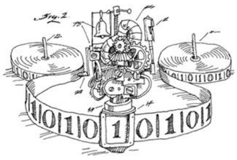
EECS 376: Foundations of Computer Science

Lecture 02 - Potential Method and Divide and Conquer



Agenda

- Runtime analysis of Euclid's algorithm
  - New analysis tool: **Potential method**
- Divide & Conquer algorithmic paradigm
  - Mergesort
  - Master Theorem
  - Karatuba's Integer Multiplication

The Potential Method

Today we will analyze the running time of Euclid's algorithm using the **potential method**.

… But first, a toy example to illustrate this method

A Flipping Game

- 3 x 3 board covered with two-sided chips: M / Ohio State
- Two players, **R (row)** and **C (column)**, alternately perform "flips":
  - **R** flips every chip in a **row** with # Ohio State > # M
  - **C** flips every chip in a **column** with # Ohio State > # M
- If no flip is possible, then the game ends.
- **Question:** Must the game always end?

因而：每次flip, # ohio 一定严格减小

因而一定有end, 并且总局数一定≤9

Let's formalize this reasoning into a general-purpose method

Intuitively, a **potential function argument** says:
If I start with a finite amount of water in a leaky bucket, then eventually water must stop leaking out.

4 steps of the argument:

1. Define **unit of time i** = 0,1,2,…
   (e.g. iteration of algo, recursion depth)
2. Define **potential function Φ(i)** as **non-negative interger**
   (i.e. amount of water in bucket at timestep i)
3. Bound **initial potential Φ(0)**
   (i.e. water is finite)
4. Show **potential decreases Φ(i+1) < Φ(i)**
   (i.e. water is leaking)

- **Conclude:** Bound **the total time in term of Φ(0)** (i.e. water must stop)

因而 $\boxed{\text{potential method}}$ 指：

1. 用一个 set A 来表示 states（比如 unit of time / iteration）
   0, 1, 2, ….
2. φ : A → ℝ 被定义为一个 potential function.
   if (1) 它是 strictly decreasing with states 的
   (2) 它是 bounded below 的
3. 通过这个 def, 我们可以求 number of steps 的 upper bound (O(?)) by : establish φ 的 decrease 速度，从而用 φ(0) 来表示 φ(n)

Analyzing Euclid's Algorithm via a Bad Potential Function

1. Unit of time = one recursive call.
2. Potential function 如果def **Φ(i) = y_i.**
3. 只能保证 We have **Φ(i+1) ≤ Φ(i) − 1.** Why?

So, the total number of calls is at most **Φ(0) = y.**

x mod y : remainder
一定 < y−1
divisor

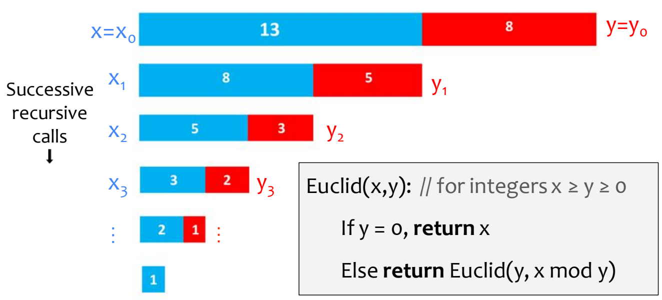
But the runtime bound y = O(10^n) looks bad like before…

因为对于n个 digits 的数，有 O(10^n) 个 1.

What's wrong? Not algorithm. Just need new **Φ** that decreases faster.

Pause and Think:

What is a good potential function?



Euclid(x,y): // for integers x ≥ y ≥ 0
 If y = 0, return x
 Else **return** Euclid(y, x mod y)

Analyzing Euclid's Algorithm via a Potential Function

Finding the right potential function can be a fine art.

1. Unit of time = one recursive call.
2. New potential function $\Phi(i) = x_i+y_i$.
3. **Claim 1:** $\Phi(i+1) \le 3/4\ \Phi(i)$. (will show) O(log)
4. **Claim 2.** Thus: total # recursive calls is O(log (x+y)) = O(n). (will show) digits 位数

Euclid running time = (# recursive calls) × (time to mod of n-digit numbers) — Grade-school algorithm

= O(n) × poly(n) = poly(n)

Analyzing Euclid's Algorithm via a Potential Function

Claim 1. $\Phi(i+1) \le 3/4\ \Phi(i)$.

Idea: The larger number is halved in each call: x → x mod y.

Proof. Show (x mod y) + y ≤ ¾ (x+y) for all integers x ≥ y ≥ 0.

Let's first show: x mod y ≤ x/2.

If x ≥ 2y, then x mod y < y ≤ x/2.

If x < 2y, then x mod y = x-y ≤ x - x/2 = x/2.

- This implies: (x mod y) + y ≤ y + x/2 ≤ ¾ (x+y). (since x≥y)

Optional Challenges:
1. Show $\Phi(i+1) \le 2/3\ \Phi(i)$.
2. Show $\Phi(i+1) \le \varphi\ \Phi(i)$ where φ = 0.618… is the golden ratio

Further proof: $\varphi(i+1) \le \frac{2}{3}\varphi(i)$

Goal: show $x_{i+1} + y_{i+1} \le \frac{2}{3}(x_i+y_i)$

where $x_{i+1} = y_i$, $y_{i+1} = x_i \bmod y_i$

(write $x_i = q_i y_i + r_i$ by div algo, then $y_{i+1} = r_i$,)

因而即证: $y_i + r_i \le \frac{2}{3}(y_i + x_i)$

Pf. $x_i + y_i = q_i y_i + r_i + y_i$

$= \underbrace{(q_i+1)}_{\geq 2} y_i + r_i$

$\geq 2y_i + r_i$ (因为 $y_i \geq r_i + 1$)

$\geq 2y_i + r_i \geq \frac{3}{2}(y_i + r_i)$

⇒ Corollary 12

$\forall i,\ \varphi(i) = x_i + y_i \le (\frac{2}{3})^i (x+y)$ for Euclid (x, y)

Analyzing Euclid's Algorithm via a Potential Function

Claim 2: Total # recursive calls is $1+\log_{4/3}(x+y)$ = O(log(x+y)). → $\frac{3}{2}$

Proof. $\Phi(0) = x+y$,

$\Phi(1) \le (x+y) \cdot (¾)$ → $\frac{3}{2}$ …

$\Phi(i) \le (x+y) \cdot (¾)^i$ → $(\frac{3}{2})^i$

When $i > \log_{4/3}(x+y)$: $(4/3)^i > (4/3)^{\log_{4/3}(x+y)} = (x+y)^{\log_{4/3} 4/3} = x+y$.

So, $\Phi(i) \le (x+y) \cdot (¾)^i < 1$. 即 O(log(x+y))

So, after $1+\log_{4/3}(x+y)$ recursive calls, $\Phi(i) < 1$.
- So $\Phi(i) = 0$ as $\Phi(i)$ is always an integer,
- At this point the algorithm terminates.

Thm 13. Euclid (x, y) perform O(log(x+y)) iterations.

Analyzing Euclid's Algorithm via a Potential Function

Finding the right potential function can be a fine art.

1. Unit of time = one recursive call.
2. New potential function $\Phi(i) = x_i+y_i$.
3. ✓ **Claim 1:** $\Phi(i+1) \le 3/4\ \Phi(i)$. (will show)
4. ✓ **Claim 2.** Thus: total # recursive calls is O(log (x+y)) = O(n). (will show)

(later) Grade-school algorithm

Euclid running time = (# recursive calls) × (time to mod of n-digit numbers)

= O(n) × poly(n) = poly(n)

(2)

Next:

Introduction to Divide and Conquer

Overview: Divide-and-Conquer Algorithms

Main Idea:
1. **Divide** the problem into smaller sub-problems (creative step)
2. **Conquer** (solve) each sub-problem *recursively* (easy step)
3. **Combine** the solutions (creative step)

https://www.youtube.com/watch?v=ZRPoEKHXTJg

Mergesort

Input: Array of numbers

1	2	3	4					n
6	0	4	6	…	…	…	…	3

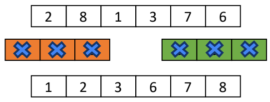
Output: Sorted

1	2	3	4					
0	3	4	6	6	…	…	…	…

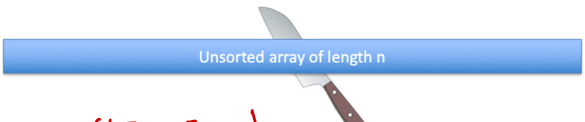
Discovered by John von Neumann in 1945

Code and Example

```
MergeSort(A[1..n]):        // sorts a list of integers
if n = 1 then return A     // base case
L = MergeSort(A[1..n/2])   // recursively sort 1st half
R = MergeSort(A[n/2+1..n]) // recursively sort 2nd half
return Merge(L, R)         // combine solutions
```



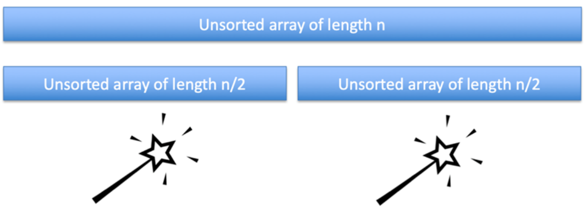
Mergesort



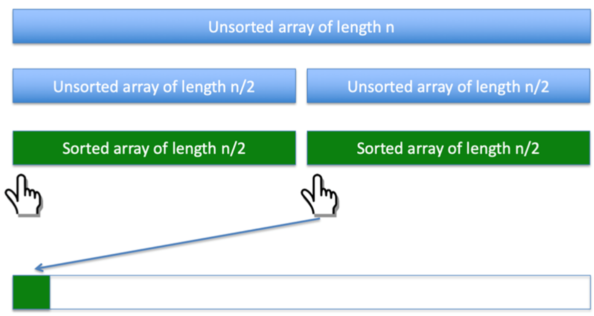
function Merge(L[1,...,l],
R[1,...,r])
if l=0 then return R
if r=0 then return L
if L[1] ≤ R[1] then
return L[1]: Merge(L[2,..,l],
R[1,..,r])
else
return R[1]. Merge(L[1,..,l], R[2,..,r])

function Mergesort (A[1,...,n])
if n=1 then return A
m = ⌊n/2⌋
L = Mergesort (A[1,...,m])
R = Mergesort (A[m+1,...,n])
return Merge (A,B)

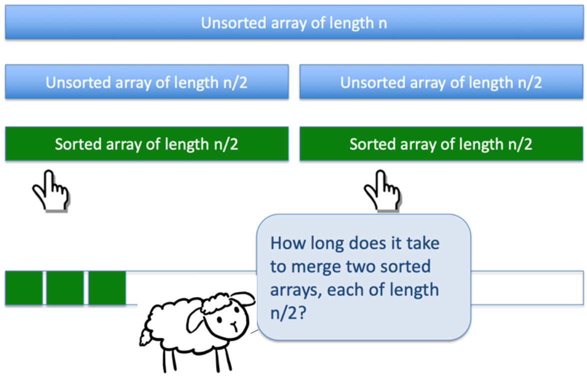
Mergesort



Mergesort

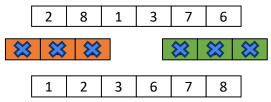


Mergesort



Correctness

- Strong induction on size of list, n.
- Base case:
  - **MS** is correct on lists of size 1.
- Inductive step:
  - Suppose **MS** is correct on lists of size $< n$.
  - Then **MS** is correct on 1st /2nd half, by assumption.
  - Since **Merge** is correct, **MS** is correct on n.



Recurrence of Running Time

Let $T(n)$ be worst-case running time on input of size n

$$\underbrace{T(n)}_{\substack{\text{time to MS} \\ \text{a list of } n \\ \text{integers}}} = \begin{cases} O(1) & n = 1 \\ \underbrace{T\left(\frac{n}{2}\right)}_{\substack{\text{time to MS} \\ 1^{st} \text{ half}}} + \underbrace{T\left(\frac{n}{2}\right)}_{\substack{\text{time to MS} \\ 2^{nd} \text{ half}}} + \underbrace{O(n)}_{\substack{\text{time to} \\ \text{Merge}}} & n > 1 \end{cases}$$

Note: we typically omit the base case

```
MergeSort(A[1..n]):        // sorts a list of integers
if n = 1 then return A     // base case
L = MergeSort(A[1..n/2])   // recursively sort 1st half
R = MergeSort(A[n/2+1..n]) // recursively sort 2nd half
return Merge(L, R)         // combine solutions
```

How do we solve this recurrence?

The Master Theorem

Master Theorem
(Runtime of Divide and Conquer Algorithms)

- Given an input of size n, an algorithm
  - makes k recursive calls,
  - Each on an input of size n/b, and
  - then "combines" the results in $O(n^d)$ time.
- Let $T(n)$ be the runtime of the algorithm on inputs of size n.

- **Theorem:**

  if $T(n) = kT(n/b) + O(n^d)$

  then,

$$T(n) = \begin{cases} O(n^d) & \text{if } k < b^d \\ O(n^d \log n) & \text{if } k = b^d \\ O(n^{\log_b k}) & \text{if } k > b^d \end{cases}$$

Will prove the theorem if time permitted

Example: MergeSort

```
MergeSort(A[1..n]): // sorts a list of integers
if n = 1 then return A                  // base case
L = MergeSort(A[1..n/2])                // recursively sort 1st half
R = MergeSort(A[n/2+1..n])              // recursively sort 2nd half
return Merge(L, R)                      // combine solutions
```

- On an input of size n, the **MergeSort** algorithm makes
  - $k = \boxed{2}$ recursive calls,
  - each on an input of size $n/b = \boxed{n/2}$,
  - and then spends $O(n^d) = \boxed{O(n^1)}$ time "combining" the results.
- So,

子问题分割大小比

$$T(n) = kT(n/b) + \underline{O(n^d)} = \begin{cases} O(n^d) & \text{if } k < b^d \\ \boxed{O(n^d \log n) \quad \text{if } k = b^d} \\ O(n^{\log_b k}) & \text{if } k > b^d \end{cases}$$

分为子问题数量　　recurr step effect

∴The runtime of **MergeSort** is $O(n \log n)$.

(Another example of divide and conquer)

Karatsuba's integer multiplication

General Goal: Fast Integer Arithmetic

- **Goal**:
  - implement basic arithmetic operations, e.g., $+, -, *, /, \ll$, etc
  - on **big integers** with a ***non-constant*** **number of digits**
- Many programming languages support this.
- **Want**: fast algo in term of the input size ($n = \#$ digits)?

Integer Addition

- Given n-digit integers x and y
- **Goal:** compute $x + y$ and $x - y$
- **Easy:** add digits one at a time and keep a "carry" digit
- **Q:** What's the runtime?
  - $O(n)$. Nice!

	1	1	1	
		9	4	6
+		9	8	5
	1	9	3	1

Today's Goal: Integer Multiplication

每位 × 每位 ⇒ $O(n^2)$

- Given n-digit *positive* integers x and y
- **Goal:** compute $x * y$
- **Easy:** do "grade-school" method
- **Q:** What's the runtime?
  - $O(n^2)$. Yikes!

a

$b_n \; b_{n-1} \cdots b_1$

$\ll n \quad \ll n-1$

$+ \quad + \quad \ldots\ldots$

			3	4
*			3	9
		3	0	6
	1	0	2	
	1	3	2	6

						1	2	3	4	5
×						5	4	3	2	1
+						1	2	3	4	5
+					2	4	6	9	0	
+				3	7	0	3	5		
+			4	9	3	8	0			
+		6	1	7	2	5				
=		6	7	0	5	9	2	7	4	5

Splitting a Number

- Let's try to apply Divide & Conquer approach for Multiplication.
- **Starting point**: we can "divide" number…
- $376280 = 376 \cdot 10^3 + 280$
- **Observation 1:** N an n-digit number (assume n is even)
- N can be split into $n/2$ low-order digits & $n/2$ high-order digits:
  - $N = a \cdot 10^{\frac{n}{2}} + b$

	←$n/2$ digits→	←$n/2$ digits→
N	a	b

Divide and Conquer Multiplication

- **Input:** x and y, two n-digit numbers (assume n is a power of 2)
- Split x and y into $n/2$ low-order digits & $n/2$ high-order digits:
  - $x = a \cdot 10^{n/2} + b$
  - $y = c \cdot 10^{n/2} + d$

	←$n/2$ digits→	←$n/2$ digits→
x	a	b
y	c	d

- Compute $x \times y = a \times c \cdot 10^n + (a \times d + b \times c) \cdot 10^{n/2} + b \times d$

Divide and Conquer?

Mult(x, y): // x, y are n-digit positive integers
if $n = 1$ then return $x \cdot y$ // base case; hard-code
$(a, b) \leftarrow$ split digits of x into halves // $x = a \cdot 10^{n/2} + b$
$(c, d) \leftarrow$ split digits of y into halves // $y = c \cdot 10^{n/2} + d$
$t_1 \leftarrow$ **Mult**(a, c) // $= ac$
$t_2 \leftarrow$ **Mult**(a, d) + **Mult**(b, c) // $= ad + bc$
$t_3 \leftarrow$ **Mult**(b, d) // $= bd$
return $(t_1 \ll n) + (t_2 \ll n/2) + t_3$

x: 1 2 y: 2 3
a: 1 b: 2 c: 2 d: 3
t_1: 2 t_2: 7 t_3: 6
return: 2 7 6

Analysis

- **Correctness:** Clear
  - **Input:** x and y
  - We correctly compute

$$x \times y = a \times c \cdot 10^n + (a \times d + b \times c) \cdot 10^{\frac{n}{2}} + b \times d$$

where $x = a \cdot 10^{n/2} + b$
$y = c \cdot 10^{n/2} + d$

- **Runtime:**
  - 4 (recursive) multiplications of $n/2$-digit numbers
  - 2 left shifts ($O(n)$ time)
  - 3 additions ($O(n)$ time)

$$T(n) = \begin{cases} O(n^d) & \text{if } (k/b^d) < 1 \\ O(n^d \log n) & \text{if } (k/b^d) = 1 \\ O(n^{\log_b k}) & \text{if } (k/b^d) > 1 \end{cases}$$

- $T(n)$ = time to multiply two n-digit numbers
  - $T(n) = 4T(n/2) + O(n)$. So $k = 4, b = 2, d = 1 \Longrightarrow k/b^d = 2 > 1$
  - **Conclusion:** $T(n) = O(n^{\log_2 4}) = O(n^2)$.

Divide and Conquer Multiplication

Conclusion:
- Simple, well-known long-multiplication algorithm: $O(n^2)$
- Complicated and scary Divide and Conquer algorithm: $O(n^2)$

(Earlier, **Gauss** used the same trick in a different context)

Karatsuba's idea!

$O(n^2)$

Around 1956, the famous Soviet mathematician Andrey Kolmogorov conjectured that this is the *best possible way* to multiply two numbers together.

Just a few years later, Kolmogorov's conjecture was shown to be spectacularly wrong.

In 1960, Anatoly Karatsuba, a 23-year-old mathematics student in Russia, discovered a sneaky algebraic trick that reduces the number of multiplications needed.

We can multiply using 3 recursive calls, not 4!

$O(n^{\log_2 3})$

A Neat Trick

Previous slow algo

Mult(x, y): // x, y are n-digit positive integers
... // split x, y
$t_1 \leftarrow$ **Mult**(a, c) // $= ac$
$t_2 \leftarrow$ **Mult**(a, d) + **Mult**(b, c) // $= ad + bc$
$t_3 \leftarrow$ **Mult**(b, d) // $= bd$
return $(t_1 \ll n) + (t_2 \ll n/2) + t_3$

- Let's stare at this identity again:

$$xy = (a \cdot 10^{n/2} + b)(c \cdot 10^{n/2} + d) = ac \cdot 10^n + (ad + bc) \cdot 10^{n/2} + bd$$

- **Think:**
  - Could we write $ad + bc$ in terms of ac (t_1), bd (t_3),
  - and *something else* that only uses ***one multiplication*** (not two)?

$$ad + bc = (a + b)(c + d) - ac - bd$$

- **So:** can compute $t_2 = ad + bc$ as $(a + b)(c + d) - t_1 - t_3$, using only a ***one recursive call*** to **Mult** (not two)!

Karatsuba's Algorithm

Karatsuba(x, y): // x, y are n-digit positive integers
if $n = 1$ then return $x \cdot y$ // base case; hard-code
$(a, b) \leftarrow$ split digits of x into halves // $x = a \cdot 10^{n/2} + b$
$(c, d) \leftarrow$ split digits of y into halves // $y = c \cdot 10^{n/2} + d$
$t_1 \leftarrow$ **Karatsuba**(a, c) // $= ac$
$t_4 \leftarrow$ **Karatsuba**$(a + b, c + d)$ // $= (a + b)(c + d)$
$t_3 \leftarrow$ **Karatsuba**(b, d) // $= bd$
$t_2 \leftarrow t_4 - t_1 - t_3$ // $= ad + bc$
return $(t_1 \ll n) + (t_2 \ll n/2) + t_3$

Next: The runtime of **Karatsuba** is $O(n^{1.585})$.

Example: Karatsuba

- On an input of size n, the **Mult** algorithm makes
  - $k = \boxed{3}$ recursive calls,
  - each on an input of size $n/b = \boxed{n/2}$, and then
  - spends $O(n^d) = \boxed{O(n^1)}$ time "combining" the results.
- Let $T(n)$ be the runtime of the algorithm on inputs of size n.
- Then we can write:

$$T(n) = kT(n/b) + O(n^d) = \begin{cases} O(n^d) & \text{if } k < b^d \\ O(n^d \log n) & \text{if } k = b^d \\ \boxed{O(n^{\log_b k}) \quad \text{if } k > b^d} \end{cases}$$

∴The runtime of **Mult** is $O(n^{\log_2 3})$.

Question: It is possible to do even better than Karatsuba multiplication?
Answer: Yes - the best known result is O(n log n) by Harvey and van der Hoeven. It's from 2019!

Unfortunately, the hidden constants are *enormous*:
"…the proof given in our paper only works for ludicrously large numbers. Even if each digit was written on a hydrogen atom, there would not be nearly enough room available in the observable universe to write them down." – David Harvey

Open problem: Can this be improved to O(n)?
Conjecture: No (but we don't know—maybe possible!)

History

- 1960: **Kolmogorov** conjectured "you need $\Omega(n^2)$ ops."
- Within a week: **Karatsuba** $O\left(n^{\log_2 3}\right) = O(n^{1.58})$

- 1971: **Schönhage**, **Strassen** $O(n \log n \log\log n)$
- 2007: **Fürer** $O(n \log n \, 2^{O(\log^* n)})$
  and some more works after…
- 2019: **Harvey**, **Hoeven** $O(n \log n)$

Based on Fast Fourier Transform (take EECS 477)

- 2019: **Afshani et al.** $\Omega(n \log n)$ assuming the **network coding conjecture!?**

$< n^{1.0001}$

$\emptyset$

Surprising connection!

Upshot: Divide-and-Conquer Algorithms

Main Idea:

1. **Divide** the problem into smaller sub-problems (creative step)
2. **Conquer** (solve) each sub-problem *recursively* (easy step)
3. **Combine** the solutions (creative step)

Designing the Algorithm + Proving Correctness: an "art"

- Depends on problem structure, ad-hoc, creative

Running time Analysis: "mechanical"

- Express runtime using a recurrence
- Can often solve using the "Master Theorem"