Divide and Conquer

General method

Divide Split the input with n sample points into k subsets, $1 < k \leq n$, with no overlap between subsets

Conquer Solve each of the k subproblems, possibly by splitting recursively

- If the subproblems are small enough, solve those using a simple (non-recursive) method
- Also known as the *bottoming out* of recursion, or arrival at a *base case*

Combine the result of the k subproblems to solve the original problem

- Detecting a counterfeit coin
  - Bag of 16 coins with one of them possibly counterfeit
  - Counterfeit coins are lighter than genuine ones
  - Determine if the bag contains a counterfeit coin
  - Algorithm 1
    - Compare a pair of coins; if one of them is counterfeit, it will be lighter
    - Compare next pair, and so on
    - You will find the counterfeit coin in at most 8 trials
  - Algorithm 2: Divide and conquer
    - Divide the 16 coin instance into two instances of 8 coins each
    - Compare the weight of two sets; if same, no counterfeit coin; else divide the lighter instance into two of half the original size and repeat
    - Presence of counterfeit coin is determined in 1 trial
    - Identification of coin happens in four trials
- Natural to express the solution as a recursive algorithm
- Control abstraction for recursive divide and conquer, with problem instance `P`

```
divide_and_conquer ( P )
{
    if ( small ( P ) )          // P is very small so that a solution is trivial
        return solution ( n );
    divide the problem P into k instances P1, P2, ..., Pk;
    return ( combine ( divide_and_conquer ( P1 ),
                       divide_and_conquer ( P2 ),
                       ...
                       divide_and_conquer ( Pk ) ) );
}
```

- The solution to the above problem is described by the recurrence, assuming size of `P` denoted by n

$$T_n = \begin{cases} g(n) & n \text{ is small} \\ T_{n_1} + T_{n_2} + \cdots + T_{n_k} + f(n) \end{cases}$$

  where $f(n)$ is the time to divide n elements and to combine their solution

- Recursively decompose a large problem into a set of smaller problems
  - Decomposition is directly reflected in analysis

  - Run-time determined by the size and number of subproblems to be solved in addition to the time required for decomposition
  - General complexity computation

$$T_n = \begin{cases} T_1 & n = 1 \\ aT_{n/b} + f(n) & n > 1 \end{cases}$$

    where a and b are known constants; assume that $n = b^k$

- Example, mergesort recurrence

$$T_n = \begin{cases} \Theta(1) & \text{if } n = 1 \\ 2T_{\frac{n}{2}} + \Theta(n) & \text{if } n > 1 \end{cases}$$

  Solution for the mergesort recurrence: $\Theta(n \lg n)$

- You can ignore extreme details like floor, ceiling, and boundary in recurrence description.
- Solving recurrence relations by substitution method
  - Guess the form of solution and use mathematical induction to find constants
  - Determine upper bound on the recurrence

$$T_n = 2T_{\lfloor \frac{n}{2} \rfloor} + n$$

    Guess the solution as: $T_n = O(n \lg n)$
    Now, prove that $T_n \leq cn \lg n$ for some $c > 0$
    Assume that the bound holds for $\lfloor \frac{n}{2} \rfloor$
    Substituting into the recurrence

$$\begin{aligned} T_n &\leq 2(c \left\lfloor \frac{n}{2} \right\rfloor \lg(\left\lfloor \frac{n}{2} \right\rfloor)) + n \\ &\leq cn \lg \left(\frac{n}{2}\right) + n \\ &= cn \lg n - cn \lg 2 + n \\ &= cn \lg n - cn + n \\ &\leq cn \lg n \quad \forall c \geq 1 \end{aligned}$$

    Boundary condition: Let the only bound be $T_1 = 1$

$$\nexists c \mid T_1 \leq c1 \lg 1 = 0$$

    Problem overcome by the fact that asymptotic notation requires us to prove

$$T_n \leq cn \lg n \text{ for } n \geq n_0$$

    Include T_2 and T_3 as boundary conditions for the proof

$$T_2 = 4 \quad T_3 = 5$$

    Choose c such that $T_2 \leq c2 \lg 2$ and $T_3 \leq c3 \lg 3$
    True for any $c \geq 2$
  - Making a good guess
    * If a recurrence is similar to a known recurrence, it is reasonable to guess a similar solution

$$T_n = 2T_{\lfloor \frac{n}{2} \rfloor} + n$$

      If n is large, difference between $T_{\lfloor \frac{n}{2} \rfloor}$ and $T_{\lfloor \frac{n}{2} \rfloor + 17}$ is relatively small
    * Prove upper and lower bounds on a recurrence and reduce the range of uncertainty
      Start with a lower bound of $T_n = \Omega(n)$ and an initial upper bound of $T_n = O(n^2)$. Gradually lower the upper bound and raise the lower bound to get asymptotically tight solution of $T_n = \Theta(n \lg n)$

- Pitfall
  - $T_n = 2T_{\lfloor \frac{n}{2} \rfloor} + n$
    Assume inductively that $T_n \leq cn$ implying that $T_n = O(n)$

$$\begin{aligned} T_n &\leq 2c\left\lfloor \frac{n}{2} \right\rfloor + n \\ &\leq cn + n \\ &= O(n) \quad \Leftarrow \text{ wrong} \end{aligned}$$

    We haven't proved the exact form of inductive hypothesis $T_n \leq cn$
- Changing variables
  - Consider the recurrence

$$T_n = 2T_{\lfloor \sqrt{n} \rfloor} + \lg n$$

    Let $m = \lg n$.

$$T_{2^m} = 2T_{2^{\frac{m}{2}}} + m$$

    Rename $S_m = T_{2^m}$

$$S_m = 2S_{\frac{m}{2}} + m$$

    Solution for the recurrence: $S_m = m \lg m$
    Change back from S_m to T_n

$$T_n = T_{2^m} = S_m = O(m \lg m) = O(\lg n \lg \lg n)$$

- Solving recurrence relations by iteration method
  - No guessing but more algebra
  - Expand the recurrence and express it as summation dependent on only n and initial conditions
  - Recurrence

$$T_n = 3T_{\lfloor \frac{n}{4} \rfloor} + n$$

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

    ith term is given by $3^i \lfloor \frac{n}{4^i} \rfloor$
    Bound $n = 1$ when $\lfloor \frac{n}{4^i} \rfloor = 1$ or $i > \log_4 n$
    Bound $\lfloor \frac{n}{4^i} \rfloor \leq \frac{n}{4^i}$
    Decreasing geometric series

$$\begin{aligned} T_n &\leq n + \frac{3}{4}n + \frac{9}{16}n + \frac{27}{64}n + \cdots + 3^{\log_4 n}\Theta(1) \\ &\leq n\sum_{i=0}^{\infty}\left(\frac{3}{4}\right)^i + \Theta(n^{log_4 3}) \quad 3^{\log_4 n} = n^{log_4 3} \\ &= 4n + o(n) \quad log_4 3 < 1 \Rightarrow \Theta(n^{log_4 3} = o(n) \\ &= O(n) \end{aligned}$$

    Focus on
    - Number of iterations to reach boundary condition

  - Sum of terms arising from each level of iteration
- Recursion trees
  - Recurrence

$$T_n = 2T_{\frac{n}{2}} + n^2$$

Assume n to be an exact power of 2.

$$\begin{aligned} T_n &= n^2 + 2T_{\frac{n}{2}} \\ &= n^2 + 2\left(\left(\frac{n}{2}\right)^2 + 2T_{\frac{n}{4}}\right) \\ &= n^2 + \frac{n^2}{2} + 4\left(\left(\frac{n}{4}\right)^2 + 2T_{\frac{n}{8}}\right) \\ &= n^2 + \frac{n^2}{2} + \frac{n^2}{4} + 8\left(\left(\frac{n}{8}\right)^2 + 2T_{\frac{n}{16}}\right) \\ &= n^2 + \frac{n^2}{2} + \frac{n^2}{4} + \frac{n^2}{8} + \cdots \\ &= n^2(1 + \frac{1}{2} + \frac{1}{4} + \frac{1}{8} + \cdots) \\ &= \Theta(n^2) \end{aligned}$$

The values above decrease geometrically by a constant factor.

  - Recurrence

$$T_n = T_{\frac{n}{3}} + T_{\frac{2n}{3}} + n$$

Longest path from root to a leaf

$$n \to \left(\frac{2}{3}\right) n \to \left(\frac{2}{3}\right)^2 n \to \cdots 1$$

$\left(\frac{2}{3}\right)^k n = 1$ when $k = \log_{\frac{3}{2}} n$, k being the height of the tree
Upper bound to the solution to the recurrence – $n \log_{\frac{3}{2}} n$, or $O(n \log n)$

- The Master Method
  - Suitable for recurrences of the form

$$T_n = aT_{\frac{n}{b}} + f(n)$$

where $a \geq 1$ and $b > 1$ are constants, and
$f(n)$ is an asymptotically positive function
  - For mergesort, $a = 2$, $b = 2$, and $f(n) = \Theta(n)$
  - Master Theorem

Theorem 1 *Let $a \geq 1$ and $b > 1$ be constants, let $f(n)$ be a function, and let T_n be defined on the nonnegative integers by the recurrence*

$$T_n = aT_{\frac{n}{b}} + f(n)$$

where we interpret $\frac{n}{b}$ to mean either $\left\lfloor \frac{n}{b} \right\rfloor$ or $\left\lceil \frac{n}{b} \right\rceil$. Then T_n can be bounded asymptotically as follows

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 $af\left(\frac{n}{b}\right) \leq cf(n)$ for some constant $c < 1$ and all sufficiently large n, then $T_n = \Theta(f(n))$*

    - In all three cases, compare $f(n)$ with $n^{\log_b a}$
    - Solution determined by the larger of the two
      - Case 1: $n^{\log_b a} > f(n)$
        Solution $T_n = \Theta(n^{\log_b a})$

  - Case 2: $n^{\log_b a} \approx f(n)$
    Multiply by a logarithmic factor
    Solution $T_n = \Theta(n^{\log_b a} \lg n) = \Theta(f(n) \lg n)$
  - Case 3: $f(n) > n^{\log_b a}$
    Solution $T_n = \Theta(f(n))$
* In case 1, $f(n)$ must be asymptotically smaller than $n^{\log_b a}$ by a factor of n^ϵ for some constant $\epsilon > 0$
* In case 3, f_n must be polynomially larger than $n^{\log_b a}$ and satisfy the "regularity" condition that $af(\frac{n}{b}) \leq cf(n)$

– Using the master method

* Recurrence
$$T_n = 9T_{\frac{n}{3}} + n$$
$a = 9, b = 3, f(n) = n$
$n^{\log_b a} = n^{\log_3 9} = \Theta(n^2)$
$f(n) = O(n^{\log_3 9 - \epsilon})$, where $\epsilon = 1$
Apply case 1 of master theorem and conclude $T_n = \Theta(n^2)$
* Recurrence
$$T_n = T_{\frac{2n}{3}} + 1$$
$a = 1, b = \frac{3}{2}, f(n) = 1$
$n^{\log_b a} = n^{\log_{\frac{3}{2}} 1} = n^0 = 1$
$f(n) = \Theta(n^{\log_b a}) = \Theta(1)$
Apply case 2 of master theorem and conclude $T_n = \Theta(\lg n)$
* Recurrence
$$T_n = 3T_{\frac{n}{4}} + n \lg n$$
$a = 3, b = 4, f(n) = n \lg n$
$n^{\log_b a} = n^{\log_4 3} = O(n^{0.793})$
$f(n) = \Omega(n^{\log_4 3 + \epsilon})$, where $\epsilon \approx 0.2$
Apply case 3, if regularity condition holds for $f(n)$
For large n, $af(\frac{n}{b}) = 3\frac{n}{4} \lg(\frac{n}{4}) \leq \frac{3}{4} n \lg n = cf(n)$ for $c = \frac{3}{4}$
Therefore, $T_n = \Theta(n \lg n)$
* Recurrence
$$T_n = 2T_{\frac{n}{2}} + n \lg n$$
Recurrence has proper form – $a = 2, b = 2, f(n) = n \lg n$ and $n^{\log_b a} = n$
$f(n) = n \lg n$ is asymptotically larger than $n^{\log_b} = n$ but not *polynomially* larger
Ratio $\frac{f(n)}{n^{log_b a}} = \frac{n \lg n}{n} = \lg n$ is asymptotically less than n^ϵ for any positive constant ϵ
Recurrence falls between case 2 and case 3

- Technicalities in recurrences

– Simplify the solution by ignoring certain details

* Consider mergesort; if n is odd, we end up with subproblems of size $\lfloor n/2 \rfloor$ and $\lceil n/2 \rceil$
* Technically correct definition of mergesort recurrence is
$$T_n = \begin{cases} \Theta(1) & \text{if } n = 1 \\ T_{\lfloor \frac{n}{2} \rfloor} + T_{\lceil \frac{n}{2} \rceil} + \Theta(n) & \text{if } n > 1 \end{cases}$$

– We may also choose to ignore boundary conditions

* Running time of a constant-size input is a constant
* For sufficiently small n, we can describe the run-time as
$$T_n = \Theta(1)$$
* As a convenience, we may omit boundary conditions from recurrence

  - We can describe the recurrence for mergesort as

$$T_n = 2T_{\frac{n}{2}} + \Theta(n)$$

  - Omitting the value of T_1 changes the exact solution for the recurrence, but not by more than a constant factor; it has no effect on the order of growth

Binary search

- Input characterized by an array a_i, $0 \leq i < n$
  - Elements in a are sorted in nondecreasing order
  - Problem to determine whether an element k is present in a
    - If it is present, return its index j such that $a_j = k$
    - If the element is not present, return -1
- Instance of problem given by

$$P = (n, a_l, \ldots, a_r, k)$$

  where $a_l, \ldots, a_r$ are n elements in the list to be searched for k
- Assume `small ( P )` is true if $r == l$
  - There is only one element to be tested
  - In this case, `solution ( P )` returns i if $a_i = x$; otherwise it returns -1
  - $g(1) = \Theta(1)$
- If $r - l \geq 1$, compute $m = (l + r)/2$ leading to three cases
  1. $a_m = k$
     - Problem is immediately solved
  2. $a_m < k$
     - Discard the elements whose index is smaller than m
     - Search for k in $a[m + 1..r]$
  3. $a_m > k$
     - Discard the elements whose index is larger than m
     - Search for k in $a[l..m - 1]$
- Case 2 and 3 result in only one subproblem; division takes $\Theta(1)$ time
  - Answer to the remaining subproblem is also the answer to the original problem; no need to combine the solutions
- Problem initially invoked by `bin_search ( a, 1, n, k );`
- Recursive Algorithm

```
// Given an array a[l..r] of elements in nondecreasing order, 0 <= l <= r,
// determine whether k is present, and if so, return i such that k = a[i];
// else return -1
Algorithm bin_search ( a, l, r, k )
{
    if ( r == l )    // Small problem instance
    {
        if ( k == a[r] )
```

```
    return ( r );
else
    return ( -1 );
    }

    // Reduce problem to smaller instances

    m = ( r + l ) / 2;
    if ( k == a[m] )
        return ( m );

    if ( k < a[m] )
        return ( bin_search ( a, l, m - 1, k ) );
    else
        return ( bin_search ( a, m + 1, r, k ) );
}
```

- Iterative version

```
// Given an array a[0..n-1] of elements in nondecreasing order, n >= 0,
// determine whether k is present, and if so, return i such that k = a[i];
// else return -1
Algorithm bin_search ( a, n, k )
{
    low = 0;
    high = n-1;
    while ( low <= high )
    {
        mid = ( low + high ) / 2;
        if ( k < a[mid] )
            high = mid - 1;
        else
            if ( k > a[mid] )
                low = mid + 1;
            else
                return ( mid );
    }

    return ( -1 );
}
```

Finding the maximum and minimum

- Linear scan algorithm
  - Straightforward comparison – $2(n-1)$ comparisons
  - Compare for min only if comparison for max fails
  - Best case: increasing order – $n-1$ comparisons
  - Worst case: decreasing order – $2(n-1)$ comparisons
  - Average case: $3n/2-1$ comparisons
- Divide and conquer algorithm

```
item_t a[n]                         // Global array
maxmin ( i, j, max, min )
{
    if ( i == j )                   // Only one element
    {
        max = min = a[i]
        return
    }

    if ( i == j - 1 )               // Only two elements
    {
        if ( a[i] < a[j] )
        {
            max = a[j]
            min = a[i]
        }
        else
        {
            max = a[i]
            min = a[j]
        }
        return
    }

    // Divide

    mid = floor ( ( i + j ) / 2 )
    maxmin ( i, mid, max, min )
    maxmin ( mid+1, j, max1, min1 )

    // Conquer

    if ( max < max1 ) max = max1
    if ( min > min1 ) min = min1
    return
}
```

- Analyzing divide and conquer `maxmin` described by the recurrence

$$T_n = \begin{cases} T_{\frac{n}{2}} + T_{\frac{n}{2}} + 2 & n > 2 \\ 1 & n = 2 \\ 0 & n = 1 \end{cases}$$

Let $n = 2^k$, for some $k > 0$

$$\begin{aligned} T_n &= 2T_{\frac{n}{2}} + 2 \\ &= 2(2T_{\frac{n}{4}} + 2) + 2 \\ &= 2^2 T_{\frac{n}{2^2}} + 2^2 + 2 \\ &= 2^2(2T_{\frac{n}{2^3}} + 2) + 2^2 + 2 \\ &= 2^3 T_{\frac{n}{2^3}} + 2^3 + 2^2 + 2 \\ &\vdots \quad \vdots \\ &= 2^{k-1} T_{\frac{n}{2^{k-1}}} + \sum_{i=1}^{k-1} 2^i \end{aligned}$$

$$
\begin{aligned}
&= 2^{k-1} + \sum_{i=1}^{k-1} 2^i \\
&= 2^{k-1} + \frac{2^{(k-1)+1} - 1}{2-1} \\
&= 2^{k-1} + 2^k - 1 \\
&= \frac{n}{2} + n - 1 \\
&= \frac{3n}{2} - 1
\end{aligned}
$$

Merge sort

- Recursive algorithm
- Algorithm can be described by the following recurrence

$$
T_n = \begin{cases} 2T_{\frac{n}{2}} + cn & n > 1, c \text{ is a constant} \\ a & n = 1, a \text{ is a constant} \end{cases}
$$

Quick sort

- The array to be sorted is partitioned at a *pivot element* such that the elements at indices less than that of the pivot are less than the pivot while the elements with indices greater than the pivot are larger than the pivot
- Quicksort eliminates the need for a subsequent merge as required by merge sort
- Performance analysis
  - Worst case when elements are already in sorted order
- Randomized quick sort
  - Select pivot as median of three
  - Select a random element as the pivot (Las Vegas algorithm)

Selection

- Selecting kth smallest element using partition from quicksort

Strassen's matrix multiplication

- Given two $n \times n$ matrices A and B; their product C is given by

$$
C_{ij} = \sum_{k=1}^{n} A_{ik} B_{kj}
$$

  - It requires n multiplications for each element in C
  - Total time to perform above computation is $\Theta(n^3)$
- Divide and conquer strategy
  - Assume that $n = 2^k$

  - If $n \neq 2^k$, we can add enough rows and columns to make it satisfy our assumption
- Each of A, B, and C is partitioned into submatrices of size $\frac{n}{2}$ such that

$$\begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix} = \begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix}$$

- This yields eight multiplications of $\frac{n}{2} \times \frac{n}{2}$ matrices ($T_{\frac{n}{2}}$), and four additions of the same size of matrices (assume cn^2 for some constant c, starting with $4 \cdot (\frac{n}{2})^2$)
- The recurrence is

$$T_n = \begin{cases} b & n \leq 2 \\ 8T_{\frac{n}{2}} + cn^2 & n > 2 \end{cases}$$

- A solution of this recurrence yields $T_n = O(n^3)$ (no improvement)
- Observation: Matrix multiplications are more expensive ($O(n^3)$) than matrix additions ($O(n^2)$)
- Strassen's method relies on devising a clever workaround to minimize the number of multiplications while increasing the number of additions
  - 7 multiplications and 18 additions/subtractions
- The intermediate computations are

$$\begin{aligned} P_1 &= (A_{11} + A_{22})(B_{11} + B_{22}) \\ P_2 &= (A_{21} + A_{22})B_{11} \\ P_3 &= A_{11}(B_{12} - B_{22}) \\ P_4 &= A_{22}(B_{21} - B_{11}) \\ P_5 &= (A_{11} + A_{12})B_{22} \\ P_6 &= (A_{21} - A_{11})(B_{11} + B_{12}) \\ P_7 &= (A_{12} - A_{22})(B_{21} + B_{22}) \end{aligned}$$

- The elements of C are given as

$$\begin{aligned} C_{11} &= P_1 + P_4 - P_5 + P_7 \\ C_{12} &= P_3 + P_5 \\ C_{21} &= P_2 + P_4 \\ C_{22} &= P_1 + P_3 - P_2 + P_6 \end{aligned}$$

- The recurrence relation is

$$T_n = \begin{cases} b & n \leq 2 \\ 7T_{\frac{n}{2}} + an^2 & n > 2 \end{cases}$$

  where a and b are constants
- Solving the recurrence

$$\begin{aligned} T_n &= an^2[1 + \frac{7}{4} + \left(\frac{7}{4}\right)^2 + \cdots + \left(\frac{7}{4}\right)^{k-1}] + 7^k T(1) \\ &\leq cn^2 \left(\frac{7}{4}\right)^{\lg n} + 7^{\lg n}, c \text{ is a constant} \\ &= cn^{\lg 4 + \lg 7 - \lg 4} + n^{\lg 7} \\ &= O(n^{\lg 7}) \\ &\approx O(n^{2.81}) \end{aligned}$$

Convex hull

- Structure (smallest *convex* polygon) used in the construction of geometric structures
- Polygon: A piecewise-linear, closed curve in a plane
  - Curve composed of a sequence of straight line segments, or *sides* of polygon
  - Curve starts and ends at the same point
  - A point that is common to two sides is called a *vertex*

Definition 1 *A polygon is defined to be **convex** if for any two vertices p_1 and p_2 inside the polygon, the directed line segment from p_1 to p_2 (denoted as $\langle p_1, p_2 \rangle$) is fully contained in the polygon.*

Definition 2 *The **convex hull** of a set S of points in the plane is defined to be the smallest convex polygon containing all the points of S.*

- Vertices of the convex hull of a set S of points form a [not necessarily proper] subset of S
- Two variants of the convex hull problem
  1. Obtain the vertices (*extreme points*) of the convex hull
  2. Obtain the vertices of the convex hull in some order, such as clockwise
- Obtaining extreme points of a given set S of points in a plane

```
for each p in S
{
    for each possible triplet of points in S
        if p is not inside the triangle formed by any triplet
            mark p as an extreme point;
}
```

  - Testing for p being inside a given triangle is performed in $\Theta(1)$ time
  - Number of possible triangles is $\Theta(n^3)$
  - Since there are n points, the above algorithm runs in $\Theta(n^4)$ time
- Divide and conquer allows us to solve the convex hull problem (either form) in $O(n \log n)$ time
- Geometric primitives
  - Let A be an $n \times n$ matrix with elements denoted by a_{ij}

    Definition 3 *The ij**th minor** of A, denoted by A_{ij}, is defined to be the submatrix of A obtained by deleting the ith row and jth column.*

    Definition 4 *The **determinant** of A, denoted by $|A|$, is given by*

$$|A| = \begin{cases} a_{11} & n = 1 \\ a_{11} \cdot |A_{11}| - a_{12} \cdot |A_{12}| + \cdots + (-1)^{n-1} \cdot |A_{1n}| & n > 1 \end{cases}$$

  - Consider the directed line segment $\langle p_1, p_2 \rangle$ from some point $p_1 = (x_1, y_1)$ to some other point $p_2 = (x_2, y_2)$. If $q = (x_3, y_3)$ is another point, we say q ***is to the left [right] of*** $\langle p_1, p_2 \rangle$ if the angle $p_1 p_2 q$ is a left [right] turn.
    - An angle θ is said to be a left turn if $\theta \leq 180$; otherwise, it is considered to be a right turn
    - We can check whether q is to the left or right of $\langle p_1, p_2 \rangle$ by evaluating the determinant of their coordinates

$$\begin{vmatrix} x_1 & x_2 & x_3 \\ y_1 & y_2 & y_3 \\ 1 & 1 & 1 \end{vmatrix}$$

Determinant > 0**.** q is to the left of $\langle p_1, p_2 \rangle$

Determinant $= 0$**.** The three points are colinear

Determinant < 0**.** q is to the right of $\langle p_1, p_2 \rangle$

- Point p is within the triangle formed by p_1, p_2, and p_3 iff p is to the right of each of the three lines p_1p_2, p_2p_3, and p_3p_1
- For any three points (x_1, y_1), (x_2, y_2), and (x_3, y_3), the *signed area* formed by the corresponding triangle is given by one-half of the above determinant

– Checking for point p to be inside a convex polygon Q given by vertices $p_1, p_2, \ldots, p_n$

* Consider a horizontal line h from $-\infty$ to ∞ and passing through p
* Two possibilities
  1. h does not intersect any of the edges of Q
     - p is outside Q
  2. h intersects some of the edges of Q
     - There can be at most two points of intersection; if h intersects Q at a single point, it is considered as two points
     - Count the number of points to the left of p
     - If the number is even, p is outside Q; otherwise it is inside Q
* The method takes $\Theta(n)$ time to check whether p is interior to Q

- QuickHull Algorithm

– Similar to quicksort

– Computes the convex hull of a set X of n points in the plane

```
convex_hull quick_hull ( X )
{
    // Identify p_1 and p_2 as part of the convex hull

    p_1 = point in X with the smallest x-coordinate value
    p_2 = point in X with the largest x-coordinate value
    divide X into X_1 and X_2 such that
    {
        X_1 = set of points to the left of line segment <p_1, p_2>
        X_2 = set of points to the right of line segment <p_1, p_2>
        // Convex hull of X_1 is the upper hull
        // Convex hull of X_2 is the lower hull
        // Compute the upper and lower hull using the divide and conquer
        // algorithm called hull; union of the two hulls is the overall convex
        // hull

        H1 = hull ( X_1, p_1, p_2 );
        H2 = hull ( X_2, p_2, p_1 );
    }

    return ( H1 + H2 );    // Union of H1 and H2
}
```

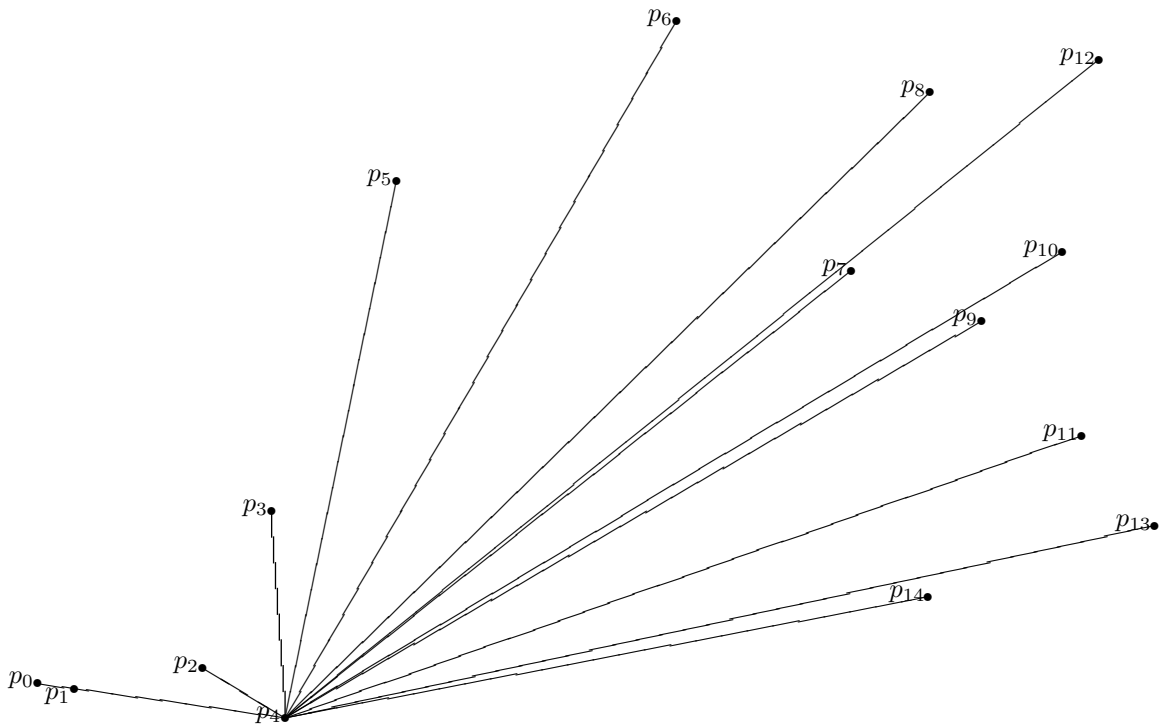
– The case of ties when more than one point has the extreme x-coordinate, can be handled as a special case

```
convex_hull hull ( X, p1, p2 )
{
    for each point p in X
        compute the area of the triangle formed by p, p1, and p2
```

```
        p3 = p, for which the above area is maximized

        // In case of a tie for point with maximum area, select p3 to be the point
        // for which the angle p3p1p2 is maximized

        // Divide X into two parts X1 and X2 based on p3

        X1 = points in X to the left of <p1,p3>
        X2 = points in X to the left of <p3,p2>

        // There is no point in X1 that is to the left of both <p1,p3> and <p3,p2>
        // Remaining points are interior points and can be dropped from further
        // consideration

        H1 = hull ( X1, p1, p3 );
        H2 = hull ( X2, p3, p2 );

        return ( H1 + H2 );         // Union of H1 and H2
    }
```

  - Analyzing the algorithm
    - Let there be m points in X_1
    - We can identify p_3 in $O(m)$ time
    - Partitioning X_1 into two is also done in $O(m)$ time
    - Merging the two convex hulls is done in $O(1)$ time
    - Run time of `hull` on m points is given by T_m
    - Size of resultant parts given by m_1 and m_2; $m_1 + m_2 \leq m$
    - Recurrence relation is:
      $$T_m = T_{m_1} + T_{m_2} + O(m)$$
      - Similar to `quicksort`
    - Worst case run-time: $O(m^2)$ for m points when the partitioning is highly uneven
    - For nearly even partitioning, run-time given by $O(m \lg m)$

- Graham's scan
  - Given a set of points S in a 2D plane
  - Identify the point p with smallest y coordinate
    - Break ties by picking the point with smallest x coordinate
  - Sort the points by angle subtended by points and p with the horizontal axis
  - Scan through sorted list starting with p, three points at a time
    - If points p_1, p_2, p_3 form a left turn, they all are on the hull
    - If p_1, p_2, p_3 form a right turn, p_2 is not on the convex hull
  - Example


p_6
p_{12}
p_8
p_5
p_{10}
p_7
p_9
p_{11}
p_3
p_{13}
p_{14}
p_2
p_0
p_1
p_4