Algoritmer, datastrukturer och komplexitet EDAF05

Emil Wihlander
dat15ewi@student.lu.se

May 26, 2017

Exam 140526

Analysis of algorithms

1. **(a)** $\boxed{\text{A}}$ Because N needs to be twice as big every time you want to print one more star the algorithm has logarithmic growth.

(b) $\boxed{\text{D}}$ The function f will call g N times which means g will print $0, 1, \dots, N$ stars. The total number of stars will therefore be the sum of that serie of numbers. The formula for arethmetic sums is $N \cdot (0 + N)/2 = (1/2) \cdot N^2$ which give it an order of $O(N^2)$.

Divide-and-conquer

2. **(a)** $\boxed{\text{A}}$ Subtree, you can easily break it down into subproblems by recursively solving it for the children. The conquer part is a bit more tricky. You need to save both the largest perfect subtree (lps) and the largest perfect subtree with the top node as base (tps). With that information you can calculate the lps and tps one level up, see **(b)**.

(b) The following function solves it by a recursive divide-and-conquer algorithm. `Subtree` is a class that contains the largest perfect subtree (`.lps()`) and the largest perfect subtree with the top node as base (`.tps()`). `TreeNode` contains the children (`.left()` and `.right()`).

```
private Subtree f(TreeNode node) {
    // base case
    if (node == null)
        return new Subtree(0, 0)
    // DIVIDE
    Subtree left = f(node.left());
    Subtree right = f(node.right());
    // CONQUER
    // Builds the new top perfect subtree (tps)
    int tps = 1 + Math.min(left.tps(), right.tps());
    // Finds the new largest perfect subtree (lps)
    int lps = Math.max(left.lps(), right.lps(), tps);

    return new Subtree(x, y);
}
```

(c) $\boxed{\text{B}}$ Since there are no cycles it will traverse through each node exactly once. The conquer step is $O(1)$ which means the algorithm will run in $O(n)$ where n is the number of nodes.

Graph connectivity

3. (a) $\boxed{\text{B}}$ Clearance, see **(b)** for explanation.

(b) Start with an empty graph, add each $w \in E$ in ascending order until there is a connection between s and t. If you use Breadth-first search for finding a connection the worst case performance for each iteration is $0, 2, 4, \ldots, 2 \cdot n$ the aretmethic sum is $2\sum_{k=0}^{n} k = 2 \cdot n^2/2 = n^2$ the order is therefore $O(n^2)$.

Dynamic programming

4. (a) $\boxed{\text{C}}$ Long. Since we know all edges are directed from left to right we analyze all vertices from left to right. The value for each vertex will be the value of the max of all past vertices that is connected (if none is, assign value 0) to the current vertex plus one, and that is the dynamic programming part.

(b) $\boxed{\text{C}}$ The problem is one-dimensional and therefore only one of the $OPT(i)$ options can be correct ($\boxed{\text{C}}$ or $\boxed{\text{G}}$). The past vertices you should compare are depented on which edges exist rather than vertices with the same iterative distance. $\boxed{\text{C}}$ is therefore the only possible option.

(c) $\boxed{\text{B}}$ The algorithm will iterate through all vertices, n, and edges, m, exactly once $(n + m)$. The maximum number of incoming edges is for $v_1 = 0, v_2 = 1, \ldots, v_n = n - 1$ which gives the aretmethic sum of $m \leq (n-1)^2/2$ which is larger than n for large values. The order of the algorithm is therefore $O(m)$.

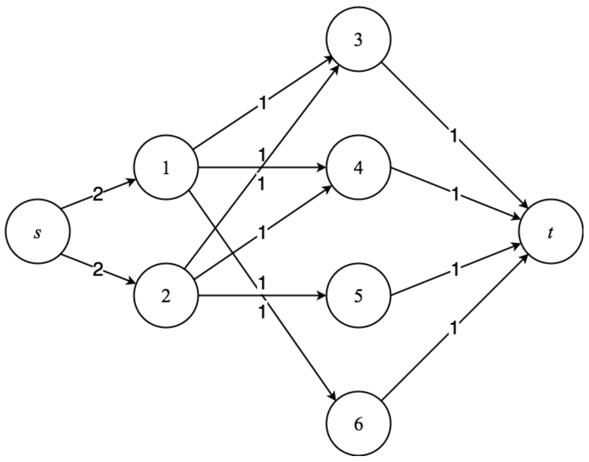
(d) $\boxed{\text{A}}$ You will save one value per vertex (maximum path length to current vertex) and the space order will therefore be $O(n)$.

Network flow

5. (a) $\boxed{\text{D}}$ V, you can look at this as a maximum flow problem, see **(b)** for futher explanation.

(b) Let L be the first column of vertices and R the second column vertices. Then add the edges from the original problem and set their weight to 1, add edges from s to all vertices in the first column with the weight 2 and finally add edges from all vertices in the second column to t with the weight 1.

If the maximum flow equals $2n$ there is a solution, otherwise not. The example from the problem definition is visualized below.



(c) $\boxed{\text{D}}$ As seen in **(b)**, the number of nodes is $n + m + 2 \Rightarrow O(n + m)$

Computational complexity

6. (a) $\boxed{\text{E}}$ Crossing, the specific case of when $k = 0$ is the Hamiltonian path problem which means Crossing is at least as hard. Hamiltonian path problem is NP-complete which therefore means Crossing is at least NP-complete as well.

(b) $\boxed{\text{H}}$ See **(a)**.

(c) $\boxed{\text{B}}$ "The easiest way to see that is to take the following NP-hard problem, Hamiltonian path, and prove that Crossing is at least as hard as Hamiltonian path (Hamiltonian path $\leq_p$ Crossing)"

(d) Given an instance to problem Hamiltonian path, H, consting of $G(V, E)$, construct an instance of Crossing, C, consting of $G'(V', E'), k$. Let $G = G'$ and $k = 0$. This reduction runs in constant time.

If T is a solution to G it's also a solution to C. Hence Hamiltonian path $\leq_p$ Crossing.

Exam 130529

Analysis of algorithms

1. (a) $\boxed{\text{C}}$ The code in the `if` clause has the running time $O(n^2)$ and the code in the `else` clause has $O(n)$. Since the code block has the running time $O(n^2)$ for large integers that is the answer.

2. (a) $\boxed{\text{A}}$ The opposite argument can be made here. Since the code block has the running time $O(n)$ for large integers that is the answer.

Greedy

3. (a) $\boxed{\text{A}}$ Buildstacks, you can solve this by using the greedy condition "always take the coin with the highest value" and then filling the stacks, starting with S_1, then S_2 and so on, until you run out of coins.

(b) $\boxed{\text{A}}$ See **(a)**.

(c) $\boxed{\text{B}}$ See **(a)**.

(d)

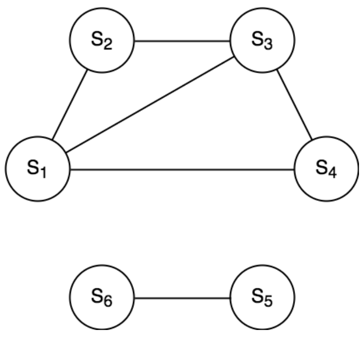
S_1	$150, 125, 125$
S_2	$100, 100, 75$
S_3	$50, 50, 38$
S_4	25

(e) $\boxed{\text{A}}$ The only thing the algorithm has to do is sort the coins in descending order and then you have all your stacks lined up one after the other. This means the algorithm depends on the number of coins, n, and not the stack size, m.

Graph connectivity

4. (a) $\boxed{\text{D}}$ Rotten, let each stack be a node and if two stacks have a common type of coin there is an edge between them. If all nodes are connected the disease will spread to all stack regardless of what coin you choose, otherwise not.

(b) The example instance is non-solvable since all nodes aren't connected.



(c) $\boxed{\text{B}}$ See **(a)**.

(d) $\boxed{\text{C}}$ The first one isn't correct since the shortest path isn't relevant to this problem. The second one isn't correct either since the edges aren't weighted, and an unweighted graph can't have negative cycles which means the last one is false as well. The third one is correct because both Prim's algorithm and BFS (Breadth-first search) finds a minimal spanning tree (if the number of nodes in the tree equals m there is a solution). Prim's running time is $O(m + n \log n)$ and BFS's is $O(m + n)$ where m is the number of nodes and n is the number of edges, BFS is therefore faster.

(e) $\boxed{\text{A}}$ If BFS is used the running time is $O(n + m)$.

Dynamic programming

5. (a) $\boxed{\text{B}}$ Neighbours, you can iterate, i, from left to right where the maximum amount you can take up to the current coin is dependent only on the coins to the left and what the maximum amount at each of those coins is.

(b) $\boxed{\text{G}}$ The optimal solution is the maximum of either the optimal solution for the coin to the left or the current coin value plus the optimal solution for the coin two place to the left.

The example has the following solution (the image is missing a 5 between the 25 and 1, see input/output box at end):

$$\begin{aligned}
\text{OPT}(1) &= \max\{\text{OPT}(0), \text{OPT}(-1) + 100\} = \max\{0, 100\} = 100\\
\text{OPT}(2) &= \max\{\text{OPT}(1), \text{OPT}(0) + 50\} = \max\{100, 50\} = 100\\
\text{OPT}(3) &= \max\{\text{OPT}(2), \text{OPT}(1) + 100\} = \max\{100, 200\} = 200\\
\text{OPT}(4) &= \max\{\text{OPT}(3), \text{OPT}(2) + 50\} = \max\{200, 150\} = 200\\
\text{OPT}(5) &= \max\{\text{OPT}(4), \text{OPT}(3) + 150\} = \max\{200, 350\} = 350\\
\text{OPT}(6) &= \max\{\text{OPT}(5), \text{OPT}(4) + 125\} = \max\{350, 325\} = 350\\
\text{OPT}(7) &= \max\{\text{OPT}(6), \text{OPT}(5) + 25\} = \max\{350, 375\} = 375\\
\text{OPT}(8) &= \max\{\text{OPT}(7), \text{OPT}(6) + 5\} = \max\{375, 355\} = 375\\
\text{OPT}(9) &= \max\{\text{OPT}(8), \text{OPT}(7) + 1\} = \max\{375, 376\} = 376\\
\text{OPT}(10) &= \max\{\text{OPT}(9), \text{OPT}(8) + 75\} = \max\{376, 450\} = 450
\end{aligned}$$

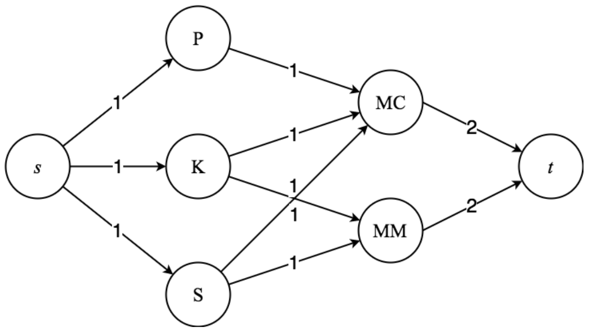
This is the maximum sum while the output should be which coins to pick, you can easily sovle that by saving the path the algorithm takes.

(c) $\boxed{\text{A}}$ As seen in **(b)**, you need to traverse each coin exactly once.

Network flow

6. (a) $\boxed{\text{E}}$ Strike, the children shall "flow" through the hosts who have a capacity, attributes typical to Network Flow.

(b) The first row is all the chilren and the second is all the hosts. The edges between children and hosts are defined by the relationships. the weight between each host and the sink should be the hosts capacity and all other weights should be 1.



(c) $\boxed{\text{D}}$ the number of nodes is $n + m + 2$, see **(b)**.

Computational compexity

7. (a) $\boxed{\text{C}}$ Takestacks, let each stack be a node and if two stacks have a common type of coin there is an edge between them. Takestacks then becomes a maximum independent set problem, which is NP-hard.

(b) $\boxed{\text{C}}$ It is a maximum independent set problem.

(c) $\boxed{\text{B}}$ "The easiest way to see that is to take the following NP-hard problem, Independent set, and prove that Takestacks is at least as hard as Independent set (Independent set $\leq_p$ Takestacks)"

(d) Given an instance to problem Independent Set, I, consisting of $G(V, E)$. We construct an instance of Takestacks with $m = |V|$ stacks as follows:

For each vertex $v_i \in V, i = 1, 2, \ldots, |V|$, we construct a stack S_i. For each edge $\{v_i, v_j\} \in E$, we insert a coin of the same unique arbitrary value in stacks S_i and S_j. We now have coin values $c_1, c_2, \ldots, c_n, n = |E|$. This reduction runs in linear time.

If $T(S_1, S_2, \ldots, S_m, c_1, c_2, \ldots, c_n)$ is a solution to Takestacks, T is also a solution to the Independent set problem I. Hence Independent Set $\leq_p$ Takestacks

Exam 120523

Analysis of algorithms

1. (a) $\boxed{\text{A}}$ For each factor you analyze which term is dominant for large n's (n^3 is more dominant than n^2).

(b) $\boxed{\text{B}}$ $O(n^3 \log n) \neq O(n^3/\log n)$

(c) $\boxed{\text{B}}$ $O(n^3 \log n) \neq O(n^3)$

2. (a) $\boxed{\text{C}}$ The inner most for-loop runs in constant time (can be written `print "12345678910"`) which means the code blocks order is $O(n^2)$.

3. (a) $\boxed{\text{A}}$ Every time **remove(S)** gets called $|S| = 1$ since each element that gets inserted is removed directly afterwards. This means both **remove(S)** and **insert(i, S)** will run in $O(1)$ here. The running time therefore is $O(n)$.

(b) $\boxed{\text{B}}$ The size of S linearly goes from 1 to $n/2$. The upper bound can clearly be set to $O(n \log n)$ (if $|S| = n$ is true for every iteration and **remove(S)** gets called every iteration). To set a lower bound we look at the last half of the sum (from $i = n/2$ to n) and lower the bound for each summand to $\log(n/2) = \log(n) - c$ which means the sum is at least $\frac{1}{2} \cdot \frac{1}{2} \cdot n \cdot (\log(n) - c)$. That's $\Omega(n \log n)$.

4. (a) $\boxed{\text{A}}$ This is the correct option since this should represent the running time. Even though the algorithm takes the maximum of the two you need to calculate both which takes $T(n-1) + T(n-2)$ time. Checking $n > 1$, calculate $\max(x, y)$ and returning 1 takes constant time ($O(1)$).

5. (a) $\boxed{\text{A}}$ Since L and R are the only sets in $B_{l,r}$ the total number of vertices are $l + r$.

(b) $\boxed{\text{C}}$ Each node in L will have r number of edges which means the total is lr.

(c) $\boxed{\text{B}}$ No two nodes in L will have an edge nor two nodes in R. All nodes in L have edges with all nodes in R. This means the answer is either l or r.

(d) $\boxed{\text{C}}$ If $l + r \leq 3$ either $r = 1$ or $l = 1$ since L and R are nonempty sets. The solo node can then be seen as an root to a tree the other nodes as children.

Greedy

6. (a) Consider the following input:

```
aa
ab
ba
ca *
```

The greedy algorithm will output `aa -> ab`, while the optimal solution is `aa -> ba -> ca`.

Graph connectivity

7. (a) $\boxed{\text{A}}$ Ladder, you can solve this with BFS (Breadth-first search) which is a graph connectivity algorithm.

(b) $\boxed{\text{A}}$ BFS is an algorithm for finding shortest path between to vertices (words) or realizing there is no path. DFS nor MST doesn't necessarily find the shortest path and Topological sorting only works on directed acyclic graphs (which this isn't).

(c) Use the defintions from page 2 and let $m = |E|$. Start BFS on one of the vertices (words) with a star and continue until you find the other or you run out of edges (neighbour relations). The example instance is drawn on page 2. The running time for BFS is $O(n + m)$.

Dynamic programming

8. (a) $\boxed{\text{D}}$ Alphabetic, you can iterate through all words and calculate the local optimal based on all alphabetically smaller neighbours.

(b) Let the function star(w) return 1 if w has a star else 0.

$$\text{OPT}(x) = \max_{y<x}\{\text{OPT}(y) : \{y, x\} \in E\} + \text{star}(x)$$

(c) $\boxed{\text{A}}$ You only need to calulate the optimal of each word once, and that is constants if you start with the smallest word and iterate through the rest.

(d) $\boxed{\text{A}}$ You only need to save the optimal of each word once which makes it linear.

Network flow

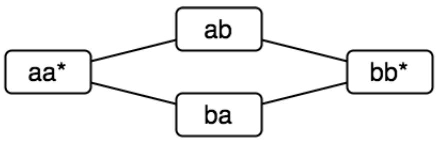
9. (a) $\boxed{\text{B}}$ Cycle, you can see this as an vertex-disjoint paths problem where you need to find at least 2 paths.

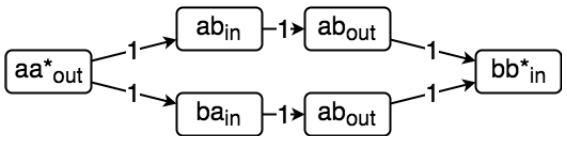
(b) Use the graph definition from page 2. Let s represent the source and t represent the sink.

1. Replace each undirected edge, $\{u, v\}$, with the directed edges (u, v) and (v, u).
2. Replace each vertex, v, with v_{in} and v_{out}, add the edge (v_{in}, v_{out}) and change all edges, (u, v), to (u_{out}, v_{in}).
3. Remove s_{in} and t_{out} and all connected edges since they aren't useful.
4. Set the weight of all edges to 1.

The maximum flow problem represents the maximum vertex-disjoint paths problem and there is a solution to the original problem if maximum flow ≥ 2.

Let m' and m denote the number of edges in the resulting graph and the original respectively, this means $m' = n+2m$. Use the Ford-Fulkerson algorithm which has the running time $O(fm')$ where f is the number of found paths, but since we only need 2 we can set $f = 2$. Which gives the running time $O(2(n + 2m)) = O(n + m)$


ab
aa*
bb*
ba


ab$_{in}$
ab$_{out}$
aa*$_{out}$
bb*$_{in}$
ba$_{in}$
ab$_{out}$
1
1
1
1
1
1