Analysis of Algorithms, I
CSOR W4231

Eleni Drinea
Computer Science Department

Columbia University

More divide & conquer algorithms: fast int/matrix multiplication

Outline

Today

Review of the last lecture

In the last lecture we discussed

- Asymptotic notation $(O, \Omega, \Theta, o, \omega)$
- The divide & conquer principle
  - **Divide** the problem into a number of subproblems that are smaller instances of the same problem.
  - **Conquer** the subproblems by solving them recursively.
  - **Combine** the solutions to the subproblems into the solution for the original problem.
- Application: `mergesort`
- Solving recurrences

mergesort

```
mergesort (A, left, right)
    if right == left  then
        return
    end if
    mid = left + ⌊(right − left)/2⌋
    mergesort (A, left, mid)
    mergesort (A, mid + 1, right)
    merge (A, left, right, mid)
```

- Initial call: `mergesort`$(A, 1, n)$
- Subroutine `merge` merges two sorted lists of sizes $\lceil n/2 \rceil$, $\lfloor n/2 \rfloor$ into one sorted list of size n in time $\Theta(n)$.

Running time of `mergesort`

The running time of `mergesort` satisfies:

$$T(n) = 2T(n/2) + cn, \text{ for } n \geq 2, \text{ constant } c > 0$$
$$T(1) = c$$

This structure is typical of recurrence relations:

- an *inequality* or *equation* bounds $T(n)$ in terms of an expression involving $T(m)$ for $m < n$
- a base case generally says that $T(n)$ is constant for small constant n

Remarks

- We ignore floor and ceiling notations – Because they do not effect the asymptotic bound.
- A recurrence does **not** provide an asymptotic bound for $T(n)$: to this end, we must solve the recurrence

We will be interested in solving these recurrence relations to get asymptotic bounds on $T(n)$.

Solving recurrences, method 1: recursion trees

This is just one of several methods for solving recursions. The textbook shows other approaches (e.g., the substitution method).

The technique consists of three steps

1. Analyze the first few levels of the tree of recursive calls
2. Identify a pattern
3. Sum over all levels of recursion

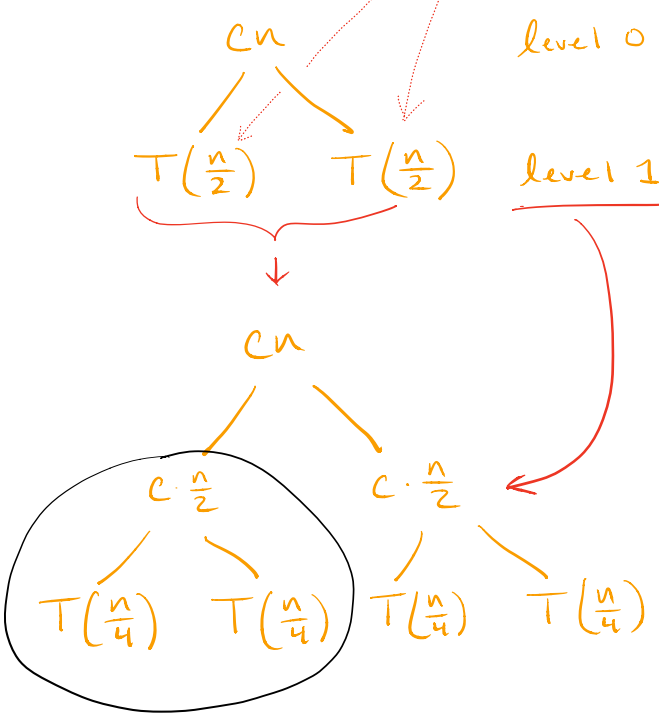
Example: analysis of running time of `mergesort`

$T(n) = 2T(n/2) + cn, n \geq 2$

$T(1) = c$

$$T(n) = 2T(n/2) + cn,$$
$$T(1) = c$$

$T(\frac{n}{2}) = 2T(\frac{n}{4}) + c \cdot \frac{n}{2}$



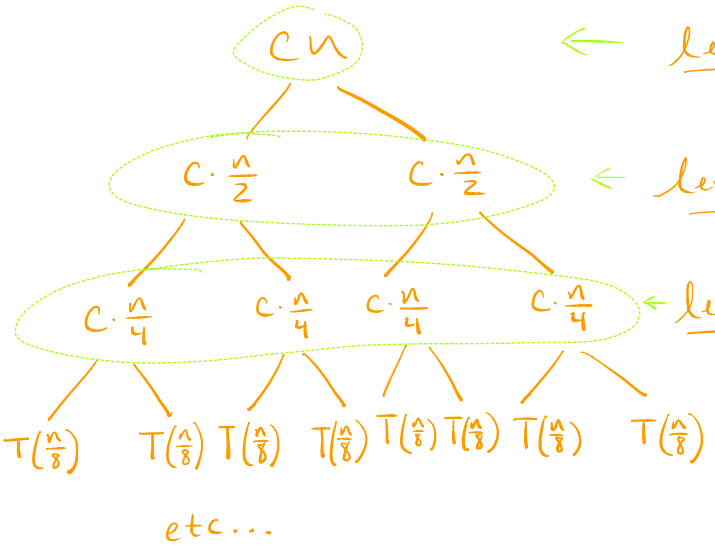
level 0 : 1 subproblem, size: n, total work: cn

level 1 : 2 subproblems, each of size: $\frac{n}{2}$, Total work: $c \cdot \frac{n}{2} + c \cdot \frac{n}{2} = cn$

→ Keep expanding in this way and you are eventually left with only expressions that are in terms of n.

$T(\frac{n}{4}) = 2T(\frac{n}{8}) + c \cdot \frac{n}{4}$

Then,

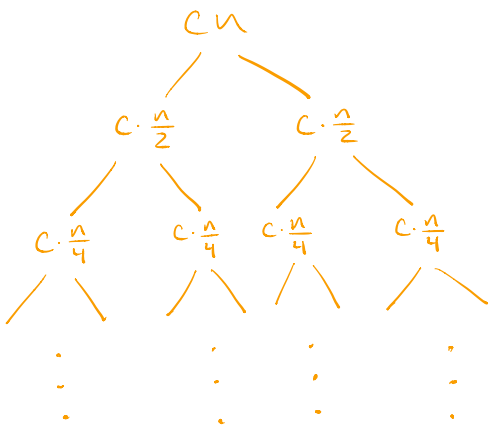


← level 0 : 1 subproblem, size: n, total work: cn

← level 1 : 2 subproblems, each: size n/2, total work: cn

← level 2 : 4 subproblems, each: size n/4, total work: cn

⋮



$\underbrace{T(1)}_{=c} \cdots \underbrace{T(1)}_{=c} \cdots\cdots \underbrace{T(1)}_{=c}$

Size of each subproblem is $\frac{n}{2^d} = 1 \Rightarrow d = \log_2 n$

So the depth of this tree is $\log_2 n$

level i: 2^i subproblems

each: size $n/2^i$

total work: cn

$\left(= 2^i \cdot c \cdot \frac{n}{2^i} = cn\right)$

The tree will eventually bottom out at some level d. It bottoms out because at the bottom level each subproblem will have size 1. At that point, mergesort will just terminate.

We reach level d when the size of the subproblem is 1.

---

level d: 2^d subproblems

each size: $n/2^d = 1$

$\Rightarrow n = 2^d$

$\Rightarrow \log_2 n = \log_2 2^d$

$\Leftrightarrow d = \log_2 n$

$T(n)$ is the sum of ALL the work spent on every level of the tree:

$$T(n) = \sum_{i=0}^{d} cn = (d+1) \cdot cn$$

$$= (\log_2 n + 1) \cdot cn$$

$$= \Theta(n \log n)$$

Therefore, the running time of mergesort is $\Theta(n \log n)$.

A frequently occurring recurrence and its solution

The running time of many recursive algorithms is given by

$$T(n) = aT\left(\frac{n}{b}\right) + cn^k, \qquad \text{for } a, c > 0,\ b > 1,\ k \geq 0$$

each subproblem has size n/b

combining phase takes time some polynomial in n.

divide & conquer algorithm is generating a subproblems.

What is the recursion tree for this recurrence?

- a is the branching factor
- b is the factor by which the size of each subproblem shrinks

$\Rightarrow$ at level i, there are a^i subproblems, each of size n/b^i

$\Rightarrow$ each subproblem at level i requires $c(n/b^i)^k$ work

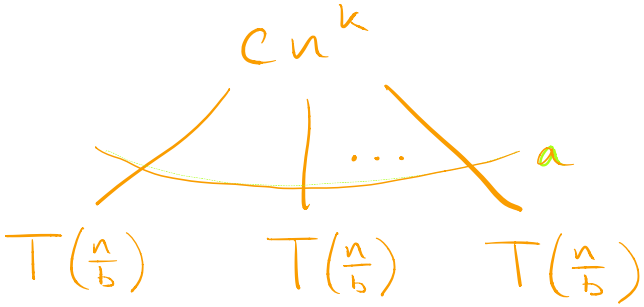
- the height of the tree is $\log_b n$ levels

$\Rightarrow$ Total work: $\sum_{i=0}^{\log_b n} a^i c(n/b^i)^k = cn^k \sum_{i=0}^{\log_b n} \left(\frac{a}{b^k}\right)^i$

$$T(n) = aT\left(\frac{n}{b}\right) + cn^k,$$

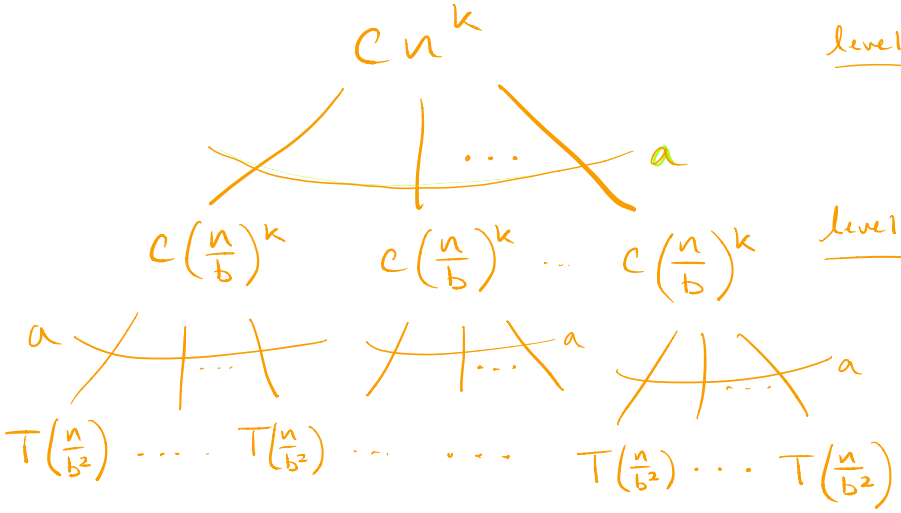
$T\left(\frac{n}{b}\right) = a\,T\left(\frac{n}{b^2}\right) + c\left(\frac{n}{b}\right)^k$

$T\left(\frac{n}{b^2}\right) = a\,T\left(\frac{n}{b^3}\right) + c\left(\frac{n}{b^2}\right)^k$



level 0: 1 subproblem
size: n
total work: cn^k

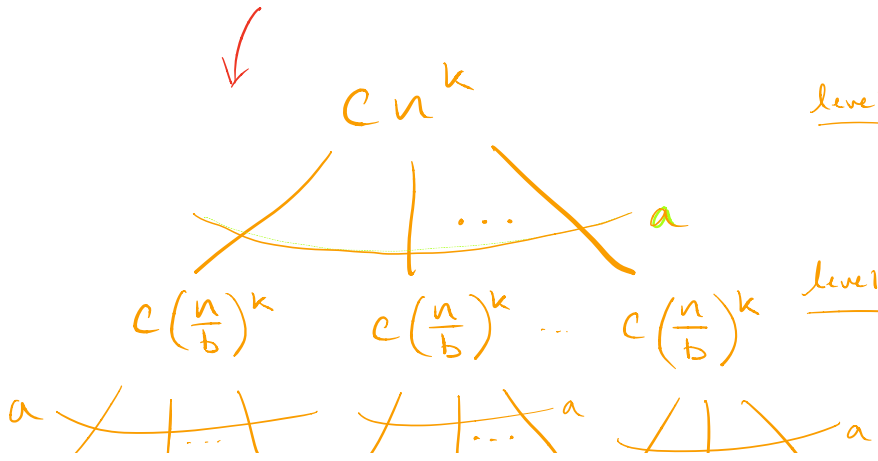
level 1: a subproblems



level 0: 1 subproblem
size: n
total work: cn^k

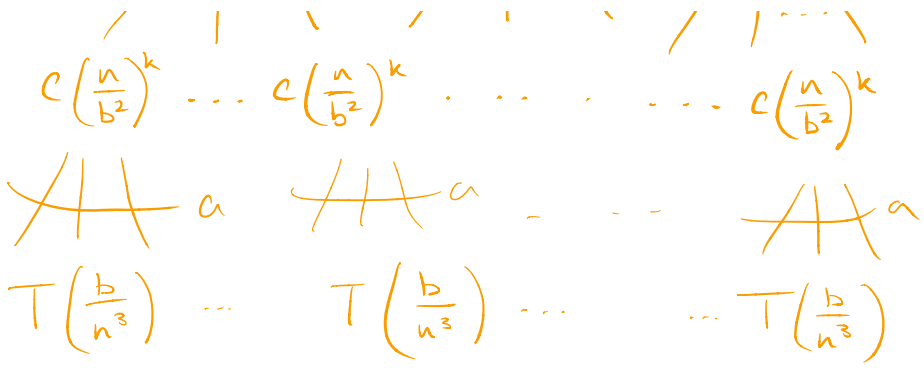
level 1: a subproblems
each: size $\frac{n}{b}$
total work: $a \cdot c\left(\frac{n}{b}\right)^k$

[Indeed, we spend $c\cdot\left(\frac{n}{b}\right)^k$ per subproblem, and there are 'a' of them.]



level 0: 1 subproblem
size: n
total work: cn^k

level 1: a subproblems
each: size $\frac{n}{b}$
total work: $a \cdot c\left(\frac{n}{b}\right)^k$



level 2: a^2 subproblems

each: size $\frac{n}{b^2}$

total work: $a^2 \cdot c\left(\frac{n}{b^2}\right)^k$

⋮

level i: a^i subproblems

each: size $\frac{n}{b^i}$

total work: $a^i \cdot c\left(\frac{n}{b^i}\right)^k$

Bottoms out at some level d, where each subproblem is of size

$$\frac{n}{b^d} = 1 \Rightarrow d = \log_b n$$

Tree bottoms out at some level d, where each subproblem is of size 1.

level d: a^d subproblems

each: size $n/b^d = 1$

$\Rightarrow d = \log_b n$

total work: $T(n)$

$$T(n) = \sum_{i=0}^{d} a^i \cdot c \cdot \left(\frac{n}{b^i}\right)^k = cn^k \cdot \sum_{i=0}^{\log_b n} \left(\frac{a}{b^k}\right)^i$$

$$\Leftrightarrow \boxed{T(n) = cn^k \cdot \sum_{i=0}^{\log_b n} \left(\frac{a}{b^k}\right)^i}$$

$i = 0$

- If $a = b^k \Rightarrow T(n) = (\log_b n + 1) \cdot cn^k$

$$= O(n^k \log_b n)$$

Don't need to keep $\log_b(\cdot)$ in base b notation (Why?)

$$= O(n^k \log n)$$

- If $a < b^k$ (In Homework) $\Rightarrow \cdots \sum_{i=0}^{\log_b n} \left(\frac{a}{b^k}\right)^i = O(1)$

$$\Rightarrow T(n) = O(n^k)$$

- If $a > b^k \Rightarrow \cdots \sum \left(\frac{a}{b^k}\right)^i = O\left(\left(\frac{a}{b^k}\right) \log_b n\right)$

$$= O\left(\frac{a^{\log_b n}}{b^{k \log_b n}}\right)$$

$$= O\left(\frac{n^{\log_b a}}{n^k}\right)$$

$$\Rightarrow T(n) = O(n^{\log_b a})$$

Recall / Review:

$$\log_b n = \frac{\log_2 n}{\log_2 b}$$

and

$$\frac{\log_2 n}{\log_2 b} > \log_2 n$$

but only by a constant factor.

Solving recurrences, method 2: Master theorem

Theorem 1 (Master theorem).

If $T(n) = aT(\lceil n/b \rceil) + O(n^k)$ *for some constants* $a > 0$, $b > 1$, $k \geq 0$, *then*

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

So from now on if we have an algorithm which can be expressed in this form, we can just use the Master Theorem to analyze the running time.

Example: running time of `mergesort`

- $T(n) = 2T(n/2) + cn$:
  $a = 2, b = 2, k = 1, b^k = 2 = a \Rightarrow T(n) = O(n \log n)$

Today

Searching a sorted array

- **Input:**
  1. sorted list A of n integers; → Use an array
  2. integer x
- **Output:**
  - index j such that $1 \leq j \leq n$ and $A[j] = x$; *or*
  - **no** if x is not in A

How do we solve this problem?

Binary Search

- 1st locate the mid-point
- 2nd compare the integer x with the item stored at the mid-point. If $x > mid$, then search in the right half of the array.

Searching a sorted array

- **Input:**
  1. sorted list A of n integers;
  2. integer x
- **Output:**
  - index j such that $1 \leq j \leq n$ and $A[j] = x$; *or*
  - **no** if x is not in A

Example: $A = \{0, 2, 3, 5, 6, 7, 9, 11, 13\}$, $n = 9$, $x = 7$

Searching a sorted array

- **Input:**
  1. sorted list A of n integers;
  2. integer x
- **Output:**
  - index j such that $1 \leq j \leq n$ and $A[j] = x$; *or*
  - **no** if x is not in A

Example: $A = \{0, 2, 3, 5, 6, 7, 9, 11, 13\}$, $n = 9$, $x = 7$

Idea: use the fact that the array is sorted and probe specific entries in the array.

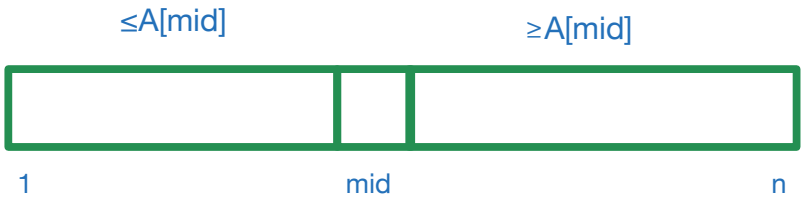
probe - physically explore or examine (something)

Binary search

First, probe the middle entry. Let $mid = \lceil n/2 \rceil$.

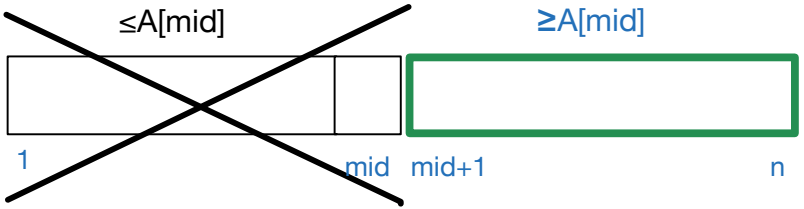
- If $x == A[mid]$, return mid.
- If $x < A[mid]$ then look for x in $A[1, mid - 1]$;
- Else if $x > A[mid]$ look for x in $A[mid + 1, n]$.

Initially, the entire array is “active”, that is, x might be anywhere in the array.



Suppose x > A[mid].

Then the active area of the array, where x might be, is to the right of mid.



Binary search pseudocode

You may assume X is a global variable or you can pass it within the binarysearch alg./function/environment.

```
binarysearch(A, left, right)
O(1) → mid = left + ⌈(right − left)/2⌉
O(1) → if x == A[mid] then
           return mid
O(1) → else if right == left then
           return no
O(1) → else if x > A[mid] then
           left = mid + 1
O(1) → else right = mid − 1
       end if
       binarysearch(A, left, right)
```

Assume X is a global variable in your overall code.

binarysearch has 2 inputs: i) the array
ii) the integer X you are looking for in the array.

Initial call: `binarysearch`$(A, 1, n)$

* Prove correctness by induction on the input size (Exercise @ Home).

Running time of bin search: $\log n$ (show using Master Thm.)

Binary search pseudocode

```
binarysearch(A, left, right)
    mid = left + ⌈(right − left)/2⌉
    if x == A[mid] then
        return mid
    else if right == left then
        return no
    else if x > A[mid] then
        left = mid + 1
    else right = mid − 1
    end if
    binarysearch(A, left, right)
```

constant time.

Initial call: `binarysearch`$(A, 1, n)$

* let $T(n)$ = time for binary search on input size n.

$$T(n) = c + T\left(\frac{n}{2}\right)$$

$a = 1$, $b = 2$

$k = 0$

$\Rightarrow a = b^k$

Therefore, by the Master Theorem:

$$T(n) = O(\log n)$$

$a \neq b$

You are generating a diff. # of subproblems than the factor in which you are shrinking.

Is the $O(\log n)$ running time for binary search a significant improvement from a linear search $O(n)$ algorithm?

Yes, definitely.

$$O(\log n) = o(n) \quad \leftarrow \text{little-o}$$

Suppose $n = 1000000$, $1000 < 2^{10}$

$$\Rightarrow \quad 10^6 < 2^{10} 2^{10} = 2^{20}$$

$$\text{binary search run time} < \log_2 2^{10} 2^{10} = \log_2 2^{20} = 20$$

- Significant improvement compared to linear search.

We don't even need to throw out $\frac{1}{2}$ of our input to achieve a significant improvement. We could throw out just $\frac{1}{3}$ of our input and still get a better run time compared to linear search:

$$n \to \frac{2n}{3} \to \left(\frac{2}{3}\right)^2 n \to \cdots \to \left(\frac{2}{3}\right)^d n \Bigg\} \Rightarrow O(\log n)$$

- After the 1st comparison/probe your input size shrinks by $\frac{1}{2}$: $n/2$
- After the 2nd comparison : $n/4$
- After the 3rd probe : $n/8$
- ⋮ ⋮
- After the k^{th} probe : $n/2^k$

If the input size is small just solve using brute force approach. because that will just require constant time.

 $\longrightarrow O(1)$

How many probes do we need before the input size becomes 1 : $\boxed{\frac{n}{2^k} = 1}$ $\Longrightarrow$ $n = 2^k \Rightarrow \boxed{k = \log_2 n}$

Need approx. $k = \log_2 n$ probes before the input size is small enough to search for the item in constant time in a a constant-size array (using brute force method on small input size arrays.

- You need $\log n$ recursive calls, then you spend constant time to find the item. How much time do you need to shrink the input size by half?

Answer: Constant time.

Therefore,

1st Probe : $n/2$
2nd Probe : $n/4$
⋮ ⋮
k^{th} Probe : $n/2^k$

constant time to reduce input size by half

k iterations and $k = \log_2 n$.

constant time to search small arrays.

$$c \cdot \log_2 n + O(1) = O(\log n)$$

$$\underbrace{c + c + \cdots + c}_{k \text{ - times}} \quad (k = \log_2 n)$$

($\log_2 n$ times)

$$\Rightarrow c \cdot \log_2 n$$

Need to use an array to store original set of n integers because we need random access in order to be able to make the comparison in constant time.

If you used a linked list instead c becomes $c \cdot n$.

i.e. $T(n) = c + T(n/2)$.

To access an element in a linked list it takes time proportional to the position of the element in the linked list.

Using a linked list changes the run-time to $T(n) = c \cdot \frac{n}{2} + T\left(\frac{n}{2}\right)$.

Then, $a = 1$, $b = 2$, $k = 1 \Rightarrow a < b^k$

$\Rightarrow T(n) = O(n)$

However, accessing an element in an array only requires constant time.

Therefore, using the right data structure is important. Your algorithm's running time is often dependent on the data structure(s) being used.

Binary search running time

Observation: At each step there is a region of A where x could be and we **shrink** the size of this region by a factor of 2 with every probe:

- If n is odd, then we are throwing away $\lceil n/2 \rceil$ elements.
- If n is even, then we are throwing away at least $n/2$ elements.

Binary search running time

Observation: At each step there is a region of A where x could be and we **shrink** the size of this region by a factor of 2 with every probe:

- If n is odd, then we are throwing away $\lceil n/2 \rceil$ elements.
- If n is even, then we are throwing away at least $n/2$ elements.

Hence the recurrence for the running time is

$$T(n) \leq T(n/2) + O(1)$$

Sublinear running time

Here are two ways to argue about the running time:

1. Master theorem: $b = 2, a = 1, k = 0 \Rightarrow T(n) = O(\log n)$.
2. We can reason as follows: starting with an array of size n,
   - After k probes, the array has size at most $\frac{n}{2^k}$ (every time we probe an entry, the active portion of the array halves).
   - After $k = \log n$ probes, the array has **constant** size. We can now search **linearly** for x in the constant size array.
   - We spend **constant** work to halve the array (*why?*). Thus the total work spent is $O(\log n)$.

Concluding remarks on binary search

1. The right data structure can improve the running time of the algorithm significantly.
   - *What if we used a* **linked list** *to store the input?*
   - Arrays allow for **random access** of their elements: given an index, we can read any entry in an array in time $O(1)$ (constant time).

2. In general, we obtain running time $O(\log n)$ when the algorithm does a **constant amount of work** to throw away a **constant fraction** of the input.

Today

Integer multiplication

- *How do we multiply two integers x and y?*
- Elementary school method: compute a partial product by multiplying every digit of y separately with x and then add up all the partial products. → Requires $O(n^2)$ time
- Remark: this method works the same in base 10 or base 2.

Examples: $(12)_{10} \cdot (11)_{10}$ and $(1100)_2 \cdot (1011)_2$

Multiplying fixed-sized integers takes constant time.

Say want to multiply

$x_1 x_2 \cdots x_n$
$\times\ y_1 y_2 \cdots y_n$

} $= O(n^2)$

```
   12
 × 11
----
   12
+ 12
----
  132
```

```
   1100
 × 1011
------
   1100
  1100
  0000
+ 1100
--------
 10000100
```

When multiplying very large integers, we will introduce a different model of computation: Every digit-by-digit operation (or bit-by-bit) requires constant time

X_H = high-order digits
X_L = low-order digits

Ex: $\overbrace{14}^{X_H}\overbrace{01}^{X_L}$ (4-digits)

$= 14 \cdot 100 + 1$ $\quad X_H = 14$, $X_L = 01$

$= 14 \cdot 10^2 + 1$

$= X_H \cdot 10^2 + X_L$

Ex: $\overbrace{135}^{X_H}\overbrace{015}^{X_L} = 135 \cdot 1000 + 15$

$= 135 \cdot 10^3 + 15$

$= X_H \cdot 10^3 + X_L$

Want to multiply x and y (both n-digit #'s):

$$x = x_{n-1}\, x_{n-2} \cdots x_{\frac{n}{2}}\, x_{\frac{n}{2}-1} \cdots x_0$$

$$\longleftarrow X_H \longrightarrow \longleftarrow X_L \longrightarrow$$

$$y = y_{n-1}\, y_{n-2} \cdots y_{\frac{n}{2}}\, y_{\frac{n}{2}-1} \cdots y_0$$

$$\longleftarrow Y_H \longrightarrow \longleftarrow Y_L \longrightarrow$$

Notice: Each of X_H, Y_H, X_L, Y_L is a $\frac{n}{2}$-digit number

Now,

$$x \cdot y = (X_H \cdot 10^{n/2} + X_L)(Y_H \cdot 10^{n/2} + Y_L)$$

$$= X_H Y_H \cdot 10^n + (X_H Y_L + X_L Y_H) \cdot 10^{n/2} + X_L Y_L$$

We have reduced the problem of computing two n-digit #'s to the problem of computing four products of two $\frac{n}{2}$-digit numbers.

Let $T(n)$ = time to multiply 2 n-digit #'s.

$T(n) = 4 \cdot T\left(\frac{n}{2}\right) +$ ... ← (computed below)

extra work to multiply by 10^n and $10^{n/2}$ and add #'s together.

Sizes of numbers:

- Each of X_H, Y_H, X_L, Y_L is an $\frac{n}{2}$-digit #'s
- $X_H Y_H$, $X_L Y_H$, $X_H Y_L$, $X_L Y_L$: $\leq$ n-digit #'s
- $X_L Y_H + X_H Y_L$: $\leq$ $(n+1)$-digit number
- $(X_L Y_H + X_H Y_L) \cdot 10^{n/2}$ $\leq$ $2n$-digits
- $X_H Y_H \cdot 10^n$ $\leq$ $2n$-digits

Time to compute the final sum:

$$\underbrace{X_H Y_H \cdot 10^n}_{O(n)} + \underbrace{(X_H Y_L + X_L Y_H) \cdot 10^{n/2}}_{O(n)+O(n)=O(n)} + \underbrace{X_L Y_L}_{O(n)}$$

$O(n)$ to sum the #'s:

$(\leq 2n\text{-digit \#}) + (\leq 2n\text{-digit \#}) + (\leq n\text{-digit \#})$

Therefore,

$$T(n) = 4T\left(\frac{n}{2}\right) + c \cdot n$$

$a = 4,\ b = 2,\ k = 1, \quad a > b^k$

Then, by the Master Theo

$$T(n) = O\left(n^{\log_2 4}\right) = O(n^2)$$

Not an improvement of original problem!

In fact, this is worse since we introduce other n-digit #'s.

We can improve this by reducing the number of sub-problems so that 'a' is smaller.

Elementary algorithm running time

A more reasonable model of computation: a **single** operation on a pair of digits (bits) is a primitive computational step.

Assume we are multiplying n-digit (bit) numbers.

- $O(n)$ time to compute a partial product.
- $O(n)$ time to combine it in a running sum of all partial products so far.

$\Rightarrow$ There are n partial products, each consisting of n bits, hence total number of operations is $O(n^2)$.

Can we do better?

A first divide & conquer approach

Consider n-digit decimal numbers x, y.

$$\begin{aligned} x &= x_{n-1}x_{n-2}\cdots x_0 \\ y &= y_{n-1}y_{n-2}\cdots y_0 \end{aligned}$$

Idea: rewrite each number as the sum of the $n/2$ high-order digits and the $n/2$ low-order digits.

$$x = \underbrace{x_{n-1}\cdots x_{n/2}}_{x_H}\underbrace{x_{n/2-1}\cdots x_0}_{x_L} = x_H \cdot 10^{n/2} + x_L$$

$$y = \underbrace{y_{n-1}\cdots y_{n/2}}_{y_H}\underbrace{y_{n/2-1}\cdots y_0}_{y_L} = y_H \cdot 10^{n/2} + y_L$$

where each of x_H, x_L, y_H, y_L is an $n/2$-digit number.

Examples

- $n = 2$, $x = 12$, $y = 11$

$$\underbrace{12}_{x} = \underbrace{1}_{x_H} \cdot \underbrace{10^1}_{10^{n/2}} + \underbrace{2}_{x_L}$$
$$\underbrace{11}_{y} = \underbrace{1}_{y_H} \cdot \underbrace{10^1}_{10^{n/2}} + \underbrace{1}_{y_L}$$

- $n = 4$, $x = 1000$, $y = 1110$

$$\underbrace{1000}_{x} = \underbrace{10}_{x_H} \cdot \underbrace{10^2}_{10^{n/2}} + \underbrace{0}_{x_L}$$
$$\underbrace{1110}_{y} = \underbrace{11}_{y_H} \cdot \underbrace{10^2}_{10^{n/2}} + \underbrace{10}_{y_L}$$

A first divide & conquer approach

$$\begin{aligned} x \cdot y &= (x_H 10^{n/2} + x_L) \cdot (y_H 10^{n/2} + y_L) \\ &= x_H y_H \cdot 10^n + (x_H y_L + x_L y_H) \cdot 10^{n/2} + x_L y_L \end{aligned}$$

In words, we reduced the problem of solving 1 instance of size n (i.e., one multiplication between two n-digit numbers) to the problem of solving 4 instances, each of size $n/2$ (i.e., computing the products $x_H y_H, x_H y_L, x_L y_H$ and $x_L y_L$).

A first divide & conquer approach

$$\begin{aligned} x \cdot y &= (x_H 10^{n/2} + x_L) \cdot (y_H 10^{n/2} + y_L) \\ &= x_H y_H \cdot 10^n + (x_H y_L + x_L y_H) \cdot 10^{n/2} + x_L y_L \end{aligned}$$

In words, we reduced the problem of solving 1 instance of size n (i.e., one multiplication between two n-digit numbers) to the problem of solving 4 instances, each of size $n/2$ (i.e., computing the products $x_H y_H, x_H y_L, x_L y_H$ and $x_L y_L$).

This is a divide and conquer solution!

- Recursively solve the 4 subproblems.
- Multiplication by 10^n is easy (**shifting**): $O(n)$ time.
- Combine the solutions from the 4 subproblems to an overall solution using 3 additions on $O(n)$-digit numbers: $O(n)$ time.

Karatsuba's observation

Running time: $T(n) \leq 4T(n/2) + cn$

- by the Master Theorem: $T(n) = O(n^2)$
- **no** improvement

Karatsuba's observation

Running time: $T(n) \leq 4T(n/2) + cn$

- by the Master Theorem: $T(n) = O(n^2)$
- **no** improvement

However, if we only needed three $n/2$-digit multiplications, then by the Master theorem

$$T(n) \leq 3T(n/2) + cn = O(n^{1.59}) = o(n^2).$$

Karatsuba's observation

Running time: $T(n) \leq 4T(n/2) + cn$

- by the Master Theorem: $T(n) = O(n^2)$
- **no** improvement

However, if we only needed three $n/2$-digit multiplications, then by the Master theorem

$$T(n) \leq 3T(n/2) + cn = O(n^{1.59}) = o(n^2).$$

Recall that

$$x \cdot y \quad = \quad x_H y_H \cdot 10^n + (x_H y_L + x_L y_H) 10^{n/2} + x_L y_L$$

Key observation: we do not need each of $x_H y_L, x_L y_H$.
We only need their sum, $x_H y_L + x_L y_H$.

Gauss's observation on multiplying complex numbers

A similar situation: multiply two complex numbers $a + bi, c + di$

$$(a + bi)(c + di) = ac + (ad + bc)i + bdi^2$$

Gauss's observation on multiplying complex numbers

A similar situation: multiply two complex numbers $a+bi, c+di$

$$(a+bi)(c+di) = ac + (ad+bc)i + bdi^2$$

Gauss's observation: can be done with just 3 multiplications

$$(a+bi)(c+di) = ac + ((a+b)(c+d) - ac - bd)i + bdi^2,$$

at the cost of few extra additions and subtractions.

$*$ Unlike multiplications, additions and subtractions of n-digit numbers are cheap: $O(n)$ time!

Karatsuba's algorithm

$$\begin{aligned} x \cdot y &= (x_H 10^{n/2} + x_L) \cdot (y_H 10^{n/2} + y_H) \\ &= x_H y_H \cdot 10^n + (x_H y_L + x_L y_H) 10^{n/2} + x_L y_L \end{aligned}$$

Similarly to Gauss's method for multiplying two complex numbers, compute only the three products

$$x_H y_H, \; x_L y_L, \; (x_H + x_L)(y_H + y_L)$$

← Product of two $\frac{n}{2}$-digit (roughly) integers.

and obtain the sum $x_H y_L + x_L y_H$ from

$$(x_H + x_L)(y_H + y_L) - x_H y_H - x_L y_L = x_H y_L + x_L y_H.$$

Combining requires $O(n)$ time hence

$$T(n) \leq 3T(n/2) + cn = O(n^{\log_2 3}) = O(n^{1.59})$$

Theoretical Lower Bound for multiplying 2 n-digit #'s ? : $O(n)$ Because

Substantial improvement

Pseudocode

the output could consist of up to $2n$-digits.

More beneficial for VERY long digit numbers.

Let k be a small constant.

Integer-Multiply(x, y)

if $n == k$ **then**

return xy

end if

write $x = x_H 10^{n/2} + x_L, y = y_H 10^{n/2} + y_L$

compute $x_H + x_L$, $y_H + y_L$

$product =$ **Integer-Multiply**$(x_H + x_L, y_H + y_L)$

$x_H y_H =$ **Integer-Multiply**(x_H, y_H)

$x_L y_L =$ **Integer-Multiply**(x_L, y_L)

return $x_H y_H 10^n + (product - x_H y_H - x_L y_L)10^{n/2} + x_L y_L$

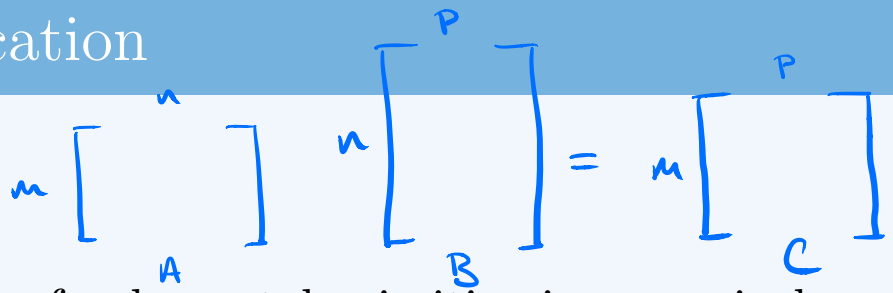
Concluding remarks

- To reduce the number of multiplications we do few more additions/subtractions: these are fast compared to multiplications.
- There is no reason to continue with recursion once n is small enough: the conventional algorithm is probably more efficient since it uses fewer additions.
- When we recursively compute $(x_H + x_L)(y_H + y_L)$, each of $x_H + x_L$, $y_H + y_L$ might be $(n/2 + 1)$-digit integers. This does not affect the asymptotics.

Today

1. Recap
2. Binary search
3. Integer multiplication
4. Fast matrix multiplication (Strassen's algorithm)

Fast matrix multiplication

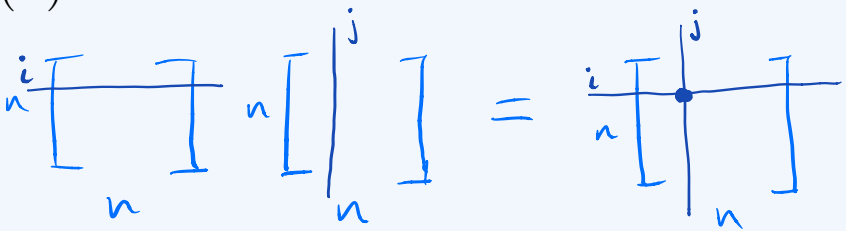


Matrix multiplication: a fundamental primitive in numerical linear algebra, scientific computing, machine learning and large-scale data analysis.

- Input: $m \times n$ matrix A, $n \times p$ matrix B
- Output: $m \times p$ matrix $C = AB$

Example: $A = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}, B = \begin{pmatrix} 1 & 1 \\ 1 & 1 \end{pmatrix}, C = \begin{pmatrix} 1 & 1 \\ 1 & 1 \end{pmatrix}$

Lower bounds on matrix multiplication algorithms for $m, p = \Theta(n)$*?*



Traditional Matrix multiplication Algorithm

Conventional matrix multiplication

Traditional Matrix multiplication algorithm:

for $1 \leq i \leq n$ **do**
  for $1 \leq j \leq n$ **do**
    $c_{i,j} = 0$
    for $1 \leq k \leq n$ **do**
      $c_{i,j} += a_{i,k} \cdot b_{k,j}$
    end for
  end for
end for

There are 3 nested for-loops, each one goes from 1 to n, so the complexity is $O(n^3)$.

- *Running time?* $O(n^3)$
- *Can we do better?*

↳ Lower bound is $\Omega(n^2)$ since each entry of $C_{n \times n}$ has to be computed.

A first divide & conquer approach: 8 subproblems

Assume square A, B where $n = 2^k$ for some $k > 0$.
Idea: express A, B as 2×2 block matrices and use the conventional algorithm to multiply the two block matrices.

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

(A_{11}: $n/2 \times n/2$; B_{11}: $n/2 \times n/2$; A_{21}: $n/2 \times n/2$; A: $n \times n$; B: $n \times n$; C: $n \times n$)

where

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

Let $T(n)$ = time to multiply 2 $n \times n$ matrices.

8 $\frac{n}{2} \times \frac{n}{2}$ matrix computations.

Running time?

$T(n) = 8T\left(\frac{n}{2}\right) + cn^2$

$a = 8$, $b = 2$, $k = 2$, $a > b^k$ $\Rightarrow$ $O(n^{\log_2 8}) = O(n^3)$

Strassen wanted to reduce the # of sub-problems

That is:

$$T(n) = 7\,T\left(\frac{n}{2}\right) + cn^2$$

$$\Rightarrow = O(n^{\log_2 7})$$

$$\approx O(n^{2.81})$$

$$n^{\log_2 7} = o(n^3)$$

$$\approx n^{2.81}$$

little-o

Reduces 8 subproblems to 7 subproblems

Strassen's breakthrough: 7 subproblems suffice (part 1)

Do NOT need to memorize this!!

Compute the following ten $n/2 \times n/2$ matrices.

1. $S_1 = B_{11} - B_{22}$
2. $S_2 = A_{11} + A_{12}$
3. $S_3 = A_{21} + A_{22}$
4. $S_4 = B_{21} - B_{11}$
5. $S_5 = A_{11} + A_{22}$
6. $S_6 = B_{11} + B_{22}$
7. $S_7 = A_{12} - A_{22}$
8. $S_8 = B_{21} + B_{22}$
9. $S_9 = A_{11} - A_{21}$
10. $S_{10} = B_{11} + B_{12}$

Running time?

Each of these matrices is of dimension $\frac{n}{2} \times \frac{n}{2}$ so they have $\frac{n^2}{4}$ entries.

$\Rightarrow O(n^2)$ for each

$\Theta(n^2)$ or cn^2

Strassen's breakthrough: 7 subproblems suffice (part 2)

Compute the following seven products of $n/2 \times n/2$ matrices.

1. $P_1 = A_{11}S_1$
2. $P_2 = S_2B_{22}$
3. $P_3 = S_3B_{11}$
4. $P_4 = A_{22}S_4$
5. $P_5 = S_5S_6$
6. $P_6 = S_7S_8$
7. $P_7 = S_9S_{10}$

Each S_i is an $n/2 \times n/2$

$7 \cdot T(n/2)$

Each of the P_i is also an $n/2 \times n/2$ matrix

Compute C as follows:

1. $C_{11} = P_4 + P_5 + P_6 - P_2$
2. $C_{12} = P_1 + P_2$
3. $C_{21} = P_3 + P_4$
4. $C_{22} = P_1 + P_5 - P_3 - P_7$

cn^2 (or $\Theta(n^2)$)

Running time?

Strassen's running time and concluding remarks

$T(n) = 7T(\frac{n}{2}) + 2\Theta(n^2) = 7T(\frac{n}{2}) + cn^2$

$\Rightarrow T(n) = O(n^{\log_2 7})$

$= O(n^{2.81})$

- Recurrence: $T(n) = 7T(n/2) + cn^2$
- By the Master theorem:

$$T(n) = O(n^{\log_2 7}) = O(n^{2.81})$$

- Recently, there is renewed interest in Strassen's algorithm for **high-performance computing**: thanks to its lower communication cost (number of bits exchanged between machines in the network or data center), it is better suited than the traditional algorithm for multi-core processors.