Алгоритмы и структуры данных. Семинар XX. Строки. Часть первая.
Григорьев Дмитрий БПМИ-163

Задача 1.

Для начала проверим, что $|s_1| = |s_2|$, если это неверно, то строка s_1 не может являться циклическим сдвигом строки s_2.
Склеим две строки s_2, то есть получим две подряд идущих строки s_2, $s_3 = s_2 + s_2$.
Далее проверим с помощью функции $isSubstring$ является ли строка s_1 подстрокой новой строки s_3. Таким образом, функцию $isSubstring$ вызовем один раз.
Почему это будет работать?
Очевидно что в строке s_2 разных циклических двигов – $|s_2|$, так как они будут повторятся потом. Циклический сдвиг – это склееные две строки:
1) суффикс s_2 длины $m, m <= |s_2|$
2) префикс s_2 длины $|s_2| - m$
А новая строка s_3 как раз содержит все возможные варианты склеенных суффиксов и префиксов строки s_2.

Задача 2.

Нам нужно построить префикс-функцию, имея посчитанную Z-функцию, не восстанавливая строки.
Если внимательно посмотреть на Z-функцию и на ее определение и так же на префикс-функцию, то заметим, что:
1) в Z-функции – каждый элемент $Z[i]$ равен длиннейшему префиксу подстроки, начинающейся с позиции i в строке S, который одновременно является и префиксом всей строки S.
2) в префикс-функции – $P[i]$ – длина наибольшего префикса строки $S[0 \dots i]$, (исключая вырожденный случай, где префикс совпадает с этой строкой), который одновременно является её суффиксом.
Поэтому если $Z[i] > 0$, то для всех элементов с индексом $i + j$, где $0 \leq j < Z[i]$, значение $P[i+j]$ будет не меньше, чем длина подстроки с i по $i+j$, что равно $j+1$.
Но если мы до этого обновили значение $P[i + j]$, то это было раньше чем в позиции i, поэтому нам не нужно обновлять значение с $P[i]$ по $P[i + j]$, так как мы только сможем уменьшить его, так как если $i' < i$, то $j' > j$, так как $i' + j' = i + j$, следовательно каждый элемент $P[i]$ мы обновм только один раз, поэтому этот алгоритм будет линейным.

Пример кода:

```
for (int i = 0; i < n; ++i) {
   P[i] = 0;
}
for(int i = 1; i < n; ++i) {
   if (Z[i] > 0) {
      for(int j = Z[i] - 1; j >= 0 && (P[i + j]) == 0; --j) {
         P[i + j] = j + 1;
      }
   }
}
```

Задача 3.

Заметим, что ответом будет длина наименьшей строки, порождающей исходную строку, то есть длина такой строки, при склеивании которой несколько раз получается исходная.
Найти длину такой строки можно с помощью префикс-функции.
Посчитаем префикс-функцию для строки $|s|$.
Далее если $|s|$ делится на $|s| - p[|s| - 1]$, то ответом и будет $|s| - p[|s| - 1]$, иначе ответ $|s|$.
Если $|s|$ делится на $|s| - p[|s| - 1]$, то строку s можно представить в виде суммы строк длины $|s| - p[|s| - 1]$, причём, по определению префикс-функции, префикс длины $n - |s| - p[|s| - 1]$ будет совпадать с её суффиксом. Но тогда последний блок должен будет совпадать с предпоследним, предпоследний - с предпредпоследним, и т.д. В итоге получится, что все блоки блоки совпадают, и такое $|s| - p[|s| - 1]$ действительно подходит под ответ.

Задача 4.
Разобьем стороку p на подстроки по символу *. Тогда p можно разбить на некоторое множество подстрок. Если все эти подстроки входят в строку s и позиции их вхождения возрастают, то можно сделать так, чтобы строка p стала подстрокой s. Для того, чтобы проверить, что все подстроки p входили в s, с возрастающим вхождением, посчитаем префикс-функцию для строки s, далее с помощью алгоритма Кнута-Морриса-Пратта будем считать, является ли очередная подстрока из p подстрокой s. Но при переходе к новой подстроке p мы будем не заново пересчитывать префикс-функцию для строки s, а пересчитывать с вхождения предыдущей подстроки p. Если же вхождения какой либо подстроки p не существует, то p не может быть подстрокой s. Таким образом мы вычислим префикс функцию для всех подстрок p и префикс функцию для s. Используемое время – $O(|t| + |s|)$.

Задача 7.

а)
Воспользуемся алгоритмом Ахо-Карасика

Обычный Ахо-Карасик будет иметь квадратичную ассимптотику, поэтому будем хранить "хорошую суффиксную ссылку" – это ближайший суффикс, имеющийся в боре, для которого $flag = true$. Число "скачков" при использование таких ссылок уменьшится и станет пропорционально количеству искомых вхождений, оканчивающихся в этой позиции.
Вычислять будем ленивой динамикой. Если для вершины по суффиксной ссылке $flag = true$, то это и есть искомая вершина, в ином случае рекурсивно запускаемся от этой же вершины.

Задача 9.

а) Воспользуемся алгоритмом Манакера
Введем два массива d_1, d_2 – количество палиндромов соответственно нечётной и чётной длины с центром в позиции i. Для быстрого вычисления будем поддерживать границы (l,r) самого правого из обнаруженных подпалиндрома (т.е. подпалиндрома с наибольшим значением r).
При этом все предыдущие значения в массиве d уже посчитаны. Возможны два случая:
$i > r$, т.е. текущая позиция не попадает в границы самого правого из найденных палиндромов. Тогда просто запустим наивный алгоритм для позиции i.
$i \leqslant r$. Тогда попробуем воспользоваться значениями, посчитанным ранее. Отразим нашу текущую позицию внутри палиндрома $[l;r] : j = (r - i) + l$. Поскольку i и j — симметричные позиции, то если $d_1[j] = k$, мы можем утверждать, что и $d_1[i] = k$. Это объясняется тем, что палиндром симметричен относительно своей центральной позиции. Т.е. если имеем некоторый палиндром длины k с центром в позиции $l \leqslant i \leqslant r$, то в позиции j, симметричной i относительно отрезка $[l;r]$ тоже может находиться палиндром длины k. Это можно лучше понять, посмотрев на рисунок. Снизу фигурными скобками обозначены равные подстроки. Однако стоит не забыть про один граничный случай: что если $i + d_1[j] - 1$ выходит за границы самого правого палиндрома? Так как информации о том, что происходит за границами это палинлрома у нас нет (а значит мы не можем утверждать, что симметрия сохраняется), то необходимо ограничить значение $d_1[i]$ следующим образом: $d_1[i] = \min(r - i, d_1[j])$. После этого запустим наивный алгоритм, который будет увеличивать значение $d_1[i]$, пока это возможно. После каждого шага важно не забывать обновлять значения $[l;r]$.
Заметим, что массив d_2 считается аналогичным образом, нужно лишь немного изменить индексы.
Описанный выше алгоритм работает за время $O(n)$.