Untitled-1

```c
/*
 * floatPower2 - Return bit-level equivalent of the expression 2.0^x
 *   (2.0 raised to the power x) for any 32-bit integer x.
 *
 *   The unsigned value that is returned should have the identical bit
 *   representation as the single-precision floating-point number 2.0^x.
 *   If the result is too small to be represented as a denorm, return
 *   0. If too large, return +INF.
 *
 *   Legal ops: Any integer/unsigned operations incl. ||, &&. Also if, while
 *   Max ops: 30
 *   Rating: 4
 */
unsigned floatPower2(int x) {
  // (1) first check if the result is too large to be represented as a normalized float
  // (2) then check if the reuslt is too small to be represented as a denormal float
  // (3) if the result it neither too small, nor too large, then we calculate and return
the appropriate bit-level equivalent o fthe expression 2.0^x

  // (1)
  // largest normalized < 2 * 2 ^ 127 = 2 ^ 128
  if (x > 127)  return 0xFF << 23;

  // (2)
  // smallest denormalized = 2 ^ (1 - 127) + 2 ^ (-23) = 2 ^ (-149)
  else if (x < -149)  return 0;

  // (3)
  // check whether the number should be represented in as a normalized float or
denormalized float

  // acutal exponent e = E - 127 (for single precision floats)
  // E = e + 127

  // normalized float is simple, we just convert x to E and shift it by 23 bits into the
correct position

  // dormalized float is a bit different, we use the mantissa to represent the value
  // 0.111 gives 1/2 1/4 1/8 ...
  // therefore the position of 1 in the mantissa behind the decimal has the effect of
2^(-x)
  // hence we first convert x to E, E = 1 - 127, therefore E = -126
  // to represent 1 at the proper position, we add 126 to offest E, then add 23 again to
shift it to the left of mantissa

  // smallest normalized = 2 ^ (-126)
  else if (x >= -126)  return (x + 127) << 23;
  // denormalized representation for negative powers
  else return 1 << (x + 126 + 23);
}
```