G22.3520: Honors Analysis of Algorithms

Problem Set 4+5

Solutions

Problem 1

a) Perform binary search on each full array until the element is found. The worst-case run time is to do binary-search on each of the $\log(n)$ arrays which takes at most $\log^2(n)$ time.

b)The insertion is quite similar with the incrementing binary counter. When we insert an element, the binary representation of n changes exactly the way the binary counter does. For all t digits on the right that change from 1 to 0, we merge the t arrays (pairwise) to get an array of size 2^{t+1} which we associate with the $t+1$st digit which now becomes 1. The cost of this insertion is $O(2^t)$ (recall: merging two sorted arrays of size m takes $O(m)$ time). The worst case occurs when $n = 2^{t+1}$ after the insertion (a 1 followed by all 0s in the binary representation). The cost in this case is $O(n)$.

For the amortized cost we define the following *negative* potential function: $\Phi(D) = -\sum_{i=1}^{k} i|A_i|$ where $k = \log(n)$ and $|A_i|$ is the size of the array corresponding to bit i in the binary representation of n (i.e. $|A_i| = 0$ or $|A_i| = 2^{i-1}$). For any operation j, the true cost of the operation is $c_j = O(2^t)$ and the change in potential is

$$\begin{aligned}
\Phi(D_j) - \Phi(D_{j-1}) &= -\sum_{i=1}^{k} i|A'_i| - a(-\sum_{i=1}^{k} i|A_i|) = a\sum_{i=1}^{k} i(|A_i| - |A'_i|) \\
&= \sum_{i=1}^{t} i2^{i-1} - (t+1)2^t \\
&\leq t(\sum_{i=1}^{t} 2^{i-1}) - (t+1)2^t \\
&\leq -2^t
\end{aligned}$$

where $|A'_i|$ are the new array lengths and $|A_i|$ are the old array lengths and t is the number of 1s to the right that change to 0. Therefore the accounted cost of an operation is $\hat{c}_j = c_j + \Phi(D_j) - \Phi(D_{j-1}) = 0$ (we need to scale Φ by some appropriate *constant* to make this work). Recall that the total cost of any n operations is:

$$\sum_{i=1}^{n} c_i \leq \sum_{i=1}^{n} \hat{c}_i + \Phi(D_0) - \Phi(D_n) \leq -\Phi(D_n) \leq \sum_{i=1}^{k} i|A_i| \leq n\log(n).$$

Therefore the amortized cost per operation is $\log(n)$.

Alternatively, we can use the aggregate method to show an $O(\log n)$ amortized upper bound. As we described above, any insertion takes time 2^{t+1} if we have t consecutive 1s in the least significant bits of the binary representation of n. This happens though at most $\frac{n}{2^{t+1}}$ times for a sequence of n insertions. This means that the total cost of n operations is bounded by $\sum_{t=0}^{k} \frac{n}{2^{t+1}} 2^{t+1}$ which is $O(nk) = O(n\log n)$. If we divide with the total number of n operations we get $O(\log n)$ amortized cost per insert operation.

c) In order to implement the delete operation, we first search for the element to be deleted following the steps that we showed above. Then, if we find this element x in array A_i we remove it. Let t be the right-most bit which is a 1 in the binary representation of n. Then take an arbitrary element y from A_t and insert it into A_i in the appropriate position (so the number of elements in A_i will not change). Now take the array A_t and split it into $t-1$ sorted arrays of sizes $1, 2, 4, 2^{t-1}$ respectively. These arrays correspond to the $t-1$ right-most bits in the representation of $n-1$, which are now 1. This operation may take $O(n)$ time - for example if n is a power of 2.

Problem 2

a) Perform in-order traversal of the tree rooted at x to get a sorted array of elements. Then we build a balanced tree out of this array as follows: choose the median element and make it a root. Then recursively

build a left subtree from the portion of the array that's lower than the median, and a right subtree from the portion of the array which is greater than the median.
b) If $T(n)$ is the amount of time that it needs in order to search a n-node α-balanced binary search tree, then we have $T(n) = 1 + T(\alpha n)$. If α is a constant then $T(n) = O(\log n)$ (for example, by the master theorem).
c)Since c must be a positive constant and given that $\Delta(x)$ is never negative (absolute value), we conclude that the potential can't take negative values. By the definition of 1/2-balanced trees we know that for every vertex x we have $size[right[x]] \leq \frac{1}{2}size[x]$ and $size[left[x]] \leq \frac{1}{2}size[x]$. If there existed a vertex x for which $\Delta(x) \geq 2$, then one of the two children, say $right[x]$ would have at least two elements more than $left[x]$. This would mean that $2size[right[x]] \geq size[x] + 1$ which in turn means that $size[right[x]] \geq \frac{1}{2}size[x] + \frac{1}{2}$ which is a contradiction. Therefore $\Delta(x) < 2$ and $\Phi(T) = 0$.
d)The amortized cost is $\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1})$ and we want it to be $O(1)$. Since after each rebuild we have $\Phi(D_i) = 0$ and the rebuilding costs m, we want $O(1) = m - \Phi(D_{i-1}) \Rightarrow \Phi(D_{i-1}) \geq m$. We want to find the minimum value of the potential that could cause the need for rebuilding. This would mean that one of the subtrees, say the right one, would violate the α-balanced constraint and therefore $size[right[x]] > \alpha m$. We also know that $size[right[x]] + size[left[x]] = m - 1$ so $size[left[x]] < m - 1 - \alpha m$. In this case, we have $\Delta(x) = size[right[x]] - size[left[x]] > \alpha m - (m - 1 - \alpha m)$ and therefore $\Delta(x) > 2\alpha m - m + 1$. Summing up all the above we want $m \leq c(\alpha m - m + 1) \Rightarrow c \geq \frac{m}{\alpha m - m + 1} \geq \frac{1}{2\alpha}$.
e)Now that we have shown how to rebuild a subtree in $O(1)$ amortized time, we only need to worry about the actual cost of an insertion or deletion and about the increase of the potential function. It is easy to see that the actual cost of these operations is $O(\log n)$ just like we showed the same upper bound for searching above. What remains to be shown is an upper bound for the increase of the potential function after some insertion or deletion. Such an operation only affects the potential function of the vertices that belong to the traversed path from the root. To be more specific, the worst case is when the new element is added to the biggest subtree or deleted from the smallest subtree and therefore leads to an increase of $\Delta(x)$ by one. Since there are only $O(\log n)$ vertices that get affected though, this means that the potential function cannot increase by more than $O(\log n)$ and this proves that the amortized time complexity of both operations is $O(\log n)$.

Problem 3

1)We start from some arbitrary node and perform depth first search[1]. At each step of this process, we mark the node that we have just visited and then we check all the "unused" edges from its adjacency list. If one of these edges (u, v) leads to a node v which is already marked, this can only mean one thing. Since this edge was not already used although node v was marked, then node v is a predecessor of u in the DFS tree. This makes it very easy to output the cycle by just traversing the path of the predecessors of u until we reach v. It is obvious that if a cycle exists within the graph, then a DFS search would inevitably visit the same vertex twice and since we check every vertex and every edge once, the running time is $O(m + n)$.
2)They can just assign each butterfly to a vertex in a graph and connect the corresponding vertices for each judgment either with a blue edge if they are believed to be the same or with a red edge if they are believed to be different. We start by marking some arbitrary vertex with the letter A, implying that the corresponding butterfly belongs to this species. We then perform breadth first search, marking all adjacent vertices with different letters from $\{A, B\}$ if they are connected with red edges and with the same letters otherwise. If at some point during the execution of the algorithm we try to label some vertex with the opposite letter from the one that it's already labelled with, the algorithm rejects the input. If this algorithm terminates successfully, it returns a consistent classification of the specimens based on the consistent judgements.

Problem 4

We begin by creating a directed graph which has two vertices for each person P_i, one that corresponds to his birth: "P_i born" and one to his death: "P_i dead". We add a directed edge from "P_i born' to "P_i dead"

[1]We follow the exact same procedure for every connected component

implying chronological order. For each fact of type "P_i died before P_j was born", we as a directed edge from "P_i dead" to "P_j born". For each fact "the lives of P_i and P_j overlap" we add one directed edge from "P_i born" to "P_j dead"and another edge from "P_j born" to "P_i dead".
We then do a topological sort and quit if we find a cycle. It is clear that, if a cycle exists, then there is an inconsistency (all edges should point to future times). Otherwise, if no cycle exists then a topological sort orders the sequence of births and deaths in a way that is consistent with all facts.

Problem 5
The initial residual graph is the same as the flow network. The Ford-Fulkerson algorithm could choose the (s, a, b, t) augmenting path and push a flow of 1 through it. Then it could do the same thing for the path (s, b, a, t), therefore, after two steps, the residual graph would be similar to the initial one with 999 instead of 1000. This sequence could continue for another 1998 steps until the residual capacities of the four peripheral edges become zero, so the running time would be 2000 steps.

Problem 6
See http://www.cs.nyu.edu/courses/fall06/G22.3520-001/lec20.pdf slides 8-11.

Problem 7
1)Every edge of the matching M must have at least one of its vertices in S. Therefore, every edge in M, can be mapped to some *unique* vertex in S.
2)A fully connected graph of three vertices has matchings of size 1 but its vertex cover must be of size at least two.
3)In order to verify that $(U \backslash A) \cup B \cup C$ is a vertex cover, we notice that all possible edges have at least one of their endpoints in this set. All possible edges must either start from $(U \backslash A)$ and end in W or start from U and end in B or start from A and end in C.
It is easy to see that the size of this set is equal to the number of edges that cross the minimum cut. We have exactly $|U \backslash A|$ edges from s to $(U \backslash A)$, exactly $|B|$ edges from B to t and exactly C edges from A to C. Therefore, this is the capacity of the cut and it is equal to the value of the maximum flow. Since we already know that the value of the maximum flow equals the size of the maximum matching, we conclude that it suffices to just run the Ford-Fulkerson algorithm for the new graph.

Problem 8
We will use a randomized algorithm that colors each vertex uniformly at random. For each edge of the graph, the probability that this edge is satisfied is $\frac{2}{3}$ since the only case when it is not satisfied is if both its vertices are colored with the same color. If $X_{i,j}$ is the indicator variable that is 1 if edge (i, j) is satisfied and 0 otherwise, then $E[X_{i,j}] = \frac{2}{3}$. The expected number of edges that our algorithm satisfies is therefore equal to $E[X] = E[\sum_{(i,j)\in E} X_{i,j}]$ which by linearity of expectation is equal to $\sum_{(i,j)\in E} E[X_{i,j}] = \sum_{(i,j)\in E} \frac{2}{3} = \frac{2}{3}|E|$ which is obviously at least $\frac{2}{3}c^*$.

Problem 9
Let us order the bids $b_1, \ldots, b_n$ in *decreasing* order. If some bid b_k modifies the value of b^*, it must be chosen before any of the $k-1$ bids $b_1, \ldots, b_{k-1}$. There are $k!$ ways to order the first k bids and $(k-1)!$ of these orderings have her bid b_k in the first place. This means that the probability that the bid b_k results in an update of b^* is equal to $\frac{1}{k}$. If X_k is the indicator variable which is 1 if the bid of b_k affects b^* and 0 otherwise, we have that $E[X_k] = \frac{1}{k}$. The expected number of changes of b^* is therefore equal to $E[X] = E[\sum_{i=1}^{n} X_k]$ which by linearity of expectation is equal to $\sum_{i=1}^{n} \frac{1}{k} = \Theta(\log n)$.

Problem 10
a) The probability that some particular machine will still remain idle is equal to $(\frac{k-1}{k})^k$. Therefore, if we let a random variable X_i be an indicator variable for machine i being idle, then $E[X_i] = (\frac{k-1}{k})^k$ and $N(k) = E[\sum X_i] = k(\frac{k-1}{k})^k$ and therefore $\frac{N(k)}{k} = (\frac{k-1}{k})^k$. The limit of this fraction as k goes to infinity is $\frac{1}{e}$.

b)Since the number of jobs that are assigned is equal to the number of machines, the number of jobs that get rejected must be equal to the number of machines that remain idle. Based on the previous subproblem we conclude that the limit is again equal to $\frac{1}{e}$.
c)We first calculate the expected number of rejected jobs for each machine and then add them to find the value of $R_2(k)$. Since the size of the buffer is now 2, in order to have i rejected jobs from some machine, there must have been exactly $i+2$ jobs assigned to it. If we count all possible ways that this can happen we get that the expected number of rejections per machine is equal to

$$\sum_{i=3}^{k}(i-2)\binom{k}{i}(\frac{1}{k})^i(1-\frac{1}{k})^{k-i} = \sum_{i=3}^{k} i\binom{k}{i}(\frac{1}{k})^i(1-\frac{1}{k})^{k-i} - 2\sum_{i=3}^{k}\binom{k}{i}(\frac{1}{k})^i(1-\frac{1}{k})^{k-i}$$

We notice that the first part of the sum resembles the expected value of the binomial distribution and the second part resembles the probability of the binomial distribution. For the specific case that we are working on, both these would be equal to 1 and we only need to deal with the missing values for $i = 0, 1, 2$. After some computations we conclude that the expected number of rejections per machine is equal to $-1 + 2(1-\frac{1}{k})^k + (1-\frac{1}{k})^{k-1}$. By once again using the same arguments as above we conclude that $\lim_{k\to\infty}\frac{R_2(k)}{k} = \frac{3}{e} - 1$.

<u>Problem 11</u>
See http://www.cs.nyu.edu/courses/fall06/G22.3520-001/lec15.pdf for an $O(|V|+|E|)$ algorithm. Here we give a simpler poly-time algorithm.
To begin with, we perform a depth first search from each vertex and we aggregate all the child vertices that we reached to the root after backtracking. This way, in polynomial time, for each vertex we have a list of all the vertices that can be reached from it through directed paths. We we pick some arbitrary vertex $u \in V$. For every vertex v in the reachability list of u, we check if u also exists in the reachability list of v. We put u along with all the vertices of its list which prove to be equivalent with it in a set P_1. We delete all the vertices of P_1 from V and continue by picking an arbitrary next vertex and repeating the above process, defining one new set P_i each time.
Now that we have a decomposition of the graph into strongly directed components, we assume that there exists some directed cycle in the induced graph G'. It is easy to see that if we have an edge in G' between two sets P_i and P_j, this means that there exists a directed path from every vertex of the first set to every vertex of the second set. This obviously means that the existence of a directed cycle in G' implies the existence of a directed cycle in G which passes through vertices that are not equivalent. This can't be true though, since all vertices that belong to a directed cycle must be equivalent with eachother and therefore G' must be acyclic.

Solution: Let A be a context-free language recognized by a PDA M. We will construct a PDA R that recognizes SUFFIX(A). On input v, the PDA R works in two phases. The first phase operates without looking at the input. The PDA non-deterministically generates a symbol $a \in \Sigma$ and simulates (one or more) steps of M until the symbol a is read. The PDA repeats this sequence of moves (as many times as it wishes). It non-deterministically decides when to switch to second phase. In the second phase, the PDA looks at the input v and simulates M on v.

Note that R accepts v if and only if there exists $u \in \Sigma^*$ such that M accepts uv. The string u (and its length!) is "guessed".

Problem 6

Show that the following languages are decidable:

1. $\text{INFINITE}_{\text{DFA}} = \{\langle D \rangle \mid D \text{ is a DFA such that } L(D) \text{ is infinite}\}$.
2. $L = \{\langle R, S \rangle \mid R \text{ and } S \text{ are regular expressions such that } L(R) \subseteq L(S)\}$.

Solution: Using the proof of pumping lemma from Problem 2, it is clear that $L(D)$ is infinite if and only if there is a state q of the DFA such that (i) q is reachable from the start state q_0 (ii) some accept state is reachable from q (iii) q is on some non-trivial cycle. All three conditions can be checked for every state q using standard algorithms.

Solution: We observe that $L(R) \subseteq L(S)$ if and only $L(R) \cap \overline{L(S)} = \emptyset$. Therefore the following TM decides

$$L = \{\langle R, S \rangle : R \text{ and } S \text{ are regular expressions and } L(R) \subseteq L(S)\}.$$

On input $\langle R, S \rangle$:

1. Construct a DFA D such that $L(D) = L(R) \cap \overline{L(S)}$.
2. Run a TM T that decides E_{DFA} on input $\langle D \rangle$ (i.e. decides whether $L(D)$ is empty. We did this in class).
3. If T accepts, output ACCEPT. If T rejects, output REJECT.

Problem 7

In this problem, we explore the notion of *oracle reducibility*. If A is a language, then a *Turing machine with oracle* A is a Turing machine with a "magical" subroutine that decides membership in A. In other words, the subroutine, when given a string w, tells the machine whether or not $w \in A$. Let

$$\text{HALT}_{\text{TM}} = \{\langle M, x \rangle \mid M \text{ is a Turing machine that halts on } x\}.$$

Show that there is a Turing machine with oracle HALT_{TM} that decides the following problem with only *two* questions to the oracle: Given three (machine, input) pairs $\langle M_1, x_1 \rangle, \langle M_2, x_2 \rangle, \langle M_3, x_3 \rangle$, decide for each pair whether the Turing machine halts on the corresponding input.

Note: This is trivial if one allows *three* questions. Just ask the oracle whether $\langle M_i, x_i \rangle \in \text{HALT}_{\text{TM}}$ for $i = 1, 2, 3$.

Solution: Suppose that exactly k of the machines $\{M_1, M_2, M_3\}$ halt on corresponding inputs. The idea is to figure out the value of k. Once we know k, we can decide which ones halt: simply simulate the three machines on corresponding inputs simultaneously (one step of every machine at a time), and halt when exactly k of them halt. Since we know k, we are guaranteed to halt.

This is how we figure out the value of k. First construct a machine M that simulates the three machines on corresponding inputs simultaneously and halts if at least two of them halt. Now decide whether M halts by asking the oracle.

(Case 1): If M halts, we know that $k \geq 2$. By running the machines simultaneously again, we figure out which two of the machines halt. Then by asking the oracle, we can figure out whether the third machine halts.

(Case 2): If M does not halt, we know that $k \leq 1$. Construct a machine M' that simulates the three machines on corresponding inputs simultaneously and halts if at least one of them halts. Decide whether M' halts by asking the oracle. If M' halts, then we know that $k = 1$. If M' does not halt, we know that $k = 0$.

Problem 8

Show that the collection of Turing-recognizable languages is closed under the operation of (a) union (b) concatenation (c) star and (d) intersection. What about complementation operation ?

Solution:

(a) Let L_1 and L_2 be two Turing-recognizable languages, and let M_1 and M_2 be TMs that recognizes L_1 and L_2 respectively. We construct a TM M that recognizes $L_1 \cup L_2$: On input w,

1. Run M_1 and M_2 on w "in parallel". That is, in each step, M runs one step of M_1 and one step of M_2.
2. If either of M_1 and M_2 accepts, output ACCEPT. If both reject, output REJECT.

If M_1 or M_2 accepts w, then M will halt and accept w since M_1 and M_2 are run in parallel and an accepting TM will halt and accept w in a finite number of steps. If both M_1 and M_2 reject w, then M will reject w. If neither M_1 nor M_2 accepts w and one of them loops on w, then M will loop on w. Thus $L(M) = L_1 \cup L_2$, and Turing-recognizable languages are closed under union.

(b) Let L_1 and L_2 be two Turing-recognizable languages, and let M_1 and M_2 be TMs that recognizes L_1 and L_2 respectively. We construct a NTM N that recognizes $L_1 \circ L_2$: On input w,

1. Nondeterministically split w into two parts $w = xy$.
2. Run M_1 on x. If it rejects, halt output REJECT. If it accepts, go to Step 3.
3. Run M_2 on w. If it accepts, output ACCEPT. If it rejects, output REJECT.

If $w \in L_1 \circ L_2$, then there is a way to split w into two parts $w = xy$ such that $x \in L_1$ and $y \in L_2$, thus, M_1 halts and accepts x, and M_2 halts and accepts y. Hence at least one branch of N will accept w. On the other hand, if $w \notin L_1 \circ L_2$, then for every possible splitting $w = xy$, $x \notin L_1$ or $y \notin L_2$, so M_1 does not accept x or M_2 does not accept y. Thus, all branch of N will reject w. Therefore, $L(N) = L_1 \circ L_2$, and Turing-recognizable languages are closed under concatenation.

(c) Let L be a Turing-recognizable languages, and let M be a TM that recognizes L. We construct a TM M' that recognizes L^*: On input w,

1. Nondeterministically choose an integer k and split w into k parts $w = w_1 w_2 \cdots w_k$.
2. Run M on each w_i. If M accepts all of $w_1, \ldots, w_k$, output ACCEPT. If M rejects one of $w_1, \ldots, w_k$, output REJECT.

If $w \in L^*$, then there is a splitting of $w = w_1 w_2 \ldots w_k$ such that $w_i \in L$ for each i, and thus at least one branch of M' will accept w. On the other hand, if $w \notin L^*$, then for every possible splitting $w = w_1 w_2 \ldots w_k$, $w_i \notin L$ for at least one i, thus all branches of M' will reject w. Therefore, $L(M') = L^*$, and Turing-recognizable languages are closed under the star operation.

(d) Let L_1 and L_2 be two Turing-recognizable languages, and let M_1 and M_2 be TMs that recognizes L_1 and L_2 respectively. We construct a TM M that recognizes $L_1 \cap L_2$: On input w,

1. Run M_1 on w. If it rejects, halt and output REJECT. If it accepts, go to Step 2.
2. Run M_2 on w. If it accepts, output ACCEPT. If it rejects, output REJECT.

Clearly $L(M) = L_1 \cap L_2$, and thus Turing-recognizable languages are closed under intersection.

(e) Not closed under complementation. Suppose it were true that whenever L is Turing-recognizable, so is $\overline{L}$. So whenever L is Turing-recognizable, then both $L, \overline{L}$ would be Turing-recognizable, and hence L would be decidable (as proved in class). This is a contradiction since there exists a language that is Turing-recognizable but not decidable, e.g. the language A_{TM}.

Problem 9

Show that every context free language is in P. In other words, for any fixed grammar G, design a polynomial time algorithm to test whether a given string w is generated by the grammar. The algorithm should run in time polynomial in $|w|$. The size of the grammar itself is thought of as a constant. *Hint: Use dynamic programming. Assume w.l.o.g. that the grammar is in Chomsky normal form.*

Problem 10

Show that P is closed under the star operation (*Hint: Use dynamic programming.*) Recall that for a language L,

$$L^* = \{x_1 x_2 \ldots x_k \mid k \geq 0,\ x_i \in L\ \forall 1 \leq i \leq k\}$$

Solution: We show that $\mathbf{P}$ is closed under the star operation by dynamic programming. Let $L \in \mathbf{P}$, and let A be a polynomial-time algorithm that decides L. We construct a polynomial-time algorithm B that decides L^*, as follows. On input $w = w_1 \cdots w_n$, algorithm B constructs a table T such that $T[i,j] = 1$ if $w_i \cdots w_j \in L^*$, and $T[i,j] = 0$ otherwise. Details follow.

Algorithm B:

On input $w = w_1 \cdots w_n$,

1. If $w = \varepsilon$, output ACCEPT and halt.