

Basic question 1: Quicksort pivot selection

Consider the following 9-element array:

| | | | | | | | | |
|----|----|----|----|----|----|----|----|----|
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
| 23 | 47 | 17 | 82 | 58 | 94 | 32 | 70 | 66 |

You want to partition the whole array, as the first step in running Quicksort.

- a) For each of the following pivot selection strategies, specify what element will be the pivot, and how big the left and right parts will be after performing the partitioning.

| Strategy | Pivot index | Pivot value | Left part size | Right part size |
|--------------|-------------|-------------|----------------|-----------------|
| First-index | 0 | 23 | 1 | 7 |
| Middle-index | 4 | 58 | 4 | 4 |
| Last-index | 8 | 66 | 5 | 3 |

- b) Which of the strategies gives the best partitioning *for this specific array*?

Middle-index

- c) Give an example of a 5-element array where that strategy (i.e., your answer to question (b)) will give the **worst possible partitioning**:

| | | | | |
|----|----|----|----|----|
| 0 | 1 | 2 | 3 | 4 |
| 11 | 22 | 55 | 44 | 33 |

(There are an infinite number of answers.)

- d) What will now be the pivot value in (c)?

The pivot will be 55.

Any array where the pivot is the unique smallest or the largest element will be the worst possible, because it results in a partition with an empty left or right part.

Basic question 2: Hash tables

To the right is an open addressing hash table containing binary numbers. It uses modular compression (using the modulo operator) and linear probing (with probing constant 1, as usual).

The hash function is the **number of ones** in the binary number.

(Note: the numbers in parentheses are *not* the hash values – they are just the decimal representations of the binary numbers, so that, e.g., $11100111_2 = 231_{10}$.)

- a) Give one possible ordering in which the numbers could have been added to the hash table. It's enough to use the decimal representations in your answer.

(Note: no elements have been deleted from the table.)

231, 183, 254, 0, 20, 255, 2

also known as:

11100111, 10110111, 11111110, 00000000, 00010100, 11111111, 00000010

(There are many possible orderings. For example, 00010100 (20) could have been added any time earlier in the sequence.)

| | |
|---|----------------|
| 0 | 11111110 (254) |
| 1 | 00000000 (0) |
| 2 | 00010100 (20) |
| 3 | 11111111 (255) |
| 4 | 00000010 (2) |
| 5 | |
| 6 | 11100111 (231) |
| 7 | 10110111 (183) |

The table is almost overfull, so we have to resize it.

- b) Insert all the numbers into the table of size 11 to the right, **in the same order** as the order you gave in (a) above.

You don't have to write both the binary and decimal representation – either of them is fine.

| | |
|----|----------------|
| 0 | 00000000 (0) |
| 1 | 00000010 (2) |
| 2 | 00010100 (20) |
| 3 | |
| 4 | |
| 5 | |
| 6 | 11100111 (231) |
| 7 | 10110111 (183) |
| 8 | 11111110 (254) |
| 9 | 11111111 (255) |
| 10 | |

Basic question 3: Unbalanced BSTs

You create a binary search tree by inserting the first N positive integers in *reverse order* into an initially empty tree using the standard insertion procedure.

So, you start by inserting N into an empty tree, then you insert $N-1$, ..., 3, 2, and finally 1.

- a) What is the root node value? N
- b) What are the value(s) of all the leaf nodes (if any)? 1 (there's only one leaf)
(**Note**: a leaf node is a node with only null children.)
- c) What is the time complexity of building the final tree? $O(N^2)$

Now you want to delete all elements, one at the time, using the standard deletion procedure.

- d) In which order should you delete the elements to get the **worst possible** time complexity?
 the opposite of the insertion order, i.e., increasing (1, 2, 3, ..., N)
- e) What is the time complexity of deleting all elements in this order? $O(N^2)$
- f) In which order should you delete the elements to get the **best possible** time complexity?
 the same as the insertion order, i.e., decreasing (N , ..., 3, 2, 1)
- g) What is the time complexity now? $O(N)$

Note: In questions (c), (e) and (g) we mean the *asymptotic time complexity* of the whole procedure (i.e., adding/deleting all elements, not just one single element), in terms of the number of elements N . Please answer in O -notation, and as tight and simple as possible.

Basic question 4: Priority queues

Here is a broken binary max-heap with 10 elements represented by a dynamic array:

| | | | | | | | | | | |
|----|----|----|----|---|----|---|---|---|---|------|
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
| 23 | 14 | 20 | 16 | 8 | 13 | 7 | 2 | 7 | 5 | null |

- a) Two array elements are misplaced and need to swap to make this a correct heap. What are their indices? Specify their values as well for clarity.

Indices: 1 and 3 Values: 14 and 16

Now, answer the following two questions on the *corrected* heap (after you have swapped the misplaced elements):

- b) Which cells of the array would be modified (written to) in the worst case when adding a new element to the corrected heap? (Specify all indices.)

indices 10, 4, 1 and 0 – happens when adding a number greater than 23

- c) Which cells of the array would be modified (written to) when deleting the maximum from the corrected heap using the standard algorithm? (Specify all indices.)

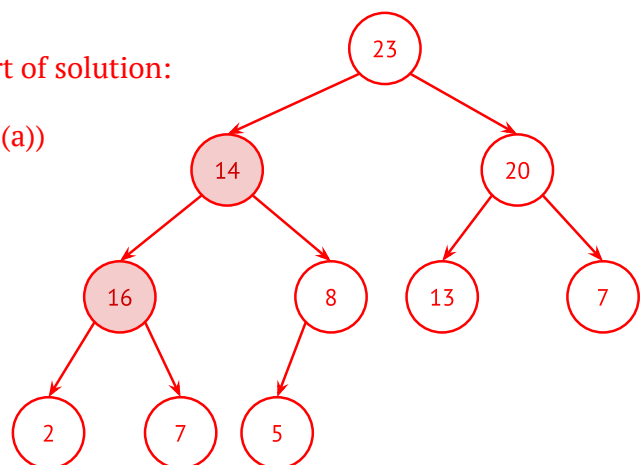
indices 9, 0, 2, 5

Note: This is performed on the original corrected array, i.e., (b) never happened.

Hint: Assigning null to a cell counts as a modification.

Tree form for explanatory purposes only, not part of solution:

The two filled nodes are misplaced (corrected in (a))

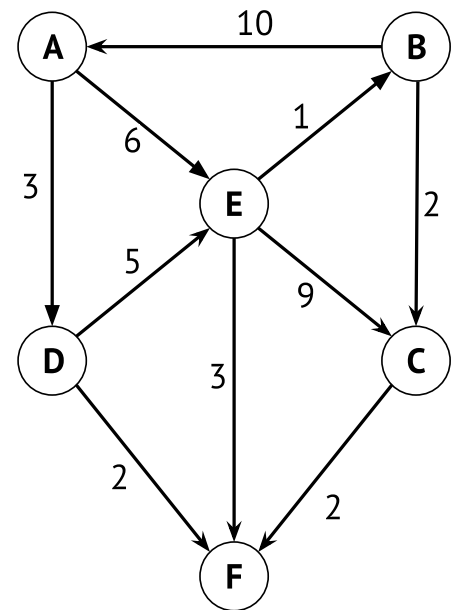


Basic question 5: Graphs

We run this familiar piece of code on the graph to the right:

```
visited = new set of nodes
agenda = new min-priority queue of pairs of cost and node
agenda.add( (0, A) )
while agenda is not empty:
    (cost, node) = agenda.removeMin()
    if not visited.contains(node):
        visited.add(node)
        for e in outgoingEdges(node):
            agenda.add( (cost + e.weight, e.target) )
```

What are the contents of the set *visited* and the priority queue *agenda*, initially and after each of the first three iterations of the while loop? (Some values are pre-filled for explanatory purposes.)



| Iteration | <i>visited</i> | <i>agenda (in order of priority)</i> |
|-------------------|----------------|--------------------------------------|
| before the loop | — | (0, A) |
| after iteration 1 | A | (3, D), (6, E) |
| after iteration 2 | A, D | (5, F), (6, E), (8, E) |
| after iteration 3 | A, D, F | (6, E), (8, E) |
| after iteration 4 | A, D, F, E | (7, B), (8, E), (9, F), (15, C) |

Basic question 6: Complexity

Below are two (mostly nonsense) functions, each taking an integer array as a parameter.

State the asymptotic time complexity of each function **in terms of the length N of the array**.

In each case, **briefly state** how you concluded this (you may do this by annotating the programs).

Notes:

- Assume that all arithmetic operations and array indexing take $O(1)$ time.
- Answer in O -notation, and as tight and simple as possible.

Algorithm 1

function fun1(arr):

$i = 0$ [Mistake in question. Should be: $i = 1$]

while $i < \text{length}(\text{arr})$:

for $j = 0$ **to** $\text{length}(\text{arr})$:

$\text{arr}[i] = \text{arr}[i] + \text{arr}[j]$

$i = i \times 2$

Complexity: $O(N \log(N))$

Justification:

The outer while-loop does $\log(N)$ iterations, the inner for-loop always does N .

The body of the inner for-loop is $O(1)$.

Note: With $i = 0$ as in the original question, the function does not terminate and thus does not have a defined time complexity.

Algorithm 2

function fun2(arr):

$\text{result} = 0$

$i = \text{length}(\text{arr}) - 1$

while $i \geq 0$:

$\text{result} = \text{result} \times \text{arr}[i]$

if $\text{arr}[i] \% 2 == 1$:

$i = i - 1$

else:

$i = i - 2$

return result

Complexity: $O(N)$

Justification:

The while-loop decreases i by one or two in each iteration, so it runs at most N (and at least $N/2$) iterations.

The body of the loop is $O(1)$.

Advanced question 7: Word chains

You are given a list of words and two letters of the alphabet (called *start* and *goal*), and your task is to come up with an algorithm that finds a word chain between *start* and *goal*, using as few words as possible.

A *word chain* is a sequence of words from the given list, where the first letter of each word is the last letter of the previous word in the sequence. In addition, the first word must begin with the starting letter and the last word must end with the goal letter.

Example: For the word list [*ape*, *axis*, *data*, *dog*, *end*, *salad*, *sinus*], and *start=a*, *goal=g*, there are two word chains with three words in them. One is [*ape*, *end*, *dog*] and the other is [*axis*, *salad*, *dog*], and your algorithm should return either of those. There are many other possible chains for this list of words, but they all have more than three words.

So, the following call:

```
wordchain(["ape", "axis", "data", "dog", "end", "salad", "sinus"], "a", "g")
```

should return either ["ape", "end", "dog"] or ["axis", "salad", "dog"].

Also state the asymptotic worst case time complexity of your algorithm, in terms of the number of words N . Please answer in O -notation, and as tight and simple as possible. Motivate your answer.

You can use every data structure and algorithm in the course, but be clear about how they are used.

We can view the problem as a graph:

- The nodes are letters.
- The edges from c to d are words starting with c and ending with d .
- Finding a word chain between *start* and *goal* with as few words as possible means finding a shortest path from *start* to *goal*.

We use adjacency lists (a map from letters to lists of outgoing words) to represent this graph. To build it, we simply go over the given N words and add each word.

We can then use breadth-first search to find a shortest path. It has $O(N)$ iterations.

The asymptotic time complexity depends on how the map from letters to lists of words is implemented. For the lists, it does not matter if we use dynamic array lists or linked lists. For the map, we have the following options:

- Use a self-balancing search tree. Every access is $O(\log(N))$, so the overall asymptotic complexity is $O(N \log(N))$.
- If characters are encoded using unique natural number codes, we can use an array of lists. Every access is $O(1)$, so the overall asymptotic complexity is $O(N)$.

Advanced question 8: Mystery data structure

Here is a chunk of pseudocode defining two classes:

```
class Y:
    a: integer
    b: Y

class X:
    t: array of Y
    k: integer

    def f(x):
        i = x % length(t)
        q = t[i]

        if q is null:
            return

        if q.a == x:
            t[i] = q.b
            k = k - 1
            return

        while q.b is not null:
            if q.b.a == x:
                q.b = q.b.b
                k = k - 1
                return
            q = q.b
```

What does this strange code represent?

- a) What common data structure is defined by the class **X**?
- b) What do the instance variables **t** and **k** represent?
- c) What does the method **f** do?

Be as precise as possible and use correct technical terms taught in the course.

- a) Separate chaining hash table for a set of integers, using modular hashing.
- b) **t** is the internal hash table, where the cells are linked lists of integers.
k is the number of elements (not the size of the internal table).
- c) **f** deletes an element from the set.

Advanced question 9: Anagram finder

An anagram is a word or phrase formed by rearranging the letters of another word or phrase, e.g., "listen" is an anagram of "silent". Your assignment is to make a data structure for finding anagrams in large sets of words, with two operations:

- **addWord(w)**: adds the word **w** to the data structure.
- **anagrams(w)**: returns a list of all anagrams of **w** that have been added to the data structure (this should include **w** itself only if it has been added).

Example: Assume that I run **addWord("silent"); addWord("listen"); addWord("stilton")** on an empty instance of the data structure. Then **anagrams("listen")** and **anagrams("enlist")** should both return **["silent", "listen"]** (or **["listen", "silent"]**). In contrast, **anagrams("tonsil")** should return the singleton list **["stilton"]**.

- a) Describe how you implement the data structure and the two operations. Make it as efficient as you can. You can use every data structure and algorithm in the course, but be clear about how they are used.

Hint: the words "cabs" and "scab" are both anagrams of the character sequence "abcs".

- b) What is the complexity of the **anagrams** operation in terms of the number N of separate anagram lists in the data structure (in the example, there would be two anagram lists, one for "listen"/"silent" and one for "stilton"). Assume that the number of characters K in the word searched for does not exceed $\log(N)$, meaning that K is in $O(\log(N))$.

Explain your reasoning.

Note: remember that comparing two strings is linear in the length of the strings, i.e., $O(K)$ if K is the number of characters in the strings.

- a) We can use an AVL map from strings to lists of strings, with the keys being sorted character sequences. To add a word, sort its characters and add the word to the list of strings the sorted sequence maps to. To find anagrams, sort the word using Mergesort and look it up in the map (defaulting to an empty list for missing keys).
- b) First we have to sort the letters in the input word: Mergesort has complexity $O(K \cdot \log(K))$, where K is the number of letters. Since K is $O(\log(N))$, sorting takes $O(\log(N) \cdot \log(\log(N)))$.

Then we look up the sorted string in the AVL tree: this makes $O(\log(N))$ comparisons, and each comparison takes $O(K)$ time. The total time is therefore $O(\log(N) \cdot K) = O(\log(N) \cdot \log(N))$, which is slightly slower than the sorting time.

Therefore, the worst case complexity is $O(\log(N)^2)$.

Note 1: We can use Quicksort or even insertion sort to sort the letters. Then the worst case for sorting will be $O(K^2) = O(\log(N)^2)$. This will not change the overall time complexity (unless we're using a hash map – see below).

Note 2: We can also use a hash map instead of an AVL tree. Then in the worst case (if we are unlucky with the hash function) we will use $O(N)$ comparisons, so the complexity will be $O(N \cdot K) = O(N \cdot \log(N))$. If the hash function is good, the expected number of comparisons will be constant, $O(1)$, so the expected complexity will be $O(1 \cdot K) = O(\log(N))$. In this case the time for sorting dominates, so the total expected complexity is $O(\log(N) \cdot \log(\log(N)))$.

Note 3: An even faster alternative is to implement the data structure as a trie. Then the lookup time of the sorted letters will be $O(K) = O(\log(N))$, and then the sorting time will dominate.