Tutorial Sheet (November 16, 2018)

ESC101 – Fundamentals of Computing

Announcements

1. **Marks Proration**: Students who have applied for proration of marks in any component of the course should have received an email from the instructor confirming the same. If you have applied for proration but not received a confirmation, contact the instructor immediately.
   There will be no makeup for labs or minor quizzes since we are already taking best 10 of 13 labs and best 8 of 11 quizzes.
2. **End-sem examination**: November 25th Sunday 9AM – 12noon. Venue and seating arrangement will be announced soon. Syllabus for the examination – everything that has been covered up till and including this tutorial.

Sorting and Searching (ask for doubts)

Sorted arrays can allow us to do several operations much faster

1. Search for a given number x in the array
2. Find out how many times (if at all) a given number x occurs
3. Find out the successor/predecessor of a given element x
4. Find the k-th largest/smallest element in the array for a given k

For unsorted arrays, these problems can take $\mathcal{O}(n)$ time. However, for sorted arrays, all these problems can be solved in at most $\mathcal{O}(\log n)$ time

Several techniques exist for sorting arrays that are jumbled up

1. Some guarantee that they will sort an array in $\mathcal{O}(n \log n)$ (maybe on average) no matter what – e.g. Merge Sort, Quick Sort.
2. Some guarantee that they may take at most $\mathcal{O}(n^2)$ time in general, but be much faster if array is almost sorted or if the array is small – e.g. insertion sort

3. Some guarantee *stability* i.e. that they will not change the order of two elements that are identical – e.g. merge sort, insertion sort. Quick sort as we discussed may not be stable but can be made stable with more careful programming.
   Consider the array **5** 6 5 1 3 2 7. The element 5 is repeated and one of its instances is highlighted to distinguish the two.
   Stable Sort: 1 2 3 **5** 5 6 7
   Unstable Sort: 1 2 3 5 **5** 6 7
   The highlighted **5** came before the non-highlighted 5 in the unsorted array so a stable sort preserves that order.
4. Some guarantee *in-place* sorting i.e. the algorithms do not require additional arrays to be used to sort an array e.g. selection sort and carefully programmed quick sort.
5. Some very special algorithms guarantee that they can sort any array with n elements in $\mathcal{O}(n)$ time if the array contains only integers in a small range, say $[0{:}\,k]$. These are called *integer sorting* algorithms – e.g. bucket sort, radix sort.

Bit Representation of Integers

Integers are represented as 4 byte (32 bit) strings inside memory. For example, 38 is represented in binary as 32 + 4 + 2 (as a sum of powers of 2) so its internal representation as an int variable would be
0000 0000 0000 0000 0000 0000 0010 0110

43 is represented in binary as 32 + 8 + 2 + 1 (as a sum of powers of 2) so its internal representation as an int variable would be
0000 0000 0000 0000 0000 0000 0010 1011

Rules of sum and carry for binary strings: same as how we do sums with decimal numbers except that only 2 digits are available – 0 and 1

1. 0 + 0 + 0 (carry) = 0 = 0 carry and 0 sum (1 is represented as 01)
2. 0 + 1 + 0 (carry) = 1 = 0 carry and 1 sum
3. 1 + 0 + 0 (carry) = 1 = 0 carry and 1 sum
4. 1 + 1 + 0 (carry) = 2 = 1 carry and 0 sum (2 is represented as 10)

5. 0 + 0 + 1 (carry) = 1 = 0 carry and 1 sum
6. 0 + 1 + 1 (carry) = 2 = 1 carry and 0 sum
7. 1 + 0 + 1 (carry) = 2 = 1 carry and 0 sum
8. 1 + 1 + 1 (carry) = 3 = 1 carry and 1 sum (3 is represented as 11)

Adding two binary strings: binary strings are added just as decimal numbers are added, using a system of sum and carry

```
  0000 0000   0000 0000   0000 0000   0010 0110 = 38
+ 0000 0000   0000 0000   0000 0000   0010 1011 = 43
  ---------------------------------------------------------------------------
  0000 0000   0000 0000   0000 0000   0101 0001 = 81
  ---------------------------------------------------------------------------

  0000 0000   0000 0000   0000 0000   0010 0110 = 38
+ 0000 0000   0000 0000   0000 0000   0011 1111 = 63
  ---------------------------------------------------------------------------
  0000 0000   0000 0000   0000 0000   0110 0101 = 101
  ---------------------------------------------------------------------------
```

Bitwise Operators:

```
  0000 0000   0000 0000   0000 0000   0010 0110 = 38
& 0000 0000   0000 0000   0000 0000   0000 1111 = 15
  ---------------------------------------------------------------------------
  0000 0000   0000 0000   0000 0000   0000 0110 = 6
  ---------------------------------------------------------------------------

  0000 0000   0000 0000   0000 0000   0010 0110 = 38
| 0000 0000   0000 0000   0000 0000   0000 1111 = 15
  ---------------------------------------------------------------------------
  0000 0000   0000 0000   0000 0000   0010 1111 = 47
  ---------------------------------------------------------------------------

  0000 0000   0000 0000   0000 0000   0010 0110 = 38
^ 0000 0000   0000 0000   0000 0000   0000 1111 = 15
  ---------------------------------------------------------------------------
  0000 0000   0000 0000   0000 0000   0010 1001 = 41
  ---------------------------------------------------------------------------
```

One's and Two's Complement: To find the one's complement of an integer, look at its binary string and then flip each bit.

int a = 1;
0000 0000 0000 0000 0000 0000 0000 0001

a = a << 31;
1000 0000 0000 0000 0000 0000 0000 0000

a = ~a;
0111 1111 1111 1111 1111 1111 1111 1111

To find the two's complement of a number, find its one's complement and then add 1 to that binary string. The binary string that you get will be the negative of the original number with which you started.

int b = ~a + 1;

printf("%d %d", a, b);
2147483647 -2147483647

Subtracting Binary Numbers: two's complement makes subtracting binary numbers exceedingly simple. If you wish to subtract q from p, simple add the two's complement of q to p ☺

int p = 5;
0000 0000 0000 0000 0000 0000 0000 0101
int q = 12;
0000 0000 0000 0000 0000 0000 0000 1100

One's complement of q is
1111 1111 1111 1111 1111 1111 1111 0011
Two's complement of q is
1111 1111 1111 1111 1111 1111 1111 0100

Thus, p – q must be the sum of p and the two's complement of q

```
  0000 0000  0000 0000  0000 0000  0000 0101 = 5
+ 1111 1111  1111 1111  1111 1111  1111 0100 = -12
  ---------------------------------------------------------------------------
  1111 1111  1111 1111  1111 1111  1111 1001 = ?
  ---------------------------------------------------------------------------
```

To find what this mystery number is (it is a negative number since sign bit is 1) simply take its two's complement to find out the positive number.
One's complement of ? is 0000 0000 0000 0000 0000 0000 0000 0110
Two's complement of ? is 0000 0000 0000 0000 0000 0000 0000 0111
Thus, the mystery number is -7 which is correct since indeed 5 – 12 = -7 ☺

Making sense of overflow instances

int c = a + 3;

```
  0111 1111  1111 1111  1111 1111  1111 1111 = 2147483647
+ 0000 0000  0000 0000  0000 0000  0000 0011 = 3
  ---------------------------------------------------------------------------
  1000 0000  0000 0000  0000 0000  0000 0010 = c
  ---------------------------------------------------------------------------
```

c is clearly a negative number (sign bit is 1). To find what c is, take its two's complement to find out what is the positive version of c

~c + 1 = 0111 1111 1111 1111 1111 1111 1111 1110 = 2147483646

printf("%d", c);
-2147483646

int d = a + a;

```
  0111 1111  1111 1111  1111 1111  1111 1111 = 2147483647
+ 0111 1111  1111 1111  1111 1111  1111 1111 = 2147483647
  ---------------------------------------------------------------------------
  1111 1111  1111 1111  1111 1111  1111 1110 = d
  ---------------------------------------------------------------------------
```

~d + 1 = 2 (To find what the negative number d is, negate it)

printf("%d", d);
-2