**Dictionary**

In-place

It means that you should update the original string rather than creating a new one

Loop Invariant

It means loop start from the end of the array S 🡪 for i = (S.length – 1) to 0 do {…}

**Mathematic problems**

Is mathematics consistent?

No.

Example

x+y+z=3

x+y+z=4

Is mathematics complete?

No

Example

Consider the following Axioms of NT

x (f(x) != 0)

x (x + 0 = x)

x (x.o = 0)

x (x < f(x) where f is Unary function i.e. f(x) = x + 1.

…………..

Unfortunately a complete list of axioms like above does NOT exist to make Number System / Arithmetic Complete.

Is mathematics decidable?

No, mathematics is not decidable.

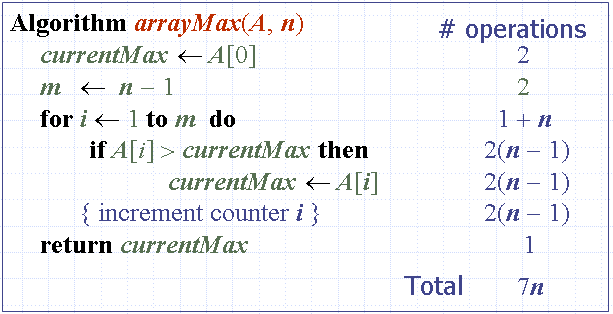
Example

This is can be verified by using the Halting Problem which states that given a program P, there is no way one can determine whether P will halt, terminate normally or will continue to run for ever. An algorithm to define the Halting problem exists but there is no algorithm to solve the Halting Problem.

**Primitive Operations**

* Performing an arithmetic operation (+, \*, etc)
* Comparing two numbers
* Assigning a value to a variable
* Indexing into an array
* Calling a method
* Returning from a method
* Following an object reference

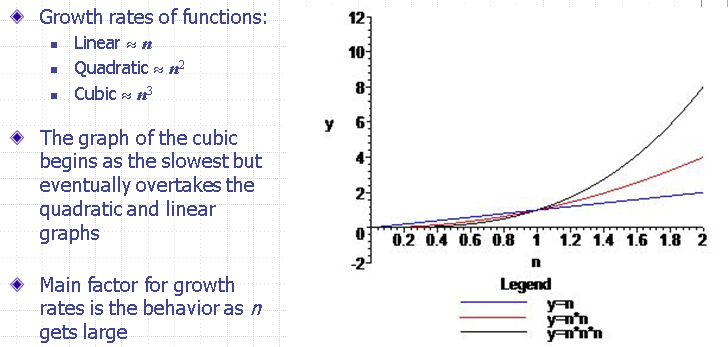
Example



**Grow Rate of running time**

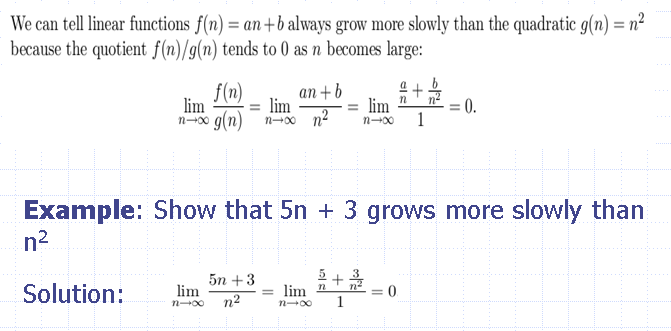
Changing the hardware / software environment

Affects T(n) by a **constant factor**, but does not alter the growth rate of T(n)

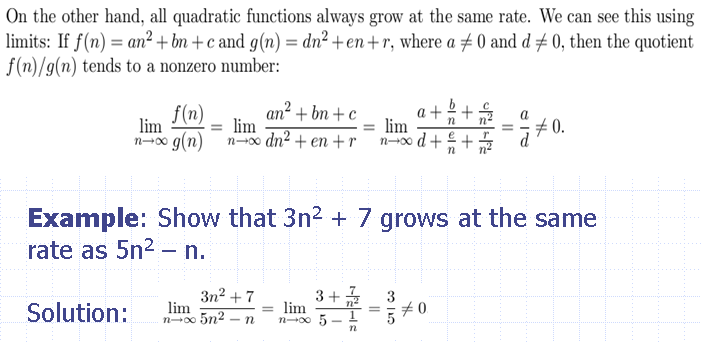


Detecting Growth Rates Using Limit

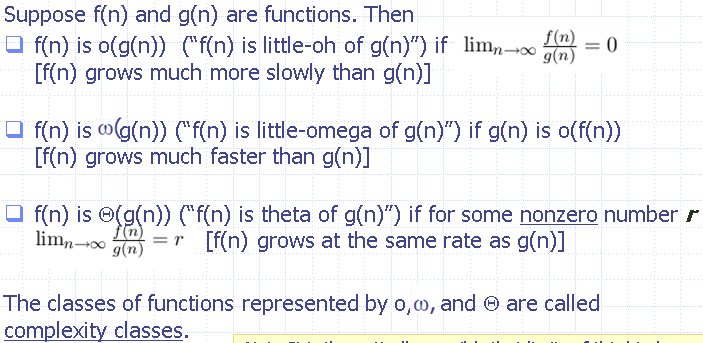
* If Limit = 0 as n becomes large then f(n) slower than g(n)



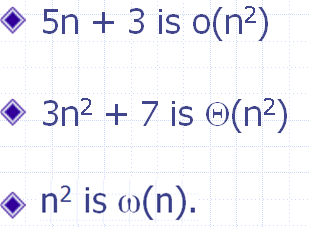
* If Limit = 0 as n becomes large then f(n) grow at the same rate with g(n)



**Theta, Little-Oh, Little-Omega**

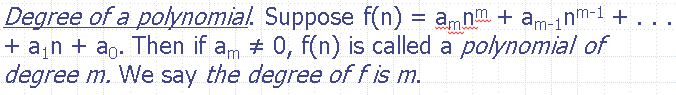


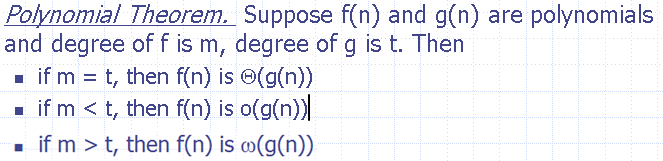
Example



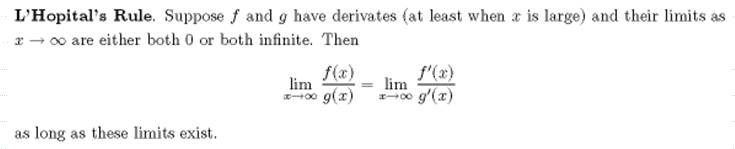
Polynomial Theorem (Định lý đa thức)

Degree of a polynomial (bậc của đa thức)

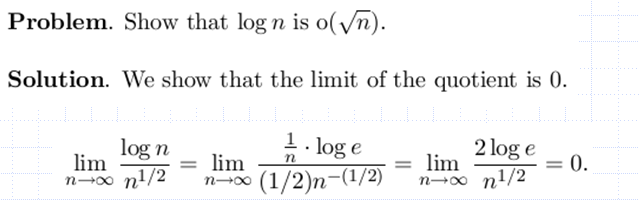




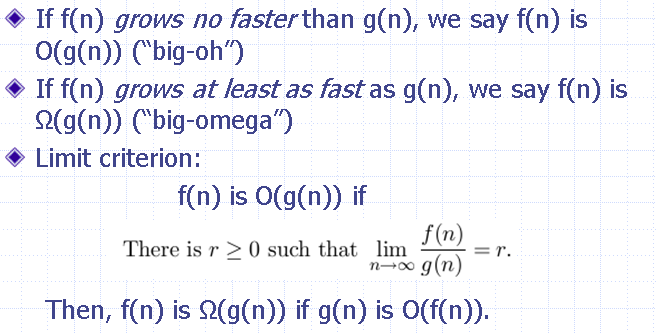
L’Hopital’s Rule



Example



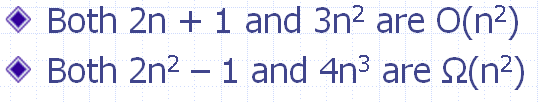
**Big-Oh & Big Omega**



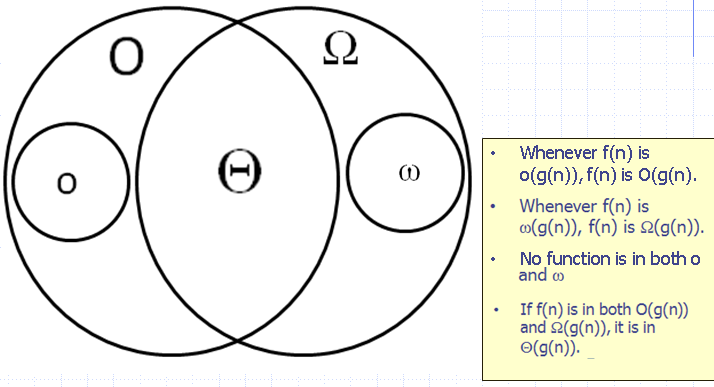
We have the limit of f/g = r

* If r >= 0 || r = infinite then O (g(n))
* If r <= 0 || r = -infinite then big Omega of g(n)

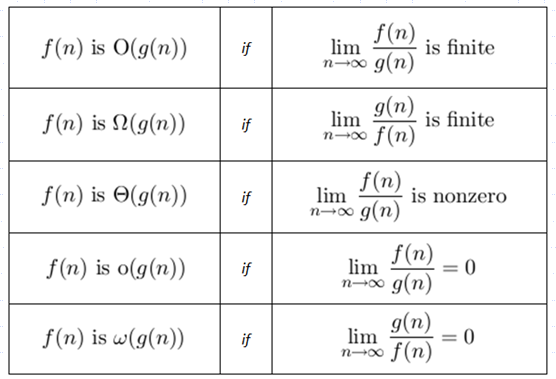
Example



Relationship between the Complexity Classes



Summary of Criteria for Determining Complexity



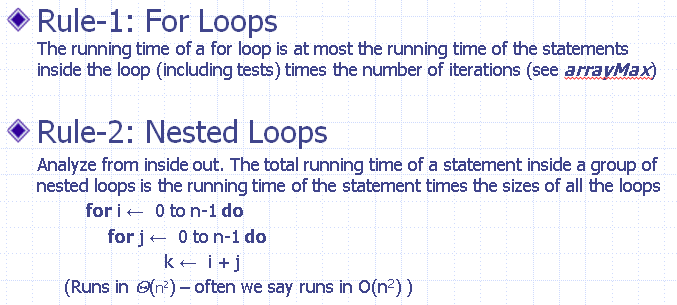
Asymptotic Algorithm (Tiệm cận của giải thuật)

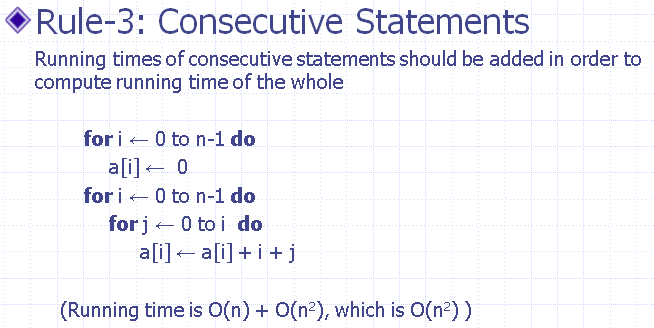
The asymptotic analysis of an algorithm determines which complexity class the running time belongs to

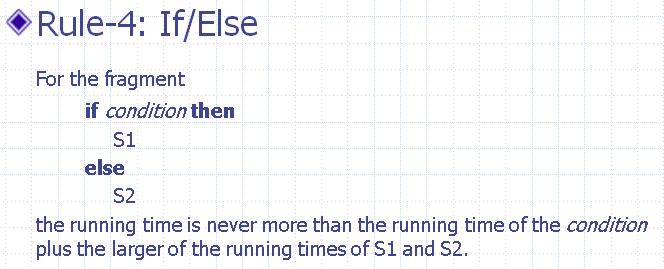
To perform the (worst-case) asymptotic analysis

* We find the worst-case number of primitive operations executed as a function of the input size
* We express this function using big-Oh notation (or one of its variants)

Basic rules for computing Asymptotic Running Times





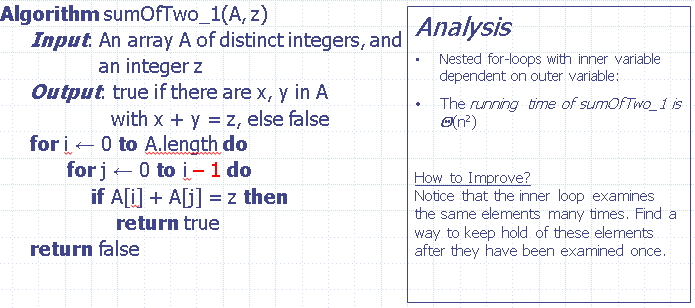


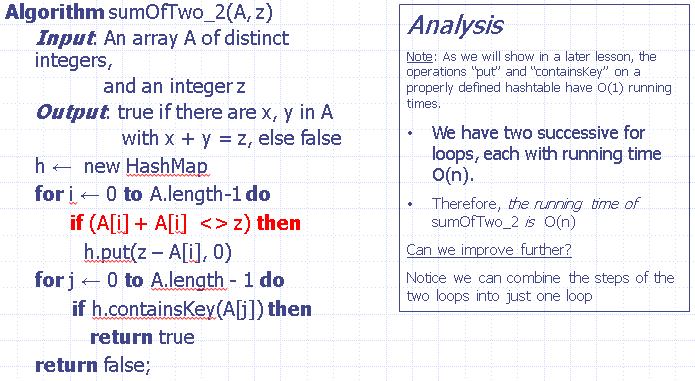
**The Sum of Two problem**

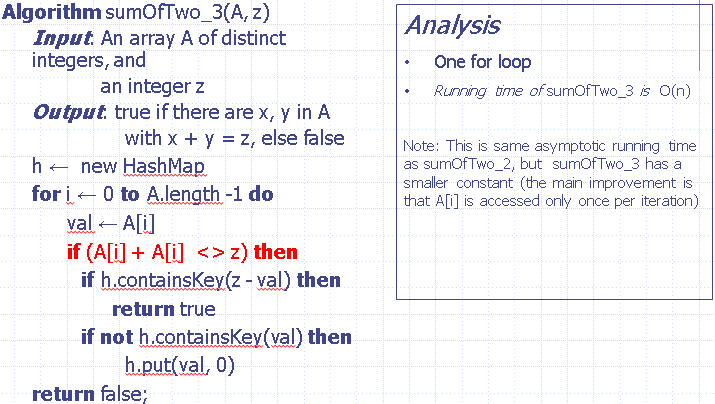
The problem

Given an array A of distinct integers and another integer z, determine whether A contains two different numbers x, y so that x + y = z

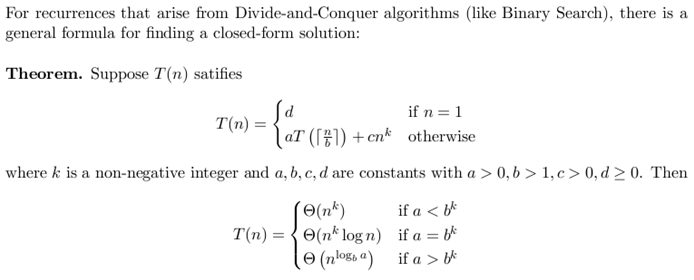
Solution

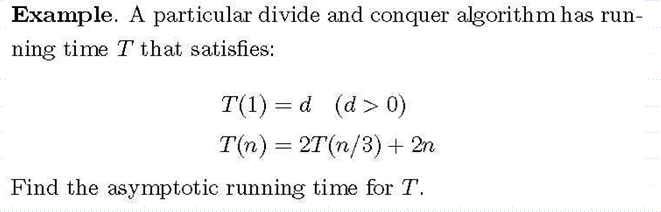


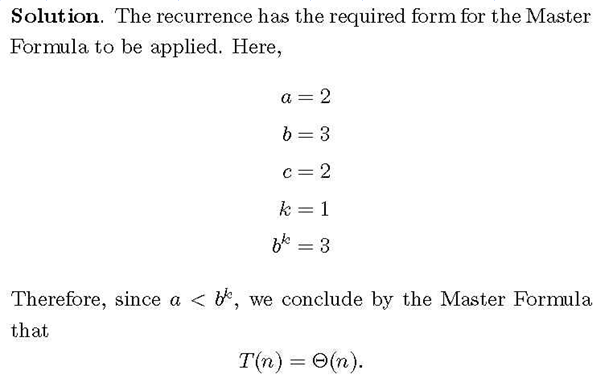




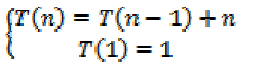
**Running Time of Recursive Algorithm: The Master Formula**







**Iterative Method**



Using iterative method, we have, T(n) = T(n-2) + (n-1) + n

T(n) = T(n-3) + (n-2) + (n-1) + n

T(n) = T(n-4) + (n-3) + (n-2) + (n-1) + n

T(n) = T(n-5) + (n-4) + (n-3) + (n-2) + (n-1) + n

So now rewrite these five equations and look for a pattern:

T(n) = T(n - 1) + n

T(n) = T(n-2) + (n-1) + n

T(n) = T(n-3) + (n-2) + (n-1) + n

T(n) = T(n-4) + (n-3) + (n-2) + (n-1) + n

T(n) = T(n-5) + (n-4) + (n-3) + (n-2) + (n-1) + n

Generalized recurrence relation at the kth step of the recursion:

T(n) = T(n - k) + (n – k + 1) + (n - k + 2) + … + (n – 1) + n

We have T(n-k) and we want T(1). So, we let n – k = 1. Also, solve for k, k = n – 1.

Now, plug this in all across the board:

T(n) = T(1) + 2 + 3 + … + (n-1) + n

At this step, using sum serials formula Sn = n (a1 + an) / 2

T(n) = n(n + 1) / 2 = O (n2)

Another example

Consider the following recurrence algorithm

procedure T(n, x)

if(n==1)

return true

else

{

x = x + n

Call T(n-1,x)

}

end procedure

**Write a recurrence equation for T(n)**

Solution:

T(n) = 1 if n = 1,

T(n) = 1 + T(n-1) if n > 1

**Solve recurrence equation using iterative method i.e. give an expression for the runtime T(n).**

Solution:

T(n) = T(n - 1) + 1

T(n) = ((T(n - 2) + 1) + 1

T(n) = ((T(n - 3) + 1) + 1) + 1

:

:

T(n) = T (n-k) + 1 + 1 + … + 1

T(n) = T (n - k) + k

Set n = k + 1 and use T(1) = 1, we get

**T(n) = 1 + n - 1 = O(n).**

**Halting problem**

**Is mathematics decidable? Explain the Halting problem in your own words (no need to prove).**

No, mathematics is not decidable. This is can be verified by using the Halting Problem which states that given a program P, there is no way one can determine whether P will halt, terminate normally or will continue to run forever. An algorithm to define the Halting problem exists but there is no algorithm to solve the Halting Problem.

**Amortized**

**Explain why Amortized analysis is better than Average Case analysis using probabilistic method.**

Amortized analysis uses the worst case timings of all available and takes the average. It uses Amortization cost which is set to be more than actual cost i.e. it ensures that there is always enough profit. No other input data are needed. On the other hand, Average Case analysis uses probabilities, associated equations (e.g. Expected Values). Enough number of data would need to be generated to calculate good probabilities and all associated functions, which is not always practical. Besides, it is expensive. Thus, Amortized Analysis is a better approach.

Example

**Amortized method**

Ci = cost of Ith operation

We have: **Ci = (if i is an exact power of 2) ? i : 1;**

We also have amortized cost ^Ci = charge each operation = $3

* If i is not an exact power of 2, pay $1 and have $2 as credit
* If i is an exact power of 2, pay $i and using stored credit

|  |  |  |  |
| --- | --- | --- | --- |
| Operation | Cost | Actual cost | Credit remaining = credit remaining of (ith-1) + (cost - actual cost) |
| 1 | 3 | 1 | 2 |
| 2 | 3 | 2 | 2 + (3 - 2) = 3 |
| 3 | 3 | 1 | 3 + (3 - 1) = 5 |
| 4 | 3 | 4 | 5 + (4 - 3) = 4 |
| 5 | 3 | 1 | 4 + (3 - 2) = 6 |
| 6 | 3 | 1 | 6 + (3 - 1) = 8 |
| 7 | 3 | 1 | 8 + (3 - 1) = 10 |
| 8 | 3 | 8 | 10 + (3 - 8) = 5 |
| 9 | 3 | 1 | 5 + (3 -1) = 7 |
| … | 3 | … | … |
| … | 3 | … | … |

The amortized cost is $3 per operation,

We have that

Then we have => credit = amortized cost – actual cost >= 0

Because the amortized cost of each operation is O(1) and the amount of credit never goes negative, the total cost of n operation is O(n)

**Average Case**

Problem

* Generating a complex set of data and calculating probability
* Complex computation using probability statistics

Benefit

* More realistic timing estimate

**Bubble Sort**

Pseudo code

void sort(){

int len = arr.length;

for(int i = 0; i < len; ++i) {

for(int j = 0; j < len−1; ++j) {

if(arr[j] > arr[j+1]){

swap(j,j+1);

}

}

}

}

void swap(int i, int j){

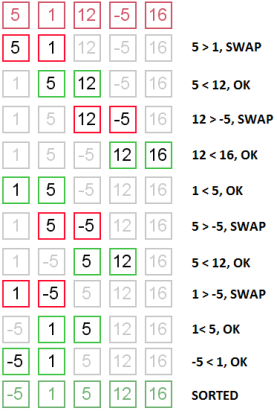
int temp = arr[i];

arr[i] = arr[j];

arr[j] = temp;

}

Illustration



Time Complexity

* In this implementation, there is no “best” or “worst” case
* Because there are two loops, nested, both depending on n it is (n2)

Possible Improvements

* It is possible to implement BubbleSort slightly differently so that in the best case (which means here that the input is already sorted), the algorithm runs in (n) time.
* As the Loop Invariant shows, at the end of iteration i, the values in arr[n-i-1] through arr[n-1] are in final sorted order. This observation can be used to shorten the inner loop. The result is to cut the running time in half (though it must still be (n2)).

**Selection Sort**

Pseudo code

void sort(){

int len = arr.length;

int temp = 0;

for(int i = 0; i < len; ++i) {

int nextMinPos = minpos(i,len-1);

swap(i,nextMinPos);

}

}

void swap(int i, int j){

int temp = arr[i];

arr[i] = arr[j];

arr[j] = temp;

}

int minpos(int bottom, int top){

int m = arr[bottom];

int index = bottom;

for(int i = bottom+1; i <= top; ++i) {

if(arr[i]<m){

m=arr[i];

index=i;

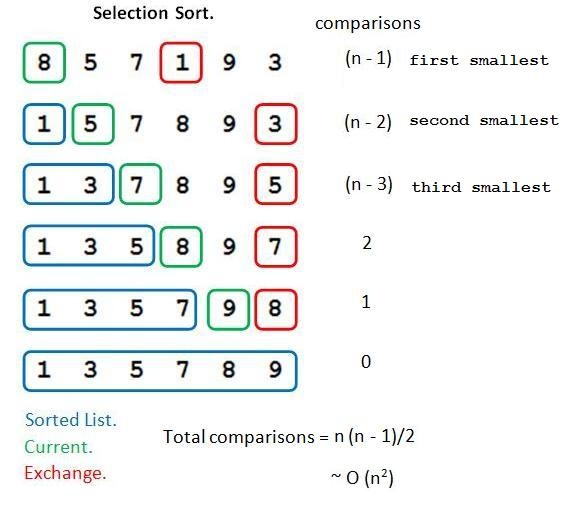
}

}

return index;

}

Illustration



Time Complexity

* In this implementation, there is no “best” or “worst” case
* Because there are two loops, nested, both depending on n it is (n2)

**Insertion Sort**

Pseudo code

void sort(){

int len = arr.length;

int temp = 0;

int j = 0;

for(int i = 1; i < len; ++i) {

temp = arr[i];

j=i;

while(j>0 && temp < arr[j−1]){

arr[j] = arr[j−1];

j−−;

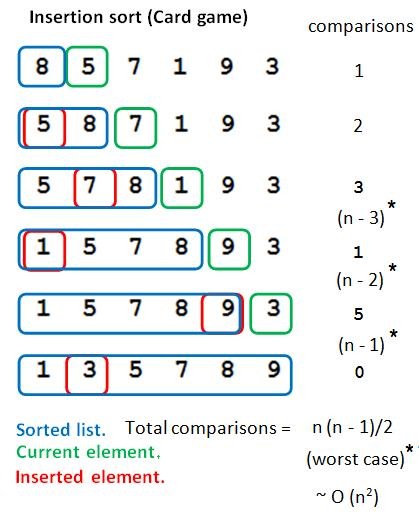
}

arr[j]=temp;

}

}

Illustration



Time Complexity

* Best-Case Analysis: is O (n). The best case for InsertionSort occurs when the input array is already sorted.
* Worst-Case Analysis. Since there are two loops, nested, even in the worst case, the running time is only (n2). The worst case for InsertionSort occurs when the input array is reverse-sorted.
* Average-Case Analysis. It is reasonable to expect that typically, the inner while loop will not work as hard as it does in the worst-case

**Inversion-Bound Sorting Algorithms**

In an array **arr** of integers, an inversion is a **pair (arr[i], arr[j])** for which **i < j and arr[i] > arr[j]**.

Example

The array arr = {34, 8, 64, 51, 32, 21} has nine inversions: (34, 8), (34, 32), (34, 21), (64, 51), (64, 32), (64, 21), (51, 32), (51, 21), (32, 21)

Theorem (Number of Inversions Theorem)

Assuming that **input arrays** contain **no duplicates** and values are **randomly generated**, the **expected number of inversions** in an **array of size n** is **n(n−1)/4**.

Corollary

The average-case running time of this algorithm acting on arrays of distinct elements is Omega (n2)

Definition

A **sorting algorithm** that always performs at least as **many comparisons** as there are **inversions** on any input array arr is called an **inversion-bound algorithm**

**Note:** We assume all elements of input arrays are distinct, BubbleSort, SelectionSort, and InsertionSort can all be shown to be inversion-bound.

**LibrarySort**

Their paper is entitled INSERTION SORT is O(n log n).)

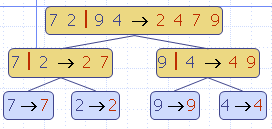
Their algorithm achieves average case running time of O(n log n).

**MergeSort**

Sorting algorithm

Devide & Conquer

Illustrate



Pseudo code

**Algorithm** ***mergeSort***(***S***)

**Input** sequence ***S*** with ***n***

**Output** sequence ***S*** sorted

**if** ***S.size***() **>** 1 **then**

(***S***1, ***S***2) ← ***partition***(***S***, ***n***/2)

***mergeSort***(***S***1)

***mergeSort***(***S***2)

***S*** ← ***merge***(***S***1, ***S***2)

**return** *S*

**Algorithm** ***merge***(***A, B***)

**Input** sorted sequences ***A*** and ***B*** with  
 ***n***2 integers each

**Output** sorted sequence ***S*** of ***A*** ∪ ***B***

***S*** ←empty sequence

**while** ¬***A.isEmpty***() **∧**¬***B.isEmpty***() **do**

**if** ***A.first***() <= ***B.first***() **then**

***S.insertLast***(***A.remove***(***A.first***()))

**else**

***S.insertLast***(***B.remove***(***B.first***()))

**while** ¬***A.isEmpty***() **do**

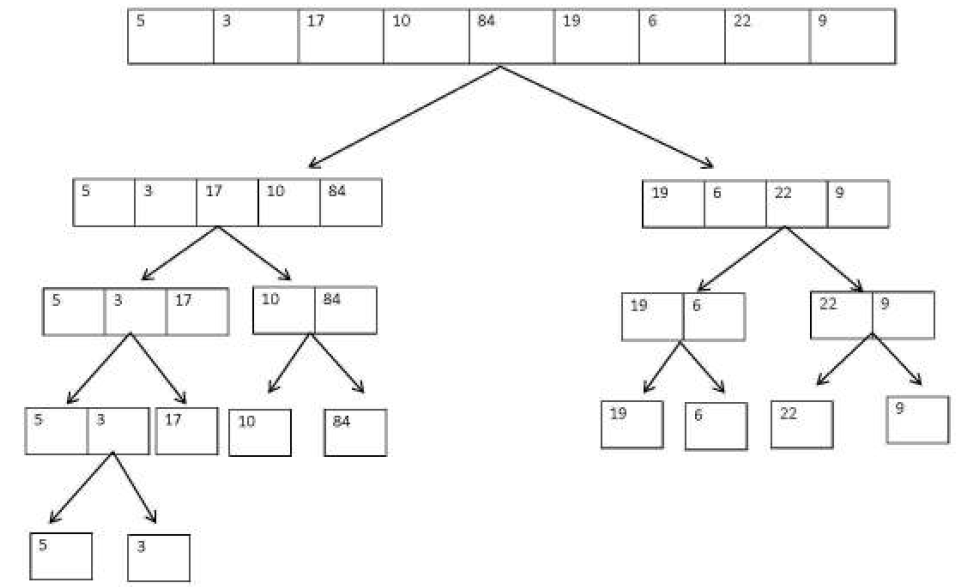
***S.insertLast***(***A.remove***(***A.first***()))

**while** ¬***B.isEmpty***() **do**

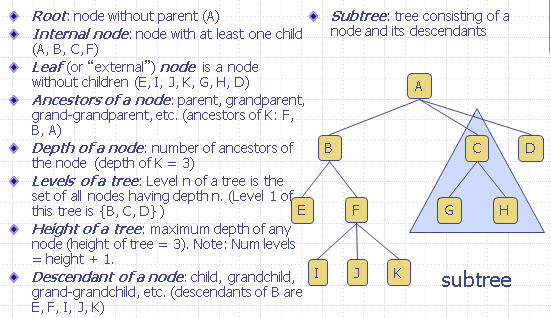
***S.insertLast***(***B.remove***(***B.first***()))

**return *S***

Example

****

Tree Terminology



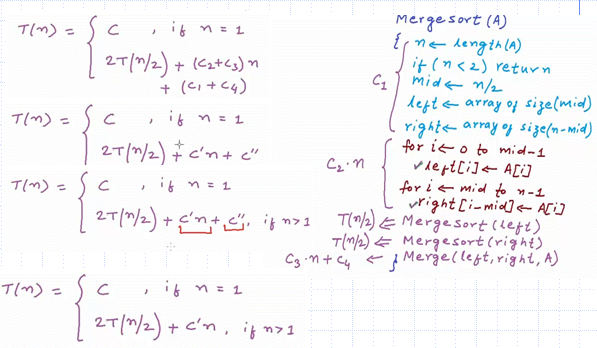
Handling of duplicates

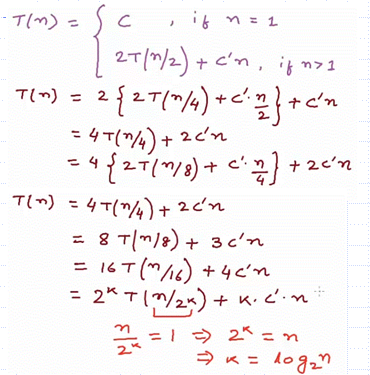
Stable sorting does not change the order of duplicates

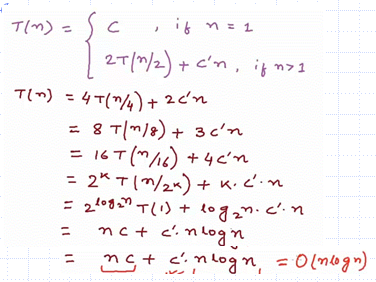
Stability

Suppose S = <(k0,e0), (k1,e1),…,(kn,en)> is a list of pairs with keys k0, k1, …, kn. A sorting algorithm is stable if, whenever it is the case that (ki, ei) precedes (kj, ej) before sorting (so that i < j) and ki = kj, then it continues to be true after sorting by keys that the pair (ki, ei) precedes (kj, ej)

Time Complexity: **O (n log n)**





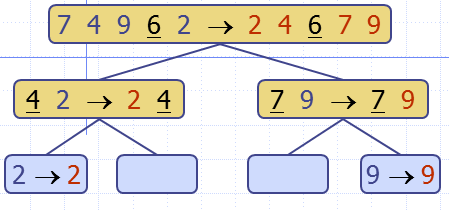


**QuickSort**

Sorting algorithm

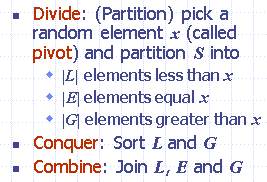
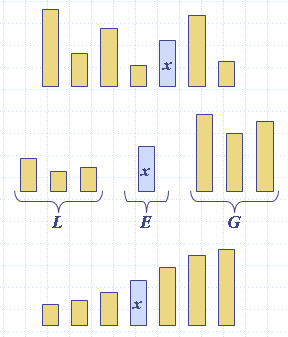
Devide & Conquer

Illustrate



Random version

Concept

****

Pseudo code

**Algorithm** ***partition***(***S,*** ***p***)

**Input** sequence ***S***, position ***p*** of pivot

**Output** subsequences ***L,*** ***E, G*** of the   
 elements of ***S*** less than, equal to,  
 or greater than the pivot, resp.

***L,*** ***E, G*** ←empty sequences

***x*** ← ***S.removeElementAt***(***p***)

**while** ***S.isEmpty***()

***y*** ← ***S.removeFirst()***

**if** ***y*** < ***x***

***L.insertLast***(***y***)

**else if *y*** = ***x***

***E.insertLast***(***y***)

**else** { ***y*** > ***x*** }

***G.insertLast***(***y***)

**return *L,*** ***E, G***

**Algorithm** ***quickSort***(***S***)

**Input** sequence ***S***

**Output** S in sorted order  
if(|S|=0 or |S|=1) then **return** S

***p*** ← ***pickPivot()***

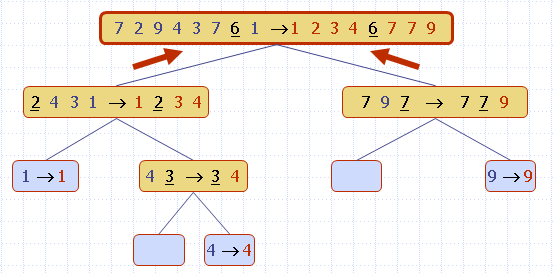
(***L,E,G***) ← ***partition***(***S,p***)

***quickSort***(***L***)

***quickSort***(***G***)

**return** ***L* U *E*** U ***G***

Example

****

Worst case running time

Occurs when the pivot selected is always the unique minimum (or maximum) element

Sum of ***n*** + (***n*** - 1) + … + 2 + 1 => Ω(n2)

Expected running time

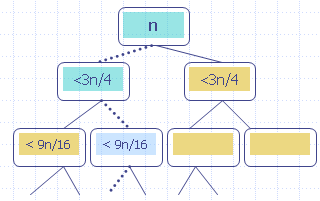
Consider a recursive call of quick-sort on an array of size **n**

* Good recursive call: the sizes of L and G are each less than **3n/4** (normal division)
* Bad recursive call: one of L and G has size greater than or equal to **3n/4**

A recursive call is good with probability at least 1/2 [**NOTE**: alg is randomized]

Good case running time: is O (n)

Average case running time: is O (n log n)



In-place version

Pseudo code

**Algorithm** ***inPlaceQuickSort***(***S,*** ***l,*** ***r***)

**Input** array ***S***, positions ***l*** and ***r***

**Output** array ***S*** with the  
 elements from positions ***l*** to ***r*** rearranged in increasing order

**if** ***l*** > ***r***

**return**

***k*** ←a random integer between ***l*** and ***r***

***swap(k,r) //place pivot at the right***

***x*** ← ***S.elemAtPos***(***r***) **//*the pivot***

***i*** ← ***inPlacePartition***(***x***) //new pos of piv

***swap(r,i)*** //place pivot in proper location

***inPlaceQuickSort***(***S,*** ***l,*** ***i***  1)

***inPlaceQuickSort***(***S,*** ***i***  1***,*** ***r***)

**Algorithm** ***inPlacePartition***(***x***)

1. Move pivot to the far right (pos = r)

2. Begin with pointers i, j with i at pos l and j at pos r – 1 (for readability, we let l = 0 in this discussion).

3. Move i to the right past all values < pivot

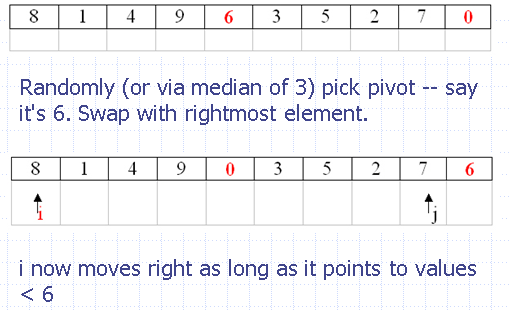
4. Move j to the left past all values > pivot

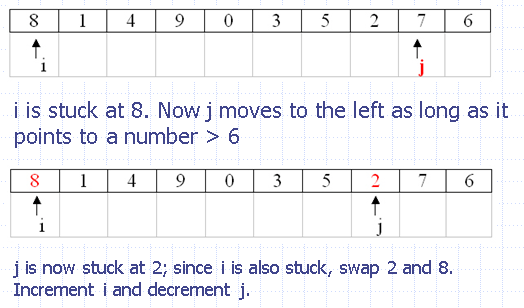
5. If stuck (neither i nor j can move further), then swap the values at i, j, and repeat 3, 4, allowing i to move one to the right and j to the left

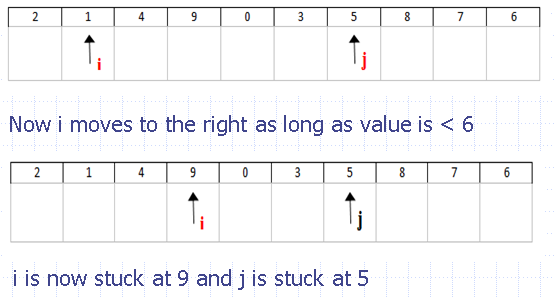
6. After this step, if j has crossed i (moved to the left of i), or i has crossed j (moved to the right of j), then stop

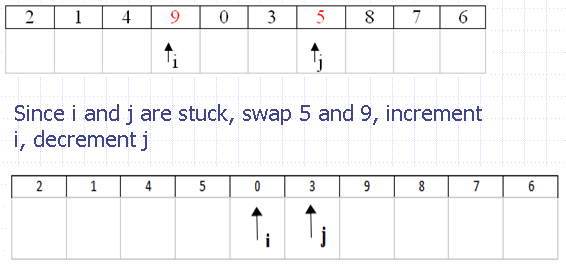
7. After stopping, swap positions of pivot and value at position i

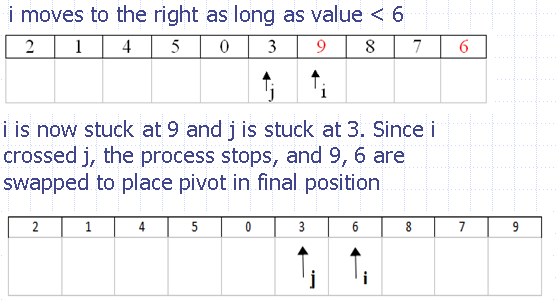
Illustrate











Now elements < pivot 6 are to the left of 6 and elements > pivot 6 are to the right of 6

----------------------------------------------------------

Note: in case both i and j encounter values equal to the pivot, after a swap, they will still be pointing to a value equal to a pivot. For this reason, the rule says: "After a swap, i is automatically moved ahead 1 step and j is moved back"

Handling duplicates

In the partition algorithm, if the left pointer encounters a value equal to the pivot, it halts; and the right pointer behaves in the same way. This strategy has the effect of evenly distributing duplicates into the left and right halves of the partition

Non-stable

But the above strategy also makes QuickSort a non-stable sorting algorithm

Example

Assume the pivot always happens to be value at the far right

Original array: (1, a), (1, b), (1, c), (1, d)

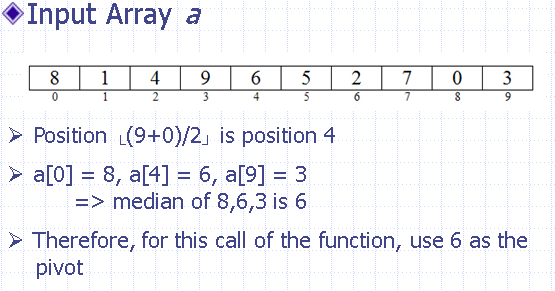
Using the algorithm, one discovers that after the first partition step

Sorted array: **(1, c)**, (1, b), **(1, d)**, **(1, a)**

Choices of Pivot

* Choosing pivot at **random**. Usually a good choice. Repeated calls to random number generator could slow it down a little.
* Choosing **first or last element** as pivot. This is a dangerous approach: using first element as pivot when data is already sorted leads to worst case. If data is known to be random (or is randomized), this is a good choice.
* **Median of Three**. Many consider this the best alternative. If i = lower pos, u = upper pos, pick the median of elements at positions i, u, and └(i+u)/2┘

Example: median of three



**Note:** The **probability** that QuickSort, running on an array of length **n**, has a **worst-case** running time is < **1/n2**

**QuickSelect**

A randomized selection algorithm for finding the kth smallest element in a list

Pseudo code

**Algorithm** ***QuickSelect***(***S,k***)

**Input** sequence ***S,*** rank *k*

**Output** *k*th smallest element of S

***p*** ← ***pickPivot()***

(***L,E,G***) ← ***partition***(***S,p***)

**if** |L| < k ≤ |L| + |E| **then**

**return** any element of ***E***

**else if** k ≤ |L| **then**

**return** ***QuickSelect***(***L***, k)

**else** {k > |L| + |E|}

**return** ***QuickSelect***(***G***, k – |***L***| – |***E***|)

**Algorithm** ***partition***(***S,*** ***p***)

**Input** sequence ***S***, position ***p*** of pivot

**Output** subsequences ***L,*** ***E, G*** of the   
 elements of ***S*** less than, equal to,  
 or greater than the pivot, resp.

***L,*** ***E, G*** ←empty sequences

***x*** ← ***S.elementAt***(***p***)

**while** ¬***S.isEmpty***()

***y*** ← ***S.remove***(***S.first***())

**if** ***y*** < ***x***

***L.insertLast***(***y***)

**else if *y*** = ***x***

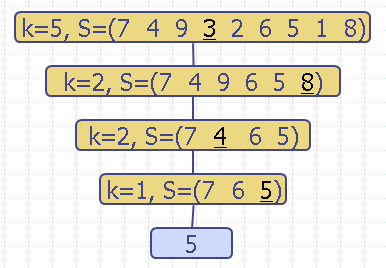
***E.insertLast***(***y***)

**else** { ***y*** > ***x*** }

***G.insertLast***(***y***)

**return *L,*** ***E, G***

Quick-Select visualization



Worst case running time

Occurs when the input array is sorted, k = 1, and the pivot is always chosen to be the rightmost element

1 + 2 + 3 + … + (n-1), which is Θ(n2)

Expected running time

Consider a recursive call of quick-select on a sequence of size **n**

* **Good self-call**: the sizes of L and G are each less than **3n/4**
* **Bad self-call**: one of L and G has size greater than or equal to **3n/4**

**A call is good** with probability ½



Average-Case running time: is **O(n)**

Example

S = {**1, 5, 23, 0, 8, 4, 33**}, compute the **5th** smallest element. Assume that the **rightmost** element is used as the pivot in each case

Solution

QS (1, 5, 23, 0, 8, 4, 33), k = 5

L = [1, 5, 23, 0, 8, 4], E = [33], G = [ ]

k = 5 <= |L|

QS (1, 5, 23, 0, 8, 4) , k = 5

L = [1, 0], E = [4], G = [5, 23, 8]

k' = k - |L| - |E| = 5 – 2 - 1 = 2

QS (5, 23, 8), k = 2

L = [5], E = [8], G = [23]

|L| < k ≤ |L| + |E|, so **return 8**

Read more: ModifiedQuickSelect(S, k)

**Bucket Sort**

Context

Suppose we have an array A of **n** distinct integers A= {a1, a2… an} in range of [0 – **m**-1]

Sorting strategy

* Create an array Bucket of size m and all its entries initialized to 0. Create an output array B of size n (requires **O (m + n)** )
* Scan A, when A[i] is encountered, increment the value bucket A[i] (requires **O (n)** )
* Scan Bucket[]. For each j < m for which bucket[j] > 0, copy A[j] into next available slot in B. (requires **O (m)** )

Therefore, BucketSort runs in O (m + n). **When m is O (n), BucketSort runs in O (n)**

Example

Given n integers a1, a2, …, an in the range [0, m-1], possibly with duplicates, and n matched objects o1, o2, …, on. We sort array A whose elements are pairs (a1,o1), (a2,o2),…,(an,on).

Pseudo code

**Algorithm** ***bucketSort***(***A,*** ***m***)

**Input** array ***A*** of (key, element) items with keys in the range [0, ***m*** 1]  
 **Output** array ***B*** sorted by increasing keys

***bucket*** ←array of ***N*** empty lists

**for *i*** ← 0 **to** ***n*** 1

(***k***, ***o***) ← ***A[i]***

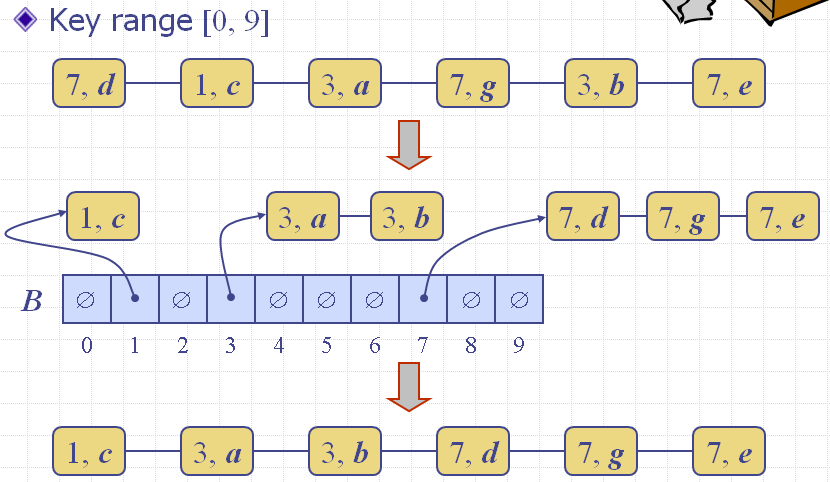
***bucket***[***k***]***.insertLast***((***k***, ***o***))

**for *j*** ← 0 **to** ***m*** 1

**while** ¬***bucket***[***j***]***.isEmpty***()

***(k, o)*** ← ***bucket***[***j***]***.removeFirst***()

***B.insertLast***((***k***, ***o***))



**Radix Sort**

Context

Radix-sort is a generalization of BucketSort that uses multiple bucket arrays. BucketSort doesn’t work because range is too big – would run in Ω(n2)

Example

Sort {48, 1, 6, 23, 37, 19, 21}

Strategy

* + is to use 2 bucket arrays, each of size 7 (7 is the radix)
  + Based on observation that every **k** in [0,48] can be written:  
     **k** = 7**q** + **r** where 0 ≤ **q** < 7, 0 ≤ **r** < 7
  + Procedure
    - Pass #1: Scan initial array and place values in the “remainders” bucket r[] – put x in r[i] if x % 7 = i. (Need to assume bucket array consists of lists)
    - Pass #2: Scan r[], reading from front of each list to back, and place values in the “quotients” bucket q[] – put x in q[i] if x/7 = i.
    - Output: Scan q[], again reading lists front to back

Illustrate

Keys: 48, 1, 6, 23, 37, 19, 21

Pass #1: **Key % 7**

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| i | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
| r[i] | 21 | 1 | 23 |  |  | 19 | 48 |
| 37 | 6 |

Pass #2: **Key / 7**

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| j | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
| q[j] | 1 |  | 19 | 21 |  | 37 | 48 |
| 6 | 23 |

Sorted order: 1, 6, 19, 21, 23, 37, 48

Running time is O(d(n+m)) where

* d = # bucket arrays (in example d=2)
* n = size of initial array (example: 7)
* m = size of each bucket array (ex: 7)

RadixSort runs in O(n) whenever the number of bucket arrays is O(1), and size of each bucket array is O(n).

Using 3 Bucket Arrays

Generalizing previous example, we observe that whenever k belongs to [0, 342] (note 342 = 73 – 1), then we can write  
 **k = 49q1 + 7q2 + r**

where 0 ≤ q1 < 7, 0 ≤ q2 < 7, 0 ≤ r < 7.

Example: 340 = 49\*6 + 7\*6 + 4

Therefore to handle sorting approx 7 elements in the range [0, 342], create bucket arrays r[], q1[], q2[], and compute,for each x in input array, values x%7, (x/7)%7, x/49, respectively

Example

RadixSort of S = {125, 27, 729, 1, 27, 8, 64, 343, 216}, using radix = 9

Observation

Because 729 is max number in S and 729 = 81 \* 9, therefore, we need to change radix from 9 to 10, we have

**k = 100q1 + 10q2 + r**, where 0<= q1 < 10, 0<= q2 < 10, 0<= r < 10

Procedure

With **x** belongs **S** and **x** is inserted first into the slot **x%10** in **r[]** as below

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |  |  | 27 |  |  |
| r[i] |  | 1 |  | 343 | 64 | 125 | 216 | 27 | 8 | 729 |
| i | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |

With **y** belongs **r[]** and **y** is inserted first into the slot (**x/10)%10** in **q2[]** as below

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  |  |  | 729 |  |  |  |  |  |  |  |
|  |  |  | 27 |  |  |  |  |  |  |  |
|  | 8 |  | 27 |  |  |  |  |  |  |  |
| q2[j] | 1 | 216 | 125 |  | 343 |  | 64 |  |  |  |
| j | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |

With **z** belongs **q2[]** and **z** is inserted first into the slot **z/100** in **q1[]** as below

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | 64 |  |  |  |  |  |  |  |  |  |
|  | 27 |  |  |  |  |  |  |  |  |  |
|  | 27 |  |  |  |  |  |  |  |  |  |
|  | 8 |  |  |  |  |  |  |  |  |  |
| q1[z] | 1 | 125 | 216 | 343 |  |  |  | 729 |  |  |
| z | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |

Sorted order: 1, 8, 27, 27, 64, 125, 216, 343, 729

**More questions**

