Midterm var2

MIDTERM, 2022-04-29

Question 1: Consider the following algorithm which runs on a (zero-based) array `A[]`. The length of the array is n; the elements of array are non-negative integers smaller than 2^m (i.e. `unsigned` type using m bits each).

The initial call is COMPUTESOMETHING(`A[]`, n); after that the function calls itself recursively. This function calls GCD(x, y) or GCD(x, y, z) (the *greatest common divisor* of two or three numbers). This function uses Euclidean algorithm; you may assume that its time complexity is $O(\max(|x|, |y|))$ or $O(\max(|x|, |y|, |z|))$ respectively, where $|x|$ denotes the length of argument x (in bits).

COMPUTESOMETHING(`A[]`, n)
  if $n == 1$:
    return $A[0]$
  else if $n == 2$:
    return GCD$(A[0], A[1])$
  else if $n == 3$:
    return GCD$(A[0], A[1], A[2])$
  else if $n \equiv 0 \pmod 5$:
    return GCD$(A[n-1], A[n-2], A[n-3])$ + COMPUTESOMETHING(`A[]`, $n-3$)
  else:
    return GCD$(A[n-1], A[n-2])$ + COMPUTESOMETHING(`A[]`, $n-2$)

(A) Denote by $T(n, m)$ the time complexity of this algorithm on an array with n elements of size m. Express $T(n, m)$ in terms of $T(\ldots)$, where one or both arguments are smaller to reflect how the time complexity of the recursive call affects the time complexity of the overall algorithm.

(B) Find some $g(n, m)$ such that $T(n)$ is in $O(g(n))$ and justify your answer. If multiple asymptotic bounds are possible, select the smallest one among them.

Answer:

(A) The algorithm is recursive – it replaces the call to COMPUTESOMETHING(`A[]`, n) into a call COMPUTESOMETHING(`A[]`, $n-2$) or COMPUTESOMETHING(`A[]`, $n-3$).

The time complexity of this algorithm is a recurrent relation (just on the parameter n; not on m – the upper bound on the length of parameters, which does not change). Denote by $C_1 m$ the upper bound when you compute the GCD for two or three arguments of length m. (Such a constant C_1 must exist, since we know that the *Euclidean algorithm* to compute GCD takes $O(m)$ time ($O(\max(|x|, |y|))$ or $O(\max(|x|, |y|, |z|))$ is actually $O(m)$, since the length in bits $|x|, |y|, |z| \leq m$).

$$T(n, m) = \begin{cases} C_1 m & \text{if } n \leq 3, \\ C_1 m + T(n-3, m) & \text{if } n \equiv 0 \pmod 5 \\ C_1 m + T(n-2, m) & \text{if } n \not\equiv 0 \pmod 5 \end{cases}$$

(B) For example, if you start by $n = 17$, you would have:

$$T(17, m) = C_1 m + T(15, m) = C_1 m + C_1 m + T(12, m) = C_1 m + C_1 m + C_1 m + T(10, m) =$$
$$= C_1 m + C_1 m + C_1 m + C_1 m + T(7, m) = C_1 m + C_1 m + C_1 m + C_1 m + C_1 m + T(5, m) =$$
$$= C_1 m + C_1 m + C_1 m + C_1 m + C_1 m + T(2, m) = C_1 m + C_1 m + C_1 m + C_1 m + C_1 m + C_1 m = 6C_1 m.$$

As we evaluate the recursion, the number of recursive calls (6 in our case for $n = 17$) is somewhere between $n/2$ and $n/3$. In either case it is $O(n)$. If we do $O(n)$ recursive calls and each call costs $O(m)$ (as GCD algorithm). The ultimate upper bound of $T(n, m)$ is $O(n \cdot m)$.

Question 2: Assume that you know the degrees of some graph with 6 vertices; and we need to know what kind of graph can or cannot be constructed, if the degrees are like the given ones.

(A) Write a sequence of six numbers that are values in the interval $[1; 5]$ such that there is **no graph** with the given degrees. (Or explain, why such sequence does not exist.)

(B) Write a sequence of six numbers that are values in the interval $[1; 5]$ such that there exists a graph with such degrees, but there is **no bipartite graph** with the given vertices. (Or explain, why such sequence does not exist.)

(C) Write a sequence of six numbers that are values in the interval $[1; 5]$ such that there exists a graph with such degrees that there exists a bipartite graph with the given vertices, but this graph does not have a perfect matching. (Or explain, why such sequence does not exist.)

Note: To make (B) more tricky, you can ask the degrees to be such that the sum of degrees equals 14 and one can build a connected graph out of it, but not a bipartite one.

Answer:

(A) You can pick vertex degrees that add up to an odd number. For example,

$$2, 2, 2, 3, 3, 3.$$

Because of the handshake theorem $2 + 2 + 2 + 3 + 3 + 3 = 15 = 2|E|$, where E' is the number of edges in the graph. Clearly it cannot be a fraction. Hence a graph with such degrees cannot exist.

(B) You can have various sequences of degrees that will never serve as degrees of a bipartite graph. For example,

$$5, 5, 5, 5, 5, 5.$$

These degrees means that this is a complete graph K_6.

A more minimalistic example would be

$$5, 2, 2, 1, 1, 1.$$

(C) It is certainly possible to build a graph that is bipartite and does not have a perfect matching. Consider the following degrees:

$$1, 2, 1, 1, 2, 1.$$

You can create two disconnected paths of length 3 (such graph is bipartite, since any tree is bipartite; so the union of two trees is also bipartite). On the other hand, it does not have a perfect matching; no matter how you select two edges, there will be two vertices that are unmatched (each one is in a different component of the disjoint graph).

Note: It is still possoible to build another bipartite graph with the vertex degrees $(1, 2, 1, 1, 2, 1)$ and where the perfect matching exists. How?

If you wish to create list of degrees for which a bipartite graph exists, but it cannot have a perfect matching, then consider this sequence:

$$2, 2, 2, 2, 4, 4.$$

It shows a full graph $K_{2,4}$ with two partitions (two and four vertices on the respective sides).

Question 3: Assume there is a queue implemented as a cyclical array. A queue is implemented as an array with $size = 15$ elements; it has two extra variables $front$ (pointer to the first element) and $length$ (the current number of elements in the queue). Enumeration of array elements starts with 0. The array is filled in a circular fashion. The command `enqueue(elt)` inserts a new element at $(front + length)$ mod $size$. The `enqueue(elt)` command also increments the `length`.

The command `dequeue()` does not change anything in the `array`, but increments `front` by 1 and decreases `length` by 1. Thus the queue becomes shorter by 1.

The initial state of the queue is the following:

```
size = 15
front = 0
length = 0
array[] = 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
```

The following list of size 25 contains all prime numbers from $[1; 100]$. After that we enqueue five elements, dequeue three elements (and repeat these actions five times).

```
list<int> L = {2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61,
↪ 67, 71, 73, 79, 83, 89, 97};
for (int i = 0; i < 5; i++) {
    queue.enqueue(L[5*i]);
    queue.enqueue(L[5*i+1]);
    queue.enqueue(L[5*i+2]);
    queue.enqueue(L[5*i+3]);
    queue.enqueue(L[5*i+4]);
    queue.dequeue();
    queue.dequeue();
    queue.dequeue();
}
```

Show the final state of the queue (`front`, `length` and also the contents of the `array`).

Answer:

```
# Stage #1: After inserting 2, 3, 5, 7, 11       (length += 5)
front = 0
length = 5
array[] = 2 3 5 7 11 0 0 0 0 0 0 0 0 0 0
# After three deletes                  (front += 3;  length -= 3)
front = 3
length = 2
array[] = (2) (3) (5) 7 11 0 0 0 0 0 0 0 0 0 0

# Stage #2: After inserting 13, 17, 19, 23, 29   (length += 5)
front = 3
length = 7
array[] = (2) (3) (5) 7 11 13 17 19 23 29 0 0 0 0 0
# After three deletes                  (front += 3;  length -= 3)
front = 6
```

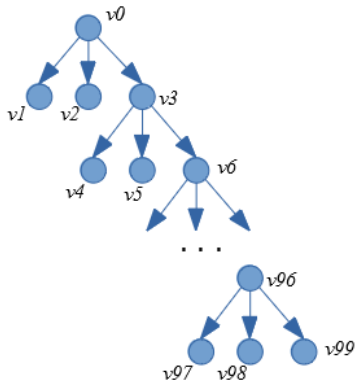
(continues on next page)

(continued from previous page)

```
length = 4
array[] = (2) (3) (5) (7) (11) (13) 17 19 23 29 0 0 0 0 0

# Stage #3: After inserting 31, 37, 41, 43, 47   (length += 5)
front = 6
length = 9
array[] = (2) (3) (5) (7) (11) (13) 17 19 23 29 31 37 41 43 47
# After three deletes                   (front += 3;  length -= 3)
front = 9
length = 6
array[] = (2) (3) (5) (7) (11) (13) (17) (19) (23) 29 31 37 41 43 47

# Stage #4: After inserting 53, 59, 61, 67, 71   (length += 5)
front = 9
length = 11
array[] = 53 59 61 67 71 (13) (17) (19) (23) 29 31 37 41 43 47
# After three deletes                   (front += 3;  length -= 3)
front = 12
length = 8
array[] = 53 59 61 67 71 (13) (17) (19) (23) (29) (31) (37) 41 43 47

# Stage #5: After inserting 73, 79, 83, 89, 97   (length += 5)
front = 12
length = 13
array[] = 53 59 61 67 71 73 79 83 89 97 (31) (37) 41 43 47
# After three deletes                   (front += 3;  length -= 3)
front = 0
length = 10
array[] = 53 59 61 67 71 73 79 83 89 97 (31) (37) (41) (43) (47)
```

Eventually we get queue with 10 filled in elements $(53, \ldots, 97)$. All the array elements that are in deleted/invalid state are written in parentheses.

Question 4: We build a ternary tree as follows: Add node v_0. Then 33 times pick the rightmost leaf on the last level and add three children to it. You will end up with a tree with 100 nodes (see figure):



(A) Run the post-order traversal of this tree; find the four "middle" vertices a_{48}, a_{49}, a_{50}, a_{51}.

(B) Run the in-order traversal of this tree; find the four "middle" vertices b_{48}, b_{49}, b_{50}, b_{51}. (In-order traversal of a ternary tree – visit the leftmost subtree, then the parent, then the two remaining subtrees).

Note: Both the $a_0, \ldots, a_{99}$ and $b_0, \ldots, b_{99}$ are zero-based (and they in some order traverse through the vertices

$v_0, \ldots, v_{99}$). In both cases you need to find the "middle four" vertices during the traversal.

Answer:

(A) In the post-order traversal you need to visit all the children (left to right) before visiting the parents.

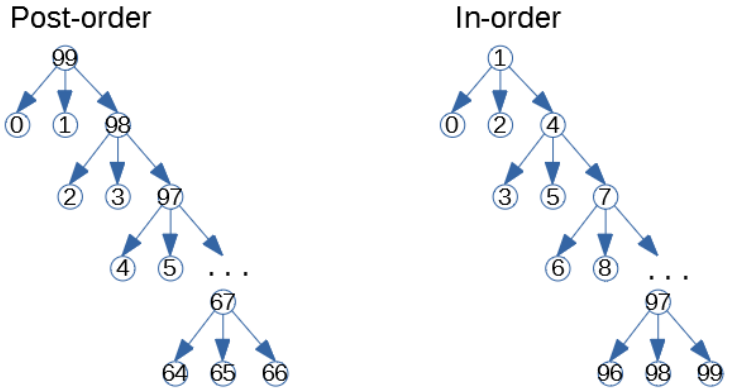
$$a_0 = v_1,\ a_1 = v_2,\ a_2 = v_4,\ a_3 = v_5,\ a_4 = v_7,\ a_5 = v_8,\ \ldots$$

We need to skip the first 48 vertices ($a_0, \ldots, a_{47}$). We have $a_{2k} = v_{3k+1}$ and $a_{2k+1} = v_{3k+2}$. When $k = 24$ and $k = 25$ the vertices are the following:

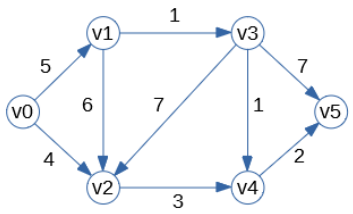
$$a_{2\cdot 24} = v_{73},\ a_{2\cdot 24+1} = v_{74},\ a_{2\cdot 25} = v_{76},\ a_{2\cdot 25+1} = v_{77}.$$

(B) The in-order traversal (for this particular kind of tree) we visit the vertices in an order that is very similar to the pre-order (which is the original order of vertices). In particular, in the in-order traversal all the pairs (v_0, v_1), (v_3, v_4), (v_6, v_7), ... switch their order. On the other hand, vertices $v_2, v_5, v_8, \ldots$ (everything congruent to 2 modulo 3) does not change the order at all. We have the following:

$$b_{48} = v_{49},\ b_{49} = v_{48},\ b_{50} = v_{50},\ b_{51} = v_{52}.$$



Question 5: Run the Bellman-Ford algorithm to find the minimum distance from the source v_0 to all the other vertices.



The pseudocode of Bellman-Ford algorithm is this:

BELLMANFORD(G, w, s):
    for each vertex $v \in V$: *(initialize vertices to run shortest paths)*
        $v.d = \infty$
        $v.p =$ NULL
    $s.d = 0$ *(the distance from source vertex to itself is 0)*
    for $i = 1$ **to** $|V| - 1$ *(repeat $|V| - 1$ times)*
        for each edge $(u, v) \in E$

if $v.d > u.d + w(u,v)$: *(relax an edge, if necessary)*
$v.d = u.d + w(u,v)$
$v.p = u$

As you run the algorithm, build a table (current distances from the source v_0 to all the other vertices) every time when some distances change (due to edge relaxing). The table looks something like this (but at each stage you specify actual edge that was relaxed – instead of (v_i, v_j); and also the actual distances).

Vertices	v_0	v_1	v_2	v_3	v_4	v_5
Initial distances	0	∞	∞	∞	∞	∞
Relax (v_i, v_j)	?	?	?	?	?	?
...	...					

Make sure that in the **for each** loop you visit all the edges in their lexicographical order. Show all the the relaxed edges in this table, but in case some edge does not result in changes of any distances, do not enter it into the table.

Answer:

After we run the algorithm, we get the following table (every time some edge is relaxed). This time it was sufficient to run just one iteration of Bellman-Ford algorithm (in some worst-case scenarios – long cycles etc.) you may need $|V| - 1$ iterations, where each iteration passes through all the edges in the graph.

Vertices	v_0	v_1	v_2	v_3	v_4	v_5
Initial distances	0	∞	∞	∞	∞	∞
Relax (v_0, v_1)	0	5	∞	∞	∞	∞
Relax (v_0, v_2)	0	5	4	∞	∞	∞
Relax (v_1, v_3)	0	5	4	6	∞	∞
Relax (v_2, v_4)	0	5	4	6	7	∞
Relax (v_3, v_5)	0	5	4	6	7	13
Relax (v_4, v_5)	0	5	4	6	7	9