Introduction to Sorting

Why learn algorithm and data structure

In general, algorithm is a well-defined procedure with given input and expected output. More concretely, an algorithm defines a set of actions to be taken in a prescribed order that will transform the required inputs to the desired outputs. The ***sorting*** problem and its associated algorithms are often used as an introduction for beginning students, given its relevance in many areas of computer science. The problem is presented as follows:

Input: A sequence A of n numbers $< a_0, a_1, \ldots, a_n >$.
Output: A sequence A' that is a reodering of A such that $a'_0 \leq a'_1 \leq \cdots \leq a'_n$.

The focus of this course, however, is not on the recognition or the memorisation of well-known sorting algorithms (though doing so does not hurt), but on the understanding of the tools for analysing algorithms. As you can tell from experience, solutions in computer science are often not unique: there are many ways to arrive at the same output. However, depending on the context, some solution are "better" than the others. Algorithm analysis allows us to analyse the strengths and weaknesses of different solution, hence to select the right algorithm for a given task.

Chapter layout

We will begin by looking at two variants of sorting algorithms - insertion sort and merge sort. We will then look at some of the tools for analysing algorithms, the first of which is the loop invariant which allows us to show whether an algorithm is correct. The second is running time analysis that introduces us to the big O notation that is often seen in computer science.

Insertion Sort

The algorithm works by initialising two arrays:

- Array A with elements that need to be sorted.
- Array A' intially empty or intialised with the first element of A.

At every loop iteration, we will pick an element to be removed from A and insert it at an appropriate location in A' such that A' will be in a sorted arrangement. This can be done by checking that element (hereon refered as a_i) against elements a'_j starting from the last position (the largest element of A'). If $a_i \geq a'_j$ then a_i is inserted at index $j + 1$, otherwise, j is decreased by one the the process continues until termination, when the condition is met, or when every element in A' has been scanned, in which case a_i is inserted to the first index. At this point, it is useful to think about an efficient way to do insertion. When inserting a_i to index j of A', we are basically doing the following:

- Index incrementation for indices that come after j: $a'_{k+1} = a'_k$ for $k \geq j$
- Assignment: $a'_j = a_i$

A naiive way to implement this is to run the previously described process until the inserting index is found and then do index incrementation and assignment. A more efficient way is to do a swap operation at every comparison step if the comparison is not true: if $a_i < a'_j$ then $a'_{j+1} = a_j$ and $a'_j = a_i$. An illustration for insertion sort is provided as follows: (figure reproduced from CLRS)

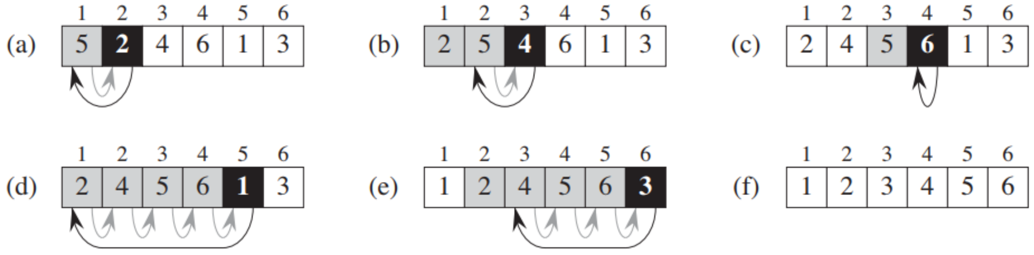

Figure 2.2 The operation of INSERTION-SORT on the array $A = \langle 5, 2, 4, 6, 1, 3 \rangle$. Array indices appear above the rectangles, and values stored in the array positions appear within the rectangles. **(a)–(e)** The iterations of the **for** loop of lines 1–8. In each iteration, the black rectangle holds the key taken from $A[j]$, which is compared with the values in shaded rectangles to its left in the test of line 5. Shaded arrows show array values moved one position to the right in line 6, and black arrows indicate where the key moves to in line 8. **(f)** The final sorted array.

Pseudo-code for Insertion Sort

```
**Insertion Sort Algorithm**
Input: array A of N elements
Output: array A sorted in increasing order
-------------------------------------------
for i from 1 to N-1:
    key = A[i]
    j = i - 1
    while j >= 0 and key < A[j]
        A[j+1] = A[j]
        j--
    A[j+1]=key
```

C/C++ Implementation

```
void insertion_sort_while(int A[], int N){
    for (int i = 1; i < N; i++){
        int key = A[i];
        int j = i-1;
        while (j >= 0 && key < A[j]){
            A[j+1]=A[j];
            j--;
        }
        A[j+1]=key;
    }
}
```

Python Implementation

```
def insertion_sort(A:
    for i in range(1,len(A)):
        key=A[i]
        j = i - 1
        while key < A[j] and j >= 0:
            A[j+1]=A[j]
            j=j-1
        A[j+1]=key
```

Loop invariant

Loop invariant is a framework for proving that an algorithm is correct. An algorithm is correct if for every instance, its termination leads to the desired output. There are three main components of any loop invariant proof:

- **Initialisation**: it is true at the begining of the loop.
- **Maintenance**: if it's true before an iteration of the loop, then it is true before the next iteration begins.
- **Termination**: it remains true when the loop ends.

Proof of Loop Invariant Property

The property that we wants to show is that from initialisation to termination, the sub array A[0:i] is always correctly sorted.

- Initialisation: when i = 1, the array A[0] is correctly sorted.

- Maintenance: when i > 1, A[0:i] is correctly sorted by construction. It is possible to make a proof by contradiction here.
- Termination: the algorithm terminates when it has scanned through all elements of A. Since we have built a sub-array A[0:N] that is correctly sorted, A is then correctly sorted.

Exercise

1.1 Rewrite insertion sort to produce non-increasing instead of non-decreasing order.

```
def reverse_sort(A):
    for i in range(len(A)-2,-1,-1):
        key = A[i]
        j = i + 1
        while (j<len(A) and key < A[j]):
            A[j-1]=A[j]
            j=j+1
        A[j-1]=key
```

1.2 Consider the searching problem:
- **Input**: $A =< a_0, \ldots, a_N >$ and a value v
- **Output**: index i such that $A[i] == v$ or -1 if does not exist.
- Write pseudo code for **linear search** which scans through the sequence looking for v.
- Prove using **loop invariant** that the algorithm is correct.

```
def linear_search(A,v):
    i = -1
    for i in range(len(A)):
        if A[i]==v:
            return i
    return -1
```

```
Proof using loop invariant concept:
We prove that using the linear search algorithm, the correct
answer has already been provided, otherwise -1.
Initialisation: set i = -1: trivially correct since we haven't
found the matching instance.
Maintenance: If at the beginning of each loop, the matching
element/index has not been found, then by the end of the loop
iteration, the index has either been found - returned or set
to -1.
Termination: Also trivial since if the index has been found
```

```
-> Returned, otherwise it has scanned through all elements
without finding the matching index. Hence return -1.
```

1.3 Consider the problem of adding two n-bit integers, stored in two n-element arrays A and B. The sum should be stored in binary form in (n+1) element array C. Write pseudo code.

```
def binary_addition(A,B):
    if len(A) != len(B):
        N = max(len(A),len(B))
        A = np.concatenate([np.zeros(N-len(A)),A],0)
        B = np.concatenate([np.zeros(N-len(B)),B],0)
    N = len(A)
    C = np.zeros(N+1,dtype=np.int8)
    for i in range(N-1,-1,-1):
        C[i+1] = A[i] + B[i]
        if C[i+1] > 1:
            C[i+1] = 0
            C[i] = 1
    return C
```

Some functions for testing:

```
def d2b(d):
    #Converts declimal to binary
    if d == 0:
        return np.array([])
    N = int(np.floor(np.log2(d))+1)
    B = np.zeros(N,dtype=np.int8)
    for i in range(N-1,-1,-1):
        if d >= pow(2,i):
            B[N-1-i]=1
            d-=pow(2,i)
    return B
```

```
def b2d(B):
    #Converts binary to decimal
    N = len(B)
    d = 0
    for i in range(N-1,-1,-1):
        d+= pow(2,N-1-i)*B[i]
    return d
```

Analysing algorithms:

Actual time taken for an execution of an algorithm varies based on the underlying software and hardware. What is often more useful is to think about is the number of calculations that the computer needs to carry out for a given algorithm. This can be measured in the number of multiplication, addition, and other operations carried out, but we can abstract away this concept by thinking about how such number changes with an increase in the amount of input data. Here, analysis is often simplified using the notion of input size. Assuming that each operation costs the computer an amount of c seconds (c is often a random variable), we will show later on that instead of using a messy formulation of computation cost with a lot of c, we can abstract away the idea using the big O notion.

Let's provide an example by analysing the run time of insertion sort:

```
**Insertion Sort Algorithm**
Input: array A of N elements
Output: array A sorted in increasing order
-------------------------------------------
Command:                            Cost:   Times:
for i from 1 to N-1:                c1      N
    key = A[i]                      c2      N-1
    j = i - 1                       c3      N-1
    while j >= 0 and key < A[j]     c4      (sum from 1 to N-1)*t_i
        A[j+1] = A[j]               c5      (sum from 1 to N-1)*(t_i-1)
        j--                         c6      (sum from 1 to N-1)*(t_i-1)
    A[j+1]=key                      c7      N-1
```

Assuming that the for and while loop exit by doing a comparison check (-i.e if the exit condition is met), then the number of times to taken to run through an N-iteration loop is N+1 instead of N like the usual operations. Note that the third column times shows the total number of times that the computer execute this line, hence the times for an inner loop should be multiplied by the number of iterations taken by the outter loops. t_i shows the number of times the inner while loop is executed - i.e. when $i = 1$, the maximum number of backtracks is 1, since j can only take 0. When $i = 3$, the maximum number of backtracks is 3 since j can take $2, 1, 0$. Hence for the worst case analysis, when the inner loop has to run the maximum number of backtracks, $t_i = i$ and we have:

$$
\begin{aligned}
T(N) &= (c_1 + c_2 + c_3 + c_7) * N - (c_2 + c_3 + c_7) + \sum_{1}^{N-1}(c_4 t_i + c_5(t_i - 1) + c_6(t_i - 1)) \\
&= C_1 * N - C_2 + \sum_{i=1}^{N-1}(C_3 i + C_4(i-1))
\end{aligned}
$$

where:

- $C_1 = c_1 + c_2 + c_3 + c_7$
- $C_2 = c_2 + c_3 + c_7$
- $C_3 = c_4$
- $C_4 = c_5 + c_6$

From geometric sum formula, we have:

$$\sum_{1}^{N} i = \frac{N(N+1)}{2}$$

therfore we have:

$$T(N) = C_1 N - C_2 + (C_3 + C_4)\frac{N(N-1)}{2} - C_4(N-1)$$

After some algebra manipulations, we have the following form:

$$T(N) = aN^2 + bN + C$$

We can see that the running time for worst case follows a quadratic form of N. We simplify this notion with the idea of **rate of growth**. The idea can be understood intuitively as follows: as N becomes larger and larger, the N^2 component dominates in value. At this point, we will use the notation $O(N^2)$ to denote the worst-case running time of the insertion sort algorithm. The mechanics behind this notation will be revisited in later chapter. We usually consider one algorithm to be more efficient than the other by comparing their worst case running times. We will also explain some of the caveats behind this analysis in later chapters.

Exercise

1.4 Express $n^3/1000 - 100n^2 - 100n + 3$ in the big O notation:

```
O(N^3)
```

1.5. Consider the following **selection sort** algorithm:

- Sort an array A by first finding the smallest element of A then exchange it with $A[0]$, then second smallest $A[1]$ and continues until $A[n-2]$.
- Write a pseudo code for this algorithm.
- Prove loop invariant property.
- Give the best and worst case analysis.

```python
def selection_sort(A):
    N = len(A)
    for i in range(N-1):
        key = A[i]
        for j in range(i+1,N): #Find current minimum
            if key > A[j]: #Make a swap if minimum found
                temp=key
                key = A[j]
                A[j]= temp
        A[i] = key
```

```
The loop invariant property here is the left section of the array from the
current iterating idex is sorted:
Initialisation: A[-1] is trivially sorted since it is empty.
Maintenance: if A[0:i] is sorted, by the end of the current iteration,
A[0:i+1] is sorted. This is because A[i] = min(A[i:]) should be smaller
than every element of A[0:i]
since A[0] = min(A[0,:]) <= A[1] = min A'[1,:] <= A[2] = min A''[2,:] ...
where A^n denotes the resulting array after n iteration.
Termination: The loop terminates once reaching N-1, when A[0:N-1] is sorted,
and the left over element A[N-1] should be larger than everything in A[0:N-
1]
due to the idea shown above. Hence A is sorted.
```

```
----------------------------------------------------------------------------
-
Command:                            Cost:       Times:
def selection_sort(A):
    N = len(A)                      c1          1
    for i in range(N-1):            c2          N
        key = A[i]                  c3          N-1
        for j in range(i+1,N):      c4          (sum from 1 to N-1)t_i
            if key > A[j]:          c5          (sum from 1 to N-1)(t_i-1)
                temp=key            c6          (sum from 1 to N-1)(t_i-1)
                key = A[j]          c7          (sum from 1 to N-1)(t_i-1)
                A[j]= temp          c8          (sum from 1 to N-1)(t_i-1)
        A[i] = key                  c9          N-1
```

The time taken for c_6, c_7, c_8 components depend on the outcome of the conditional statement in c_5. For worst case analysis, assuming that the array is arranged in reversed order (every condition check is positive), then time calculation:

$$T(N) = c_1 + c_2N + c_3(N-1) + (c_4 + c_9)\sum_{i=1}^{N-1}(N-i) + (c_5 + c_6 + c_7 + c_8)\sum_{i=1}^{N}(N-i-1)$$

After simplifying this expression, the running time is also quadratic of N and is therefore $O(N^2)$. Note that worst and best case running times for selection sort are the same.

1.6 Consider linear search algorithm in previous exercise:

- How many elements in A need to be checked on average, assuming equal likelihood of the search value being in any position in the array. $P(v = A[0]) = P(v = A[1]) = \cdots = \frac{1}{N}$
- What about average case and worst case?

```
Best case: found v after 1st attempt
Worst case: found v after Nth attempt
The average case is a bit tricky:
let X be the random variable denoting the number of attempt taken,
hence we have the probability that X = k - i.e. it takes k attempt to
find
v:
```

$$\begin{aligned} P(k) = P(X = k) &= P(v \neq A[0]) \times P(v \neq A[1]) \times \ldots P(v \neq A[k-2]) \times P(v = A[k-1]) \\ &= \left(\frac{N-1}{N}\right)^{k-1} \times \frac{1}{N} \end{aligned}$$

Hence the expected number of attempts:

$$\begin{aligned} E(X) &= \sum_{i=0}^{N-1} iP(i) = 0 + P(1) + 2 \times P(2) + N \times P(N) \\ &= 0 + \frac{1}{N} + 2 \times \frac{N-1}{N} \times \frac{1}{N} + \cdots + (N-1) \times \left(\frac{N-1}{N}\right)^{N-1} \times \frac{1}{N} \\ &= \frac{1}{N}\sum_{i=1}^{N}\left(\frac{N-1}{N}\right)^{i-1} \times i \end{aligned}$$

As $N \to \infty$, $\frac{N-1}{N} \to 1$, and we can hence simplify the above expression to:

$$E(X) = \frac{1}{N} \times \frac{N(N+1)}{2} = \frac{N+1}{2}$$

A python code with Monte-Carlo simulation:

```
N_array = 100
N_trials = 100000
attempt = 0
```

```
A = np.arange(N_array)
for i in range(N_trials):
    A = np.random.permutation(A) #Shuffle A around
    attempt += linear_search(A,1) + 1 #Plus one since index from 0
attempt /= N_trials #Average number of attempts
```

Divide and Conquer - Merge Sort

So far, the technique that we use to design insertion sort was based on an **incremental** approach, in which we sort an array A by arranging its element one at a time. Another approach **divide and conquer** is based on the idea of breaking down a large problem into smaller and managable chunk to process. At this point, it is also useful to talk about **recursive** algorithms: algorithms that keep calling themselves until the desired solution has been met. Recursive algorithms often follow the divide and conquer approach, in which the original problem is partitioned into sub-problems that are to be solved recursively and the results combined at the end. We will go into greater details on this topic in future chapters. For now, let's present the idea of **merge sort**:

- Divide: divide an array A of n elements into A'_0 and A'_1, each of length $n/2$.
- Conquer: sort the two sub-arrays recursively. If the length of the array is small enough, we can directly solve the problem.
- Combine: merge the two sorted sub-arrays.

Merge Function

It's important to note that the combine step involves more than simply concatenating the two sub-sequence; for example, given two sub-arrays $(1, 3, 7)$ and $(2, 4, 5)$, we also need additional processing step so that the final array $(1, 2, 3, 4, 5, 7)$ is sorted.

Now let's think about how to merge two already sorted sub-arrays, this process involves the following:

- Let i and j be the iterating indices through each sub-arrays. For simplicity, we denote E and F as the sub-arrays, and $i \in \{0, 1, \ldots, N_E - 1\}, j \in \{0, 1, \ldots, N_F - 1\}$.
- Let $S = \{\}$ be the sorted array, with $N_S = N_E + N_F$.
- Initialise $i = 0, j = 0$ and begin to iterate through the index k of S.
- At each iteration of k, set $S[k] = min(E[i], F[j])$, if $E[i] = min(E[i], F[j])$ then increment i by 1. Otherwise, increment j by one. This ensures that as k is incremented, either i or j is incremented, and since k can iterate through a maximum of $N_S = N_E + N_F$ times, both sub-arrays E and F have been scanned by the end of N_S iterations.
- Once one array has been out-of-index (will definitely happen if one sub-array is longer than the other -i.e. F is longer than E), the compiler will return an index-

out-of-bound error if we try to compare $E[N_E]$ and $F[j]$. To avoid this problem, we can either pre-pad each sub-array with a large number so that once E runs out of legitimate values, comparison will be between the legitimate values of F and the very large illegitimate value of E, in which case, values of F should always be selected.

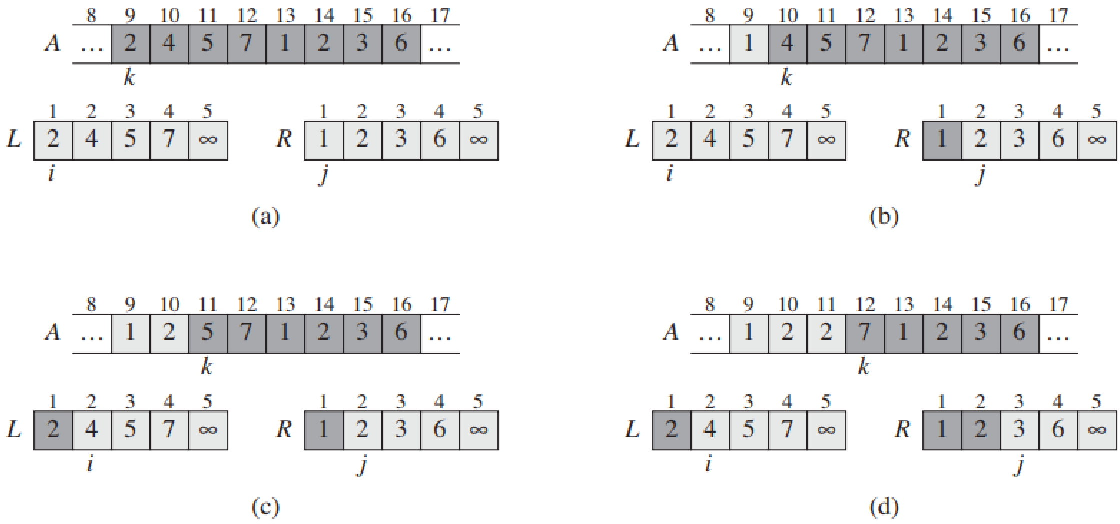

Figure 2.3 The operation of lines 10–17 in the call MERGE(A, 9, 12, 16), when the subarray $A[9..16]$ contains the sequence $\langle 2, 4, 5, 7, 1, 2, 3, 6 \rangle$. After copying and inserting sentinels, the array L contains $\langle 2, 4, 5, 7, \infty \rangle$, and the array R contains $\langle 1, 2, 3, 6, \infty \rangle$. Lightly shaded positions in A contain their final values, and lightly shaded positions in L and R contain values that have yet to be copied back into A. Taken together, the lightly shaded positions always comprise the values originally in $A[9..16]$, along with the two sentinels. Heavily shaded positions in A contain values that will be copied over, and heavily shaded positions in L and R contain values that have already been copied back into A. **(a)–(h)** The arrays A, L, and R, and their respective indices k, i, and j prior to each iteration of the loop of lines 12–17.

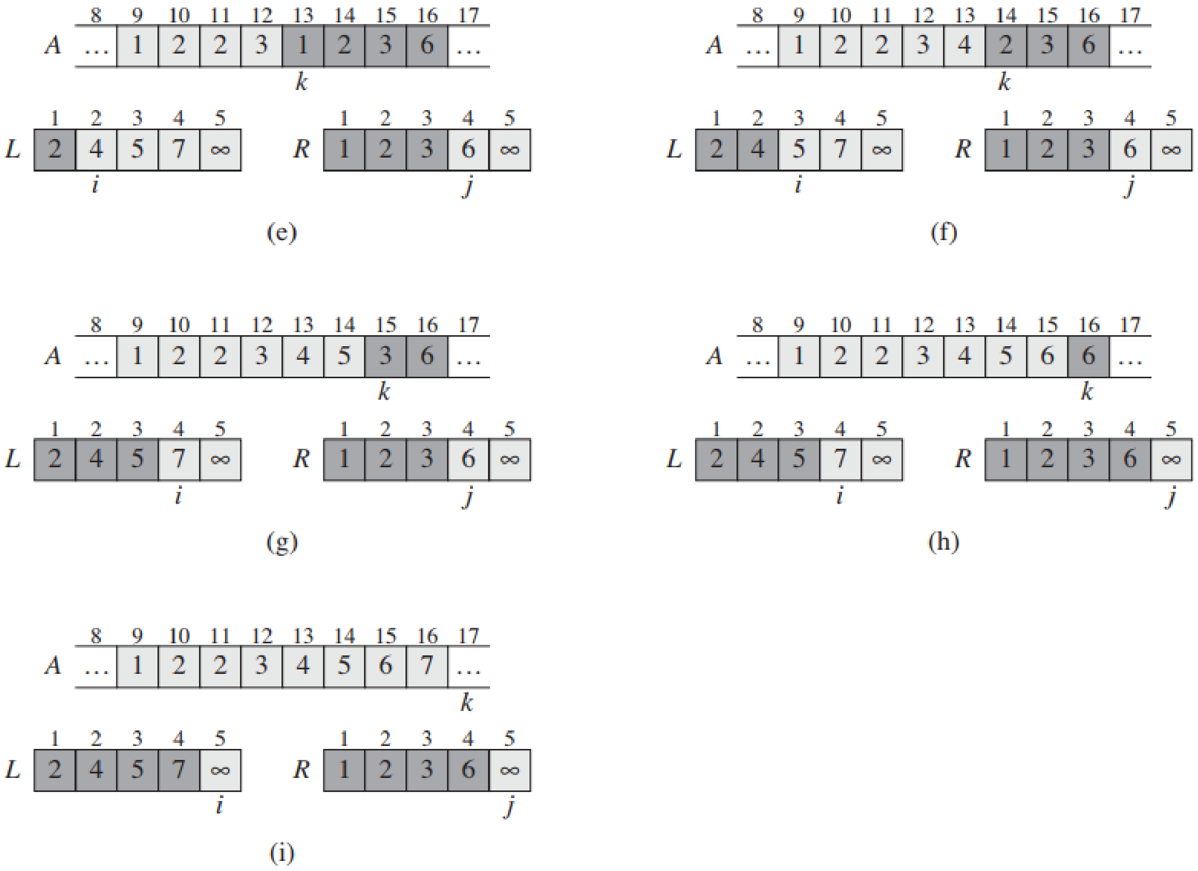

Figure 2.3, continued **(i)** The arrays and indices at termination. At this point, the subarray in $A[9\,..\,16]$ is sorted, and the two sentinels in L and R are the only two elements in these arrays that have not been copied into A.

The code for this process is written as follows:

```
def merge_subarrays(E,F,large_val=np.array([1e8])):
    N_E, N_F = len(E), len(F)
    i,j=0,0
    #Prepading
    E = np.concatenate([E,large_val],0)
    F = np.concatenate([F,large_val],0)
    S = np.zeros(N_E+N_F)

    for k in range(N_E+N_F):
        if E[i]<F[j]:
            S[k]=E[i]
            i+=1
        else:
            S[k]=F[j]
            j+=1
    return S
```

Divide Function

In merge sort, the algorithm bottoms out when the divided sub-array has a length of 1, in which case, is in a correct order. We will see later on that merge-sort can be improved by combining with insertion-sort when the sub-array is small. For now, let's think of a way to implement this procedure:

- Let A be the unsorted array of N elements. Let I_j be the index set whose consecutive elements denote the partition index of A at iteration j. For example:
  - $j = 0:$
    - $I_0 = [0, N]$,
    - $A[0 : N] = A$
  - $j = 1:$
    - $I_1 = [0, \lceil \frac{N}{2} \rceil, N]$
    - $A_0 = A[0 : \lceil \frac{N}{2} \rceil]$
    - $A_1 = A[\lceil \frac{N}{2} \rceil : N]$
  - $j = k:$
    - $I_k = [0, \ldots, \lceil \frac{I_{k-1}[n] + I_{k-1}[n+1]}{2} \rceil, \ldots, N]$
    - $n \in \{0, 1, \ldots, N-1\}$
    - $A_m = A[2I_k[m] : 2I_k[m] + 1]$
    - $m \in \{0, 1, \ldots, 2^k - 1\}$
- If you are overwhelmed at this point by the mathematical notations, do not fret, all of the above just means that at every iteration, we insert the median index between every two elements in the index set. At iteration k, we partition each previous parition into parts that are roughly equal in length. The splitting indices (start-end) are the consecutive elements of the index set I_k. As an example, let N = 5 and $A = [0, 1, 2, 3, 4]$:
  - $j = 0$:
    - $I_0 = [0, 5]$
    - $A_0 = A[0 : 5] = A$
  - $j = 1$:
    - $I_1 = [0, 3, 5]$
    - $A_0 = [0, 1, 2]$
    - $A_1 = [3, 4]$
  - $j = 2$:
    - $I_2 = [0, 2, 3, 4, 5]$
    - $A_0 = [0, 1]$
    - $A_1 = [2]$
    - $A_3 = [3]$
    - $A_4 = [4]$
  - $j = 3$:

    - $I_3 = [0, 1, 2, 3, 3, 4, 4, 5, 5]$
    - $A_0 = [0]$
    - $A_1 = [1]$
    - $A_2 = [2]$
    - $A_3 = []$
    - $A_4 = [3]$
    - $A_5 = []$
    - $A_6 = [4]$
    - $A_7 = []$

- At this point, let's think about how many iterations k are required to split the original array A such that each parition A_m has at most one element (A_m can be empty). It is meaningful to consider k to be first iteration in which each partition has at most one element; obviously if the partitioning process continues, every parition will also has at most one element. Let N_i denotes the length of the longest partition at iteration i -i.e. $N_0 = N$:

  - If A is an even-length array, after each iteration, the length of each partition is exactly half that of its parent: $N_i = N_{i-1}/2$. Hence:

$$\begin{aligned} &N/2^k = 1 \\ &k = \log_2(N) \end{aligned}$$

  - If A is an odd-length array, the first iteration splits A into two partitions of length $(N+1)/2$ and $(N+1)/2 - 1$, the former of which is longer than the later. We can see that if we divide each partition by half, the longest partition at iteration i is always the one derived from the longest partition at iteration $i - 1$. Hence:

$$N_i = \begin{cases} N_{i-1}/2 & \text{if } N_i \text{ is even} \\ (N_{i-1}+1)/2 & \text{otherwise} \end{cases}$$

  - We observe that from the previous definition:

$$\begin{aligned} &2N_i \geq N_{i-1} \quad \text{equality occurs when } N_{i-1} \text{ is even} \\ &2^2 N_i \geq 2N_{i-1} \geq N_{i-2} \\ &2^i N_i \geq N_0 \\ &2^k N_k = 2^k \geq N_0 \end{aligned}$$

  - Additionally $2^{k-1} < N_0$ by construction -i.e. $N_{k-1} > 1$. Hence:

    - $2^k \geq N$
    - $k \geq \log_2 N$, again equality only happen if N_i is even at every step, or N_0 is even.
    - $k = \lceil \log_2(N) \rceil$

- From the result shown, we should run for $\lceil \log_2(N) \rceil$ iterations for the final partitions to be at most one in length. The python script to verify this process is shown:

```
import numpy as np

def split_index_array(A):
    S = np.zeros(2*len(A)-1,dtype=np.int8)
    S[0] = A[0]
    i_S = 1
    for i_A in range(len(A)-1):
        S[i_S] = np.ceil((A[i_A] + A[i_A+1])/2)
        S[i_S+1] = A[i_A+1]
        i_S+=2
    return S

def print_partitions(A,index_array):
    for i in range(len(index_array)-1):
        print(A[index_array[i]:index_array[i+1]])
    print("\n")

N = 5
A = np.arange(N)
index_array = np.array([0,N])
n_iter = int(np.ceil(np.log2(N)))

for j in range(n_iter):
    index_array = split_index_array(index_array)
    print(index_array)
    print_partitions(A,index_array)
```

The code given above started out with the index array that contains just two elements and subsequently modify the index array based on the recurrent relationship described earlier, you will see that this is going to be very useful for recursive function later on. However, at some points, it is useful to have a direct formulation of the index array. The previous scheme can be translated into direct formulation, but I will provide a second index paritioning scheme that is both correct and easier to obtain:

$$I_k[i] = \left\lceil \frac{i \times N}{2^k} \right\rceil$$

This can be implemented very efficiently as follows:

```
def get_index_array(k,N):
    I = np.arange(2**(k+1)+1)/2**(k+1)
    I = np.ceil(I*N)
    return I.astype(int)
```

```
def merge_sort_non_recursive(A):
    n_iter = int(np.ceil(np.log2(N)))
    for i in range(n_iter):
        I = get_index_array(n_iter-i-1,N)
        for j in range(0,len(I)-1,2):
            E = A[I[j]:I[j+1]]
            F = A[I[j+1]:I[j+2]]
            A[I[j]:I[j+2]] = merge_subarrays(E,F)
```

```
def merge_sort_recursive(A,i_start=0,i_end=len(A)):
    i_mid = int(np.ceil((i_start+i_end)/2))
    if (i_end - i_start) > (i_mid-i_start):
        merge_sort_recursive(A,i_start,i_mid)
        merge_sort_recursive(A,i_mid,i_end)
    A[i_start:i_end] = merge_subarrays(A[i_start:i_mid],A[i_mid:i_end])
```

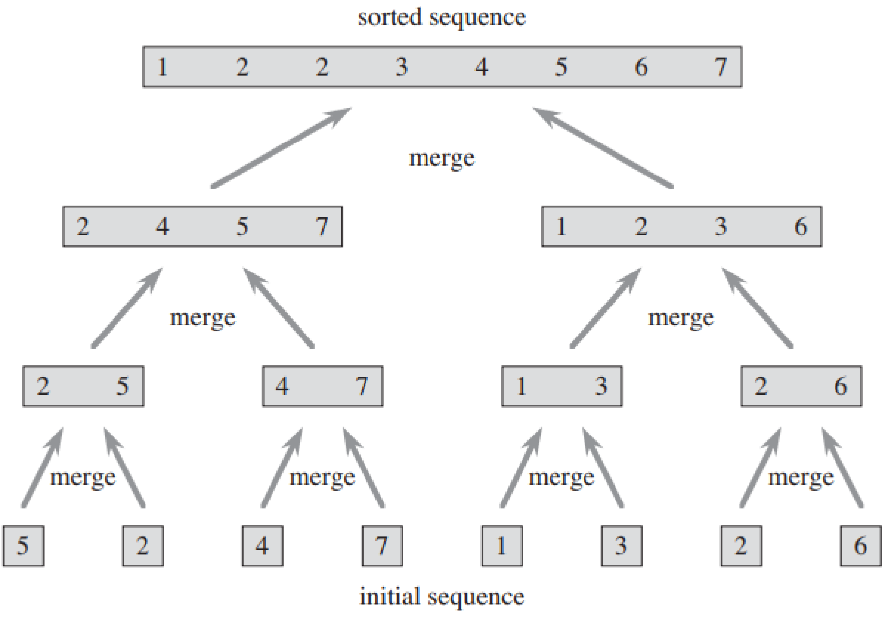

Figure 2.4 The operation of merge sort on the array $A = \langle 5, 2, 4, 7, 1, 3, 2, 6 \rangle$. The lengths of the sorted sequences being merged increase as the algorithm progresses from bottom to top.

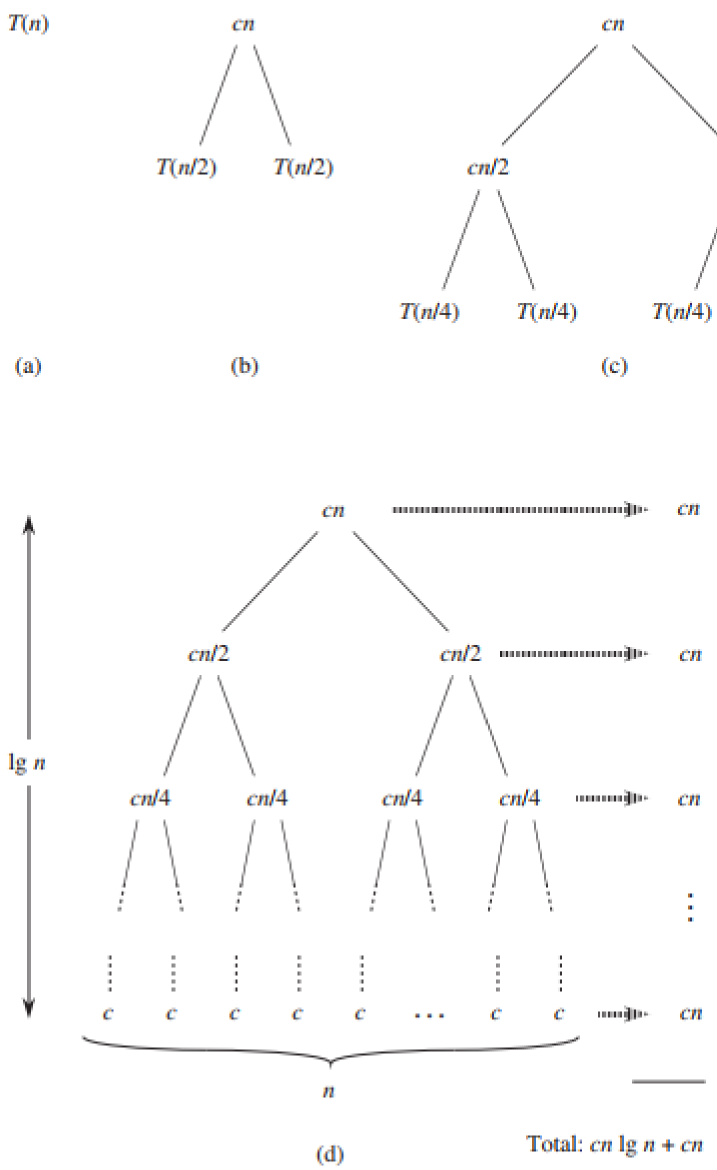

Figure 2.5 How to construct a recursion tree for the recurrence $T(n) = 2T(n/2) + cn$. Part **(a)** shows $T(n)$, which progressively expands in **(b)–(d)** to form the recursion tree. The fully expanded tree in part (d) has $\lg n + 1$ levels (i.e., it has height $\lg n$, as indicated), and each level contributes a total cost of cn. The total cost, therefore, is $cn \lg n + cn$, which is $\Theta(n \lg n)$.

Run time complexity of merge sort

For now, let's just take for granted that the running time complexity of merge sort is $O(n \log n)$. This can be roughly understood by referring to the recursion tree in Figure 2.5 in the book. It takes at most $\log_2(n) + 1$ operations to split the original array to sub-arrays of length one, and when we traverse back the recursion tree from the bottom to the top, each time we merge sub-arrays with a total length of n each time, hence the total time taken for the merge operations in each level is cn (remembering that the merge operation between two sub-arrays of length a and b would require $a + b$ iterations). In summary, there are $\log_2(n) + 1$ levels, each costs cn, hence the total operation costs $cn \log(n) + cn$, hence the time complexity can be taken to be $O(n \log n)$.

The specific mechanism of the big O notation will be discussed in the next chapter, while the derivation of divide and conquer's recurrence relation and time complexity will be discussed in chapter 4.

Exercise

1.7 Use mathematical induction to show that when n is an exact power of 2, the solution to the recurrence:

$$T(n) = \begin{cases} 2 & \text{if } n = 2, \\ 2T(n/2) + n & \text{if } n = 2^k, \text{for } k > 1 \end{cases}$$

is $T(n) = n\log_2(n)$

Solution:
When $n = 2^m$, the solution $T(n) = T(2^m) = n\log_2(n) = 2^m \log_2(2^m) = m2^m$. When $k = 1, T(2^1) = T(2) = 2 = 1 \times 2^1 = 2$, hence the solution is true for $k = 1$. Assuming that the solution is true for some m - i.e. $T(2^m) = m2^m$, we need to show that this implies $T(2^{m+1}) = (m+1)2^{m+1}$ is true. Indeed, we have:

$$T(2^{m+1}) = 2T(2^m) = 2 \times m \times 2^m + 2^{m+1} = 2^{m+1}(m+1)$$

The second equality of which comes from the recurrence relation. Hence If the solution is true for some m, it is also true for $m + 1$...

1.8 We can express insertion sort as a recursive procedure: to sort $A[0:n]$, recursively sort $A[0:n-1]$ then insert $A[n-1]$ into the sorted array. Write a recurrence for this:

1.9 Referring back to the searching problem, observe that if the sequence A is sorted, we can check the mid point of the sequence against the search value v and eliminate half the sequence. The **binary search** algorithm repeats this procedure, halving the size of the search space each time. Write a python code for this procedure and argue that the worst case running time is $O(\log(n))$.

1.10 Observe that the insertion process of insertion sort involves scanning backward the sorted array to find the insertion index using linear search. Is it possible to use binary search to improve the worst case running time of insertion sort to $O(n\log(n))$

1.11 Describe a $O(n\log(n))$ algorithm that, given a set S of n integers and another integer x, determine whether there exist two elements in S whose sum is exactly x.

1.12 Insertion sort on small arrays in merge sort:
Although insertion sort is $O(n^2)$ and merge sort is $O(n \log n)$, when sorting small arrays, insertion sort can be faster than merge sort. Therefore, it makes sense to coarsen the leaves of the recursion by using insertion sort within merge sort when the subproblem becomes small. Consider a modification to merge sort in

which $n = k$ sublists of length k are sorted using insertion sort and then merged using the standard merging mechanism, where k is a value to be determined.

- Show that insertion sort can sort the $n = k$ sublists, each of length k, in $O(nk)$ worst-case time.
- Show how to merge the sublists in $O(n \log(n/k))$ worst-case time.
- Given that the modified algorithm runs in $O(nk + n \log(n/k))$ worst-case time, what is the largest value of k as a function of n for which the modified algorithm has the same running time as standard merge sort, in terms of O notation?