B-book

Introduction

Computers are used to process data and produce useful information.

Terminology

Bit

A binary digit. Either a 0 or a 1.

Byte

A collection of 8 bytes. Aka. octet.

Octet vs Byte

In the early days of computing, a byte wasn't always 8 bits - different computers used different byte lengths ranging from 1 to 48 bits. An octet specifically refers to 8 bits and was used to avoid ambiguity, especially in networking protocols. Over time, the 8-bit byte became standard and now byte and octet mean the exact same thing, though octet remains common in networking contexts.

Information

Computers can process various types of data. When a communication link (such as Internet) is provided, the data can be transferred to other users despite of distance. The computer and communication technologies that made this possible are together referred to as information technology or IT (or, sometimes as information and communication technology or ICT). Computers are therefore at the heart of IT.

An information system is a system with well-defined procedures and techniques to collect, store, process, and disseminate information.

Number Systems

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, A - F

Conversion between number systems

10 —> n

Integer part

- Repeatedly divide the number (and the quotients) by n until reaching 1
- Write the remainders in reverse order

Fractional part

- Repeatedly multiply by n until fractional part reaches 0
- Write the integer parts in normal order

n —> 10

Multiply each digit by its positional value, and sum those values. Positional value is n^k where k is the position.

2 —> 8

- Split the given binary number into length 3 parts (prepend 0s if required)
- Convert each part to octal
- Join those together

2 —> 16

- Split the given binary number into length 4 parts (prepend 0s if required)
- Convert each part to hexagonal
- Join those together

16 —> 2

Convert each digit to 4-bit binary and join them together.

8 —> 2

Convert each digit to 3-bit binary and join them together.

8 <—> 16

Convert the number to base 2 or 10 and then conver to the target base.

⚠ **Caution**

These are required in s1:

- Addition, subtraction in base 2, 8, 16
- Multiplication, division in base 2

But I don’t know how to include them in a easy-to-understand way.

Confusion about unit prefixes

In computing, the prefix *kilo* —just like other prefixes— has been used to refer either 2^{10} or 10^3 depending on the context.

- 10^3 - Marketing of disk capacities (by disk manufacturers)
- 2^{10} - Memory capacities, and file sizes, disk capacities by operating systems

To avoid this confusion, 2 unit prefixes are used while measuring amounts of data.

- SI prefixes Defined by ISO. Based on powers of 10^3 . Examples: kilo, mega, giga.
- Binary prefixes Defined by IEC. Based on powers of 2^{10} . Examples: kibi, mebi, gibi.

Data Representation

There are 2 types of data in computers.

Computer memory can be thought of an array of memory cells that each store 1 bit. Total number of bits a memory can hold is limited.

Most Significant Bit (MSB)

In a n-bit memory, memory cell at the $n-1$-th position is the most significant bit. Or left-most bit.

Least Significant Bit (LSB)

In a n-bit memory, memory cell at the 0-th position is the most significant bit. Or right-most bit.

Number of states

A n-bit memory can denote 2^n different states. Each state can be mapped to some information.

Word

In **computing**, a word is a fixed-size datum handled as a unit by the instruction set or hardware of a processor.

Word size

Size of a processor's word.

Representation methods

> **Note**
>
> In s1, representation of integers, floating point numbers, strings are studied.

Numerical

Represents quantifiable and countable things. For example: integers, floating-point numbers.

Integers are considered in 2 sections: signed and unsigned integers.

- Intgers
- Floating-point numbers

Non-numerical

Represents all other data other than numerical. For example: text, images, videos, phone numbers.

- Strings

Integers

Unsigned integers

Unsigned integers can be represented in memory in binary. Only positive integers are supported, by convention.

In n-bit register, 2^n number of integers can be represented, usually in the range $[0, 2^n - 1]$

Signed integers

Both positive and negative integers are supported. There are 2 ways to represent them.

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.

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

Represent n in two's complement

If n is positive or zero: n is converted into binary and mentioned as is.

If n is negative:

1. Starting with the absolute binary representation of n
2. If MSB is not a 0, add a leading 0 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 Numbers

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, 127 (half of 254) is subtracted 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 the inital 1 is not included in the mantissa.

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, 1023 (half of 2046) is subtracted 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:
  $0\ 10000000010\ 00000000000000000000000000000000000000000000001110101$

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:
  $1\ 1111111100\ 000000000000000000000000000000000000000000000000000000$

Strings

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
- Queue
- Tree
- Hash Table
- Graph

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.

Stack

A “Last-In-First-Out” model.

```python
class Stack:
    def __init__(self):
        self.items = [] # to store stack elements

    def is_empty(self):
        # Return True if stack is empty
        return len(self.items) == 0

    def push(self, item):
        # Add item to top of stack
        self.items.append(item)

    def pop(self):
        # Remove and return top item from stack
        if not self.is_empty():
            return self.items.pop()
        return None

    def peek(self):
        # Return top item without removing it
        if not self.is_empty():
            return self.items[-1]
        return None

    def size(self):
        # Return number of items in stack
        return len(self.items)

# Example usage:
# stack = Stack()
# stack.push(1)
# stack.push(2)
# print(stack.pop())  # Returns 2
# print(stack.peek()) # Returns 1
```

Queue

A “First-In-First-Out” model. Implemented in Python as `deque`.

Tree

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

```python
class Tree:
    def __init__(self, value=None):
        # Initialize node with value and empty list of children
        self.value = value
        self.children = []

    def add_child(self, child_node):
        # Add a child node to this node
        self.children.append(child_node)

    def remove_child(self, child_node):
        # Remove a child node from this node
        self.children.remove(child_node)

    def get_value(self):
        # Get the value stored in this node
        return self.value

    def get_children(self):
        # Get list of child nodes
        return self.children

# Example usage:
# tree = Tree(1)
# child1 = Tree(2)
# child2 = Tree(3)
# tree.add_child(child1)
# tree.add_child(child2)
```

Binary Tree

Tree with the restriction of at most 2 child nodes per node. Binary tree can be implemented similarly as a tree class. Instead of a `children` array, `left` and `right` nodes are preferred.

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.

Searching algorithms

Iterative sequential search

```python
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

```python
def recursive_sequential_search(a_list, item, offset=0):
    if len(a_list) == offset - 1:
        return False
    if a_list[offset] == item:
        return True

    return recursive_sequential_search(a_list, item, offset+1)
```

Binary search

Works in a sorted array.

```
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 None
```

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.

Properties

1. Number of swaps required
2. Number of comparisons - represented using “big-o” notation
3. Stability - it’s stable when relative order of the equal items are maintained.
4. Recursive or iterative
5. Amount of extra space

Bubble sort

Makes multiple passes through a collection and compare adjacent items to reorder those.

```
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

Iterates through the list to find the smallest (or highest) value. Swaps its position with the first (or last) element. Then redo this starting for the remaining indices. An improved version of bubble sort.

```
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[smalle
```

Shell sort

A specific “gap” is chosen. Start from any index (which is smaller than gap), and use insertion sort to sort the elements that are gap number of indices away. Redo this after reducing the gap. Repeat until the gap eventually becomes 1.

The performance depends on the sequence of gaps chosen.

```
# a modified version of insertion sort
```

```python
def gap_insertion_sort(a_list, start_index, gap):
    while start_index < len(a_list):
        pointer = start_index
        while pointer >= gap and a_list[pointer - gap] > a_list[pointer]:
            # swap the position
            a_list[pointer], a_list[pointer - gap] = \
                a_list[pointer-gap], a_list[pointer]

            pointer -= gap
        start_index += gap



def shell_sort(a_list):
    for gap in range(4, 0, -1):
        for starting_index in range(0, gap):
            gap_insertion_sort(a_list, starting_index, gap)
```

Merge sort

Recursive algorithm that continually splits a list in half and sorts them.

- 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 sorted list

```python
def merge_sort(a_list):
    if len(a_list) < 2:  # then it's sorted
        return a_list

    # break at the middle and sort
    mid_index = len(a_list)//2
    left_half = merge_sort(a_list[:mid_index])
    right_half = merge_sort(a_list[mid_index:])

    # merge the sides
    cursor_left = 0
    cursor_right = 0
    sorted_list = []

    # merging step 1: loop through each side and add the smallest
    while cursor_left < len(left_half) and cursor_right < len(right_half):
        if left_half[cursor_left] > right_half[cursor_right]:
            sorted_list.append(right_half[cursor_right])
            cursor_right += 1
        else:
            sorted_list.append(left_half[cursor_left])
            cursor_left += 1

    # merging step 2: add left over items
    while cursor_left < len(left_half):
        sorted_list.append(left_half[cursor_left])
        cursor_left += 1
    while cursor_right < len(right_half):
        sorted_list.append(right_half[cursor_right])
        cursor_right += 1

    return sorted_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):
    # Only proceed if there are at least 2 elements to sort
    if first < last:
        # Get the partition point and sort the pivot into its final position
        split_point = partition(a_list, first, last)
        # Recursively sort the left portion (elements smaller than pivot)
        quick_sort(a_list, first, split_point - 1)
        # Recursively sort the right portion (elements larger than pivot)
        quick_sort(a_list, split_point + 1, last)

def partition(a_list, first, last):
    # Choose the first element as the pivot
    pivot_value = a_list[first]
    # Set initial positions for left and right markers
    left_mark = first + 1
    right_mark = last
    done = False

    while not done:
        # Move left marker right until we find an element greater than pivot
        while left_mark <= right_mark and a_list[left_mark] <= pivot_value:
            left_mark = left_mark + 1

        # Move right marker left until we find an element less than pivot
        while a_list[right_mark] >= pivot_value and right_mark >= left_mark:
            right_mark = right_mark - 1

        # If markers have crossed, partitioning is complete
        if right_mark < left_mark:
            done = True
        else:
            # Swap elements at left and right markers since they are in wrong positions
            temp = a_list[left_mark]
            a_list[left_mark] = a_list[right_mark]
            a_list[right_mark] = temp

    # Place pivot in its final position by swapping with right_mark
    temp = a_list[first]
    a_list[first] = a_list[right_mark]
    a_list[right_mark] = temp

    # Return the position of the pivot
    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)

Algorithms	Best	Average	Worst
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