Binary Number with Alternating Bits

Given a positive integer, check whether it has alternating bits: namely, if two adjacent bits will always have different values.

Example 1:

```
Input: 5
Output: True
Explanation:
The binary representation of 5 is: 101
```

Example 2:

```
Input: 7
Output: False
Explanation:
The binary representation of 7 is: 111.
```

Example 3:

```
Input: 11
Output: False
Explanation:
The binary representation of 11 is: 1011.
```

Example 4:

```
Input: 10
Output: True
Explanation:
The binary representation of 10 is: 1010.
```

Solution 1

Why not give the fu*king precise definition of the fuc*king "Alternating bits"?

written by knight-Wang original link here

Solution 2

Solution 1 - Cancel Bits

```
bool hasAlternatingBits(int n) {
    return !((n ^= n/4) & n-1);
}
```

Xor the number with itself shifted right twice and check whether everything after the leading 1-bit became/stayed 0. Xor is 0 iff the bits are equal, so we get 0-bits iff the pair of leading 1-bit and the 0-bit in front of it are repeated until the end.

```
    000101010
  ^ 000001010
  = 000100000
```

Solution 2 - Complete Bits

```
bool hasAlternatingBits(int n) {
    return !((n ^= n/2) & n+1);
}
```

Xor the number with itself shifted right once and check whether everything after the leading 1-bit became/stayed 1. Xor is 1 iff the bits differ, so we get 1-bits iff starting with the leading 1-bit, the bits alternate between 1 and 0.

```
    000101010
  ^ 000010101
  = 000111111
```

Solution 3 - Positive RegEx

```
public boolean hasAlternatingBits(int n) {
    return Integer.toBinaryString(n).matches("(10)*1?");
}
```

It's simple to describe with a regular expression.

Solution 4 - Negative RegEx

```
def has_alternating_bits(n)
  n.to_s(2) !~ /00|11/
end
```

It's even simpler to describe what we **don't** want: two zeros or ones in a row.

Solution 5 - Negative String

```python
def hasAlternatingBits(self, n):
    return '00' not in bin(n) and '11' not in bin(n)
```

Same as before, just not using regex.

Solution 6 - Golfed

```ruby
def has_alternating_bits(n)
  (n^=n/2)&n+1<1
end
```

Solution 7 - Recursion

```ruby
def has_alternating_bits(n)
  n < 3 || n%2 != n/2%2 && has_alternating_bits(n/2)
end
```

Compare the last two bits and recurse with the last bit shifted out.

Solution 8 - Complete Bits + RegEx

```java
public boolean hasAlternatingBits(int n) {
    return Integer.toBinaryString(n ^ n/2).matches("1+");
}
```

written by StefanPochmann original link here

Solution 3

If `n` has alternating bits, then `(n>>1) + n` must be like `111...11`.

Now, let's consider the case when `n` does not have alternating bits, that is, `n` has at least one subsequence with continuous `1` or `0` (we assume `n` has continuous `1` in the after) . We write `n` in its binary format as `xxx011...110xxx`, where `xxx0` and `0xxx` could be empty. Denote `A` as `xxx0`, `B` as `11...11` and `C` as `0xxx`, `n` then can be expressed as `ABC`. We can observe that,

1. If the leftmost bit of `C + C>>1` is `1`, then the leftmost two bits of `C+(1C)>>1` is `10`. E.g., if `C = 011`, then `C + (1C)>>1 = 011 + 101 = 1000`. So `n + (n>>1)` will have a bit with `0`.
2. If the leftmost bit of `C + C>>1` is `0`, then the leftmost two bits of `1C+(11C)>>1` is also `10`. E.g., if C = `010`, then `1C + (11C)>>1 = 1010 + 1101 = 10111`. Note that `B` has a length of at least `2`. So `n + (n>>1)` will also have a bit with `0`.

Similar analysis can be done when `n` has continuous `0`. Therefore, if `n` does not have alternating bits, then `(n>>1) + n` must **Not** be like `111...11`.

At last, for solving this quesiton, we just need to check if `(n>>1) + n + 1` is power of 2.

Java version:

```
public boolean hasAlternatingBits(int n) {
    return ( ((long)n + (n>>1) + 1) & ( (long)n + (n>>1) )) == 0;
}
```

C++ version:

```
bool hasAlternatingBits(int n) {
    return ( ( long(n) + (n>>1) + 1) & ( long(n) + (n>>1) ) ) == 0;
}
```

written by Vincent Cai original link here