Problem Set 2 Solutions

This problem set is due **at 11:59pm** on **Thursday, February 19, 2015.**

Exercise 2-1. Read CLRS, Sections 30.1 and 30.2.

Exercise 2-2. Exercise 30-2.3.

Exercise 2-3. Exercise 30-2.4.

Exercise 2-4. Read CLRS, Chapter 18.

Exercise 2-5. Exercise 18.2-5

Exercise 2-6. Exercise 18.3-2

Problem 2-1. Pattern Matching [25 points]

Suppose you are given a ***source string*** $S[0\,..\,n-1]$ of length n, consisting of symbols a and b. Suppose further that you are given a ***pattern string*** $P[0\,..\,m-1]$ of length $m \ll n$, consisting of symbols a, b, and $*$, representing a pattern to be found in string S. The symbol $*$ is a "wild card" symbol, which matches a single symbol, either a or b. The other symbols must match exactly.

The problem is to output a sorted list M of valid "match positions", which are positions j in S such that pattern P matches the substring $S[j\,..\,j+|P|-1]$. For example, if $S = a\,b\,a\,b\,b\,a\,b$ and $P = a\,b\,*$, then the output M should be $[0, 2]$.

(a) [4 points] Describe a straightforward, naïve algorithm to solve the problem. Your algorithm should run in time $O(nm)$.

Solution: One can explicitly check every possible starting position $s \in \{0, 1, \ldots, n-m\}$ by checking whether each entry in P matches from s to $s+m-1$.

```
NAIVE-ALGORITHM(S, P)
1  M = [ ]
2  for s = 0 to n − m
3       valid = TRUE
4       for j = 0 to m − 1
5            if P[j] ≠ ∗ and P[j] ≠ S[s + j]
6                 valid = FALSE
7       if valid
8            M.APPEND(s)
9  return M
```

(b) [12 points] Give an algorithm to solve the problem by reducing it to the problem of polynomial multiplication. Specifically, describe how to convert strings S and P into polynomials such that the product of the polynomials allows you to determine the answer M. Give examples to illustrate your polynomial representation of the inputs and your way of determining outputs from the product, based on the example S and P strings given above.

Solution: Let's represent a by 1, b by -1, and $*$ by 0. We will use these representations instead of the original symbols in this solution.

Notice that P matches S starting from position j, $0 \leq j \leq n-m$, if and only if for every i, $0 \leq i \leq m-1$, either $P[i] = 0$ or $S[j+i]P[i] = 1$. This is true if and only if

$$\sum_{i=0}^{m-1} S[j+i]P[i] = k,$$

where k is the number of non-$*$ symbols in P.

We would like to express these summations as coefficients of a product of polynomials. Let x be a variable. Represent S as

$$f_S(x) = S[0] + S[1]x + \cdots + S[n-1]x^{n-1}.$$

Represent P as

$$f_P(x) = Q[0] + Q[1]x + \cdots + Q[m-1]x^{m-1},$$

where each $Q[i] = P[m-1-i]$. Thus, we have reversed the order of the coefficients in this last representation.

Suppose C is the product of the S polynomial and the P polynomial. Then the coefficient of x^{m-1+j} in C is

$$\sum_{i=0}^{m-1} S[m-1+j-i]Q[i],$$

which is equal to

$$\sum_{i=0}^{m-1} S[i+j]P[i].$$

This is the same as the sum above. To obtain the output M, we simply examine all the coefficients of C, outputting position number j, $0 \leq j \leq n-m$, exactly if the coefficient of x^{m-1+j} is equal to k, the total number of non-$*$ symbols in P. We output these in order of increasing j, as required.

In the example above, S is represented by

$$f_S(x) = 1 - x + x^2 - x^3 - x^4 + x^5 - x^6$$

and P by

$$f_P(x) = -x + x^2.$$

The product C is

$$-x + 2x^2 - 2x^3 + 2x^4 - 2x^6 + 2x^7 - x^8.$$

The number k is equal to 2, so the terms of interest are $2x^2$, $2x^4$, and $2x^7$. These would yield $j = 0, 2, 5$, but 5 is ruled out because we are only considering $j \leq n - m = 7 - 3 = 4$.

(c) [3 points] Suppose you combine your solution to Part (b) with an FFT algorithm for polynomial multiplication, as presented in Lecture 3. What is the time complexity of the resulting solution to the string matching problem?

Solution: It's $O(n \lg n)$. It takes time $O(n \lg n)$ to perform the needed DFT and inverse DFT algorithms, and $O(n)$ for producing inputs for the DFT algorithm and extracting M from the outputs.

(d) [6 points] Now consider the same problem but with a larger symbol alphabet. Specifically, suppose you are given a representation of a DNA strand as a string $D[0\,..\,n-1]$ of length n, consisting of symbols A, C, G, and T; and you are given a pattern string $P[0\,..\,m-1]$ of length $m \ll n$, consisting of symbols A, C, G, T, and $*$.

The problem is, again, to output a sorted list M of valid "match positions", which are positions j in D such that pattern P matches the substring $D[j\,..\,j + |P| - 1]$. For example, if $D = A\,C\,G\,A\,C\,C\,A\,T$ and $P = A\,C * A$, then the output M should be $[0, 3]$.

Based on your solutions to Parts (b) and (c), give an efficient algorithm for this setting. Illustrate your algorithm on the example above.

Solution: Use a reduction. Encode A as $a\,a$, C as $a\,b$, G as $b\,a$, T as $b\,b$, and $*$ as $**$. Use our previous solution to solve the problem on the resulting string, obtaining a list M' of positions.

The final output list M will consist of just the even numbers from the list M', all divided by 2.

This will take the time it takes to convert and then solve the original problem on arrays of length $2n$ and $2m$ respectively:

$$O(2n + 2n \lg(2n)) = O(n \lg n).$$

Problem 2-2. Combining B-trees [25 points]

Consider a new B-tree operation COMBINE(T_1, T_2, k). This operation takes as input two B-trees T_1 and T_2 with the same minimum degree parameter t, plus a new key k that does not appear in either T_1 or T_2. We assume that all the keys in T_1 are strictly smaller than k and all the keys in T_2 are strictly larger than k. The COMBINE operation produces a new B-tree T, with the same minimum degree t, whose keys are those in T_1, those in T_2, plus k. In the process, it destroys the original trees T_1 and T_2.

In this problem, you will design an algorithm to implement the COMBINE operation. Your algorithm should run in time $O(|h_1 - h_2| + 1)$, where h_1 and h_2 are the heights of trees T_1 and T_2 respectively. In analyzing the costs, you should regard t as a constant.

(a) [5 points] First consider the special case of the problem in which h_1 is assumed to be equal to h_2. Give an algorithm to combine the trees that runs in constant time.

Solution: Construct a new root node for T consisting of the root nodes for T_1 and T_2, with the root of T_1 at the left, and with k inserted between the keys of the two original roots. Then, if the number of keys in the resulting root node is at least $2t - 1$ then split the root node around its median, forming a new root node containing one key and two child nodes containing at least $t - 1$ keys apiece.

```
COMBINE(T_1, T_2, k)
 1  T = T_1
 2  R = T.root
 3  R_2 = T_2.root
 4  i = R.n + 1
 5  ▷ Merging the two roots into one
 6  T.k_i = k
 7  for j = 1 to R_2.n
 8       i = i + 1
 9       T.k_i = R_2.k_j
10       T.c_i = R_2.c_j
11  T.c_{i+1} = R_2.c_{R_2.n+1}
12  R.n = i + 1
13  if R.n ≥ 2t − 1
14       ▷ Splitting node
15       mid = ⌈R.n/2⌉
16       dummy = ALLOCATE-NODE()
17       dummy.n = 1
18       dummy.k_1 = R.k_{mid}
19       ▷ Allocating Children
20       C_1 = ALLOCATE-NODE()
21       C_1.n = mid − 1
22       C_2 = ALLOCATE-NODE()
23       C_2.n = R.n − mid
24       ▷ Creating first child
25       C_1.c_1 = T.c_1
26       for j = 1 to C_1.n
27            C_1.k_j = T.k_j
28            C_1.c_{j+1} = T.c_{j+1}
29       ▷ Creating second child
30       C_2.c_1 = T.c_{mid+1}
31       for j = 1 to C_2.n
32            C_1.k_j = T.k_{j+mid}
33            C_1.c_{j+1} = T.c_{j+mid+1}
34       dummy.c_1 = C_1
35       dummy.c_2 = C_2
36       T.root = dummy
37  return T
```

(b) [5 points] Consider another special case, in which h_1 is assumed to be exactly equal to $h_2 + 1$. Give a constant-time algorithm to combine the trees.

Solution:

Append k to the right end of the right child node of T_1 and append the root of T_2 to that. Clearly this preserves sorted order. Now the right child may have anywhere from $t+1$ to $4t-1$ keys. If it has $2t-1$ or more keys, then split it around its median key. If that causes the root node to have $2t-1$ nodes then split that around its median key, thus adding another level to the tree.

```
COMBINE(T_1, T_2, k)
 1  x = T_1.root
 2  x = x.c_{x.n+1}
 3  r = T_2.root
 4  n = x.n + 1 + r.n
 5  ▷ Append k to the rightmost child
 6  x.k_{x.n+1} = k
 7  ▷ Append the root of T_2 to the node
 8  x.c_{x.n+2} = r.c_1
 9  for j = 1 to r.n
10      x.k_{x.n+j+2} = r.k_j
11      x.c_{x.n+j+3} = r.c_{j+1}
12  ▷ Split node if too big
13  if n ≥ 2t − 1
14      p = x.parent
15      n = p.n
16      B-TREE-SPLIT-CHILD(p, n)
```

(c) [5 points] Now consider the more general case in which h_1 and h_2 are arbitrary. Because the algorithm must work in such a small amount of time, and must work for arbitrary heights, a first step is to develop a new kind of ***augmented B-tree*** data structure in which each node x always carries information about the height of the subtree below x. Describe how to augment the common B-tree insertion and deletion operations to maintain this information, while still maintaining the asymptotic time complexity of all operations.

Solution: Augment the tree by adding a height attribute for each node. The height of a leaf node is 0. For internal nodes, $\text{HEIGHT}(x) = \text{HEIGHT}(x.c_1) + 1$.

Insertion: New nodes are added during splitting. The newly allocated node in a split has the same height as the original node. The only other time a node is added is when the root is split. This is done by making the root the child of a dummy node and then splitting it. The height of the new root is set to one greater than the height of the old root.

Deletion: In deletion, no nodes are deleted except the root. Since height values are indexed starting at the leaf, deletion does not affect node heights.

With these additions, the asymptotic running time for insertion and deletion is that same as before, $O(\lg n)$.

(d) [10 points] Now give an algorithm for combining two B-trees T_1 and T_2, in the general case where h_1 and h_2 are arbitrary. Your algorithm should run in time $O(|h_1 - h_2| + 1)$.

Solution:

If $|h_1 - h_2| < 2$, use part (a) or (b). Otherwise, assume that $h_1 > h_2 + 1$ ($h_2 > h_1 + 1$ works symmetrically). Let x be the rightmost node of T_1 at level h_2. Add k at the right end of x and append the root of T_2 to that. Now node x may have anywhere from $t + 1$ to $4t - 1$ keys. If it has $2t - 1$ or more keys, then split it around its median key. The split may propagate upwards, possibly as far as the root. So, the time complexity depends linearly on the height difference $O(|h_1 - h_2|)$.

```
COMBINE(T_1, T_2, k)
 1  T = T_1
 2  h_1 = T_1.height
 3  h_2 = T_2.height
 4  x = T.root
 5  ▷ Move to the rightmost node at level h_2
 6  for j = 1 to h_1 − h_2
 7       n = x.n
 8       x = x.c_n
 9  r = T_2.root
10  n = x.n + 1 + r.n
11  ▷ Append k to the node
12  x.k_{x.n+1} = k
13  ▷ Append the root of T_2 to the node
14  x.c_{x.n+2} = r.c_1
15  for j = 1 to r.n
16       x.k_{x.n+j+2} = r.k_j
17       x.c_{x.n+j+3} = r.c_{j+1}
18  ▷ Split node if too big
19  if n ≥ 2t − 1
20       p = x.parent
21       n = p.n
22       B-TREE-SPLIT-CHILD(p, n)
23  return T
```

6.046J / 18.410J Design and Analysis of Algorithms
Spring 2015