# Master Theorem

If we have a recurrence of this form:

$T(n) = aT(n/b) + f(n)$ and $f(n) \in \Theta\ (n^2)$ then:

- $T(n) \in \Theta(n^d)$ if $a < b^d$
- $T(n) \in \Theta(n^d \log n)$ if $a = b^d$
- $T(n) \in \Theta(n^{\log_b a})$ if $a > b^d$

Analogous results for O and Ω

- For this course we have to know how to apply the Master Theorem, not prove it or derive it

## Example

```
int CountBits(int n):
    if(n==1):
        return 1
    else:
        return 1 + CountBits(n/2)
```

Recall that this is how we found the algorithmic efficiency before hand:

- $A(1) = 0$ (Addition doesn't take place when n == 1)
- $A(n) = A(n/2) + 1$ for $n > 1$
- by letting $n = 2^2$ which is the same as saying $k = \log_2 n$
- $n/2 = 1/2 * 2^2 = 2^{-1} * 2^k = 2^{2-1}$
- $A(1) = A(2^0) = 0$
- $A(n) = A(2^2) = A(2^{k-1}) + 1$ for $k > 0$
- $= [A(2^{k-2}) + 1] + 1 = A(2^{k-2}) + 2$
- $= A(2^{k-k}) + k = A(2^0) + k = k = \log_2 n \in \Theta\ (\log n)$

But with the master theorem we can see that

- a = 1
- b = 2
- d = 0

Because if we put CountBits in the form of the master theorem i.e.: $T(n) = aT(n/b) + f(n)$ it will look as follows:

- $A(n) = A(n/2) + 1$

Thus the values for a, b and d are as outlined above and by subbing them into the equation: $a = b^d$ then, with the master theorem we get the following:

- $1 = 2^0$
- Which is equal

And if we look at the master theorem cases again:

- $T(n) \in \Theta(n^d)$ if $a < b^d$
- $T(n) \in \Theta(n^d \log n)$ if **$\underline{a = b^d}$**
- $T(n) \in \Theta(n^{\log_b a})$ if $a > b^d$

According to the second case, the recursive function will have the efficiency of $T(n) \in \Theta(n^d \log n)$. Thus, CountBits has the algorithmic efficiency of $\Theta(n^0 \log n)$ i.e.: CountBits is **$\Theta(\log n)$**

But what do a, b and d mean?

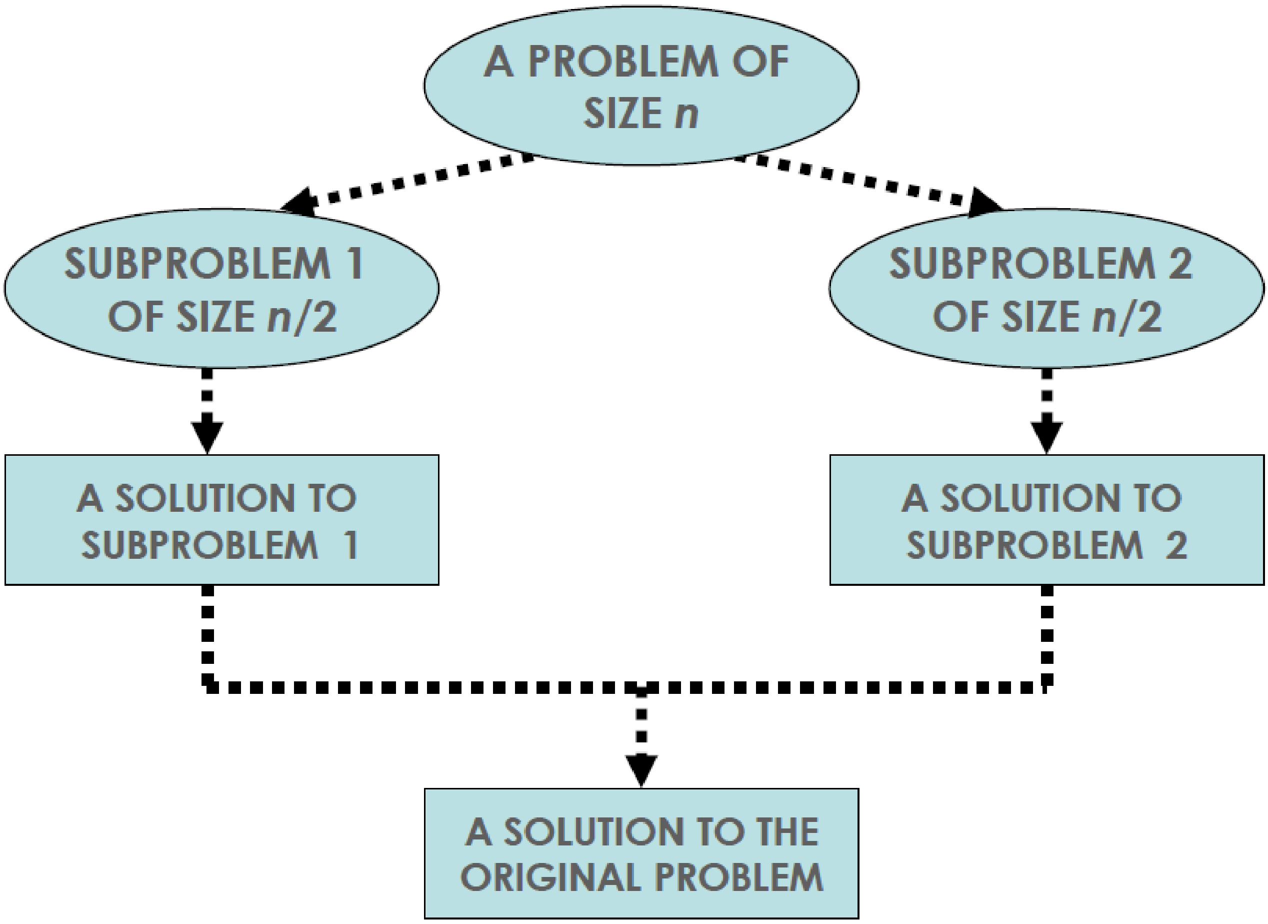
- a is the number of recursive calls
- b is the fraction of the input the recursive call works on
- d is the other work done

# Divide & Conquer

- Divide & Conquer is the best known algorithm design strategy:
    1. Divide instances of problem into two or more smaller instances
    2. Solve smaller instances recursively
    3. Obtain solution to original (larger) instance by combining these solutions

Here is an example of finding a recursive solution:

- What would you do if you had the smallest (non-trivial) number of input values? (say n)
- If you did the above for 2 separate inputs of that size, could you use those 2 results to give you the solution to the problem for all 2n input values?
- Try it by hand and see. Maybe for n, then 2n, and then 4n. It it works for those it will (probably) work for any n
- Code and test with input size n, then 2n, then 4n




## Matrix Addition

$$\begin{bmatrix} A & B \\ C & D \end{bmatrix} + \begin{bmatrix} E & F \\ G & H \end{bmatrix} = \begin{bmatrix} A+E & B+F \\ C+G & D+H \end{bmatrix}$$

Which we can represent algorithmically as:

```
for(int row = 0; row<n; row++){
    for(int col = 0; col < n; col++){
        C[row][col] = A[row][col] + B[row][col]
    }
}
```

Add matrix A to matrix B to get matrix C - $\Theta(n^2)$

## Matrix Multiplication

### Brute Force

$$\begin{bmatrix} A & B \\ C & D \end{bmatrix} * \begin{bmatrix} E & F \\ G & H \end{bmatrix} = \begin{bmatrix} AE+BG & AF+BH \\ CE+DG & CF+DH \end{bmatrix}$$

Algorithmically:

```
for(int row = 0; row<n; row++){
```

```
for(int row = 0; row<n; row++){
    for(int j = 0; j< n;j++){
        for(int col = 0; col < n; col++){
            C[row][col] += A[row][j] + B[j][col]
        }
    }
}
```

There are n multiplication to calculate each of the $n^2$ values - $\Theta(n^3)$

How many multiplications: $\sum_{i=0}^{n-1}\sum_{k=0}^{n-1}\sum_{j=0}^{n-1}$

Applying the rules we learned, we get:

$$\sum_{i=0}^{n-1}\sum_{k=0}^{n-1} n = \sum_{i=0}^{n-1} n^2 = n^3 \epsilon \Theta(n^3)$$

## Divide & Conquer

Below is a complicated recursive matrix multiplication method - only a high level understanding is needed.

So the same process as above is followed:

$$A = \begin{bmatrix} A_{1,1} & A_{1,2} \\ A_{2,1} & A_{2,2} \end{bmatrix} \qquad B = \begin{bmatrix} B_{1,1} & B_{1,2} \\ B_{2,1} & B_{2,2} \end{bmatrix}$$

$$A * B = C = \begin{bmatrix} C_{1,1} & C_{1,2} \\ C_{2,1} & C_{2,2} \end{bmatrix}$$

$$C_{1,1} = A_{1,1} * B_{1,1} + A_{1,2} * B_{2,1} \qquad C_{2,1} = A_{2,1} * B_{1,1} + A_{2,2} * B_{2,1}$$

$$C_{1,2} = A_{1,1} * B_{1,2} + A_{1,2} * B_{2,2} \qquad C_{2,2} = A_{2,1} * B_{1,2} + A_{2,2} * B_{2,2}$$

- Imagine that A, B & C are not 2 x 2 matrices, but n X n matrices.
- Then the $A_{1,1}$, $A_{1,2}$...$B_{i,j}$ (i.e.: all the elements of the matrices) are n/2 x n/2 matrices.
- With matrices, you can treat n/2 x n/2 matrices atomically
- We can recursively calculate each of the $C_{i,j}$ matrices
- The base can would be when n is 2, or you can make it when n is 1

There is a good walkthrough of this idea in the slides. Which you can view here - but I will simply lay out the algorithm:

The following pseudocode multiplies 2 nXn matrices. If n is not a power of 2, pad the matrices with Os to make n power of 2.

$$
\begin{aligned}
&\text{MM(A, B), where A and B are both } n \times n \text{ matrices} \\
&\quad \text{If } n == 1\text{:} \\
&\quad\quad \text{output } A_{1,1} * B_{1,1} \\
&\quad \text{Else:} \\
&\quad\quad \text{Compute MM } (A_{1,1}, B_{1,1}) + \text{MM}(A_{1,2}, B_{2,1}) \\
&\quad\quad \text{Compute MM } (A_{1,1}, B_{1,2}) + \text{MM } (A_{1,2}, B_{2,2}) \\
&\quad\quad \text{Compute MM } (A_{2,1}, B_{1,1}) + \text{MM } (A_{2,2}, B_{2,1}) \\
&\quad\quad \text{Compute MM } (A_{2,1}, B_{1,2}) + \text{MM } (A_{2,2}, B_{2,2})
\end{aligned}
$$

**Recurrence relation for D & C Matric Multiplication**

- T(1) = 1 - one multiplication, zero additions
- For n > 1:
  - We make 8 recursive calls for multiplying n/2 x n/2 matrices
  - T(n) = 8 * T(n/2) + number of additions
    - Matrix addition is an $n^2$ operation
    - In each of the 4 compute steps we add two n/2 x n/2 matrices
    - Therefore there are $4(n/2)^2$ additions:
      - $\Theta(n^2)$

Thus the recurrence relation is $T(n) = 8 * T(n/2) + \Theta(n^2)$

Now if we substitute those values into the masters theorem:

- a = 8
- b = 2
- d = 2 and $8 > 2^2$

So $T(n) \in \Theta(n^{\log_2 8})$ and therefore:

$\Theta(n^3)$

So the Divide and Conquer method has exactly the same efficiency class as Brute Force

## Strassen's Method

- For 2x2 matrices, the standard method using the definition makes 8 multiplications and 4 additions

$$\begin{bmatrix} A & B \\ C & D \end{bmatrix} * \begin{bmatrix} E & F \\ G & H \end{bmatrix} = \begin{bmatrix} AE+BG & AF+BH \\ CE+DG & CF+DH \end{bmatrix}$$

- But we can change this to 7 multiplications and 18 additions/subtractions using Strassen's method
- By simply reducing the number of multiplications by 1, we can increase the efficiency of matrix multiplication
- It is the multiplications that effect how many recursive calls are made

- The extra additions are just a constant factor

Here is Strassen's method for multiplying a 2x2 matrix:

$$\begin{bmatrix} a_{00} & a_{01} \\ a_{10} & a_{11} \end{bmatrix} \times \begin{bmatrix} b_{00} & b_{01} \\ b_{10} & b_{11} \end{bmatrix} = \begin{bmatrix} c_{00} & c_{01} \\ c_{10} & c_{11} \end{bmatrix}$$

$$= \begin{bmatrix} m_1 + m_4 - m_5 + m_7 & m_3 + m_5 \\ m_2 + m_4 & m_1 + m_3 - m_2 + m_6 \end{bmatrix}$$

$$\begin{aligned} m_1 &= (a_{00} + a_{11}) * (b_{00} + b_{11}) \\ m_2 &= (a_{10} + a_{11}) * b_{00} \\ m_3 &= a_{00} * (b_{01} - b_{11}) \\ m_4 &= a_{11} * (b_{10} - b_{00}) \\ m_5 &= (a_{00} + a_{01}) * b_{11} \\ m_6 &= (a_{10} - a_{00}) * (b_{00} + b_{01}) \\ m_7 &= (a_{01} - a_{11}) * (b_{10} + b_{11}) \end{aligned}$$

To put this in an algorithm, we first have to layout some guidelines:

- A and B are n X n matrices. n is power of 2
- If n is not power of 2, pad A and B with Os
- Divide A and B into n/2 x n/2 matrices & recurse:
- By following the same structure as above, this should give us a recurrence relation of:
    - $T(n) = 7\,T(n/2) + \Theta(n^2)$

The algorithm is as follows:

```
STRASSEN(A, B):
    1. If n == 1  Output  A11 × B11
    2. Else
    3.          Split matrices into 8 n/2 X n/2 parts: A11 , B11 , . . . , A22 , B22
    4.          P1 = Strassen(A11, B12 − B22 )
    5.          P2 = Strassen(A11 + A12 , B22 )
    6.          P3 = Strassen(A21 + A22 , B11 )
    7.          P4 = Strassen(A22 , B 21 − B11 )
    8.          P5 = Strassen(A11 + A22 , B11 + B22 )
    9.          P6 = Strassen(A12 − A22 , B21 + B22 )
    10          P7 = Strassen(A11 − A21 , B11 + B12 )
    11.         C 11 = P5 + P4 − P2 + P6
    12.         C 12 = P1 + P2
    13.         C 21 = P3 + P4
    14.         C 22 = P1 + P5 − P3 − P7
    15.         Output C
```

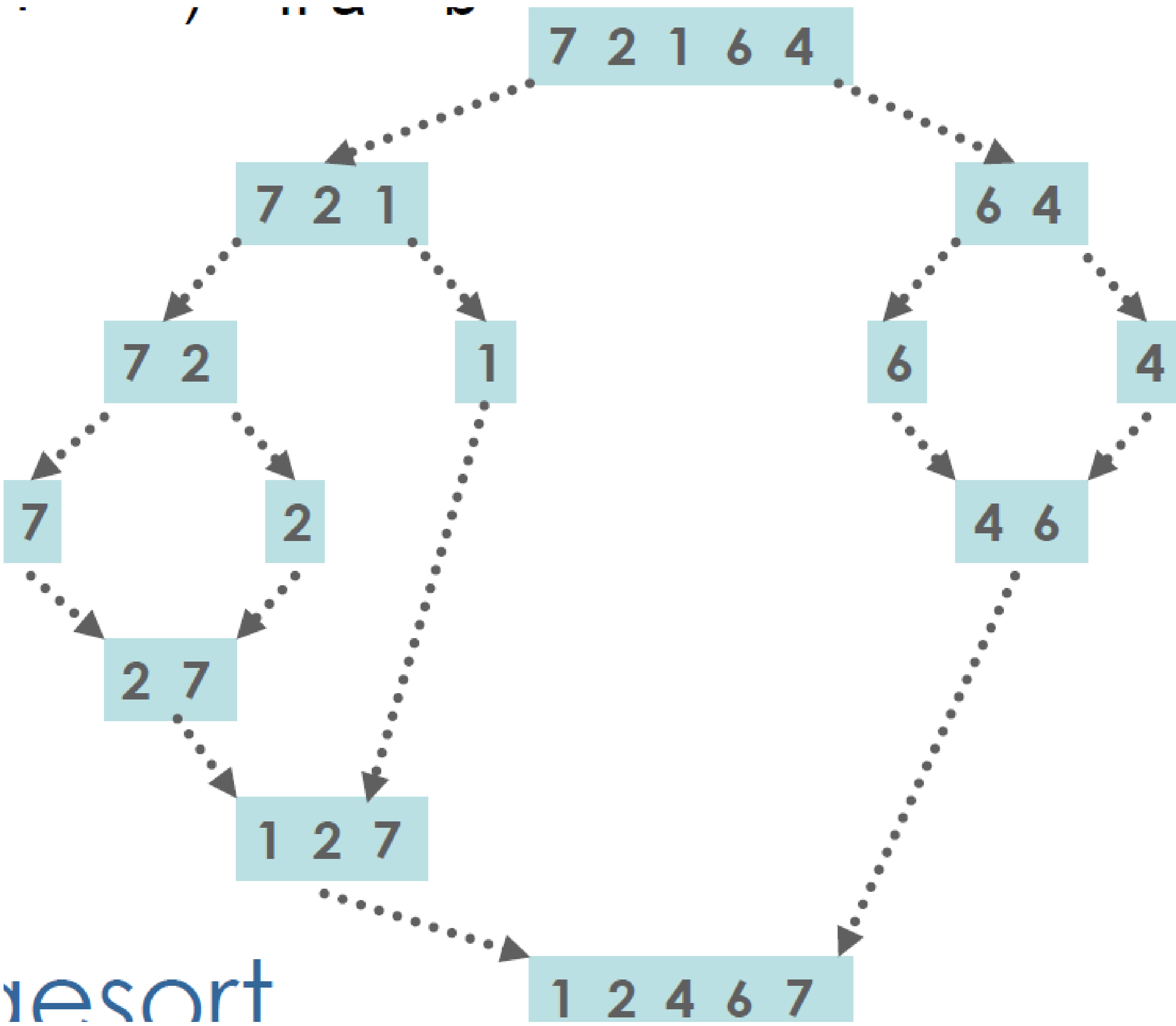
We can solve the relation using the Master Theorem to find out the algorithmic efficiency:

- $T(n) = 7 T(n/2) + \Theta(n^2)$
  - a = 7
  - b = 2
  - d = 2
- $7 > 2^2$
- $A(n) \in \Theta(n^{\log_2 7})$
- Which means: $T(n) \in \Theta(n^{2.807})$
- Which is less than $n^3$
- This solution was developed in 1969
- No one thought it possible to improve $\Theta(n^3)$
- Improvements continue:
  - Virginia Williams created a new algorithm in 2011 which has a complexity $\Theta(n^{2.4})$
  - You can read her paper on it here

## Mergesort

- Algorithm
  1. Split A[1..n] in half and put copy of each half into arrays B[1.. n/2] and C[1.. n/2]
  2. Recursively Mergesort arrays B and C
  3. Merge sorted arrays B and C into array A
- Merging
  - Repeat until no elements remain in one of B or C
  1. Compare 1st elements in the rest of B and C

2. Copy smaller into A, incrementing index of corresponding array
3. Once all elements in one of B or C are processed, copy the remaining unprocessed elements from the other array into A



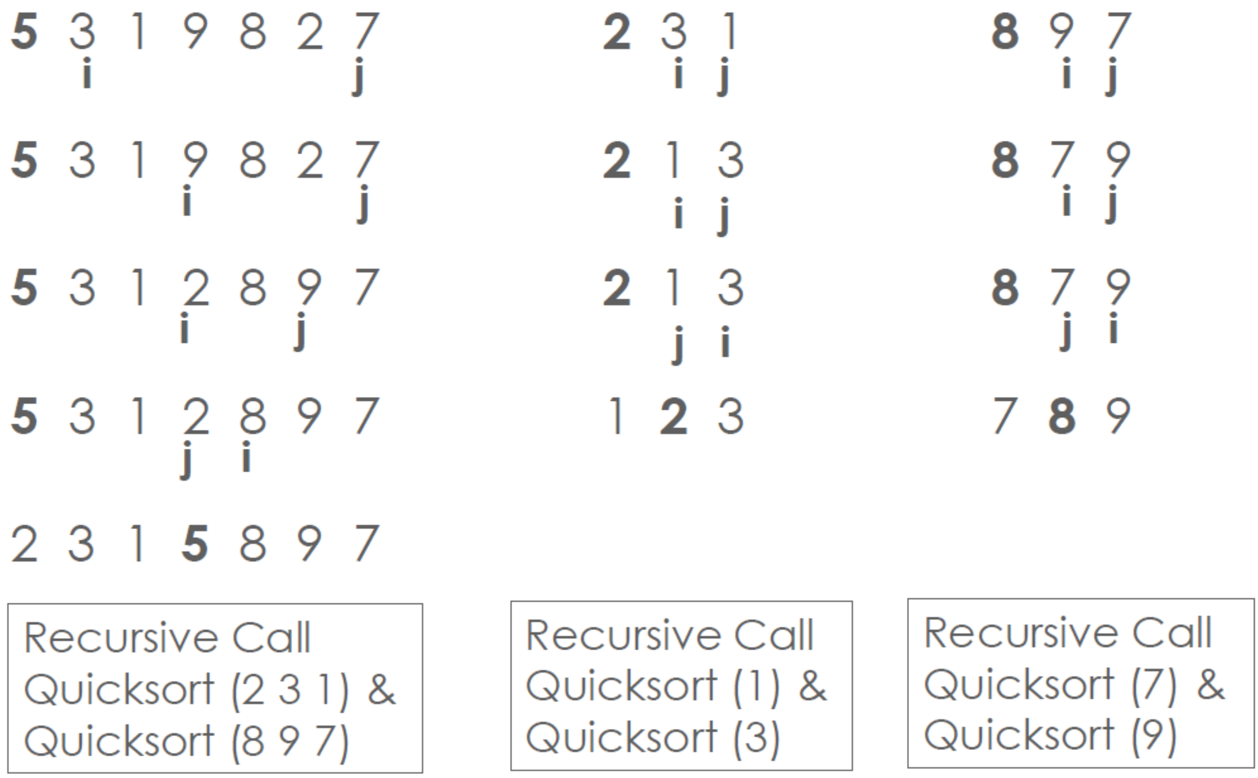

**Efficiency**

- Recurrence:
  - $C(n) = 2C(n/2) + C_{merge}(n)$ for $n > 1$, $C(1) = 0$
  - $C_{merge}(n) = n-1$ in the worst case
- All cases have same efficiency: Θ(nlogn)
- Number of comparisons is close to theoretical minimum for comparison-based sorting:
  - $\log n! \approx n\log n - 1{,}44n$
  - pace requirement Θ(n) (NOT in-place)
  - Can be implemented without recursion (bottom-up)

## Quicksort

- Select a pivot (partitioning element)
- Rearrange the list into two sublists:
  - All elements positioned before the pivot are <= the pivot
  - Those positioned after the pivot are > the pivot

    - Requires a pivoting algorithm
- Exchange the pivot with the last element in the first sublist
    - The pivot is now in its final position
- QuickSort the two sublists




**Partitioning Algorithm**

```
Algorithm Partition(A[l..r])
//Partitions a subarray by using its first element as a pivot
//Input: A subarray A[l..r] of A[0..n − 1], defined by its left and right
//        indices l and r (l < r)
//Output: A partition of A[l..r], with the split position returned as
//         this function's value
p ← A[l]
i ← l;   j ← r + 1
repeat
    repeat i ← i + 1 until A[i] ≥ p
    repeat j ← j − 1 until A[j] ·  p
    swap(A[i], A[j])
until i ≥ j
swap(A[i], A[j])   //undo last swap when i ≥ j
swap(A[l], A[j])
return j
```

**Efficiency**

- In the worst case all splits are completely skewed

- For instance, an already sorted list!
- One subarray is empty, other reduced by only one:
    - Make n+1 comparisons
    - Exchange pivot with itself
    - Quicksort left = empty, right = A[1..n-1]
    - $C_{worst} = (n+1) + n + ... + 3 = (n+1)(n+2)/2 - 3 = \Theta(n^2)$
- While the worst case is $\Theta(n^2)$, best case (split in the middle) is $\Theta(n\log n)$ and average case (random split) is $\Theta(n\log n)$
- Improvements (in combination 20-25% faster):
    - Better pivot selection: median of three partitioning avoids worst case in sorted files
    - Switch to insertion sort on small subfiles
    - Elimination of recursion
- Considered the method of choice for sorting for large files (n >= 10 000)

## Quicksort & Mergesort

```
Quicksort(L,R)
{if L<R
    {    p=Partition(L,R);
         Quicksort(L, p-1);
         Quicksort(p+1,R);
    }
}
```

```
Mergesort(A, n)
{if n>1
    {    CopyArr(A,L,0,n/2 -1);
         CopyArr(A,R,n/2,n);
         Mergesort(L, n/2);
         Mergesort(R, n/2);
         Merge(L,R,A);
    }
}
```

## Closest Pair

The slides have a really great walkthrough of the divide & conquer algorithm for Closest Pair, I am going to simply add the pseudo-code and efficiency here

**Algorithm Overview**

1. Sort points according to their x-coordinates
2. Split the set of points into two equal-sized subsets by a vertical line $x = x_{median}$
3. Solve the problem recursively in the left and right subsets, so that we get the left-side and right-side minimum distances $d_l$ and $d_r$. Therefore, $d_{min} = \min(d_l,d_r)$
4. Find the minimal distance in the set S of points of width 2d around the vertical line. Update $d_{min}$ if necessary

**Algorithm Pseudo-code**

```
Sort points in order of x co-ordinates into array P[0 ... n-1]
Sort P in y co-ordinate order into array Q[0 ... n-1]
```

```
double ClosestPair(P,Q):
    if n <= 3:
        return minDistance;
    copy the first ceiling [n/2] points of P to PL (L for left)
    copy the same points of Q to QL
    copy the remaining floor[n/2] points of P to PR
    copy the same points of Q to QR
    dl = ClosestPair(PL,QL)
    dr = ClosestPair(PR,QR)
    d = min(dl,dr)
    m = P[ceiling(n/2)-1].x
    copy all points of Q for which |x-m| < d into S[0 ... num - 1]
    dminsq = d * d
    for i = 0 to num -2:
        k = i + 1
        while k <= num - 1 and distance(S[K],S[i]) < dminsq
            dminsq = distance(S[K],S[i])
            k = k + 1
    return sqrt(dminsq)
```

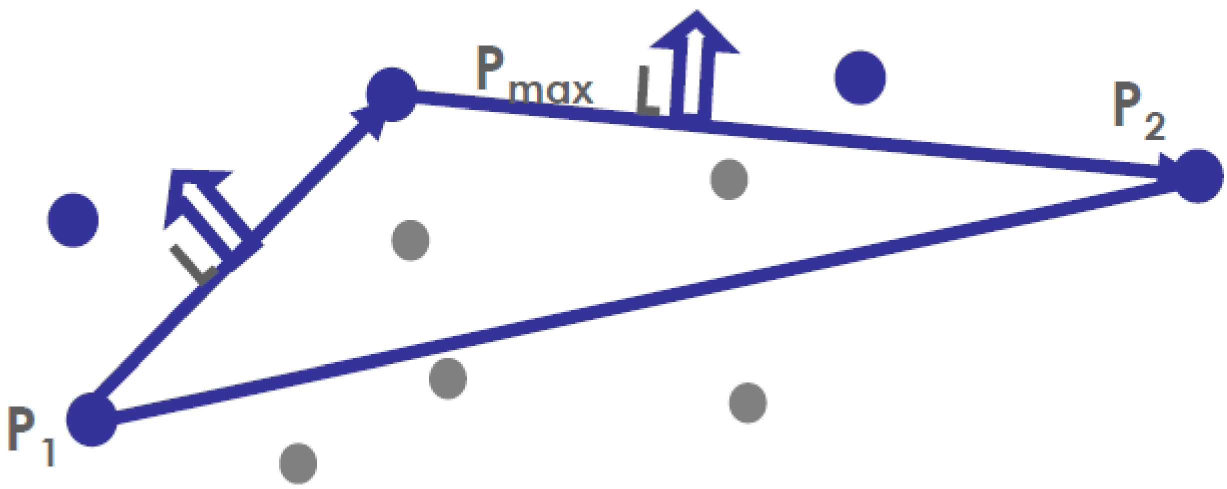
### Algorithm Efficiency

- Pre-sorting is $\Theta(n\log n)$
- Leaving aside the recursion every other step in the algorithm is at worst $\Theta(n)$
- So we can set up a recurrence relation in terms of a function T of whatever basic operation you choose.
- Recursion is executed twice. So we have:
- $T(n) = 2T(n/2) + f(n)$
- Now we have seen that f(n) is in $\Theta(n)$ because there are at most 6 points to consider each time.
- Therefore, according to the master theorem:
    - a = 2
    - b = 2
    - d = 1
- So $a = b^d$
    - $T(n) \in \Theta(n^d \log n)$
    - $T(n) \in \Theta(n\log n)$
- Main algorithm is the same efficiency class as the pre-sorting part

## Convex Hull Problem

- Remember this from the last set of notes
- There is a divide & conquer solution:

1. Sort points by increasing x-coordinate values
2. Identify leftmost and rightmost extreme points $P_1$ and $P_2$ (part of the hull)
3. Compute upper hull
    - Find point $P_{max}$ that is farthest away from line $P_1P_2$
    - Quickhull the points to the left of line $P_1P_{max}$
    - Quickhull the points to the left of the line $P_{max}P_2$
4. Similarly compute lower hull




**Finding the furthest point**

Given three points in the plane $p_1$ , $p_2$ , $p_3$

Area of Triangle = $\Delta\, p_1\, p_2\, p_3 = 1/2 \;\boxtimes D \boxtimes$

$$D = \begin{bmatrix} x_1 & y_1 & 1 \\ x_2 & y_2 & 1 \\ x_3 & y_3 & 1 \end{bmatrix}$$

$$\boxtimes D \boxtimes = x_1\, y_2 + x_3\, y_1 + x_2\, y_3 - x_3\, y_2 - x_2\, y_1 - x_1\, y_3$$

Properties of $\boxtimes D \boxtimes$ :

- Positive *iff* $p_3$ is to the left of $p_1 p_2$
- Correlates with distance of $p_3$ from $p_1 p_2$

**Efficiency**

- Finding points to the left and their distance from line $P_1P_2$ is linear in the number of points
- Efficiency
  - Worst case $\Theta(n^2)$
  - Average case: $\Theta(n\log n)$
- Alternative Divide-and-Conquer Convex Hull
  - Graham's Scan and DCHull
  - Also $\Theta(n\log n)$ but with lower coefficients