Ch 3

3.1 general method

Example 1

- Give a bag with 16 coins, if one of the coins is fake (the fake one is lighter), how can we find the exact fake one?
- → compare 2 by 2
- → divide into two group

The divide and coquer

```
Algorithm DAndC(P)
{
    if Small(P) then return S(P);
    else
    {
        divide P into smaller instances P_1, P_2, ..., P_k, k ≥ 1;
        Apply DAndC to each of these subproblems;
        return Combine(DAndC(P_1),DAndC(P_2),...,DAndC(P_k));
    }
}
```

The complexity

- $T(n) \rightarrow$
    - $T(1)$
    - $aT(n/b) + f(n)$

Example 3.2

- $a = 2$, $b = 2$, $T(1) = 2$, $f(n) = n$

Master's theorem

- Let f is an increasing function.

$$f(n) = a\, f(n/b) + cn^d$$

where a is a positive integer, b is an integer > 1, c is a positive real number, d is a non-negative real number

$$f(n) = \begin{cases} O(n^d) & a < b^d \\ O(n^d \log n) & a = b^d \\ O(n^{\log_b a}) & a > b^d \end{cases}$$

3.2 defective chessboard

The chessboard

- Give a 2^k*2^k chessboard, and there is a defective square.
- We aim to fill the chessboard by triominoes (L shape)
- Concept: 2^{2k} -1 / 3 is divisible

3.3 binary search

List search

- Give a sorted list, and the members contain a_1, a_2, a_3, .., a_n
- The target is to identify if a variable x is located in the list
- If so, reply the location a_i

- The simplest way is to search from the head to the tail → O(n)

- The given list is sorted --> we can adopt binary search

Recursive version

```
Algorithm BinSrch(a, i, l, x)
// Given an array a[i : l] of elements in nondecreasing
// order, 1 ≤ i ≤ l, determine whether x is present, and
// if so, return j such that x = a[j]; else return 0.
{
    if (l = i) then  // If Small(P)
    {
        if (x = a[i]) then return i;
        else return 0;
    }
    else
    { // Reduce P into a smaller subproblem.
        mid := ⌊(i + l)/2⌋;
        if (x = a[mid]) then return mid;
        else  if (x < a[mid]) then
                  return BinSrch(a, i, mid − 1, x);
              else return BinSrch(a, mid + 1, l, x);
    }
}
```

Iterative version

```
Algorithm BinSearch(a, n, x)
// Given an array a[1 : n] of elements in nondecreasing
// order, n ≥ 0, determine whether x is present, and
// if so, return j such that x = a[j]; else return 0.
{
    low := 1; high := n;
    while (low ≤ high) do
    {
        mid := ⌊(low + high)/2⌋;
        if (x < a[mid]) then high := mid − 1;
        else if (x > a[mid]) then low := mid + 1;
            else return mid;
    }
    return 0;
}
```

Example

$-15, -6, 0, 7, 9, 23, 54, 82, 101, 112, 125, 131, 142, 151$

$x = 151$	*low*	*high*	*mid*
	1	14	7
	8	14	11
	12	14	13
	14	14	14
			found

$x = -14$	*low*	*high*	*mid*
	1	14	7
	1	6	3
	1	2	1
	2	2	2
	2	1	not found

$x = 9$	*low*	*high*	*mid*
	1	14	7
	1	6	3
	4	6	5
			found

successful searches			**unsuccessful searches**
$\Theta(1)$,	$\Theta(\log n)$,	$\Theta(\log n)$	$\Theta(\log n)$
best,	average,	worst	best, average, worst

One comparison version

```
Algorithm BinSearch1(a, n, x)
// Same specifications as BinSearch except n > 0
{
    low := 1; high := n + 1;
    // high is one more than possible.
    while (low < (high − 1)) do
    {
        mid := ⌊(low + high)/2⌋;
        if (x < a[mid]) then high := mid;
            // Only one comparison in the loop.
        else low := mid; // x ≥ a[mid]
    }
    if (x = a[low]) then return low; // x is present.
    else return 0; // x is not present.
}
```

3.4 find the maximum and minimum

A simple version

```
1   Algorithm StraightMaxMin(a, n, max, min)
2   // Set max to the maximum and min to the minimum of a[1 : n].
3   {
4       max := min := a[1];
5       for i := 2 to n do
6       {
7           if (a[i] > max) then max := a[i];
8           if (a[i] < min) then min := a[i];
9       }
10  }
```

- In lines 7 & 8, the algorithm uses IF to comparison
  → total comparison 2(n-1) times
- How to improve?

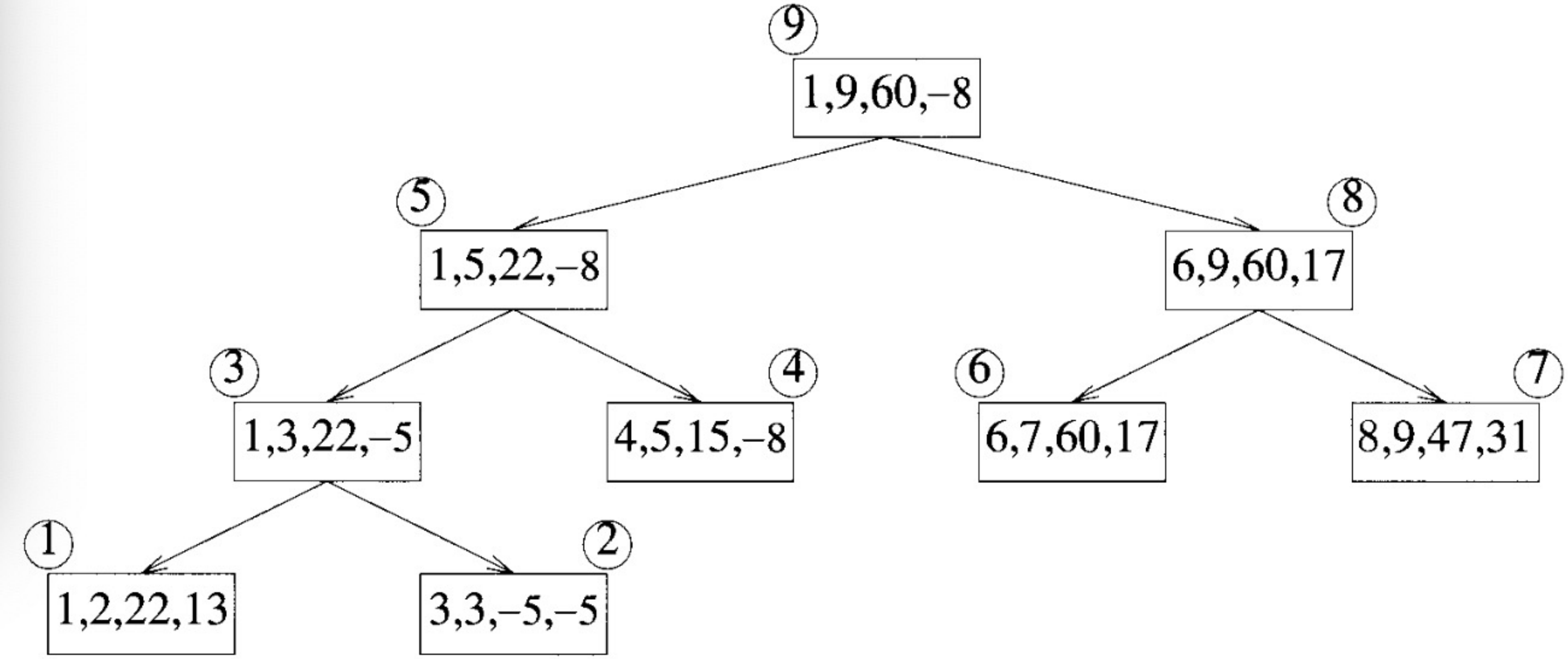
Use divide and conquer

- Split the list continuously until there are one or two elements
- If there is only one → the element is both min and max
- If there are two → one is min and the other is max

- Combine the split list and return the final max and min

```
Algorithm MaxMin(i, j, max, min)
// a[1 : n] is a global array. Parameters i and j are integers,
// 1 ≤ i ≤ j ≤ n. The effect is to set max and min to the
// largest and smallest values in a[i : j], respectively.
{
    if (i = j) then max := min := a[i]; // Small(P)
    else if (i = j − 1) then  // Another case of Small(P)
        {
            if (a[i] < a[j]) then
            {
                max := a[j]; min := a[i];
            }
            else
            {
                max := a[i]; min := a[j];
            }
        }
        else
        {   // If P is not small, divide P into subproblems.
            // Find where to split the set.
                mid := ⌊(i + j)/2⌋;
            // Solve the subproblems.
                MaxMin(i, mid, max, min);
                MaxMin(mid + 1, j, max1, min1);
            // Combine the solutions.
                if (max < max1) then max := max1;
                if (min > min1) then min := min1;
        }
}
```

a:	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
	22	13	-5	-8	15	60	17	31	47



Comparison on StraightMaxMin and MaxMin

- In theorem, the complexities are both O(n)
- The number of comparisons
  - MaxMin: 3/2n - 2
  - StraightMaxMin: 2n -2

- The StraightMaxMin needs extra memory to store ____________

3.5 Merge sort

Concept

- Two subprocesses

- mergesort: main recursion

- merge: merge two sorted sub-lists

8 2 4 6 9 7 10 1 5 3

The time complexity

```
Algorithm MergeSort(low, high)
// a[low : high] is a global array to be sorted.
// Small(P) is true if there is only one element
// to sort. In this case the list is already sorted.
{
    if (low < high) then  // If there are more than one element
    {
        // Divide P into subproblems.
            // Find where to split the set.
                mid := ⌊(low + high)/2⌋;
        // Solve the subproblems.
            MergeSort(low, mid);
            MergeSort(mid + 1, high);
        // Combine the solutions.
            Merge(low, mid, high);
    }
}
```

```
Algorithm Merge(low, mid, high)
// a[low : high] is a global array containing two sorted
// subsets in a[low : mid] and in a[mid + 1 : high]. The goal
// is to merge these two sets into a single set residing
// in a[low : high]. b[ ] is an auxiliary global array.
{
    h := low; i := low; j := mid + 1;
    while ((h ≤ mid) and (j ≤ high)) do
    {
        if (a[h] ≤ a[j]) then
        {
            b[i] := a[h];h := h + 1;
        }
        else
        {
            b[i] := a[j]; j := j + 1;
        }
        i := i + 1;
    }
    if (h > mid) then
        for k := j to high do
        {
            b[i] := a[k]; i := i + 1;
        }
    else
        for k := h to mid do
        {
            b[i] := a[k]; i := i + 1;
        }
    for k := low to high do a[k] := b[k];
}
```

4 | 8 | 11 | 13 | 6 | 10 | 14 | 15

Improvements

- Need 2n memories → use “link” concept to store sorted data
- Along with the original array a[], use an auxiliary array link [1:n]

$link$:	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]
	6	4	7	1	3	0	8	0

- Q=(2, 4, 1, 6)
- R=(5, 3, 7, 8)

```
Algorithm MergeSort1(low, high)
// The global array a[low : high] is sorted in nondecreasing order
// using the auxiliary array link[low : high]. The values in link
// represent a list of the indices low through high giving a[ ] in
// sorted order. A pointer to the beginning of the list is returned.
{
    if ((high − low) < 15) then
        return InsertionSort1(a, link, low, high);
    else
    {
        mid := ⌊(low + high)/2⌋;
        q := MergeSort1(low, mid);
        r := MergeSort1(mid + 1, high);
        return Merge1(q, r);
    }
}
```

```
Algorithm InsertionSort(a, n)
// Sort the array a[1 : n] into nondecreasing order, n ≥ 1.
{
    for j := 2 to n do
    {
        // a[1 : j − 1] is already sorted.
        item := a[j]; i := j − 1;
        while ((i ≥ 1) and (item < a[i])) do
        {
            a[i + 1] := a[i]; i := i − 1;
        }
        a[i + 1] := item;
    }
}
```

```
Algorithm Merge1(q, r)
// q and r are pointers to lists contained in the global array
// link[0 : n]. link[0] is introduced only for convenience and need
// not be initialized. The lists pointed at by q and r are merged
// and a pointer to the beginning of the merged list is returned.
{
    i := q; j := r; k := 0;
    // The new list starts at link[0].
    while ((i ≠ 0) and (j ≠ 0)) do
    { // While both lists are nonempty do
        if (a[i] ≤ a[j]) then
        { // Find the smaller key.
            link[k] := i; k := i; i := link[i];
            // Add a new key to the list.
        }
        else
        {
            link[k] := j; k := j; j := link[j];
        }
    }
    if (i = 0) then link[k] := j;
    else link[k] := i;
    return link[0];
}
```

1. The k is initially zero
2. For the first time it will fill the start from i or j
3. Then the k will be a link to sorted elements

The next one

The lefthand side finished

return the start point for MergeSort1

The returned link[0]

	(0)	(1)	(2)	(3)	(4)	(5)	(6)	(7)	(8)	
a:	-	50	10	25	30	15	70	35	55	
$link$:	0	0	0	0	0	0	0	0	0	
q r p										
1 2 2	2	0	1	0	0	0	0	0	0	(10, 50)
3 4 3	3	0	1	4	0	0	0	0	0	(10, 50), (25, 30)
2 3 2	2	0	3	4	1	0	0	0	0	(10, 25, 30, 50)
5 6 5	5	0	3	4	1	6	0	0	0	(10, 25, 30, 50), (15, 70)
7 8 7	7	0	3	4	1	6	0	8	0	(10, 25, 30, 50), (15, 70), (35, 55)
5 7 5	5	0	3	4	1	7	0	8	6	(10, 25, 30, 50) (15, 35, 55, 70)
2 5 2	2	8	5	4	7	3	0	1	6	(10, 15, 25, 30, 35, 50, 55, 70)

MergeSort1 applied to $a[1:8] = (50, 10, 25, 30, 15, 70, 35, 55)$

3.6 quick sort

Concept

- Give a unsorted list → find a partition element
    - For those numbers that are **smaller** than the partition element → put to **left**
    - For those numbers that are **larger** than the partition element → put to **right**

- In the left part, find another partition element, and then perform the same procedure above

- In the right part, find another partition element, and then perform the same procedure above

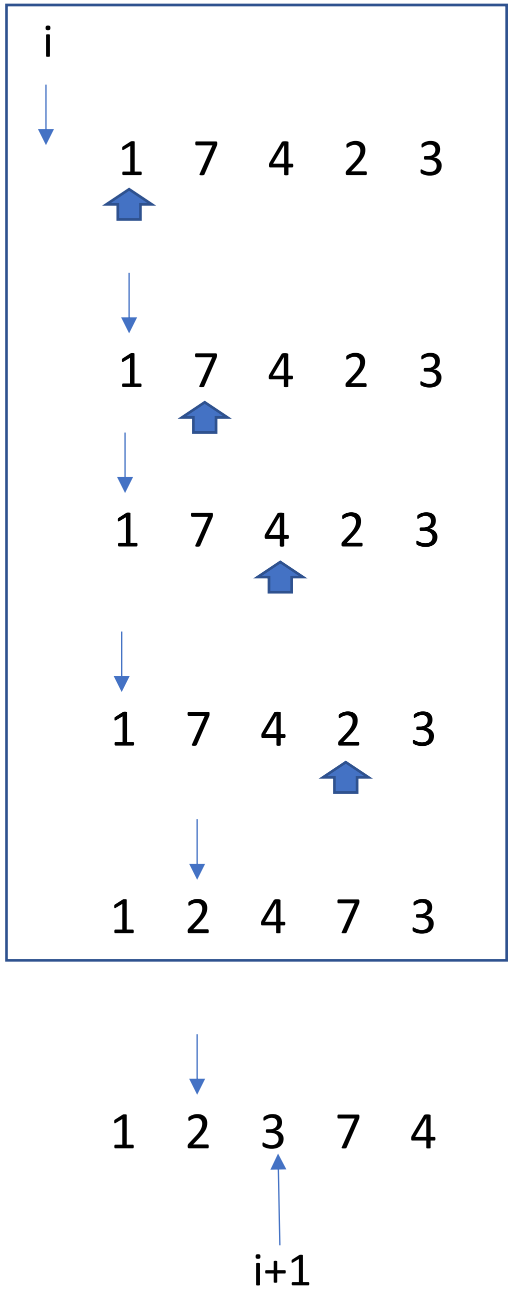
```
Algorithm QuickSort(p, q)
// Sorts the elements a[p], ..., a[q] which reside in the global
// array a[1 : n] into ascending order; a[n + 1] is considered to
// be defined and must be ≥ all the elements in a[1 : n].
{
    if (p < q) then  // If there are more than one element
    {
        // divide P into two subproblems.
            j := Partition(a, p, q + 1);
                // j is the position of the partitioning element.
        // Solve the subproblems.
            QuickSort(p, j - 1);
            QuickSort(j + 1, q);
        // There is no need for combining solutions.
    }
}
```

```
Algorithm Partition(a, m, p)
// Within a[m], a[m+1], ..., a[p-1] the elements are
// rearranged in such a manner that if initially t = a[m],
// then after completion a[q] = t for some q between m
// and p-1, a[k] ≤ t for m ≤ k < q, and a[k] ≥ t
// for q < k < p. q is returned. Set a[p] = ∞.
{
    v := a[m]; i := m; j := p;
    repeat
    {
        repeat
            i := i + 1;
        until (a[i] ≥ v);

        repeat
            j := j - 1;
        until (a[j] ≤ v);

        if (i < j) then Interchange(a, i, j);

    } until (i ≥ j);

    a[m] := a[j]; a[j] := v; return j;
}
```

```
Algorithm Interchange(a, i, j)
// Exchange a[i] with a[j].
{
    p := a[i];
    a[i] := a[j]; a[j] := p;
}
```

```python
def quickSort(array, low, high):
    if low < high:

        # Find pivot element such that
        # element smaller than pivot are on the left
        # element greater than pivot are on the right
        pi = partition(array, low, high)

        # Recursive call on the left of pivot
        quickSort(array, low, pi - 1)

        # Recursive call on the right of pivot
        quickSort(array, pi + 1, high)


data = [1, 7, 4, 1, 10, 9, -2]
print("Unsorted Array")
print(data)

size = len(data)

quickSort(data, 0, size - 1)

print('Sorted Array in Ascending Order:')
print(data)
```

```
def partition(array, low, high):

    # choose the rightmost element as pivot
    pivot = array[high]

    # pointer for greater element
    i = low - 1

    # traverse through all elements
    # compare each element with pivot
    for j in range(low, high):
        if array[j] <= pivot:

            # If element smaller than pivot is found
            # swap it with the greater element pointed by i
            i = i + 1

            # Swapping element at i with element at j
            (array[i], array[j]) = (array[j], array[i])

    # Swap the pivot element with the greater element specified by i
    (array[i + 1], array[high]) = (array[high], array[i + 1])

    # Return the position from where partition is done
    return i + 1
```



Analysis

- For the worst case
  - Decide a improper partition element → the max or min in the list
  - → time complexity: $O(n^2)$
  - → space complexity: $O(n)$

- For the general case → time complexity $O(n \log n)$

Reduce time complexity

- Middle of three
  - Let middle = $\lfloor (m+p-1)/2 \rfloor$
  - Compare a[m], a[p], and a[middle]
  - Use the medium value as the partition element
  - → it is possible induce the worst case scenario

Reduce time complexity

- Randomly pick partition element

```
Algorithm RQuickSort(p, q)
// Sorts the elements a[p], ..., a[q] which reside in the global
// array a[1 : n] into ascending order. a[n + 1] is considered to
// be defined and must be ≥ all the elements in a[1 : n].
{
    if (p < q) then
    {
        if ((q − p) > 5) then
            Interchange(a, Random() mod (q − p + 1) + p, p);
        j := Partition(a, p, q + 1);
            // j is the position of the partitioning element.
        RQuickSort(p, j − 1);
        RQuickSort(j + 1, q);
    }
}
```

Reduce time complexity

- Randomly pick a set of partition elements

```
Algorithm RSort(a, n)
// Sort the elements a[1 : n].
{
    Randomly sample s elements from a[ ];
    Sort this sample;
    Partition the input using the sorted sample as partition keys;
    Sort each part separately;
}
```

Reduce space complexity

```
Algorithm QuickSort2(p, q)
// Sorts the elements in a[p : q].
{
    // stack is a stack of size 2 log(n).
    repeat
    {
        while (p < q) do
        {
            j := Partition(a, p, q + 1);
            if ((j − p) < (q − j)) then
            {
                Add(j + 1); // Add j + 1 to stack.
                Add(q); q := j − 1; // Add q to stack
            }
            else
            {
                Add(p); // Add p to stack.
                Add(j − 1); p := j + 1; // Add j − 1 to stack
            }
        } // Sort the smaller subfile.
        if stack is empty then return;
        Delete(q); Delete(p); // Delete q and p from stack.
    } until (false);
}
```

The updated partition element

If there are less numbers in left part → perform the left part earlier

Add the right part's left boundary

Add the right part's right boundary

Update q to the left part's right boundary

→ Then the while loop will execute based on the left part

1. After finish the loop above, check the stack.
2. Pop the top most two element to update p and q.
3. Then, the while loop will perform again.

Infinite loop

3.7 selection

Objective

- Give an array, we aim to find the kth-smallest element
  → adopt the concept of partition in quick sort

- The worst case complexity will be $O(n^2)$
- The average case complexity will be $O(n \log n)$

```
Algorithm Select1(a, n, k)
// Selects the kth-smallest element in a[1 : n] and places it
// in the kth position of a[ ]. The remaining elements are
// rearranged such that a[m] ≤ a[k] for 1 ≤ m < k, and
// a[m] ≥ a[k] for k < m ≤ n.
{
    low := 1; up := n + 1;
    a[n + 1] := ∞; // a[n + 1] is set to infinity.
    repeat
    {
        // Each time the loop is entered,
        // 1 ≤ low ≤ k ≤ up ≤ n + 1.
        j := Partition(a, low, up);
            // j is such that a[j] is the jth-smallest value in a[ ].
        if (k = j) then return;
        else if (k < j) then up := j; // j is the new upper limit.
            else low := j + 1; // j + 1 is the new lower limit.
    } until (false);
}
```