Rishad Rahman (Deepak's Recitation) February 19, 2015

6.046 Problem 2-1

Collaborators: *None*

(a) Compare $S[i...i+m-1]$ with $P[0...m-1]$ to see if they are equal. Do this for $i = 0$ to $i = n-m$. If there is a match return $[i, i+m-1]$. Correctness is easy since we are brute forcing all the possible matches. Running time is $O(m(n-m+1)) = O(nm)$.

(b) **Algorithm:** Let $S(x) = \sum s_i x^i$ where $s_i = -1, 1$ when $S[i] = a, b$ respectively. Let $P(x) = \sum p_i x^i$ where $p_i = -1, 1, 0$ when $P[m-1-i] = a, b, *$ respectively. Using FFT compute $S(x)P(x)$. Let j be the number of $*$'s in P. Iterate the the coefficients from $[x^0]$ to $[x^{n-1}]$ and if the coefficient of x^l is equal to $m-j$, then there is a match between $S[l-m+1...l]$ and $P[0...m-1]$ so we include $l-m+1$ in the output.

Correctness: We have to prove $S[l-m+1...l]$ matches with $P[0..m-1]$ iff $[x^l][S(x)P(x)] = m-j$ where j is the number of $*$'s in P. For the forward direction notice that $s_{l-i} = p_i$ except when $p_i = 0$ which happens j times so

$$[x^l][S(x)P(x)] = \sum_{i=0}^{m-1} s_{l-i}p_i = m-j$$

For the reverse direction suppose $\sum_{i=0}^{m-1} s_{l-i}p_i = m-j$. Since we must have j of the p_i's be 0 we get that for some S' where $|S'| = m-j$ that $\sum_{i\in S'} s_{l-i}p_i = m-j$ but this implies $\forall i \in S'$ $s_{l-i} = p_i$ in order to obtain the sum since $|S| = m-j$ so we need a product of 1 from each term. Therefore either $S[l-i] = P[m-1-i]$ or $P[m-1-i] = *$ for $i = 0, 1, ...m-1$ hence we get $S[l-m+1...l]$ matches with $P[0...m-1]$.

Example: $S = ababbab \rightarrow S(x) = -1 + x - x^2 + x^3 + x^4 - x^5 + x^6$ and $P = ab* \rightarrow P(x) = x - x^2 \Rightarrow S(x)P(x) = -x + 2x^2 - 2x^3 + 2x^4 - 2x^6 + 2x^7 - x^8$. We have $m-j = 2$ and $n-1 = 6$ so we only look up to $[x^6]$ and find $[x^2, x^4][S(x)P(x)] = 2$ so we return $[2-3+1, 4-3+1] = [0, 2]$.

(c) **Runtime:** Everything done takes linear time except for FFT which multiplies an $n-1$ degree polynomial with a $m-1$ degree polynomial hence we get a running time of $O(n \log n)$.

(d) **Algorithm:** I will be using the letter S for the source/DNA strand instead of D. Let $S_1(x) = \sum s_i^{(1)} x^i$ where $s_i^{(1)} = -1, -1, 1, 1$ when $S[i] = A, C, G, T$ respectively. Let $S_2(x) = \sum s_i^{(2)} x^i$ where $s_i^{(2)} = -1, 1, -1, 1$ when $S[i] = A, C, G, T$ respectively. Let $P_1(x) = \sum p_i^{(1)} x^i$ where $p_i^{(1)} = -1, -1, 1, 1, 0$ when $P[m-1-i] = A, C, G, T, *$ respectively. Let $P_2(x) = \sum p_i^{(2)} x^i$ where $p_i^{(2)} = -1, 1, -1, 1, 0$ when $S[i] = A, C, G, T, *$

respectively. Use FFT to compute $S_1(x)P_1(x)$ and $S_2(x)P_2(x)$. Let j be the number of $*$'s in P. Iterate the the coefficients from $[x^0]$ to $[x^{n-1}]$ of each result and if the coefficient of x^l is equal to $m-j$ for both polynomial products, then there is a match between $S[l-m+1...l]$ and $P[0...m-1]$ so we include $l-m+1$ in the output.

Correctness: Suppose $S[l-m+1...l]$ matches with $P[0...m-1]$. Then $s_{l-i}=p_i$ whether we are looking at $S_1(x)P_1(x)$ or $S_2(x)P_2(x)$, except when we have $p_i=0$ which occurs j times so

$$[x^l][S_1(x)P_1(x)] = \sum_{i=0}^{m-1} s_{l-i}^{(1)}p_i^{(1)} = m-j$$
$$[x^l][S_2(x)P_2(x)] = \sum_{i=0}^{m-1} s_{l-i}^{(2)}p_i^{(2)} = m-j$$

On the other hand suppose

$$\sum_{i=0}^{m-1} s_{l-i}^{(1)}p_i^{(1)} = \sum_{i=0}^{m-1} s_{l-i}^{(2)}p_i^{(2)} = m-j$$

Then by the same argument as in the correctness for (b) we have that for some S' we must have $\forall i \in S', s_{l-i}^{(1)} = p_i^{(1)}, s_{l-i}^{(2)} = p_i^{(2)}$ and $p_i = 0$ for all other $i \in S$. However note that any combination of $s_k^{(1)}, s_k^{(2)}$ values leads to a distinct letter for $S[k]$ as we see in the following table

$s_k^{(1)}$	Possible $S[k]$	$s_k^{(2)}$	Possible $S[k]$	$S[k]$ Intersection
-1	A,C	-1	A,G	A
-1	A,C	1	C,T	C
1	G,T	-1	A,G	G
1	G,T	1	C,T	T

The above also follows for non-zero $(p_k^{(1)}, p_k^{(2)})$ combinations giving a unique $P[m-1-k]$. Hence we get either $S[l-i] = P[m-1-i]$ or $P[m-1-i] = *$ for $i = 0, 1, ...m-1$ therefore $S[l-m+1...l]$ matches with $P[0...m-1]$.

Runtime: Same as in (b) since the most computationally heavy steps now are the two instances of polynomial multiplication so the runtime is $O(n \log n)$.

Example: $D = ACGACCAT \rightarrow S_1(x) = -1-x+x^2-x^3-x^4-x^5-x^6+x^7, S_2(x) = -1+x-x^2-x^3+x^4+x^5-x^6+x^7$ and $P = AC*A \rightarrow P_1(x) = -1-x^2-x^3, P_2(x) = -1+x^2-x^3 \Rightarrow S_1(x)P_1(x) = 1+x+3x^3+x^4+x^5+3x^6+x^7+2x^8-x^{10}, S_2(x)P_2(x) = 1-x+3x^3-3x^4-x^5+3x^6-x^7-2x^8+2x^9-x^{10}$. We have $m-j=3$ and $n-1=7$ so we only look up to $[x^7]$ and find $[x^3, x^6][S_1(x)P_1(x), S_2(x)P_2(x)] = 3$ so we return $[3-4+1, 6-4+1] = [0,3]$.

Rishad Rahman (Deepak's Recitation) February 19, 2015

6.046 Problem 2-2

Collaborators: *None*

(a) **Algorithm:** Let the roots of the trees be $R_1 = [key^1_1, key^1_2, ...key^1_m]$ and $R_2 = [key^2_1, key^2_2, ...key^2_n]$. If $m, n \geq t - 1$ then we combine them by having $[k]$ be the root of the combined tree with $k.left = R_1$ and $k.right = R_2$. Otherwise concatenate R_1,k, and R_2 into one root $[key^1_1, key^1_2, ...key^1_m, k, key^2_1, key^2_2, ...key^2_n]$ with $k.left = key^1_m.right$ and $k.right = key^2_1.left$. If there is overflow at the root, use the overflow correction algorithm where we push the median element a level above, in this case making it the root of the entire tree.

Correctness: Since $m, n \geq t - 1$ the first case tells us that R_1 and R_2 can be valid subtrees in a t-tree hence making $[k]$ the root works. The second case tells us $m + n \leq t - 2 + 2t - 1 = 3t - 3$ so after we concatenate the root has $\leq 3t - 2$ keys. Clearly everything below the root satisfy the t-tree conditions so we only get a violation if the number of keys in the root is $\geq 2t$. However the overflow algorithm will push the median key into a root and each half of the previous root becomes its children hence the lower bound on the number of keys is $\frac{2t-1}{2} > t - 1$, the upper bound is $\frac{3t-3}{2} < 2t - 1$, and obviously the height is constant since we either concatenate the roots or make them children of a new root, so the resulting tree is a valid t-tree.

Runtime: All the operations involve the topmost elements of the trees a constant number of times without ever needing to traverse down a tree including the overflow correction since everything below the roots satisfy the tree conditions and we aren't messing with anything below the roots, hence we get $O(1)$.

(b) **Algorithm:** Concatenate k with R_1 so that $k.left = key^1_m.right$ and $k.right = R_2$. Use overflow correction on the root if necessary then use underflow correction on $k.right = R_2$ if necessary.

Correctness: Concatenating R_1 with k gives a left height of h_1 while making $k.right = R_2$ gives a right height of $h_2 + 1 = h_1$ so the heights match. If there is overflow then it must be at the root which will contain $2t$ elements so the correction properly executes and will not mess with anything below the root i.e. an underflow node. Then we have at maximum one underflow which is at max 2 levels below the root which is properly handled by the underflow correction since this is one the cases that can arise from a deletion operation on a B-tree.

Runtime: We still only do a constant number of operations on the original roots, the possible overflow is only at the root, and the possible underflow is at maximum 2 levels below the root so each step takes constant time hence $O(1)$.

(c) The base case of insertion is when we insert for the very first time where we give the node a height of 0. When we insert we push keys upwards to fix overflow. If this results in a node splitting we give each of the new nodes the height of the original node they were a part of. The only other time the height can possibly change is when a new root is made from pushing an element from the previous overflowed root. We then declare $h_{newroot} = h_{oldroot} + 1$ and everything below the new root has the same height as before so we are good. The only time heights can possibly change in deletion is when we reach the root since no nodes are shifted up or down during the correction, just keys being shifted around or nodes on the same level merging which obviously doesn't affect heights. At the root there is a special case where it may be empty, but then it gets deleted and the root pointer gets passed down. Therefore the height augmentations really do not change during deletion. The only work we do is constant time whenever we split (which is done at max once per level of the tree) during insertion and $h_{newroot} = h_{oldroot} + 1$ which obviously takes constant time so the runtimes of insert and delete are not changed.

(d) **Algorithm:** Suppose $h_1 \geq h_2$. Go down the rightmost children pointers of h_1 until you get to a node, $N = [key_1 ... key_n]$, of height $h_2 + 1$. Concatenate N with k and then insert R_2 (the root of T_2) so that $key_n.right = k.left$ and $k.right = R_2$. Correct overflow at the new N and then underflow at R_2. The case with $h_1 < h_2$ is completely symmetric except that we go down leftmost children pointers of h_2 and then concatenate k and insert R_1 to the left of a node with height h_1.

Correctness: I will only argue where $h_1 \geq h_2$ since the other case is symmetric. The argument is essentially the same as the one in (b). $k.left$ has a height of $h_2 + 1 - 1 = h_2$ and now $k.right$ has a height of h_2. Now the total height of the tree including the concatenation of k and insertion of T_2 is still $h_1 - h_2 + h_2 = h_1$. Now there might be overflow at N after concatenation but as said before there would be t keys so this is analogous to the overflow correction during insertion where we keep pushing a key upwards but since the overflow started at height $h_2 + 1$ it won't affect anything below including the possible underflow node. Again as before, the single underflow node can be corrected by the underflow correction seen in the deletion operation of a B tree by moving keys or merging same level nodes and recursing upwards.

Runtime: Concatenating k is $O(1)$. Traversing down to a node of height h_2, correcting a possible overflow at height h_2, and correcting a possible underflow at maximum $h_1 - h_2 + 1$ levels below the root, all take $O(h_1 - h_2)$ time. If $h_1 < h_2$ then it would've been $O(h_2 - h_1)$ because of the symmetry. Hence our total runtime is $O(|h_1 - h_2| + 1)$.