بسم اللّٰه الرّحمن الرّحیم

دانشگاه صنعتی اصفهان ــ دانشکدهٔ مهندسی برق و کامپیوتر
(نیم‌سال تحصیلی ۴۰۰۱)

طراحی الگوریتم‌ها

حسین فلسفین

Multiplication of Large Integers

A straightforward way to represent a large integer is to use an array of integers, in which each array slot stores one digit. For example, the integer 543,127 can be represented in the array S as follows:

$$\underset{S[6]}{\underline{5}} \quad \underset{S[5]}{\underline{4}} \quad \underset{S[4]}{\underline{3}} \quad \underset{S[3]}{\underline{1}} \quad \underset{S[2]}{\underline{2}} \quad \underset{S[1]}{\underline{7}}.$$

We will assume this representation and use the defined data type large integer to mean an array big enough to represent the integers in the application of interest.

Linear-time algorithms can readily be written that do the operations

$$u \times 10^m \qquad u \ \mathtt{divide}\ 10^m \qquad u \ \mathtt{rem}\ 10^m$$

u *represents a large integer,* m *is a non-negative integer, divide returns the quotient in integer division, and rem returns the remainder.*

rem را با *mod* نیز ممکن است نشان دهند.

A simple quadratic-time algorithm for multiplying large integers is one that mimics the standard way learned in grammar school. We will develop one that is better than quadratic time. Our algorithm is based on using divide-and-conquer to split an n-digit integer into two integers of approximately $n/2$ digits.

$$\underbrace{567,832}_{6\text{ digits}} = \underbrace{567}_{3\text{ digits}} \times 10^3 + \underbrace{832}_{3\text{ digits}}$$

$$\underbrace{9,423,723}_{7\text{ digits}} = \underbrace{9423}_{4\text{ digits}} \times 10^3 + \underbrace{723}_{3\text{ digits}}$$

In general, if n is the number of digits in the integer u, we will split the integer into two integers, one with $\lceil n/2 \rceil$ and the other with $\lfloor n/2 \rfloor$ as follows:

$$\underbrace{u}_{n \text{ digits}} = \underbrace{x}_{\lceil n/2 \rceil \text{ digits}} \times 10^m + \underbrace{y}_{\lfloor n/2 \rfloor \text{ digits}}$$

With this representation, the exponent m of 10 is given by $m = \lfloor n/2 \rfloor$.

If we have two n-digit integers $u = x \times 10^m + y$ and $v = w \times 10^m + z$ their product is given by

$$uv = (x \times 10^m + y)(w \times 10^m + z) = \\ xw \times 10^{2m} + (xz + yw) \times 10^m + yz.$$

We can multiply u and v by doing four multiplications on integers with about half as many digits and performing linear-time operations.

$$\begin{aligned}567,832 \times 9,423,723 &= \left(567 \times 10^3 + 832\right)\left(9423 \times 10^3 + 723\right)\\ &= 567 \times 9423 \times 10^6 + (567 \times 723 + 9423 \times 832)\\ &\quad \times 10^3 + 832 \times 723\end{aligned}$$

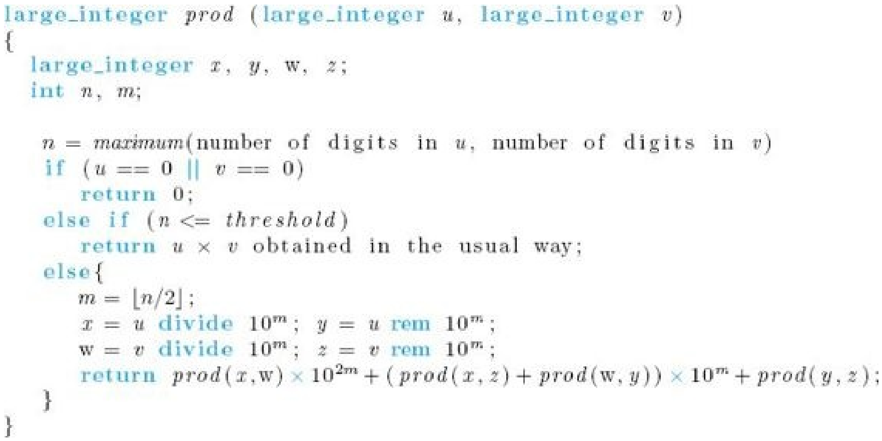
Recursively, these smaller integers can then be multiplied by dividing them into yet smaller integers. This division process is continued until a threshold value is reached, at which time the multiplication can be done in the standard way.

Large Integer Multiplication

Problem: ***Multiply two large integers, u and v.***

Inputs: ***large integers u and v.***

Outputs: ***prod, the product of u and v.***



```
large_integer prod (large_integer u, large_integer v)
{
  large_integer x, y, w, z;
  int n, m;

   n = maximum(number of digits in u, number of digits in v)
   if (u == 0 || v == 0)
      return 0;
   else if (n <= threshold)
      return u × v obtained in the usual way;
   else{
      m = ⌊n/2⌋;
      x = u divide $10^m$; y = u rem $10^m$;
      w = v divide $10^m$; z = v rem $10^m$;
      return prod(x,w) × $10^{2m}$ + (prod(x, z) + prod(w, y)) × $10^m$ + prod(y, z);
   }
}
```

Remember that divide, rem, and $\times$ represent linear-time functions that we need to write.

Worst-Case Time Complexity of Large Integer Multiplication

The worst case is when both integers have no digits equal to 0, because the recursion only ends when threshold is passed. We will analyse this case. Suppose n is a power of 2. Then x, y, w, and z all have exactly $n/2$ digits, which means that the input size to each of the four recursive calls to prod is $n/2$. Because $m = n/2$, the linear-time operations of addition, subtraction, divide 10^m, rem 10^m, and $\times 10^m$ all have linear-time complexities in terms of n. We group all the linear-time operations in the one term cn, where c is a positive constant.

$$W(n) = 4W\left(\frac{n}{2}\right) + cn \qquad \text{for } n > s,\ n \text{ a power of } 2$$
$$W(s) = 0.$$

Our algorithm for multiplying large integers is still quadratic. The problem is that the algorithm does four multiplications on integers with half as many digits as the original integers. If we can reduce the numbers of these multiplications, we can obtain an algorithm that is better than quadratic. Recall that function prod must determine xw, $xz + yw$, and yz. We accomplished this by calling function prod recursively four times to compute xw, xz, yw, and yz.

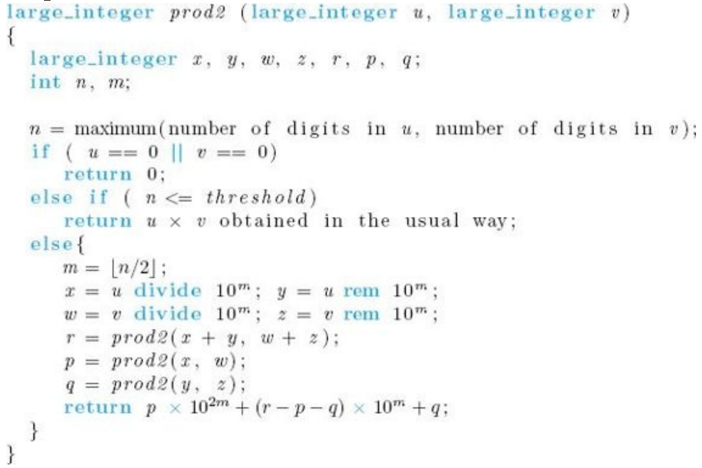
*If instead we set $r = (x + y)(w + z) = xw + (xz + yw) + yz$, then $xz + yw = r - xw - yz$. This means we can get the three values xw, $xz + yw$, and yz by determining the following three values: $r = (x + y)(w + z)$, xw, and yz. To get these three values we need to do only **three** multiplications, while doing some additional linear-time additions and subtractions.*

Large Integer Multiplication 2

Problem: ***Multiply two large integers, u and v.***

Inputs: ***large integers u and v.***

Outputs: ***$prod2$, the product of u and v.***



```
large_integer prod2 (large_integer u, large_integer v)
{
  large_integer x, y, w, z, r, p, q;
  int n, m;

  n = maximum(number of digits in u, number of digits in v);
  if ( u == 0 || v == 0)
     return 0;
  else if ( n <= threshold)
     return $u \times v$ obtained in the usual way;
  else{
     $m = \lfloor n/2 \rfloor$;
     x = u divide $10^m$; y = u rem $10^m$;
     w = v divide $10^m$; z = v rem $10^m$;
     r = prod2(x + y, w + z);
     p = prod2(x, w);
     q = prod2(y, z);
     return $p \times 10^{2m} + (r - p - q) \times 10^m + q$;
  }
}
```

Worst-Case Time Complexity (Large Integer Multiplication 2)

We analyze how long it takes to multiply two n-digit integers. The worst case happens when both integers have no digits equal to 0, because in this case the recursion ends only when the threshold is passed. We analyze this case. If n is a power of 2, then x, y, w, and z all have $n/2$ digits.

$$\frac{n}{2} \leq \text{ digits in } x+y \leq \frac{n}{2}+1.$$

$$\frac{n}{2} \leq \text{ digits in } w+z \leq \frac{n}{2}+1.$$

	Input Size
$prod2(x+y, w+z)$	$\frac{n}{2} \leq \text{input size} \leq \frac{n}{2}+1$
$prod2(x, w)$	$\frac{n}{2}$
$prod2(y, z)$	$\frac{n}{2}$

n	x	y	$x+y$	Number of Digits in $x+y$
4	10	10	20	$2=\frac{n}{2}$
4	99	99	198	$3=\frac{n}{2}+1$
8	1000	1000	2000	$4=\frac{n}{2}$
8	9999	9999	19,998	$5=\frac{n}{2}+1$

$$\boxed{\begin{array}{l} 3W\left(\frac{n}{2}\right)+cn \leq W(n) \leq 3W\left(\frac{n}{2}+1\right)+cn \quad \text{for } n>s,\ n \text{ a power of } 2 \\ W(s)=0, \end{array}}$$

$$W(n) \in \Theta\left(n^{\log_2 3}\right) \approx \Theta\left(n^{1.58}\right).$$

When Not to Use Divide-and-Conquer

If possible, we should avoid divide-and-conquer in the following two cases:

1. *An instance of size n is divided into two or more instances each almost of size n.*

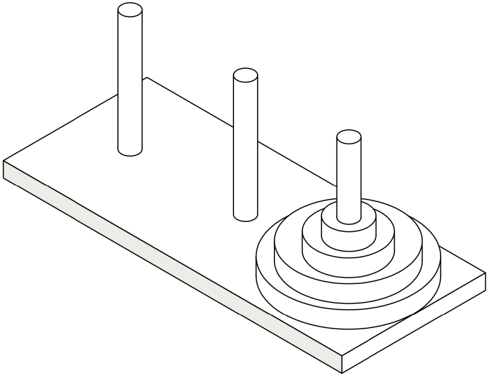
2. *An instance of size n is divided into almost n instances of size n/c, where c is a constant.*

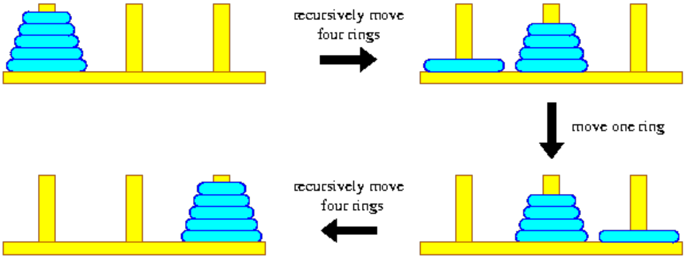
The first partitioning leads to an exponential-time algorithm, where the second leads to a $n^{\Theta(\log n)}$ algorithm. Neither of these is acceptable for large values of n.

Sometimes, on the other hand, a problem requires exponentiality, and in such a case there is no reason to avoid the simple divide-and-conquer solution.

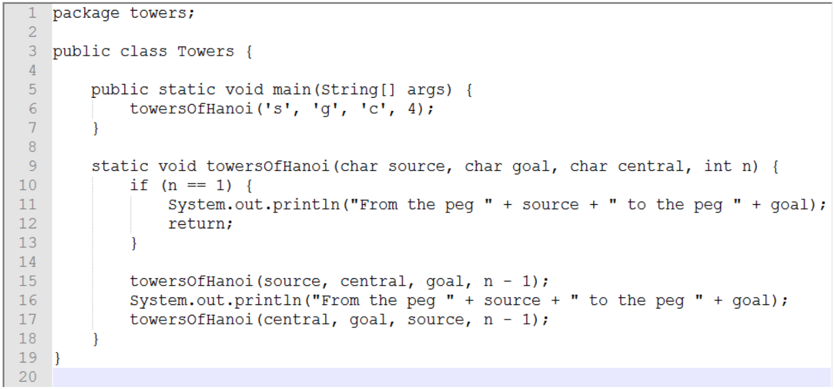
The Towers of Hanoi problem consists of three pegs and n disks of different sizes. The object is to move the disks that are stacked, in decreasing order of their size, on one of the three pegs to a new peg using the third one as a temporary peg. The problem should be solved according to the following rules: (1) when a disk is moved, it must be placed on one of the three pegs; (2) only one disk may be moved at a time, and it must be the top disk on one of the pegs; and (3) a larger disk may never be placed on top of a smaller disk.

The problem requires an exponentially large number of moves in terms of n.




recursively move four rings
move one ring
recursively move four rings

حل مسئلهٔ برج‌های هانوی با بهره‌گیری از راهبرد تقسیم و غلبه (کد جاوا)



```java
package towers;

public class Towers {

    public static void main(String[] args) {
        towersOfHanoi('s', 'g', 'c', 4);
    }

    static void towersOfHanoi(char source, char goal, char central, int n) {
        if (n == 1) {
            System.out.println("From the peg " + source + " to the peg " + goal);
            return;
        }

        towersOfHanoi(source, central, goal, n - 1);
        System.out.println("From the peg " + source + " to the peg " + goal);
        towersOfHanoi(central, goal, source, n - 1);
    }
}
```

به‌ازای $n = 4$ خروجی الگوریتم به‌شکل زیر خواهد بود:

```
From the peg s to the peg c
From the peg s to the peg g
From the peg c to the peg g
From the peg s to the peg c
From the peg g to the peg s
From the peg g to the peg c
From the peg s to the peg c
From the peg s to the peg g
From the peg c to the peg g
From the peg c to the peg s
From the peg g to the peg s
From the peg c to the peg g
From the peg s to the peg c
From the peg s to the peg g
From the peg c to the peg g
```

اگر تعداد دفعات جابجا کردن دیسک‌ها را با $S(n)$ نشان دهیم:

$$S(n) = 2S(n-1) + 1$$

حال با استقرا نشان می‌دهیم که $S(n) = 2^n - 1$:

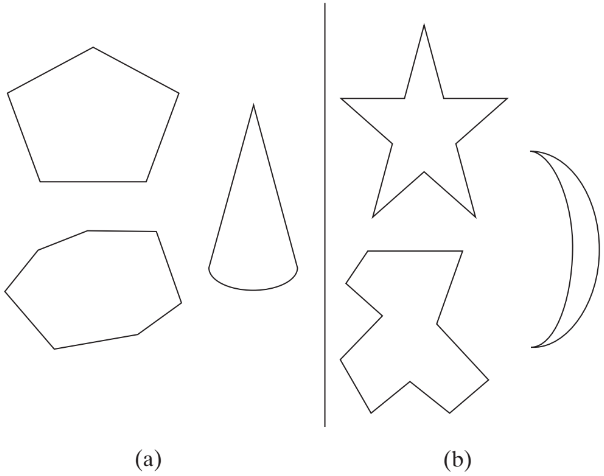
پایهٔ استقرا: $S(1) = 1 = 2^1 - 1$؛
فرض استقرا: فرض کنید که برای یک عدد صحیح مثبت همچون k داریم $S(k) = 2^k - 1$؛
باید نشان دهیم که $S(k+1) = 2^{k+1} - 1$. برای این منظور:

$$S(k+1) = 2S(k) + 1 = 2(2^k - 1) + 1 = 2^{k+1} - 1.$$

Convex-Hull Problem

Finding the convex hull for a given set of points in the plane or a higher dimensional space is one of the most important—some people believe the most important—problems in computational geometry.

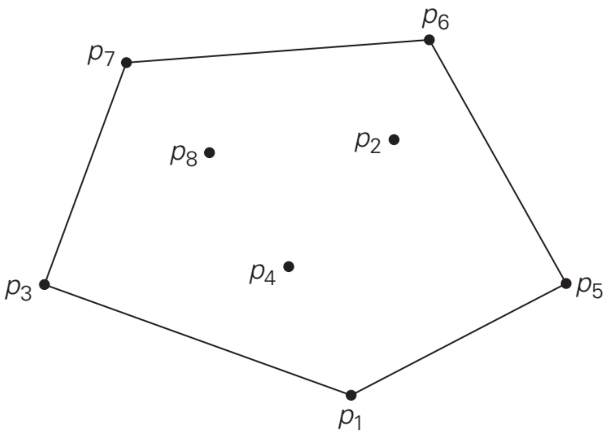
DEFINITION A set of points (finite or infinite) in the plane is called convex if for any two points p and q in the set, the entire line segment with the endpoints at p and q belongs to the set.



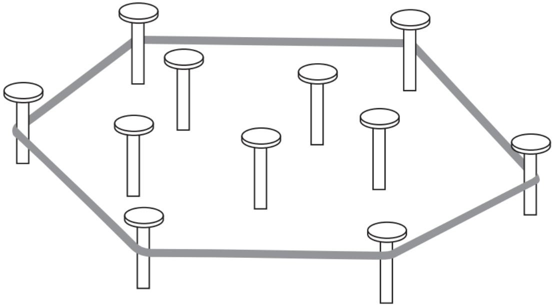
(a) Convex sets. (b) Sets that are not convex.

DEFINITION The convex hull of a set S of points is the smallest convex set containing S. (The "smallest" requirement means that the convex hull of S must be a subset of any convex set containing S.)

If S is convex, its convex hull is obviously S itself. If S is a set of two points, its convex hull is the line segment connecting these points. If S is a set of three points not on the same line, its convex hull is the triangle with the vertices at the three points given; if the three points do lie on the same line, the convex hull is the line segment with its endpoints at the two points that are farthest apart.



p_6
p_7
p_8
p_2
p_4
p_3
p_5
p_1



Rubber-band interpretation of the convex hull.

How can we solve the convex-hull problem in a brute-force manner?

Observation: A line segment connecting two points p_i and p_j of a set of n points is a part of the convex hull's boundary if and only if all the other points of the set lie on the same side of the straight line through these two points. (For the sake of simplicity, we assume here that no three points of a given set lie on the same line.) Repeating this test for every pair of points yields a list of line segments that make up the convex hull's boundary.

☞ *The straight line through two points (x_1, y_1), (x_2, y_2) in the coordinate plane can be defined by the equation $ax + by = c$, where $a = y_2 - y_1$, $b = x_1 - x_2$, $c = x_1y_2 - y_1x_2$. Such a line divides the plane into two half-planes: for all the points in one of them, $ax + by > c$, while for all the points in the other, $ax + by < c$. (For the points on the line itself, of course, $ax + by = c$.)*

☞ *To check whether certain points lie on the same side of the line, we can simply check whether the expression $ax + by - c$ has the same sign for each of these points.*

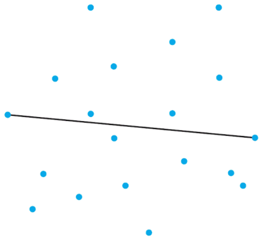
☞ *What is the time efficiency of this algorithm? It is in $O(n^3)$: for each of $\binom{n}{2}$ pairs of distinct points, we may need to find the sign of $ax + by - c$ for each of the other $n - 2$ points.*

الگوریتم‌های متعددی برای مسئلهٔ یافتن پوش محدب یک مجموعهٔ متناهی از نقاط در صفحه (و در حالت کلی‌تر در $\mathbb{R}^n$) پیشنهاد شده اند. بخش ۳.۳۳ از کتاب **CLRS** را ملاحظه بفرمایید: ***Graham's scan*** و ***Jarvis's march***.

QuickHull

Let S be a set of $n > 1$ points $p_1(x_1, y_1), \ldots, p_n(x_n, y_n)$ in the Cartesian plane. We assume that the points are sorted in nondecreasing order of their x coordinates, with ties resolved by increasing order of the y coordinates of the points involved.

It is not difficult to prove the geometrically obvious fact that the leftmost point p_1 and the rightmost point p_n are two distinct extreme points of the set's convex hull.



☞ *Let $\overrightarrow{p_1p_n}$ be the straight line through points p_1 and p_n directed from p_1 to p_n. This line separates the points of S into two sets: S_1 is the set of points to the left of this line, and S_2 is the set of points to the right of this line.*

☞ *The points of S on the line $\overrightarrow{p_1p_n}$, other than p_1 and p_n, cannot be extreme points of the convex hull and hence are excluded from further consideration.*

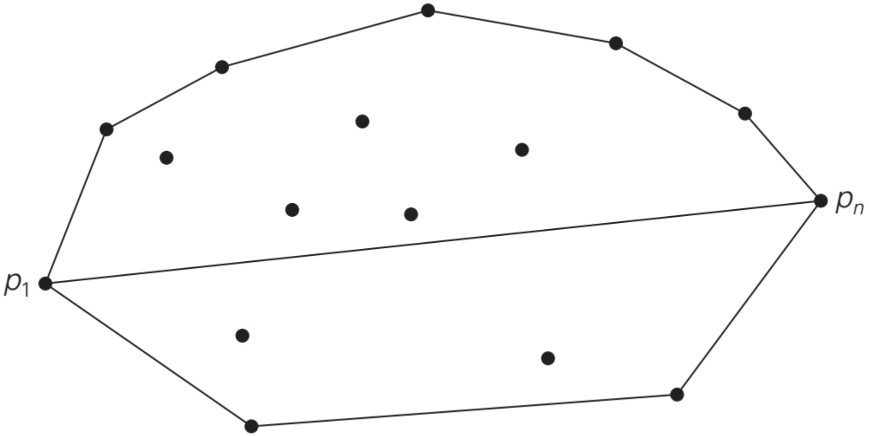
☞ *The boundary of the convex hull of S is made up of two polygonal chains: an “upper” boundary and a “lower” boundary. The “upper” boundary, called the upper hull, is a sequence of line segments with vertices at p_1, some of the points in S_1 (if S_1 is not empty) and p_n. The “lower” boundary, called the lower hull, is a sequence of line segments with vertices at p_1, some of the points in S_2 (if S_2 is not empty) and p_n. The convex hull of the entire set S is composed of the upper and lower hulls, which can be constructed independently and in a similar fashion. For concreteness, let us discuss how QuickHull proceeds to construct the upper hull; the lower hull can be constructed in the same manner.*

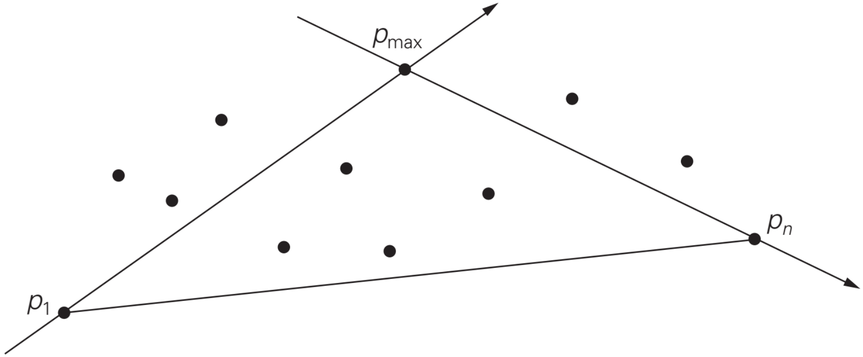
☞ *If S_1 is not empty, the algorithm identifies point $p_{\max}$ in S_1, which is the* *farthest* *from the line $\overrightarrow{p_1p_n}$. Then the algorithm identifies all the points of set S_1 that are to the left of the line $\overrightarrow{p_1p_{\max}}$; these are the points that will make up the set $S_{1,1}$. The points of S_1 to the left of the line $\overrightarrow{p_{\max}p_n}$ will make up the set $S_{1,2}$.*

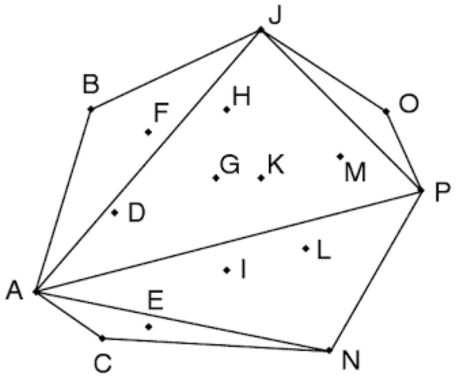
☞ *$p_{\max}$ is a vertex of the upper hull.*

☞ *The points inside $\triangle p_1p_{\max}p_n$* *cannot* *be vertices of the upper hull (and hence can be* *eliminated* *from further consideration).*

☞ *There are no points to the left of both lines $\overrightarrow{p_1p_{\max}}$ and $\overrightarrow{p_{\max}p_n}$. Therefore, the algorithm can continue constructing the upper hulls of $p_1 \cup S_{1,1} \cup p_{\max}$ and $p_{\max} \cup S_{1,2} \cup p_n$ recursively and then simply concatenate them to get the upper hull of the entire set $p_1 \cup S_1 \cup p_n$.*


p_1
p_n


$p_{\max}$
p_n
p_1



[A B C D E F G H I J K L M N O P]

A [B D F G H J K M O] P [C E I L N]

A [B F] J [O] P N [C E]

A B J O P N C

An analysis of the time complexity of QuickHull

QuickHull دارای پیچیدگی زمانی بدترین حالت $O(n^2)$ است که در مقایسه با الگوریتم ***Brute-force*** (با پیچیدگی زمانی $O(n^3)$) بهتر است.