MIDTERM, 2022-04-06

Question 1: Consider the following algorithm which can run on an array A[]; we also specify the leftmost and the rightmost element of the array in every call of COMPUTESOMETHING(...). The initial call is COMPUTESOMETHING(A[], 0, $n-1$); after that the function calls itself recursively. It may also call $f(\ell)$, which runs in time $O(\ell)$.

COMPUTESOMETHING(A[], ℓ, r)
  total = 0
  if $\ell - r \leq 2$:
    total = *total* + $f(\ell)$
  else:
    $m_1 = \lfloor (2\ell + r)/3 \rfloor$
    $m_2 = \lfloor (\ell + 2r)/3 \rfloor$
    total += COMPUTESOMETHING(A[], ℓ, m_1)
    total += COMPUTESOMETHING(A[], m_1, m_2)
    total += COMPUTESOMETHING(A[], m_2, r)
  return *total*

(A) Denote by $T(n)$ the time complexity of this algorithm on an array with $n-1$ elements. Write the recurrence for the time complexity of this algorithm. Namely, express $T(n)$ in terms of $T(\ldots)$ for some smaller arguments.

(B) Apply Master Theorem to find some $g(n)$ such that $T(n)$ is in $\Theta(g(n))$.

Answer:

(A) Let us compute the array size in the call of this function by $r - \ell + 1$ (1 is added, since both endpoints are included). Thus, the first call on array $[0; n-1]$ uses array length equal to $(n-1) - 0 + 1 = n$.

The recurrence should reflect the three recursive calls of this function. Take into account that the recursive calls are now on shorter fragments of the input array – all the new array lengths are $m_1 - \ell + 1$, $m_2 - m_1 + 1$, $r - m_2 + 1$. They are all less or equal than $\lceil n/3 \rceil$. Computing the m_1 and m_2 takes $O(1)$ time.

So the recurrence relation describing the time complexity is

$$T(k) = 3T(\lceil k/3 \rceil) + O(1), \text{ if } k > 4$$
$$T(k) = O(n), \text{ if } k <\leq 3$$

Observe that the last line contains the only expression that is $O(n)$ (does not depend on the current length of the array being processed (the variable k), but rather on the length of the original array n (more precisely, it depends on ℓ, the numeric value of the left endpoint of the input, which does not decrease as the size of the processed array decreases).

We replace $O(\ell)$ by $O(n)$, because ℓ changes every time, but n is the upper bound of all values ℓ.

(B) By Masters theorem the number of recursive calls of COMPUTESOMETHING$(\ldots)$ is in $O(n \log n)$. Since the recursion every time ends with a call that costs $O(n)$, we need to multiply this estimate from the Masters theorem by $O(n)$ to get the overall complexity $O(n^2 \log n)$.

Question 2: Some binary tree T has exactly 100 *internal nodes*. Other nodes in this tree are *leaves* – they also contain information payload (keys, values, etc.), but they do not have non-empty children (both child pointers are NULL).

(A) Can tree T be a full binary tree? Can tree T be a complete binary tree? Can tree T be a perfect binary tree?

To remind the tree terminology: In a full binary tree every node has either two children or no children at all. Complete trees are used to implement heaps (nodes are filled in level by level). In perfect binary trees all leaves have the same depth.

(B) What is the largest and the smallest value for n – the total number of nodes in the tree T? Explain your estimates.

(C) What is the largest and the smallest value for the NULL pointers in this tree (such pointers do not count as internal nodes or leaves).

(D) What is the largest and the smallest value for h – the height of T? Explain your estimates.

Answer:

(A) The tree T can be full (every time you need a new internal node, add to it two children – until you reach the necessary number of children).

The tree can be complete (you build a perfect binary tree containing all the nodes, but leave the last level incomplete – fill it in from the left side).

The tree cannot be perfect, because the only perfect trees (containing all levels filled in up to the maximum) may contain $0, 1, 3, 7, 15, 31, 63, 127, \ldots$ internal nodes; in general it must be $2^k - 1$. The number 100 is not in this list.

(B) The smallest number of nodes is $100 + 1$ (we just build a long skinny path and add one leaf at the very bottom).

The largest number of nodes is $100 + 101$ (for every internal node add two children; then the count of leaves are always one more than the count of the internal nodes).

(C) The count of `Null` pointers in a binary tree is always one more than the number of nodes.

- If the total number of nodes is 101 (the smallest one), then the number of `Null` pointers is 102.
- If the total number of nodes is 201 (the largest one), then the number of `Null` pointers is 202.

(D) The largest height is for the skinny tree (with 101 nodes). The height is 100.

The smallest height is for the tree which is complete. It is easy to figure out its height inductively:

- If there are up to 3 inner nodes, the complete binary tree has height 2
- If there are up to 7 inner nodes, the complete binary tree has height 3
- If there are up to $2^k - 1$ inner nodes, the complete binary tree has height k

In our case there are up to $127 = 2^7 - 1$ inner nodes, so the height must be 7.

Question 3: An array of 10 elements are inserted into a minimum heap in the order specified here:

$$1, 7, 9, 4, 5, 3, 6, 8, 2, 10.$$

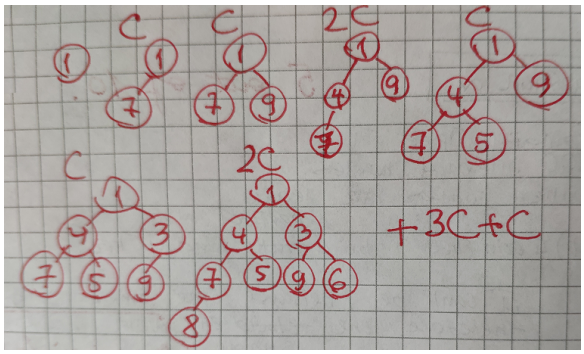
Assume that we do not use any fast heap-building algorithms; we just insert new elements and let them sift up.

(A) Show the final state of the tree after all the nodes are inserted in this way. Draw this heap as a complete binary tree. (You can also show intermediate results to show your approach.)

(B) What is the total number of comparisons ($a <^? b$) that is used during this heap building process.

Answer:

(A) The stages of building the heap are shown in the image:

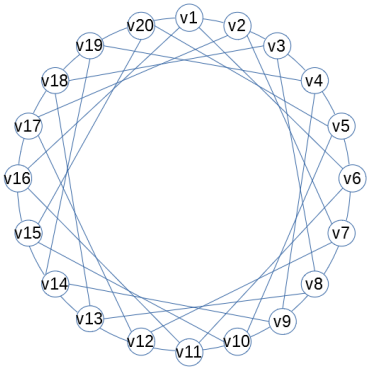


For each heap insert operation we also write the number of comparisons (C, 2C etc.) needed to complete that operation.

(B) It needs about 13 comparisons (typically, 1 comparison per one heap insert; sometimes 2 comparisons). In larger trees the cost per one heap insert is $O(n \log n)$.

(In fact, it is possible to do cheaper heap inserts if we need to build a heap from the predefined list of nodes. If you proceed by layers bottom up, it takes $O(n \log n)$ steps.)

Question 4: Consider the following regular 20-gon as a graph. It has 20 vertices $V_1, \ldots, V_{20}$, it has exactly 40 undirected edges – all sides of the 20-gon, and also the diagonals that connect vertices having distance exactly 5. Namely, (V_i, V_j) exists in the graph iff $(i - j) \equiv \pm 1 \pmod{20}$ or $(i - j) \equiv \pm 5 \pmod{20}$.



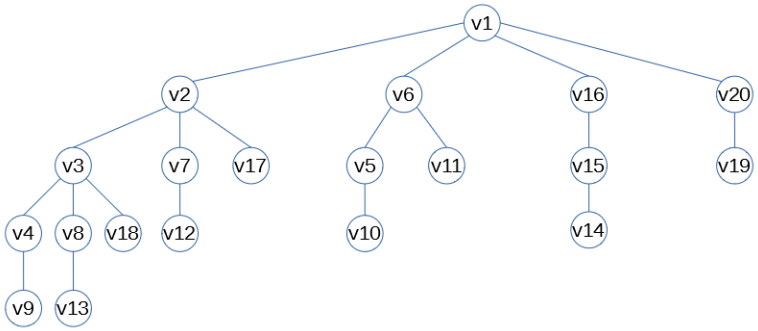
(A) Draw the BFS tree that is created if this graph is traversed in the BFS order, and vertex V_1 is the root. Make sure to show the labels of all vertices. Assume that the children for each internal node in the BFS tree are visited in the order of increasing numbers (namely, if a parent node V_i discovers two neighbors V_j and V_k where $k > j$, then V_k is a sibling drawn to the right of V_j).

(B) What is the number of internal nodes in this BFS tree? What is the number of leaves? What is the height of the BFS tree (the number of edges leading from its root to the deepest leaf)?

(C) Consider a graph – regular 100-gon with edges that are all its sides and also those diagonals that connect vertices with distance exactly 5. What is the height of the BFS tree created from this graph?

Answer:

(A) The BFS tree is shown in the picture below:



(B) The number of internal nodes is 11, the number of leaves is 9. The height of the tree is 4 – the shortest paths from v_1 to the worst vertices (either v_9 or v_{13}) have length 4.

(C) In case of a regular 100-gon, we need to find the worst vertices – the ones which are furthest away from v_1, if we are only allowed to jump 1 or 5 units back or forth.

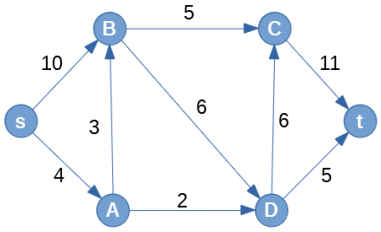
Consider vertices v_{49} and v_{53}. Both of them can be reached within steps:

$$v_1 \to v_2 \to v_3 \to v_4 \to v_9 \to v_{14} \to v_{19} \to v_{24} \to v_{29} \to v_{34} \to v_{39} \to v_{44} \to v_{49}.$$
$$v_1 \to v_2 \to v_3 \to v_8 \to v_{13} \to v_{18} \to v_{23} \to v_{28} \to v_{33} \to v_{38} \to v_{43} \to v_{48} \to v_{53}.$$

In both cases the paths are of length 12, so the BFS tree height must also be 12. There can be **no** vertices that need longer paths. You can classify all the vertices in this graph accordingly to the remainder when dividing by 5.

- Among all the vertices v_i such that $i \equiv 4 \pmod 5$ the vertex v_{49} is the furthest away from v_1. All the others need fewer jumps of length 5 (either clockwise or counter-clockwise).
- Among all the vertices v_i such that $i \equiv 3 \pmod 5$ the vertex v_{53} is the furthest away from v_1. Verification is similar to the previous case.

Question 5: Run the Edmonds-Karp maximum flow algorithm on the graph provided.



(A) Run Edmonds-Karp algorithm on the graph shown above. For every phase highlight the the augmenting path (or simply list its vertices), find the *residual flow* of this augmenting graph. Next to it draw a copy of the flow graph where every age is labeled by two numbers `f/c` – the actual flow `f` and also the capacity `c` of the edge. Thus, every phase shows two oriented graphs: First: The current residual graph (initially – it is simply the given graph with all flows equal to 0). Second: The original graph with edge capacities and new flows.

(B) Finally, redraw the original graph with all the maximum flows (use the same two-number labels for edges `f/c`). Show the min-cut which prevents any further augmenting paths (either highlight with another color, or simply list the partition of graphs vertices into two disjoint sets that describe the cut).

Answer:

See "Max Flow Handout" under Week9 in E-Studijas. It contains examples how to write down such solutions.