of bits needed to represent the number x, i.e. $\log_2(x)$. Then the complexity of `indices` would be linear instead of logarithmic.

Possible answer. For `indexes`, The while-loop in the function iterates over all bits of x, of which there are $\log_2(x)$. All inner operations take $O(1)$, so the total complexity of the function is $O(\log(x))$ (linear). Note also that the output list has $O(\log(x))$ elements.

For `powerset`, the outer loop has $O(2^n)$ iterations. Its body calls `indexes(j)`, which since $j < 2^n$ is $O(\log(2^n))$, simplifies to $O(n)$. The inner loop iterates over the list returned by `indexes(j)`, hence also takes $O(n)$. In total, this gives $O(n\,2^n)$ time complexity.

Advanced question 10: Disjoint arrays

Design an algorithm that takes two arrays, and returns true if the arrays are disjoint, i.e. have no elements in common.

You may freely use standard data structures and algorithms from the course in your solution, without explaining how they are implemented.

Your algorithm should take O($n \log(m)$) time, where n is the size of the larger array and m is the size of the smaller array.

Write down your algorithm as pseudocode (but Python, Java or Haskell are also fine). Explain which standard data structures and algorithms from the course that you have used.

Answer

We need to make the additional assumption that the type of elements of the two arrays is ordered, with a comparison operation taking O(1).

We store the elements from the smaller array in a set implemented using a self-balancing search tree. This allows for an efficient check if any given element of the larger array is in that set.

In pseudocode:

```
function disjoint(xs, ys) -> bool:
    if not xs.size < ys.size:
        (xs, ys) = (ys, xs)

    s = new set (using a red-black or AVL tree)
    for y in ys:
        s.add(y)

    for x in xs:
        if s.contains(x):
            return false

    return true
```

An alternative is to first sort the smaller array and then use binary search in it for each element of the larger array.

In both implementations, the first part takes O($m \log(m)$) and the second part takes O($n \log(m)$). Together, this is O($(m + n) \log(m)$). Since $n + m$ is O($\max(m, n)$) = O(n), it is also O($n \log(m)$).

Advanced question 11: Median Set

Your task is to create a data structure that can store a set of comparable elements, with special support for retrieving the *median* element in constant time. Note that we don't have any duplicates in a set. If the set is non-empty and the n elements in the set are:

$$x_0, \ x_1, \ ..., \ x_{n-1} \text{ with } x_{i-1} < x_i \text{ for all } 0 < i < n$$

then the median element is:

$$x_{\frac{n-1}{2}}$$

The other standard operations on a set should be supported as well. To summarise, the data structure should support the following operations, with time complexities as stated:

- add, O(log n)
- remove, O(log n)
- contains, O(log n)
- getMedian, O(1) (precondition: the set is non-empty)

You may freely use any standard data structure or algorithm from the course. Write down your data structure and the implementation of the operations using pseudo code, but you may use Java, Python or Haskell if you are more comfortable with that. Justify why your implementation of the operations has the required complexity.

Answer

There are many possible options. Here is one.

```
class MedianSet<E>:
    s: Set<E> (using a self-balancing binary search tree)
    lower, upper: SortedSet<E>
     median : E

    def balance():
        if upper.size() < lower.size(): upper.add(lower.removeMax())
        if lower.size() < upper.size(): lower.add(upper.removeMin())
        median = lower.getMax()  [if lower.size() > 0, else NULL]

    def add(x: E):
        if s.add(x) (membership changed):
            lower.add(x)
            balance()

    def remove(x: E):
        if s.remove(x) (membership changed):
            lower.remove(x)
            upper.remove(x)
            balance()

    def getMedian():
        return median
```

The invariants are as follows:
- s is the set of elements
- s is the disjoint union of lower and upper
- all elements in lower are smaller than all elements in upper
- `lower.size() - upper.size()` is 0 or 1
- `median` is always the maximum element in `lower` (this is to get O(1) complexity for getMedian)

In `add(x)`, we add *x* to `lower` so that the following `balance()` moves it to upper if appropriate.

From the invariants, we see that the median is given by the maximum of lower.

The required complexity is evident from the logarithmic complexity of operations on (sorted) sets when implemented using self-balancing binary search trees (e.g. red-black or AVL trees).

Remark. Another strategy would be to use a self-balancing BST with nodes storing the size of their subtree. This allows for finding the *i*-th element in order in O(log(*n*)) where *n* is the size. To make `getMedian` constant time, we store it as a separate value and recompute it whenever we add or remove.

Advanced question 12: Bipartite graphs

An undirected graph is *bipartite* iff the nodes can be divided into two disjoint subsets V_0 and V_1 such that all edges have one endpoint in V_0 and another in V_1. To the right is an example of a bipartite graph with V_0 = {A,B,C} and V_1 = {D,E,F}.
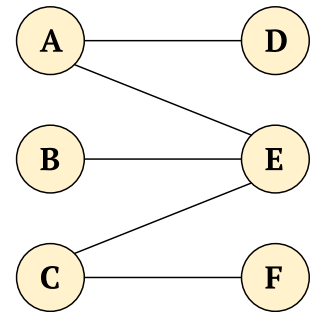
Design an efficient algorithm that takes a connected graph as input and returns true if and only if the graph is bipartite. By efficient we mean that its runtime complexity should be *better* than quadratic, O(N^2). Also describe which data structure(s) you use to implement the graph.

What is the asymptotic runtime complexity of your algorithm, in terms of the size N? (The size of the graph is the total number of vertices and edges, or $N = |V| + |E|$). Explain your reasoning.

Answer

There are many possible options. We describe just one.

We solve the following more general problem: given a graph (not necessarily connected) and a vertex v, we determine if the connected component of v is bipartite. This specialises to the given problem by taking any vertex of the graph (a vertex exists because the graph is connected).

We perform a graph search (the traversal strategy doesn't matter). Instead of a visited set, we use a visited map that stores a boolean value for each vertex. This value determines which part of the partition the vertex needs to belong to. We arbitrarily say that the starting vertex is in the part corresponding to `true`.

Here is a recursive presentation of the graph search using an inner function:

```
function bipartite(g: graph, v: vertex of g) -> bool:
    visited = new map (e.g. using AVL tree) from vertices to bool

    def visit(w: vertex of g, b: bool):
        if w in visited with value c:
            if b != c: return false in outer function
        else:
            visited.put(w, b)
            for each edge w -> w' in g:
                visit(w', not b)

    visit(v, true)
    return true
```

Ignoring the for-loop, each call of `visit` is O(log(|V|)). The for-loop makes a call to `visit` for each outgoing edge. Its cost is accounted for by the cost of the nested call to `visit`. In total, `visit` is called at most |E| many times (once for each outgoing edge). That gives O(|E| log(|V|)), so in particular O(N log(N)).