# Searching and Sorting

Jeremy Chow
10-2-2018

# Table Of Content - Searching

- Linear Search
- Binary Search
- Binary Search on Answer
- Ternary Search

# Table Of Content - Sorting

- Comparison Based Sorting
  - Bubble Sort
  - Insertion Sort
  - Selection Sort
  - Merge Sort
  - Quick Sort

- Non-comparison Based Sorting
  - Counting Sort
  - Radix Sort

# Searching - Introduction

- Usage
  - Locating an object in an array
  - Finding an optimal number for a problem
- Often require preprocessing

# Linear Search (線性搜尋法)

- aka Sequential Search
- Most basic and freuently used seaching algorithm
- Start checking from the begining to the end
- (Situational) Optimization - Stop until a match is found

# Linear Search

❖ Example - Searching an element X in an array A with distinct elements

```
for(int i = 0; i < n; i++)
    if(a[i] == x){
        pos = i;
        break;
    }
```

# Linear Search

| Situation | Time Complexity |
| --- | --- |
| Best | *O(1) / O(N)* |
| Worst | *O(N)* |
| Average | *O(N)* |

# Linear Search

- If we need to perform searching for Q times on an array with size N
- Overall time complexity = O(NQ)
- When N and Q is large (e.g. N, Q <= $10^5$)
- Program will not be able to execute in 1s

- We need some searching algorithm faster than linear search!

# Binary Search (二分搜尋法)

- Recall the “Guess Number” (估數字) game
- We do not need to guess 100 times in order to guess the target number
- Instead the optimal way is to guess the middle number within the range
- e.g target = 11

- 1 - 49 → 1 - 24 → 1 - 11 → 7 - 11
- 10 - 11 → 11 - 11 → 11
- Required almost 7 guess!

# Binary Search

- Requirment : Sorted
- For each search, eliminate impossible region
  - $A_0 \le A_1 \le \ldots \le A_{n-1}$
  - If key < $A_k$ , then key < $A_i$ for $i \ge k$
  - If key > $A_k$ , then key > $A_i$ for $i \le k$

| | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 3 | 3 | 4 | 6 | 7 | 7 | 8 | 9 | 9 | 10 | 11 | 12 | 15 | 18 |

- If key = 6 and k = 7 ($A_k$= 8)
- There is no point on searching $A_{8..14}$

# Binary Search

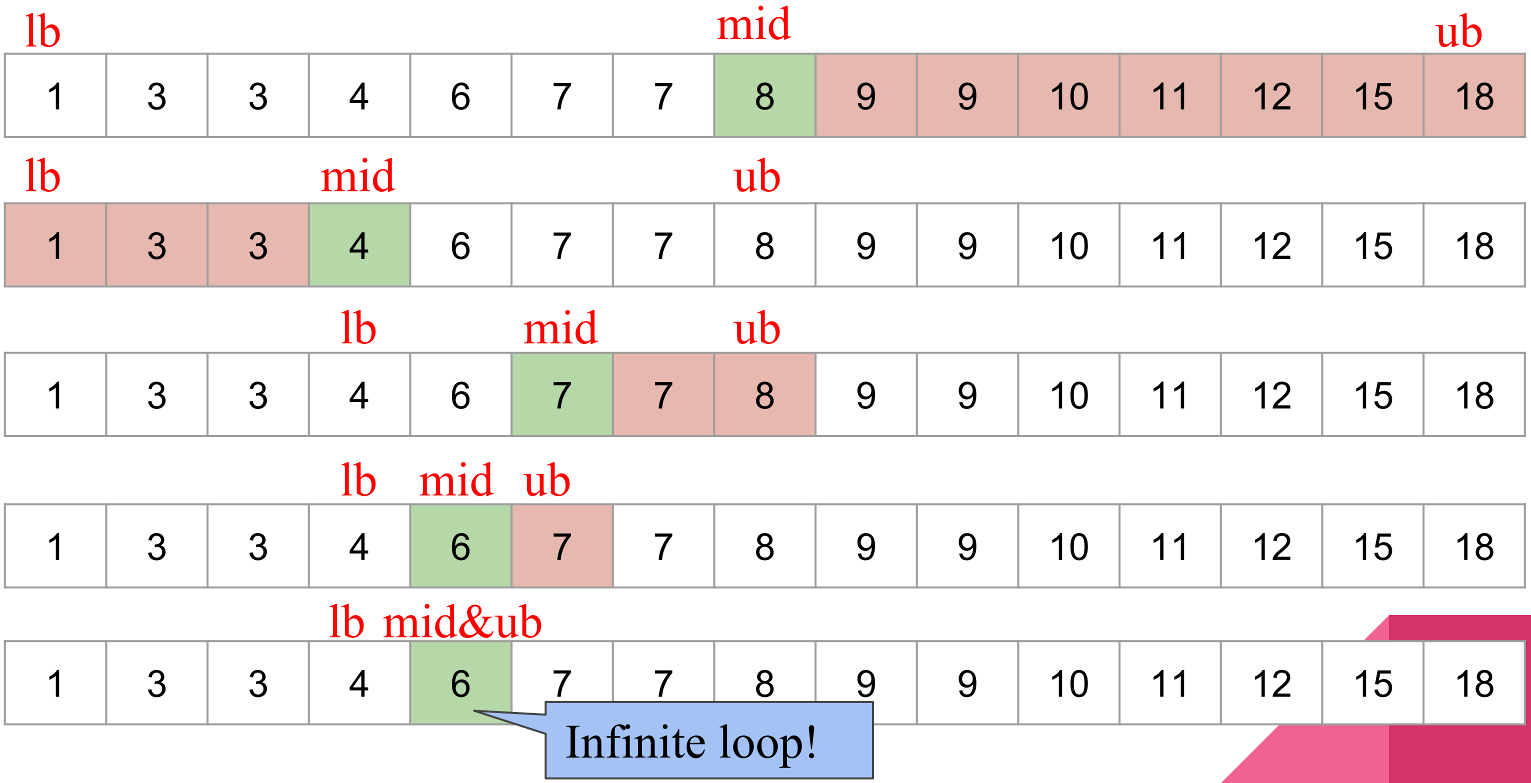
- Set the searching range as the entire array
- Repeat the following process until the key is found
  - Calculate the midpoint
  - Compare key with A[midpoint]
  - If key < A[midpoint], then continue searching on the first half of the array
    - A[midpoint] to A[upper_bound] does not contain the key
  - If key > A[midpoint], then continue searching on the second half of the array
    - A[lower_bound] to A[midpoint] does not contain the key

# Binary Search

- Seems like correct….
- Try to search 6 on array A

```
int lb, mid, ub;
lb = 0; ub = n - 1;

while (lb <= ub) {
    mid = (lb + ub) / 2;
    if (key <= a[mid]) ub = mid;
    else lb = mid;
}

if(key == a[ub]) printf("FOUND\n");
else printf("NOT FOUND\n");
```

# Binary Search



# Binary Search

- Correct Implementation

```
int lb, mid, ub;
lb = -1; ub = n;

while (ub - lb > 1) {
        mid = (ub + lb) / 2;
        if (key <= a[mid]) ub = mid;
        else lb = mid;
}

if (ub < n && key == a[ub]) printf("FOUND\n");
else printf("NOT FOUND\n");
```

# Binary Search

- Time Complexity : O(log N)

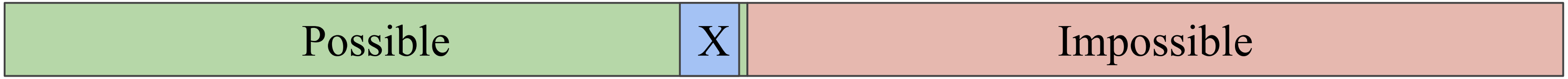
- Why “mid = (ub + lb) / 2" ?
  - The expected area eliminated is largest

# Binary Search

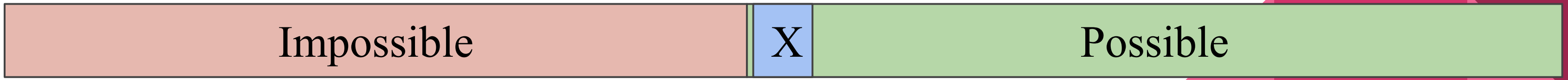
- Applications
  - Check if an element exists in an array
  - Find its position if it exists
  - If it is not exists, the position it should be inserted into
  - The smallest element >= x / > x
  - The largest element <= x / < x
- C++ useful functions
  - binary_search(begin, end, x)
    - returns true / false whether x is present
  - lower_bound(begin, end, x)
    - returns the pointer to leftmost element >= x
  - upper_bound(begin, end, x)
    - returns the pointer to leftmost element > x

# Binary Search on Answer

- Binary Search on “Answer” instead of array
- If a task asks you to find max possible value such that…
- Assume the answer is x, then you can binary search for x if:
  - There is an efficient way to check whether a value v is possible
  - It is possible for all x’ <= x; and it is impossible for all x’ > x




- Vice versa, you can binary search for minimum possible value if...



# Binary Search on Answer

```
int lb = 0;
int ub = 1000000001;

while(ub - lb > 1){
        int mid = (lb + ub) / 2;

        if(check(mid)) lb = mid;
        else ub = mid;
}
```

Careful with the Range!

Function that check whether mid is possible

- ❖ Greedy, Dynamic Programming, … etc. may be used in check(x)

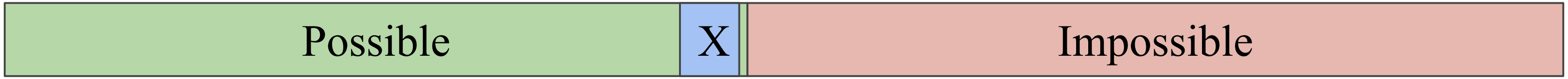
# Binary Search on Answer

- Let the function be $f(x)$
- lower_bound and upper_bound of answer be *lb* and *ub*

- Time Complexity : $O(f(x) \log(ub - lb))$

# M1023 Seating Plan

- Given an array a[1..n], choose m elements such that the minimum absolute difference is maximized
- Observation 1
  - For x >= 0, if we can choose m elements such that the minimum absolute difference is >= x
  - We can always choose m elements such that the minimum absolute difference is >= x - 1
  - We are going to find the maximium x !

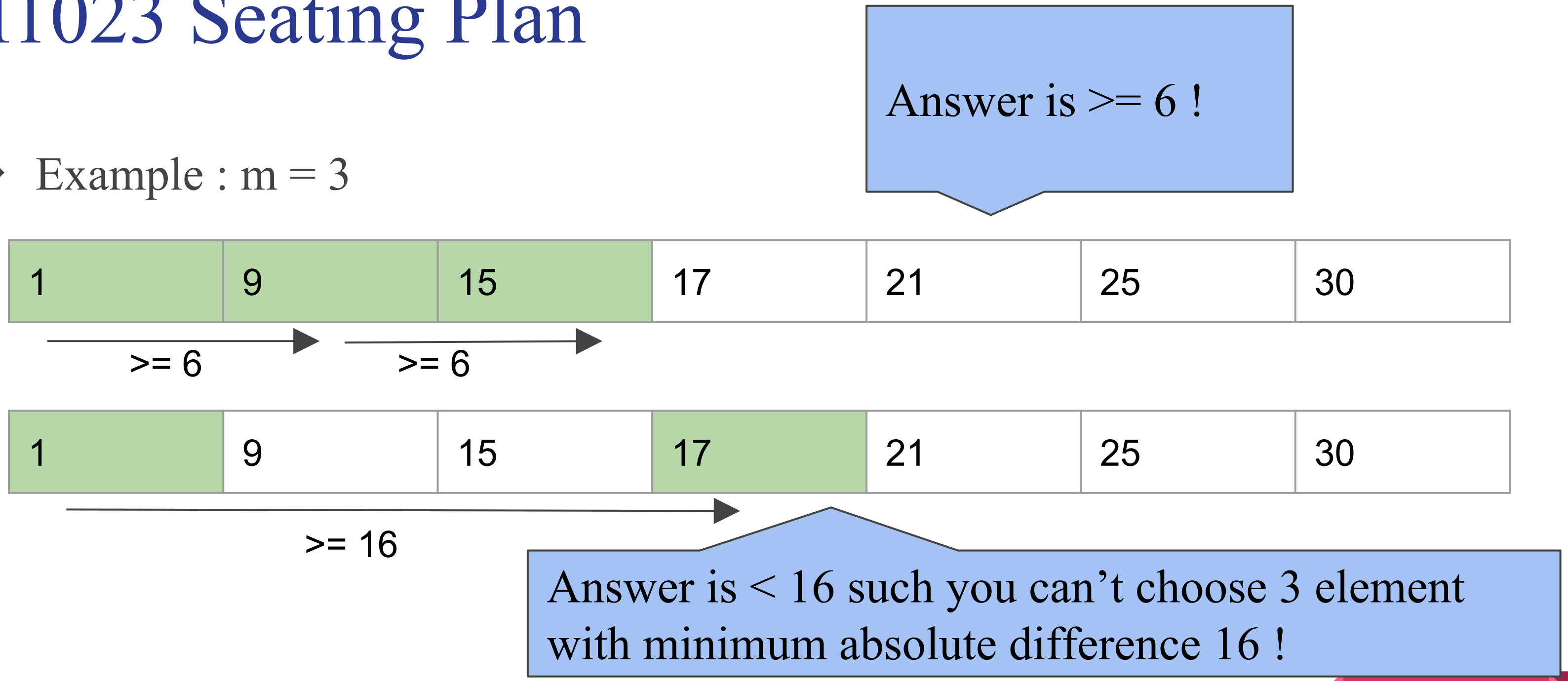


# M1023 Seating Plan

- Observation 2
  - Optimal way to choose m element with minimum absolute difference >= target :
  - Assume the array is sorted, we always choose the a[0](smallest)
  - Then we choose the first element such that a[i] - a[pre] >= target
  - Repeat previous step until m element is chosen
  - If m element can not be chosen, then there is no way to choose m element with minimum absolute difference >= target

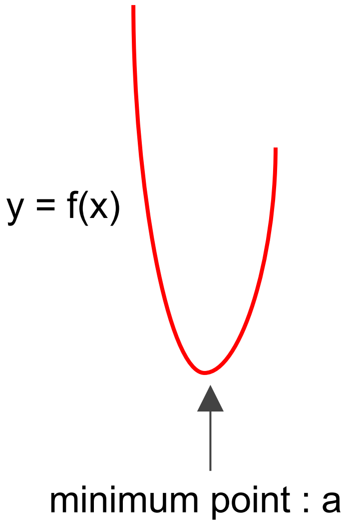
# M1023 Seating Plan

- Example : m = 3



# Ternary Search (三分搜尋法)

- Find the max/min of a single-peek function
- If we want to find minimum value of a function
- Requirments :
  - f(x-1) < f(x) for all lb <= x < a (to the left of a)
  - f(x) < f(x+1) for all a < x <= ub (to the right of a)
  - lb and ub are the lower_bound and upper_bound of the answer
  - Note the stricty < condition
- Exception: f(x) = f(x + 1) = minimum is acceptable
- Useful in many optimization problems



# Ternary Search

- Repeat the following process until the precision is high enough
  - Let m1 be the one-third point, m2 be the two-third point of the searching range
    - m1=low+(high-low)/3, m2=high-(high-low)/3
  - If f(m1) < f(m2), then continue searching from m1 to high
    - The peak value does not lie between low to m1
  - Else continue searching from low to m2
    - The peak value does not lie between m2 to high
- Time Complexity : *O(logN)*

# Ternary Search

```
double lo = lb, hi = ub, m1, m2;

while(hi - lo > EPS){
        m1 = (lo * 2 + hi) / 3;
        m2 = (lo + hi * 2) / 3;
        if(f(m1) > f(m2)) lo = m1;
        else hi = m2;
}

printf("MINIMUM %.12lf\n", lo);
```

# Conclusion - Searching

- Use different searching algorithm in different situations

- Number of element is small / time limit is not strict → linear search
- Searching on sorted array → binary search
- Searching extreme value on a single-peek function → ternary search

# Sorting - Introduction

- Reordering array elements into specific order (usually ascending)
- Sometime sorting is used for algorithm preprecessing
  - Searching
  - Greedy
  - Dynamic programing

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 12 | 8 | 4 | 5 | 4 | 7 | 4 | 20 | 3 |

↓

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 3 | 4 | 4 | 4 | 5 | 7 | 8 | 12 | 20 |

# Sorting - Introduction

- Many ways to sort an array
- Some are faster, and some are slower
- Some are easier to code, some are harder to code

| Comparison Based Sorting (Slow) | Comparison Based Sorting (Faster) | Non-comparison Based Sorting |
|---|---|---|
| • Bubble Sort<br>• Insertion Sort<br>• Selection Sort | • Quick Sort<br>• Merge Sort | • Counting Sort<br>• Radix Sort |

# Bubble Sort (冒泡排序法)

- Starting from the start of the array, compare two adjacent elements
  - if (a[i] > a[i + 1]) swap(a[i], a[i + 1]);
- After we process the array for one round, the greatest element will be in the correct place (right-most)
- In $k^{th}$ iteration, the $k^{th}$ largest element will be bubbled to the correct place
- Other elements may still be out of order
- Repeat this process for n - 1 times

# Bubble Sort

- Bubble Sort Dry run
- Sorting an array with size 5

```
for(int i = 0; i < n - 1; i++)
    for(int j = 0; j < n - i - 1; j++)
        if (a[j] > a[j + 1]) {
            int tmp = a[j];
            a[j] = a[j + 1];
            a[j + 1] = tmp;
        }
```

| Round | A[0] | A[1] | A[2] | A[3] | A[4] |
|---|---|---|---|---|---|
| i=0,j=0 | 2 | 4 | 5 | 1 | 3 |
| i=0,j=1 | 2 | 4 | 5 | 1 | 3 |
| i=0,j=2 | 2 | 4 | 5 | 1 | 3 |
| i=0,j=3 | 2 | 4 | 1 | 5 | 3 |
| i=1,j=0 | 2 | 4 | 1 | 3 | 5 |
| i=1,j=1 | 2 | 4 | 1 | 3 | 5 |
| i=1,j=2 | 2 | 1 | 4 | 3 | 5 |
| i=2,j=0 | 2 | 1 | 3 | 4 | 5 |
| i=2,j=1 | 1 | 2 | 3 | 4 | 5 |
| i=3,j=0 | 1 | 2 | 3 | 4 | 5 |
| Result | 1 | 2 | 3 | 4 | 5 |

# Bubble Sort

```
for(int i = 0; i < n - 1; i++)
    for(int j = 0; j < n - i - 1; j++)
        if (a[j] > a[j + 1]) {
            int tmp = a[j];
            a[j] = a[j + 1];
            a[j + 1] = tmp;
        }
```

# Bubble Sort

❖ Optimized version

```
for (int i = 0; i < n - 1; i++){
        bool flag = 0;
        for(int j = 0; j < n - i - 1; j++)
                if(a[j] > a[j + 1]){
                        int tmp = a[j];
                        a[j] = a[j + 1];
                        a[j + 1] = tmp;
                        swap = 1;
                        //c++ users : swap(a[j], a[j + 1])
                }
        }
        if(!flag) break;
}
```

# Bubble Sort

- Advantage
  - Easy to code, understand and memorize
  - Require little additional space
  - O(N) when array is almost sorted (Optimized)
- Disadvantage
  - Best (without optimization) = Worst = Average time complexity = $O(N^2)$
  - Worst case : Reverse order, the total number of comparisons
    - $(n-1) + (n-2) + \ldots + 2 + 1 = n * (n - 1) / 2$

# Bubble Sort

- Want to know no. of adjacent swapping to sort an array without using bubble sort?
- Inversion
  - number of pair (i, j) where i < j but a[i] > a[j]
  - Sorted = 0 inversion
  - Reversed order = n * (n - 1) / 2 inversions
- Inversion can be find in O(NlogN) in many ways

# Insertion Sort (插入排序法)

- Method we often used in sorting playing cards
- We have some sorted playing cards in our hand
- Now we receieved a new playing card
- Insert it into the right position

# Insertion Sort

- Iterate for N-1 times, from 1 to N-1 (Zero Based)
- In $i^{th}$ iteration, we have i sorted playing cards in our hand and now we receive a new playing cards A[i] (
- We find the right position of A[i] and insert it into there

# Insertion Sort

- Insertion Sort Dry run
- Sorting an array with size 5

```
for(int i = 1; i < n; i++){
        for(int j = i; j >= 1; j--)
                if(a[j - 1] > a[j])
                        swap(a[j - 1], a[j]);
                else break;
}
```

| Round | A[0] | A[1] | A[2] | A[3] | A[4] |
|---|---|---|---|---|---|
| i=1,j=1 | 2 | 4 | 5 | 1 | 3 |
| i=2,j=2 | 2 | 4 | 5 | 1 | 3 |
| i=3,j=3 | 2 | 4 | 5 | 1 | 3 |
| i=3,j=2 | 2 | 4 | 1 | 5 | 3 |
| i=3,j=1 | 2 | 1 | 4 | 5 | 3 |
| i=4,j=4 | 1 | 2 | 4 | 5 | 3 |
| i=4,j=3 | 1 | 2 | 4 | 3 | 5 |
| i=4,j=2 | 1 | 2 | 3 | 4 | 5 |
| Result | 1 | 2 | 3 | 4 | 5 |

# Insertion Sort

```
for(int i = 1; i < n; i++){
        for(int j = i; j >= 1; j--)
                if(a[j - 1] > a[j])
                        swap(a[j - 1], a[j]);
                else break;
}
```

❖ Number of swap = inversions too!

# Insertion Sort

- Advantage
  - Similar to those mentioned in Bubble Sort

- Disadvantage
  - Time Complexity is still $O(N^2)$

# Selection Sort (選擇排序法)

- Repeatedly moving the maximum / minimum in the unsorted part to the front of the unsorted part
- Like what we usually did in the beginning of 鋤大D

# Selection Sort

- Iterate for N-1 times
- In $i^{th}$ iteration, find the maximum element in a[0..n-i-1]
- Swap it with a[n-i-1]

# Selection Sort

- Selection Sort Dry run
- Sorting an array with size 5

```
for (int i = 0; i < n - 1; i++){
        int ind = 0;
        for (int j = 1; j < n - i; j++){
                if (a[j] > a[ind])
                ind = j;
        }
        swap(a[ind], a[n - i - 1]);
}
```

| Round | A[0] | A[1] | A[2] | A[3] | A[4] |
|---|---|---|---|---|---|
| Original | 2 | 4 | 5 | 1 | 3 |
| i=0 | 2 | 4 | 3 | 1 | 5 |
| i=1 | 2 | 1 | 3 | 4 | 5 |
| i=2 | 2 | 1 | 3 | 4 | 5 |
| i=3 | 1 | 2 | 3 | 4 | 5 |
| Result | 1 | 2 | 3 | 4 | 5 |

# Selection Sort

- Advantage
  - Easy to code, understand and memorize
  - Require little additional space
- Disadvantage
  - Time complexity is still $O(N^2)$

- Max no. of swap = N-1 instead of N * (N-1) / 2

# Merge Sort (合併排序法)

- If we have 2 sorted array, we can merge them into 1 sorted array in O(N)
- Make sure of this idea
- Divide-and-conquer
  - Split large problem into smaller problems

# Merge Sort

- Split an array a[lo..hi] into two halves and recursively sort them
  - a[lo..mid] and a[mid + 1..hi]
  - Split "sorting N elements" into two "sorting N / 2 elements"
- Merge a[lo..mid] and a[mid + 1..hi] into a[lo..hi] in O(N)
- Now we sorted a[lo..hi] !

# Merge Sort

- Split an array a[lo..hi] into two halves and recursively sort them
  - a[lo..mid] and a[mid + 1..hi]
  - Split "sorting N elements" into two "sorting N / 2 elements"

```
void merge_sort(int lo, int hi){
    if(lo == hi) return;

    int mid = (lo + hi) / 2;
    merge_sort(lo, mid);
    merge_sort(mid + 1, hi);
```

Base Case

# Merge Sort

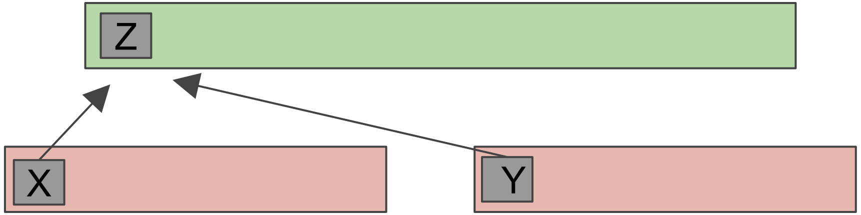
❖ Merge a[lo..mid] and a[mid + 1..hi] into a[lo..hi] in O(N)

```
int p = lo;
int p1 = mid + 1;
int ind = lo;

while(p <= mid && p1 <= hi){
      if(a[p] <= a[p1])
      tmp[ind++] = a[p++];
      else
      tmp[ind++] = a[p1++];
}

while(p <= mid) tmp[ind++] = a[p++];
while(p1 <= hi) tmp[ind++] = a[p1++];

for(int i = lo; i <= hi; i++)
a[i] = tmp[i];
```

# Merge Sort

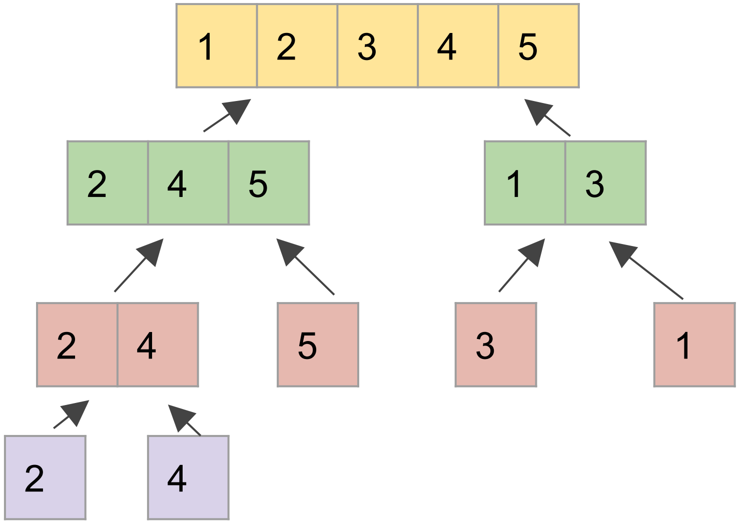
- Compare elements of a[lo..mid] and a[mid+1..hi] from the beginning
- if X <= Y then Z = X
- else Z = Y
- Insert the rest of another array

  when one of the array finish processing




```
int p = lo;
int p1 = mid + 1;
int ind = lo;

while(p <= mid && p1 <= hi){
        if(a[p] <= a[p1])
        tmp[ind++] = a[p++];
        else
        tmp[ind++] = a[p1++];
}

while(p <= mid) tmp[ind++] = a[p++];
while(p1 <= hi) tmp[ind++] = a[p1++];

for(int i = lo; i <= hi; i++)
a[i] = tmp[i];
```

# Merge Sort




```
int p = lo;
int p1 = mid + 1;
int ind = lo;

while(p <= mid && p1 <= hi){
        if(a[p] <= a[p1])
        tmp[ind++] = a[p++];
        else
        tmp[ind++] = a[p1++];
}

while(p <= mid) tmp[ind++] = a[p++];
while(p1 <= hi) tmp[ind++] = a[p1++];

for(int i = lo; i <= hi; i++)
a[i] = tmp[i];
```

# Merge Sort

❖ Complete Implementation (One Based)

```
void merge_sort(int lo, int hi){
    if(lo == hi) return;

    int mid = (lo + hi) / 2;
    merge_sort(lo, mid);
    merge_sort(mid + 1, hi);

    int p = lo;
    int p1 = mid + 1;
    int ind = lo;

    while(p <= mid && p1 <= hi){
        if(a[p] <= a[p1])
        tmp[ind++] = a[p++];
        else
        tmp[ind++] = a[p1++];
    }

    while(p <= mid) tmp[ind++] = a[p++];
    while(p1 <= hi) tmp[ind++] = a[p1++];

    for(int i = lo; i <= hi; i++)
    a[i] = tmp[i];
}
```

# Merge Sort

- Merge Sort follows divide-and-conquer approach
- Divide:
  - Divide the n-element sequence into two (n/2)-element sequences
- Conquer:
  - Sort the two subsequences recursively
- Combine:
  - Merge the two sorted subsequence to produce the answer

# Merge Sort

- Best = Worst = Average time complexity : O(NlogN)
- Way better compare to $O(N^2)$
- Can sort $10^5$ numbers within a second

- Can be used to find out the no. of inversions!

# Quick Sort (快速排序)

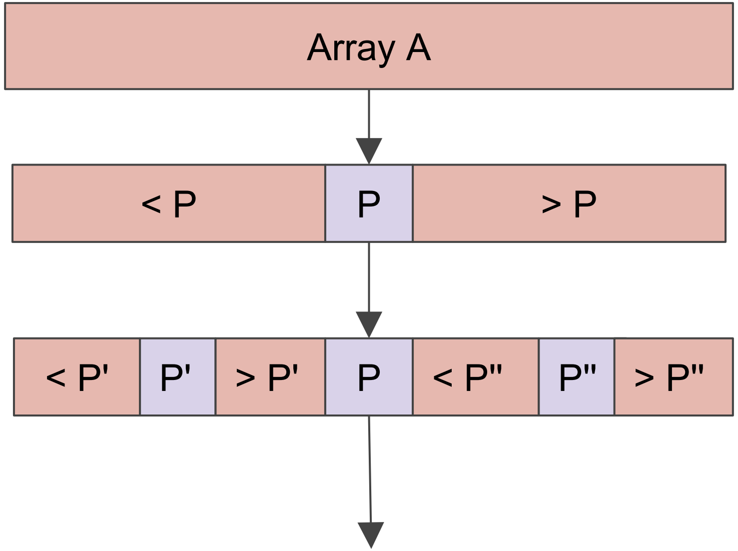
- Similar algorithm with Merge Sort
- Use Divide-and-Conquer approach again!
- Instead of split the array into two equal part
- Pick a pivot (支點) p, separate the array into two part
- One contain value < p, One contain value > p

# Quick Sort

- In each function call

  we need to sort the red part

- We split them into two red part
- After some number of steps
- The array will become sorted



# Quick Sort

- We choose the middle element as pivot
- You can choose random element as pivot too
- Performance depends on choice of pivot

```
void quick_sort(int lo, int hi){
	if(lo >= hi) return;

	int mid = (lo + hi) / 2;
	int pivot = a[mid];
	int i = lo - 1, j = hi + 1;

	while (i < j) {
		do ++i; while (a[i] < pivot);
		do --j; while (a[j] > pivot);
		if(i < j) swap(a[i], a[j]);
	}

	quick_sort(lo, i - 1);
	quick_sort(j + 1, hi);
}
```

# Quick Sort

- Divide-and-conquer process for sorting an array A[lo..hi]
- Divide:
  - A[lo..hi] is partitioned into two nonempty subarrays A[lo..q] and A[q+1..hi] such that each element of A[lo..q] is less than each element of A[q+1..hi]
- Conquer:
  - The two subarrays A[lo..q] and A[q+1..hi] are sorted by recursive calls to quicksort.

# Quick Sort

- Best and Average time complexity = O(NlogN)
- However, Worst time complexity = $O(N^2)$
- We can construct a data such that it runs really really slow

- Although in most of the case, the time complexity of Quick Sort is O(NlogN)

# Comparison Based Sorting

- We have some $O(N^2)$ sorting algorithm
- We can speed it up by using some O(NlogN) sorting algorithm
- Can we improve more?

- By some mathematical proof (refers to last year slide, P.36-37)
- We can prove that the time complexity lower bound for comparison based sorting is O(NlogN)

# Non-comparison Based algorithm

- Sorting without comparison ( < or > )
- Does not always work
  - Sorting floating point number
- Use depends on situations
  - data type
  - data range

# Counting Sort (計數排序法)

- Assume 1 <= A[i] <= M and all a[i] are integers
- Count the occurrence of numbers in the array
- Add 1 to index x of the array cnt
  - If a[i] = x, then cnt[x]++;
- After processing for the n numbers
- We get the frequency array cnt
- Iterate from 1 to M, we print number i for cnt[i] times

# Counting Sort

Array A

| | | | | | |
|---|---|---|---|---|---|
| 2 | 4 | 5 | 1 | 3 | 1 |

Array cnt

| Index | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|
| cnt[i] | 2 | 1 | 1 | 1 | 1 |

Result

| | | | | | |
|---|---|---|---|---|---|
| 1 | 1 | 2 | 3 | 4 | 5 |

# Counting Sort

```c
int a[] = {2, 4, 5, 1, 3, 1};
int n = 6;
int cnt[6];

int main(){
	for(int i = 0; i < n; i++)
	cnt[a[i]]++;

	for(int i = 1; i <= 5; i++)
		for(int j = 0; j < cnt[i]; j++)
		printf(" %d ", i);

	printf("\n");
}
```

Result

```
1  1  2  3  4  5
```

# Counting Sort

- M is the range of the data
- Time complexity = O(M + N), Space complexity = O(M)
- Very fast sorting algorithm when M is small
- Works for sorting character too (M = 26 / 52)

- When M is large, e.g. $1 <= A[i] <= 10^9$
- Counting Sort would be too slow

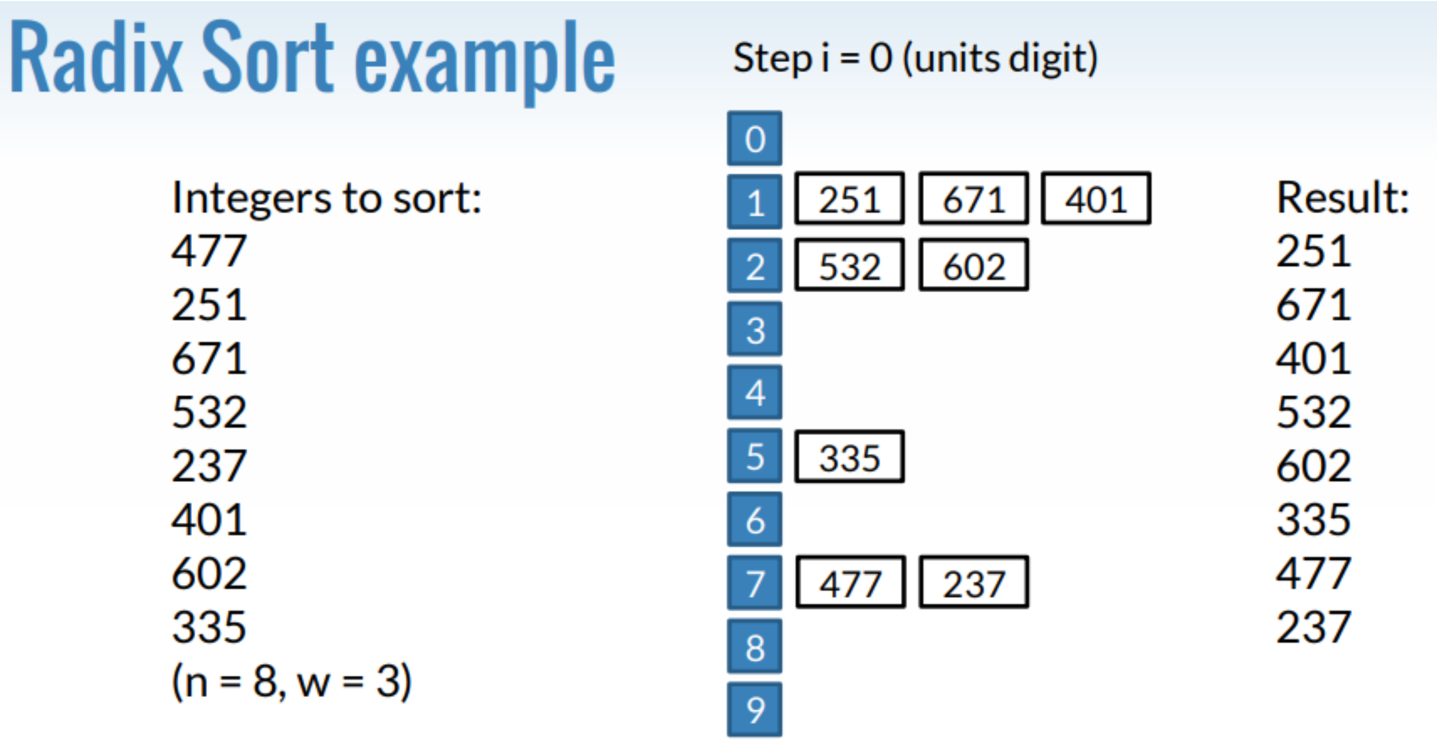
# Radix Sort (基數排序法)

- aka card sort
- Sort n integers. Each integer has w digits
- For digit i = 0 (least-significant) to w – 1 (most significant)
  - Prepare 10 lists, one for each digit 0, 1, 2, ..., 9
  - Loop through the array: If the i-th digit of a number is x, insert it into list x
  - Concatenate the 10 lists to form the new array for the next step

# Radix Sort

- Initialize an array of 10 buckets to empty
- for i = 1 to N
  - place A[i] into the bucket with its last digit
- Use the same process to sort the second last digit
- Repeat until the first digit

# Radix Sort


Radix Sort example
Step i = 0 (units digit)
Integers to sort:
477
251
671
532
237
401
602
335
(n = 8, w = 3)
0
1
251
671
401
2
532
602
3
4
5
335
6
7
477
237
8
9
Result:
251
671
401
532
602
335
477
237

# Radix Sort

## Radix Sort example

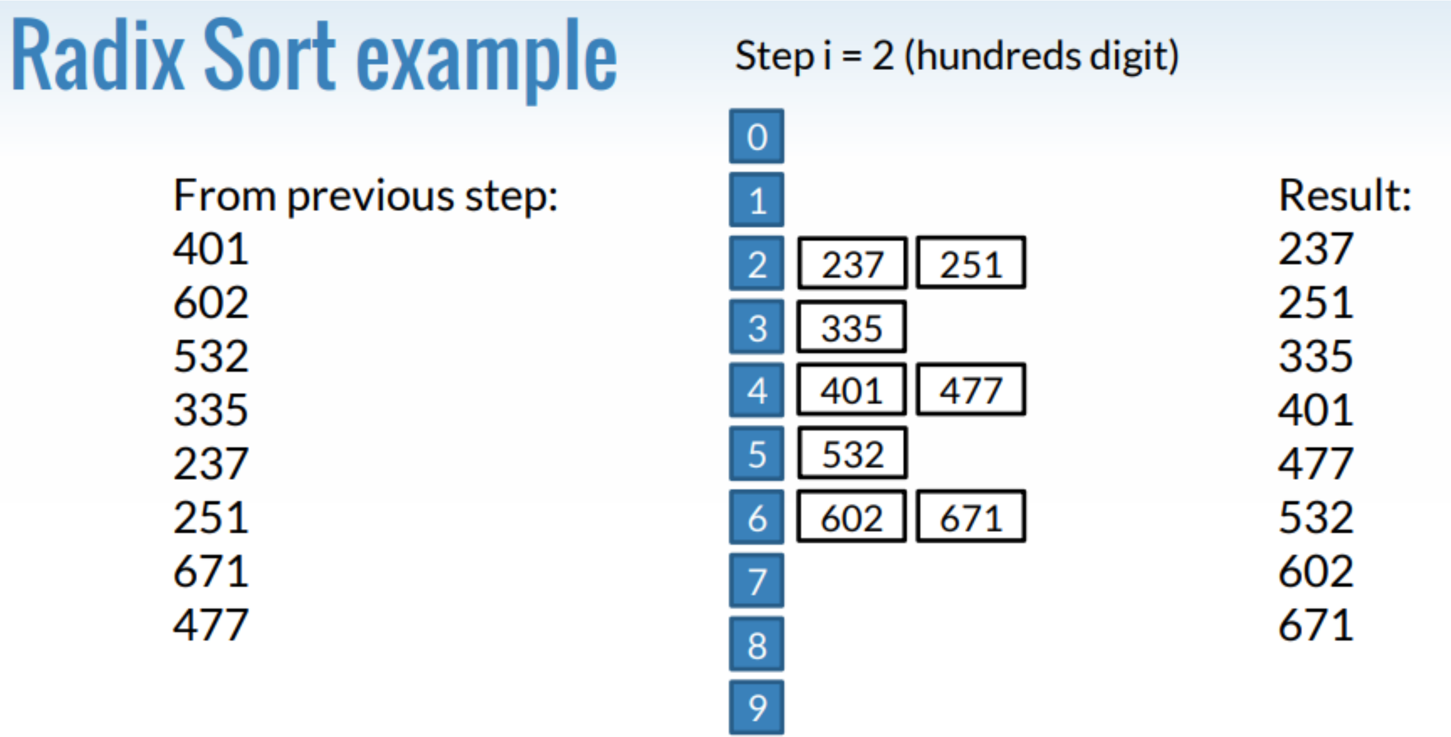
Step i = 1 (tens digit)

From previous step:
251
671
401
532
602
335
477
237

| | | | |
|---|---|---|---|
| 0 | 401 | 602 | |
| 1 | | | |
| 2 | | | |
| 3 | 532 | 335 | 237 |
| 4 | | | |
| 5 | 251 | | |
| 6 | | | |
| 7 | 671 | 477 | |
| 8 | | | |
| 9 | | | |

Result:
401
602
532
335
237
251
671
477

# Radix Sort


Radix Sort example
Step i = 2 (hundreds digit)
From previous step:
401
602
532
335
237
251
671
477
0
1
2
237
251
3
335
4
401
477
5
532
6
602
671
7
8
9
Result:
237
251
335
401
477
532
602
671

# Radix Sort

```
int a[] = {477, 251, 671, 532, 237, 401, 602, 335};
int n = 8;
vector <int> bucket[10];

int main(){
        for(int i = 0; i < 3; i++){
                for(int j = 0; j < 10; j++)
                bucket[j].clear();

                for(int j = 0; j < n; j++){
                        int digit = (a[j] % (int)pow(10, i + 1)) / (int)pow(10, i);
                        bucket[digit].push_back(a[j]);
                }

                int p = 0;

                for(int j = 0; j < 10; j++){
                        for(int k = 0; k < bucket[j].size(); k++)
                        a[p++] = bucket[j][k];
                }
        }

        for(int i = 0; i < n; i++)
        printf(" %d ", a[i]);
        printf("\n");
```

Result

```
237  251  335  401  477  532  602  671
```

# Radix Sort

- Time Complexity : $O(nw)$
- Space Complexity : $O(n + w)$

# STL Support

- Searching
  - lower_bound
  - upper_bound
  - binary_search
  - equal_range
- Sorting
  - sort (default ascending)
  - You can write your own comparison criteria
  - e.g. sort descending

```
bool cmp(int u, int v){
        return u > v;
}

sort(a, a + n, cmp);
```

# Q&A