The ASCII Table

Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char
00	00	NUL	32	20	SP	64	40	@	96	60	‘
01	01	SOH	33	21	!	65	41	A	97	61	a
02	02	STX	34	22	"	66	42	B	98	62	b
03	03	ETX	35	23	#	67	43	C	99	63	c
04	04	EOT	36	24	$	68	44	D	100	64	d
05	05	ENQ	37	25	%	69	45	E	101	65	e
06	06	ACK	38	26	&	70	46	F	102	66	f
07	07	BEL	39	27	’	71	47	G	103	67	g
08	08	BS	40	28	(	72	48	H	104	68	h
09	09	HT	41	29	)	73	49	I	105	69	i
10	0A	LF	42	2A	*	74	4A	J	106	6A	j
11	0B	VT	43	2B	+	75	4B	K	107	6B	k
12	0C	FF	44	2C	,	76	4C	L	108	6C	l
13	0D	CR	45	2D	-	77	4D	M	109	6D	m
14	0E	SO	46	2E	.	78	4E	N	110	6E	n
15	0F	SI	47	2F	/	79	4F	O	111	6F	o
16	10	DLE	48	30	0	80	50	P	112	70	p
17	11	DC1	49	31	1	81	51	Q	113	71	q
18	12	DC2	50	32	2	82	52	R	114	72	r
19	13	DC3	51	33	3	83	53	S	115	73	s
20	14	DC4	52	34	4	84	54	T	116	74	t
21	15	NAK	53	35	5	85	55	U	117	75	u
22	16	SYN	54	36	6	86	56	V	118	76	v
23	17	ETB	55	37	7	87	57	W	119	77	w
24	18	CAN	56	38	8	88	58	X	120	78	x
25	19	EM	57	39	9	89	59	Y	121	79	y
26	1A	SUB	58	3A	:	90	5A	Z	122	7A	z
27	1B	ESC	59	3B	;	91	5B	[	123	7B	{
28	1C	FS	60	3C	<	92	5C	\	124	7C	|
29	1D	GS	61	3D	=	93	5D	]	125	7D	}
30	1E	RS	62	3E	>	94	5E	^	126	7E	~
31	1F	US	63	3F	?	95	5F	_	127	7F	DEL

1 Parentheses (Recursion)

An arithmetic expression may include one or multiple pairs of parentheses, for example

```
(3 + 5) * 4
(3 + (5 + 4) * 6) * 2
3 + (5 + 4) * (6 + 2)
(3 + (5 + 4)) * (6 + 2)
```

This question asks you to write a program that generates pairs of parentheses by following these rules

- The number of left parentheses (`(`) must be the same as number of right parentheses (`)`).
- From left to right, the number of right parentheses (`)`) must not exceed the number of left parentheses (`(`) that have already been seen.

Please fill the following code. The program's ouputs for one, two, three pairs are

```
()

(())
()()

((()))
(()())
(())()
()(())
()()()
```

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  void generate(char * parentheses, int num, int left, int right)
4  {
5    // parentheses: stores the parentheses.
6    // Each element is either ( or )
7    //
8    // num: total number of pairs
9    // left: how many left parentheses have been used
10   // right: how many right parentheses have been used
11   int ind = left + right;
12   if (left == num)   // use up all '('
13     {
```

```
14        for (int i = 0; i < ind; i ++)
15          {
16            printf("%c", parentheses[i]);
17          }
18        // use all remaining ')'
19        for (int i = ????; i ++) // <--- ANSWER A
20          {
21            printf(")");
22          }
23        printf("\n");
24        return;
25      }
26    // case 1: add '('. always possible because left < num
27    parentheses[ind] = '(';
28    generate(parentheses, num, ???); // <--- ANSWER B
29    // case 2: check whether ')' can be added
30    if (????) // <--- ANSWER C
31      {
32        parentheses[ind] = ')';
33        generate(parentheses, num, ????); // <--- ANSWER D
34      }
35  }
36
37  int main(int argc, char * * argv)
38  {
39    if (argc < 2)
40      {
41        return EXIT_FAILURE;
42      }
43    // num: how many pairs
44    int num = (int) strtol(argv[1], NULL, 10);
45    if (num < 1)
46      {
47        return EXIT_FAILURE;
48      }
49    char * parentheses = malloc(sizeof (* parentheses) * num * 2);
50    generate(parentheses, num, 0, 0);
51    free (parentheses);
52    return EXIT_SUCCESS;
53  }
```

Answer:

`A: = right; i < num`

```
B: left + 1, right
C: left > right
D: left, right + 1
```

2 Integer Partition

A positive integer can be expressed as the sum of a sequence of positive integers, or itself. *Integer partition* creates such sequences of integers. For example, 5 can be broken in to the sum of 1 + 2 + 2 or 2 + 3. These two partitions use different numbers, and thus are considered unique partitions. The order of the number in the partition is also important. Thus, 1 + 2 + 2 and 2 + 1 + 2 are considered different partitions because 1 appears in different positions. Below are some examples of integer partitions:

```
1 = 1          2  = 1 + 1        3  = 1 + 1 + 1     4  = 1 + 1+ 1 + 1
                  = 2               = 1 + 2            = 1 + 1 + 2
                                    = 2 + 1            = 1 + 2 + 1
                                    = 3                = 1 + 3
                                                       = 2 + 1 + 1
                                                       = 2 + 2
                                                       = 3 + 1
                                                       = 4
```

In general, number n can be partitioned in 2^{n-1} different ways. For example, 4 can be partitioned in $2^{4-1} = 8$ different ways.
When partitioning 3, "1" is used 5 times and "2" is used twice.
When partitioning 4, "1" is used 12 times and "2" is used 5 times.
1 + 1 + 1 ... will count every occurrence of 1.

2.1 How many 1 is Used

How many times is 1 used when partitioning 7?

2.2 How many 2 is Used

How many times is 2 used when partitioning 6?

2.3 Odd Numbers Only

If only odd numbers (1, 3, 5, 7, 9, ...) are allowed, how many ways can number 8 be partitioned?

2.4 Even Numbers Only

If only even numbers (2, 4, 6, 8, 10, ...) are allowed, how many ways can number 8 be partitioned?
Answer:

```
A 144
```

B 28
C 21
D 8

3 Recursive Program

Consider the following program and write down the answers.

```
 1 #include <stdlib.h>
 2 #include <stdio.h>
 3 #include <time.h>
 4
 5 typedef struct
 6 {
 7   int cnt1;
 8   int cnt2;
 9   int cnt3;
10 } Counters;
11
12 int func(int n, Counters * count)
13 {
14   (count -> cnt1) += n;
15   if (n == 0)
16     {
17       return 0;
18     }
19   (count -> cnt2) ++;
20   int i;
21   int sum = 0;
22   for (i = 1; i < n; i ++)
23     {
24       (count -> cnt3) += i;
25       func(n - i, count);
26       sum += i;
27     }
28   return sum;
29 }
30
31 int main(int argc, char * argv[])
32 {
33   Counters count = // initialize attributes to zero
34     { .cnt1 = 0,
35       .cnt2 = 0,
36       .cnt3 = 0
37     };
38   int result = func(5, & count);
39   printf("result = %d\n", result); // <--- ANSWER A
```

```
40    printf("Counters = %d, %d, %d\n",
41           count.cnt1,
42           count.cnt2,
43           count.cnt3); // <--- ANSWERS B to D
44    return EXIT_SUCCESS;
45  }
```

Answer:
10
31, 16, 26

4 Bit Operations

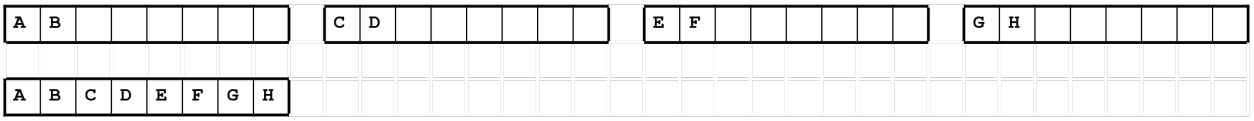

Figure 1: Compress data by taking the leftmost two bits from each byte.

This problem asks you to compress an input file by taking two bits from each byte. If the length of the input file is not a multiple of four, zeros are added at the end of the last byte of the output.
Fill the following code

```
 1 #include <stdio.h>
 2 #include <stdlib.h>
 3 #include <string.h>
 4
 5 // take the leftmost two bits from each byte in orig
 6 // pack these bits to dest
 7 // size is the number of bytes in orig
 8 //
 9 // dest should have enough memory to store the result
10 // convert does not allocate memory in heap
11
12 void convert(const unsigned char * orig,
13              unsigned char * dest,
14              int size)
15 {
16   int bitCount = 0;  // how many bits have been added
17   int destCount = 0; // index for the destination array
18   unsigned char bitMask = ???; // <--- ANSWER A
19   unsigned char destByte = 0;  // destination byte
20   int cnt;
21   for (cnt = 0; cnt < size; cnt ++)
22     {
23       // get the leftmost two bits
24       unsigned char oneBit = ???; // <--- ANSWER B
25
26       // shift right to the correct location
27       unsigned char destBit = ??? // <--- ANSWER C
28
29       // add the bit to the destination
30       destByte = destByte | destBit;
```

```
31
32        // increment bitCount
33        bitCount += 2;
34
35        // fill one byte
36        if (???) // <--- ANSWER D
37          {
38            dest[destCount] = destByte;
39            destCount ++;
40            destByte = 0;
41          }
42      }
43    // If size is not a multliple of 8, handle the last byte
44    if ((size % 4) != 0)
45      {
46        dest[destCount] = destByte;
47        // no need to update destCount since it is the last
48      }
49  }
50
51  int main(int argc, char * argv[])
52  {
53    // argv[1]: input file
54    // argv[2]: output file
55    if (argc < 3)
56      {
57        return EXIT_FAILURE;
58      }
59    FILE * fpin = fopen(argv[1], "r");
60    if (fpin == NULL)
61      {
62        return EXIT_FAILURE;
63      }
64    FILE * fpout = fopen(argv[2], "w");
65    if (fpout == NULL)
66      {
67        fclose (fpin);
68        return EXIT_FAILURE;
69      }
70
71    // find the length of the input file
72    fseek(fpin, 0, SEEK_END);
```

```
 73   long length = ftell(fpin);
 74   // allocate memory for the input
 75   unsigned char * orig = malloc(sizeof(unsigned char) * length);
 76   // return the beginning of the file
 77   fseek(fpin, 0, SEEK_SET);
 78
 79   // read the input from the file
 80   int rtv = fread(orig, sizeof(unsigned char), length, fpin);
 81   if (rtv != length)
 82     {
 83       printf("fread fail, rtv = %d, length = %ld\n",
 84              rtv, length);
 85     }
 86
 87   // calculate the output's length
 88   long outlength = length / 4;
 89   if ((length % 4) != 0)
 90     {
 91       outlength ++;
 92     }
 93
 94   // allocate memory for the output
 95   unsigned char * dest = malloc(sizeof(unsigned char) * outlength);
 96
 97   convert(orig, dest, length);
 98
 99   // write the result to the output file
100   rtv = fwrite(dest, sizeof(unsigned char), outlength, fpout);
101
102   if (rtv != outlength)
103     {
104       printf("fwrite fail, rtv = %d, outlength = %ld\n",
105              rtv, outlength);
106     }
107
108   // close the files
109   fclose (fpin)
110   fclose (fpout);
111   return EXIT_SUCCESS;
112 }
```

For your reference:

```
#include <stdio.h>

size_t fread(void *ptr, size_t size, size_t nmemb, FILE *stream);

size_t fwrite(const void *ptr, size_t size, size_t nmemb,
              FILE *stream);
```

```
DESCRIPTION

       The function fread() reads nmemb items of data, each size bytes
       long, from the stream pointed to by stream, storing them at the
       location given by ptr.

       The function fwrite() writes nmemb items of data, each size
       bytes long, to the stream pointed to by stream, obtaining them
       from the location given by ptr.

       For nonlocking counterparts, see unlocked_stdio(3).

RETURN VALUE

       On success, fread() and fwrite() return the number of items
       read or written.  This number equals the number of bytes
       transferred only when size is 1.  If an error occurs, or the
       end of the file is reached, the return value is a short item
       count (or zero).
```

Answer:

5 IsContained

The following function checks whether every element in array `arr2` is an element in array `arr1`. Their lengths are `length2` and `length1` respectively. Both arrays are sorted in ascending order before calling this function

```
1  // is every element in arr2 also in arr1?
2  // return true if yes
3  // return false if no
4  //
5  // === Example 1 ===
6  // arr1 = {1, 2, 3, 4, 5}, length1 = 5
7  // arr2 = {1, 2, 3}, length = 3
8  // isContained returns true
9  //
10 // === Example 2 ===
11 // arr1 = {1, 2, 3, 4, 5}, length1 = 5
12 // arr2 = {-1, 2, 3}, length = 3
13 // isContained returns false
14 //
15 // === Example 3 ===
16 // arr1 = {1, 2, 3, 4, 5}, length1 = 5
17 // arr2 = {2, 3, 6}, length = 3
18 // isContained returns false
19 //
20 // === Example 4 ===
21 // arr1 = {1, 2, 3, 4, 5}, length1 = 5
22 // arr2 = {2, 5}, length = 2
23 // isContained returns true
24 //
25 // === Example 5 ===
26 // arr1 = {1, 2, 3}, length1 = 3
27 // arr2 = {1, 2, 3, 4, 5}, length = 5
28 // isContained returns false
29
30 // Hint: the following answers are wrong:
31 // compare val1 with length1
32 // compare val1 with length2
33 // compare val2 with length1
34 // compare val2 with length2
35
36 bool isContained(const int * arr1, int length1,
37                  const int * arr2, int length2)
```

```
38 {
39   int ind1 = 0;
40   int ind2 = 0;
41   int val1;
42   int val2;
43   if (length1 < length2)
44     {
45       return false;
46     }
47   while ((ind1 < length1) && (ind2 < length2))
48     {
49       val1 = arr1[ind1];
50       val2 = arr2[ind2];
51       // hint: there is no else
52       if (???) // <--- ANSWER A
53         // hint: val1 != val2 is wrong
54         // length1 < length2 is wrong; it has been checked
55         {
56           return false;
57         }
58       if (???) // <--- ANSWER B
59         {
60           ind1 ++;
61           ind2 ++;
62         }
63       if (???) // <--- ANSWER C
64         {
65           ind1 ++;
66         }
67     }
68   if (???) // <--- ANSWER D
69     // hint: using val1 or val2 is wrong
70     {
71       // some elements in arr2 have not been checked yet
72       return false;
73     }
74   return true;
75 }
```

Answer:

```
val1 > val2
val1 == val2
val1 < val2
```

```
ind2 != length2
```