Design and Analysis of Algorithm

Yu-Tai Ching
Department of Computer Science
National Chiao-Tung University

- Lecture in English,
- Book, Introduction to Algorithms, 3rd edition, The MIT Press, by Cormen, etc.
- programming assignment.
- Maybe some quizzes, homework.
- Midterm exam and Final exam. 70%.

- We have a roll call for each class, purpose is to make sure who are in the class. Just in case there is confirmed case of new-coronavirus.
- Pickup a piece of paper I prepare for you, write down your id (or name) and the seat you take.
- Try to sit "as sparse as possible".
- And make sure I can see you, because I will take a picture of you. Just in case, I have to know the neighbors of a person.
- Wash your hands with soap before and after class, to protect others and yourself.

- An introduction, sorting algorithms, asymptotic notations, recursion,
- balance tree, red-black tree,
- divide and conquer, randomized algorithm (quick sort), prune and search, greedy approach, dynamic programming, amortized analysis,
- graph algorithms,
- lower bound to a computational problem, decision tree model, reduction,
- NP
- hopefully, parallel algorithm, ...

Algorithm

- Directions to solve a problem
- To build a DIY furniture
- How to come to NCTU from Taipei
- "directions"=steps

Algorithm

- In Computer Science, computer algorithms defined under the RAM model
- RAM, Random Access Machine, there are finite number of instructions, $+$, -, *, … . These instructions are basic enough (assembly language of X86). There are infinite number of memory, direct (random) access, retrieve and store in constant time.
- An algorithm, a sequence of finite number instructions, if follows, accomplish a specific task (a computational problem), (*Solve a problem*).

A Computational Problem, Sorting Problem

- Sorting Problem: Given n numbers, determine a permutation so that they are arranged in non-decreasing order.
- Given n=5 numbers 1, 10, 9, 4, and 5, this is an "instance" for the sorting problem.
- You design a sequence of instructions (instruction that can fit into machine instructions). Run the instructions (each instruction = a step) to solve the problem, (to arrange the numbers). Correct for all possible input *instances* of any size (how can you make sure?). Then you have a correct algorithm to sorting problem.

Algorithm and Program

- Algorithm will be translated into program to fit into a computer, or
- a program is an implementation of an algorithm
- Correctness of algorithm based on mathematics,
- correctness of algorithm doesn't imply the correctness of the program.

Performance of an Algorithm

- How to evaluate the performance of an algorithm?
- Transfer algorithm to a program, run the program, and record the time?
- Time depends on a particular input instance.
- Time depends on the machine.

Performance of an Algorithm

- Under RAM model
- Recall that, operators are basic enough, thus can be done in constant time; memory access in constant.
- Calculate the number of operations required and present the time required as a function of the input size.

Performance of an Algorithm

- An insertion sort example draw a picture
- Given n numbers, stored in an array,
- initially, the first one is sorted,
- each iteration, we increase the length of the sorted list by 1.
- After the ith iteration, we have a sorted sequence of length i.

Performance of an Algorithm

- How many steps does it take for the ith iteration?
- two parameters,
- *How many are compared?*
- How many basic operations does it take to compare one? To compare one, load from memory, compare, if $>$ key then store in the next position, if i$<$ key, done. Constant number of steps, c_1, each step takes constant time, c_2 (clock cycle), one move takes $c_1 + c_2$ time, it still constant.

Performance of an Algorithm

- How many are compared
- best case, compare one and no stores are required.
- worst case, each time you have a key which is smallest among the previous sorted list, and thus you have to compare all.
- generally follow the *worst case* convention.

Performance of an Algorithm

- In the worst case, the ith iteration needs $i-1$ comparisons and moving i data (ignore constant)

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

Performance of an Algorithm

- Correctness of the algorithm
- Obvious when the length if 1 and 2. Suppose it is correct to insertion sort i numbers, we have sorted sequence of length i. To process the $i+1$ number k, if $k >$ A[i], done (sorted). If k $<$ A[i], A[i] moves to A[$i+1$], we are inserting key to a sorted sequence of length i. Since the previous $n-1$ numbers are sorted, by induction, the algorithm correctly sort n numbers.

Description of an Algorithm

- Pseudo code
- PASCAL or C liked code
- Mixed with natural language
- many details are ignored

Performance of an Algorithm

- Another example, Selection Sort
- first iteration, find smallest from n numbers, n steps,
  2nd iteration, find smallest from $n-1$ numbers, $n-1$ steps,
  ...
  Find the smaller from the 2, and finally move the last to output list

$$\sum(n-1) = \frac{(n+1)*n}{2}$$

Performance of an Algorithm

- Which one is more efficient, the insertion sort or the selection sort?
- Why?
- Is there a way to accurately convey this message?

Asymptotic Notation Θ notation
Θ-Notation, asymptotic tight bound

Given a function $g(n)$, $\Theta(g(n))$ is the set of functions
$\Theta(g(n)) = \{f(n) | \exists$ positive constants c_1, c_2, and n_0 s.t.
$0 \leq c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n), \forall n > n_0\}$

Asymptotic Notation

- Insertion sort,
- in the worst case, $\Theta(n^2)$, best case is linear time, i.e., $\Theta(n^2)$ is not appropriate to describe the time complexity of the insertion sort.

Asymptotic Notation

- Selection sort
  - Best case, find the minimum at the first time, and compare with all the others,
  - Worst case, find the largest at the first time, the you need to compare with all the others, and each time, you find a smaller one. So you will need swap.
  - both cases need n^2 operations.

Asymptotic Notation
O-Notation, asymptotic upper bound

Given a function $g(n)$, $O(g(n)$ is the set of functions that
$O(g(n))=\{f(n)|\exists$ positive constants c and n_0, s.t.
$0 \leq f(n) \leq c \cdot g(n), \forall n \geq n_0\}$

Asymptotic Notation
Some Examples

- Insertion Sort: $O(n^2)$ best describes time required for insertion sort
- Binary Search: Given a sorted sequence stored in an array `A`. Given x and ask if x is in the set.

Binary Search

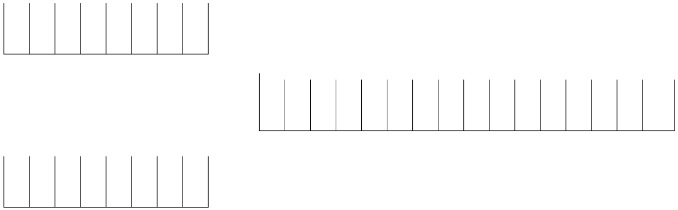
- Suppose x could fall in between i and j in array `A`
- compare x against `A[`$\frac{i+j}{2}$`]`
- if $x = \texttt{A}[\frac{i+j}{2}]$, done.
- if $x < \texttt{A}[\frac{i+j}{2}]$,if x presents, x can be in between i to $\frac{i+j}{2} - 1$.
- if $x > \texttt{A}[\frac{i+j}{2}]$,if x presents, x can be in between $\frac{i+j}{2} + 1$ to j.

Binary Search

- How fast we can find x, or we can make sure x is not present?
- Best case?
- Worst case?
- Which one best describe the time required, $\Theta(\log n)$ or $O(\log n)$?

Merge Sort

- Need another array.
- Consider the problem to merge two sorted sequence of length n.
- draw a figure



Merge Sort

- How many data elements moved? How many data elements compared?
- What is the total time for merging? $O(n)$ or $\Theta(n)$
- Question: How to get the 2 sorted lists?
- Given 2 sorted sequences of length $\frac{n}{4}$, merge them to get the sorted list of length $\frac{n}{2}$. Then how to get the sorted sequences of length $\frac{n}{4}$? ...
- draw the tree like figure

Merge Sort

- Each row "n moves" $+$ "$< n$ comparisons".
- How many rows?
- Which is the best to describe the time complexity, $O(n \log n)$ or $\Theta(n \log n)$?

Two Issues Need to Discuss

- Divide and Conquer: A technique to solve problem, very good especially when proving the correctness of the algorithm.
- Recursion: Fibonacci Series, define a function by itself.
- MergeSort, Solve a problem by solving same but smaller problem.

$$F(i) = F(i-1) + F(i-2), i > 2$$

$$F(0) = 0, F(1) = 1$$

$$\text{boundary condition}$$

Divide and Conquer- Recursion

- Solve a problem by solving the same problems (obtained by dividing the original problem) with smaller problem size, then merge the solutions to get the solution to the original problem.
- Time required for merge sort n numbers $=$ Solve two sub problems of size $\frac{n}{2}$, then merge the two in cn time.

- Let the time for merge sort n numbers be $T(n)$, then merge sort $\frac{n}{2}$ numbers takes $T(\frac{n}{2})$

Total time, $T(n)$, can be written as:

$$T(n) = 2 \cdot T(\frac{n}{2}) + cn$$

What is $T(n)$?

To Solve the Recursion

- substitution method
- changing variable
- Recursion tree
- iteration method, to expand the recurrence

$$\text{Solve } T(n) = 2T(\tfrac{n}{2}) + n^2$$

$$n^2 ---> n^2$$

$$(\tfrac{n}{2})^2 \; (\tfrac{n}{2})^2 ---> \tfrac{1}{2}n^2$$

$$(\tfrac{n}{4})^2 \; (\tfrac{n}{4})^2 \; (\tfrac{n}{4})^2 \; (\tfrac{n}{4})^2 ---> \tfrac{1}{4}n^2$$

$$\ldots\ldots\ldots\ldots$$

$$\sum \tfrac{1}{2^i}n^2 = n^2 \sum \tfrac{1}{2^i}$$

$\sum \frac{1}{2^i}$ converge to a constant.

Solve $T(n) = 3T(\lfloor \frac{n}{4} \rfloor) + n$

$$\begin{aligned}
T(n) &= n + 3T(\lfloor \frac{n}{4} \rfloor) \\
&= n + 3(\lfloor \frac{n}{4} \rfloor + 3T(\lfloor \frac{n}{16} \rfloor))) \\
&= n + 3(\lfloor \frac{n}{4} \rfloor + 3(\lfloor \frac{n}{16} \rfloor + 3T(\lfloor \frac{n}{64} \rfloor))) \\
&= n + 3\lfloor \frac{n}{4} \rfloor + 9\lfloor \frac{n}{16} \rfloor + 27T(\lfloor \frac{n}{64} \rfloor)
\end{aligned}$$

$$
\begin{aligned}
T(n) &\le (\frac{3}{4})^0 n + (\frac{3}{4})^1 n + (\frac{3}{4})^2 n + (\frac{3}{4})^3 n + \ldots + (\frac{3}{4})^{\log_4 n} n \\
& (\frac{3}{4})^{\log_4 n} n = n^{\log_4 \frac{3}{4}} n = n^{\log_4 3-1} n = n^{\log_4 3-1+1} \\
&\le n \sum_{i=0}^{\infty} (\frac{3}{4})^i + \Theta(n^{\log_4 3}) \\
& (3^{\log_4 n} = n^{\log_4 3}) \\
&= 4n + o(n) = O(n)
\end{aligned}
$$

o-notation

$o(g(n)) = \{f(n)|$ for any positive constant $c > 0$, $\exists$ a constant $n_0 > 0$ s.t. $0 \leq f(n) < cg(n)\ \forall n \geq n_0\}$

What if $T(n) = T(n/3) + T(2n/3) + n$ (balance partition) or

$$T(n) = 4T(\lfloor \frac{n}{3} \rfloor) + n?$$

A simplified Master Theorem

a, b, and c are non-negative constant that $T(1) = b$, and $T(n) = aT(\frac{n}{c}) + bn$, $n > 1$.
What if $a = c$, $a > c$, or $a < c$?

Proof of the above theorem
If n is a power of c, then

$$T(n) = bn \sum_{i=0}^{\log_c n} r^i, \quad \text{where} \quad r = a/c.$$

If $a < c$, $\sum_{i=0}^{\infty} r^i$ converges, $T(n)$ is $O(n)$.
If $a = c$, each term in the sum is unity, there are $O(\log n)$ term.
Thus $T(n)$ is $O(n \log n)$.
If $a > c$, then

$$bn \sum_{i=0}^{\log_c n} r^i = bn \frac{r^{1+\log_c n} - 1}{r - 1},$$

which is $O(a^{\log_c n}) = O(n^{\log_c a})$.

Changing Variable

- Solve $T(n) = 2T(\lfloor\sqrt{n}\rfloor) + \lg n$
- Let $m = \lg n$.
- Then we have $T(2^m) = 2T(2^{\frac{m}{2}}) + m$
- $S(m) = 2S(\frac{m}{2}) + m$
- $S(m) = m \lg m$ or
  $T(n) = T(2^m) = S(m) = m \lg m = \lg n \lg \lg n$.

Divide and Conquer

- A technique to solve problem
- Solve a problem by dividing the problem into (two) smaller size subproblems,
- solve the small subproblems,
- combine the solutions to the subproblems to get the solutions to the original problem.

Divide and Conquer- merge sort

MergeSort n numbers consists of the following 3 steps.

- DIVIDE: divide the problem into 2 sub-problems of the same size.
- CONQUER: merge sort the two subproblems
- COMBINE: merge the two sorted sequences

Divide and Conquer- quick sort

- DIVIDE: array `A[p..r]` is partitioned into nonempty `A[p..q]` and `A[q+1..r]` s.t. `A[p..q]` is less than or equal to each element of `A[q+1..r]`.
- CONQUER: quick sort the two arrays.
- COMBINE: Since the subarrays are sorted in place, no further work is needed to combine them.

```
(A,p,r)
if (p<r) then {
  q = Partition (A,p,r);
  QuickSort (A,p,q);
  QuickSort (A,q+1,r)
}
```

`A[q]` is **pivot**, after Partition, q is in the final position (rank of the pivot). Pivot is generally the first one in the array

draw a picture

Run time for Partition`(A,p,r)` is $\Theta(n)$, $n = r - p + 1$.

```
PARTITION (A,p,r)
x = A[r]
i = p-1
for i=p to r-1
  if A[j] ≤ x
    i=i+1
    exchange A[i] to A[i]
exchange A[i+1] with A[r]
return i+1
```

Quick Sort- Worst Case

$$
\begin{aligned}
T(n) &= T(n-1) + \Theta(n) \\
&= \sum_{k=1}^{n} \Theta(k) \\
&= \Theta(\sum_{k=1}^{n} k) \\
&= \Theta(n^2)
\end{aligned}
$$

draw a tree like structure

Quick Sort-Best case

`Partition` produces two subarrays of same length $\frac{n}{2}$

$$
\begin{aligned}
T(n) &= 2T(\frac{n}{2}) + \Theta(n) \\
&= \Theta(n \log n)
\end{aligned}
$$

Quick Sort- Balance Partition

- Suppose partition always produces 9-1 split
- $T(n) = T(\frac{9}{10}) + T(\frac{1}{10}) + n$

Quick Sort- Average case

- average case, expected computing time.
- need an assumption, all permutations of input numbers are equally likely
- or we say ranks of the pivot have equal probability
- Most of the time (80%) more balance than 9-1, 20% less balance than 9-1. An intuition that the average case will be $O(n \log n)$.

Quick Sort- Average Case Analysis

$$
\begin{aligned}
T(n) &= \frac{1}{n}\left(T(1) + T(n-1) + \sum_{q=1}^{n-1}(T(q) + T(n-q))\right) + \Theta(n) \\
&= \frac{1}{n}\sum_{q=1}^{n-1}(T(q) + T(n-q)) + \Theta(n) \\
&= \frac{2}{n}\sum_{k=1}^{n-1} T(k) + \Theta(n)
\end{aligned}
$$

The following, assume $T(n) \leq an \lg n + b$ for some $a > 0$, $b > 0$, to be determined.
By substitution we can show
$T(n) \leq \frac{2a}{n} \sum_{k=1}^{n-1} k \lg k + \frac{2b}{n}(n-1) + \Theta(n)$
Then to show $\sum_{k=1}^{n-1} k \lg k \leq \frac{1}{2} n^2 \lg n - \frac{1}{8} n^2$.
Finally, using the bound to show $T(n) \leq an \lg n + b$

Randomized Quick Sort

- What is the bad input?
- A randomized quick sort, need a random number generator, randomly choose a pivot. (Choose a pivot, swap with the first one).
- Question: What is the wost case time complexity for randomized quick sort?
- Question: Is there bad input?

Some remarks regarding the Quick Sort

- It is fast, performance is the best, since it moves only when necessary.
- Mix quick sort and insertion sort to obtain a faster algorithm: to sort array `A[p..q]`, if the length of `A[p..q]` is less than a given constant, then stop. Otherwise partition `A[p..q]`. Finally, insertion sort the whole array `A`.

Heap Sort

- In selection sort, can we make the selection of minimum (maximum) faster?
- Need a *priority queue*
- Priority queue is implemented using the data structure **heap**.

Priority Queue

- abstract data type, define the data type as well as its operations, detail implementations are ignored.
- a priority queue, a data structure stores *key*, the operations are "insert arbitrary" and "delete the minimum (maximum)" lead.
- Priority queue can be implemented by using unsorted array, sorted array, or a heap.

Priority Queue

- What if implemented using a unsorted array?
- What if implemented using a sorted array?
- Implemented using a heap, delete minimum and inserted arbitrary are done in $O(\log n)$ time.

Heap Structure

- A heap is a *complete binary tree*
- The tree is stored in an array.
- A max-heap, for any subtree, the root of the subtree is the maximum.
- An example, 16, 14, 10, 8, 7, 9, 3, 2, 4, 1. $_{\text{draw the max}-\text{heap}}$

Heap Structure

- In a tree, children of a node or parent of a node should be accessed in constant time.
- A node i (the index of the array), it root is $\lfloor \frac{i}{2} \rfloor$. Its children are $2i$ and $2i+1$ if $2i \leq n$ or $2i+1 \leq n$, where n is the number of nodes in the heap.
- Height of the heap is $\Theta(\log n)$.

Maintain the heap property

- A function HEAPIFY
- Apply HEAPIFY to a tree, T, only when T meets the conditions, both left subtree and right subtree of T are maximum heap; root of T is not the maximum.
- HEAPIFY moves the root down to the place it should go, and makes T a max-heap again.
- show an example
- What is the time complexity, Θ or O.

Heap Sort

- Suppose that there is a max-heap of n numbers, we are going to sort these n numbers.
- Move the root (the maximum) to the end of the array (move to the place it should go), and move the last one, p, to the root. p may not be the maximum, we then need to modify (maintain) the heap- HEAPIFY. Note that the size of the heap is reduced by one.

- After HEPAIFY, we have a max-heap of $n-1$ nodes.
- We then go to the first step and then iterate the steps until there are no nodes.
- What is the cost? Θ or O

Build the Heap

Input of the Heap Sort is an unsorted array. The first step is to build the Max-heap.

- Given a heap of size n, how many are leaves? $\lceil \frac{n}{2} \rceil$
- These leaves are max-heap.
- for $(\lceil \frac{n}{2} \rceil - 1)$ down to 1 do HEAPIFY. a quick example
- Time complexity: each Hepaify takes $O(\log n)$ time, there are $\frac{n}{2}$ HEAPIFIES, so the total cost is $O(n \log n)$??
- accurate analysis leads to linear time upper bound.

Build a Heap

Time required to by HEAPIFY a sub-tree of height h is $O(h)$

leaves	$\lceil \frac{n}{2} \rceil$
height 1	$\lceil \frac{n}{2^2} \rceil$
height 2	$\lceil \frac{n}{2^3} \rceil$
....	
height h	$\lceil \frac{n}{2^{h+1}} \rceil$

So the total cost is

$$\begin{aligned}
\sum_{h=0}^{\lfloor \lg n \rfloor} ch \lceil \frac{n}{2^{h+1}} \rceil &\leq cn\Big(\sum_{h=0}^{\lfloor \lg n \rfloor} \frac{h}{2^h}\Big) \\
&\leq cn \sum_{h=0}^{\infty} \Big(\frac{h}{2^h}\Big) \\
&\leq cn \cdot 2 \\
&= O(n)
\end{aligned}$$

Building the Heap

To get $\sum_{h=0}^{\infty}(\frac{h}{2^h})$,

Eq. ((A.8) or (3.6)) $\sum_{k=0}^{\infty} kx^k = \frac{x}{(1-x)^2}$

Substituting $x = \frac{1}{2}$ yields $\sum_{k=0}^{\infty} k\frac{1}{2^k}$

- So building a max-heap actually takes $O(n)$ time.
- Question: Do you think accurate analysis of heap sort leads better time complexity ($o(n \log n)$ time)??

Can you beat $n \log n$ bound for Sorting Problem?

- Linear decision tree model: a binary tree, internal nodes are comparisons, leaves are solutions
- computation: proceed by comparing in the internal nodes, outcome decides the branching directions, when the computation reach a leave, a solution is obtained
- The number of operations = the path from root to the leaf (result).

- the number of leaves $=$ the all possible results. For sorting problem (to decide a permutation that meets a certain property), if the input size is n, there are $n!$ possible results.
- the least possible path length (in the worst case) is at least $\log n!$ which is greater than $n \log n$, ($n! \geq (\frac{n}{e})^n$), Stirling's approximation)
- $n \log n$ is the best possible result.

Ω-notation- Asymptotic Lower Bound

For a given $g(n)$, $\Omega(g(n))$ is the set of functions
$\Omega(g(n)) = \{\ f(n) | \exists$ positive constants c and n_0, s.t.
$0 \leq cg(n) \leq f(n), \forall n \geq n_0\ \}$
For any two functions $f(n)$ and $g(n)$, $f(n) = \Theta(g(n))$ if and only if
$f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$

The Master Theorem

Let $a \geq 1$ and $b \geq 1$ be constants, let $f(n)$ be a function, and let $T(n)$ be defined on the nonnegative integers by the recurence

$$T(n) = aT(n/b) + f(n).$$

$T(n)$ can be bounded asymptotically as

1. If $f(n) = O(n^{\log_b a - \epsilon})$ for some constant $\epsilon > 0$, then $T(n) = \Theta(n^{\log_b a})$.
2. If $f(n) = \Theta(n^{\log_b a})$, then $T(n) = \Theta(n^{\log_b a} \lg n)$.
3. If $f(n) = \Omega(n^{\log_b a + \epsilon})$ for some constant $\epsilon > 0$, and if $af(n/b) \leq cf(n)$ for some constant $c < 1$ and all sufficiently large n, then $T(n) = \Theta(f(n))$.

Sorting in Linear Time

- Assume that each of the n input elements is an integer in the range 1 to k.
- When $k = O(n)$, the sort runs in $O(n)$ time.
- basic idea, for each input element x, how many are less than x- counting sort.

- input sequence is $(3, 6, 4, 1, 3, 4, 1, 4)$, we consider the sequence as $(3^1, 6^2, 4^3, 1^4, 3^5, 4^6, 1^7, 4^8)$ where the superscript is the position of the key in the input sequence.
- *stable*: a sorting algorithm is stable if the order of two elements having identical keys is preserved after applying the algorithm.
- i.e., we get the result, $(1^4, 1^7, 3^1, 3^5, 4^3, 4^6, 4^8, 6^2)$.

Counting Sort

First Iteration, calculate the number of occurences of a key.

Number of Occurences	0	2	0	2	3	0	1	0	0	0
Key	0	1	2	3	4	5	6	7	8	9

Second Iteration, Calculate the "positions" of the keys after sorting.

Position	0	2	2	4	7	7	8	8	8	8
Key	0	1	2	3	4	5	6	7	8	9

Combine these two tables, we know, for examples,
there are 2 1s (from first table), the last 1 will be at position 2 after sorted (from second table), or
there are three 4s (from first table), the last 4 will be at position 7 after sorted (from second table).

Radix Sort

- A d-digit number is considered d keys.
- most significant digit first
- least significant digit first
- an example 329, 457, 657, 839, 436, 720, 355

Discussion on the lower bound

- Why radix sort beats the lower bound?
- the operation used is not “comparison”, thus it is beyond the the linear decision tree model.
- powerful operator could reduce the time required

Minimum Gap and Maximum Gap

- Minimum Gap: Given a set $A = \{a_1, a_2, ..., a_n\}$, determine i and j that $|a_i - a_j|$ is minimum.
- Maximum Gap: Given a set $A = \{a_1, a_2, ..., a_n\}$, determine i and j that there is no a_k, $a_i < a_k < a_j$ and $|a_i - a_j|$ is maximized.
- Both have $\Omega(n \log n)$ lower bound under linear decision tree model.
- Maximum can be solved in $O(n)$ time when floor function is allowed.

Linear time algorithm for Max-Gap

- Given $A = \{a_1, a_2, ..., a_n\}$, find the maximum gap.
- normalize the numbers into the range $[0, 1]$.
- Equally divide the range into $n + 1$ intervals.
- put these n number into the buckets

- by pigeon hole principle, there must be at least an empty bucket
- The maximum gap cannot be determined by the element in the same bucket
- maximum gap can be determined by the largest in a bucket and the smallest in the next non-empty bucket.
- a linear time algorithm