

Given a **non-empty** array of numbers, $a_0, a_1, a_2, \dots, a_{n-1}$, where $0 \leq a_i < 2^{31}$.

Find the maximum result of $a_i \oplus a_j$, where $0 \leq i, j < n$.

Could you do this in $O(n)$ runtime?

Example:

Input: [3, 10, 5, 25, 2, 8]

Output: 28

Explanation: The maximum result is $5 \oplus 25 = 28$.

Solution

Overview

Requirements are to have $O(N)$ time complexity, and we'll discuss here two standard approaches to achieve that complexity.

1. Bitwise Prefixes in HashSet.
2. Bitwise Prefixes in Trie.

The idea behind both solutions is the same: to convert all numbers into the binary form, and to construct the maximum XOR bit by bit, starting from the leftmost one. The difference is in the data structure used to store unique bitwise prefixes, i.e. the first i th bits.

The first approach works faster on the given testcase set, but the second one is standard, more simple, and easily generalised for more complex problems like *Find maximum subarray XOR in a given array*.

Prerequisites

XOR of zero and a bit results in that bit

$$0 \oplus x = x$$

XOR of two equal bits (even if they are zeros) results in a zero

$$x \oplus x = 0$$

Approach 1: Bitwise Prefixes in HashSet

Let's start from rewriting all numbers [3, 10, 5, 25, 2, 8] in binary from

$$3 = (00011)_2$$

$$10 = (01010)_2$$

$$5 = (00101)_2$$

$$25 = (11001)_2$$

$$2 = (00010)_2$$

$$8 = (01000)_2$$

To simplify the work with prefixes, better to use the same number of bits L for all the numbers. It's enough to take L equal to the length of the max number in the binary representation.

Now let's construct the max XOR starting from the leftmost bit. The absolute maximum one could have with $L = 5$ bits here is $(11111)_2$. So let's check bit by bit:

- Could we have the leftmost bit for XOR to be equal to 1-bit, i.e. max XOR to be equal to $(1 * ***)_2$?

Yes, for that it's enough to pair $25 = (11001)_2$ with another number starting with the zero leftmost bit. So the max XOR is $(1 * ***)_2$.

- Next step. Could we have max XOR to be equal to $(11 * ***)_2$?

For that, let's consider all prefixes of length 2 and check if there is a pair of them, p_1 and p_2 , such that its XOR is equal to 11 : $p_1 \oplus p_2 == 11$

$$3 = (00 * **)_2$$

$$10 = (01 * **)_2$$

$$5 = (00 * **)_2$$

$$25 = (11 * **)_2$$

$$2 = (00 * **)_2$$

$$8 = (01 * **)_2$$

Yes, it's the case, for example, pair $5 = (00 * **)_2$ and $25 = (11 * **)_2$, or $2 = (00 * **)_2$ and $25 = (11 * **)_2$, or $3 = (00 * **)_2$ and $25 = (11 * **)_2$.

And so on, and so forth. The complexity remains linear. One has to perform N operations to compute prefixes, though the number of prefixes containing $L - i$ bits could not be greater than 2^{L-i} . Hence the check if XOR could have the i th bit to be equal to 1-bit takes $2^{L-i} \times 2^{L-i}$ operations.

Algorithm

- Compute the number of bits L to be used. It's a length of max number in binary representation.
- Initiate `max_xor = 0`.
- Loop from $i = L - 1$ down to $i = 0$ (from the leftmost bit $L - 1$ to the rightmost bit 0):
 - Left shift the `max_xor` to free the next bit.
 - Initiate variable `curr_xor = max_xor | 1` by setting 1 in the rightmost bit of `max_xor`. Now let's check if `curr_xor` could be done using available prefixes.
 - Compute all possible prefixes of length $L - i$ by iterating over `nums`.
 - Put in the hashset `prefixes` the prefix of the current number of the length $L - i$: `num >> i`.
 - Iterate over all prefixes and check if `curr_xor` could be done using two of them: `p1^p2 == curr_xor`. Using self-inverse property of XOR `p1^p2^p2 = p1`, one could rewrite it as `p1 == curr_xor^p2` and simply check for each `p` if `curr_xor^p` is in prefixes. If so, set `max_xor` to be equal to `curr_xor`, i.e. set 1-bit in the rightmost bit. Otherwise, let `max_xor` keep 0-bit in the rightmost bit.
- Return `max_xor`.

```
Java Python Copy
class Solution:
    def findMaximumXOR(self, nums: List[int]) -> int:
        # length of max number in a binary representation
        L = len(bin(max(nums))) - 2
        max_xor = 0
        for i in range(L::-1):
            # go to the next bit by the left shift
            max_xor <<= 1
            # set 1 in the smallest bit
            curr_xor = max_xor | 1
            # compute all existing prefixes
            # of length (L - i) in binary representation
            prefixes = {num >> i for num in nums}
            # update max_xor, if two of these prefixes could result in curr_xor.
            # Check if p1^p2 == curr_xor, i.e. p1 == curr_xor^p2
            max_xor |= any(curr_xor^p in prefixes for p in prefixes)
        return max_xor
```

Complexity Analysis

- Time complexity: $O(N)$. One has to perform N operations to compute prefixes, though the number of prefixes containing $L - i$ bits is 2^{L-i} . Check if XOR could have the i th bit to be equal to 1-bit takes $2^{L-i} \times 2^{L-i}$ operations. Altogether that results in $\sum_{i=0}^{L-1} (N + 4^{L-i}) = NL + \frac{1}{3}(4^L - 1)$ operations, that means $O(N)$ time complexity.
- Space complexity: $O(1)$. One has to keep not more than L prefixes, and $L = 1 + \lceil \log_2 M \rceil$, where M is maximum number in `nums`.

Approach 2: Bitwise Trie

Why HashSet is not a Good Structure to Store Prefixes

HashSet structure, used to store the prefixes in Approach 1, doesn't provide the functionality to cut off some paths which don't lead to the solution.

For example, after two steps of max XOR computation $(11 * **)_2$ it's quite obvious that 25 should be paired with 00 prefix, i.e. with 2, 3, or 5.

$$3 = (00011)_2$$

$$10 = (01010)_2$$

$$5 = (00101)_2$$

$$25 = (11001)_2$$

$$2 = (00010)_2$$

$$8 = (01000)_2$$

Although for the third step we'll again compute all possible prefixes, including the ones for 10 and 8, even if it's quite obvious that they will not lead to the solution.

$$3 = (000 * *)_2$$

$$10 = (010 * *)_2$$

$$5 = (001 * *)_2$$

$$25 = (110 * *)_2$$

$$2 = (000 * *)_2$$

$$8 = (010 * *)_2$$

To cut these branches off, would be great to use some sort of tree structure.

Bitwise Trie: What is it and How to Construct

The standard way is to use **Bitwise Trie**. It's a special type of **Trie**, which is used to store binary prefixes in an efficient way. There are plenty of real-life examples of bitwise trie usage, for example, [in GCC](#).

Let's start with Bitwise Trie for the array [3, 10, 5, 25, 2].

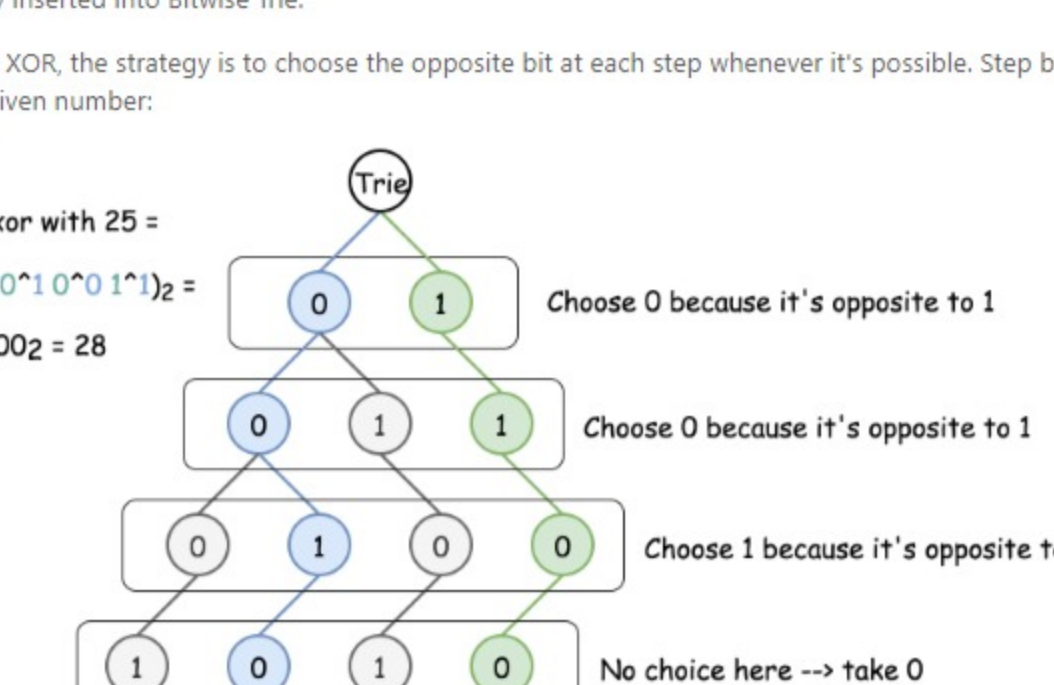
$$3 = (00011)_2$$

$$10 = (01010)_2$$

$$5 = (00101)_2$$

$$25 = (11001)_2$$

$$2 = (00010)_2$$



Each root -> leaf path in Bitwise Trie represents a binary form of a number in `nums`, for example, $0 \rightarrow 0 \rightarrow 0 \rightarrow 1 \rightarrow 1$ is 3. As before, the same number of bits L is used for all numbers, and $L = 1 + \lceil \log_2 M \rceil$, where M is a maximum number in `nums`. The depth of Bitwise Trie is equal to L as well, and all leafs are on the same level.

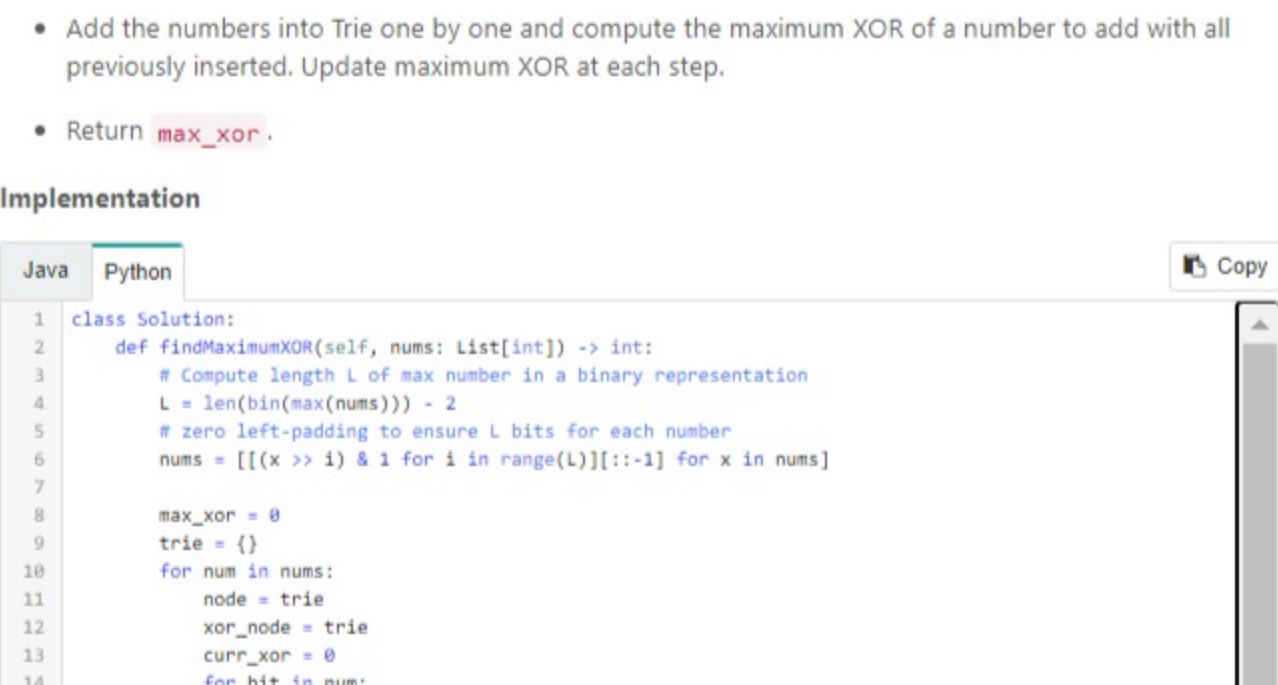
Bitwise Trie is a perfect way to see how different the binary forms of numbers are, for example, 3 and 2 share 4 bits of 5. The construction of Bitwise Trie is pretty straightforward, it's basically nested hashmaps. At each step one has to verify, if the child node to add (0 or 1) is already present. If yes, just go one step down. If not, add it into the Trie and then go one step down.

```
Java Python Copy
1 trie = {}
2 for num in nums:
3     node = trie
4     for bit in num:
5         if not bit in node:
6             node[bit] = {}
7             node = node[bit]
```

Maximum XOR of a Given Number with All Numbers in Trie

Now the Trie is constructed, so let's find the maximum XOR of a given number with all numbers that have been already inserted into Bitwise Trie.

To maximize XOR, the strategy is to choose the opposite bit at each step whenever it's possible. Step by step for 25 as a given number:



The implementation is also pretty simple:

- Try to go down to the opposite bit at each step if it's possible. Add 1-bit at the end of current XOR.
- If not, just go down to the same bit. Add 0-bit at the end of current XOR.

```
Java Python Copy
1 trie = {}
2 for num in nums:
3     xor_node = trie
4     xor_node = {}
5     for bit in num:
6         opp_bit = 1 - bit
7         if opp_bit in xor_node:
8             xor_node = xor_node[opp_bit]
9         else:
10            xor_node = xor_node[bit]
```

Algorithm

To summarise, now one could

- Insert a number into Bitwise Trie.
- Find maximum XOR of a given number with all numbers that have been inserted so far.

That's all one needs to solve the initial problem:

- Convert all numbers to the binary form.
- Add the numbers into Trie one by one and compute the maximum XOR of a number to add with all previously inserted. Update maximum XOR at each step.
- Return `max_xor`.

Implementation

```
Java Python Copy
class Solution:
    def findMaximumXOR(self, nums: List[int]) -> int:
        # Compute length L of max number in a binary representation
        L = len(bin(max(nums))) - 2
        # zero left-padding to ensure L bits for each number
        nums = [(x >> i) & 1 for i in range(L) for x in nums]
        max_xor = 0
        trie = {}
        for num in nums:
            xor_node = trie
            xor_node = {}
            for bit in num:
                # Insert new number in trie
                if not bit in xor_node:
                    xor_node[bit] = {}
                xor_node = xor_node[bit]
            # to compute max xor of that new number
            # with all previously inserted
            toggled_bit = 1 - bit
            if toggled_bit in xor_node:
                curr_xor = xor_node[toggled_bit]
            else:
                curr_xor = xor_node[bit]
            max_xor = curr_xor << 1
```

Complexity Analysis

- Time complexity: $O(N)$. It takes $O(L)$ to insert a number in Trie, and $O(L)$ to find the max XOR of the given number with all already inserted ones. $L = 1 + \lceil \log_2 M \rceil$ is defined by the maximum number in the array and could be considered as a constant here. Hence the overall time complexity is $O(N)$.
- Space complexity: $O(1)$, since one needs at maximum $O(2^L) = O(M)$ space to keep Trie, and L and M could be considered as constants here because of input limitations.

Rate this article: ★★★★★

Previous Next

Comments: 16

Sort By

Type comment here... (Markdown is supported)

Preview Post

AlgorithmImplementer 551 December 15, 2019 5:29 PM
The first solution is cruel. How do people come up with such solution?

SHOW 1 REPLY

s961206 751 October 26, 2019 1:11 PM
How u came up with the idea "zero left-padding", it's fantastic!

4 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63 64 65 66 67 68 69 70 71 72 73 74 75 76 77 78 79 80 81 82 83 84 85 86 87 88 89 90 91 92 93 94 95 96 97 98 99 100 101 102 103 104 105 106 107 108 109 110 111 112 113 114 115 116 117 118 119 120 121 122 123 124 125 126 127 128 129 130 131 132 133 134 135 136 137 138 139 140 141 142 143 144 145 146 147 148 149 150 151 152 153 154 155 156 157 158 159 160 161 162 163 164 165 166 167 168 169 170 171 172 173 174 175 176 177 178 179 180 181 182 183 184 185 186 187 188 189 190 191 192 193 194 195 196 197 198 199 200 201 202 203 204 205 206 207 208 209 210 211 212 213 214 215 216 217 218 219 220 221 222 223 224 225 226 227 228 229 230 231 232 233 234 235 236 237 238 239 240 241 242 243 244 245 246 247 248 249 250 251 252 253 254 255 256 257 258 259 260 261 262 263 264 265 266 267 268 269 270 271 272 273 274 275 276 277 278 279 280 281 282 283 284 285 286 287 288 289 290 291 292 293 294 295 296 297 298 299 300 301 302 303 304 305 306 307 308 309 310 311 312 313 314 315 316 317 318 319 320 321 322 323 324 325 326 327 328 329 330 331 332 333 334 335 336 337 338 339 340 341 342 343 344 345 346 347 348 349 350 351 352 353 354 355 356 357 358 359 360 361 362 363 364 365 366 367 368 369 370 371 372 373 374 375 376 377 378 379 380 381 382 383 384 385 386 387 388 389 390 391 392 393 394 395 396 397 398 399 400 401 402 403 404 405 406 407 408 409 410 411 412 413 414 415 416 417 418 419 420 421 422 423 424 425 426 427 428 429 430 431 432 433 434 435 436 437 438 439 440 441 442 443 444 445 446 447 448 449 450 451 452 453 454 455 456 457 458 459 460 461 462 463 464 465 466 467 468 469 470 471 472 473 474 475 476 477 478 479 480 481 482 483 484 485 486 487 488 489 490 491 492 493 494 495 496 497 498 499 500 501 502 503 504 505 506 507 508 509 510 511 512 513 514 515 516 517 518 519 520 521 522 523 524 525 526 527 528 529 530 531 532 533 534 535 536 537 538 539 540 541 542 543 544 545 546 547 548 549 550 551 552 553 554 555 556 557 558 559 560 561 562 563 564 565 566 567 568 569 570 571 572 573 574 575 576 577 578 579 580 581 582 583 584 585 586 587 588 589 590 591 592 593 594 595 596 597 598 599 600 601 602 603 604 605 606 607 608 609 610 611 612 613 614 615 616 617 618 619 620 621 622 623 624 625 626 627 628 629 630 631 632 633 634 635 636 637 638 639 640 641 642 643 644 645 646 647 648 649 650 651 652 653 654 655 656 657 658 659 660 661 662 663 664 665 666 667 668 669 670 671 672 673 674 675 676 677 678 679 680 681 682 683 684 685 686 687 688 689 690 691 692 693 694 695 696 697 698 699 700 701 702 703 704 705 706 707 708 709 710 711 712 713 714 715 716 717 718 719 720 721 722 723 724 725 726 727 728 729 730 731 732 733 734 735 736 737 738 739 740 741 742 743 744 745 746 747 748 749 750 751 752 753 754 755 756 757 758 759 760 761 762 763 764 765 766 767 768 769 770 771 772 773 774 775 776 777 778 779 780 781 782 783 784 785 786 787 788 789 790 791 792 793 794 795 796 797 798 799 800 801 802 803 804 805 806 807 808 809 810 811 812 813 814 815 816 817 818 819 820 821 822 823 824 825 826 827 828 829 830 831 832 833 834 835 836 837 838 839 840 841 842 843 844 845 846 847 848 849 850 851 852 853 854 855 856 857 858 859 860 861 862 863 864 865 866 867 868 869 870 871 872 873 874 875 876 877 878 879 880 881 882 883 884 885 886 887 888 889 890 891 892 893 894 895 896 897 898 899 900 901 902 903 904 905 906 907 908 909 910 911 912 913 914 915 916 917 918 919 920 921 922 923 924 925 926 927 928 929 930 931 932 933 934 935 936 937 938 939 940 941 942 943 944 945 946 947 948 949 950 951 952 953 954 955 956 957 958 959 960 961 962 963 964 965 966 967 968 969 970 971 972 973 974 975 976 977 978 979 980 981 982 983 984 985 986 987 988 989 990 991 992 993 994 995 996 997 998 999 1000 1001 1002 1003 1004 1005 1006 1007 1008 1009 1010 1011 1012 1013 1014 1015 1016 1017 1018 1019 1020 1021 1022 1023 1024 1025 1026 1027 1028 1029 1030 1031 1032 1033 1034 1035 1036 1037 1038 1039 1040 1041 1042 1043 1044 1045 1046 1047 1048 1049 1050 1051 1052 1053 1054 1055 1056 1057 1058 1059 1060 1061 1062 1063 1064 1065 1066 1067 1068 1069 1070 1071 1072 1073 1074 1075 1076 1077 1078 1079 1080 1081 1082 1083 1084 1085 1086 1087 1088 1089 1090 1091 1092 1093 1094 1095 1096 1097 1098 1099 1100 1101 1102 1103 1104 1105 1106 1107 1108 1109 1110 1111 1112 1113 1114 1115 1116 1117 1118 1119 1120 1121 1122 1123 1124 1125 1126 1127 1128 1129 1130 1131 1132 1133 1134 1135 1136 1137 1138 1139 1140 1141 1142 1143 1144 1145 1146 1147 1148 1149 1150 1151 1152 1153 1154 1155 1156 1157 1158 1159 1160 1161 1162 1163 1164 1165 1166 1167 1168 1169 1170 1171 1172 1173 1174 1175 1176 1177 1178 1179 1180 1181 1182 1183 1184 1185 1186 1187 1188 1189 1190 1191 1192 1193 1194 1195 1196 1197 1198 1199 1200 1201 1202 1203 1204 1205 1206 1207 1208 1209 1210 1211 1212 1213 1214 1215 1216 1217 1218 1219 1220 1221 1222 1223 1224 1225 1226 1227 1228 1229 1230 1231 1232 1233 1234 1235 1236 1237 1238 1239 1240 1241 1242 1243 1244 1245 124