Chapter 2 Getting Started

Harrisburg University of Science and Technology
Mina Gabriel

2 Introduction

In computer Science the sorting algorithm is the process of putting the elements of a given list -*I will refer to this list as A*- in order, mainly a list of positive integer numbers, the main focus is not the implementation itself, the main focus will be the efficiency of the algorithm.

Why sorting? on the application level a telephone directory maintain a sorted dictionary of all it's input, and we also have seen the advantage of a sorted list in Binary search, in statistics a sorted list can help finding the median very easily.

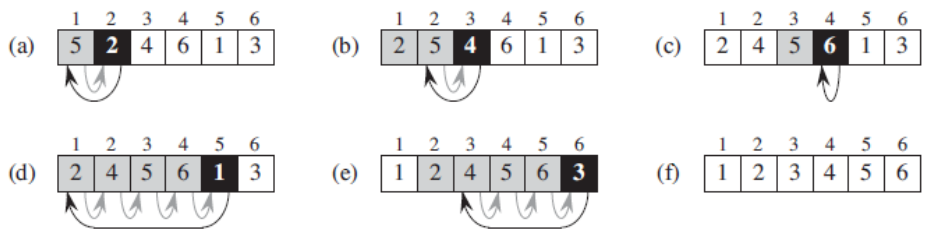
The mechanism of any sorting procedure we will work with, is *Input, Processing and output*, the input is the unsorted list, the processing is the applied or implemented algorithm and finally the output is the sorted list.

Example:

$$\begin{aligned} Input &= A[a_1, a_2, \ldots, a_n] \\ output &= \text{Permutation of A such that } < a_1 \leq a_2 \leq \cdots \leq a_n > \end{aligned}$$

Remember: Trying all the arrangement or permute the list is not a good algorithm,for a smaller number of elements we may think permutation has better performance than quadratic function,for example, a list of 3 elements may seems it takes $\Theta(n!)$ or $3 \times 2 \times 1 = 6$ steps, which is better than many sorting algorithms, like bubble sort and insertion sort with $\Theta(n^2)$, however the price we pay to check if the list is sorted is $\Theta(n \times n!)$ or $3 \times 3! = 18$ steps, as for every step during arranging the list we have to check if it is sorted, and when the number of elements n tends towards infinity permutation will grow much faster than any quadratic run time algorithms.

2.1 Insertion Sort



Insertion sort works by maintain a *sub-sorted* list at the beginning of the given list, mainly, the first step is to assume that the first element in list A is a sorted list and the second element should be inserted in this list such that a_2 will be the first element in the lest, and element a_1 will be the second element in the list, if $a_2 \leq a_1$, by shifting a_1 one position to the right to make space for a_2.

Loop invariants and the correctness of insertion sort

The method of loop invariants is used to prove correctness of a loop with respect to certain pre- and post-conditions.

pre and post-condition example:

Algorithm to compute a product of nonnegative integers

- *Pre-condition: The input variables m and n are nonnegative integers.*
- *Post-condition: The output variable p equals mn.*

Suppose that an algorithm contains a while loop and that entry to this loop is restricted by a condition **G**, called the **guard**. Suppose also that assertions describing the current states of algorithm variables have been placed immediately preceding and immediately following the loop.

```
Pre-condition for the loop
while (G)
Statements in the body of the loop.
None contain branching statements
that lead outside the loop.
end while
Post-condition for the loop
```

2.2 Analyzing algorithms

Analysis of insertion sort

- INSERTIONSORT can take different amounts of time to sort two input sequences of the same size depending on how nearly sorted they already are.
- We describe the running time of an algorithm as a function.
- The running time of an algorithm on a particular input is the number of primitive operations or "steps" executed.
- the following code is to be assumed to take the same amount of time, lets call it c_i, where c_i is a constant time:

We start by presenting the INSERTION-SORT procedure with the time "cost" of each statement and the number of times each statement is executed.

INSERTION–SORT(A)	cost	times
for j = 2 to A.length	c_1	n
key = A[j]	c_2	$n-1$
//Comment	0	$n-1$
i = j − 1	c_4	$n-1$
while i > 0 and A[i] > key	c_5	$\sum_{j=2}^{n} t_j$
A[i+1] = A[i]	c_6	$\sum_{j=2}^{n}(t_j-1)$
i = i − 1	c_7	$\sum_{j=2}^{n}(t_j-1)$
A[i+1] = key	c_8	$n-1$

The outer for loop in the previous code moves the key one slot to the right until it hits the end of the given list, we start with the second element in the list as our key, according to the list given in the previous picture it's element 2, when we ask how many of this for loop iterates during the life time of this program? the answer should be $n-1$ since we start at the second element, but this is not true, as the time is set to be n, the reason for this being that, the extra step is the step the for loop does to check the condition when it hits the end of the list. The key is set to be the value of element with index j, variable i is always 1 unit of index behind j. if we apply this to the list we have in the picture above, key is pointing to element of index 2 and value 2, i is equal 1 -*counting the index from 1 this time*-

The while loop iterates backwards in the sub-sorted list, comparing all the sorted elements with the new coming item from the unsorted list, not just this, if the element $A[i]$ is greater than the key -*the new element to be sorted*- it will shift itself to the right.

The number of times we see the while loop iterates is proportional to the rate of change of the sub-sorted list, the very far top left list has only one element in the sorted sublist, the following list - to the right- has two elements in its sublist then three and so on, if we add all 1these together, the aggregation will be the summation of all integer number between 1 and the size of the list n plus any overload for checking the while loop condition - which mainly an extra step at the end:

$$\sum_{j=2}^{n} j = \frac{n(n+1)}{2} - 1$$

and

$$\sum_{j=2}^{n} (j-1) = \frac{n(n-1)}{2}$$

Now we sum the product of the cost and the times, we obtain:

$$T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5 \sum_{j=2}^{n} t_j + c_6 \sum_{j=2}^{n}(t_j-1) + c_7 \sum_{j=2}^{n}(t_j-1) + c_8(n-1)$$

We find that the **worst-case** running time of **Insertion Sort** is:

$$
\begin{aligned}
T(n) =& c_1 n + c_2(n-1) + c_4(n-1) + c_5\left(\frac{n(n+1)}{2} - 1\right) \\
& + c_6\left(\frac{n(n-1)}{2}\right) + c_7\left(\frac{n(n-1)}{2}\right) + c_8(n-1) \\
=& \left(\frac{c_5}{2} + \frac{c_6}{2} + \frac{c_7}{2}\right) n^2 + \left(c_1 + c_2 + c_4 + \frac{c_5}{2} - \frac{c_6}{2} - \frac{c_7}{2} + c_8\right) n \\
& - (c_2 + c_4 + c_5 + c_8)
\end{aligned}
$$

$T(n) = \text{constant} \times n^2 + \text{constant} \times n - \text{constant}$

Thus insertion sort runs in $O(n^2)$

Worst-case and average-case analysis

The best running time of Insertion Sort is that the list is sorted, the worst case is, the list is reverse sorted, worst case of some data structures is finding an element that doesn't exist.

Order of growth

During the analysis of Insertion Sort we ignored constants c_i or the cost, we can express the running time as $an^2 + bn + c$ for some constant a, b and c, We shall now make one more simplifying abstraction: it is the rate of growth, or order of growth, of the running time that really interests us.

3 Binary Search with Recursive Call

Binary search is an efficient algorithm for finding an element in a sorted array. The recursive implementation of binary search can be described as follows:

```python
def binary_search_recursive(arr, target, low, high):
    if low <= high:
        mid = (low + high) // 2

        if arr[mid] == target:
            return mid
        elif arr[mid] < target:
            return binary_search_recursive(arr, target, mid + 1, high)
        else:
            return binary_search_recursive(arr, target, low, mid - 1)
    else:
        return -1
```

Explanation:

- The function `binary_search_recursive` takes a sorted array (`arr`), a target value (`target`), and the low and high indices of the current subarray (`low` and `high`).

- It calculates the midpoint (`mid`) of the current subarray.
- If the middle element is equal to the target, it returns the index.
- If the middle element is less than the target, it recursively searches in the right half of the array.
- If the middle element is greater than the target, it recursively searches in the left half of the array.
- If the low index exceeds the high index, it means the target is not in the array, and it returns -1.

Master Theorem Analysis

Wouldn't it be great to have a formula to tell us what the solution is instead of using the previous recurrence tree? There is, the **Master Theorem:**

If $T(n) = aT(\lceil \frac{n}{b} \rceil) + O(n^d)$ (for constants $a > 0, b > 1, d \geq 0$), then

$$x = \begin{cases} O(n^d) & if\ d > log_b a \\ O(n^d logn) & if\ d = log_b a \\ O(n^{log_b^a}) & if\ d < log_b a \end{cases}$$

Notice: a is the number of sub-problems crested every recursive call

The Master Theorem is a mathematical tool for analyzing the time complexity of divide-and-conquer algorithms with a specific recurrence relation. In the case of binary search, the recurrence relation is $T(n) = T\left(\frac{n}{2}\right) + 1$, where:

- $a = 1$ (one recursive call),
- $b = 2$ (input size is halved),
- $f(n) = 1$ (constant cost).

According to the Master Theorem, since $f(n) = \Theta(1)$ corresponds to $c = 0$ and $a = b^c = 1$, the time complexity is $\Theta(\log n)$.

Therefore, the time complexity of binary search, when analyzed using the Master Theorem, is $\Theta(\log n)$.

4 MERGE SORT

Merge sort uses a design paradigm called *Divide and Conquer*, where we have to brake things down to it's simplest form of n sub-problems, and recursively solving those sub-problems. Each sub-problem has to be smaller than the original problem.

Three steps of *Divide and Conquer* paradigm:

- Divide: Into number of sub-problems.
- Conquer: sub-problems by solving the recursively.
- Combine: The solution of Sub-problem.

We described the worst-case running time $T(n)$of the MERGE-SORT procedure by the recurrence:

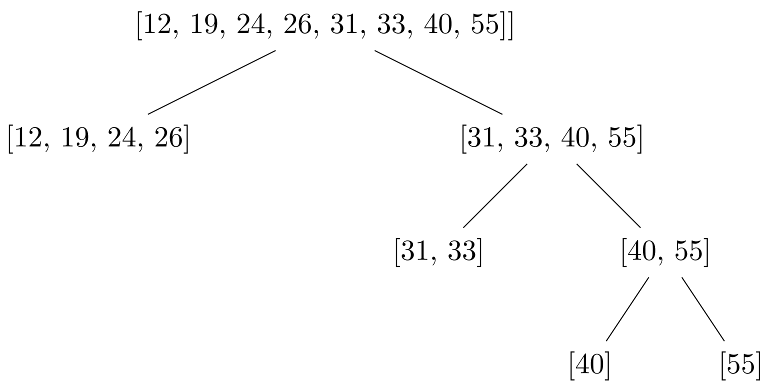
$$T(n) = \begin{cases} O(1) & if n = 1 \\ 2T(\frac{n}{2}) + O(n) & if n > 1 \end{cases}$$

Where $2T(\frac{n}{2})$ is the recursive call and is equal to $T(\lceil \frac{n}{2} \rceil) + T(\lfloor \frac{n}{2} \rfloor)$, neither of these sub-problem is $\frac{n}{2}$ when n is odd, $\Theta(n)$ is the amount of time we have to pay for every recursive call.

Recursion is not new, we have seen it before with **Binary Search** when we had to break the list down in two halves and compare the item we are trying to find.

$$T(n) = T(\frac{n}{2}) + O(1)$$

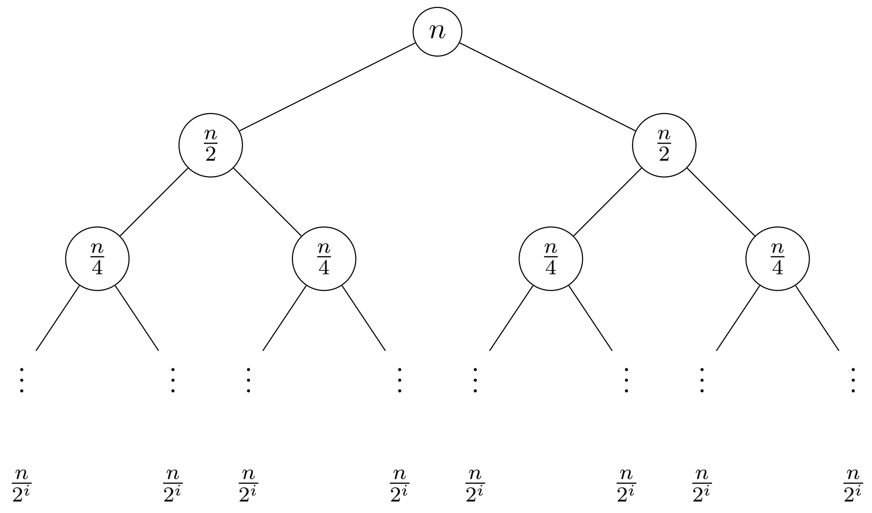
The following tree show a binary search of an item ≥ 55, the total number of steps to find/not find this item is equal to the number of times we had to break the list for, that is $T(n) = O(log_2 n)$ times.



Previously, we described the running time of the Merge Sort as $2T(\frac{n}{2}) + O(n)$ lets apply the master theorem which will help us figuring the complexity of such algorithm:

$$\begin{aligned} T(n) &= 2T(\frac{n}{2}) + O(n) \\ a &= 2 \\ b &= 2 \\ d &= 1 \\ Since\ d &= log_b a \\ Then\ T(n) &= O(n^d log\, n) = O(n\, log\, n) \end{aligned}$$

The recursion-tree method for Merge Sort



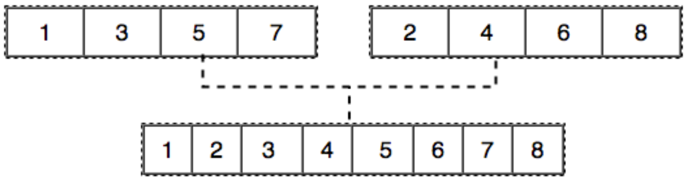
The number of elements at the root is $\frac{n}{2^0}$, the number of elements in level 1 in each sub-problem is $\frac{n}{2^1}$, the next level is $\frac{n}{2^2}$ and so on until the leaves which are $\frac{n}{2^i}$, and since:

$$\begin{aligned} \frac{n}{2^i} &= \frac{n}{n} \\ 2^i &= n \\ log_2\, n &= i \end{aligned}$$

Where i is the number of levels, and since every level need O(n) times to solve its sub-problem, then the total amount of work needed is O(n log n)

The Two Finger Algorithm

The total amount of work needed to merge the following two sorted list into a new sorted list is linear O(n), and this is the key idea of the **Merge Sort**



Listing 1: TWO FINGER ALGORITHM

```
C=[]
def TwoFingerAlgorithm(A, B):
   print(A,B)
   if len(B) == 0 and len(A) >= 1:
     C.extend(A)
     return
   if len(A) == 0 and len(B) >= 1:
     C.extend(B)
     return
   if A[0] < B[0]:
     C.append(A[0])
     TwoFingerAlgorithm(A[1:], B)
   if A[0] > B[0]:
     C.append(B[0])
     TwoFingerAlgorithm(A, B[1:])

TwoFingerAlgorithm([1,3,5,7],[2,4,6,8])
print(C)
```

Listing 2: MERGE SORT

```
def mergeSort(alist):
    print("Splitting ",alist)
    if len(alist)>1:
        mid = len(alist)//2
        lefthalf = alist[:mid]
        righthalf = alist[mid:]

        mergeSort(lefthalf)
        mergeSort(righthalf)

        i=0
        j=0
        k=0
        while i < len(lefthalf) and j < len(righthalf):
            if lefthalf[i] < righthalf[j]:
                alist[k]=lefthalf[i]
                i=i+1
            else:
                alist[k]=righthalf[j]
                j=j+1
            k=k+1

        while i < len(lefthalf):
            alist[k]=lefthalf[i]
            i=i+1
            k=k+1

        while j < len(righthalf):
            alist[k]=righthalf[j]
            j=j+1
            k=k+1
    print("Merging ", alist)

alist = [54,26,93,17,77,31,44,55,20]
mergeSort(alist)
print(alist)
```

Loop Invariant Examples

A loop invariant is a property or condition that is true before and after each iteration of a loop. Loop invariants are used to reason about the correctness of loops, ensuring that certain properties hold at the beginning and end of each iteration. They are essential in algorithm design and analysis.

Example 1: Insertion Sort

From CLRS for the Insertion Sort algorithm:

```
for j = 2 to A.length
    key = A[j]
    // Insert A[j] into the sorted sequence A[1..j-1]
    i = j - 1
    while i > 0 and A[i] > key
        A[i + 1] = A[i]
        i = i - 1
    A[i + 1] = key
```

Loop Invariant: At the start of each iteration of the for loop, the subarray $A[1..j-1]$ consists of the elements originally in $A[1..j-1]$, but in sorted order.

Initialization: Before the first iteration ($j = 2$), the subarray $A[1..j-1]$ is just $A[1]$, which is trivially sorted.

Maintenance: Assuming the invariant holds at the start of the iteration ($A[1..j-1]$ is sorted), the while loop shifts elements to the right until the correct position for $A[j]$ is found. After the while loop, $A[1..j]$ is sorted.

Termination: When the for loop exits ($j = A.length + 1$), the entire array A is sorted.

Example 2: Binary Search

Binary Search is another example where a loop invariant is crucial. Here's the iterative version:

```
BinarySearch(A, v)
    low = 1, high = A.length
    while low <= high
        mid =     (low + high)/2
        if A[mid] == v
            return mid
        else if A[mid] < v
            low = mid + 1
        else
            high = mid - 1
    return -1
```

Loop Invariant: At the start of each iteration of the while loop, if v is in A, it must be in $A[low..high]$.

Initialization: Before the first iteration, the entire array A is considered ($low = 1$, $high = A.length$), and the invariant holds.

Maintenance: The while loop halves the search interval based on comparisons with $A[mid]$, ensuring that the invariant holds for the new subarray.

Termination: When the while loop exits, $low > high$, and v is not in $A[low..high]$, indicating that v is not in A.