

Unit 3.3 More on Divide and Conquer Algorithms

Algorithms

EE/NTHU

Apr. 9, 2020

Selection Algorithm

- The selection problem is to find the k 'th smallest element of an array A and place all elements less than or equal to $A[k]$ in $A[1 : k - 1]$ and the rest in $A[k + 1 : n]$.
- Divide and conquer can be applied to this problem as well.
 - The **Partition** algorithm can be effective in this selection problem.

Algorithm 3.3.1. Selection

```
// Partition the array into  $A[1 : k - 1] \leq A[k] \leq A[k + 1 : n]$ .  
// Input:  $A$ , int  $n$ ,  $k$   
// Output:  $k$ -th smallest element of  $A$ , and with  $A$  rearranged.  
1 Algorithm Select1( $A$ ,  $n$ ,  $k$ )  
2 {  
3      $low := 1$ ;  $high := n + 1$ ;  $A[n + 1] := \infty$ ; // Initialize ranges and  $j$ .  
4     repeat { // Loop until value found by Partition is  $k$ .  
5          $j := \text{Partition}(A, low, high)$ ;  
6         if ( $j > k$ ) then  $high := j$ ; //  $j \neq k$  then update range for search.  
7         else if ( $j < k$ ) then  $low := j + 1$ ;  
8     } until ( $j = k$ );  
9     return  $A[k]$ ;  
10 }
```

Selection Algorithm – Complexity

- Note that the **Partition**($A, low, high$) decrease the range of the array A by at least 1.
- Thus, the worst-case complexity of the **Select1** algorithm is $\mathcal{O}(n^2)$.
- Let $T_A^k(n)$ be the average time to find the k 'th smallest element in $A[1 : n]$
 - The average is taken over all $n!$ different permutations.
- Define

$$T_A(n) = \frac{1}{n} \sum_{k=1}^n T_A^k(n) \quad (3.3.1)$$

$$R(n) = \max_k T_A^k(n) \quad (3.3.2)$$

- $T_A(n)$ is the average execution time of **Select1** algorithm,
- And it is obvious that $T_A(n) \leq R(n)$.

Selection Algorithm – Complexity, II

Theorem 3.3.2.

The average execution time $T_A(n)$ of **Select1** algorithm is $\mathcal{O}(n)$.

Proof. The complexity of **Partition** algorithm is $\mathcal{O}(n)$ and hence there is a constant c such that

$$T_A^k(n) \leq c \cdot n + \frac{1}{n} \left(\sum_{i=1}^{k-1} T_A^k(n-i) + \sum_{i=k+1}^n T_A^k(i-1) \right).$$

Define $R(n) = \max_k T_A^k(n)$, then

$$\begin{aligned} R(n) &\leq c \cdot n + \frac{1}{n} \max_k \left(\sum_{i=1}^{k-1} R(n-i) + \sum_{i=k+1}^n R(i-1) \right), \\ &= c \cdot n + \frac{1}{n} \max_k \left(\sum_{i=n-k+1}^{n-1} R(i) + \sum_{i=k}^{n-1} R(i) \right). \end{aligned}$$

To show by induction that $R(n) \leq 4c \cdot n$.

Selection Algorithm – Complexity, III

For $n = 2$,

$$\begin{aligned} R(n) &\leq 2 \cdot c + \frac{1}{2} \max(R(1), R(1)) = 2 \cdot c + \frac{1}{2} \max(4c, 4c) \\ &= 2 \cdot c + 2 \cdot c < 4 \cdot c \cdot n \end{aligned}$$

Next assume $R(n) \leq 4 \cdot c \cdot n$ for $2 \leq n < m$.

For $n = m$,

$$R(m) \leq c \cdot m + \frac{1}{m} \max_k \left(\sum_{i=m-k+1}^{m-1} R(i) + \sum_{i=k}^{m-1} R(i) \right).$$

Since $R(n)$ is a nondecreasing function of n , the term

$$\sum_{i=m-k+1}^{m-1} R(i) + \sum_{i=k}^{m-1} R(i)$$

is maximum when $k = m/2$ when m is even, and $k = (m+1)/2$ when m is odd.

Selection Algorithm – Complexity, IV

When m is even

$$\begin{aligned} R(m) &\leq c \cdot m + \frac{2}{m} \sum_{i=m/2}^{m-1} R(i) = c \cdot m + \frac{8c (3m/2 - 1)(m/2)}{2} \\ &= c \cdot m + c(3m - 2) = 4 \cdot c \cdot m - 2c < 4 \cdot c \cdot m \end{aligned}$$

Similarly, when m is odd

$$\begin{aligned} R(m) &\leq c \cdot m + \frac{2}{m} \sum_{i=(m+1)/2}^{m-1} R(i) = c \cdot m + \frac{8c (3m-1)/2 \cdot (m-1)/2}{2} \\ &= c \cdot m + \frac{c}{m} (3m^2 - 4m + 1) = 4 \cdot c \cdot m - 4c + c/m < 4 \cdot c \cdot m \end{aligned}$$

Since $T_A(n) \leq R(n)$, therefore $T_A(n) \leq 4 \cdot c \cdot n$ and $T_A(n)$ is $\mathcal{O}(n)$. \square

- The space complexity of the **Select1** algorithm is $\mathcal{O}(n)$ for the array A .
- The **Select1** algorithm can also be randomized as **RQuickSort** algorithm.
 - The expected time complexity is still $\mathcal{O}(n)$.
 - But the average performance is expected to be better.

Selection Algorithm – Complexity, V

- The execution time of **Select1** in the worst-case is $\mathcal{O}(n^2)$.
 - The worst-case can happen if the partition element, $A[\text{low}]$, is close to extreme.
 - If the partition element is close to the median, $A[(\text{low} + \text{high})/2]$, then the number of iterations can be reduced significantly.
 - Using this argument, the selection algorithm is modified to have worst-case linear time complexity.
- The array A is divided into subarrays each has r elements
 - $\lceil n/r \rceil$ groups
 - A small r is usually preferred ($r = 5$, for example).
- Then the median of each group is found and move to the front array A
- The median of the medians, mm , is then found using **Partition** function
- Now, this mm can be used to partition array A .
- Since mm is used for each partition step in the selection algorithm, worst-case linear time can be guaranteed.
- Note that though **Select2** is worst-case linear, it has a much larger coefficient, as compared to **Select1**, thus for small to median-size problems, **Select2** may not be faster in execution.
 - **Select2** returns the position j such that $A[j]$ is the k 'th smallest element.

Selection Algorithm – Worst-case Linear

Algorithm 3.3.3. Selection – Worst-case Linear

```
// Partition the array into  $A[1 : k - 1] \leq A[k] \leq A[k + 1 : n]$ .
// Input:  $A$ , int  $k$ ,  $\text{low}$ ,  $\text{high}$ ,  $r$ 
// Output:  $k$ -th element of  $A$ .
1 Algorithm Select2( $A, k, \text{low}, \text{high}, r$ )
2 {
3      $j := k + \text{low} - 1$ ;
4     while ( $k \neq j - \text{low}$ ) do {
5          $n := \text{high} - \text{low} + 1$ ;
6         if ( $n \leq r$ ) then { // small array
7             InsertionSort( $A, \text{low}, \text{high}$ );
8             return  $\text{low} + k$ ;
9         }
10        for  $i := 1$  to  $\lceil n/r \rceil$  do { // find median of each group and move to front
11            InsertionSort( $A, \text{low} + (i - 1) * r, \text{low} + i * r - 1$ );
12            Swap( $\text{low} + i - 1, \text{low} + (i - 1) * r + \lceil r/2 \rceil - 1$ );
13        }
14         $j := \text{Select2}(A, \lceil \lceil n/r \rceil / 2 \rceil, \text{low}, \text{low} + \lceil n/r \rceil - 1)$ ; // find median of medians
15        Swap( $\text{low}, j$ ); // move median of median to  $A[\text{low}]$ 
16         $j := \text{Partition}(A, \text{low}, \text{high} + 1)$ ;
17        if ( $k < j - \text{low}$ )  $\text{high} := j$ ; // reduce to  $A[\text{low} : j]$ 
18        else if ( $k > j - \text{low}$ ) { // reduce to  $A[j + 1 : \text{high}]$ 
19             $k := k - (j - \text{low} + 1)$ ;
20             $\text{low} := j + 1$ ;
21        }
22    }
23    return  $j$ ;
24 }
```

Matrix Multiplication

- Given two $n \times n$ matrices \mathbf{A} and \mathbf{B} , $\mathbf{A}[i, j] \in \mathbb{R}$, $\mathbf{B}[i, j] \in \mathbb{R}$, $1 \leq i, j \leq n$, then $n \times n$ matrix \mathbf{C} is the product of \mathbf{A} and \mathbf{B} , ($\mathbf{C} = \mathbf{A} \cdot \mathbf{B}$),

$$\mathbf{C}[i, j] = \sum_{k=1}^n \mathbf{A}[i, k] \times \mathbf{B}[k, j], \quad 1 \leq i, j \leq n. \quad (3.3.3)$$

- Note that to calculate $\mathbf{C}[i, j]$, one needs n multiplications and $n - 1$ additions.
- Thus to calculate \mathbf{C} , which has n^2 elements, the time complexity is $\Theta(n^3)$.

Matrix Multiplication – Divide and Conquer

- Suppose $n = 2^k$, we can apply divide and conquer approach to matrix multiplication problem.
- Divide each matrix into 4 submatrices with $\frac{n}{2} \times \frac{n}{2}$ dimensions each, then

$$\begin{bmatrix} \mathbf{A}_{11} & \mathbf{A}_{12} \\ \mathbf{A}_{21} & \mathbf{A}_{22} \end{bmatrix} \begin{bmatrix} \mathbf{B}_{11} & \mathbf{B}_{12} \\ \mathbf{B}_{21} & \mathbf{B}_{22} \end{bmatrix} = \begin{bmatrix} \mathbf{C}_{11} & \mathbf{C}_{12} \\ \mathbf{C}_{21} & \mathbf{C}_{22} \end{bmatrix} \quad (3.3.4)$$

where

$$\begin{aligned} \mathbf{C}_{11} &= \mathbf{A}_{11}\mathbf{B}_{11} + \mathbf{A}_{12}\mathbf{B}_{21} \\ \mathbf{C}_{12} &= \mathbf{A}_{11}\mathbf{B}_{12} + \mathbf{A}_{12}\mathbf{B}_{22} \\ \mathbf{C}_{21} &= \mathbf{A}_{21}\mathbf{B}_{11} + \mathbf{A}_{22}\mathbf{B}_{21} \\ \mathbf{C}_{22} &= \mathbf{A}_{21}\mathbf{B}_{12} + \mathbf{A}_{22}\mathbf{B}_{22} \end{aligned} \quad (3.3.5)$$

- To calculate matrix \mathbf{C} , we need
 - Eight matrix multiplications ($\frac{n}{2} \times \frac{n}{2}$),
 - Four matrix additions ($\mathcal{O}(n^2)$ complexity due to n^2 elements in \mathbf{C}).
- Let $T(n)$ be the complexity, then

$$T(n) = \begin{cases} b, & n \leq 2 \\ 8 \cdot T(n/2) + c \cdot n^2, & n > 2. \end{cases} \quad (3.3.6)$$

where b and c are constants.

Matrix Multiplication – Divide and Conquer, II

- If $n = 2^k$

$$\begin{aligned}T(n) &= 8T(n/2) + c \cdot n^2 \\&= 8 \left[8T(n/4) + c \cdot \left(\frac{n}{2}\right)^2 \right] + c \cdot n^2 \\&= 8^2 T(n/4) + c \cdot n^2 (2 + 1) \\&= 8^3 T(n/8) + c \cdot n^2 (4 + 2 + 1) \\&= 8^{k-1} T(n/2^{k-1}) + c \cdot n^2 \sum_{i=0}^{k-2} 2^i \\&= 2^{3k-3} b + c \cdot n^2 (2^{k-1}) \\&= \frac{n^3}{8} b + c \cdot n^2 \left(\frac{n}{2}\right) \\&= \left(\frac{b}{8} + \frac{c}{2}\right) n^3 \\&= \mathcal{O}(n^3)\end{aligned}$$

- Thus, this divide and conquer approach does not improve the computational complexity

Strassen's Matrix Multiplication

- Given Equations (3.3.4) and (3.3.5), define the following

$$\begin{aligned}\mathbf{P} &= (\mathbf{A}_{11} + \mathbf{A}_{22})(\mathbf{B}_{11} + \mathbf{B}_{22}) \\ \mathbf{Q} &= (\mathbf{A}_{21} + \mathbf{A}_{22})\mathbf{B}_{11} \\ \mathbf{R} &= \mathbf{A}_{11}(\mathbf{B}_{12} - \mathbf{B}_{22}) \\ \mathbf{S} &= \mathbf{A}_{22}(\mathbf{B}_{21} - \mathbf{B}_{11}) \\ \mathbf{T} &= (\mathbf{A}_{11} + \mathbf{A}_{12})\mathbf{B}_{22} \\ \mathbf{U} &= (\mathbf{A}_{21} - \mathbf{A}_{11})(\mathbf{B}_{11} + \mathbf{B}_{12}) \\ \mathbf{V} &= (\mathbf{A}_{12} - \mathbf{A}_{22})(\mathbf{B}_{21} + \mathbf{B}_{22})\end{aligned}\tag{3.3.7}$$

Then

$$\begin{aligned}\mathbf{C}_{11} &= \mathbf{P} + \mathbf{S} - \mathbf{T} + \mathbf{V} \\ \mathbf{C}_{12} &= \mathbf{R} + \mathbf{T} \\ \mathbf{C}_{21} &= \mathbf{Q} + \mathbf{S} \\ \mathbf{C}_{22} &= \mathbf{P} + \mathbf{R} - \mathbf{Q} + \mathbf{U}\end{aligned}\tag{3.3.8}$$

- To find matrix \mathbf{C} , we need 7 matrix multiplications of $\frac{n}{2} \times \frac{n}{2}$ and 18 matrix additions.
- Since matrix multiplications, $\mathcal{O}(n^3)$, is more expensive than matrix addition, $\mathcal{O}(n^2)$, for large n this approach might be more efficient.

Strassen's Matrix Multiplication, II

- The recurrence relation for the computation time $T(n)$ is

$$T(n) = \begin{cases} b, & n \leq 2, \\ 7 \cdot T(n/2) + c \cdot n^2, & n > 2. \end{cases} \quad (3.3.9)$$

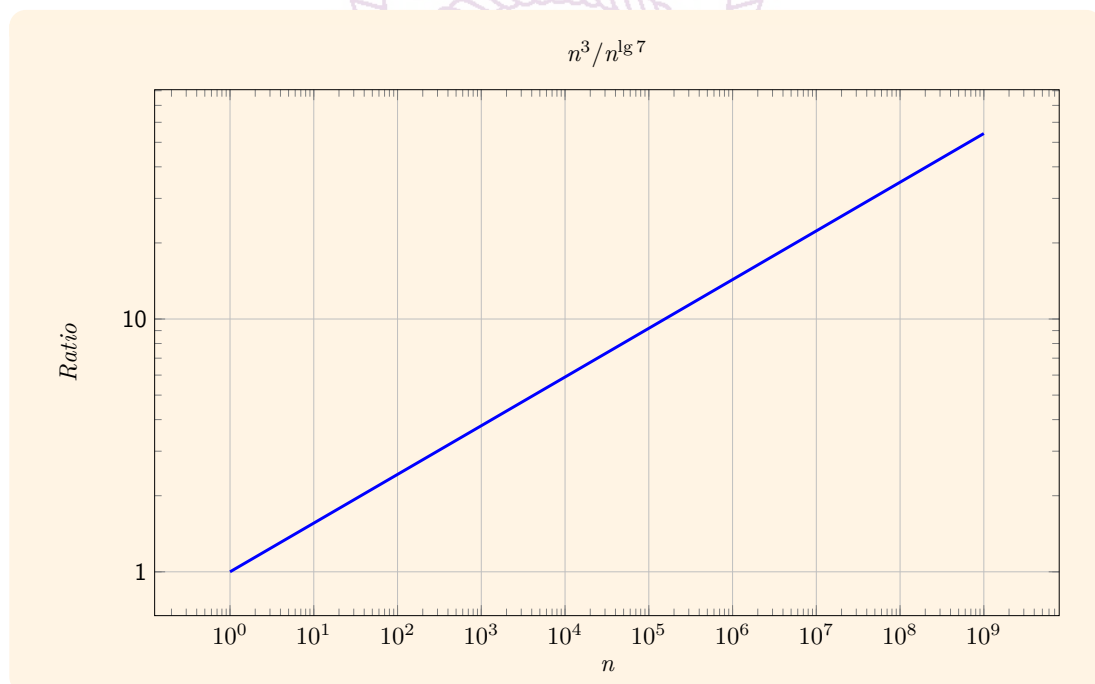
where b and c are two constants.

- If $n = 2^k$, then

$$\begin{aligned} T(n) &= 7 \cdot T(n/2) + c \cdot n^2 \\ &= 7^2 \cdot T(n/4) + 7 \cdot c \cdot (n/2)^2 + c \cdot n^2 \\ &= 7^2 \cdot T(n/4) + c \cdot n^2 (7/4 + 1) \\ &= 7^{k-1} \cdot T(n/2^{k-1}) + c \cdot n^2 \sum_{i=0}^{k-2} (7/4)^i \\ &= 7^{k-1} \cdot b + c \cdot n^2 \left((7/4)^{k-1} - 1 \right) / (3/4) \\ &\approx n^{\lg 7} \cdot \frac{b}{7} + c' n^{\lg 4 + \lg 7 - \lg 4} \\ &= \mathcal{O}(n^{\lg 7}) \approx \mathcal{O}(n^{2.807}) \end{aligned}$$

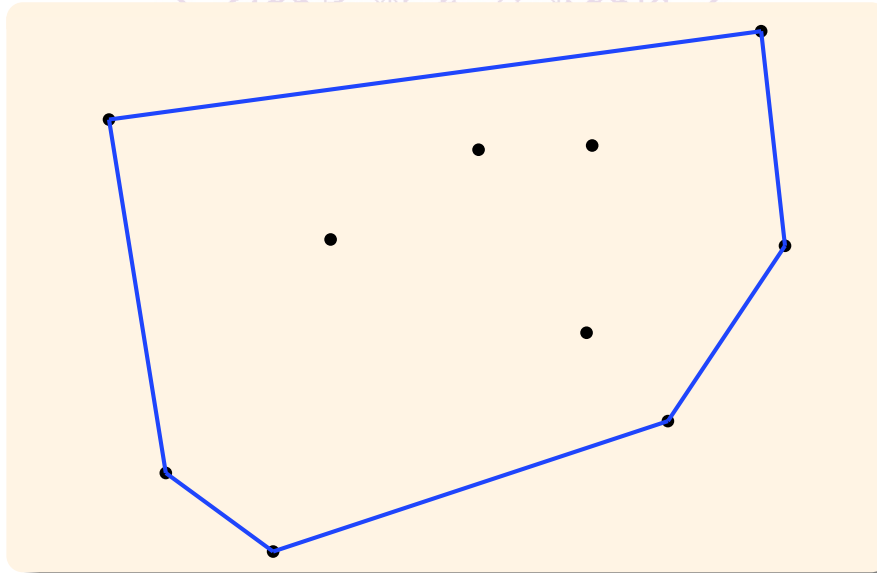
Strassen's Matrix Multiplication, III

- Compared to direct matrix multiplication, $\mathcal{O}(n^3)$, Strassen's approach can be faster for large n .
 - But, coding is much more complex



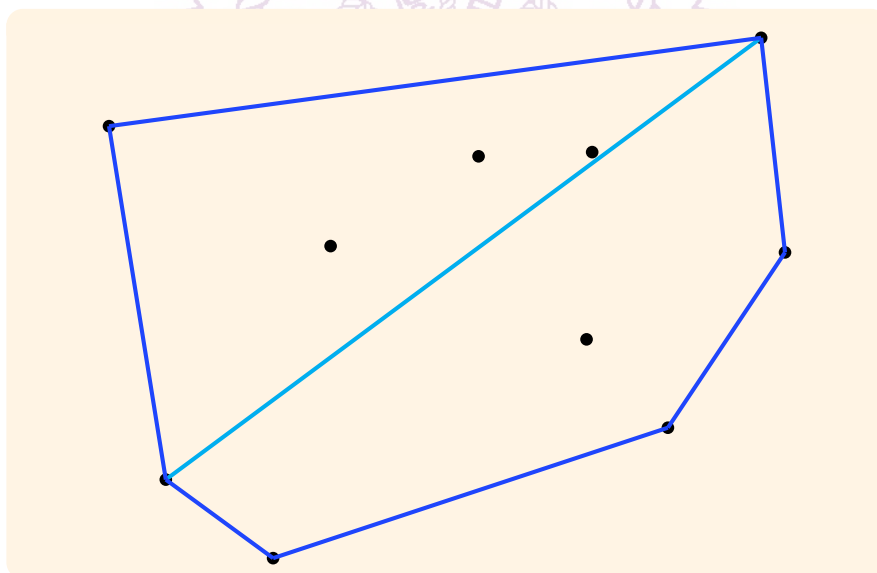
Convex Hull Problem

- Given a set S that contains points on a 2-D plane, the **convex hull** is defined as the smallest convex polygon that contains all the points in S .
- A polygon is **convex** if for any two points p_1, p_2 inside of the polygon, the straight line segment connecting p_1 and p_2 is fully contained in the polygon.
- The vertices of the convex hull of a set S is a subset of S .
 - But, not necessarily a proper subset.



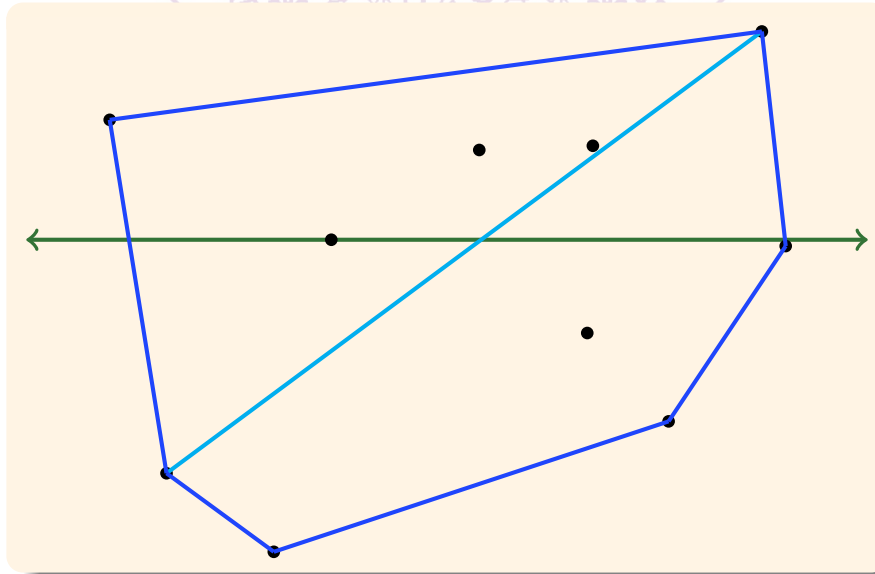
Convex Hull – Direct Implementation

- The convex hull of S can be found using the definition above
 - For any $p_1 \in S$, if p_1 is inside the triangle formed by $p_2, p_3, p_4 \in S$, with $p_1 \notin \{p_2, p_3, p_4\}$, then p_1 is not a vertex of the convex hull.
- This direct implementation has the time complexity of $\mathcal{O}(n^4)$.
 - n points to be tested, n^3 for all possible triangles.



Convex Hull – Direct Implementation, II

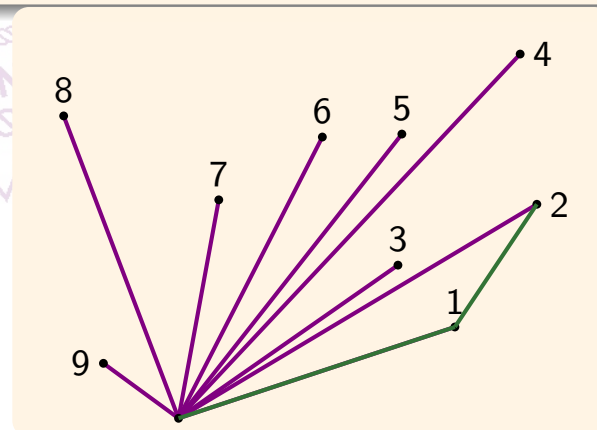
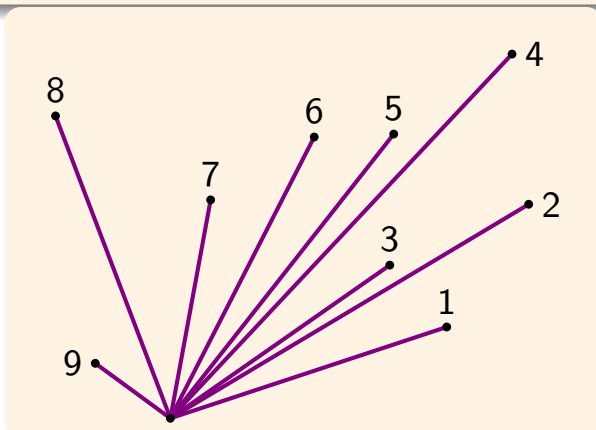
- To test if a point p_1 is inside of a triangle $\triangle p_2 p_3 p_4$
 - Let L be the horizontal line passing through $p_1 = (x_1, y_1)$, note that L can be described by the linear equation $y = y_1$, then check if L intersects with any of the line segments, $\overline{p_2 p_3}$, $\overline{p_3 p_4}$, $\overline{p_4 p_2}$. If not, then p_1 is outside of $\triangle p_2 p_3 p_4$. Otherwise let (x_a, y_1) and (x_b, y_1) be the intersect points, if $x_a \leq x_1 \leq x_b$ then p_1 is inside of the triangle. (Note that it is possible $x_a = x_b$.)



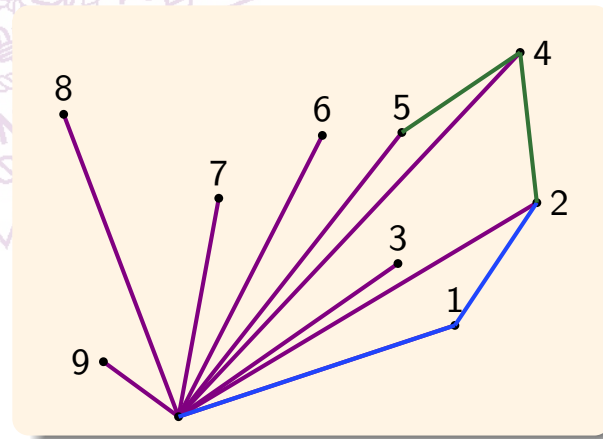
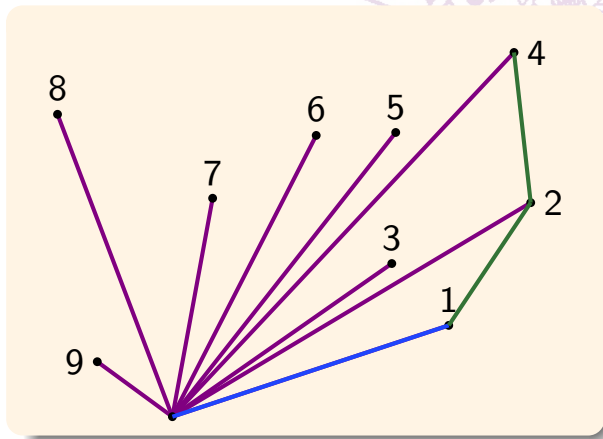
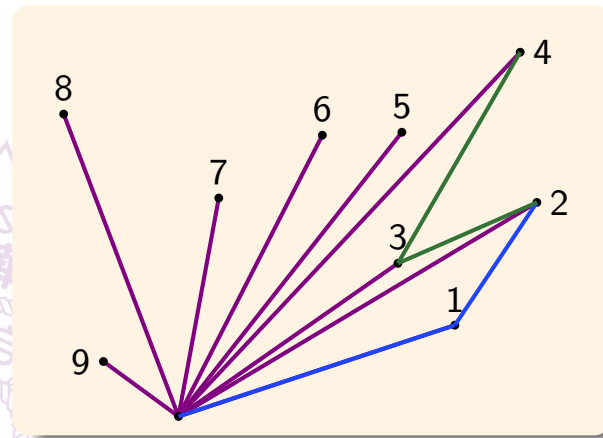
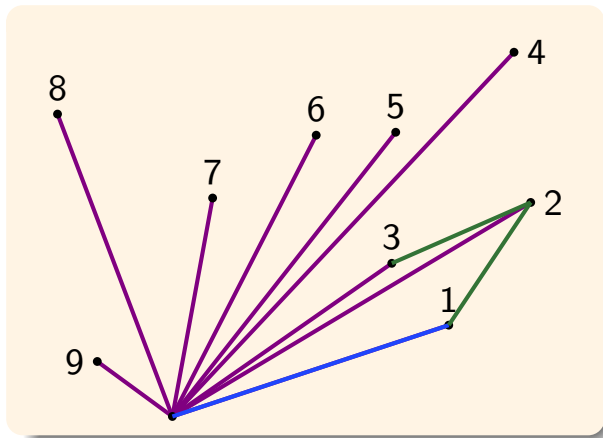
Convex Hull – Graham's Scan

Algorithm 3.3.4. Convex Hull

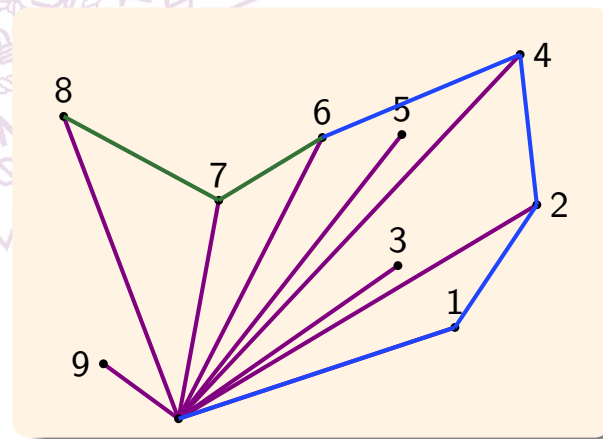
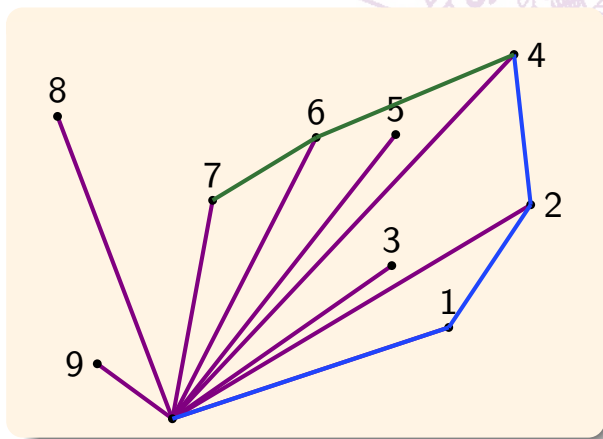
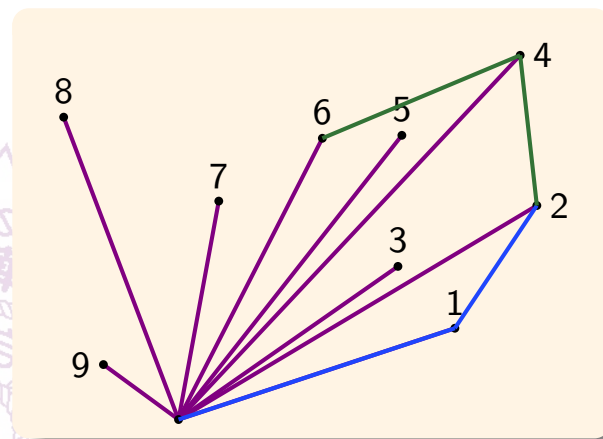
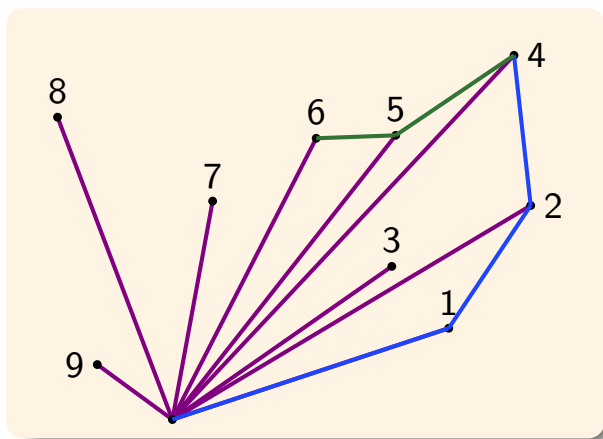
```
// Find the convex hull of a set of points.
// Input: ptslist linked list of points
// Output: convex hull of ptslist.
1 Algorithm ConvexHull(ptslist)
2 {
3     Let the first element of ptrlist has the smallest y coordinate ;
4     Sort(ptrlist) ; // Sort by angle between p and x-axis.
5     Scan(ptrlist) ;
6     PrintList(ptrlist) ;
7 }
```



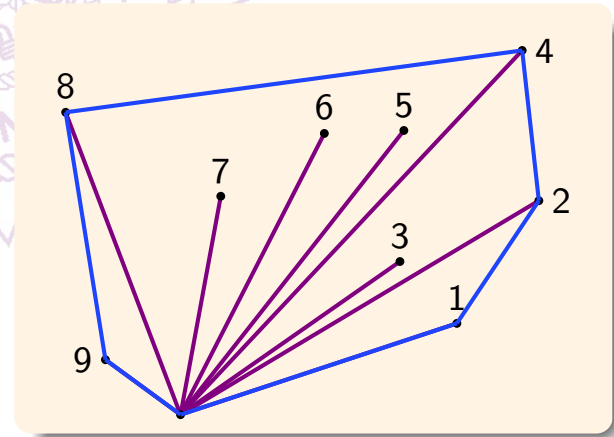
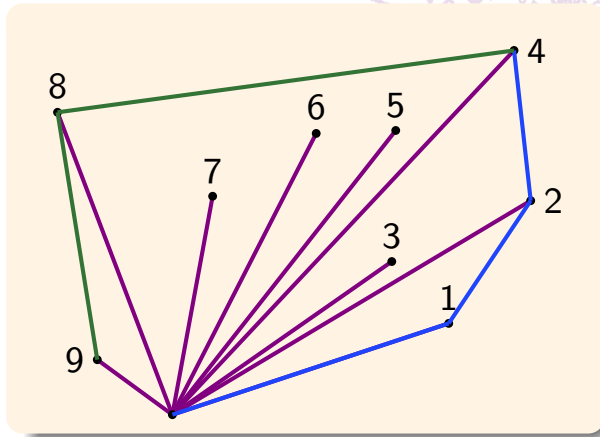
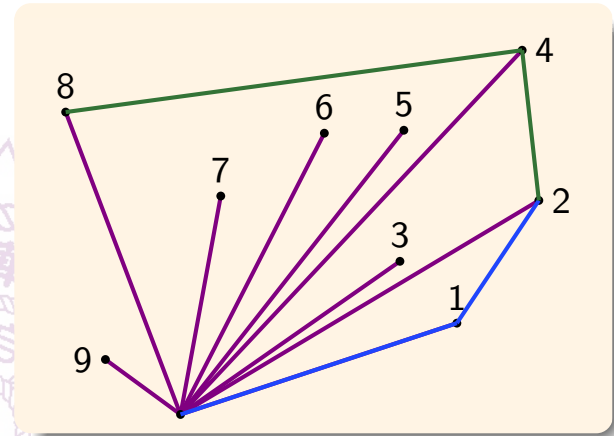
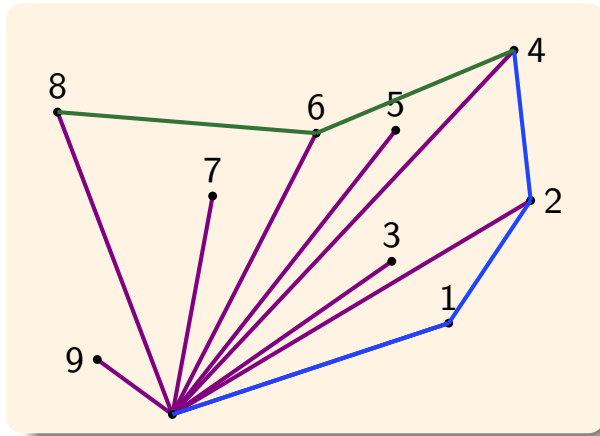
Convex Hull – Graham's Scan, II



Convex Hull – Graham's Scan, III



Convex Hull – Graham's Scan, IV



Convex Hull – Graham's Scan Algorithm

Algorithm 3.3.5. Graham's Scan Algorithm

```
// Remove internal points while doing convex hull.
// Input: list, linked list of points
// Output: none, list modified.
1 Algorithm Scan(list)
2 {
3     p1 := list; p2 := p1→next;
4     while (p2→next ≠ NULL) do { // For all points on the sorted list.
5         p3 := p2→next;
6         if (Area(p1, p2, p3) > 0) then //  $\overrightarrow{p_1 p_2 p_3}$  turning left, accept p2.
7             p1 := p1→next;
8         else {
9             p1→next := p3; // Remove p2 from the list.
10            p3→prev := p1;
11            delete p2;
12            p1 := p1→prev; // Backtrack p1.
13        }
14        p2 := p1→next;
15    }
16 }
```

Convex Hull – Graham's Scan Algorithm, II

- In the preceding algorithm, the points are in linked list form consists

```
struct Point {  
    double x, y;  
    struct Point *next, *prev;  
}
```

- Note that this is a double linked list.
- Let $p_1(x_1, y_1)$, $p_2(x_2, y_2)$ and $p_3(x_3, y_3)$ be three points in a plane the function $\text{Area}(p_1, p_2, p_3)$ is defined as

$$\det \begin{vmatrix} x_1 & y_1 & 1 \\ x_2 & y_2 & 1 \\ x_3 & y_3 & 1 \end{vmatrix} \quad (3.3.10)$$

- It can be shown that
 - If the area is positive then p_3 is located to the left of the vector $\overrightarrow{p_1 p_2}$.
 - If the area is negative then p_3 is located to the right of the vector $\overrightarrow{p_1 p_2}$.
 - If the area is zero then p_3 is colinear with $\overrightarrow{p_1 p_2}$.

Convex Hull – Graham's Scan Algorithm, Complexity

- The Algorithm (3.3.4) consists of 3 steps
 - (line 4) finding the first element with the smallest y coordinate can be done in $\mathcal{O}(n)$ time,
 - (line 5) sort by the angle can be done in $\mathcal{O}(n \lg n)$ time,
 - (line 6) Graham's Scan can be done in $\mathcal{O}(n)$ time.
- Thus, the time complexity is $\mathcal{O}(n \lg n)$.

Quick Hull Algorithm

- Divide and conquer approach can be used to find the convex hull.

Algorithm 3.3.6. QuickHull

```
// Find Convex Hull for points in list.
// Input: list, linked list of points
// Output: CHull: convex hull of list.
1 Algorithm QuickHull(list, CHull)
2 {
3     Find  $p_1 \in list$  with the smallest  $x$  coordinate .
4     Find  $p_2 \in list$  with the largest  $x$  coordinate .
5     Let  $X_1 := \{p \mid \text{Area}(p_1, p_2, p) > 0\}$ . // Upper half.
6     Let  $X_2 := \{p \mid \text{Area}(p_1, p_2, p) < 0\}$ . // Lower half.
7     Hull( $p_1, p_2, X_1$ , UpperHull); // Create upper hull.
8     Hull( $p_2, p_1, X_2$ , LowerHull); // Lower hull.
9     CHull := Merge(UpperHull, LowerHull); // Merge them.
10 }
```

- Finding p_1 and p_2 takes $\mathcal{O}(n)$ time.
- Finding X_1 and X_2 takes $\mathcal{O}(n)$ time.
- Merge takes no more than $\mathcal{O}(n)$ time.
- The time complexity can be dominated by Hull function.

Quick Hull Algorithm, II

Algorithm 3.3.7. QuickHull

```
// Find Convex Hull for  $p_1, p_2$  and list.
// Input:  $p_1, p_2$ , and list
// Output: Convex Hull, CHull.
1 Algorithm Hull( $p_1, p_2, list, CHull$ )
2 {
3     Find  $p_3 \in list$  with the largest  $|\text{Area}(p_1, p_2, p_3)|$ ;
4     Let  $X_1 := \{p \mid \text{Area}(p_1, p_3, p) > 0\}$ . // All points left to  $\overrightarrow{p_1 p_3}$ 
5     if ( $X_1 = \emptyset$ ) then  $H_1 := \{p_1, p_3\}$ ; // No more points.
6     else Hull( $p_1, p_3, X_1, H_1$ ); // Recursive call if more points.
7     Let  $X_2 := \{p \mid \text{Area}(p_3, p_2, p) > 0\}$ .
8     if ( $X_2 = \emptyset$ ) then  $H_2 := \{p_3, p_2\}$ ;
9     else Hull( $p_3, p_2, X_2, H_2$ );
10    CHull := Merge( $H_1, H_2$ ); // Combine those two hulls.
11 }
```

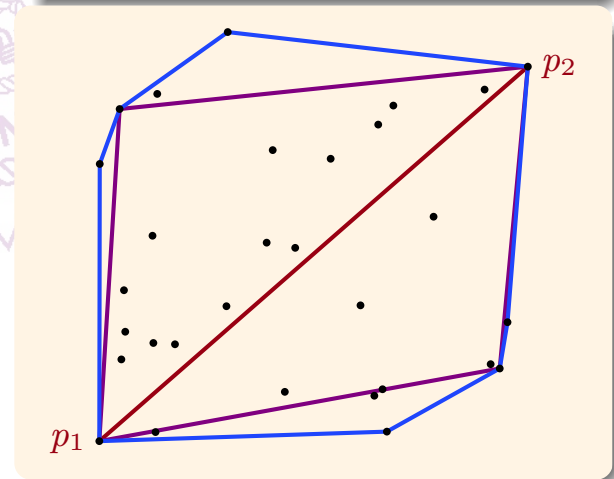
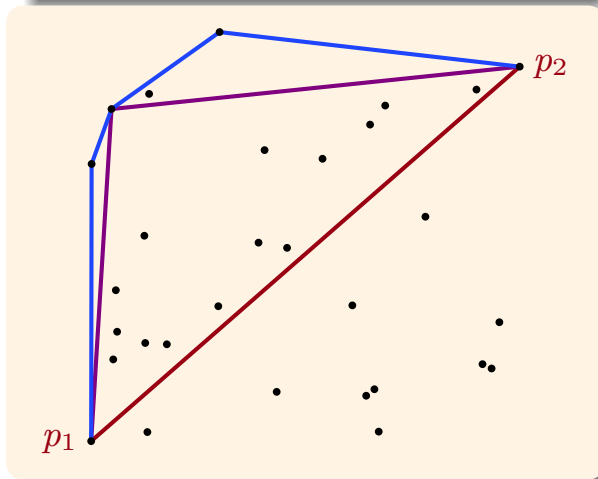
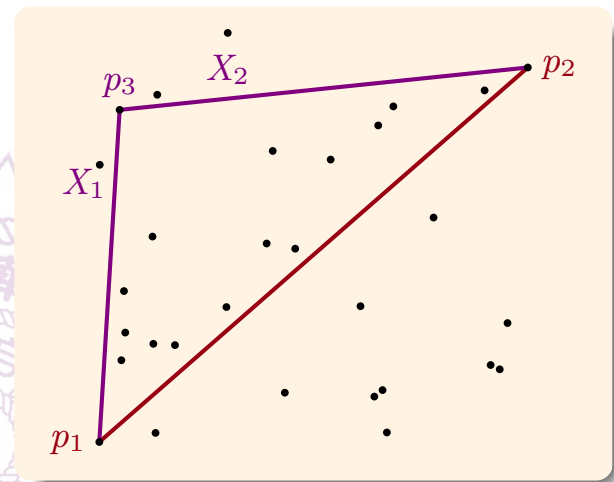
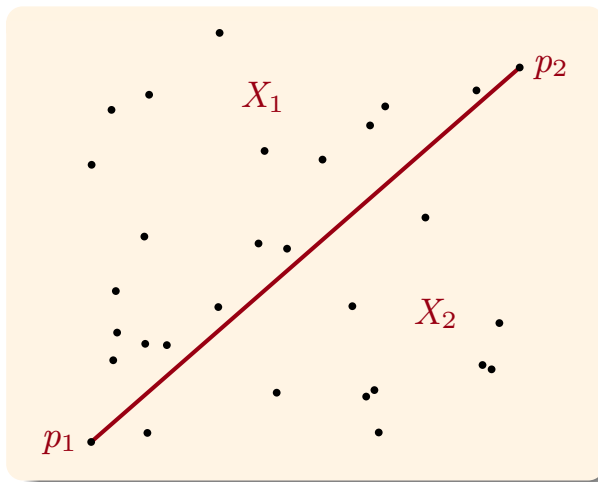
- Finding p_3, X_1 , and X_2 take $\mathcal{O}(m)$ time, if *list* has m points.
- Thus, if $T(m)$ is the time for Hull algorithm we have

$$T(m) = T(m_1) + T(m_2) + \mathcal{O}(m), \quad (3.3.11)$$

where $m_1 + m_2 \leq m$.

- This recurrence relationship is the same as QuickSort.
 - Worst-case complexity is $\mathcal{O}(m^2)$, and average-case is $\mathcal{O}(m \lg m)$.

Quick Hull Example



Time Complexity of Divide and Conquer Algorithms

- In Algorithm **DandC** (Algorithm 3.1.1) the problem is divided into k subproblems; each solved recursively; then the results are combined to form the final solution.
- The execution time can be assumed to have a general recurrence equation as

$$T(n) = a \cdot T(n/k) + f(n). \quad (3.3.12)$$

where $f(n)$ is the time to divide problem into k subsets and to combine the subsets to form the final solution.

- Let $n = k^m$, then

$$\begin{aligned} T(n) &= a \cdot T(n/k) + f(n) = a \cdot T(k^{m-1}) + f(k^m) \\ &= a \cdot \left(a \cdot T(k^{m-2}) + f(k^{m-1}) \right) + f(k^m) = a^2 \cdot T(k^{m-2}) + a \cdot f(k^{m-1}) + f(k^m) \\ &= a^m \cdot T(1) + \sum_{i=0}^{m-1} a^i \cdot f(k^{m-i}) = a^m \left(T(1) + \sum_{i=0}^{m-1} f(k^{m-i}) / a^{m-i} \right) \\ &= a^{\log_k n} \left(T(1) + \sum_{i=0}^{m-1} f(k^{m-i}) / a^{m-i} \right) = n^{\log_k a} \left(T(1) + \sum_{i=1}^{\log_k n} f(k^i) / a^i \right) \end{aligned}$$

Time Complexity of Divide and Conquer Algorithms, II

- Let's assume further that $f(n) = \Theta(n^d) \approx n^d$, then

$$\begin{aligned} T(n) &= n^{\log_k a} \left(T(1) + \sum_{i=1}^{\log_k n} f(k^i)/a^i \right) = n^{\log_k a} \left(T(1) + \sum_{i=1}^{\log_k n} k^{d \cdot i}/a^i \right) \\ &= n^{\log_k a} \left(T(1) + \sum_{i=1}^{\log_k n} (k^d/a)^i \right) \end{aligned} \quad (3.3.13)$$

- If $k^d = a$, or $d = \log_k a$, then

$$\begin{aligned} T(n) &= n^{\log_k a} \left(T(1) + \sum_{i=1}^{\log_k n} (k^d/a)^i \right) = n^{\log_k a} \left(T(1) + \sum_{i=1}^{\log_k n} 1 \right) \\ &= n^{\log_k k^d} (T(1) + \log_k n) = n^d (T(1) + \log_k n) = \Theta(n^d \log_k n) \end{aligned} \quad (3.3.14)$$

Time Complexity of Divide and Conquer Algorithms, III

- If $k^d \neq a$, then

$$T(n) = n^{\log_k a} \left(T(1) + \sum_{i=1}^{\log_k n} (k^d/a)^i \right) = n^{\log_k a} \left(T(1) + \frac{(k^d/a)^{1+\log_k n} - 1}{(k^d/a) - 1} \right)$$

- In case of $k^d > a$, or $d > \log_k a$, then

$$\begin{aligned} T(n) &= n^{\log_k a} \left(T(1) + \frac{(k^d/a)^{1+\log_k n} - 1}{(k^d/a) - 1} \right) = \Theta(n^{\log_k a} \frac{(k^d/a)^{1+\log_k n}}{(k^d/a)}) \\ &= \Theta(n^{\log_k a} (k^d/a)^{\log_k n}) = \Theta(a^{\log_k n} (k^d/a)^{\log_k n}) \\ &= \Theta(k^{d \log_k n}) = \Theta(k^{\log_k n^d}) = \Theta(n^d) \end{aligned} \quad (3.3.15)$$

- In case of $k^d < a$, or $d < \log_k a$, then

$$T(n) = n^{\log_k a} \left(T(1) + \frac{(k^d/a)^{1+\log_k n} - 1}{(k^d/a) - 1} \right) = n^{\log_k a} \cdot \Theta(1) = \Theta(n^{\log_k a}) \quad (3.3.16)$$

- Combine Equations (3.3.14), (3.3.15), and (3.3.16) together, we have

Theorem 3.3.8. Master Method

Let $T(n)$ be an eventually nondecreasing function that satisfies the recurrence relationship

$$\begin{aligned} T(n) &= a \cdot T(n/k) + f(n), & \text{for } n = k^i, i = 1, 2, \dots \\ T(1) &= c. \end{aligned}$$

where $a \geq 1$, $k \geq 2$, $c > 0$. If $f(n) = \Theta(n^d)$ where $d \geq 0$, then

$$T(n) = \begin{cases} \Theta(n^d), & \text{if } d > \log_k a, \\ \Theta(n^d \lg n), & \text{if } d = \log_k a, \\ \Theta(n^{\log_k a}), & \text{if } d < \log_k a. \end{cases}$$

Master Method – Example

- Example 1, Algorithm [MaxMin](#)

$$T(n) = 2T(n/2) + 2$$

$a = 2$, $k = 2$, $d = 0$, and $d < \log_2 2 = 1$.

Using Master Method, we have $T(n) = \Theta(n^{\log_2 2}) = \Theta(n)$.

- Example 2, Algorithm [MaxSubArray](#)

$$T(n) = 2T(n/2) + n$$

$a = 2$, $k = 2$, $d = 1$, and $d = \log_2 2$.

Using Master Method, we have $T(n) = \Theta(n \lg n)$.

- Example 3, [MatrixMultiplication](#)

$$T(n) = 8T(n/2) + n^2$$

$a = 8$, $k = 2$, $d = 2$, and $d < \log_2 8 = 3$.

Using Master Method, we have $T(n) = \Theta(n^{\log_2 8}) = \Theta(n^3)$.

Master Method – Example, II

- Example 4, Strassen's matrix multiplication

$$T(n) = 7 \cdot T(n/2) + c \cdot n^2$$

$a = 7, k = 2, d = 2$ and $d < \log_2 7$.

Using Master Method, we have $T(n) = \Theta(n^{\lg 7})$.

- Example 5,

$$T(n) = 2T(n/2) + n^2$$

$a = 2, k = 2, d = 2$ and $d > \log_2 2 = 1$.

Using Master Method, we have $T(n) = \Theta(n^d) = \Theta(n^2)$.

This can also be derived as follows.

$$\begin{aligned} T(n) &= 2T(n/2) + n^2 = 2\left(2T(n/4) + (n/2)^2\right) + n^2 \\ &= 4T(n/4) + n^2(1 + 1/2) = 2^m T(n/2^m) + n^2 \sum_{i=0}^{m-1} 1/2^i \\ &= n + n^2 \cdot 2 \cdot (1 - 2^{-m}) = \Theta(n^2) \end{aligned}$$

- Thus, the Master method can be effective to find the complexity of divide and conquer algorithms.

Summary

- Selection problem
 - Worst-case: $\mathcal{O}(n^2)$, average-case: $\mathcal{O}(n)$
- Matrix multiplication, $\mathcal{O}(n^3)$
 - Strassen's matrix multiplication, $\mathcal{O}(n^{\lg 7})$
- Convex hull problem ($\mathcal{O}(n^4)$)
 - Graham's scan algorithm, $\mathcal{O}(n \lg n)$
 - Quick hull algorithm, worst-case $\mathcal{O}(m^2)$, average-case $\mathcal{O}(m \lg m)$
- Time complexity of divide and conquer algorithms
 - Master method