NOTES (UNIT-I) Introduction

1.1 Algorithm

An algorithm is any well-defined computational procedure that takes some value, or set of values, as input and produces some value, or set of values, as output. An algorithm is thus a sequence of computational steps that transform the input into the output.

For example, given the input sequence {31, 41, 59, 26, 41, 58), a sorting algorithm returns as output the sequence {26, 31, 41, 41, 58, 59}. Such an input sequence is called an instance of the sorting problem. ,

Instance : An instance of a problem consists of the input needed to compute a solution to the problem.

An algorithm is said to be correct if, for every input instance, it halts with the correct output.

There are *two aspects* of algorithmic performance:

• Time

- What kind of data structures can be used?
- How does choice of data structure affect the runtime?

1.1.1 Analysis of Algorithms

Analysis is performed with respect to a computational model

- We will usually use a generic uniprocessor random-access machine (RAM)
  - All memory equally expensive to access
  - No concurrent operations
  - All reasonable instructions take unit time
    - Except, of course, function calls
  - Constant word size

Unless we are explicitly manipulating bits

Input Size:

- Time and space complexity
  - This is generally a function of the input size o E.g., sorting, multiplication
  - How we characterize input size depends:
    - Sorting: number of input items
    - Multiplication: total number of bits
    - Graph algorithms: number of nodes & edges
    - Etc

Running Time:

- Number of primitive steps that are executed
  - Except for time of executing a function call most statements roughly require the same amount of time
    - y = m * x + b
    - c = 5 / 9 * (t - 32) o z = f(x) + g(y)
- We can be more exact if need be

Analysis:

- Worst case
  - Provides an upper bound on running time
  - An absolute guarantee
- Average case
  - Provides the expected running time
  - Very useful, but treat with care: what is "average"? o Random (equally likely) inputs
    - Real-life inputs

1.1.2 Analyzing algorithm

Example: Insertion Sort:

```
InsertionSort(A, n) {
for i = 2 to n {
        key = A[i]
        j = i - 1;
        while (j > 0) and (A[j] > key) {
                A[j+1] = A[j]
                j = j - 1
                }
        A[j+1] = key
        }

}
```

Analysis

Statement	Effort
InsertionSort(A, n) {	
for i = 2 to n {	c_1n
key = A[i]	$c_2(n-1)$
j = i - 1;	$c_3(n-1)$
while (j > 0) and (A[j] > key) {	c_4T
A[j+1] = A[j]	$c_5(T-(n-1))$
j = j - 1	$c_6(T-(n-1))$
}	0
A[j+1] = key	$c_7(n-1)$
}	0
}	

$T = t_2 + t_3 + \ldots + t_n$ where t_i is number of while expression evaluations for the i^{th} for loop iteration

$T(n) = c_1 n + c_2(n-1) + c_3(n-1) + c_4T + c_5(T - (n-1)) + c_6(T - (n-1)) + c_7(n-1) = c_8T + c_9n + c_{10}$

Best case -- inner loop body never executed

$t_i = 1$ ➔

$$\begin{aligned} T(n) &= c_1n + c_2(n-1) + c_4(n-1) + c_5(n-1) + c_8(n-1) \\ &= (c_1 + c_2 + c_4 + c_5 + c_8)n - (c_2 + c_4 + c_5 + c_8) . \end{aligned}$$

$= an-b = \Theta(n)$

T(n) is a linear function

Worst case -- inner loop body executed for all previous elements

$t_i = i$ ➔

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

$= an^2 + bn - c = O(n^2)$

T(n) is a quadratic function

Average case: The “average case” is often roughly as bad as the worst case.

1.1.3 Designing algorithms

The divide and conquer approach

The divide-and-conquer paradigm involves three steps at each level of the recursion:

Divide the problem into a number of subproblems that are smaller instances of the same problem.

Conquer the subproblems by solving them recursively. If the subproblem sizes are small enough, however, just solve the subproblems in a straightforward manner.

Combine the solutions to the subproblems into the solution for the original problem.

1.2 Merge sort

The ***merge sort*** algorithm closely follows the divide-and-conquer paradigm. Intuitively, it operates as follows.

Divide: Divide the n-element sequence to be sorted into two subsequences of n=2 elements each.

Conquer: Sort the two subsequences recursively using merge sort.

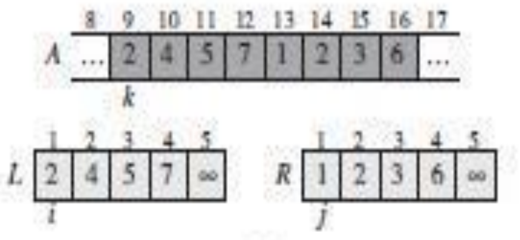
Combine: Merge the two sorted subsequences to produce the sorted answer.

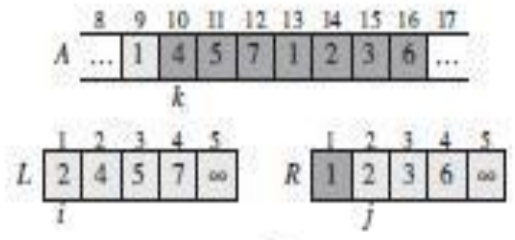
The key operation of the merge sort algorithm is the merging of two sorted sequences in the “combine” step. We merge by calling an auxiliary procedure MERGE (A, p, q, r) where A is an array and p, q, and r are indices into the array such that $p \le q < r$. The procedure assumes that the subarrays A (p…. q) and A (q+1……… r) are in sorted order. It ***merges*** them to form a single sorted subarray that replaces the current subarray A (p …….r). Our MERGE procedure takes time Θ(n), where n = r- p+1 is the total number of elements being merged.

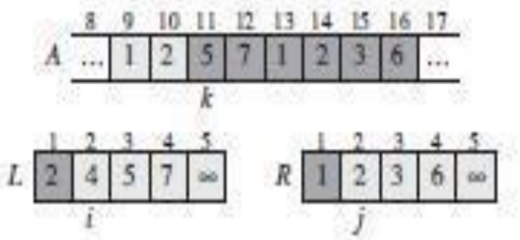
```
MERGE(A, p, q, r)
1 $n1 \leftarrow q - p + 1$
2 $n2 \leftarrow r - q$
```

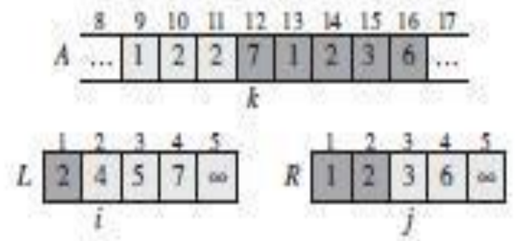
```
3 create arrays $L[1 \ldots n1 + 1]$ and $R[1 \ldots n2 + 1]$
4 for $i \leftarrow 1$ to $n1$
5     do $L[i] \leftarrow A[p + i - 1]$
6 for $j \leftarrow 1$ to $n2$
7     do $R[j] \leftarrow A[q + j]$
8 $L[n1 + 1] \leftarrow \infty$
9 $R[n2 + 1] \leftarrow \infty$
10 $i \leftarrow 1$
11 $j \leftarrow 1$
12 for $k \leftarrow p$ to $r$
13     do if $L[i] \leq R[j]$
14         then $A[k] \leftarrow L[i]$
15             $i \leftarrow i + 1$
16         else $A[k] \leftarrow R[j]$
17             $j \leftarrow j + 1$
```

Example:


(a)


(b)


(c)


(d)

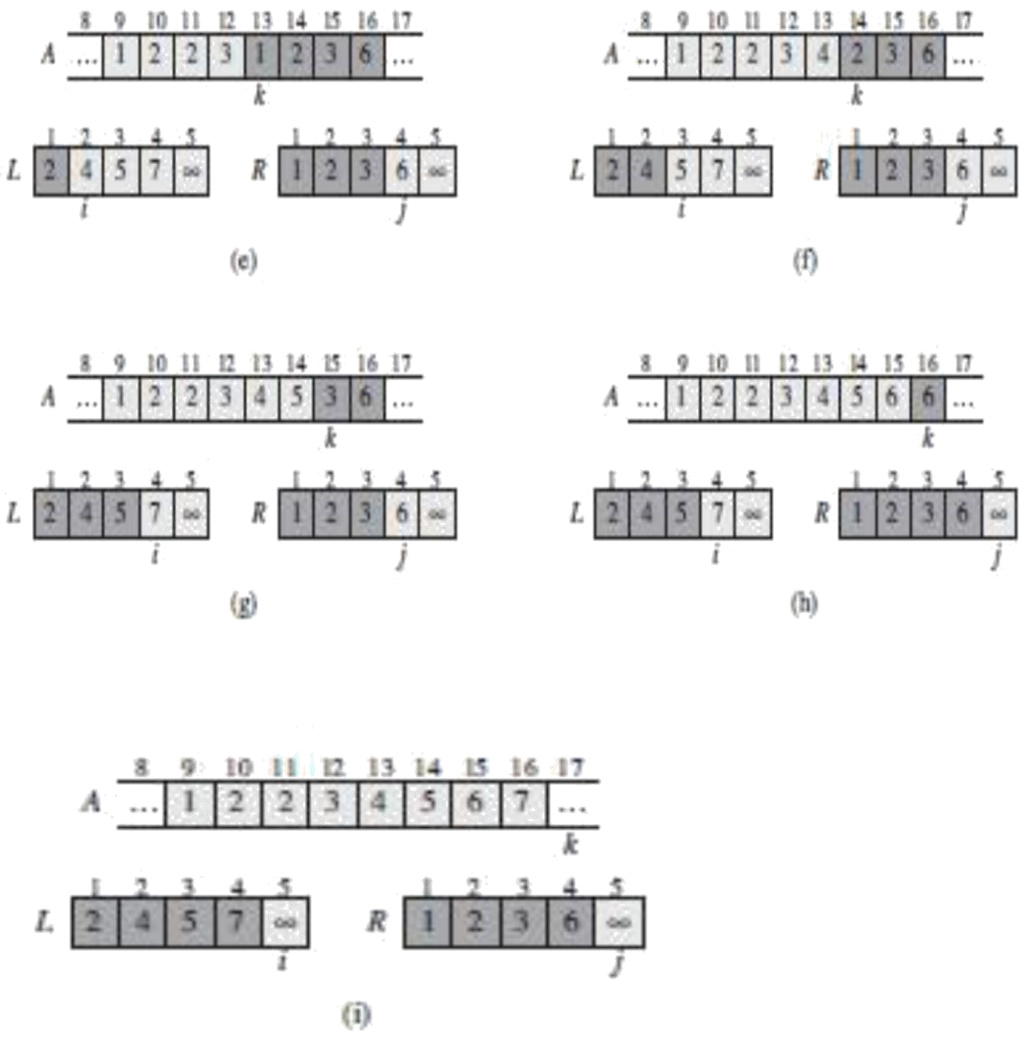


Fig: The Merge procedure applies on given array and sort and combines the solution in recursive iteration.

```
MERGE-SORT($A, p, r$)
1 if $p < r$
2 then $q \leftarrow (p + r)/2$
3 MERGE-SORT($A, p, q$)
4 MERGE-SORT($A, q + 1, r$)
5 MERGE($A, p, q, r$)
```

1.2.1 Analysis of Merge-sort

When we have n > 1 elements, we break down the running time as follows.

Divide: The divide step just computes the middle of the subarray, which takes constant time. Thus, D(n)=Ө(1).

Conquer: We recursively solve two subproblems, each of size n/2, which contributes 2T(n/2) to the running time.

Combine: We have already noted that the MERGE procedure on an n-element subarray takes time Ө(n) and so C(n)=Ө(n).

The recurrence for the worst-case running time T(n) of merge sort:

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

The solution for above recurrence is Θ (n log n).

1.3 Growth of functions

The notations we use to describe the asymptotic running time of an algorithm are defined in terms of functions whose domains are the set of natural numbers N=(0, 1, 2……). Such notations are convenient for describing the worst-case running-time function T(n), which usually is defined only on integer input sizes.

1.3.1 Asymptotic notations

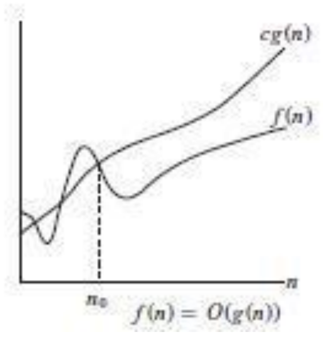
1.3.1.1 Upper Bound Notation or O-notation

We say InsertionSort's run time is *O(n^2).* Properly we should say run time is *in* O(n^2). Read O as "Big-O" (you'll also hear it as "order")

In general a function f(n) is O(g(n)) if there exist positive constants *c* and *n0* such that f(n) £ *c* × g(n) for all n ³ *n0*

Formally

O(g(n)) = { f(n): $ positive constants *c* and *n0* such that f(n) £ *c* × g(n) " n ³ *n0*



Example. 1. functions in **O(n^2)**

n^2/ 1000, $n^{1.9}$, n^2, n^2+n, 1000 n^2+50n

2. Show 2n^2 = O (n^3)

0 ≤ h(n) ≤ *c*g(n) Definition of O(g(n))

$0 \le 2n^2 \le cn^3$ Substitute

$0/n^3 \le 2n^2/n^3 \le cn^3/n^3$ Divide by n^3

Determine c

$0 \le 2/n \le c$ $2/n = 0$

$2/n$ maximum when $n=1$

$0 \le 2/1 \le c = 2$ Satisfied by c=2

Determine n_0

$0 \le 2/ n_0 \le 2$

$0 \le 2/2 \le n_0$

$0 \le 1 \le n_0 = 1$ Satisfied by $n_0=1$

$0 \le 2n^2 \le 2n^3 \; \forall \; n \ge n_0=1$

If $f(n) \le cg(n)$, $c > 0$, $\forall\; n \ge n_0$ then $f(n) \in O(g(n))$

3. 1000n2 + 50n = O(n2)

$0 \le h(n) \le cg(n)$

$0 \le 1000n2 + 50n \le cn2$ Substitute

$0/n2 \le 1000\, n2/\, n2 + 50n/\, n2 \le c\, n2/\, n2$ Divide by n2

$0 \le 1000 + 50/n \le c$ Note that $50/n \to 0$ as $n \to \infty$

Greatest when n = 1

$0 \le 1000 + 50/1 = 1050 \le c = 1050$ Satisfied by c=1050

$0 \le 1000 + 50/n0 \le 1050$ Find n0 $\forall\; n \ge n0$

$-1000 \le 50/\, n0 \le 50$

$-20 \le 1/\, n0 \le 1$

$-20\, n0 \le 1 \le n0 = 1$ Satisfied by n0=1

$0 \le 1000\, n2 + 50n \le 1050\, n2 \;\forall\; n \ge n0=1, c=1050$

If $f(n) \le cg(n)$, $c > 0$, $\forall\; n \ge n0$ then $f(n) \in O(g(n))$

Big O Fact

A polynomial of degree k is $O(n^k)$

Proof:

Suppose $f(n) = b_k n^k + b_{k-1} n^{k-1} + \ldots + b_1 n + b_0$

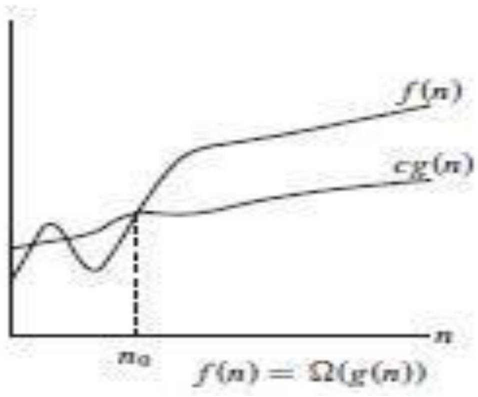
Let $a_i = | b_i |$

$f(n) \le a_k n^k + a_{k-1} n^{k-1} + \ldots + a_1 n + a_0$

$$\le n^k \sum a_i \frac{n^i}{n^k} \le n^k \sum a_i \le cn^k$$

1.3.1.2 Lower Bound Notation or Ω-notation

We say Insertion Sort's run time is W(n). In general a function f(n) is W(g(n)) if \$ positive constants *c* and n_0 such that 0 £ c×g(n) £ f(n) " n ³ n_0



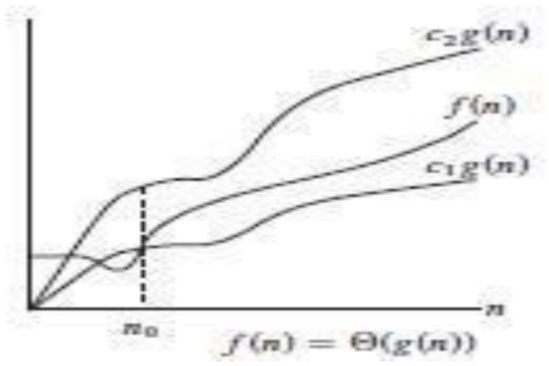
Proof: Suppose run time is an + b. Assume a and b are positive (what if b is negative?).
an £ an + b

Example. 1. Functions in Ω(n^2)
n^2/ 1000, $n^{1.999}$, n^2+n, 1000 n^2+50n

2. n3 = Ω(n^2) with c=1 and n_0=1
0 ≤ *c*g(n) ≤ h(n)
0 ≤ 1*1^2 = 1 ≤ 1 = 13
0 ≤ *c*g(n) ≤ h(n)
0 ≤ c n^2 ≤ n^3
0/ n^2 ≤ c n^2/ n^2 ≤ n^3/ n^2
0 ≤ c ≤ n
0 ≤ 1 ≤ 1 with c=1 and n_0=1 since n increases.

1.3.1.3 Asymptotic Tight Bound or -notation

A function f(n) is Q(g(n)) if \$ positive constants c_1, c_2, and n_0 such that c_1 g(n) £ f(n) £ c_2 g(n) " n ³ n_0



Theorem: f(n) is Q(g(n)) iff f(n) is both O(g(n)) and W(g(n))

Example . 1. Show that ½n^2 - 3n ∈ Θ(n^2)

Determine ∃ c1, c2, n0 *positive constants* such that:
c1 n2 ≤ ½ n2 - 3n ≤ c2 n2
c1 ≤ ½ - 3/n ≤ c2 Divide by n2
O: Determine c2 = ½
½ - 3/n ≤ c2
as n → ∞, 3/n → 0
½ - 3/n = ½
therefore ½ ≤ c2 or c2 = ½ ½ - 3/n maximum for as n → ∞
Ω: Determine c1 = 1/14
0 < c1 ≤ ½ - 3/n ½ - 3/n > 0 minimum for n0 = 7
0 < c1 = ½ - 3/7 = 7/14 - 6/14 = 1/14
n0: Determine n0 = 7
c1 ≤ ½ - 3/ n0 ≤ c2
1/14 ≤ ½ - 3/ n0 ≤ ½
½n2-3n = Θ(n2)
c1=1/14, c2=1/2 and n
Θ: ½n2 - 3n ∈ Θ(n2) when c1 = 1/14, c2 = ½ and n0 = 7

1.3.1.4 Other Asymptotic Notations

o-notation: A function f(n) is o(g(n)) if $ positive constants *c* and *n0* such that f(n) < *c* g(n) " n ³ *n0*

ω-notation: This notation is A function f(n) is w(g(n)) if $ positive constants *c* and *n0* such that *c* g(n) < f(n) " n ³ *n0*

Intuitively,

- o() is like <
- w() is like >
- Q() is like =
- W() is like ³
- O() is like £

•

1.4 Recurrences

Recurrences

A ***recurrence*** is an equation or inequality that describes a function in terms of its value on smaller inputs. For example, the worst-case running time T (n) of the MERGE-SORT procedure by the recurrence

$$T(n) = \begin{cases} c & n = 1 \\ 2T\left(\frac{n}{2}\right) + cn & n > 1 \end{cases}$$

Whose solution is to be T (n)=O(n lg n).

1.4.1 Solving Recurrences: following methods are used to solve recurrences and find its asymptotic performance

1. Substitution method
2. Recursion Tree
3. Master method
4. Iteration Method

1.4.1.1 The substitution method

This method is called “making a good guess method” and 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

We can use the substitution method to establish either upper or lower bounds on a recurrence. As an example, let us determine an upper bound on the recurrence

T(n) = 2T(ën/2û) + n

We guess that the solution is T(n)=O(n lg n). The substitution method requires us to prove T(n) ≤ c n lg n for an appropriate choice of the constant c > 0. We start by assuming that this bound holds for all positive m < n, in particular for m = ën/2û, yielding T(ën/2û) =c(ën/2û) lg (ën/2û). Substituting into the recurrence yields

T(n) ≤ 2 c(ën/2û) lg (ën/2û)+ n
≤ cn lg (n/2) +n
= cn lg n _ cn lg 2 + n
≤ cn lg n _ cn + n
≤ cn lgn

where the last step holds as long as c ≥ 1.

1.4.1.2 Changing variables:

Sometimes, a little algebraic manipulation can make an unknown recurrence similar to one you have seen before. As an example, consider the recurrence

$$T(n) = 2T\left(\lfloor \sqrt{n} \rfloor\right) + \lg n$$

which looks difficult. We can simplify this recurrence, though, with a change of variables. For convenience Renaming m= lg n yields

$T(2^m) = T(2^{m/2})+m$

We can now rename $S(m) = T(2^m)$ to produce the new recurrence S(m)= S (m/2) +m

The solution is to be S (m)= O (m lg m)

After replacing m= log n the solution becomes

$T(n) = O(\lg n \lg \lg n)$

1.4.1.3 The recursion-tree method: Substitution method is not coming up with good guess. Drawing out a recursion tree provides a method to devise a good guess. In a ***recursion tree***, each node represents the cost of a single subproblem somewhere in the set of recursive function invocations. We sum the costs within each level of the tree to obtain a set of per-level costs, and then we sum all the per-level costs to determine the total cost of all levels of the recursion.
For example, let us see how a recursion tree would provide a good guess for the recurrence

$$T(n)= 3T(\lfloor n/4 \rfloor) + \Theta(n^2)$$

we create a recursion tree for the recurrence $T(n)= 3T(n/4) + cn^2$, having written out the implied constant coefficient $c > 0$.

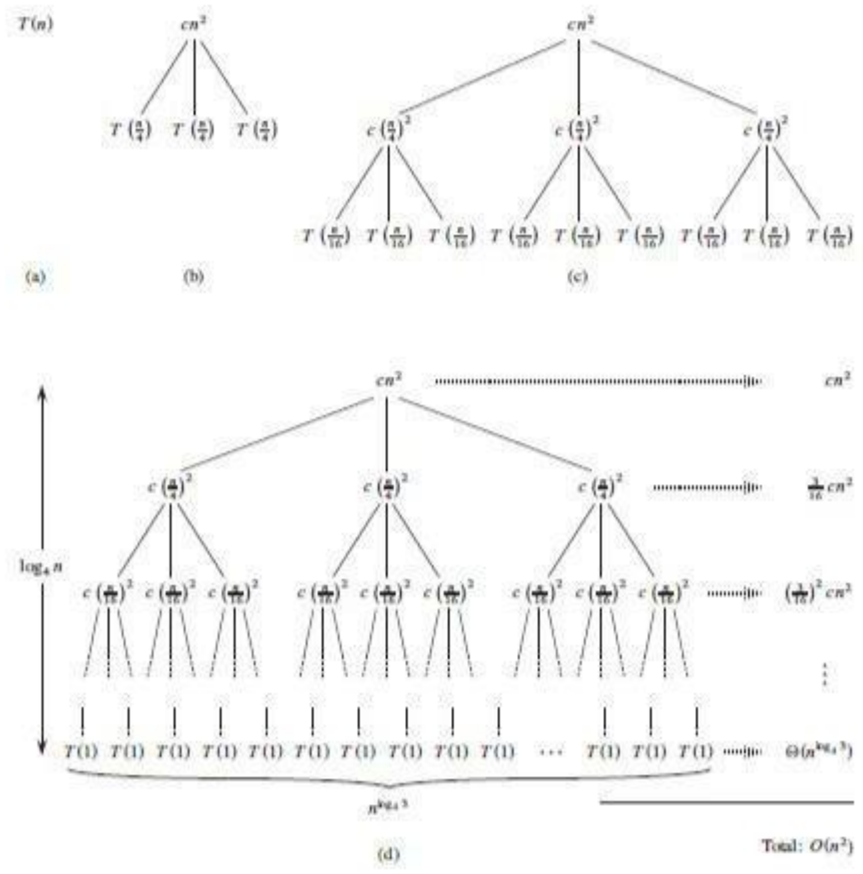


Fig: Constructing a recursion tree for the recurrence $T(n)= 3T(n/4) + cn^2$. Part **(a)** shows T (n), which progressively expands in **(b)–(d)** to form the recursion tree. The fully expanded tree in part (d) has height $\log_4 n$ (it has $\log_4 n +1$ levels).

The above fig shows how we derive the recursion tree for $T(n)= 3T(n/4) + cn^2$ .For convenience, we assume that n is an exact power of 4 so that all subproblem sizes are integers. Part (a) of the figure shows T(n), which we expand in part (b) into an equivalent tree representing the recurrence. The cn^2 term at the root represents the cost at the top level of recursion, and the three subtrees of the root represent the costs incurred by the subproblems of size n=4. Part (c) shows

this process carried one step further by expanding each node with cost T(n/4) from part (b). The cost for each of the three children of the root is $c(n/4^2)$. We continue expanding each node in the tree by breaking it into its constituent parts as determined by the recurrence.
Because subproblem sizes decrease by a factor of 4 each time the subproblem size for a node at depth i is $(n/4^i)$. Thus for subproblem size at last level n=1 $n/4^i = 1$ then $i = \log_4 n$ and the tree has $\log_4 n+1$ levels. The cost of each node is derived by generalized term at depth i where i= 0, 1, 2…….$\log_4 n-1$ by $c\,(n/4^i)^2$. the total cost over all nodes at depth i, for i= 0, 1, 2…….$\log_4 n-1$ is $3^i (n/4^i)^2 = (3/16)^i cn^2$. The last level i.e. at depth $\log_4 n$ cost is $3^{\log_4 n} = n^{\log_4 3}$ nodes each contributing cost T(1), for a total cost $n^{\log_4 3}\,T(1)$ which is $\Theta(n^{\log_4 3})$ since we assume T(1) is constant.
Now we add up the costs over all levels to determine the cost for the entire tree:

$$\begin{aligned}
T(n) &= cn^2 + \frac{3}{16}cn^2 + \left(\frac{3}{16}\right)^2 cn^2 + \cdots + \left(\frac{3}{16}\right)^{\log_4 n-1} cn^2 + \Theta(n^{\log_4 3}) \\
&= \sum_{i=0}^{\log_4 n-1} \left(\frac{3}{16}\right)^i cn^2 + \Theta(n^{\log_4 3}) \\
&< \sum_{i=0}^{\infty} \left(\frac{3}{16}\right)^i cn^2 + \Theta(n^{\log_4 3}) \\
&= \frac{1}{1-(3/16)} cn^2 + \Theta(n^{\log_4 3}) \\
&= \frac{16}{13} cn^2 + \Theta(n^{\log_4 3}) \\
&= O(n^2)\,.
\end{aligned}$$

Thus, we have derived a guess of $T(n)= O(n^2)$. Now we can use the substitution method to verify that our guess was correct, that is, $T(n)= O(n^2)$ is an upper bound for the recurrence $T(n)= 3T(n/4) + \Theta(n^2)$. We want to show that $T(n) \le dn^2$ for some constant $d > 0$. Using th~~e sa~~me constant $c > 0$ as before, we have

$$\begin{aligned}
T(n) &\le 3T(\lfloor n/4 \rfloor) + cn^2 \\
&\le 3d\lfloor n/4 \rfloor^2 + cn^2 \\
&\le 3d(n/4)^2 + cn^2 \\
&= \frac{3}{16} dn^2 + cn^2 \\
&\le dn^2\,,
\end{aligned}$$

Where the last step holds as long as d = (16/13)c.

1.4.1.4 The master method

For divide and conquer algorithm**,** An algorithm that divides the problem of size n into a subproblems, each of size n/b Let the cost of each stage (i.e., the work to divide the problem + combine solved subproblems) be described by the function f(n), then, the Master Theorem gives us a cookbook for the algorithm’s running time in the form of recurrence given below

$$T(n) = a\,T(n/b) + f(n)$$

Where $a \ge 1$and $b > 1$ are constants and f (n) is an asymptotically positive function.

1.4.1.4.1 The master theorem:

The master method depends on the following theorem.

Let a≥1and b>1 be constants, let f (n) be a function, and let T(n) be defined on the nonnegative integers by the recurrence

$$T(n) = a\,T(n/b) + f(n)$$

where we interpret n/b to mean either floor (n/b) or ceil (n/b). Then T(n) has the following asymptotic bounds:

1. If $f(n)=O(n^{\log_b a-\varepsilon})$ for some constant $\epsilon>0$ then T(n)= $\Theta(n^{\log_b a}$
2. If $f(n) = \Theta(n^{\log_b a}))$. then T(n)= $\Theta(n^{\log_b a}\log n)$.
3. If $f(n)=\Omega(n^{\log_b a+\varepsilon})$ for some constant $\epsilon>0$ and if a T (n/b) ≤c f(n)(regularity function) for some constant c<1 and all sufficiently large n then T(n)= Θ(f (n)).

In each of the three cases, we compare the function f (n) with the function $n^{\log_b a}$. Intuitively, the larger of the two functions determines the solution to the recurrence. In case 1the function $n^{\log_b a}$ is larger, solution is T(n)= $\Theta(n^{\log_b a})$. In case 3 function f(n) is larger, solution is T(n)= Θ(f (n)). In case 2 both functions has same value, solution is T(n)= $\Theta(n^{\log_b a}\log n)$.

In the first case, not only must f(n) be smaller than $n^{\log_b a}$, it must be *polynomially* smaller. In the third case, not only must f (n) be larger than $n^{\log_b a}$, it also must be polynomially larger and in addition satisfy the "regularity" condition that a T (n/b) ≤c f(n). An addition to that all three cases do not cover all possibilities. Some function might be lies in between case 1 and 2 and some other lies in between case 2 and 3 because the comparison is not polynomial larger or smaller and in case 3 the regularity condition fails.
Example. 1. . The given recurrence is

T(n) = 9T(n/3) + n

Sol: a=9, b=3, f(n) = n

$n^{\log b\ a} = n^{\log 3\ 9} = \Theta(n^2)$

Since $f(n) = O(n^{\log 3\ 9 - \varepsilon})$, where ε=1, case 1 applies:

$$T(n) = \Theta(n^{\log_b a}) \text{ when } f(n) = O(n^{\log_b a-\varepsilon})$$

Thus the solution is $T(n) = \Theta(n^2)$

2. T(n)= T(2n/3)+1
in which a = 1, b= 3/2, f(n)= 1, and $n^{\log_b a} = n^{\log 3/2} = n^0 = 1$. Case 2
applies, since f (n)= $\Theta(n^{\log_b a}) = \Theta(1)$ and thus the solution to the recurrence is T(n) =Θ(lg n)
3. The master method does not apply to the recurrence:
T(n)= 2T(n/2) +n log n
Sol: a=2, b=2, f(n)= n log n , $n^{\log_b a}=n^{\log 2}=n$
Since f(n) is larger than $n^{\log_b a}$ you mistakenly apply case 3 but the f(n) is larger but not polynomialy larger. The ratio f(n)/ $n^{\log_b a}$= log n is asymptotically less than n^{ϵ} for any positive constant ϵ. Consequently, the recurrence falls into the gap between case 2 and case 3.

1.4.1.5 Iteration method

Another option is the "iteration method" which expand the recurrence by using iterative equations, work some algebra to express as a summation and finally evaluate the summation. It's a iterative procedure which recursively used and prorogate to a final single value.

For example

Floor and ceilings are a pain to deal with. If n is assumed to be a power of 2 ($2_k = n$), this will simplify the recurrence to

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

The iteration method turns the recurrence into a summation. Let's see how it works. Let's expand the recurrence:

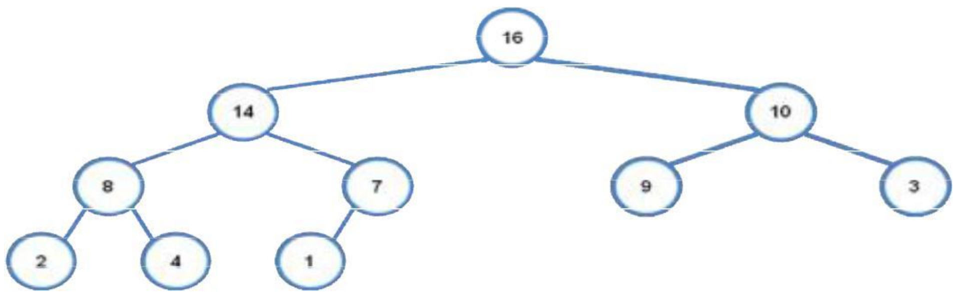
$$\begin{aligned} T(n) &= 2T(n/2) + n \\ &= 2(2T(n/4) + n/2) + n \\ &= 4T(n/4) + n + n \\ &= 4(2T(n/8) + n/4) + n + n \\ &= 8T(n/8) + n + n + n \\ &= 8(2T(n/16) + n/8) + n + n + n \\ &= 16T(n/16) + n + n + n + n \end{aligned}$$

If n is a power of 2 then let $n = 2^k$ or $k = \log n$.

$$\begin{aligned} T(n) &= 2^k T(n/(2^k)) + \underbrace{(n + n + n + \cdots + n)}_{k \text{ times}} \\ &= 2^k T(n/(2^k)) + kn \\ &= 2^{(\log n)} T(n/(2^{(\log n)})) + (\log n)n \\ &= 2^{(\log n)} T(n/n) + (\log n)n \\ &= nT(1) + n \log n = n + n \log n \end{aligned}$$

1.5 Heap sort

A heap can be seen as a nearly binary tree:



An array A that represents a heap is an object with two attributes: A.length, which (as usual) gives the number of elements in the array, and A.heap-size, which represents how many elements in the heap are stored within array A. That is, although A (1……..A.length) may contain numbers, only the elements in A(1……….A.heap-size), where 0≤ A.heap-size≤ A.length, are valid elements of the heap. The root of the tree is A(1), and given the index i of a node, we can easily compute the indices of its parent, left child, and right child:

```
Parent(i)
return floor (i/2)

Left(i)
return 2i

Right(i)
return 2i+1
```

There are two kinds of binary heaps: max-heaps and min-heaps. In both kinds, the values in the nodes satisfy a heap property, the specifics of which depend on the kind of heap. In a max-heap, the max-heap property is that for every node i other than the root,

$$A[parent(i)] \geq A[i]$$

That is the value of a node is lesser or equal to parent.

A min-heap is organized in the opposite way; the min-heap property is that for every node i other than the root,

$$A[parent(i)] \leq A[i]$$

For the heapsort algorithm, we use max-heaps.

1.5.1 Maintaining the heap property:

1.5.1.1 MAX-HEAPIFY(A, i)

```
{
        l = Left(i); r = Right(i);
        if (l <= A.heap-size && A[l] > A[i])
            largest = l;
        else
            largest = i;
        if (r <= A.heap-size && A[r] > A[largest])
            largest = r;
        if (largest != i)
            exchange A[i] with A[largest];
            MAX-HEAPIFY(A, largest);
}
```

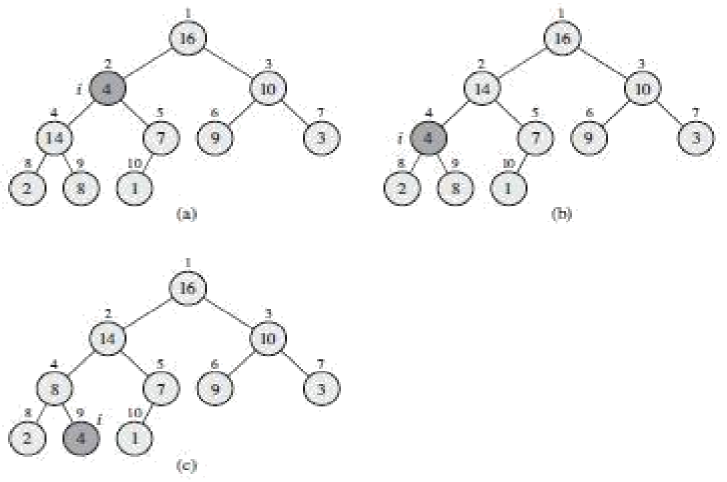


Figure: The action of MAX-HEAPIFY(*A*, 2), where *heap-size*[*A*] = 10. *(a)* The initial configuration, with *A*[2] at node *i* = 2 violating the max-heap property since it is not larger than both children. The max-heap property is restored for node 2 in *(b)* by exchanging *A*[2] with *A*[4], which destroys the max-heap property for node 4. The recursive call MAXHEAPIFY(*A*, 4) now has *i* = 4. After swapping *A*[4] with *A*[9], as shown in *(c)*, node 4 is fixed up, and the recursive call MAX-HEAPIFY(*A*, 9) yields no further change to the data structure.

1.5.1.2 Complexity

We can describe the running time of MAX-HEAPIFY by the

recurrence $T(n) \le 2T(n/3)+\Theta(1)$

The solution to this recurrence, by case 2 of the master theorem is $T(n)= O(\lg n)$

1.5.2 Building a heap:

We can build a heap in a bottom-up manner by running **Heapify()** on successive subarrays. For array of length *n*, all elements in range A[ën/2û + 1 .. n] are heaps.

1.5.2.1 BUILD-MAX-HEAP(A)

```
1. A.heap-size = A.length
2. for i = ëlength[A]/2û downto 1
3. MAX-HEAPIFY(A, i)
```

The time required by MAX-HEAPIFY when called on a node of height h is O(h), the running time of above algorithm is O(n).

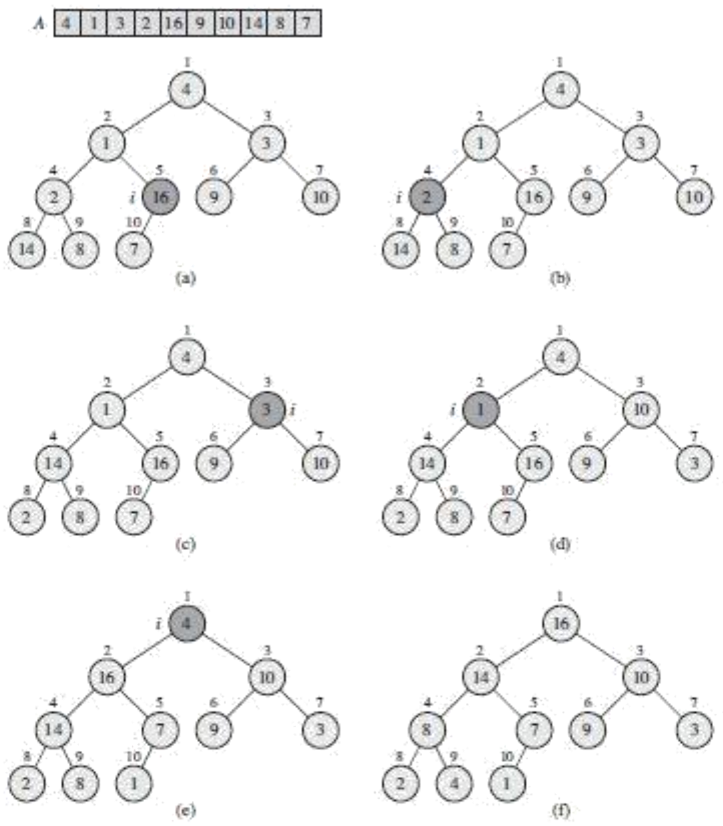


Figure: The operation of BUILD-MAX-HEAP, showing the data structure before the call to MAX-HEAPIFY in line 3 of BUILD-MAX-HEAP. *(a)* A 10- element input array *A* and the binary tree it represents. The figure shows that the loop index *i* refers to node 5 before the call MAX-HEAPIFY(*A*, *i*). *(b)* The data structure that results. The loop index *i* for the next iteration refers to node 4. *(c)-(e)* Subsequent iterations of the *for* loop in BUILD-MAXHEAP. Observe that whenever MAX-HEAPIFY is called on a node, the two subtrees of that node are both max-heaps. *(f)* The max-heap after BUILD-MAX-HEAP finishes.

1.5.3 The heapsort Algorithm

The heapsort algorithm starts to build max heap by using procedure BUILD-MAX -HEAP and then picks the root element which has the higher value. Then remove root value from the tree and built again it max heap. This process performs up to last value and sorted array is formed.

1.5.3.1 HEAPSORT(A)

1. BUILD-MAX-HEAP(A)
2. for i = length(A) downto 2
3. Exchange (A[1] with A[i]);
4. A.heap-size= A.heap-size -1;
5. MAX-HEAPIFY (A, 1);

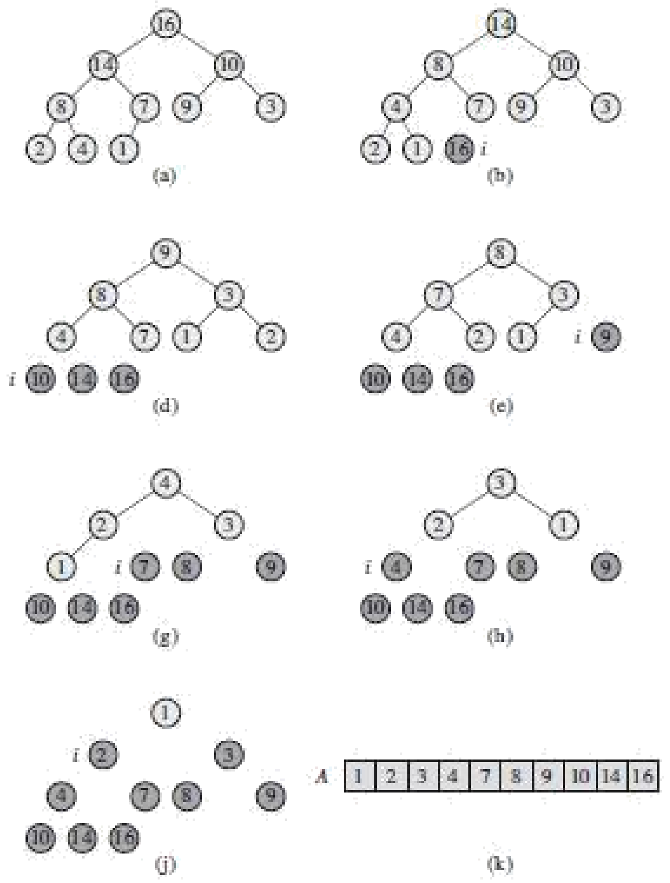


Figure: The operation of HEAPSORT. *(a)* The max-heap data structure just after it has been built by BUILD -MAX-HEAP. *(b)-(j)* The max-heap just after each call of MAXHEAPIFY in line 5. The value of *i* at that time is shown. Only lightly shaded nodes remain in the heap. *(k)* The resulting sorted array *A*.

1.6 Quick sort

Quicksort is also a divide-and-conquer algorithm. An unsorted array A taken in which p and r is the lower bound and upper bound of the elements respectivally.
Divide: The array A[p..r] is *partitioned* into two non-empty subarrays A[p..q] and A[q+1..r].
Invariant: All elements in A[p..q] are less than all elements in A[q+1..r].
Conquer: The subarrays are recursively sorted by calls to quicksort.
Combine: Unlike merge sort, no combining step: two subarrays form an already-sorted array.

The following procedure implements quicksort:

QUICKSORT(A, p, r)

```
1.  if (p < r)
2.  q = PARTITION(A, p, r)
```

```
3. QUICKSORT(A, p, q)
4. QUICKSORT(A, q+1, r)
```

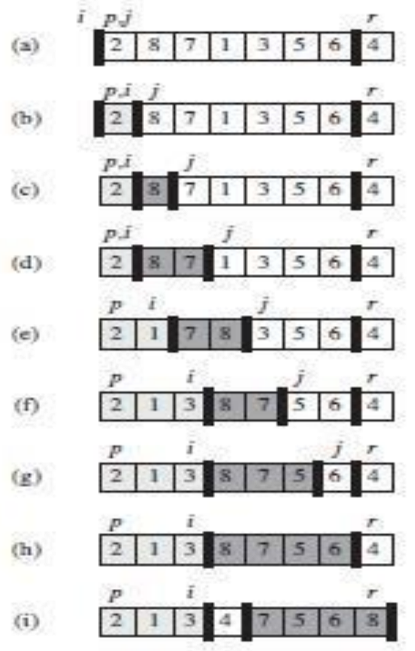
To sort an entire array A, the initial call is QUICKSORT (A, 1, A.length).

Partitioning the array:

The key to the algorithm is the PARTITION procedure, which rearranges the subarray A(p….r) in place.

PARTITION(*A*, *p*, *r*)

```
1 $x \leftarrow A[r]$
2 $i \leftarrow p - 1$
3 for $j \leftarrow p$ to $r - 1$
4 do if $A[j] \le x$
5 then $i \leftarrow i + 1$
6 exchange $A[i] \leftrightarrow A[j]$
7 exchange $A[i + 1] \leftrightarrow A[r]$
8 return $i + 1$
```



(a) i p,j r 2 8 7 1 3 5 6 4
(b) p,i j r 2 8 7 1 3 5 6 4
(c) p,i j r 2 8 7 1 3 5 6 4
(d) p,i j r 2 8 7 1 3 5 6 4
(e) p i j r 2 1 7 8 3 5 6 4
(f) p i j r 2 1 3 8 7 5 6 4
(g) p i j r 2 1 3 8 7 5 6 4
(h) p i r 2 1 3 8 7 5 6 4
(i) p i r 2 1 3 4 7 5 6 8

Figure: The operation of PARTITION on a sample array. Lightly shaded array elements are all in the first partition with values no greater than *x*. Heavily shaded elements are in the second partition with values greater than *x*. The unshaded elements have not yet been put in one of the first two partitions, and the final white element is the pivot. *(a)* The initial array and variable settings. None of the elements have been placed in either of the first two partitions. *(b)* The value 2 is "swapped with itself" and put in the partition of smaller values. *(c)-(d)* The values 8 and 7 are added to the partition of larger values. *(e)* The values 1 and 8 are swapped, and the smaller partition Grows. *(f)* The values 3 and 8 are swapped, and the smaller partition grows. *(g)-(h)* The larger partition grows to include 5 and 6 and the loop terminates. *(i)* In lines 7-8, the pivot element is swapped so that it lies between the two par titions.

1.6.1 Performance of Quicksort:

The running time of quicksort depends on whether the partitioning is balanced or unbalanced, which in turn depends on which elements are used for partitioning.

1.6.1.1 Worst Case Partitioning:

The worst-case behavior for quicksort occurs when the partitioning routine produces one subproblem with n - 1 elements and one with 0 elements. The recurrence for the running time is

$T(n)=T(n-1)+T(0) + \Theta(1)= \Theta(n^2)$ i.e $T(n) = \Theta(n^2)$

1.6.1.2 Best case partitioning:

in the best case it partition the array into equal sub arrays. The recurrence for balanced portioning is

$T(n) = 2T(n/2) + \Theta(n)= \Theta(n \lg n)$ i.e. $T(n) = \Theta(n \lg n)$

1.7 Shell sort

Shellsort, also known as Shell sort or Shell's method, is an in-place comparison sort. It can be seen as either a generalization of sorting by exchange (bubble sort) or sorting by insertion (insertion sort).The method starts by sorting elements far apart from each other and progressively reducing the gap between them.

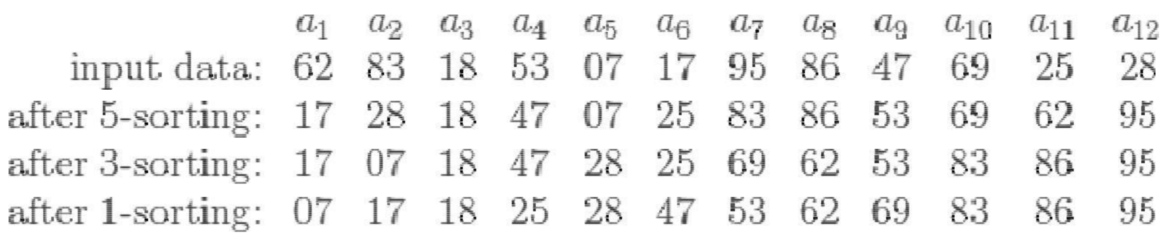
```
# Sort an array a[0...n-1].
gaps = [701, 301, 132, 57, 23, 10, 4, 1]

# Start with the largest gap and work down to a gap of 1
foreach (gap in gaps)
{
   # Do a gapped insertion sort for this gap size.
   # The first gap elements a[0..gap-1] are already in gapped order
   # keep adding one more element until the entire array is gap
   sorted for (i = gap; i < n; i += 1)
   {
      # add a[i] to the elements that have been gap sorted
      # save a[i] in temp and make a hole at position i
      temp = a[i]
      # shift earlier gap-sorted elements up until the correct location for a[i] is
      found for (j = i; j >= gap and a[j - gap] > temp; j -= gap)
```

```
        {
            a[j] = a[j - gap]
        }
        # put temp (the original a[i]) in its correct
        location a[j] = temp
    }

}
```

An example run of Shellsort with gaps 5, 3 and 1 is shown below



	a_1	a_2	a_3	a_4	a_5	a_6	a_7	a_8	a_9	a_{10}	a_{11}	a_{12}
input data:	62	83	18	53	07	17	95	86	47	69	25	28
after 5-sorting:	17	28	18	47	07	25	83	86	53	69	62	95
after 3-sorting:	17	07	18	47	28	25	69	62	53	83	86	95
after 1-sorting:	07	17	18	25	28	47	53	62	69	83	86	95

The first pass, 5-sorting, performs insertion sort on separate subarrays (a_1, a_6, a_{11}), (a_2, a_7, a_{12}), (a_3, a_8), (a_4, a_9), (a_5, a_{10}). For instance, it changes the subarray (a_1, a_6 , a_{11}) from (62, 17, 25) to (17, 25, 62). The next pass, 3-sorting, performs insertion sort on the subarrays (a_1, a_4, a_7, a_{10}), (a_2, a_5, a_8, a_{11}), (a_3, a_6, a_9, a_{12}). The last pass, 1-sorting, is an ordinary insertion sort of the entire array (a_1,..., a_{12}).
The complexity of this algorithm is O ($n^{1.25}$).

1.8 Sorting in linear time

The algorithms that can sort n numbers in O(n lg n) time is Merge sort and heap sort and achieve this upper bound in the worst case; Quick sort achieves it on average. These algorithms share an interesting property: the sorted order they determine is based only on comparisons between the input elements. We call such sorting algorithms comparison sorts. All the sorting algorithms introduced thus far are comparison sorts.
We shall prove that any comparison sort must make Θ(*n* lg *n*) comparisons in the worst case to sort *n* elements. Thus, merge sort and heap sort are asymptotically optimal, and no comparison sort exists that is faster by more than a constant factor.
Here three sorting algorithms-counting sort, radix sort, and bucket sort-that run in linear time.
These algorithms use operations other than comparisons to determine the sorted order.
Consequently, the Θ(*n* lg *n*) lower bound does not apply to them

1.8.1 Counting sort

Counting sort assumes that each of the n input elements is an integer in the range 0 to k, for some integer k. When k = O(n), the sort runs in ,Ɵ (n) time.

1.8.1.1 The algorithm:

1. Input: A[1..*n*], where A[j] ∈ {1, 2, 3, …, *k*}
2. Output: B[1..*n*], sorted (notice: not sorting in place)
3. Also: Array C[1..*k*] for auxiliary storage

```
COUNTING-SORT(A, B, k)
1 for $i \leftarrow 0$ to $k$
2 do $C[i] \leftarrow 0$
3 for $j \leftarrow 1$ to $length[A]$
4 do $C[A[j]] \leftarrow C[A[j]] + 1$
5 //$C[i]$ now contains the number of elements equal to $i$.
6 for $i \leftarrow 1$ to $k$
7 do $C[i] \leftarrow C[i] + C[i - 1]$
8 //$C[i]$ now contains the number of elements less than or equal to $i$.
9 for $j \leftarrow length[A]$ downto 1
10 do $B[C[A[j]]] \leftarrow A[j]$
11 $C[A[j]] \leftarrow C[A[j]] - 1$
```

1.8.1.2 Running time of Counting Sort

The for loop of lines 1-2 takes time $\Theta(k)$, the for loop of lines 3-4 takes time $\Theta(n)$, the for loop of lines 6-7 takes time $\Theta(k)$, and the for loop of lines 9-11 takes time $\Theta(n)$. Thus, the overall time is $\Theta(k+n)$. In practice, we usually use counting sort when we have $k = O(n)$, in which case the running time is $\Theta(n)$.

Example:

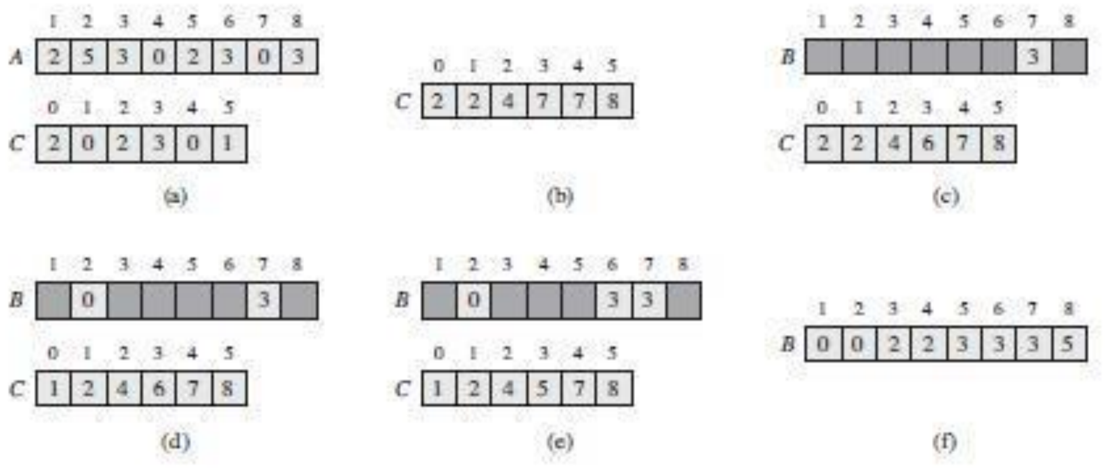


Figure: The operation of COUNTING-SORT on an input array *A*[1,….., 8], where each element of *A* is a nonnegative integer no larger than *k* = 5. *(a)* The array *A* and the auxiliary array *C* after line 4. *(b)* The array *C* after line 7. *(c)-(e)* The output array *B* and the auxiliary array *C* after one, two, and three iterations of the loop in lines 9-11, respectively. Only the lightly shaded elements of array *B* have been filled in. *(f)* The final sorted output array *B*.

1.8.2 Radix Sort

Radix sort solves the problem of card sorting—by sorting on the *least significant* digit first. The algorithm then combines the cards into a single deck, with the cards in the 0 bin preceding the cards in the 1 bin preceding the cards in the 2 bin, and so on. The process continues until the cards have been sorted on all d digits. Only d passes through the deck are required to sort. The algorithm is

```
RadixSort(A, d)
    for i=1 to d
       StableSort(A) on digit i
```

Given *n* *d*-digit numbers in which each digit can take on up to *k* possible values,
RADIXSORT correctly sorts these numbers in $\Theta(d(n + k))$ time.
Example:

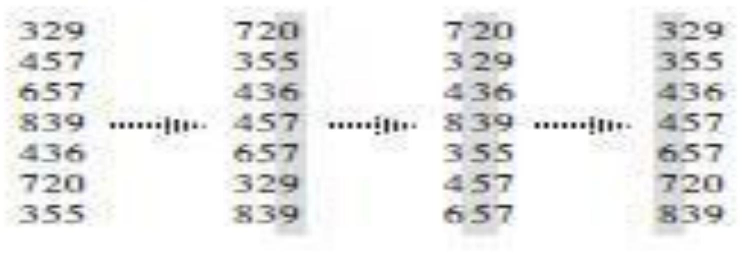


Figure: The operation of radix sort on a list of seven 3-digit numbers. The leftmost column is the input. The remaining columns show the list after successive sorts on increasingly significant digit positions. Shading indicates the digit position sorted on to produce each list from the previous one.

1.8.3 Bucket sort

Assumption: input is *n* reals from [0, 1)

Basic idea: Create *n* linked lists (*buckets*) to divide interval [0,1) into subintervals of size 1/*n*. Add each input element to appropriate bucket and sort buckets with insertion sort. Uniform input distribution → O(1) bucket size. Therefore the expected total time is O(n). These ideas will return when we study *hash tables.*

```
BUCKET-SORT(A)
1 $n \leftarrow length[A]$
2 for $i \leftarrow 1$ to $n$
3 do insert $A[i]$ into list $B[\lfloor n A[i] \rfloor]$
4 for $i \leftarrow 0$ to $n - 1$
5 do sort list $B[i]$ with insertion sort
6 concatenate the lists $B[0], B[1], \ldots, B[n - 1]$ together in order
```

Example:

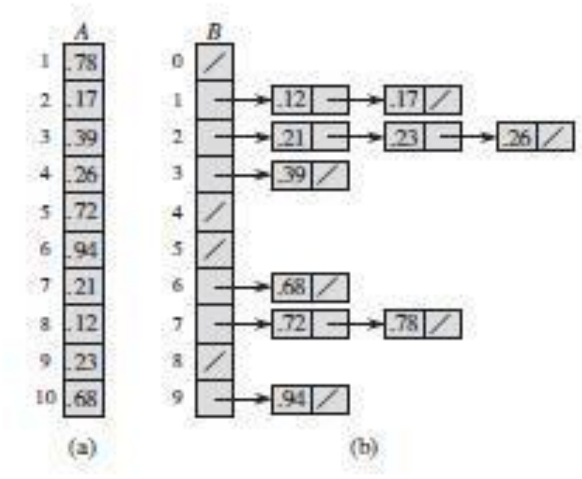


Figure: The operation of BUCKET-SORT for n= 10. **(a)** The input array A(1……..10). **(b)** The array B(0…… 9) of sorted lists (buckets) after line 8 of the algorithm. Bucket i holds values in the half-open interval [i/10,.i + 1/10). The sorted output consists of a concatenation in order of the lists B[0], B[1]…….B[9]

To analyze the running time, observe that all lines except line 5 take O (n) time in the worst case.