bm 算法与 kmp 算法的性能比较实验报告

2311269 孙威旺

2024 年 10 月 25 日

原始数据：

https://github.com/darkness650/experiment/tree/main/datastruct

一. 算法说明：

KMP 算法：KMP 算法的主要思想是避免检索字符串时在主串中的字符回溯，而是通过已经匹配的部分来调整回溯模式串，而主串指针不断向后进行。而回溯模式串应该回溯的位数则通过 next 数组确定，next 数组是一个与主串无关，只与模式串有关的数组，它里面所存放的数据即当某一位出现匹配错误时应该回溯到的位置。例如串“abcacdab”，如果在加粗的 c 处出现了匹配错误，那模式串就应该回溯到 [1] 的位置，即第二个字符 b 开始匹配。当回溯到模式串头后任无法匹配成功，则向后移动比较下一位，通过这种方式避免对主串的回溯，大幅减少匹配的次数，使效率更高。而 next 数组内的含义则是，在第【i】个数

据前的真前缀与真后缀重叠的长度。任然以上述例子，加粗的 c 以前的重叠的只有 a，即加粗的 c 以前的 a 已经和主串对应了，那么再进行回溯的时候不需要从头开始，回溯到 b 这个位置即可，因为开头 ab 中，a 已经确认是符合的了，直接检验 b 和主串是否对应即可。

BM 算法：BM 算法规定了两个原则，即好后缀原则与坏字符原则。坏字符原则即为，在逆向比对字符串时，若出现失配，则在模式串中找失配的主串的字符，这个字符为坏字符。若没有这个字符，那么整个模式串右移到主串指针的下一位。若有这个字符，就将主串中的字符与此时模式串最右边的字符对齐。好后缀原则即为，在逆向对比字符串时，已经匹配过的子串即为好后缀，在出现失配后，在模式串中寻找字串出现过最右端的位置，跳到那个位置再从最后开始逆向比对即可。利用这两个规则可以跳过很多不必要的对比，从而加快比对速度，减少对比次数，来提高对比效率。

二. 算法核心

KMP 算法中，关键就在于求出模式串对应的 next 数组，即每个字符匹配出错时应该回溯的位置。现假设 next【i】=k，意思就是 p[0]p[1]…p[k-1] 与 p[i-k]p[i k+1]…p[i-1] 是相同的，加入 p[i] 也等于 p[k]，那么显然 next[i+1]=k+1，假如不匹配，那么

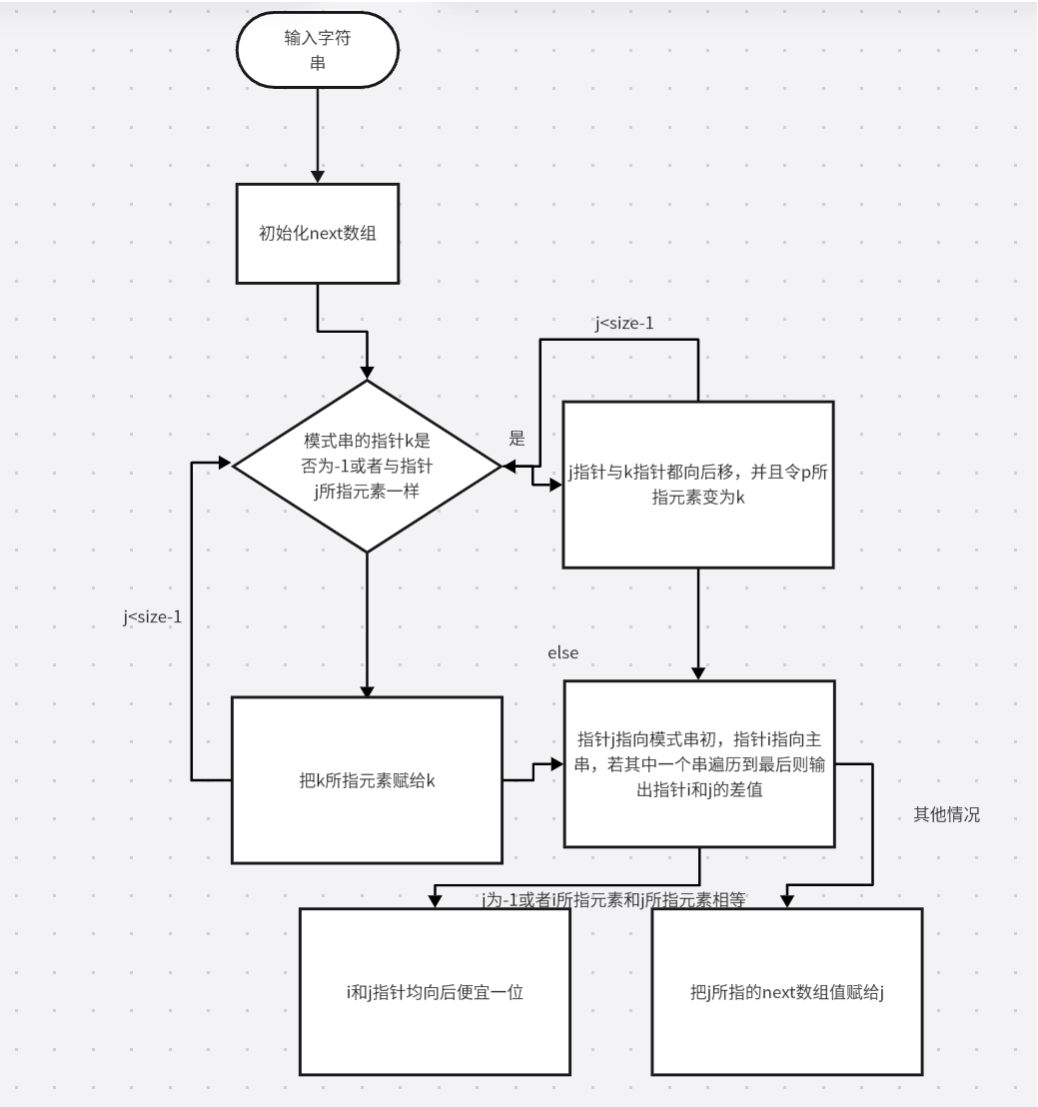
依据上述 next 的意义，应该将 p[k] 回溯到 p[next[k]]，判断是否与 p[i] 相等。重复上述步骤。令 next[0]=-1，即是标记模式串的头，也是便于后面出现头也不相匹配时 ++，且这样得到的数可以直接用于访问数组下标。而 KMP 算法也可以进一步优化，即优化 next 数组，假如 p[k] 与 p[next[k]] 是一样的，那么回溯后一定也是失配的，需要再次回溯，所以出现这种情况，可以让 next[k]=next[next[k]] 来避免。

BM 算法中，关键就是利用好后缀原则与坏字符原则，不难发现，好后缀原则里，通过好后缀的跳跃类似于 KMP 算法的 next 数组，与主串无关，只与模式串有关。例如在 abdabcab 中，c 处匹配出错，则 ab，b 均为好后缀，于是在 c 的前面寻找，则模式串向后移动三位，让 c 前的 ab 与之前匹配成功的 ab 对齐，再从末尾开始匹配。因为这种匹配模式是倒着匹配，所以为了找到对应的数组，可以先确定 SS 数组，即在 [0,i] 区域内与 [0,n-1] 相同的后缀的字符个数，记录给 SS[i]，他的意思就是，从 i 往前数 SS[i] 个数，这 SS[i] 个数是与模式串后的 SS[i] 个数是对应的，若在第 i 个数出现了失配，那么就可以根据 SS 数组来判断应该移动几位。把移动的位数存在 GS 数组中，如果 SS[i]=k，那么就是说，当前的这个数往前的一共 k 个数，与模式串的后 k 个数相同，那么在回溯的时候，在 [0,n-1-k] 之间失配的话，指针向

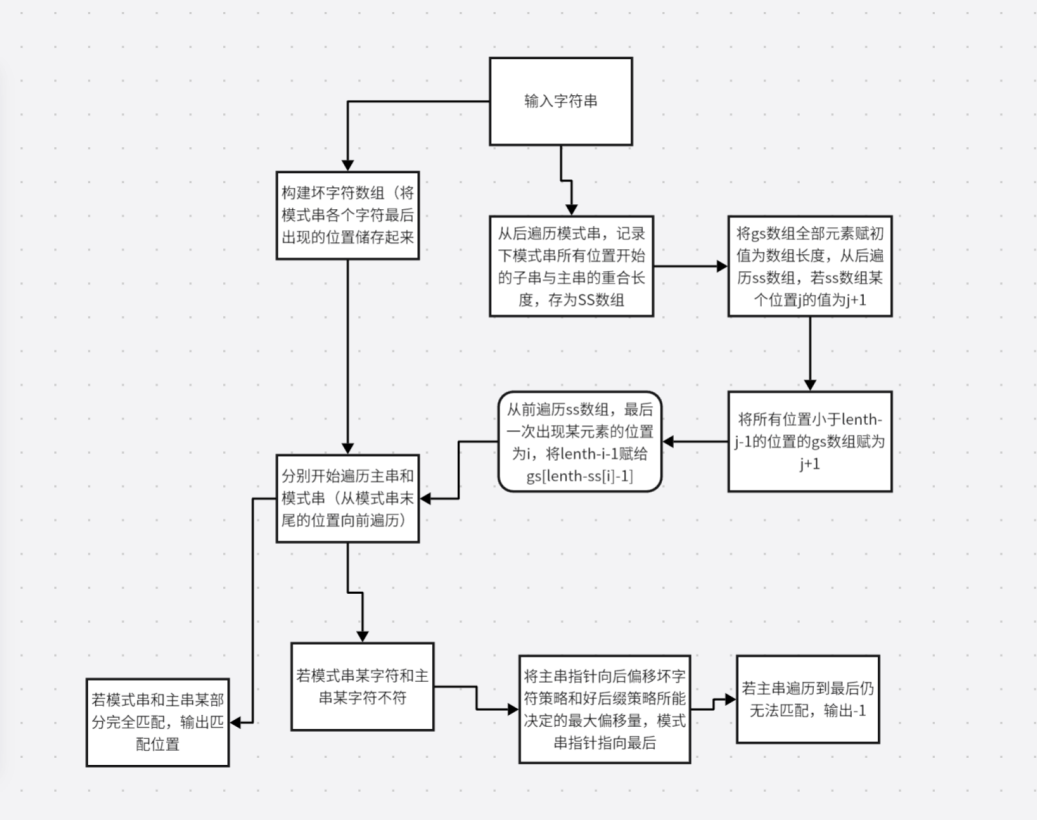
后移动 n-k 位一定是可行的，但是不一定是最安全的，按照这种方法每个计入 GS 的值，取最小值。例如 101011010 中，SS 对应为 020400209，所以在 [0,4] 失配的时候，向后移动 9-4=5 一定是可行的。再假设在 [6] 处失配，即第七个数，现在已经匹配了两个数，而 [6] 与 [1] 对应的都是 2，则应该移动 8-6=2 位，而不是 8-1=7，这样移动的更安全。如果从 SS 数组中去寻找对应规律，设 SS[i]=k，那么在第 n-k 个失配的时候，GS[n-k-1] 应该等于 n-1-i，即最后一个数到这个数的距离。使得移动更安全。

坏字符原则。在模式串与主串匹配时，匹配不上的主串的那个字符即为坏字符。然后从模式串所在的位置向前寻找坏字符，出现坏字符后将这两个坏字符对应在一起，接着向后匹配。若没有出现坏字符，那么需要将整个模式串移动到当前指针的后方，再进行匹配。可以发现坏字符原则移动的位置是与主串有关的。然后对比好后缀与坏字符的到的跳跃步数，选取更多的那个来跳跃。

在对比测试时间的时候,我用的是 QueryPerformanceCounter 来记录时刻，得到毫秒级别的时间准确度。


输入字符串
初始化next数组
模式串的指针k是否为-1或者与指针j所指元素一样
是
j<size-1
j指针与k指针都向后移，并且令p所指元素变为k
j<size-1
else
把k所指元素赋给k
指针j指向模式串初，指针i指向主串，若其中一个串遍历到最后则输出指针i和j的差值
其他情况
j为-1或者i所指元素和j所指元素相等
i和j指针均向后便宜一位
把j所指的next数组值赋给j

kmp 算法流程图



bm 算法流程图

三. 数据分析

附件中的原始数据表是将文首的 github 地址上存储的所有数据整理下来，求时间平均数。而平均性能表的 sheet1 是平均时间性能，单位为毫秒，sheet2 是平均比较次数性能。由于某种原因，平均次数只测了每个组别的第一组数据，但我认为其仍然可以说明以下的问题：

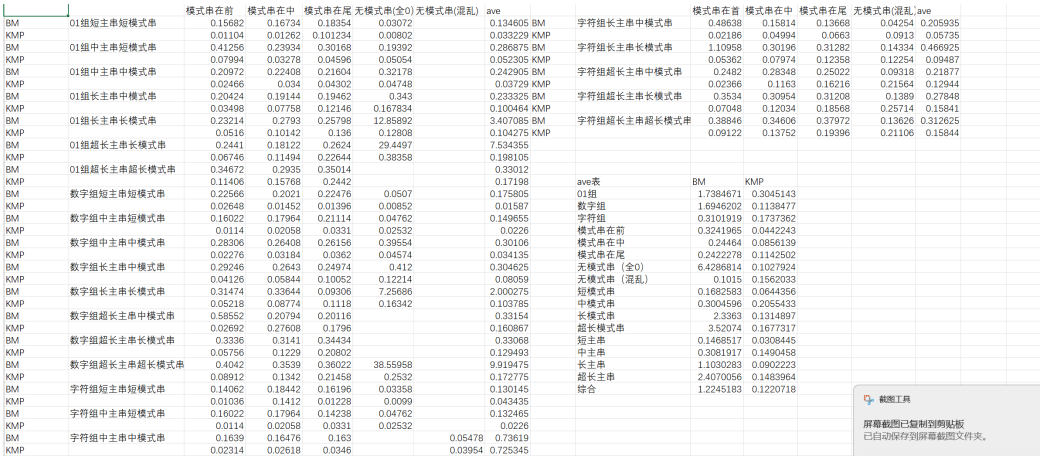
github 上的数据中，对于主串来说，（单位：字符）短：20，

中：500，长：2000，超长：4096（本来准备了 10000 长的数据，但由于 string 一次最多只能输入 4096 故超长串只有 4096）对于模式串来说，短：10 以内，中：200 以内，长：400 以内，超长：800 以上，最大 1000 多。

01 组：主串和模式串全为 0 和 1 组成。

数字组：主串和模式串均由 0 到 9 的数字组成。

字符组：主串和模式串均由大于空格 ASCII 码的字符组成。



		模式串在前	模式串在中	模式串在尾	无模式串(全0)	无模式串(混乱)	ave
BM	01组短主串短模式串	0.15682	0.16734	0.18354	0.03072		0.134605
KMP		0.01104	0.01262	0.101234	0.00802		0.033229
BM	01组中主串短模式串	0.41256	0.23934	0.30168	0.19392		0.286875
KMP		0.07994	0.03278	0.04596	0.05054		0.052305
BM	01组中主串中模式串	0.20972	0.22408	0.21604	0.32178		0.242905
KMP		0.02466	0.034	0.04302	0.04748		0.03729
BM	01组长主串中模式串	0.20424	0.19144	0.19462	0.343		0.233325
KMP		0.03498	0.07758	0.12146	0.167834		0.100464
BM	01组长主串长模式串	0.23214	0.2793	0.25798	12.85892		3.407085
KMP		0.0516	0.10142	0.136	0.12808		0.104275
BM	01组超长主串长模式串	0.2441	0.18122	0.2624	29.4497		7.534355
KMP		0.06746	0.11494	0.22644	0.38358		0.198105
BM	01组超长主串超长模式串	0.34672	0.2935	0.35014			0.33012
KMP		0.11406	0.15768	0.2442			0.17198
BM	数字组短主串短模式串	0.22566	0.2021	0.22476	0.0507		0.175805
KMP		0.02648	0.01452	0.01396	0.00852		0.01587
BM	数字组中主串短模式串	0.16022	0.17964	0.21114	0.04762		0.149655
KMP		0.0114	0.02058	0.0331	0.02532		0.0226
BM	数字组中主串中模式串	0.28306	0.26408	0.26156	0.39554		0.30106
KMP		0.02276	0.03184	0.0362	0.04574		0.034135
BM	数字组长主串中模式串	0.29246	0.2643	0.24974	0.412		0.304625
KMP		0.04126	0.05844	0.10052	0.12214		0.08059
BM	数字组长主串长模式串	0.31474	0.33644	0.09306	7.25686		2.000275
KMP		0.05218	0.08774	0.1118	0.16342		0.103785
BM	数字组超长主串中模式串	0.58552	0.20794	0.20116			0.33154
KMP		0.02692	0.27608	0.1796			0.160867
BM	数字组超长主串长模式串	0.3336	0.3141	0.34434			0.33068
KMP		0.05756	0.1229	0.20802			0.129493
BM	数字组超长主串超长模式串	0.4042	0.3539	0.36022	38.55958		9.919475
KMP		0.08912	0.1342	0.21458	0.2532		0.172775
BM	字符组短主串短模式串	0.14062	0.18442	0.16196	0.03358		0.130145
KMP		0.01036	0.1412	0.01228	0.0099		0.043435
BM	字符组中主串短模式串	0.16022	0.17964	0.14238	0.04762		0.132465
KMP		0.0114	0.02058	0.0331	0.02532		0.0226
BM	字符组中主串中模式串	0.1639	0.16476	0.163		0.05478	0.73619
KMP		0.02314	0.02618	0.0346		0.03954	0.725345

		模式串在首	模式串在中	模式串在尾	无模式串(混乱)	ave
BM	字符组长主串中模式串	0.48638	0.15814	0.13668	0.04254	0.205935
KMP		0.02186	0.04994	0.0663	0.0913	0.05735
BM	字符组长主串长模式串	1.10958	0.30196	0.31282	0.14334	0.466925
KMP		0.05362	0.07974	0.12358	0.12254	0.09487
BM	字符组超长主串中模式串	0.2482	0.28348	0.25022	0.09318	0.21877
KMP		0.02366	0.1163	0.16216	0.21564	0.12944
BM	字符组超长主串长模式串	0.3534	0.30954	0.31208	0.1389	0.27848
KMP		0.07048	0.12034	0.18568	0.25714	0.15841
BM	字符组超长主串超长模式串	0.38846	0.34606	0.37972	0.13626	0.312625
KMP		0.09122	0.13752	0.19396	0.21106	0.15844

ave表	BM	KMP
01组	1.7384671	0.3045143
数字组	1.6946202	0.1138477
字符组	0.3101919	0.1737362
模式串在前	0.3241965	0.0442243
模式串在中	0.24464	0.0856139
模式串在尾	0.2422278	0.1142502
无模式串（全0）	6.4286814	0.1027924
无模式串（混乱）	0.1015	0.1562033
短模式串	0.1682583	0.0644356
中模式串	0.3004596	0.2055433
长模式串	2.3363	0.1314897
超长模式串	3.52074	0.1677317
短主串	0.1468517	0.0308445
中主串	0.3081917	0.1490458
长主串	1.1030283	0.0902223
超长主串	2.4070056	0.1483964
综合	1.2245183	0.1220718

图 1: 时间性能表

在该表中可以看出，BM 算法在除主串中无模式串且主串为长串或者超长串的情况下从运行时间上来说被 kmp 算法全面击败，但这真的是因为 bm 算法本身不行吗？

在该表中我们可以看到，BM 算法在每一项测试用例上的比较次数均小于 kmp 算法，在超长主串，超长模式串，主串中无

		模式串在前	模式串在中	模式串在尾	无模式串(全0)	无模式串(混乱)ave			模式串在首	模式串在中	模式串在尾	无模式串(混乱)ave	
BM	01组短主串短模式串	7	8	18	6	9.75	BM	字符组长主串中模式串	122	132	149	16	104.75
KMP		8	15	33	38	23.5	KMP		185	1397	3821	4002	2351.25
BM	01组中主串短模式串	14	22	46	15	24.25	BM	字符组长主串长模式串	483	492	498	5	369.5
KMP		29	51	112	1000	298	KMP		526	1521	3421	4001	2367.25
BM	01组中主串中模式串	140	289	8		145.6667	BM	字符组超长主串中模式串	122	146	175	35	119.5
KMP		442	747	1000		729.6667	KMP		176	3578	8064	8186	5001
BM	01组长主串中模式串	127	261	446	33	216.75	BM	字符组超长主串长模式串	481	500	519	16	379
KMP		173	1324	3276	3999	2193	KMP		540	3926	7704	8188	5089.5
BM	01组长主串长模式串	501	737	742	2	495.5	BM	字符组超长主串超长模式串	845	857	1087	8	699.25
KMP		543	1569	3270	3999	2345.25	KMP		936	3075	7084	8188	4820.75
BM	01组超长主串长模式串	448	769	1635	7	714.75							
KMP		869	2834	6840	8188	4682.75							
BM	01组超长主串超长模式串	1095	1296	1581		1324							
KMP		1525	4249	6306		4026.667		ave表	BM	KMP			
BM	数字组短主串短模式串	6	8	8	5	6.75		01组	418.66667	1839.6875			
KMP		9	23	35	40	26.75		数字组	293.5	2391.9838			
BM	数字组中主串短模式串	9	79	115	98	75.25		字符组	350.63405	2676.0058			
KMP		20	565	990	1000	643.75		模式串在前	292.69565	380.43478			
BM	数字组中主串中模式串	125	134	158	4	105.25		模式串在中	363.86957	1716.3913			
KMP		170	439	972	1000	645.25		模式串在尾	466.13043	3681.3478			
BM	数字组长主串中模式串	127	188	324	16	163.75		无模式串（全0）	22.769231	2806.9231			
KMP		204	1886	3806	3999	2473.75		无模式串（混乱）	14	5594.1667			
BM	数字组长主串长模式串	487	528	583	3	400.25		短模式串	31.625	177.625			
KMP		575	1752	3466	3999	2448		中模式串	698.58196	1778.6138			
BM	数字组超长主串中模式串	123	285	478		295.3333		长模式串	470.125	3621.4583			
KMP		147	2845	8029		3673.667		超长模式串	912.58333	4609.4722			
BM	数字组超长主串长模式串	484	551	808	4	461.75		短主串	7.6666667	24.833333			
KMP		540	3204	7252	8188	4796		中主串	246.16484	678.45223			
BM	数字组超长主串超长模式串	849	903	1103	3	714.5		长主串	291.75	2363.0833			
KMP		936	3490	7310	8188	4981		超长主串	588.51042	3916.2976			
BM	字符组短主串短模式串	6	7	9	4	6.5		综合	346.02489	2408.4803			
KMP		6	19	32	40	24.25							
BM	字符组中主串短模式串	9	54	106	100	67.25							
KMP		42	546	979	1000	641.75							
BM	字符组中主串中模式串	122	123	125	4	1059.322							
KMP		149	422	869	1000	1112.297							

图 2: 比较次数性能表

模式串的情况下竟然可以以个位数的比较次数结束搜索，这说明在移动模式串的速度上，BM 算法要优于 kmp 算法，然而数字组超长主串超长模式串的模式串全 0 情况下，BM 算法的比较次数仅为个位数，kmp 算法的比较次数超过 8000，但 BM 算法竟然慢到了 38ms，与 kmp 算法的运行时间 0.2532ms 相比慢了 150 倍。

究其原因，bm 算法构建 ss 数组时的 m 的平方数量级的时间复杂度是 bm 算法在运行时间上几乎全面落败于 kmp 算法的主要原因，尤其是在全 0 串这样的构建上体现得尤为明显，因为全 0 串会让 bm 算法构建 ss 数组时完全达到 m 平方数量级的时间复杂度，而其他情况则可能由于模式串子串在最后一个字符就与模式串的最后一个字符不符导致直接终止循环，最好情况下

复杂度为 m 的数量级，这也就是 bm 算法在模式串为完全混乱的在主串中无匹配的情况下运行时间会优胜的重要原因。

综合来看，bm 算法在长主串的搜索中的跳跃速度极快，而 kmp 算法在这方面较为疲软，但在数据量较小的情况下（即使是该文中的超长串也依然属于数据量小的情况），bm 算法的跳跃优势无法完全施展出来，因为 bm 算法构建 ss 串的速度赶不上 kmp 算法跳跃的速度，用句通俗的话来说：或许在 bm 算法还在构建 ss 串的时候 kmp 算法就已经跳跃完毕了。但若是将主串的长度变得非常长，那么这时字符串的“跳跃”速度就会变成两者比较的主要矛盾，尤其是在模式串并不是非常长的时候（这样会让 bm 算法构建 ss 串的劣势降到最低）。

其次，在比较次数表中可知，模式串越是在主串的后面，bm 算法越占优势，若模式串在主串前面，kmp 算法与 bm 算法的比较次数实际上相差不大，由此，我总结出 bm 算法占优的四个条件。

1. 主串的数据量极大的时候。

2. 模式串的数据量不大（相较于主串来说不大即可）的时候。

3. 模式串的位置在主串的后面，甚至不在主串中的时候。

4. 在模式串和主串中都有很多种类的字符的时候

在上述情况下，bm 算法的 ss 串构建劣势将被缩到最小，且跳跃速度的优势被发挥到极大。

四. 反思总结

1.bm 算法有上述的三个条件的约束完全是因为它 ss 数组那 m 平方级的时间复杂度太过于高，在面对长模式串的时候这个劣势会变得无比致命，若是能找到一种线性时间复杂度的构建 ss 数组的方法，那么 bm 算法将会几乎全面优于 kmp 算法（当然，在短串等情况下依然不行，因为 bm 算法不仅要构造 ss 数组，还要构造 gs 数组和坏字符数组，而 kmp 算法只需要构造 next 数组），但这样的话在现实情况里，bm 算法劣势的情况就会变得极为稀少，导致其拥有很高的泛用性

2. 若 bm 算法只采用坏字符策略，从理论上说，它很可能在模式串和主串中有很多类型的字符的时候全方位超过 kmp 算法，因为在有很多种类字符的时候坏字符策略发挥的效能最高，同时没有了构建 ss 数组和 gs 数组的时间耗费，在坏字符策略与 kmp 算法都只构建一个数组的时候坏字符策略极有可能占据绝对优势

3. 当然，kmp 算法也有自己的优势区间，模式串在主串前，模式串中有重复部分的字串很多等时候 kmp 算法都会更优一些，但是若是 bm 算法解决了 ss 数组的问题，这部分的优势区间也

会被 bm 算法挤占，导致 kmp 算法只在极少数情况下占优，想必这也是 kmp 算法在工业上无用武之地的原因

4. 考虑到现实情况里的数据库主串的数量级可能会非常庞大，而模式串相比来说可能只是一些关键词之类的比较小的数量级，导致 bm 算法在实际情况下与如此测出来的情况并不相同，即现实情况中的情况大多属于 bm 算法的优势区间（只与 kmp 算法相比）

五. 测试用代码

详见 github：https://github.com/darkness650/experiment/tree/main/datastruct