1.1 Algorithms

- An algorithm is a sequence of computational steps that transform the input into the output.
- P5 The statement of the problem specifies in general terms the desired input/output relationship.
- P5 In general, an instance of a problem consists of the input (satisfying whatever constraints are imposed in the problem statement) needed to compute a solution to the problem.
- P6 Which algorithm is best for a given application depends on - among other factors - the number of items to be sorted, possible restrictions on the item values, the architecture of the computer, and the kind of storage devices to be used: main memory, disks, or even tapes.
- An algorithm can be specified in English, as a computer program, or even as a hardware design. The only requirement is that the specification must provide a previse description of the computation procedure to be followed.
- P9 Not every problem solved by algorithms has an easily identified set of candidate solutions (e.g. computing DFT).
- A **data structure** is a way to store and organize data in order to facilitate access and modifications.
- P13 We should consider algorithms, like computer hardware, as a technology. Total system performance depends on choosing efficient algorithms as much as on choosing fast hardware.
- Pay attention to assignment problems in chapter 1

Chapter 2

2.1 Insertion Sort

- P17 Difference between pseudocode from real code
  1. In pseudocode, we employ whatever expressive method is most clear and concise to specify a given algorithm
  2. Pseudocode does not concern with issues of software engineering (e.g. data abstraction, modularity, error handling is ignored)

- Insertion sort is efficient for small input size.
- The insertion sort sorts the input numbers **in place**: it rearranges the numbers within the array A, with at most a constant number of them sorted outside the array at any time.
- P18 INSERTION-SORT
- P20-21 pseudocode convention
- The best case of insertion sort happens when the array is already sorted.
- The worst case happens when the array is in reverse sorted order. That is, worst-case running time is $\theta(n^2)$
- P22 Our pseudocode allow multiple values to be returned in a single return statement.

- The Boolean operators “and” and “or” are short circuiting.
- P21 We treat a variable representing an array or object as a pointer to the data representing the array or object.

2.2 Analyzing algorithms

- P23 For this book, we assume a generic one-processor, **random-access machine (RAM)** model of computation as our implementation technology.
- P23 The RAM model contains instructions commonly found in real computers: arithmetic (e.g. add, subtract, multiply, divide, remainder, floor, ceiling), data movement (load, tore, copy), and control (conditional and unconditional branch, subroutine call and return). Each such instruction takes a constant amount of time

2.3 Designing algorithms

- P30 divide-and-conquer: divide -> conquer -> combine
- Merge sort
- We merge by calling an auxiliary procedure MERGE(A, p,q,r), where A is an array and p,q, and r are indeicdes into the array such that p <= q < r. The procedure assumes that the subarrays A[p,q] and A[q+1 ,r] are in sorted order.
- The MERGE procedure takes time $\theta(n)$ where n = r - p + 1
- P34 MERGE-SORT using MERGE as subroutine
- When an algorithm contains a recursive call to itself, we can often describe its running time by a **recurrence equation** or **recurrence**, which describes the overall running time on a problem of size n <u>in terms of the running time on smaller inputs</u>.
- P35 T(n) is the running time on a problem of size n
- Suppose that our division of the problem yields a subproblems, each of which is 1/b the size of the original. If we take D(n) time to divide the problem into subproblems and C(n) time to combine the solutions to the subproblems into the solution to the original problem, we get the recurrence:

  T(n) = { $\theta(1) \quad if\ n \leq c,$

  $aT\left(\frac{n}{b}\right) + D(n) + C(n) \quad otherwise$

- P36 Because the logarithm function grows more slowly than any linear function, for large enough inputs, merge sort, with its $\theta(n * \lg(n))$ running time, outperforms insertion sort, whose running time is $\theta(n^2)$.

Chapter 3

3.1 Asymptotic notation

- P44 asymptotic notation usually apply to the running time of an algorithm, but it can also apply to some other aspects of algorithms (e.g. the amount of space they use), or even to functions that have nothing to do with algorithms.
- P44

$$\theta\big(g(n)\big) = \{f(n)\text{: there exist positive constants } c_1, c_2, \text{and } n_0 \text{ such that } 0 \le c_1 g(n) \le f(n) \le c_2 g(n) \text{ for all } n \ge n_0\}$$

  Note: θ-notation is an inclusive bound

- θ - notation bounds a function to within constant factors.
- For all n ≥ n_0 in θ-notation, the function f(n) is equal to g(n) to within a constant factor. We say that g(n) is an **asymptotically tight bound** for f(n).
- We assume that every function used within θ-notation is asymptotically non-negative, that is, f(n) and g(n) be non-negative whenever n is sufficiently large. This assumption holds for the other asymptotic notations as well.

- P47 When we only have an **asymptotic upper bound**, we use O-notation.
- $O\big(g(n)\big) = \{f(n)\text{: there exist positive constants } c \text{ and } n_0 \text{ such that } 0 \le f(n) \le cg(n) \text{ for all } n \ge n_0\}$
- We use O-notation to give an upper bound on a function, to within a constant factor.
- O-notation provides asymptotic upper bound, it might not be asymptotically tight bond.
- P47 Using O-notation, we can often describe the running time of an algorithm merely by inspecting the algorithm's overall structure (e.g. double for loop -> O(n^2))
- Just as O-notation provides an asymptotic upper bond on a function, Ω-notation provides an **asymptotic lower bound**.

$$\Omega\big(g(n)\big) = \{f(n)\text{: there exist positive constants } c \text{ and } n_0 \text{ such that } 0 \le cg(n) \le f(n) \text{ for all } n \ge n_0\}$$

- Theorem 3.1
  For any two functions f(n) and g(n), we have f(n) = θ(g(n)) if and only if f(n) = O(g(n)) and f(n) = Ω(g(n)).

- P49 When we say that the running time of an algorithm is Ω(g(n)), we mean that no matter what particular input of size n is chosen for each value of n, the running time on that input is at least a constant times g(n), for sufficiently large n.
- The best case of insertion sort is Ω(n)
- The running time of insertion sort therefore belongs to both Ω(n) and O(n^2), since it falls anywhere between a linear function of n and a quadratic function of n.

- It is not contradictory, however, to say that the worst-case running time of insertion sort is Ω(n^2), since there exists an input that causes the algorithm to take Ω(n^2) time.
- The asymptotic upper bound provided by O-notation may or may not be asymptotically tight.
- We use o-notation to denote an upper bound that is not asymptotically tight.
- $o(g(n)) = \{f(n): for\ any\ positive\ constant\ c > 0,$
  $there\ exists\ a\ constant\ n_0 > 0\ such\ that\ 0\ \leq f(n) < cg(n) for\ al\ n\ \geq n_0\}$

- The difference between O-notation and o-notation is that in f(n) = O(g(n)), the bound 0 ≤ f(n) **≤** cg(n) holds for some constant c > 0, but in f(n) = o(g(n)), the bound 0 ≤ f(n) **<** cg(n) holds for all constants c > 0.
- P50 Intuitively, in o-notation, the function f(n) becomes insignificant relative to g(n) as n approaches infinity:
$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$$
  This above equation is the definition of o-notation
- P51 By analogy, ω-notation is to Ω-notation as o-notation is to O-notation.
- We use ω-notation to denote a lower bound that is not asymptotically tight.
- $\omega(g(n)) = \{f(n): for\ any\ positive\ constant\ c > 0,$
  $there\ exists\ a\ constant\ n_0 > 0\ such\ that\ 0\ \leq cg(n) < \ f(n)\ for\ al\ n\ \geq n_0\}$
- The relation f(n) = ω(g(n)) implies that
$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \ \infty$$

- P51-52 asymptotic comparison rules (Review before exam!)

Chapter 4 Divide-and-Conquer

- P65 A **recurrence** is an equation or inequality that describes a function in terms of its value on smaller inputs.
- P67 If you see recurrences as inequalities, we use O or Ω notation instead of θ-notation.
  e.g. T(n) <= 2T(n/2) + θ(n) -> use O notation
  T(n) >= 2T(n/2) + θ(n) -> use Ω notation

- P67 When we state and solve recurrences, we often omit floors, ceilings, and boundary conditions.
- Recurrence = running time to divide + running time to combine + aT(n/b)

4.2 Strassen's algorithm for matrix multiplication

- Let A = (a_{ij}) and let B = (b_{ij}) are square n*n matrices, suppose C = A*B then
$$c_{ij} = \sum_{k=1}^{n} a_{ik} * b_{kj}$$

- P78 Recurrence for SQUARE-MATRIX-MULTIPLY-RECURSIVE:
  T(n) = θ(1) if n =1
  T(n) = 8T(n/2) + θ(n^2) if n >1

  Note: θ(n^2) because of matrix addition. 8T(n/2) refers to the running time for solving subproblems
  As we shall see from the master's theorem, the above recurrence has the solution T(n) = θ(n^3)

- P79 Although asymptotic notation subsumes constant multiplicative factors, recursive notation such as T(n/2) does not subsume constant multiplicative factor.
- Strassen's method: the idea is to make the recursion tree less bushy. This is done by performing only seven multiplication. The cost of eliminating one matrix multiplication will be several new additions of n/2*n/2 matrices, but still only a constant number of additions.
- P79 read the four steps of Strassen's method
- P80 By the master's theorem, Strassen's method leads to θ($n^{\lg 7}$).

4.3 The substitution method for solving recurrences

- P83 The **substitution method** for solving recurrences comprises two steps:
  1. Guess the form of the solution.
  2. Use mathematical induction to find the constants and show that the solution works.

- P84 Ways to make a good guess:
  1. If a recurrence is similar to one you have seen before, then guessing a similar solution is reasonable.
  2. prove loose upper and lower bounds on the recurrence and then reduce the range of uncertainty

- P85 If you encounter difficulties in proving the guess, try subtracting a lower-order term from our previous guess.
- In substitution method, we must choose the constant c large enough to <u>handle the boundary conditions</u>.
- P87 Sometimes, renaming variables can make an unknown recurrence similar to one you have seen before.

4.4 The recursion-tree method for solving recurrences

- P88 In a **recursion tree**, each node represents the cost of a single subproblem somewhere in the set of recursive function invocations.

- P88 A recursion tree is best used to generate a good guess, which you can then verify by the substitution method. When using a recursion tree to generate a good guess, you can often tolerate a small amount of "sloppiness", since you will be verifying your guess later on.

4.5 The master method for solving recurrence

- P94 Theorem 4.1 (Master Theorem)
- Read the circled paragraph in P94 again
- Read the circled paragraph in P96 again
- P95 Note that the three cases in master theorem do not cover all possibilities for f(n). (read circled paragraph in P95)