Next: Correctness

- How can we show that the algorithm works correctly for all possible inputs of all possible sizes?
- Exhaustive testing not feasible.
- Analytical techniques are ~~useful~~ essential here.

Find-max revisited

Q1. Find the max of n numbers (stored in array A)

Formal specs:

INPUT: A[1..n] - an array of integers

OUTPUT: an element m of A such that $A[j] \le m$, $1 \le j \le \text{length}(A)$

Find-max (A)

```
1. max ← A[1]
2. for j ← 2 to length(A)
3.    do if (max < A[j])
4.          max ← A[j]
5. return max
```

Correctness Proof 1

INPUT: A[1..n] - an array of integers

OUTPUT: an element m of A such that $m \leq A[j]$, $1 \leq j \leq \text{length}(A)$

```
Find-max (A)
 1. max ← A[1]
 2. for j ← 2 to length(A)
 3.     do if (max < A[j])
 4.            max ← A[j]
 5. return max
```

Prove that for any valid Input, the output of Find-max satisfies the output condition.

Proof 1 [by contradiction]: Suppose the algorithm is incorrect. Then for some input A,

(a) max is not an element of A or

(b) ($\exists j$ | max < A[j]).

max is initialized to and assigned to elements of A – so (a) is impossible. WHY?

(b) After the j-th iteration of the for-loop (lines 2 – 4), max $\geq$ A[j]. From lines 3,4, max only increases.

Therefore, upon termination, max $\geq$ A[j], which contradicts (b).

Correctness Proof 1 - comments

- The preceding proof reasons about the whole algorithm
- It is possible to prove correctness by induction as well: this is left as an exercise for you.
- What if the algorithm/program was very big and had many function calls, nested loops, if-then’s and other standard features?
- Need a simpler, more “modular” strategy.

Correctness Proof – 2 (typos fixed)

INPUT: A[1..n] - an array of integers
OUTPUT: an element m of A such that $m \leq A[j]$, $1 \leq j \leq \text{length}(A)$

```
Find-max (A)
 1. max ← A[1]
 2. for j ← 2 to length(A)
 3.     do if (max < A[j])
 4.            max ← A[j]
 5. return max
```

Prove that for any valid Input, the output of Find-max satisfies the output condition.

Proof 2 [use loop invariants]:

(identify invariant) At the beginning of iteration j of for loop, max contains the maximum of A[1..j-1].

(Proof) Clearly true for j=2. For j = 3,4,…, assume that invariant holds for j-1. So at the beginning of iteration j-1 max contains the maximum of A[1..j-2].

Case (a) A[j-1] is the maximum of A[1..j-1]. In lines 3,4, max is set to A[j-1].

Case (b) A[j-1] is not the maximum of A[1..j-1], so the maximum of A[1..j-1] is in A[1..j-2]. By our assumption max already has this value and by lines 3-4 max is unchanged in this iteration.

Correctness Proof – continued

INPUT: A[1..n] - an array of integers
OUTPUT: an element m of A such that $m \leq A[j]$, $1 \leq j \leq \text{length}(A)$

```
Find-max (A)
 1. max ← A[1]
 2. for j ← 2 to length(A)
 3.     do if (max < A[j])
 4.             max ← A[j]
 5. return max
```

Proof using loop invariants - continued:

We proved that the invariant holds at the beginning of iteration j for each j used by Find-max.

Upon termination, j = length(A)+1. (WHY?)
The invariant holds, and so max contains the maximum of A[1..n]
-- STRUCTURED PROOF TECHNIQUE!
-- VERY SIMILAR TO INDUCTION!

We will see more non-trivial examples later.

More about correctness

- Don’t tack on a formal proof of correctness after coding to make the professor happy.
- It need not be mathematical mumbo jumbo.
- Goal: To think about algorithms in such way that their correctness is transparent.

1. Iterative Algorithms

“Take one step at a time towards the final destination”

```
loop (until done)
    take step
end loop
```

2. Recursive Algorithms

LATER.

Loop invariants

A good way to structure many programs:

- Store the key information you currently know in some data structure.
- In the main loop,
  - take a step forward towards destination

    by making a simple change to this data.

Insertion sort - correctness

```
for j=2 to length(A)
  do key=A[j]
    i=j-1
    while i>0 and A[i]>key
      do A[i+1]=A[i]
         i--
    A[i+1]:=key
```

What is a good loop invariant?

It is easy to write a loop invariant if you understand what the algorithm does.

Use assertions.

Assertions

An assertion is a statement about the current state of the data structure that is either true or false.

Useful for

- thinking about algorithms
- developing
- describing
- proving correctness

An assertion need not consist of formal/math mumbo jumbo

Use an informal description

An assertion is not a task for the algorithm to perform.

It is only a comment that is added for the benefit of the reader.

Assertions – contd.

Example of Assertions

- Preconditions: Any assumptions that must be true about the input instance.
- Postconditions: The statement of what must be true when the algorithm/program returns.

Correctness:

$$\langle\text{PreCond}\rangle \;\&\; \langle\text{code}\rangle \Rightarrow \langle\text{PostCond}\rangle$$

If the input meets the preconditions,
then the output must meet the postconditions.

If the input does not meet the preconditions,
then nothing is required.

Assertions – contd.

Example of Assertions

```
<preCond>
codeA
loop
    <loop-invariant>
    exit when <exit Cond>
    codeB
endloop
codeC
<postCond>
```

Partial correctness

We must show three things about loop invariants:

- **Initialization** – it is true prior to the first iteration
- **Maintenance** – if it is true before an iteration, it remains true before the next iteration
- **Termination** – when loop terminates the invariant gives a useful property to show the correctness of the algorithm

Proves that IF the program terminates then it works

Partial Correctness &
Termination $\Longrightarrow$ Correctness

Correctness of Insertion sort

```
for j=2 to length(A)
  do key=A[j]
     //Insert A[j] into the sorted
          //sequence A[1..j-1]
     i=j-1
     while i>0 and A[i]>key
       do A[i+1]=A[i]
          i--
     A[i+1]:=key
```

Invariant: *at the start of* ***for*** *loop iteration j, A[1…j-1] consists of elements originally in A[1…j-1] but in sorted order*

Initialization: *j = 2,* the invariant trivially holds because *A*[1] is a sorted array ☺

Correctness of Insertion sort – contd.

```
for j=2 to length(A)
  do key=A[j]
    i=j-1
    while i>0 and A[i]>key
      do A[i+1]=A[i]
        i--
    A[i+1]:=key
```

Invariant: *at the start of* ***for*** *loop iteration j, A[1...j-1] consists of elements originally in A[1...j-1] but in sorted order*

Maintenance: the inner **while** loop moves elements $A[j-1]$, $A[j-2]$, …, $A[k]$ one position right without changing their order. Then the former $A[j]$ element is inserted into kth position so that $A[k-1] \leq A[k] \leq A[k+1]$.

$$A[1 \ldots j-1] \text{ sorted} + A[j] \rightarrow A[1 \ldots j] \text{ sorted}$$

Correctness of Insertion sort – contd.

```
for j=2 to length(A)
  do key=A[j]
     Insert A[j] into the sorted
         sequence A[1..j-1]
     i=j-1
     while i>0 and A[i]>key
       do A[i+1]=A[i]
          i--
     A[i+1]:=key
```

Invariant: *at the start of* ***for*** *loop iteration j, A[1…j-1] consists of elements originally in A[1…j-1] but in sorted order*

Termination: the loop terminates, when *j=n+1*.

Then the invariant states: *"A[1…n] consists of elements originally in A[1…n] but in sorted order"* ☺

More on correctness of iterative algorithms

1. Spent some time formalizing asymptotic notation.
2. Have seen insertion-sort and loop invariants for it.
   The invariant falls under the "more of the input" class in Jeff Edmonds' notation.
3. Next, selection sort; the invariant for this falls under the "more of the output" class in Jeff Edmonds' notation.

Loop invariants

Recall that

1. Loop invariants allow you to reason about a single iteration of the loop.
2. The test condition of the loop is not part of the invariant.
3. Design the loop invariant so that when the termination condition is attained, and the invariant is true, then the goal is reached: invariant + termination => goal
4. Create invariants which are
   -- simple, and
   -- capture all the goals of the algorithm (except termination)

It takes practice

It is best to use mathematical symbols for loop invariants; when this is too complicated, use clear prose and common sense.

Selection sort

I/O specs: same as insertion sort

Algorithm: Given an array A of n integers, sort them by repetitively selecting the smallest among the yet unselected integers.

Is this precise enough?

Swap the smallest integer with the integer currently in the place where the smallest integer should go.

Loop invariant: at the beginning of the jth iteration

- The smallest j-1 values are sorted in descending order in locations [1,j-1]

•Is this enough? No….

and the rest are in locations [n-j,n].

See if you can prove it.

Another kind of loop invariant

Narrowed the search space, e.g. Binary search

- Preconditions
  - –Key 25
  - –Sorted List

3	5	6	13	18	21	21	25	36	43	49	51	53	60	72	74	83	88	91	95

- Postcondition

–Find key in list (if present).

3	5	6	13	18	21	21	25	36	43	49	51	53	60	72	74	83	88	91	95

Define Loop Invariant

- Maintain a sublist.
- If the key is contained in the original list, then the key is contained in the sublist.

Define an iteration of loop

•Cut sublist in half.

•Determine which half the key would be in.

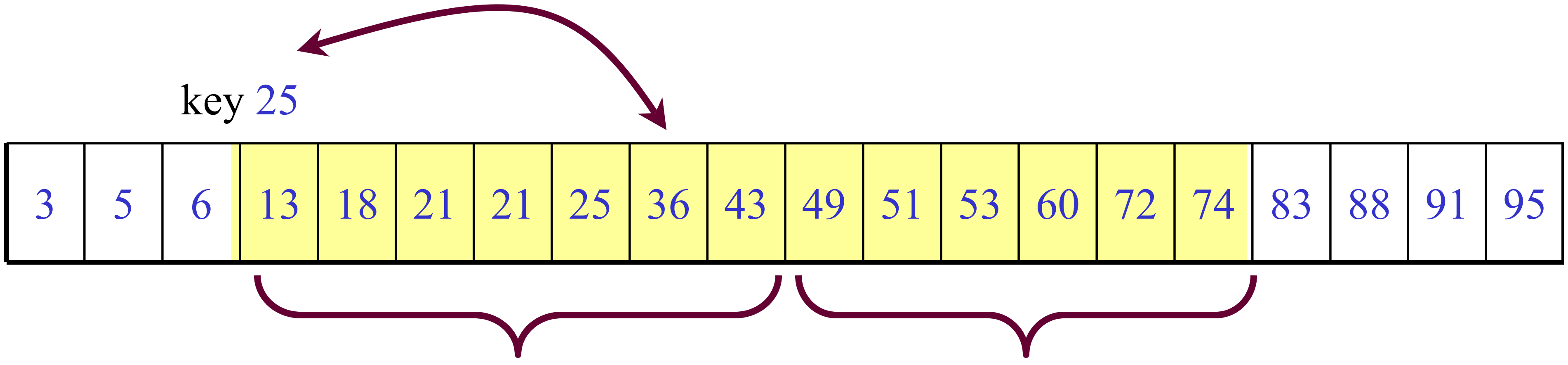
•Keep that half.

Caveat:

Invariant must not assume that the element is present in the list. So it should say something like

"If the key is contained in the original list, then the key is contained in the sublist."

Define an iteration of loop – contd.



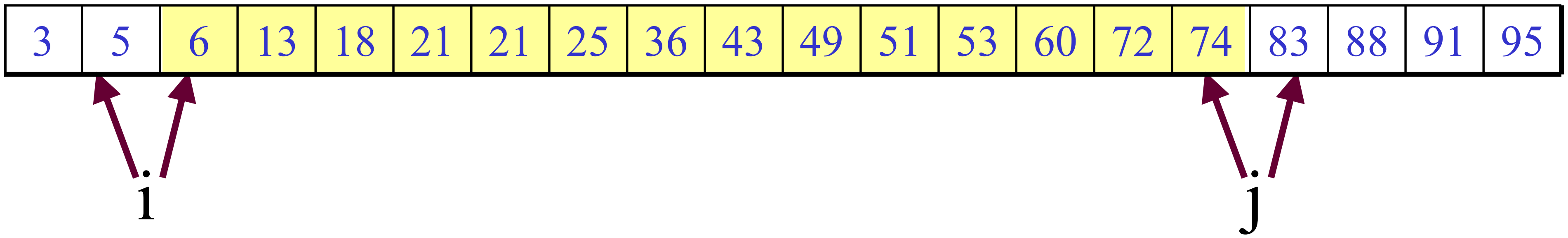
If key ≤ mid, then key is in left half.

If key > mid, then key is in right half.

It is faster not to check if the middle element is the key.

The devil is in the details...

- Maintain a sublist with end points i & j



Does not matter which, but you need to be consistent.

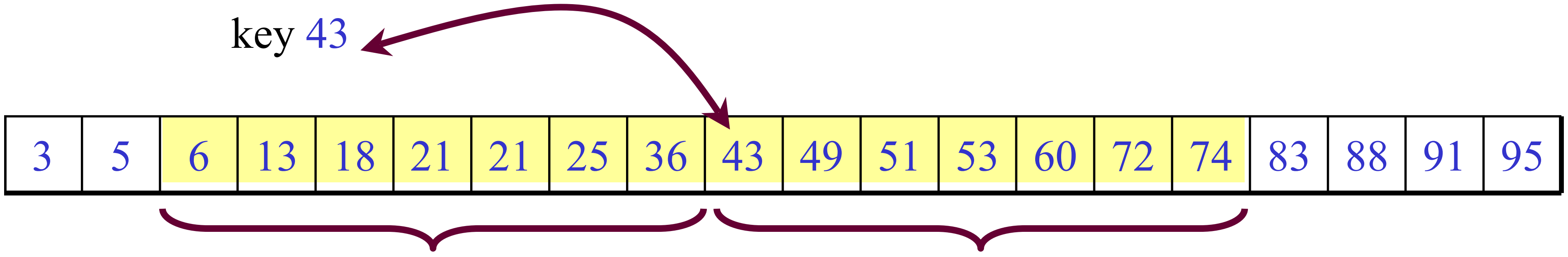
•If the sublist has even length, which element is taken to be mid?

Does not matter – choose **right**.

An easy mistake…

If key ≤ mid, then key is in left half: [i,mid-1].

If key > mid, then key is in right half: [mid,j]

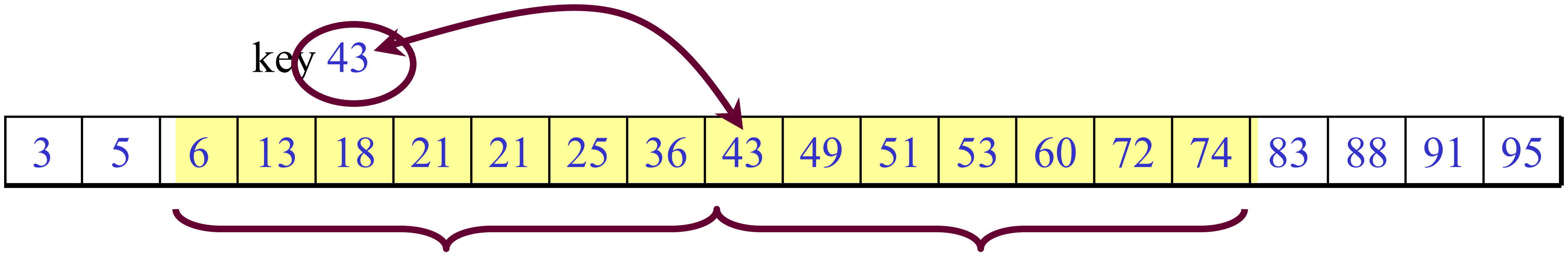


If the middle element is the key, it can be skipped over!

A fix…

If key $\cancel{\leq}$ mid,
then key is in
left half: [i,mid-1].

If key $\underline{\geq}$ mid,
then key is in
right half: [mid,j].

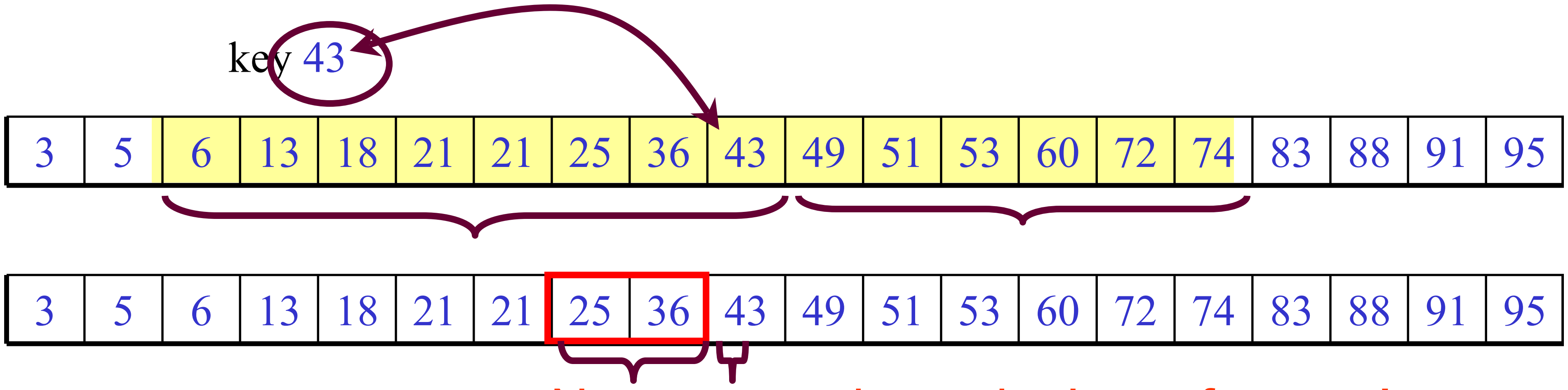


Another possible fix…

- making the left half slightly bigger.

If key ≤ mid, then key is in left half: [i,mid].

If key > mid, then key is in right half: [mid+1,j].



No progress is made. Loop for ever!

Lessons to be learnt

- Use the loop invariant method to think about algorithms.
- Be careful with your definitions.
- Be sure that the loop invariant is always maintained.
- Be sure progress is always made.

Running time of binary search

From now, we will omit details about accounting for running time as follows. The details are tedious but can be supplied easily. We will also ignore floors and ceilings. This <u>usually</u> makes no difference.

The sublist is of size $n, {}^{n}/_{2}, {}^{n}/_{4}, {}^{n}/_{8}, \ldots, 1$. How many steps is that?

Each step takes $\theta(1)$ time.

Total running time = $\theta(\log n)$

Pseudocode for binary search

```
algorithm $BinarySearch(\langle L(1..n), key\rangle)$

$\langle pre-cond\rangle$: $\langle L(1..n), key\rangle$ is a sorted list and $key$ is an element.

$\langle post-cond\rangle$: If the key is in the list, then the output consists of an index $i$
                such that $L(i) = key$.
begin
    $i = 1$, $j = n$
    loop
        $\langle loop-invariant\rangle$: If the key is contained in $L(1..n)$, then
            the key is contained in the sublist $L(i..j)$.

        exit when $j \leq i$
        $mid = \lfloor \frac{i+j}{2} \rfloor$
        if($key \leq L(mid)$) then
            $j = mid$          % Sublist changed from $L(i,j)$ to $L(i..mid)$
        else
            $i = mid + 1$      % Sublist changed from $L(i,j)$ to $L(mid+1, j)$
        end if
    end loop
    if($key = L(i)$) then
        return( $i$ )
    else
        return( “key is not in list” )
    end if
end algorithm
```

Next: Some mathematical tools

- Important to have the right tools
- Still, these are only tools; necessary but not sufficient to solve problems.

- We will cover some essential tools in this course for your repertoire.

A Quick Math Review

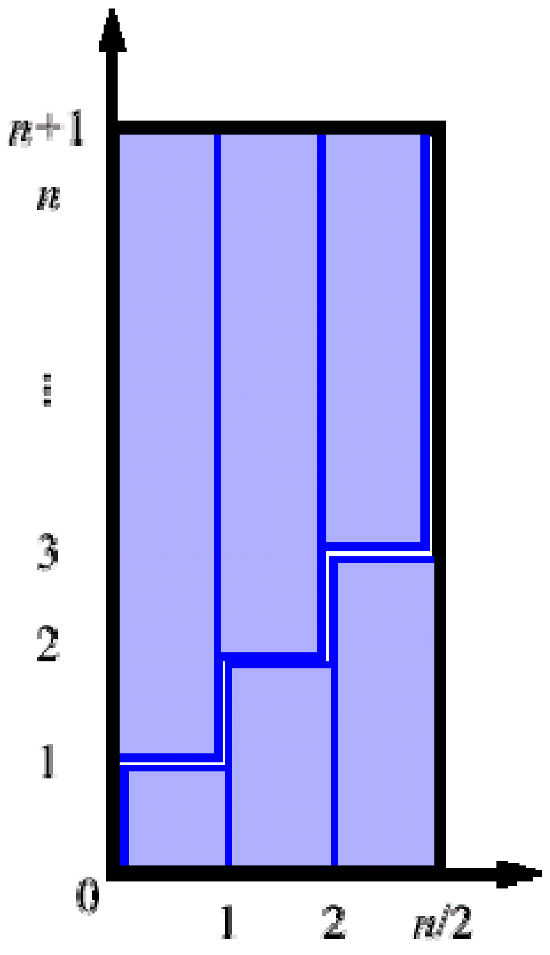
- Geometric progression
  - given an integer n_0 and a real number $0 < a \neq 1$

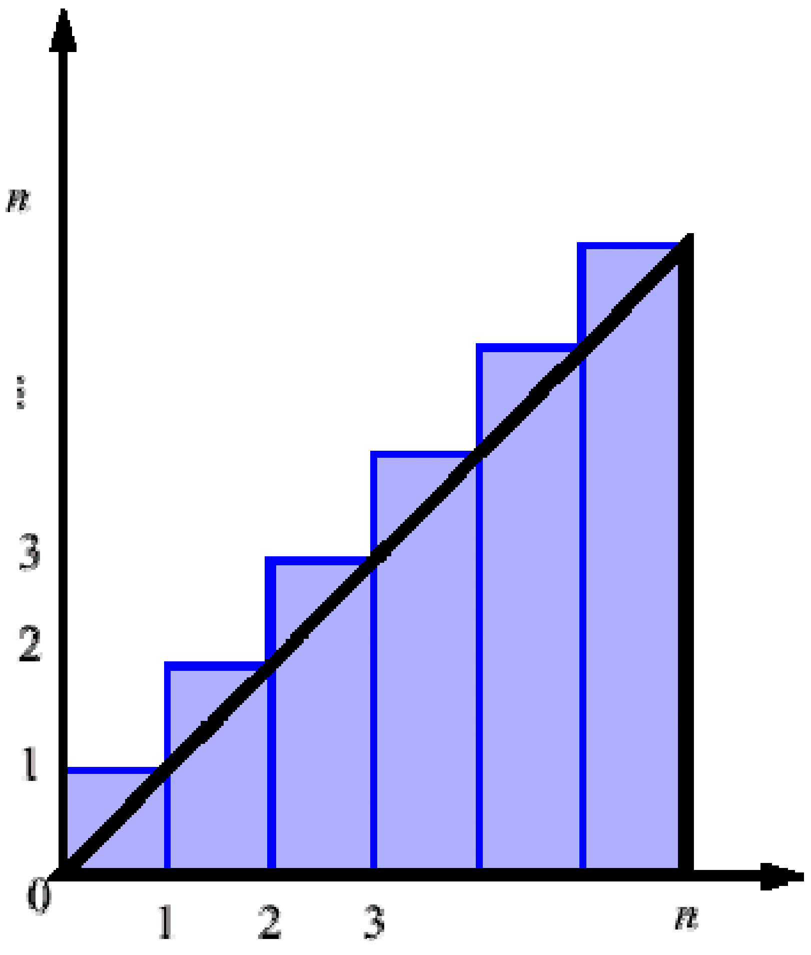
$$\sum_{i=0}^{n} a^i = 1 + a + a^2 + \ldots + a^n = \frac{1 - a^{n+1}}{1 - a}$$

  - geometric progressions exhibit exponential growth
- Arithmetic progression

$$\sum_{i=0}^{n} i = 1 + 2 + 3 + \ldots + n = \frac{n^2 + n}{2}$$

Pictorial proofs of sums





Review: Proof by Induction

- We want to show that property *P* is true for all integers $n \geq n_0$
- **Basis**: prove that *P* is true for n_0
- **Inductive step**: prove that if *P* is true for all *k* such that $n_0 \leq k \leq n-1$ then *P* is also true for *n*
- Example

$$S(n) = \sum_{i=0}^{n} i = \frac{n(n+1)}{2} \text{ for } n \geq 1$$

- Base case:

$$S(1) = \sum_{i=0}^{1} i = \frac{1(1+1)}{2}$$

Proof by Induction (2)

- Inductive Step

$$S(k) = \sum_{i=0}^{k} i = \frac{k(k+1)}{2} \text{ for } 1 \le \mathrm{k} \le n-1$$

$$S(n) = \sum_{i=0}^{n} i = \sum_{i=0}^{n-1} i + n = S(n-1) + n =$$

$$= (n-1)\frac{(n-1+1)}{2} + n = \frac{(n^2 - n + 2n)}{2} =$$

$$= \frac{n(n+1)}{2}$$

Important thumbrules for sums

"addition made easy" – Jeff Edmonds.

"Theta of last term"

Geometric like: $f(i) = 2^{\Omega(i)} \Rightarrow \sum_{i=1}^{n} f(i) = \Theta(f(n))$

no of terms x last term

Arithmetic like: $i.f(i) = i^{\Theta(1)} \Rightarrow \sum_{i=1}^{n} f(i) = \Theta(nf(n))$

Harmonic: $f(i) = 1/i \Rightarrow \sum_{i=1}^{n} f(i) = \Theta(\log n)$

"Theta of first term"

Bounded tail: $i.f(i) = 1/i^{\Theta(1)} \Rightarrow \sum_{i=1}^{n} f(i) = \Theta(1)$

Use as thumbrules only

Later: Some standard techniques

We will get into these techniques as and when we need them. If you are interested, read Appendix A.

- Approximation with integrals :Derive, rather than memorize the formula; e.g $\sum 1/k$.
- Telescoping sum: $\sum 1/(k(k+1))$
- Split a sum: $\sum k/2^k$
- Approximate crudely from both sides: e.g. $\sum 2^k$
- Integrate and differentiate series: $\sum kx^k$

GCD: iterative algorithms

Recall the definition of GCD(a,b). Recall also the high-school technique for computing GCD(a,b).

Key observation: if (a>b) GCD(a,b) = GCD(a – b, b)

How do you prove this?

Any divisor of a,b divides a-b!

Try the new idea

Input: <a,b> = <64,44>

Output: GCD(a,b) = 4

GCD(a,b) = GCD(a-b,b)

GCD(64,44) = GCD(20,44)

GCD(20,44) = GCD(44,20)

GCD(44,20) = GCD(24,20)

GCD(24,20) = GCD(4,20)

GCD(4,20) = GCD(20,4)

GCD(20,4) = GCD(16,4)

GCD(16,4) = GCD(12,4)

GCD(12,4) = GCD(8,4)

GCD(8,4) = GCD(4,4)

GCD(4,4) = GCD(0,4)

What is the running time?

Running time for GCD(a,b)

Input: <a,b> = <9999999999999,2>

<x,y> = <9999999999999,2>

= <9999999999997,2>

= <9999999999995,2>

= <9999999999993,2>

= <9999999999991,2>

Time = $O(a) = 2^{O(n)}$

Size = $n = O(\log(a))$

A faster algorithm for GCD(a,b)

$$
\begin{aligned}
\langle x,y\rangle &\Rightarrow \langle x-y,y\rangle \\
&\Rightarrow \langle x-2y,y\rangle \\
&\Rightarrow \langle x-3y,y\rangle \\
&\Rightarrow \langle x-4y,y\rangle \\
&\Rightarrow \langle x-iy,y\rangle \\
&\Rightarrow \langle x \text{ rem } y,y\rangle \\
&= \langle x \bmod y,y\rangle \\
&\Rightarrow \langle y,x \bmod y\rangle
\end{aligned}
$$

But $x \bmod y < y$

Try the improvement

GCD(a,b) = GCD(b,a mod b)

Input: <a,b> = <44,64>

<x,y> = <44,64>

= <64,44>

= <44,20>

= <20, 4>

= < 4, 0>

GCD(a,b) = 4

A bad example

Input: <a,b> = <10000000000001,9999999999999>

<x,y> = <10000000000001,9999999999999>

= <9999999999999,2>

= <2,1>

= <1,0>

GCD(a,b) = GCD(x,y) = 1

Little progress

Lots of progress

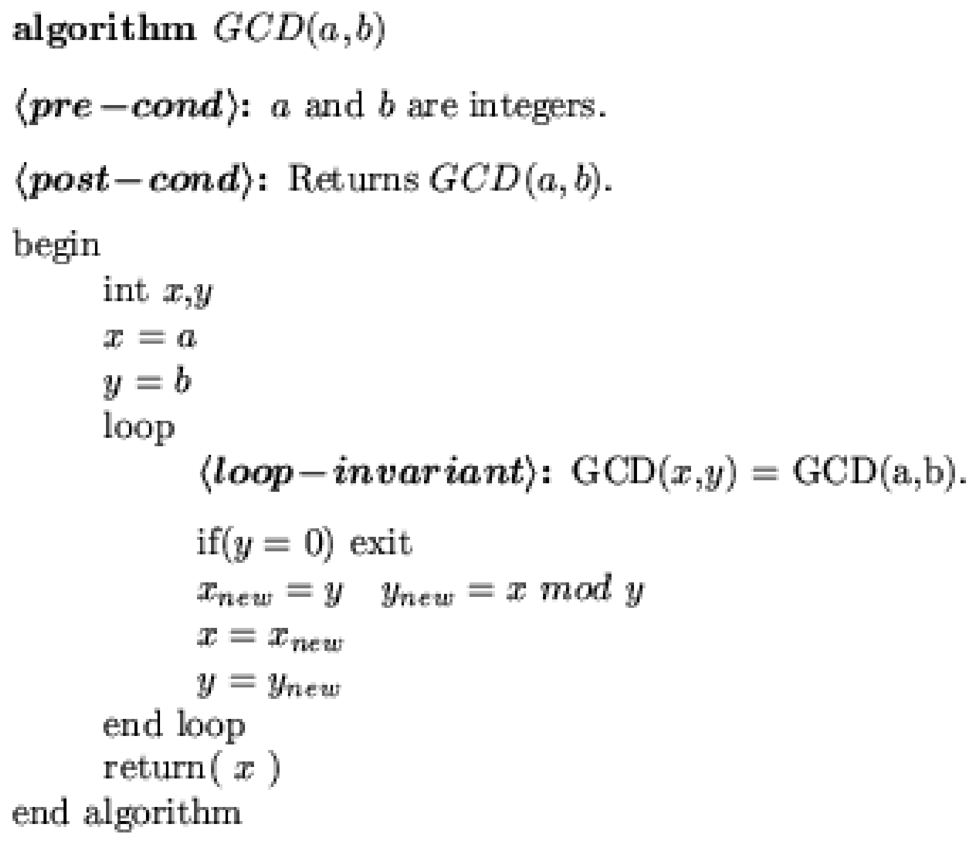
Every two iterations:

the value x decreases by at least a factor of 2.

the size of x decreases by at least one bit.

Running time: O(log(a)+log(b)) = O(n)

GCD(a,b)



```
algorithm $GCD(a,b)$
⟨pre−cond⟩: $a$ and $b$ are integers.
⟨post−cond⟩: Returns $GCD(a,b)$.
begin
    int $x$,$y$
    $x = a$
    $y = b$
    loop
        ⟨loop−invariant⟩: GCD($x$,$y$) = GCD(a,b).
        if($y = 0$) exit
        $x_{new} = y$    $y_{new} = x \ mod \ y$
        $x = x_{new}$
        $y = y_{new}$
    end loop
    return( $x$ )
end algorithm
```

A design paradigm

Divide and conquer

Multiplying complex numbers
(from Jeff Edmonds' slides)

INPUT: Two pairs of integers, (a,b), (c,d) representing complex numbers, a+ib, c+id, respectively.
OUTPUT: The pair [(ac-bd),(ad+bc)] representing the product (ac-bd) + i(ad+bc)

Naïve approach: 4 multiplications, 2 additions.
Suppose a multiplication costs $1 and an addition cost a penny. The naïve algorithm costs $4.02.

Q: Can you do better?

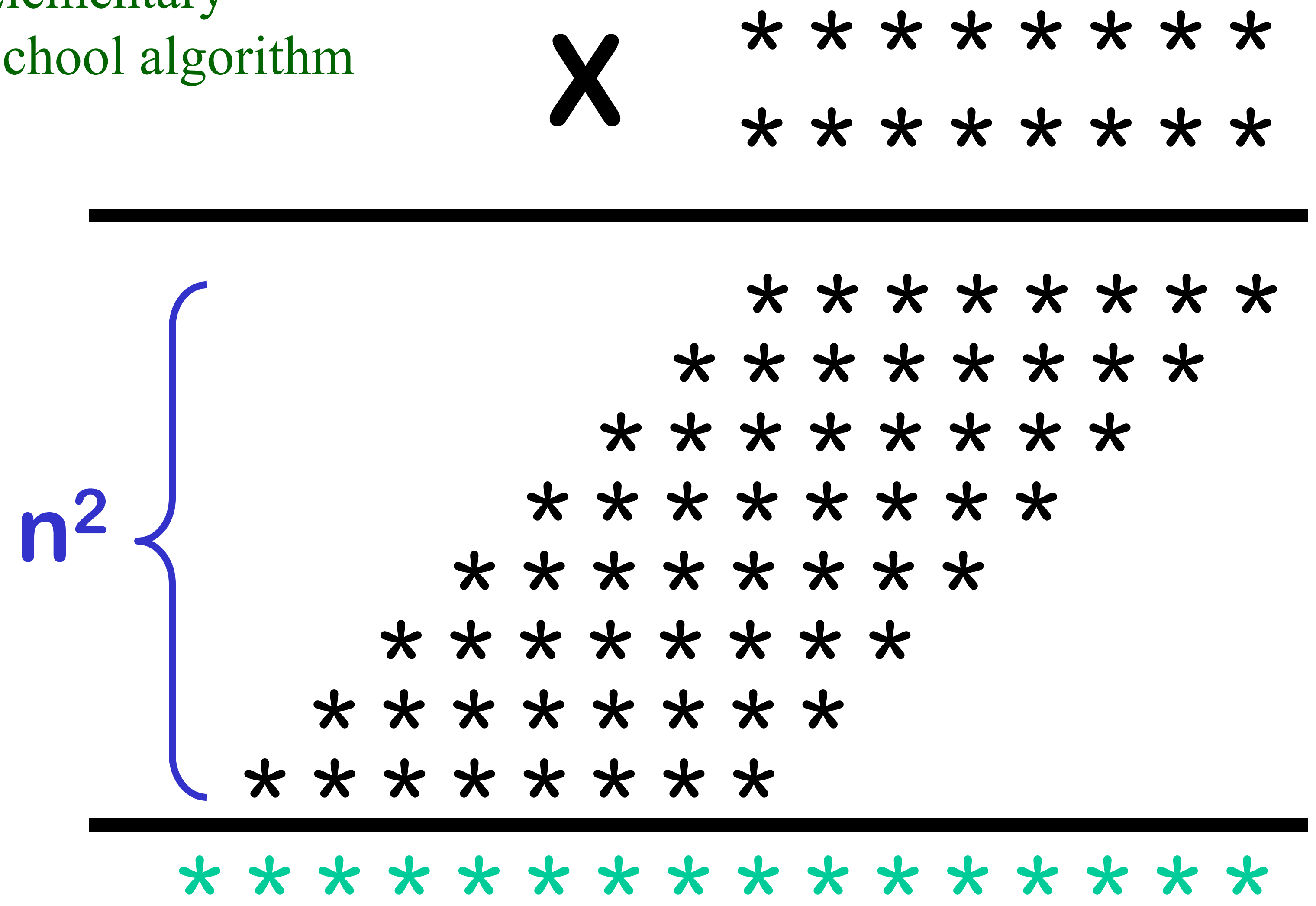
Gauss' idea

- $m_1 = ac$
- $m_2 = bd$
- $A_1 = m_1 – m_2 = ac-bd$
- $m_3 = (a+b)(c+d) = ac + ad + bc + bd$
- $A_2 = m_3 – m_1 – m_2 = ad+bc$
- **Saves 1 multiplication! Uses more additions. The cost now is $3.03.**
- This is good (saves 25% multiplications), but it leads to more dramatic asymptotic improvement elsewhere!
  (aside: look for connections to known algorithms)

Q: How fast can you multiply two n-bit numbers?

How to multiply two n-bit numbers.

Elementary School algorithm



How to multiply two n-bit numbers - contd.

Elementary
School algorithm

```
              * * * * * * * *
        X     * * * * * * * *
  -----------------------------
  * * * * * * * * * * * * * * * *
  -----------------------------
```

Q: Is there a faster algorithm?

A: YES! Use divide-and-conquer.

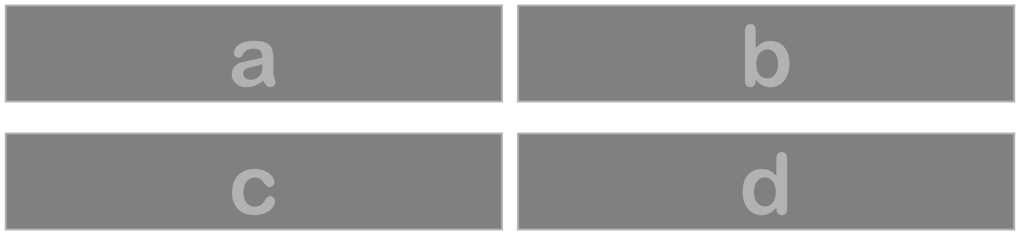
Divide and Conquer

Intuition:

- **DIVIDE** my instance to the problem into smaller instances to the same problem.
- Recursively solve them.
- **GLUE** the answers together so as to obtain the answer to your larger instance.
- Sometimes the last step may be trivial.

Multiplication of two n-bit numbers

- X =
- Y =



- $X = a\ 2^{n/2} + b \qquad Y = c\ 2^{n/2} + d$
- $XY = ac\ 2^n + (ad+bc)\ 2^{n/2} + bd$

```
MULT(X,Y):
 If |X| = |Y| = 1 then RETURN XY
 Break X into a;b and Y into c;d
 RETURN
   MULT(a,c) $2^n$ + (MULT(a,d) + MULT(b,c)) $2^{n/2}$ + MULT(b,d)
```

Time complexity of MULT

- $T(n)$ = time taken by MULT on two n-bit numbers
- What is $T(n)$? Is it $\theta(n^2)$?
- Hard to compute directly
- Easier to express as a **recurrence relation**!
- $T(1) = k$ for some constant k
- $T(n) = 4\,T(n/2) + c_1 n + c_2$ for some constants c_1 and c_2
- How can we get a $\theta()$ expression for $T(n)$?

```
MULT(X,Y):
 If |X| = |Y| = 1 then RETURN XY
 Break X into a;b and Y into c;d
 RETURN
   MULT(a,c) $2^n$ + (MULT(a,d) + MULT(b,c)) $2^{n/2}$ + MULT(b,d)
```

Time complexity of MULT

Make it concrete

- $T(1) = 1$
- $T(n) = 4\ T(n/2) + n$

<u>Technique 1:</u> Guess and verify

$T(n) = 2n^2 - n$

Holds for n=1

$T(n) = 4\ (2(n/2)^2 - n/2 + n)$

$\quad = 2n^2 - n$

Time complexity of MULT

- $T(1) = 1$ & $T(n) = 4\ T(n/2) + n$

<u>Technique 2:</u> Expand recursion

$T(n) = 4\ T(n/2) + n$

$= 4\ (4T(n/4) + n/2) + n = 4^2T(n/4) + n + 2n$

$= 4^2(4T(n/8) + n/4) + n + 2n$

$= 4^3T(n/8) + n + 2n + 4n$

$= \dots\dots\dots$

$= 4^kT(1) + n + 2n + 4n + \dots + 2^{k-1}n$ where $2^k = n$

GUESS

$= n^2 + n\ (1 + 2 + 4 + \dots + 2^{k-1})$

$= n^2 + n\ (2^k-1)$

$= 2\ n^2 - n$ [NOT FASTER THAN BEFORE]

Gaussified MULT (Karatsuba 1962)

```
MULT(X,Y):
 If |X| = |Y| = 1 then RETURN XY
 Break X into a;b and Y into c;d
 e = MULT(a,c) and f =MULT(b,d)
 RETURN  $e2^n + (\mathrm{MULT}(a+b, c+d) - e - f) 2^{n/2} + f$
```

- $T(n) = 3\,T(n/2) + n$
- Actually: $T(n) = 2\,T(n/2) + T(n/2 + 1) + kn$

Time complexity of Gaussified MULT

- $T(1) = 1$ & $T(n) = 3\ T(n/2) + n$

Technique 2: Expand recursion

$T(n) = 3\ T(n/2) + n$

$= 3\ (3T(n/4) + n/2) + n = 3^2T(n/4) + n + 3/2n$

$= 3^2(3T(n/8) + n/4) + n + 3/2n$

$= 3^3T(n/8) + n + 3/2n + (3/2)^2n$

$= \ldots\ldots\ldots$

$= 3^kT(1) + n + 3/2n + (3/2)^2n + \ldots + (3/2)^{k-1}n$ where $2^k= n$

$= 3^{\log_2 n} + n(1 + 3/2 + (3/2)^2 + \ldots + (3/2)^{k-1})$

$= n^{\log_2 3} + 2n\ ((3/2)^k-1)$

$= n^{\log_2 3} + 2n\ (n^{\log_2 3}/n -1)$

$= 3n^{\log_2 3} - 2n$

Not just 25% savings!

$\theta(n^2)$ vs $\theta(n^{1.58..})$

Multiplication Algorithms

Kindergarten ? $3*4=3+3+3+3$	$n2^n$ Show
Grade School	n^2
Karatsuba	$n^{1.58\ldots}$
Fastest Known	$n \log n \log\log n$

Next…

1. Covered basics of a simple design technique (Divide-and-conquer) – Ch. 2 of the text.
2. Next, Strassen's algorithm
3. Later: more design and conquer algorithms: MergeSort. Solving recurrences and the Master Theorem.

Similar idea to multiplication in N, C

- Divide and conquer approach provides unexpected improvements

Naïve matrix multiplication

SimpleMatrixMultiply (A,B)

```
1. N ← A.rows
2. C ← CreateMatrix(n,n)
3. for i ← 1 to n
4.   for j ← 1 to n
5.       C[i,j] ← 0
6.       for k ← 1 to n
7.           C[i,j] ← C[i,j] + A[i,k]*B[k,j]
8. return C
```

- Argue that the running time is $\theta(n^3)$

First attempt and Divide & Conquer

Divide A,B into 4 n/2 x n/2 matrices

- $C_{11} = A_{11} B_{11} + A_{12}B_{21}$
- $C_{12} = A_{11} B_{12} + A_{12}B_{22}$
- $C_{21} = A_{21} B_{11} + A_{22}B_{21}$
- $C_{22} = A_{21} B_{12} + A_{22}B_{22}$

Simple Recursive implementation. Running time is given by the following recurrence.

- $T(1) = C$, and for $n>1$
- $T(n) = 8T(n/2) + \theta(n^2)$
- $\theta(n^3)$ time-complexity

Strassen's algorithm

Avoid one multiplication (details on page 80)
(but uses more additions)

Recurrence:

- $T(1) = C$, and for $n>1$
- $T(n) = 7T(n/2) + \theta(n^2)$

- How can we solve this?
- Will see that $T(n) = \theta(n^{\lg 7})$, $\lg 7 = $ **2.8073….**