Queens College, CUNY, Department of Computer Science

Numerical Methods

CSCI 361 / 761

Spring 2018

Instructor: Dr. Sateesh Mane

February 8, 2019

1 Lecture 1

- This is a collection of "useful algorithms" and lessons in organizing calculations to save on time and/or memory.
- **Note: Although this document is called "Lecture 1" the material will be taught in class in more than one lecture.**
- In other words, ***do not panic.***

1.1 GCD greatest common divisor

For two positive integers a and b, there is a simple and efficient algorithm to compute their greatest common divisor (gcd). It dates from classical Greece and we call it "Euclid's algorithm" although Euclid himself never claimed to invent it. *But he wrote it down, and his work has survived.* It is possible that other ancient civilizations also knew it (China, India, Egypt, etc.?) but we do not know. Without loss of generality, suppose $a > b$ (and $b > 0$), else just swap a and b. Then we proceed recursively.

- Calculate $c = a \ \% \ b$ (the remainder after integer division).
- If $c == 0$, then gcd $= b$ (because b divides a).
- If $c == 1$, then gcd $= 1$.
- Else proceed recursively and compute $\gcd(b, c)$.

The process must terminate because the integers are positive and always get smaller and eventually must reach 0 or 1.

- Comment: if b is zero, the Excel returns the $\gcd(a, 0)$ equals a. This makes sense: if b is zero, then a divides a and it also divides 0 (the remainder is zero in both cases). A special case to note.

Proof of Euclid's algorithm

- We consider only positive integers.
- We are given positive integers a and b and $a > b$.
- The gcd (call it g) is the largest integer which divides both a and b.
- Let $a \,\%\, b = c$.
- This means there is some integer n and we can write a in the form as follows.

$$a = nb + c\,.$$

- Now by definition g (gcd) divides both a and b.
- So if we divide the above equation by g then g must divide c as well, because all the terms in the resulting equation must be integers:

$$c/g = (a/g) - n(b/g) = \text{integer} - \text{integer} = \text{integer}\,.$$

- Therefore c/g is an integer.
- Therefore g divides both b and c.
- Therefore $g = \gcd(b, c)$.
- Therefore we iterate and the algorithm will terminate in a finite number of steps.
- ***Do you understand?***

1.2 Conversion decimal to binary or hex, etc.

Computers store numbers internally as binary digits (or hexadecimal). But they display the numbers on screen in decimal format. We also input numbers (in online forms, etc.) in decimal format. Hence conversion of digits from decimal to/from binary or hex is a frequently performed operation. Hence it is essential to do it efficiently. For simplicity we consider only positive numbers.

1.2.1 Decimal integer to binary

We begin with integers. We shall consider fractions later. Proceed recursively:

- Divide the number by 2 (integer division) and store the remainder.
- Divide the quotient by 2 (integer division) and store the remainder.
- Repeat until the quotient reaches zero. This must happen and the algorithm will terminate.
- Read the remainders in reverse order, that is the binary representation of the decimal number.
- Example: $a = 19$

19	
9	1
4	1
2	0
1	0
0	1

- Hence $19_{\text{dec}} = 10011_{\text{bin}}$.

1.2.2 Decimal integer to hex

The fundamental algorithm is exactly the same, but we perform integer division by 16 (not 2). Proceed recursively:

- Example: $a = 30$

30	
1	14
0	1

- The new twist here is that the remainder takes values from 0 through 15. Hence we need a lookup array for hex digits $\texttt{hex\_digits} = \{\texttt{0}, \texttt{1}, \texttt{2}, \texttt{3}, \texttt{4}, \texttt{5}, \texttt{6}, \texttt{7}, \texttt{8}, \texttt{9}, \texttt{A}, \texttt{B}, \texttt{C}, \texttt{D}, \texttt{E}, \texttt{F}\}$. (Hence the output is alphanumeric.) In the above example, the lookup array says $14 \to E$ so

$$30_{\text{dec}} = 1E_{\text{hex}}\,.$$

1.2.3 Decimal integer to base b

Obviously the same algorithm works for any base $b \geq 2$. If $b > 10$ then we require a lookup table to output alphanumeric digits, but that is not a problem.

1.2.4 Decimal fraction to binary

The algorithm is essentially the same as above, but now we multiply by 2 at each step. Also, we keep the fraction and record the integer part in an array. It is simplest to explain with some examples. For ease of reading, it is convenient to store the integers on the left

- Example: $a = \frac{1}{4}$

	$\frac{1}{4}$
0	$\frac{1}{2}$
1	0

- For fractions, we read the array from the top down. Hence $(\frac{1}{4})_{\text{dec}} = 0.01_{\text{bin}}$.
- Example: $a = \frac{1}{3}$

	$\frac{1}{3}$
0	$\frac{2}{3}$
1	$\frac{1}{3}$
0	$\frac{2}{3}$
1	$\frac{1}{3}$
$\vdots$	

- **A problem with this algorithm is that the decimal or binary representation of a fraction may not terminate.**
- For example $(\frac{1}{3})_{\text{dec}} = 0.333\ldots$ and from the above we see that $(\frac{1}{3})_{\text{bin}} = 0.010101\ldots$
- Hence we need to impose a cutoff on the maximum number of digits to output.

1.2.5 Decimal fraction to hex

Obviously we follow the same procedure but we multiply by 16 instead of 2.

- Example: $a = \frac{1}{4}$

	$\frac{1}{4}$
4	0

- Hence $(\frac{1}{4})_{\text{dec}} = 0.4_{\text{hex}}$ $(= \frac{4}{16})$.

- A more complicated example is $a = 0.11$

	0.11
1	0.76
12	0.16
2	0.56
8	0.96
15	0.36
5	0.76
12	0.16
⋮	

- The expansion repeats and continues indefinitely. We require a lookup table. Hence

$$0.11_{\text{dec}} = (0.1C28F5C28F5\ldots)_{\text{hex}}\,.$$

1.3 Fibonacci numbers

- Leonardo of Pisa (in Italy) was better known as Fibonacci.
- The Fibonacci numbers are defined via the recurrence relation

$$F_n = F_{n-1} + F_{n-2}\,. \tag{1.3.1}$$

- The initial values are $F_1 = 1$ and $F_2 = 1$.
- This obviously lends itself to recursion. A very simple recursive algorithm to calculate F_n can be coded in only a few lines as follows:

```
int Fib_recursive(int n)
{
   if ((n == 1) || (n == 2)) return 1;
   return (Fib_recursive(n-1) + Fib_recursive(n-2));
}
```

- ***** This is terrible *****
- It is wasteful of both memory and time.
- To illustrate, consider how the above algorithm would compute F_6:

$$\begin{aligned} F[6] &= F[5] + F[4] \\ &= (F[4] + F[3]) + (F[3] + F[2]) \\ &= ((F[3] + F[2]) + (F[2] + F[1])) + ((F[2] + F[1]) + F[2]) \\ &= (((F[2] + F[1]) + F[2]) + (F[2] + F[1])) + ((F[2] + F[1]) + F[2])\,. \end{aligned} \tag{1.3.2}$$

- Observe how much duplication of calculation there is and how much waste of memory.
- The computational complexity rapidly gets worse for larger values of n.

- It is better to allocate an array internally and to compute the numbers from $i = 3$ up to n. The code fragment is as follows. (An array F of sufficient size must be allocated. Relevant variables must be declared and initialized.)

```
int i;
F[1] = 1;
F[2] = 1;
for (i = 3; i <= n; ++i) {
   F[i] = F[i-1] + F[i-2];
}
int result = F[n];

(relase allocated memory)

return result;
```

- Notice that there is no duplication of calculations.
- The computation time is $O(n)$ for arbitrary n.
- **Important lesson: The short simple algorithm is not always the best.**

1.4 Horner's rule: nested sums

- In the material below, in general x can be a complex number, but we shall only consider x to be a real number.
- A polynomial is a finite sum of integer powers of x

$$p(x) = a_0 + a_1 x + a_2 x^2 + \cdots + a_n x^n \,. \tag{1.4.1}$$

- We say the above polynomial is of degree n. Obviously the value of a_n must be nonzero. The values of the other coefficients can be zero.
- An efficient way to compute the above polynomial numerically is known as *Horner's rule*.
- We nest the sum as follows.

$$p(x) = a_0 + x(a_1 + x(a_2 + \cdots + x(a_{n-1} + x(a_n)) \ldots) \,. \tag{1.4.2}$$

- This can be coded in a loop as follows

```
sum = a[n];
for (i = n-1; i >= 0; --i) {
  sum = a[i] + x * sum;
}
```

- This loop requires n multiplications and n summations.
- Horner's rule has several nice features.
- If $|x|$ is small (and/or if the magnitudes of the coefficients $|a_i|$ decrease to small values for the high powers if x) then Horner's rule automatically takes care of underflow. As the nested sums are added in the loop, the contributions of the small terms disappear automatically.
- If the coefficients a_i alternate in sign (and/or if the value of x is negative), Horner's rule handles the cancellations better than a brute force summation of large terms of $+ - + - \ldots$ opposing signs.

1.5 Horner's rule: additional ideas

- Many textbooks describe Horner's rule. It optimizes the summation of the powers of x.
- ***But what about the coefficients a_i?***
- By and large, the coefficients are treated as a black box "input array" supplied by the user.
- However, depending on the situation, one can perform additional optimizations.
- Consider the exponential series

$$\exp(x) = \sum_{n=0}^{\infty} \frac{x^n}{n!} = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \cdots \tag{1.5.1}$$

  This is an infinite series. To evaluate it numerucally in practice, we truncate it after a finite number of terms, say N. Then we obtain a polynomial, say

$$p_{\exp}(x) = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \cdots + \frac{x^N}{N!}\,. \tag{1.5.2}$$

- If we apply Horner's rule to this, we obtain

$$p_{\exp}(x) = 1 + x\left(1 + x\left(\frac{1}{2!} + x\left(\frac{1}{3!} + \cdots + x\left(\frac{1}{N!}\right)\ldots\right)\right)\right). \tag{1.5.3}$$

  The coefficients are $a_n = 1/n!$ for $n = 1, 2, \ldots, N$. Their values rapidly grow large as n increases.

- A better way to nest the summation is

$$p_{\exp}(x) = 1 + x\left(1 + \frac{x}{2}\left(1 + \frac{x}{3}\left(1 + \cdots + \frac{x}{N}\right)\ldots\right)\right). \tag{1.5.4}$$

- We not only nest the sums of the powers of x, but we also nest the computation of the coefficients.

***** Examine each problem on its merits *****

****** Use your imagination ******

1.6 Convex hull

- Suppose have a set of n points $\boldsymbol{r}_i = (x_i, y_i)$, $i = 1, \ldots, n$ in a Euclidean plane.
- To avoid complications, we assume all the points (x_i, y_i) are distinct.
- Then there is obviously a "smallest circle" which encloses all the points.
- There is also a "smallest convex polygon" which encloses all the points.
- This "smallest convex polygon" is called the **convex hull** of the set of points.
- The convex hull has many useful applications, which you can look up online.
- We shall employ a method called the **Graham algorithm** to compute the convex hull.
- The Graham algorithm requires $O(n \log(n))$ steps and is very efficient.
    1. First note that if $n = 1$ or $n = 2$, it is impossible to find a convex hull.
    2. If $n = 3$, the points form a triangle, or a straight line. All triangles are convex.
    3. Hence if $n = 3$ we connect the three points and that is our answer.
- Hence we assume $n \geq 4$ below.
- Even for $n \geq 4$, all the points may lie on a straight line, but the algorithm will not fail.
- **The algorithm is also insensitive to finite precision roundoff errors.**
- If $n \geq 4$, the algorithm is as follows:
    1. First we find the point wih the minimum y coordinate. If there is more than one such point, we select the point in that subset with the minimum x coordinate.
    2. Without loss of generality, suppose this point is (x_1, y_1). It will be in the convex hull.
    3. Next, form a set of vectors $\boldsymbol{v}_i = \boldsymbol{r}_i - \boldsymbol{r}_1$ so $(v_{xi}, v_{yi}) = (x_i - x_1, y_i - y_1)$ for $i = 2, \ldots, n$.
    4. Then sort the vectors $\boldsymbol{v}_2 \ldots, \boldsymbol{v}_n$ in *increasing order of their polar angle with the x axis.*
    5. ***This sounds frightening. What does it mean?***
        (a) **Do not panic.**
        (b) In polar coordinates, the angle θ with the x axis is given by $\tan\theta_i = v_{yi}/v_{xi}$.
        (c) Hence $\theta_i = \tan^{-1}(v_{yi}/v_{xi}) = \arctan(v_{yi}/v_{xi})$.
        (d) We want to sort the vectors $\boldsymbol{v}_i$ in increasing order of θ_i.
        (e) If multiple points have equal values of θ, sort them in increasing distance from $\boldsymbol{r}_1$.
    6. Then construct a sorted set of points $\boldsymbol{p}_i$ with $\boldsymbol{p}_1 = \boldsymbol{r}_1$ and
    $$\boldsymbol{p}_i = \boldsymbol{v}_i + \boldsymbol{r}_1 = (v_{xi} + x_1, v_{yi} + y_1) \qquad (i = 2, \ldots, n)\,. \tag{1.6.1}$$
    7. Actually to save memory, $\boldsymbol{p}_i$ and $\boldsymbol{v}_i$ could share the same memory.
    8. It requires n comparisons to find the minimum value of y. It requires $O(n \log(n))$ steps to sort the vectors $\boldsymbol{v}_i$. Hence the overall computational complexity is $O(n \log(n))$.

9. The reason for the above formalism is that we shall construct the convex hull by looping over the points $\boldsymbol{p}_i$ and add points to the convex hull in a counterclockwise sense.
10. We create the convex hull by pushing points onto a stack.
    (a) Push $\boldsymbol{p}_1$ onto the stack. Next push $\boldsymbol{p}_2$ onto the stack.
    (b) **Note that just because we push points onto the stack does not mean they are in the convex hull.**
    (c) We shall loop through all the points and perform tests.
    (d) Depending on the results of the tests, some points may be popped of the stack and new points may be pushed onto the stack.
    (e) **The convex hull will be the final set of points in the stack after we have tested all the points.**
11. Loop through the points $\boldsymbol{p}_i$ for $i = 3, \ldots, n$.
12. For each value of $i = 3, \ldots, n$, perform the following tests:
    (a) Let $\boldsymbol{p}_\ell$ be the **last** point in the stack (top of stack).
    (b) Let $\boldsymbol{p}_s$ be the second last point in the stack.
    (c) Hence initially $s = 1$, $\ell = 2$ and $i = 3$.
    (d) Define the vector $\boldsymbol{u} = \boldsymbol{p}_\ell - \boldsymbol{p}_s$. This is the edge joining $\boldsymbol{p}_s$ to $\boldsymbol{p}_\ell$.
    (e) Define the vector $\boldsymbol{w} = \boldsymbol{p}_i - \boldsymbol{p}_\ell$. This is the edge joining $\boldsymbol{p}_\ell$ to $\boldsymbol{p}_i$.
    (f) If the change of direction from $\boldsymbol{u}$ to $\boldsymbol{w}$ is **counterclockwise** then we are progressing in the desired direction. **Push the point $\boldsymbol{p}_i$ onto the stack. Increment $i := i+1$.**
    (g) If the vectors $\boldsymbol{u}$ and $\boldsymbol{w}$ are **parallel** (= no change of direction), then $\boldsymbol{p}_s$, $\boldsymbol{p}_\ell$ and $\boldsymbol{p}_i$ **lie on a straight line.** The convex hull should therefore go directly from $\boldsymbol{p}_s$ to $\boldsymbol{p}_i$. **Pop the point $\boldsymbol{p}_\ell$ off the stack and push the point $\boldsymbol{p}_i$ onto the stack. No change in length of stack. Increment $i := i + 1$.**
    (h) If the change of direction from $\boldsymbol{u}$ to $\boldsymbol{w}$ is **clockwise** then it means $\boldsymbol{p}_\ell$ is an **interior point and does not belong in the convex hull. Pop the point $\boldsymbol{p}_\ell$ off the stack and repeat the above tests. *** Do not change the value of i. *****
13. There is a computationally easy way to test if the change in direction from $\boldsymbol{u}$ to $\boldsymbol{w}$ is counterclockwise, zero or clockwise. However, the derivation requires mathematical vector calculus (a vector cross product) and we shall skip it. The test is:

$$u_x w_y - u_y w_x = \begin{cases} \text{positive} & \text{counterclockwise} \\ 0 & \text{parallel} \\ \text{negative} & \text{clockwise}\,. \end{cases} \tag{1.6.2}$$

14. Exit the loop after completing the tests for $i = n$.
15. **Push the point $\boldsymbol{p}_1$ onto the stack to create a closed polygon = convex hull.**

1.6.1 Sort: computationally cheaper function #1

- The arctan function is computationally expensive.
- There is a computationally cheaper way to perform the sort.
- Note that, by construction of the "minimum" point, the angles θ_i lie in the interval $0 \le \theta_i < \pi$.
  1. The function $\cos\theta$ is also monotonic in the interval $0 \le \theta < \pi$.
  2. The function $\cos\theta$ **decreases** from 1 to -1 as θ increases from 0 to π.
  3. **Hence we sort the vectors $\boldsymbol{v}_i$ in decreasing order of $\cos(\theta_i)$.**
- The function $\cos\theta_i$ is easy to compute (it requires only a square root):

$$\cos\theta = \frac{v_{xi}}{\sqrt{v_{xi}^2 + v_{yi}^2}}\,. \tag{1.6.3}$$

- Instead of comparing $\theta_i < \theta_j$, we compare $\cos(\theta_i) > \cos(\theta_j)$:

$$\frac{v_{xi}}{\sqrt{v_{xi}^2 + v_{yi}^2}} > \frac{v_{xj}}{\sqrt{v_{xj}^2 + v_{yj}^2}}\,. \tag{1.6.4}$$

- To avoid division by zero, we reformulate the comparison as follows:

$$v_{xi}\sqrt{v_{xj}^2 + v_{yj}^2} > v_{xj}\sqrt{v_{xi}^2 + v_{yi}^2}\,. \tag{1.6.5}$$

- If the two sides are equal in the above comparison, we sort in order of increasing length of $\boldsymbol{v}_i$ and $\boldsymbol{v}_j$:

$$\sqrt{v_{xi}^2 + v_{yi}^2} < \sqrt{v_{xj}^2 + v_{yj}^2}\,. \tag{1.6.6}$$

- **Note that the polar angle θ_i never appears in the calculation, and can be omitted.**

1.6.2 Sort: computationally cheaper function #2

- It was pointed out in class that there is an even better sort function.
- ***I was impressed by it, and I am very pleased with the student.***
- Instead of the cosine, we can employ the cotangent.
- The cotangent decreases monotonically from $+\infty$ to $-\infty$ as θ increases from 0 to π.
- **Hence we sort the vectors v_i in decreasing order of $\cot(\theta_i)$.**
- The cotangent does not require the computation of a square root

$$\cot\theta = \frac{v_{xi}}{v_{yi}}\,. \tag{1.6.7}$$

- Instead of comparing $\theta_i < \theta_j$, we compare $\cot(\theta_i) > \cot(\theta_j)$:

$$\frac{v_{xi}}{v_{yi}} > \frac{v_{xj}}{v_{yj}}\,. \tag{1.6.8}$$

- To avoid division by zero, we reformulate the comparison as follows:

$$v_{xi}\,v_{yj} < v_{xj}\,v_{yi}\,. \tag{1.6.9}$$

- **This is equivalent to the vector cross product $\boldsymbol{v}_i \times \boldsymbol{v}_j$:**

$$v_{xi}\,v_{yj} - v_{xj}\,v_{yi} > 0\,. \tag{1.6.10}$$

- If the left hand side equals zero in the above comparison, that means the vectors $\boldsymbol{v}_i$ and $\boldsymbol{v}_j$ are parallel, and we sort in order of increasing length $|\boldsymbol{v}_i| < |\boldsymbol{v}_j|$.
- **However, if $|\boldsymbol{v}_i| < |\boldsymbol{v}_j|$, then also $|\boldsymbol{v}_i|^2 < |\boldsymbol{v}_j|^2$ hence we do not need a square root:**

$$v_{xi}^2 + v_{yi}^2 < v_{xj}^2 + v_{yj}^2\,. \tag{1.6.11}$$

- We can perform the sort using **only the x and y coordinates.**
  1. We never need θ.
  2. No square root.
- ***This is very good.***

1.6.3 Example

- Here is a set of 12 points in the plane.
- For simplicity all the coordinates are integers or half integers.

i	x_i	y_i
1	7	7
2	3	7
3	1.5	6.5
4	3.5	6
5	8	5.5
6	10	4.5
7	8.5	4
8	4	3
9	1.5	2.5
10	2.5	1.5
11	7	1
12	10	7

- This is the set of the points (x_{xi}, y_{ci}) which form the convex hull of the above points.
  1. The last point is the same as the first, to close the convex polygon.
  2. Note that the points with coordinates $(3, 7)$, $(7, 7)$ and $(10, 7)$ lie on a straight line.
  3. Hence the point at $(7, 7)$ is ***not*** in the convex hull.

i	x_{ci}	y_{ci}
1	7	1
2	10	4.5
3	10	7
4	3	7
5	1.5	6.5
6	1.5	2.5
7	2.5	1.5
8	7	1

- A plot of the points and the convex hull is displayed in Fig. 1.

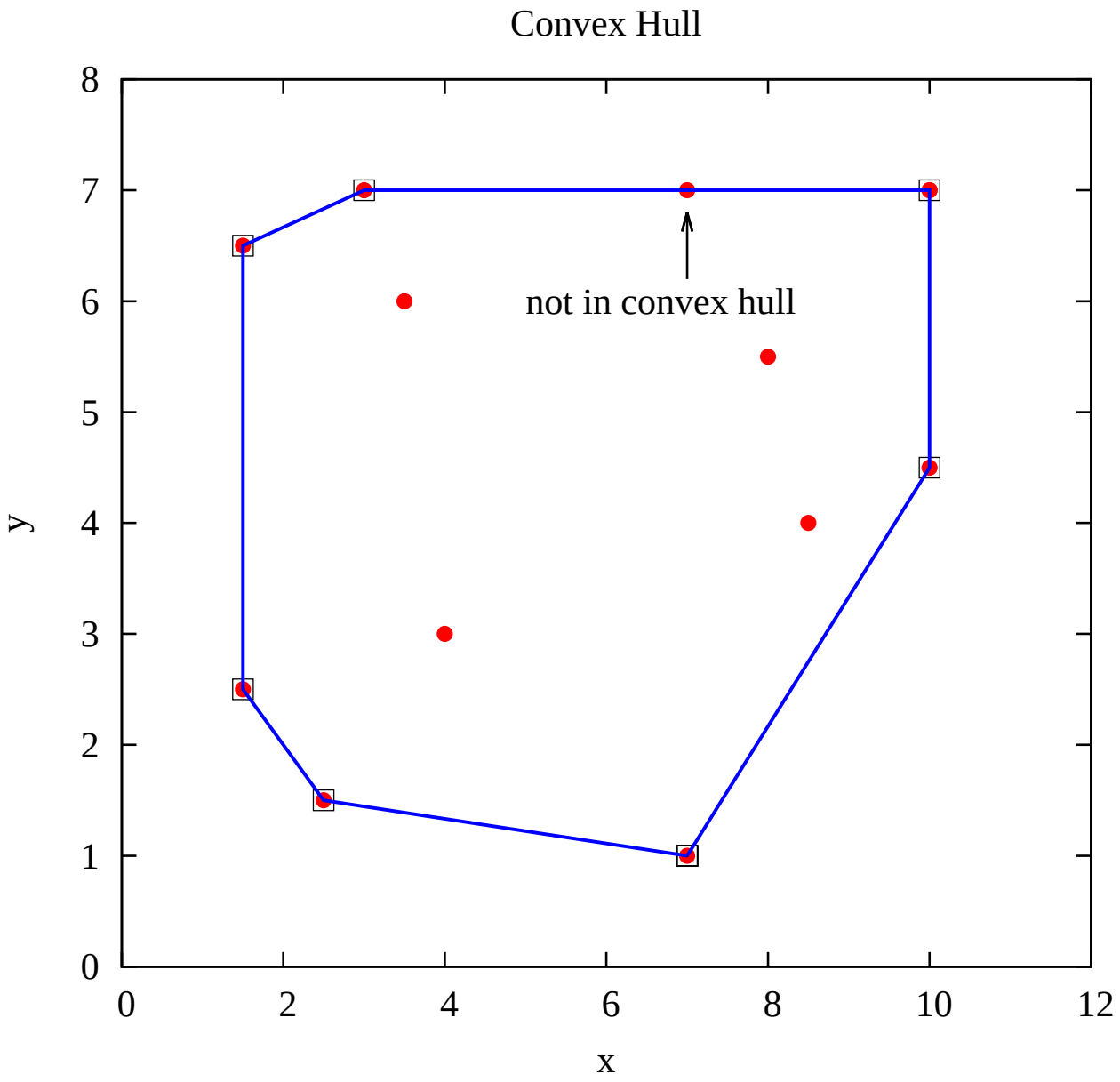

Figure 1: Plot of points in a plane and convex hull, using the algorithm in Section 1.6.

1.7 Karatsuba multiplication (reference only, not for examination)

- Suppose we wish to multiply two numbers (integers), which both have n digits (to keep the analysis simple).
- For many years it was believed that n^2 multiplications of digits was required.
- But in 1960 Karatsuba (who was 23 years old), found a faster way.
- Let the numbers be x and y, and let the base (of the number system) be B. Choose some integer m (the value is up to you). Then we can write

$$x = x_1 B^m + x_0 \,, \qquad y = y_1 B^m + y_0 \,. \tag{1.7.1}$$

  Then the product is

$$xy = (x_1 B^m + x_0)(y_1 B^m + y_0) = x_1 y_1 B^{2m} + (x_1 y_0 + x_0 y_1) B^m + x_0 y_0 \,. \tag{1.7.2}$$

  This requires four multiplications.

- But Karatsuba observed that we can do it in only three multiplications as follows. Compute $z_0 = x_0 y_0$ and $z_2 = x_1 y_1$ and save the values. Then

$$z_1 = x_1 y_0 + x_0 y_1 = (x_1 + x_0)(y_1 + y_0) - z_0 - z_2 \,. \tag{1.7.3}$$

  The last step requires only *one* multiplication, because z_0 and z_2 have been precomputed.

- Karatsuba's algorithm requires only three multiplications, at the cost of a few extra additions.
- For large n, the computational complexity of Karatsuba's algorithm is of order $n^{\log_2 3} \simeq n^{1.585}$, which is faster than $O(n^2)$.
- You can read more about Karatsuba's algorithm online.
- **Important lesson: There is room for improvement even for basic tasks.**

1.8 Subtracting square roots

- Do you know how to accurately calculate the value of $\sqrt{x+1}-\sqrt{x}$ when $x \gg 1$ (say $x = 10^{18}$)?
  1. For $x \gg 1$, the values of $\sqrt{x+1}$ and $\sqrt{x}$ are approximately equal.
  2. The answer will be a small number.
  3. We are subtracting two large numbers to obtain a small result.
  4. This is computationally inaccurate.
- Recall that $(u-v)(u+v) = u^2 - v^2$. Hence set $u = \sqrt{x+1}$ and $v = \sqrt{x}$ to derive

$$\left(\sqrt{x+1} - \sqrt{x}\right)\left(\sqrt{x+1} + \sqrt{x}\right) = \left(\sqrt{x+1}\right)^2 - \left(\sqrt{x}\right)^2 = (x+1) - x = 1\,. \tag{1.8.1}$$

- Therefore

$$\sqrt{x+1} - \sqrt{x} = \frac{1}{\sqrt{x+1} + \sqrt{x}}\,. \tag{1.8.2}$$

- This is good. In the denominator we *add* two positive numbers. There is no cancellation.
- For $x \gg 1$, the values of $\sqrt{x+1}$ and $\sqrt{x}$ may not be easy to distinguish.
- *This does not matter.* Because $\sqrt{x+1} \simeq \sqrt{x}$, we can approximate:

$$\sqrt{x+1} - \sqrt{x} \simeq \frac{1}{2\sqrt{x}}\,. \tag{1.8.3}$$

- Hence for $x = 10^{18}$, the answer is

$$\sqrt{10^{18}+1} - \sqrt{10^{18}} \simeq \frac{1}{2\sqrt{10^{18}}} = \frac{1}{2 \times 10^9} = 5 \times 10^{-10}\,. \tag{1.8.4}$$

- We do not subtract two numbers of $O(10^9)$ to obtain an answer of $O(10^{-10})$.
- **Note: eq. (1.8.3) can also be derived using the binomial theorem.**

1.9 Roots of a quadratic

- Do you know how to calculate the roots of a quadratic equation

$$ax^2 + bx + c = 0\,. \tag{1.9.1}$$

- The formula is

$$x_+ = \frac{-b + \sqrt{b^2 - 4ac}}{2a}\,, \qquad x_- = \frac{-b - \sqrt{b^2 - 4ac}}{2a}\,. \tag{1.9.2}$$

- **What if $|a|$ is very small?**

  1. If $b > 0$ the numerator for x_+ will have a cancellation of two approximately equal numbers, yielding a small numerator (i.e. numerically inaccurate). This will be divided by a small denominator. The value of x_+ may be a number of "moderate size" but it will be computed as a ratio of two small numbers, which is numerically inaccurate.
  2. If $b < 0$ the same remarks will apply to x_-. The numerator for x_- will have a cancellation of two approximately equal numbers, yielding a small numerator (i.e. numerically inaccurate). This will be divided by a small denominator. The value of x_- may be a number of "moderate size" but it will be computed as a ratio of two small numbers, which is numerically inaccurate.

- Rewrite the quadratic as follows (pull out a factor of x^2):

$$a + \frac{b}{x} + \frac{c}{x^2} = 0\,. \tag{1.9.3}$$

- This is a quadratic in $1/x$. The roots are $1/x_+$ and $1/x_-$:

$$\frac{1}{x_+} = \frac{-b - \sqrt{b^2 - 4ac}}{2c}\,, \qquad \frac{1}{x_-} = \frac{-b + \sqrt{b^2 - 4ac}}{2c}\,. \tag{1.9.4}$$

- Invert to obtain

$$x_+ = \frac{2c}{-b - \sqrt{b^2 - 4ac}}\,, \qquad x_- = \frac{2c}{-b + \sqrt{b^2 - 4ac}}\,. \tag{1.9.5}$$

- ***Do you know this?***

- Observe what happens if $|a|$ is very small.

  1. If $b > 0$ the denominator for x_+ will contain a *sum* of two approximately equal numbers, yielding a denominator without problems. The value of x_+ will be computed more accurately.
  2. If $b < 0$ the same remarks will apply to x_-. The denominator for x_- will contain a sum of two approximately equal numbers, yielding a denominator without problems. The value of x_- will be computed more accurately.

- Hence if $|a|$ is very small and $b > 0$, it is better to calculate the roots via

$$x_+ = \frac{2c}{-b - \sqrt{b^2 - 4ac}}\,, \qquad x_- = \frac{-b - \sqrt{b^2 - 4ac}}{2a}\,. \tag{1.9.6}$$

- If $|a|$ is very small and if $b < 0$, it is better to calculate the roots via

$$x_+ = \frac{-b + \sqrt{b^2 - 4ac}}{2a}\,, \qquad x_- = \frac{2c}{-b + \sqrt{b^2 - 4ac}}\,. \tag{1.9.7}$$

- Note that if $|a|$ is very small, one of the roots will be approximately $-b/a$ and will go to $\pm\infty$ as $a \to 0$. This is a mathematical fact and a computer algorithm cannot alter it.