Number Systems

Introduction

A writing system for expressing numbers. Each number system defines a set of symbols that each represent a specific value.

Base (or radix)

Number of symbols defined by a number system.

Commonly used number systems

- Base 10 - 0 - 9
- Base 2 - $0, 1$
- Base 8 - 0 - 7
- Base 16 - 0 - $9, \mathrm{A}$ - F

> **Caution**
>
> These are required for s1:
>
> - Converting integers and floats between number systems
> - Addition, subtraction, multiplication, division in base 2
>
> But I don’t know how to include it in a easy-to-understand way.

One’s & Two’s Complement

One’s complement

The ones’ complement of a binary number is the value obtained by flipping all the bits in the binary representation of the number.

- If one’s complement of a is b, then one’s complement of b is a.
- Binary representation of $a+b$ will include all 1 s.

One’s complement system

In which negative numbers are represented by the inverse of the binary representations of their corresponding positive numbers. First bit denotes the sign of the number.

- Positive numbers are the denoted as basic binary numbers with 0 as the MSB.
- Negative values are denoted by the one’s complement of their absolute value.

For example, to find the one’s complement system representation of -7, one’s complement of 7 must be found. $7 = 0111_2$. One’s complement of -7 is 1000.

Two’s complement

In which negative numbers are represented using the MSB (sign bit).

If MSB is:

- 1 : negative
- 0 : positive

Positive numbers are represented as basic binary numbers with an additional 0 as the sign bit.

For example:

Following equation can be used to convert a number in two’s complement form to decimal.

$$b = -2^{n-1}b_{n-1} + \sum_{k=0}^{n-2} 2^k b_k$$

Steps

1. Starting with the absolute binary representation of the number
2. Add a leading 0 bit being a sign bit
3. Find the one’s complement: flip all bits (which effectively subtracts the value from -1)
4. Add 1, ignoring any overflows

Floating-point Representation

IEEE 754 standard.

2 types:

- single precision
- double precision

Single precision

Uses 32 bits.

- sign bit - 1 bit
- exponent - 8 bit
- mantissa - 23 bit

Sign bit

0 if positive or zero. 1 if negative.

Exponent

Exponent field range - $[0, 255]$. In this range $[1, 254]$ is defined for normal numbers. 0 and 255 are reserved for subnormal, infinite, signed zeros and NaN.

To support negative exponents, we subtract 127 (half of 254) from this range. $[-126, 127]$. This range is the representable range.

Mantissa

In scientific notation, the part that doesn't contain the base and the power.

In binary scientific notation, there will always be exactly one 1 bit before the dot. So we don't include that one.

ⓘ **Example**

Take 31.3125.

- In binary: 1111.0101_2
- In binary scientific notation: $1.1110101_2 \times 2^3$
- Add 127 to exponent: 130
- Convert exponent to binary 10000010
- Write the final result: $0\ 10000010\ 00000000000000001110101$

Take 0.125.

- In binary: -0.001_2
- In binary scientific notation: $-1.0_2 \times 2^{-3}$
- Add 127 to exponent: 124
- Convert exponent to binary 01111100
- Write the final result: $1\ 01111100\ 00000000000000000000000$

Double precision

Uses 64 bits.

- sign bit - 1 bit
- exponent - 11 bit
- mantissa - 53 bit

Sign bit

0 if positive or zero. 1 if negative.

Exponent

Exponent field range - $[0, 2047]$. In this range $[1, 2046]$ is defined for normal numbers. 0 and 2047 are reserved for subnormal, infinite, signed zeros and NaN.

To support negative exponents, we subtract 1023 (half of 2046) from this range. $[-1022, 1023]$. This range is the representable range.

Mantissa

In scientific notation, the part that doesn't contain the base and the power.

In binary scientific notation, there will always be exactly one 1 bit before the dot. So we don't include that one.

Example

Take 31.3125.

- In binary: 1111.0101_2
- In binary scientific notation: $1.1110101_2 \times 2^3$
- Add 1023 to exponent: 1026
- Convert exponent to binary: 10000000010
- Write the final result: $0\ 10000000010\ 00000000000000000000000000000000000000000000000001110101$

Take 0.125.

- In binary: -0.001_2
- In binary scientific notation: $-1.0_2 \times 2^-3$
- Add 1023 to exponent: 1020
- Convert exponent to binary: 1111111100
- Write the final result: $1\ 1111111100\ 00000000000000000000000000000000000000000000000000000000$

String Representation

A way of representing non-numerical data.

Commonly used encodings

ASCII

Abbreviation for American Standard Code for Information Interchange. Uses 7 bits for letter representation and a parity bit (MSB). Can represent latin alphabet, digits, punctuations, and control characters.

Major limitation in ASCII is it can’t support multiple languages.

Unicode

Uses 32 bits. Supports multiple languages and emojis. Characters are presented by code points. A code point is a integer (in base 16).

Data Types

Data types can be grouped into 3 categories.

Primitive

Data types that are directly supported by a programming languages.

Examples are:

- Boolean
- Characters
- Integers
- Floating-point numbers
- Memory pointers

Composite

Data types that are built as

- structured collections of primitive types
- using other composite types already defined

Examples are:

- Array
- Record or Tuple
- Union

Tuple

Represents a finite ordered list of elements. Can contain different data types. Immutable. Tuple with length n is called as “n-tuple”.

Some tuples have special names:

- length 0 : empty-tuple or null-tuple
- length 1 : singleton
- length 2 : couple
- length 3 : triple

Abstract

Data types that are well defined in terms of properties and operations but not implementation.

Examples:

- List
- Set
- Stack - Last in; first out
- Queue - First in; first out
- Tree
- Hash Table
- Graph

> **Note**
>
> Implementations of stacks, queues, and binary search trees are required in s1.

List

Represents a countable number of values where the same value can occur more than once. Ordered. Can include different data types. Mutable. Aka. iterable collection.

Defined methods:

- isEmpty()
- prepend(item)
- append(item)
- head()
- get(i)
- set(i)
- tail()

> **Note**
>
> Lists in python can be considered as dynamically sized arrays. Methods other than above-mentioned ones are implemented in python.

Set

Represents a collection of distinct objects. Unordered. Iterable. Mutable (but elements must be immutable). No duplicate elements.

Dictionary

Collection of key-value pairs. Unordered.

Tree

Holds a set of nodes. Each node holds a value. Each node can have child nodes.

Binary Tree

Tree with the restriction of at most 2 child nodes per node.

Complete Binary Tree

A binary tree in which every level, except possibly the last, is completely filled, and all nodes are as far left as possible.

Binary Heap

A binary heap is complete binary tree where items are stored in a way such that the value in a parent node is greater/smaller than values in its 2 children nodes. Can be represented by a binary tree or an array. 2 types:

- Max heap: when the parent node value is greater than its children nodes
- Min heap: when the parent node value is smaller than its children nodes

Can be represented by either an array or a binary tree.

Array representation

If a parent node is stored at index i, the left child is stored at index $2i + 1$ and the right child is stored at index $2i + 2$ (assuming the indexing starts at 0).

Space efficient representation.

Algorithms

An algorithm is a finite set of instructions, used to solve a problem.

> **Note**
>
> In s1, only searching and sorting algorithms are discussed.

> **Caution**
>
> Even though an implementation is provided for each algorithm below, the code might have issues.

Searching algorithms

Iterative sequential search

```
def iterative_sequential_search(a_list, item):
    for i in range(len(a_list)):
        if a_list[i] == item:
            return i

    return -1
```

Recursive sequential search

```
def recursive_sequential_search(a_list, item, offset = 0):
    if len(a_list) == 0:
        return False

    if a_list[0] == item:
        return True

    return recursive_sequential_search(a_list[1:], item)
```

Binary search

```
def binary_search(a_list, item):
    first = 0
    last = len(a_list) - 1
    found = False

    while first <= last and not found:
        mid = (first + last) // 2
        if a_list[mid] == item:
            found = True
        else:
            if item < a_list[mid]:
                last = mid - 1 # search in first half
            else:
                first = mid + 1 # search in second half

    if found:
        return mid
    else:
        return -1
```

Time complexities

Algorithms	Best	Average	Worst
Sequential search	O(1)	O(n)	O(n)
Binary search	O(1)	O(log n)	O(log n)

Sorting algorithms

A sorting algorithm reorganizes a collection of items into some order as defined by values intrinsic to the items.

Bubble sort

Makes multiple passes through a collection and compare adjacent items to reorder those that are out of order.

Here is bubble sort algorithm that sorts a list of numbers in-place:

```
def bubble_sort(arr: list[int | float]):
    sorted_index_count = 0
    while sorted_index_count < len(arr):
        for i in range(len(arr)-sorted_index_count-1):
            if arr[i] > arr[i+1]:
                arr[i], arr[i+1] = arr[i+1], arr[i]
        sorted_index_count += 1
```

Selection sort

Keeps track of the position of the highest/smallest value encountered through a pass over the list and after completing the pass perform the swap operation to correctly place the item.

Here is selection sort algorithm that sorts a list of numbers in-place:

```
def selection_sort(arr):
    for current_starting_index in range(len(arr)):
        smallest_index = current_starting_index
        for i in range(current_starting_index + 1, len(arr)):
            if arr[i] < arr[smallest_index]:
                smallest_index = i
        arr[smallest_index], arr[current_starting_index] = arr[current_starting_index], arr[smallest
```

Insertion sort

Maintains a sorted sublist in the lower positions in the list. Each item picked from the unsorted sublist is inserted into the sorted sublist.

```
def insertion_sort(a_list):
    for index in range(1, len(a_list)):
        current_value = a_list[index]
        pos = index
        while pos > 0 and a_list[pos - 1] > current_value:
            a_list[pos] = a_list[pos - 1]
            pos = pos - 1

        a_list[pos] = current_value
```

Shell sort

Breaks the original list into a number of smaller sublists, each of which is sorted using an insertion sort.

```
def gap_insertion_sort(a_list, start, gap):
    for i in range(start + gap, len(a_list), gap):
        current_value = a_list[i]
        pos = i

        while pos >= gap and a_list[pos-gap] > current_value:
            a_list[pos] = a_list[pos - gap]
            pos = pos - gap

        a_list[pos] = current_value

def shell_sort(a_list):
    sublist_count = len(a_list) // 2

    while sublist_count > 0:
        for start_pos in range(sublist_count):
            gap_insertion_sort(a_list,start_pos,sublist_count)

        print("After increments of size", sublist_count,"The list is",a_list)

    sublist_count = sublist_count // 2
```

Merge sort

Recursive algorithm that continually splits a list in half.

- If the list is empty or has one item, it is sorted
- If the list has more elements, the list is split in the middle and merge sort is recursively used on those parts
- Once sorted, the halves are combined to create a new, sorted list

```
def merge_sort(a_list):
    if len(a_list) < 2:
        return

    print("Splitting ", a_list)
    mid = len(a_list) // 2
    left_half = a_list[:mid]
    right_half = a_list[mid:]

    merge_sort(left_half)
    merge_sort(right_half)

    i = 0; j = 0; k = 0
    while i < len(left_half) and j < len(right_half):
        if left_half[i] < right_half[j]:
            a_list[k] = left_half[i]
            i = i + 1
        else:
            a_list[k] = right_half[j]
            j = j + 1;
        k = k + 1

    while i < len(left_half):
        a_list[k] = left_half[i]
        i = i + 1; k = k + 1
    while j < len(right_half):
        a_list[k] = right_half[j]
        j = j + 1; k = k + 1
    print("Merging ", a_list)
```

Quick sort

Recursive algorithm that use the divide and conquer strategy to continually split a list around a selected value called the split point.

- Selects a pivot (a value in the list)
- List is partitioned into 2 parts
    - With the elements lesser than the pivot
    - With the elements greater than the pivot
- The partitions are recursively sorted

```
def quick_sort(a_list, first, last):
    if first < last:
        split_point = partition(a_list, first, last)
        quick_sort(a_list, first, split_point - 1)
        quick_sort(a_list, split_point + 1, last)

def partition(a_list, first, last):
    pivot_value = a_list[first]
    left_mark = first + 1
    right_mark = last
    done = False

    while not done:
        while left_mark <= right_mark and a_list[left_mark] <= pivot_value:
        left_mark = left_mark + 1

        while a_list[right_mark] >= pivot_value and right_mark >= left_mark:
            right_mark = right_mark - 1 36 / 46

            if right_mark < left_mark:
                done = True
            else:
                temp = a_list[left_mark]
                a_list[left_mark] = a_list[right_mark]
                a_list[right_mark] = temp

    temp = a_list[first]
    a_list[first] = a_list[right_mark]
    a_list[right_mark] = temp

    return right_mark
```

Heap sort

Uses a binary heap.

Similar to selection sort where a search is done to find the item with the minimum value and this item is placed at the beginning of the list. The same process is repeated for remaining items.

Steps:

1. A max-heap is built from the input data
2. Largest item is stored at the root of the heap. Replace it with the last item of the heap.
3. Size of the heap is reduced by 1
4. Heapify the root of the tree
5. Repeat steps 2-4 until the size of the heap is greater than 1.

The heapify procedure can be applied to a node only if its children nodes are heapified. So the heapification must be performed in the bottom-up order.

```
# To heapify subtree rooted at index i. Heap size is n.
def heapify(a_list, n, i):
  largest = i # Initialize largest as root
  l = 2 * i + 1 # left = 2*i + 1
  r = 2 * i + 2 # right = 2*i + 2

  # See if left child of root exists and is > root
  if l < n and a_list[i] < a_list[l]:
    largest = l

  # See if right child of root exists and is > root
  if r < n and a_list[largest] < a_list[r]:
    largest = r

  # Change root, if needed
  if largest != i:
    a_list[i],a_list[largest] = a_list[largest],a_list[i] # swap

    # Heapify the root.
    heapify(a_list, n, largest)

def heap_sort(a_list):
  n = len(a_list)

  # Build a maxheap. Since last parent will be
  # at ((n//2)-1) we can start at that location.
  for i in range(n // 2 - 1, -1, -1):
    heapify(a_list, n, i)
```

```
# One by one extract elements
for i in range(n-1, 0, -1):
  a_list[i], a_list[0] = a_list[0], a_list[i] # swap
  heapify(a_list, i, 0)
```

Time complexities

Algorithms	Best	Average	Worst
Bubble sort	O(n)	O(n^2)	O(n^2)
Selection sort	O(n^2)	O(n^2)	O(n^2)
Insertion sort	O(n)	O(n^2)	O(n^2)
Shell sort	O(n)	O((n log n)^2)	O((n log n)^2)
Merge sort	O(n log n)	O(n log n)	O(n log n)
Quick sort	O(n log n)	O(n log n)	O(n^2)
Heap sort	O(n log n)	O(n log n)	O(n log n)

Software Engineering

Software

Refers to all the related things that are required to make a software system work.

Includes:

- programs
- configuration files
- system and user documentation
- user support system
- bug fixes and updates

Software engineering

An engineering discipline that is concerned with all aspects of software production. From the initial stage of writing the requirements to maintaining it while being used.

Software process

Set of activities that are associated with the development of a software product.

Fundamental activities that are common to all types of software development processes:

- Specification - defining the software to be produced and the runtime constraints
- Development - design and development of the software
- Validation - testing phase to check if the software meets the specifications
- Evolution - software is modified to adapt to new specifications

Waterfall

All before-mentioned activities are done sequentially, as clear separate phases. One phase is completed before the next phase is started.

Iterative & incremental

System is developed in iteration. Smaller parts of the system is completed in each iteration, that includes:

- Small amount of requirements specification
- Design and development for the specification
- Validation for the developed parts

Component based

Existing components are combined to implement the system. Main concentration is on the integration of the components.

Quality of software

Can be measured using these aspects:

- Maintainability - how easy it is to making changes
- Dependability - how secure, reliable it is to failures or other unusual activities
- Efficiency - how efficiently hardware resources (such as memory, processor time, disk space) are used
- Usability - how easy it is to use the software from user’s perspective
- Robustness - how resilient it is to invalid inputs

Challenges in software engineering

- Complexity
  - Essential - inherent, difficult to overcome
  - Accidental - not inherent, can be overcome
- Conformity
- Changeability - expected to be changeable to greater extent
- Invisibility - not visualizable
- Can’t guarantee defect free software - no amount of testing can prove absence of defects