Comp 251: Practice problems for the Final

Instructor: Jérôme Waldispühl

Hashing

1. Suppose that you use a hash table and a hash function implementing the division method. The following keys are inserted: 5, 28, 19, 15, 20, 33, 12, 17, 10 and m = 9 (for simplicity, here we do not distinguish a key from its hashcode, so we assume h(key)=key). In which slots do collisions occur?

> **Solution:** The keys 5, 28, 19, 15, 20, 33, 12, 17, 10 map to slots 5, 1, **1**, 6, 2, **6**, 3, 8, **1**. Collisions are shown in bolded fonts.

2. Now suppose you use open addressing with linear probing and the same keys as above are inserted. More collisions occur than in the previous question. Where do the collisions occur and where do the keys end up?

> **Solution:** The key 19 collides with key 28 at slot 1 and 19 goes into slot 2. Key 20 collides with key 19 at slot 2 and needs to go to slot 3. Key 33 collides with key 15 at slot 6 and needs to go to slot 7. Key 12 collides with key 20 at slot 3 and needs to go to slot 4. Key 17 goes into position 8. Key 10 collides with key 28 at slot 1, and then collides with 19 at slot 2, etc until it finally finds an open slot at position 0. The final positions of the keys are [10, 28, 19, 20, 12, 5, 15, 33, 17].

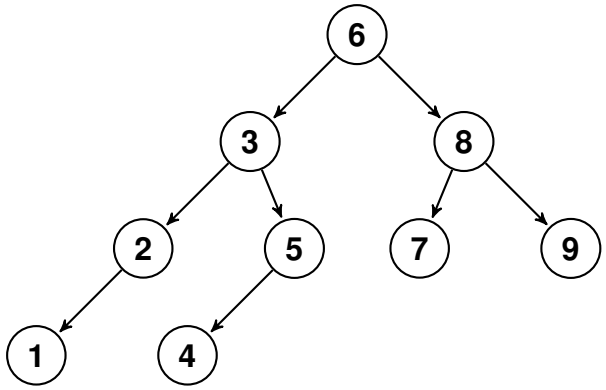
Balanced binary search trees

3. What is the maximum number of nodes in an AVL tree of a given height h?

> **Solution:** This would be a complete binary tree of height h. Such a tree has $2^{h+1} - 1$ nodes (See COMP 250!).

4. Starting with an empty tree, construct an AVL tree by inserting the following keys in the order given: 2, 3,5, 6, 9, 8, 7, 4, 1. If an insertion causes the tree to become unbalanced, then perform the necessary rotations to maintain the balance. State where the rotations were done.

Solution: Add(2), add(3), add(5), rotateLeft(2), add(6), add(9), rotateLeft(5) and note that this solves the imbalance at 3 also, add(8), rotateLeft(3), add(7), rotateRight(9), add(4), add(1). The final AVL tree is:



For your practice, we recommend you to draw all intermediate steps.

Heaps

5. Suppose you have a heap which supports the usual add() and removeMin() operations, but also supports a changePriority (name, new Priority) operation. How could you combine these operations to define a remove(name) operation?

Solution: Check the current minimum and find out what its priority is. (You can remove the min, check its value, and then add it back in.) Then use the changePriority() method to reduce the priority of the element that you wish to remove, such that its new priority is less than that of the current minimum. Then, remove the (new) minimum.

6. Suppose you used an ordered array to implement a priority queue. Give the O( ) time for the operations removeMin(), add(element, key), findMin() take?

Solution: If you order from small to large then removeMin() would be O(n). It would be better would be to order from large to small so that removeMin would be O(1). Adding an arbitrary element would be O(n) since you would have to shift all the elements in the worst case. findMin would be O(1) since the array is ordered.

Disjoint sets

7. Consider the set of all trees of height h that can be constructed by a sequence of "union-by-height" operations. How many such trees are there?

Solution: It was a trick question. There are infinitely many of these trees, since there is no constraint given on the number of nodes and "union-by-height" trees (trees built by union-by-height operations) have no constraint on the number of children of any node.

8. Consider a tree formed by union-by-height operations, without path compression. Suppose the tree has n nodes and the tree is of height h. Show that n is greater than or equal to 2^h. (Note that the tree need not be a binary tree, and so we cannot just apply properties of binary trees to this problem. Indeed, for binary trees of height h, we can only say that the number of nodes at most $2^{h+1} - 1$, which is looser than the bound stated in the question.)

Solution: Here is a proof by induction on the tree height k. The base case $k = 0$ is easy, since a tree of height 0 always has just $1 = 2^0$ node (the root). Suppose the claim is true for $h = k$. Now consider a union-by-height tree of height $k + 1$. There must have been a union that brought two trees together and increased the height of one of them from k to $k + 1$. Let those two trees (at the time of that union) be T1 and T2. We know that both T1 and T2 were of height k before the union. [Why? If one of them were of height less than k, then union-by-height would have changed the root of that shorter one to make it point to the root of the taller one, and the height of the unioned tree would still be k. But its not the unioned tree is of height k+1.]
Now we can apply the induction hypothesis: the trees T1 and T2 each have at least 2^k nodes. Thus, the unioned tree has at least $2^k + 2^k = 2^{k+1}$ nodes.

Minimum spanning-trees

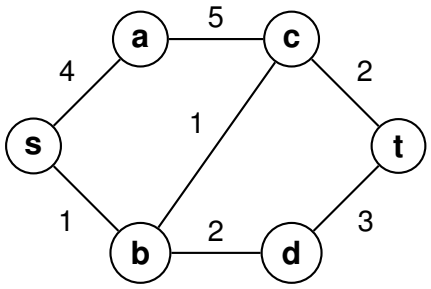
9. Prove that for any weighted undirected graph such that the weights are distinct (no two edges have the same weight), the minimal spanning tree is unique.

Solution: Suppose there are two MSTs, call them T1 and T2. Let the edges of T1 be $\{e_{i_1}, e_{i_2}, ...e_{i_{n-1}}\}$ and the edges of T2 be $\{e_{k_1}, e_{k_2}, ...e_{k_{n-1}}\}$. If the trees are different, then these sets of edges must be different.
So, let e* be the smallest cost edge in T1 that is not in T2. Deleting that edge from T1 would disconnect T1 into two components. Since T1 chose that edge, it must be the smallest cost crossing edge for those two components. But by the cut property, that edge must therefore belong to every minimum spanning tree. Thus it must belong to T2 as well. But this contradicts the definition of e*.

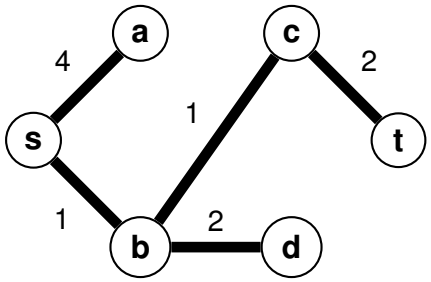
10. If a connected undirected graph has n vertices, then any spanning tree has n-1 edges.

Solution: Consider the edges in a spanning tree T and consider a graph with no edges, but all n vertices. Now add the edges of the spanning tree one by one. Each edge is a crossing edge between two connected components and adding the edge reduces the number of connected components by 1. Since adding all the edges of T (one by one) reduces the number of connected components from n down to 1, it must be that T has n-1 edges.

11. Consider the flow graph below. Apply the Kruskal algorithm to calculate the minimum spanning tree.



Solution: The minimum spanning tree is:



Single-source shortest paths

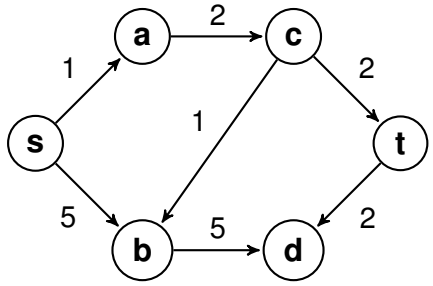
12. In breadth first search, each vertex has a "visited" field which is set to true before the vertex is put in the queue. What happens if BFS instead sets the visited field to true when the vertex is removed from the queue? Does the algorithm still work? Does it run just as fast? What if we want to find the shortest path between every pair of vertices in the graph ?

Solution: The algorithm still works. However, multiple copies of the vertex can be put the queue. The maximum number of copies is the in-degree of the vertex. The size of the queue grows as O(|E|) rather than O(|V|).

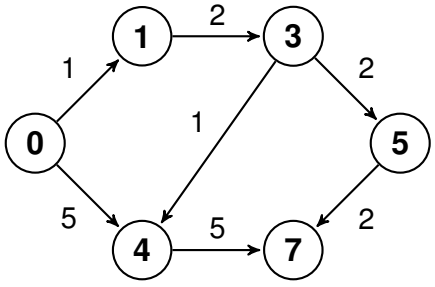
13. Dijkstra's algorithm assumes the edges have non-negative weights. (Where does this come up in the proof of correctness?) But suppose we have a graph with some negative weights, and let edge e be such that cost(e) is the smallest (most negative). Consider a new graph in which we add cost(e) to all the edge weights, thus making all weights in the new graph non-negative. Now the conditions hold for Dijkstra to find the shortest paths, so we could now run Dijkstra. Is this a valid way to solve for shortest paths in the case that some edges have negative weights? Justify your answer.

Solution: No. For any path in the original graph, the distance of that same path P in the new graph will be greater by cost (e) multiplied by the number of edges in the path P, since we incremented each edges cost by cost(e). But for two paths with a different number of edges, the totals for the extra costs that we have added will be different. So there is no reason why you should expect to get the same solutions in the two cases. [Note that by ?same solutions? here, I don?t just mean the same distances of the shortest paths; I mean the paths themselves!] For example, take the graph with three edges cost(u,v) = -2, cost(v,w) = 2, cost(u, w)=1. The shortest path to w is (u, v, w). But adding 2 to the cost of each edge and then running Dijkstra would give the shortest path as (u, w).

14. Consider the flow graph below. Apply the Dijkstra's algorithm to calculate the shortest paths from s.



Solution: The solution is:



Where the shortest path estimates are stored inside the vertices.

Bipartite graphs

15. Given the preferences shown here, use the Gale-Shapley algorithm to find a stable matching.

A	A's preferences	B	B's preferences
α_1	$\beta_1, \beta_2, \beta_3$	β_1	$\alpha_3, \alpha_1, \alpha_2$
α_2	$\beta_1, \beta_2, \beta_3$	β_2	$\alpha_1, \alpha_3, \alpha_2$
α_3	$\beta_1, \beta_2, \beta_3$	β_3	$\alpha_3, \alpha_2, \alpha_1$

Solution: The answer is $(\alpha_1, \beta_2), (\alpha_2, \beta_3), (\alpha_3, \beta_1)$.

16. Consider an instance of the stable matching problem in which, for all $\alpha \in A$, α's first choice is β if and only if β's first choice is α. In this case, there is only one stable matching. Why?

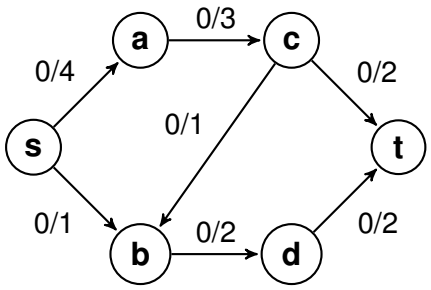
Solution: By definition, a matching is unstable if there exists an $\alpha \in A$ and a $\beta \in B$ that prefer each other over their present partners. Take any matching for which some $\alpha \in A$ doesn't get its first choice. Let the first choice of α be β. Then β is not getting its first choice either, since β's first choice is α. But then this matching is not stable. Thus, for any stable matching, every α gets its first choice, and so every β (by the constraints given in this question) gets its first choice too.

Flow networks

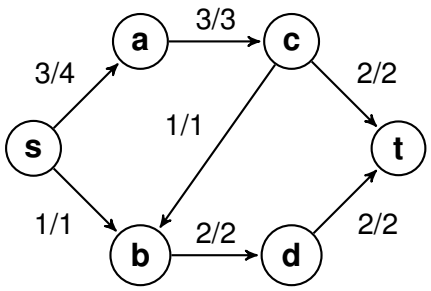
17. Suppose the capacities in a network flow are not integers. How would this change the O( ) runtime of the Ford Fulkerson algorithm? Does the algorithm terminate?

Solution: The stated runtime of Ford-Fulkerson is O(C m) as explained in the lecture, where C is an integer namely the sum of integer capacities of edges out of s. If the capacities are not integers, then this O( ) no longer makes sense because O( ) always assumes we are talking about integers i.e. number of operations. More concretely, if we applied the Ford-Fulkerson algorithm to a flow network having non-integer capacities, we would still find that the flow increases in every pass through the main loop. However, the amount by which the flow increases might be a very small number and this number might get smaller and smaller indefinitely. There is no reason why the while loop would ever terminate. (Here we ignore the precision limits of representing floating point numbers on real computers, which you will learn about in COMP 273 or ECSE 221, if you haven't already)

18. Consider the flow graph below. Apply the Ford-Fulkerson algorithm to calculate the maximum flow.



Solution: The final solution is:



Dynamic programming

19. The Coin Row Problem: Suppose you have a row of coins with values that are positive integers $c_1, \cdots, c_n$. These values might not be distinct. Your task is to pick up coins have as much total value as possible, subject to the constraint that you don't ever pick up two coins that lie beside each other. How would you solve this using dynamic programming?
Solve the problem for coins with values c_1 to c_6 as follows: $(5, 1, 2, 10, 6, 2)$.

Solution: The idea for the recurrence is as follows. Start with the last coin. You either pick it up or you don't. If you pick it up, then you cannot pick up the second to last coin but you are free to pick up any others. If you don't pick up the last coin, then you are free to pick up any of the others (subject to the problem's constraints). The recurrence that describes this is $f(n) = \max(c_n + f(n-2), f(n-1))$, with base case $f(1) = c_1$, $f(0) = 0$.
You can solve this either iteratively or recursively using dynamic programming. For the example given, the maximum value is 17 and uses coins $\{c_1 = 5, c_4 = 10, c_6 = 2\}$.

20. The Coin Change Problem: Suppose we have m types of coins with values $c_1 < c_2 < \cdots c_m$ (e.g. in the case of pennies, nickels, dimes, ... we would have $c_1 = 1$, $c_2 = 5$, $c_3 = 10$, $\cdots$). Let $f(n)$ be the minimum number of coins whose values add up to exactly n. Write a recurrence for $f(n)$ in terms of the values of the coins. You may use as many of each type of

coin as you wish.
As an example, suppose the coin values c_1, c_2, and c_3 are 1, 3, 4. Solve the problem for $n = 6$ using dynamic programming.

Solution: To write the recurrence, we consider the case that a coin of type j was used in the optimal solution. Then, if a coin of type j was used, we need to solve the subproblem of finding the minimum number of coins whose values sum up to $n - c_j$. Thus, $f(n) = min_{j \in \{1 \cdots m\}} 1 + f(n - c_j)$, $f(0) = 0$.
For the example given, the solution is two coins of value 3 each.

21. What is the optimal substructure of the Neddleman-Wunch algorithm (i.e. optimal pairwise sequence alignment)?

Solution: Recall the definition of the optimal substructure: "The optimal solution for one problem instance is formed from optimal solutions for smaller problems".
The optimal substructure is usually materialized in the dynamic programming equations. In the Neddleman-Wunch algorithm these equations are:

$$NW(i,j) = \max \begin{cases} NW(i-1,j) + \delta(a_i, -) & \text{(deletion)} \\ NW(i-1,j-1) + \delta(a_i, b_j) & \text{(match or mismatch)} \\ NW(i,j-1) + \delta(-, b_j) & \text{(insertion)} \end{cases}$$

Where a and b are the sequences to align, δ the edit cost function, and $NW(i,j)$ is the dynamic programming table that stores the score of the optimal alignment of the prefix $a_{1 \cdots i}$ and $b_{1 \cdots j}$.

The last column of any alignment is either a deletion, match/mismatch or insertion. Then, the optimal alignment of two string $a_{1 \cdots i}$ and $b_{1 \cdots j}$ is made from the concatenation of (i) an optimal alignment of $a_{1 \cdots i-1}$ and $b_{1 \cdots j}$ if the alignment ends with a deletion, (ii) an optimal alignment of $a_{1 \cdots i-1}$ and $b_{1 \cdots j-1}$ if the alignment ends with a match or mismatch, or (iii) an optimal alignment of $a_{1 \cdots i}$ and $b_{1 \cdots j-1}$ if the alignment ends with a insertion. The optimal alignment is this made from the best option among those three.

We note that the sub-problems are strictly smaller since the length of at least one of the two sequences has been reduce by 1.

Divide-and-Conquer

22. In Karatsuba multiplication, when you do the multiplication $(x_1 + x_0) \cdot (y_1 + y_0)$, the two values you are multiplying might be $n/2 + 1$ digits each, rather than $n/2$ digits, since the addition might have led to a carry e.g. $53 + 52 = 105$. Does this create a problem for the argument that the recurrence is $t(n) = 3t(n/2) + c_n$?

Solution: The recurrence would need to be written $t(n) = 2t(n/2) + t(n/2+1) + c \cdot n$. We know that $t(n)$ is $O(n^2)$ and we are trying to prove a better bound, but the fact that it is $O(n^2)$ allows us to say that $t(n) \leq c \cdot n^2$ for some constant c and for sufficiently large n (Recall the formal definition from COMP 250). Thus, $t(n/2+1) \leq c \cdot (n/2+1)^2 = c \cdot (n/2)^2 + c \cdot n/2 + c$ for some constant c and for sufficiently large n. Thus, $t(n/2+1) = t(n/2) + O(n)$, that is, $t(n/2+1)$ is bigger than $t(n/2)$ but only by an amount that grows linearly with n. Thus, we can write:

$$t(n) = 2t(n/2) + t(n/2+1) + cn = 3t(n/2) + O(n).$$

In case the above went too fast, here is the basic idea: there is no problem that the Karatsuba trick requires multiplying two numbers of size $n/2+1$ instead of $n/2$. The reason it doesn't matter is that the extra work you need to do is bounded by some O(n) term. You are already doing a bunch of $O(n)$ work at each node (of the call tree). So one extra $O(n)$ term won't make any difference.

23. Apply the master method to determine the asymptotic behavior of the function $T(n)$.

    1. $T(n) = 2 \cdot T(n/4) + n^{0.51}$
    2. $T(n) = 0.5 \cdot T(n/2) + 1/n$
    3. $T(n) = 64 \cdot T(n/8) - n^2 \cdot logn$
    4. $T(n) = \sqrt{2} \cdot T(n/2) + \log n$
    5. $T(n) = 6 \cdot T(n/3) + n^2 \cdot \log n$
    6. $T(n) = 3 \cdot T(n/3) + n/2$

Solution:

1. Case 3: $T(n) = \Theta(n^{0.51})$
2. Does not apply: $a < 1$
3. Does not apply: $f(n)$ not positive
4. Case 1: $T(n) = \Theta(\sqrt{n})$
5. Case 3: $T(n) = \Theta(n^2 \cdot \log n)$
6. Case 2: $T(n) = \Theta(n \cdot \log n)$

24. Write a recurrence that describes its worst-case running time of the quicksort algorithm.

Solution: $T(n) = T(n-1) + T(0) + \Theta(n)$

Amortized analysis

25. Suppose we perform a sequence of stack operations on a stack whose size never exceeds k. After every k operations, we make a copy of the entire stack for backup purposes. Show that the cost of n stack operations, including copying the stack, is $O(n)$ by assigning suitable amortized costs to the various stack operations.

Solution: Charge \$2 for each PUSH and POP operation and \$0 for each COPY. When we call PUSH, we use \$1 to pay for the operation, and we store the other \$1 on the item pushed. When we call POP, we again use \$1 to pay for the operation, and we store the other \$1 in the stack itself. Because the stack size never exceeds k, the actual cost of a COPY operation is at most \$k, which is paid by the \$k found in the items in the stack and the stack itself. Since there are k PUSH and POP operations between two consecutive COPY operations, there are \$k of credit stored, either on individual items (from PUSH operations) or in the stack itself (from POP operations) by the time a COPY occurs. Since the amortized cost of each operation is $O(1)$ and the amount of credit never goes negative, the total cost of n operations is $O(n)$.

26. Suppose we perform a sequence of n operations on a data structure in which the i^{th} operation costs i if i is an exact power of 2, and 1 otherwise. Use aggregate analysis or accounting method to determine the amortized cost per operation.

Solution:

Aggregate analysis:

Let c_i be the cost of i^{th} operation.

$$c_i = \begin{cases} i & \text{if i is an exact power of 2} \\ 1 & otherwise \end{cases}$$

n operations cost: $\sum_{i=1}^{n} c_i \leq n + \sum_{j=0}^{\log n} 2^j = n + (2n-1) < 3n$. (Note: We ignoring floor in upper bound of $\sum 2^j$).

Thus the Average cost of operation = Total cost < 3 number of operations. And by aggregate analysis, the amortized cost per operation = O(1).

Accounting method:

Charge each operation \$3 (amortized cost $\hat{c}_i$).

- If i is not an exact power of 2, pay \$1, and store \$2 as credit.
- If i is an exact power of 2, pay \$i, using stored credit.

Operation	Cost	Actual cost	Credit remaining
1	3	1	2
2	3	2	3
3	3	1	5
4	3	4	4
5	3	1	6
6	3	1	8
7	3	1	10
8	3	8	5
9	3	1	7
10	3	1	9
⋮	⋮	⋮	⋮

Since the amortized cost is \$3 per operation, we have $\sum_{i=1}^{n} \hat{c}_i = 3n$. Moreover, from aggregate analysis, we know that $\sum_{i=1}^{n} c_i < 3n$. Thus $\sum_{i=1}^{n} \hat{c}_i \geq \sum_{i=1}^{n} c_i \Rightarrow$ credit never goes negative.

Since the amortized cost of each operation is $O(1)$, and the amount of credit never goes negative, the total cost of n operations is $O(n)$.