DAT038/TDA417
Data structures and algorithms
Written examination
Solution suggestions

Thursday, 2021-01-14, 14:00–18:00, in Canvas and Zoom

Examiner(s)	Peter Ljunglöf, Nick Smallbone, Christian Sattler.
Allowed aids	Course literature, other books, the internet. ***You are not allowed to discuss the problems with anyone!***
Individualised	In most questions you will be asked to work on an individual problem instance. You will get your individualised problem from a specific Canvas page. ***The solutions in this document are for one specific problem instance!***
Assorted notes	You may answer in English or Swedish. Excessively complicated answers might be rejected. We need to be able to read and understand your answer!
Exam review	If you want to discuss the grading, please contact Peter via email.

There are 6 basic, and 3 advanced questions, and two points per question. So, the highest possible mark is 18 in total (12 from the basic and 6 from the advanced). Here is what you need for each grade:

- To **pass** the exam, you must get 8 out of 12 on the basic questions.
- To get a **four**, you must get 9 out of 12 on the basic questions, plus 2 out of 6 on the advanced.
- To get a **five**, you must get 10 out of 12 on the basic questions, plus 4 out of 6 on the advanced.

Grade	Total points	Basic points	Advanced
3	⩾ 8	⩾ 8	—
4	⩾ 11	⩾ 9	⩾ 2
5	⩾ 14	⩾ 10	⩾ 4

Question 1

This is how your individual program could have looked like:

```
boolean f(int[] xs):
    ys = new dynamic array of int
    for i in 0 .. xs.length-1:
        for j in i .. xs.length-1:
            ys.add(xs[i] + xs[j])

    boolean r = false
    for m in ys:
        for n in ys:
            if m + n == 0:
                r = true
    return r
```

And this is a possible individual question:

> When you run the program on your computer in 2021 on an array of size 1000, it takes one day. Your computer has gotten faster by a factor of 4 every 9 years. What is the largest array size you could have processed in one day in 1967?

Part A solution

- The first group of for-loops has complexity $O(n) \cdot O(n) \cdot O(1) = O(n^2)$: each loop has $O(n)$ iterations and the body has amortized complexity $O(1)$.
- After it has executed, **ys** has $O(n) \cdot O(n) = O(n^2)$ elements.
- With this information, the second group of for-loops has complexity $O(n^2) \cdot O(n^2) \cdot O(1) = O(n^4)$ (same reasoning as before).
- Altogether, the complexity is $O(n^2) + O(n^4) = O(n^4)$.

Part B solution

- We have (2021 - 1967) / 9 = 54 / 9 = 6, so my computer is $4^6 = 2^{12}$ times faster in 2021 than it was in 1967. So one day in 2021 corresponds to 2^{12} times as many computation steps than one day in 1967.
- Let $T(n)$ denote the run time in terms of number of computation steps on an array of size n. From Part A, we have $T(n) \approx C\, n^4$ for some constant C.
- We want to find n such that $T(1000) = 2^{12}\, T(n)$. This means $1000^4 \approx 2^{12}\, n^4$, i.e., $1000 \approx 2^3\, n = 8\, n$.
- So the answer is $n \approx 125$.

Question 2

Here is a possible individual input array:

`23 36 82 91 77 45 19 63 50`

Part A solution

- We use the partitioning algorithm from the lectures here, but there are several possible algorithms that all are correct.
- The median of (23, 77, 50) = 50 so we start by swapping **23** and **50**.
- The following table shows the steps in the partitioning. The numbers in **red bold** are the numbers that are going to be swapped at each stage.

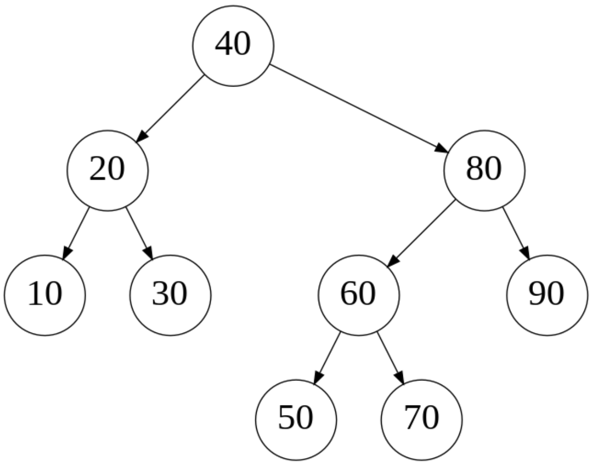
50	36	**82**	91	77	45	19	63	**23**
50	36	23	**91**	77	45	**19**	63	82
50	36	23	19	**77**	**45**	91	63	82
50	36	23	19	**45**	77	91	63	82
[45	36	23	19]	50	[77	91	63	82]

Part B solution

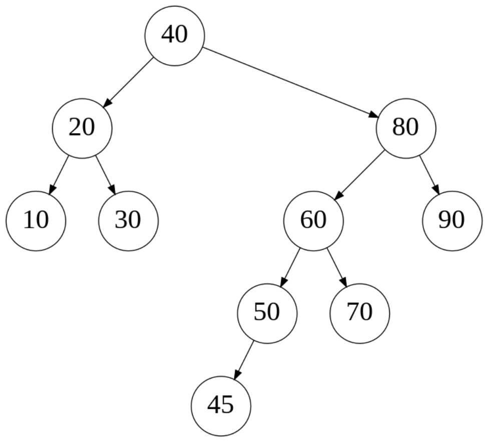
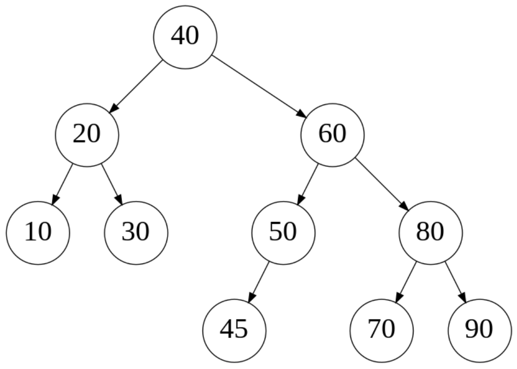
- Swap 19 with 50, or 19 with 77
- Then the three median elements will be (19, 23, 50) or (19, 23, 77), and the median of three will be 23. This is the second-smallest element of the array, and then there will be one partition with only one element, 19.

Question 3

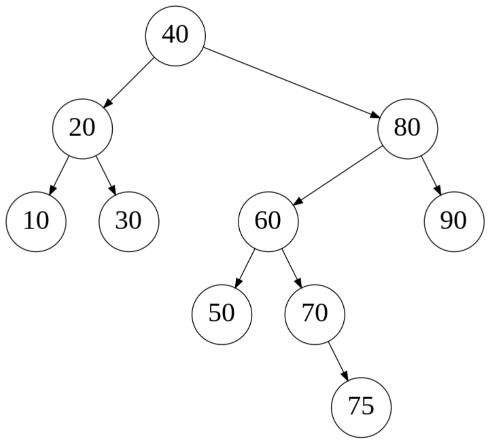
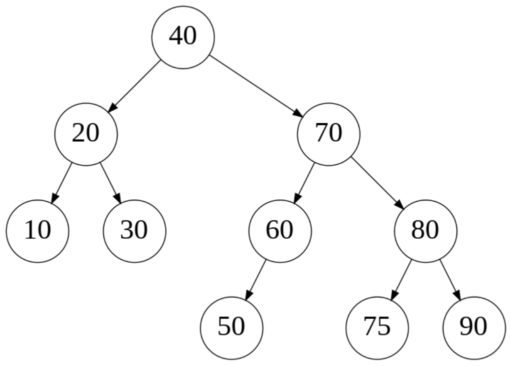
Here is one possible individual tree:



Part A solution

1) To create a left-left tree, we can insert a node as a **child of 50**. E.g. inserting 45 (but e.g. 55 also works):	2) Now, **node 80 is a left-left tree.** We solve it by **rotating 80 right**, to get:
	

Part B solution

1) To create a left-right tree, we can insert a node as a **child of 70**. E.g. inserting 75 (but e.g. 65 also works):	2) Now, **node 80 is a left-right tree.** We solve it by **first rotating 60 left,** then **rotating 80 right**, to get:
	

Question 4

Here’s a possible individual skeleton code:

```
List<Int> mergePQs(PriorityQueue<Int> pq1, PriorityQueue<Int> pq2):
    List<Int> result = new DynamicArrayList<Int>()
    while not (pq1.isEmpty() and pq2.isEmpty()):
        ???
    return result
```

Part A solution

Here is one possible solution:

```
List<Int> mergePQs(PriorityQueue<Int> pq1, PriorityQueue<Int> pq2):
    List<Int> result = new DynamicArrayList<Int>()
    while not (pq1.isEmpty() and pq2.isEmpty()):
        if pq1.isEmpty():
            // pq2 must be non-empty
            result.add(pq2.removeMin())
        else if pq2.isEmpty():
            // pq1 must be non-empty
            result.add(pq1.removeMin())
        else:
            // neither pq is empty
            if pq1.getMin() <= pq2.getMin():
                result.add(pq1.removeMin())
            else:
                result.add(pq2.removeMin())
    return result
```

But any solution is allowed as long as it is correct and doesn’t use extra data structures/library functions.

Part B solution

- The while-loop runs O(n) times taking O(log n) time per iteration.
  - `PriorityQueue.removeMin` is O(log n)
  - `DynamicArrayList`.add is O(1) (amortised)
- So the total complexity is O(n log n).

Question 5

Here is one possible individual instance:

```
[J, M, -, Q, K, N, -, L, P]
```

And with indivudual hash values:

```
hash(J) = 9
hash(K) = 13
hash(L) = 16
hash(M) = 8
hash(N) = 3
hash(P) = 7
hash(Q) = 12
```

Part A solution

- This gives the following table indices, after mod 9, and conflict resolution:

```
Indices 3-5:        Q =  3
                    K =  4
                    N =  3  -->  4  -->  5
Indices 7-8, 0-1:   J =  0
                    L =  7
                    P =  7  -->  8
                    M =  8  -->  0  -->  1
```

- Which in turn gives the following constraints on the insertion order:

```
L < P < M,  Q < N,  J < M,  K < N
```

- So one possible insertion order is: `L P J M Q K N`

Part B solution

- The hash values will give the following table indices (after mod 5):

```
L = 1,  P = 2,  J = 4,  M = 3,  Q = 2,  K = 3,  N = 3
```

- So the final hash table will look like this (if we add elements in the insertion order, and always add elements to the front of the linked list):

```
0:  -
1:  L
2:  Q --> P
3:  N --> K --> M
4:  J
```

Question 6

Here is one possible instance of the graph:

```
Edge weights:
QT:11, QX:13, PT:6, PX:5, PS:7, PZ:3, SX:1, SZ:4, RZ:10, RU:12, RY:8, UY:9

Or equivalently, inserted into the graph:

   Q -- 13 -- X --- 1 --- S           R --- 8 --- Y
    \          \         / \         / \         /
     \          \       /   \       /   \       /
     11          5     7     4    10    12     9
       \          \   /       \   /       \   /
        \          \ /         \ /         \ /
         T -- 6 --- P --- 3 --- Z           U
```

Part A solution

- **A is a spanning tree**
- **B is not a spanning tree**, because it is not connected
- **C is a shortest path tree** (SPT) from node S
- **D is a spanning tree**
- **E is a minimum spanning tree** (MST)

Part B solution

Prim's algorithm:

- will add **edge RZ** with weight 10
- (the minimum-weight edge that has exactly one node in the current MST)

Kruskal's algorithm:

- will add **edge RY** with weight 8
- (the minimum-weight edge that will not create a cycle when added)

Question 7

Here is one possible set of order-of-growth functions:

- A(n, k) = O((n + 18 sqrt(n)) (1 + 18 sqrt(k))).
- B(n, k) is the complexity of the below algorithm for `cs` of size n and `j` equal to 0:
```
boolean has_sum(int[] cs, int j, int y):
    if j == cs.length: return y == 0
    return has_sum(cs, j + 1, y) or has_sum(cs, j + 1, y - cs[j])
```
- C(n, k) is the complexity of binary search on a sorted array of size n k.
- D(n, k) = O((n + n k + k)^{12}).
- E(n, k) is the complexity of the below algorithm for `set` of size n and `ys` of size k:
```
void track_large(RedBlackSet<Int> set, int[] ys):
    for y in ys:
        set.add(y)
        set.remove(set.min())
```

Part A solution

- **A(n, k) = O(n sqrt(k))**
  because only the asymptotically biggest summand matters and constant factors don't matter (and O(-) preserves products).
- **B(n, k) = O(2^n)**
  because the program considers all possible subsets of xs to sum, of which there are 2^n. Everything outside the recursive structure is constant. k does not play a role.
- **C(n, k) = O(log(n k)) = O(log(n) + log(k))**
  because binary search is logarithmic in the array size.
- **D(n, k) = O((n k)^{12}) = O(n^{12} k^{12})**
  because only the asymptotically biggest summand matters (and O(-) preserves products).
- **E(n, k) = O(log(n) k)**
  because k times three operations logarithmic in n are performed.

Part B solution

A graph with the edges C → A → D and C → E → D, and node B is isolated. We have:

- **C ⩽ A,** because O(log(n)), O(log(k)) ⩽ O(n sqrt(k)).
- **C ⩽ E,** because O(log(n)), O(log(k)) ⩽ O(log(n) k).
- **A ⩽ D,** because O(n) ⩽ O(n^{12}) and O(sqrt(k)) ⩽ O(k^{12}).
- **E ⩽ D,** because O(log(n)) ⩽ O(n^{12}) and O(k) ⩽ O(k^{12}).
- A is not related to E because, for n = 1, A is strictly smaller than E, and for k = 1, it is strictly larger than E.
- B is not related to any of A, C, D, E because for n = 1 it is strictly smaller than the rest and for k = 1 it is strictly larger than the rest.

Question 8

Here is an example of an array in strict cyclic order: `[53, 62, 81, 94, 15, 22, 43]`

Part A solution

Calling `findMinIndex` on the example array above should return 4 because this is the index of the smallest element (15). There are several possible solutions, but all are variants of binary search. The main idea is to select the interval where the first element is larger than the last – because the smallest element must be in that interval:

```
int findMinIndex(int[] A):
    // First check if the array is ordered.
    if A[0] ⩽ A[A.length - 1]:
        return 0
    // Invariant: left < right, A[left] > A[right]
    int left = 0
    int right = A.length - 1
    // Goal: right = left + 1. Then right is the index of the minimum.
    while right - left > 1:
        mid = (left + right) / 2  // Rounding doesn't matter.
        if A[left] > A[mid]:      // A[left] > A[mid] < A[right]
            right = mid
        else:                     // A[left] < A[mid] > A[right]
            left = mid
    return right
```

Note that we have to handle the special case when the array is already ordered, i.e., when the first element is the smallest.

Part B solution

First we find the index of the smallest element using part A. Then we can compare the searched element `v` with the first array element `A[0]` to decide which part of the array we have to search in. This part is then guaranteed to be sorted, so we can use standard binary search to look for `v`. Like this:

```
boolean contains(int[] A, int v):
    orderStart = findMinIndex(A)
    if v ⩾ A[0]:
        return binarySearch(A, 0, orderStart - 1)
    else:
        return binarySearch(A, orderStart, A.length - 1)
```

Ther1e are many other possibilities, but all involve binary search in some way. E.g.:

- It’s possible to tweak the part A solution directly instead of calling it.
- Binary search in the *virtual list* of same size as A that sends index i to `A[(orderStart + i) mod A.length]`

Question 9

One solution idea is to swap the graphs after each iteration. Like this:

```
List<Node> searchBi(Node start, Node goal):
    pqueue = new PriorityQueue(comparing (e) -> e.costToHere)
    pqueue.add(new PQEntry(start, 0, null))
    Map<Node,PQEntry> visited = new Map()

    graphR = graph.reverseCopy()
    pqueueR = new PriorityQueue(comparing (e) -> e.costToHere)
    pqueueR.add(new PQEntry(goal, 0, null))
    Map<Node,PQEntry> visitedR = new Map()

    while pqueue and pqueueR are not empty:
        PQEntry entry = pqueue.removeMin()

        if entry.node in visitedR:
            // now we've found a solution
            entryR = visitedR.get(entry.node)
            return extractPath(entry) + reverse(extractPath(entryR))

        if entry.node not in visited:
            visited.set(entry.node, entry)
            for Edge edge in graph.outgoingEdges(entry.node):
                costToNext = entry.costToHere + edge.weight
                pqueue.add(new PQEntry(edge.to, costToNext, entry))

        swap pqueue and pqueueR
        swap visited and visitedR

    // if we end here we have failed
    return null
```

Note that we change the goal test, from checking if the current node is the goal, to checking if the current node has been visited by the reverse search.

To extract the solution path we just concatenate the two solution paths – we just have to remember to reverse one of them. To be able to extract the path for the reverse search, we have to find the PQEntry that led to the node, and therefore we use a map for the visited nodes, instead of a set as in the original one-directional search.

One minor problem that this solution doesn't handle is that the extracted path might be reversed (from the goal to the start). This can be handled in many ways – one is to keep a boolean flip-flop variable, saying if the current `pqueue` is the original or the reversed.