字符串2

以下如果无特殊说明，字符串下标都是从1开始

这里我们考虑字符串里最常见的问题，字符串匹配问题。字符串匹配问题是指，给定两个字符串s和p，分别长n和m，问p作为s的子串的第一次出现的起始位置（或所有出现的起始位置）。我们先考虑第一次出现位置的问题，之后再讨论所有出现的问题。

字符串匹配问题有个相对简单的做法，即利用字符串哈希。

字符串哈希

之前提过“哈希”的概念。字符串是很常见的数据结构，经常需要存储进哈希表中，也经常作为hash map的key出现。无论是C++还是Java里，字符串的哈希函数都有内置的默认版本。这里我们希望手动实现一个哈希函数。

我们将字符串哈希成`unsigned long long`。一个字符串可以看成是个`P`进制数。空串的哈希值定义为0，非空的串，例如`abc`可以哈希为$a*P^2+b*P+c$，即最左边的字母看成是最高位，向右走就是低位。一般取$P=131$或者13331，这样保证哈希冲突概率最小。

给定一个长n的字符串s。我们构造一个所谓的“前缀哈希数组”h，$h[k]$表示s的长k前缀的哈希值。所以，$h[0]=0$，而$h[k]=h[k-1]*P+s[i]$。子串$s[l:r]$的哈希值也可以通过h数组算出来，公式是：$h[r]-h[l-1]*P^{r-l+1}$。

显然如果两个串相等，那么它们的哈希值一定相等；而如果两个串的哈希值相等，这两个串不一定相等（对应着哈希冲突的情况）。但是在绝大部分算法题下，可以很安全的假设哈希值相等，两个串就相等，而不需要真的去验证。所以只要提前预处理出h数组，还有P的幂次数组（定义p数组，$p[k]=P^k$），就可以$O(1)$时间求出s的任意子串的哈希值。

看例题：https://blog.csdn.net/qq_46105170/article/details/113778417

字符串匹配之Rabin-Karp算法

字符串匹配，问题叙述为，给定两个字符串s和t，问t是否为s的子串，并且求出t在s中第一次出现的下标（或者出现的每一次的下标）。

例题：https://blog.csdn.net/qq_46105170/article/details/106157852

字符串匹配有很多种做法来做。此处不讨论暴力匹配做法。比较常用的做法有字符串哈希，KMP。更高级的做法有，后缀数组，后缀自动机，这两个知识点非常之难，以后会介绍。本文只介绍字符串哈希做法和KMP。

假设s的长度是n，t的长度是m。如果$m > n$，那显然无解。否则，一个直接的想法是，先求出s的前缀哈希数组和p数组，然后求出t的哈希值，接着暴力枚举s的所有长度为m的子串的哈希值，看是否等于t的哈希值。

这种方法比较粗暴，但时空复杂度都很优秀，时间复杂度是$O(n+m)$，空间$O(n)$，已经是最优了。

比较聪明的改进，是用所谓的滚动哈希的技巧，即Rolling Hash。这种算法叫做Rabin-Karp算法。如果我们已经求出了$s[1:m]$的哈希值（即$h[m]$），由于我们只关心s的长m子串的哈希值，我们考虑如何求$s[2:m+1],...,s[k:m+k-1]$的哈希值。方便起见，我们直接以$h(s)$表示某个串s的哈希值。

考虑$h(s[2:m+1])$，由于：

1. $h(s[1:m+1]) = h[m+1] = h[m]*P + s[m+1] = h(s[1:m])*P + s[m+1]$
2. $h(s[1:m+1]) = h(s[2:m+1]) + s[1]*P^m = h(s[2:m+1]) + s[1]*p[m]$

所以得到$h(s[2:m+1]) = h(s[1:m])*P + s[m+1] - s[1]*p[m]$。

同理：$\forall k \geq 2, h(s[k:m+k-1]) = h(s[k-1:m+k-2])*P + s[m+k-1] - s[k-1]*p[m]$

所以只需要先求出$h[m]$和$p[m]$，就可以继续递推出所有s的长m子串的哈希值，然后和t的哈希值进行比对即可。

例题：https://blog.csdn.net/qq_46105170/article/details/113805346

代码（注意原题是下标从0开始的）：

```cpp
#include <iostream>
using namespace std;
using ll = long long;

const int N = 1e6 + 10;
const ll P = 131;
int n, m;
char s[N], p[N];
ll hashP, hashS, po = 1;

int main() {
  scanf("%d", &m);
```

```
  scanf("%s", p + 1);
  scanf("%d", &n);
  scanf("%s", s + 1);

  for (int i = 1; i <= m; i++) {
    hashP = hashP * P + p[i];
    po = po * P;
  }

  for (int i = 1; i <= n; i++) {
    hashS = hashS * P + s[i];
    if (i >= m) {
      hashS -= s[i - m] * po;
      if (hashS == hashP) printf("%d ", i - m);
    }
  }
}
```

字符串匹配之KMP算法

KMP（Knuth-Morris-Pratt）算法是非常精妙，也是非常难的算法。题目叙述依然是，有两个字符串s和p，分别长n和m，求p在s中第一次出现的下标。注意下标从1开始。

我们先考虑字符串匹配的暴力解法。容易想到，对于s，我们可以暴力枚举每个位置i，然后验证$s[i:i+m-1]$是否和t相等，验证相等的过程就是从左到右依次进行字符匹配。如下图：

```
pos:  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15
s:    A  B  A  B  D  A  B  A  C  D  A  B  A  B  C
p:    A  B  A  B  C  A  B  A  B
```

如果匹配完成了，那自然就找到了答案；如果比对的过程中发现了失配，比如上图中，先从$s[1]$开始匹配，$s[5]$的地方发生了失配，那么我们自然而然想到，要将p向后挪1位，然后再从$s[2]$的位置开始匹配。如下：

```
pos:  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15
s:    A  B  A  B  D  A  B  A  C  D  A  B  A  B  C
p:       A  B  A  B  C  A  B  A  B
```

这样做肯定是没有错的，但是是否能进一步优化呢？

由于我们知道了下标5是首次失配的地方，那么就说明$s[1:4]$都是完美匹配的，也就是$s[1:4]=p[1:4]$，那么由于$p[1:3]\neq p[2:4]$，所以p只向后挪1位是不可能匹配的；但是$p[1:2]=p[3:4]$，所以p向后挪2位，起码我们知道$p[1:2]$肯定能和$s[3:4]$匹配上，只需继续从$s[5]$开始匹配就行了。如下：

```
pos:  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15
s:    A  B  A  B  D  A  B  A  C  D  A  B  A  B  C
p:          A  B  A  B  C  A  B  A  B
```

我们考虑一个问题：我们是怎么知道当$s[5] \neq p[5]$的时候，p向后挪1步肯定匹配不上，而p向后挪2步就恰好能匹配2个字符的？关键就在$p[1:k] = p[i-k:i-1]$这个等式上。如果我们知道了$p[i]$和$s[j]$这个位置发生了失配，那么由这个等式，如果将$p[k+1]$这个字符与$s[j]$对齐，起码我们能知道$p[1:k]$这一部分已经匹配了，不需要再比较，直接继续匹配后面的部分就行了。为了保证正确，对于每个i，我们要找到最长的k满足等式，这样p向后挪的步数最少，保证了正确性。令$k = ne[i]$，这个ne数组就是所谓的“next”数组，它表示的是，当$s[j]$和$p[i+1]$失配了，$s[j]$应该继续和$p[k+1]$进行匹配。在上面的例子里，$ne[4] = 2$，所以当$s[5] \neq p[5]$的时候发生了失配，$s[5]$应该继续和$p[2+1] = p[3]$进行匹配。

next数组的含义

直观上看，字符串p下的$ne[i]$的含义是，$ne[i] = \max\{k : k < i \wedge p[1:k] = p[i-k+1:i]\}$，即要找到$p[1:i]$这一段的最长的相等的真前后缀的长度。如果有了这个next数组，字符串匹配就非常快了。代码如下：

```cpp
vector<int> buildNe(string& p);

int find(string& s, string& p) {
  int n = s.size(), m = p.size();
  // 为了下标从1开始
  s = " " + s;
  p = " " + p;
  auto ne = buildNe(p);
  for (int i = 1, j = 0; i <= n; i++) {
    while (j && s[i] != p[j + 1]) j = ne[j];
    if (s[i] == p[j + 1]) j++;
    // 这个是下标从1开始的情况下的答案，如果题目约定下标从0开始，那还需要调整
    if (j == m) return i - m + 1;
  }
  return -1;
}
```

我们看一下这个代码的逻辑。for循环里，每次都是$s[i]$和$p[j+1]$进行匹配。下面这个while循环，如果匹配不上那么j就要跳到$ne[j]$，这一点是很好理解的；但如果j等于0的话，while循环也要退出来，这是因为$ne[0] = 0$，所以如果$j = 0$但是依然没有匹配上，j不会变，这就死循环了。

```cpp
while (j && s[i] != p[j + 1]) j = ne[j];
```

接下来这一句比较好理解，如果匹配上了，下一个要匹配的字符应当是向后挪一位。

```cpp
if (s[i] == p[j + 1]) j++;
```

如果$j = m$，说明找到了匹配，返回起始点下标。

```
if (j == m) return i - m + 1;
```

next数组的求法

这是最难最不好理解的部分。首先容易知道，ne数组只和p本身有关，并且$ne[0] = ne[1] = 0$。

我们考虑ne数组怎么递推。如果我们已经知道了$ne[k]$，那$ne[k+1]$应该是多少?

我们回顾一下这两个变量的含义，比如我们看`abadabada`，那么$ne[7] = 3$，因为`abc`是最长的相等的真前后缀长度。见：

```
 1 2 3  4  5 6 7  8 9
[a b a] d [a b a] d a
```

考虑$ne[8]$，由于我们看到$p[4] = p[8]$，那$ne[8]$必然等于$ne[7] + 1 = 4$。即，如果$p[k] = p[ne[k] + 1]$，那么$ne[k+1] = ne[k] + 1$。道理是很简单的，因为如果$ne[k+1] > ne[k] + 1$的话，那么我们回头看$ne[k]$，根据$ne[k+1]$的定义，$ne[k]$至少能取到$ne[k+1] - 1$，这就矛盾了（可以画个图看看）。

我们隐隐的发现，为了计算$ne[k+1]$，上面这一步是将$p[k]$与$p[ne[k] + 1]$做匹配。如果匹配上了，就是上面的情形。如果匹配不上呢？我们考虑`p=cbcccbcba`，接着将示意图画出来：

```
 1 2 3  4  5 6 7  8 9
[c b c] c [c b c] b a
          [c b c] c [c b c] d a
```

有$p[4] \neq p[8]$。

由$ne[7]$的定义，我们能保证$p[1:3] = p[5:7]$，现在既然$p[4] \neq p[8]$了，我们应该做什么呢？看起来似曾相识，当做字符串匹配的时候，$s[i]$和$p[j+1]$不匹配的时候，我们是让$j = ne[j]$接着继续匹配。这里其实道理是一样的。我们需要让$p[8]$和$p[ne[3] + 1] = p[2]$匹配！

```
 1 2 3  4  5 6 7  8 9
[c b c] c [c b c] b a
          [c b c] c [c b c] d a
          [c]b[c] c  c b c  d a
              [c] b [c]c c  b c d a
```

这里$p[8] = p[2]$了，从而$ne[8] = ne[3] + 1 = 2$。

总结一下就是要求$ne[k+1]$，我们尝试比较$p[k]$和$p[ne[k]+1]$，如果$p[k] \neq p[ne[k]+1]$，那么就继续比较$p[k]$和$p[ne[ne[k]]+1]$，直到$p[ne[ne...[k]]+1]$等等，如果中间遇到能匹配了，那么$ne[k+1] = ne[ne...[k]]+1$。如果一直匹配不了，按照定义$ne[k+1]$就为0。

递推的初始条件，$ne[0] = ne[1] = 0$，从$ne[2]$开始算，即一开始匹配$p[2]$和$p[1]$。

构造next数组的过程的代码如下：

```
void build_ne() {
  for (int i = 2, j = 0; i <= m; i++) {
    while (j && p[i] != p[j + 1]) j = ne[j];
    if (p[i] == p[j + 1]) j++;
    ne[i] = j;
  }
}
```

完整代码：

```
class Solution {
 public:
  int strStr(string s, string p) {
    int n = s.size(), m = p.size();
    // 为了让下标从1开始，前面加个空格
    s = " " + s;
    p = " " + p;

    vector<int> ne(m + 1);
    for (int i = 2, j = 0; i <= m; i++) {
      while (j && p[i] != p[j + 1]) j = ne[j];
      if (p[i] == p[j + 1]) j++;
      ne[i] = j;
    }

    for (int i = 1, j = 0; i <= n; i++) {
      while (j && s[i] != p[j + 1]) j = ne[j];
      if (s[i] == p[j + 1]) j++;
      if (j == m) return i - m;
    }

    return -1;
  }
};
```

构造next数组时间复杂度$O(m)$，m是p的长度；匹配的时间是$O(n+m)$，n是s的长度。我们额外空间只用了$O(m)$。

多次匹配问题

现在考虑，如果要求p在s中的每一次出现的下标呢？思路很简单，只需要找到匹配之后，记录答案但是不立即返回，而是让$j = ne[j]$（这一步其实就是假装最后一个字符匹配失败要做的式子），然后继续匹配就行了。我们发现，多次匹配竟然和单次匹配的时间复杂度完全一样，这是因为在匹配的过程中，字符串p总是前进的，而s的指针也不会回退，所以总体时间依然是线性的。例题：https://blog.csdn.net/qq_46105170/article/details/113805346

```cpp
#include <iostream>
using namespace std;

const int N = 1e6 + 10;
int n, m;
char p[N], s[N];
int ne[N];

void build_ne() {
  for (int i = 2, j = 0; i <= m; i++) {
    while (j && p[i] != p[j + 1]) j = ne[j];
    if (p[i] == p[j + 1]) j++;
    ne[i] = j;
  }
}

int main() {
  cin >> m >> p + 1 >> n >> s + 1;

  build_ne();

  for (int i = 1, j = 0; i <= n; i++) {
    while (j && s[i] != p[j + 1]) j = ne[j];
    if (s[i] == p[j + 1]) j++;
    // 找到了一次匹配
    if (j == m) {
      // 注意原题里输出答案需要按照下标从0开始输出
      printf("%d ", i - m);
      // 假装p[j]匹配失败，转移j，然后继续进行匹配
      j = ne[j];
    }
  }
}
```

KMP优化版本

考虑字符串 ，按照上面的算法，我们知道

考虑字符串`p=aaaaa`，按照上面的算法，我们知道：

```
i   1  2  3  4  5
p   a  a  a  a  a
ne  0  1  2  3  4
```

如果该字符串与某串s进行匹配，并且在匹配$p[4+1]=p[5]$的时候失配，那按照上面的算法，接下来要去匹配$p[ne[4]+1]=p[4]$，但是由于$p[4]=p[5]$，我们其实已经知道，如果$p[5]$失配，那么$p[4]$必然失配。那么这个逻辑应该怎么在算法里考虑呢?

我们适当修改ne数组的定义，其最初是由最长相等真前后缀来定义的，同时，$ne[j]$也可以理解为，当$s[i]$和$p[j+1]$失配的时候，如果**只考虑$s[i]$之前字符的匹配**的话，j最大可以跳到哪儿（j最大跳到哪儿，其实也就意味着p向右移动最多可以移动多少步而可以保证不遗漏解）。现在，我们不但要考虑$s[i]$之前的字符的匹配，我们**还要关心$s[i]$的匹配**，并且要排除掉一个显然不匹配的情形，这个情形就是$p[j+1]=p[ne[j]+1]$的情形。在修改定义之后，我们考虑该怎么递推ne。

同样的，假设已知$ne[1:k-1]$，考虑$ne[k]$，和上面一样，我们也是在匹配$p[k]$和$p[ne[k-1]+1]$，$p[ne[ne...[k-1]]+1]$等等。虽然ne的定义改变了，但是这里的“跳”依然是有道理的，只不过这个时候的“跳”，还考虑了$p[k]$的不匹配的特例而已。

如果发生了匹配，在之前的算法里，$ne[k]$直接等于$ne[ne...[k-1]]+1$。但是在这里，我们还要考虑$p[k+1]$的匹配问题，如果$p[k+1]\neq p[ne[ne...[k-1]]+1+1]$，那自然没什么好说的，不会遇到无效跳跃的问题；但是如果$p[k+1]=p[ne[ne...[k-1]]+1+1]$，如果$s[i]$和$p[k+1]$失配，像上面的跳跃方式，我们仅仅能保证$s[i]$之前的字符能匹配，但是一定会有$s[i]\neq p[ne[ne...[k-1]]+1+1]$。所以，我们还需要多一次比较，比较$p[k+1]$和$p[ne[ne...[k-1]]+1+1]$。如果不等，那什么也不用做。如果相等，则还要再多一次跳跃，令$ne[k]=ne[ne[ne...[k-1]]+1]$。根据新的$ne$的定义，这次一定会保证一个不等于$p[k+1]$的字符来和$s$进行匹配。

构造的代码如下：

```
void build_ne() {
  for (int i = 2, j = 0; i <= m; i++) {
    while (j && p[i] != p[j + 1]) j = ne[j];
    if (p[i] == p[j + 1]) j++;
    ne[i] = i < m && p[i + 1] != p[j + 1] ? j : ne[j];
  }
}
```

新的问题来了，优化版本的KMP依然能线性时间内进行多次匹配吗?

答案是不能。原因就在于，原版本的KMP当得到匹配之后，是假定最后一个字符不匹配，然后让p右移（即j进行跳跃）。而在优化版本KMP里，这样的假定所做的跳跃，会将潜在的能匹配的直接略过去了。

KMP算法的演变：KMP自动机

KMP算法的遗产：KMP自动机

自动机的知识之前描述过了。给定一个字符串p，我们考虑构造一个确定性有限状态自动机，使得这个自动机接受且仅接受以p为子串的串。

如果p的长度是m，那我们先有$m+1$个状态，分别是$0,1,2,\dots,m$。容易想到，我们让$\delta(i,p[i+1])=i+1$，即当读到$p[i+1]$这个字符的时候，就让状态转移到编号加1的那个状态上去。0即为起始态，m是接受态。只需要考虑$\delta(i,c),c\neq p[i+1]$的取值就行了，其实这就对应着失配，KMP算法这时候就派上了用场，当发生失配的时候，我们需要去找$p[ne[i]+1]$，$p[ne[ne\dots[i]]+1]$等等进行匹配，一旦匹配，就走到$ne[ne\dots[i]]+1$这个状态上去，如果一直不匹配，那就是停留在状态0上而已。这样我们就构造出了满足条件的自动机。这个自动机当然是用优化版KMP的next数组效率更高，但时间复杂度和用原版的没有区别。

有了这个自动机之后，对于给定的p，和每次询问一个字符串s，我们就能线性的回答s中是否存在p作为子串，以及第一次出现的位置在哪儿。当然如果要回答每次出现的位置在哪儿，需要用原版的KMP。此处不赘述。

作业

所有例题

leetcode 28, 1392