Correctness Proof – 2 (typos fixed)

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

```
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 $m \leq A[j]$,
$1 \leq j \leq length(A)$

```
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.

Analysis of Algorithms

- Measures of efficiency:
  - –Running time
  - –Space used
  - – others
- Efficiency as a function of input size (NOT value!)
  - –Number of data elements (numbers, points)
  - –Number of bits in an input number

    e.g. Find the factors of a number n,

    Determine if an integer n is prime

Model: What machine do we assume? Intel? Motorola? P4? Atom? GPU?

Factors affecting algorithm performance

Importance of platform

- Hardware matters (memory hierarchy, processor speed and architecture, network bandwidth, disk speed,…..)
- Assembly language matters
- OS matters
- Programming language matters

Importance of input instance

Some instances are easier (algorithm dependent!)

What is a machine-independent model?

- Need a generic model that models (approximately) all machines
- Modern computers are incredibly complex.
- Modeling the memory hierarchy and network connectivity generically is very difficult
- All modern computers are “similar” in that they provide the same basic operations.
- Virtually all computers today have at most eight processors. The vast majority have one.

The RAM model

- Generic abstraction of sequential computers
- RAM assumptions:
  - – Instructions (each taking constant time), we usually choose one type of instruction as a **characteristic** operation that is counted:
    - Arithmetic (add, subtract, multiply, etc.)
    - Data movement (assign)
    - Control (branch, subroutine call, return)
    - Comparison
  - – Data types – integers, characters, and floats
  - – Ignores memory hierarchy, network!

Can we compute the running time on a RAM?

- Do we know the speed of this generic machine?
- If we did, will that say anything about the running time of the same program on a real machine?
- What simplifying assumptions can we make?

Idea: efficiency as a function of input size

- Want to make statements like, “the running time of an algorithm grows linearly with input size”.
- Captures the nature of growth of running times, NOT actual values
- Very useful for studying the behavior of algorithms for LARGE inputs
- Aptly named Asymptotic Analysis

Importance of input representation

Consider the problem of factoring an integer n

Note: Public key cryptosystems depend critically on hardness of factoring – if you have a fast algorithm to factor integers, most e-commerce sites will become insecure!!

Trivial algorithm: Divide by 1,2,…, n/2 (n/2 divisions)

aside: think of an improved algorithm

Representation affects efficiency expression:
Let input size = S.

Unary: 1111…..1 (n times) -- S/2 multiplications (linear)
Binary: $\log_2$ n bits -- 2^{S-1} multiplications (exponential)
Decimal: $\log_{10}$ n digits -- $10^{S-1}/2$ multiplications (exponential)

Analysis of Find-max

COUNT the number of cycles (**running time)** as a function of the **input size**

Find-max (A)	cost	times
1. max ← A[1]	c_1	1
2. for j ← 2 to length(A)	c_2	n
3. do if (max < A[j])	c_3	n-1
4. max ← A[j]	c_4	0≤k≤n-1
5. return max	c_5	1

Running time (upper bound): $c_1 + c_5 – c_3 – c_4 + (c_2 + c_3 + c_4)n$
Running time (lower bound): $c_1 + c_5 – c_3 – c_4 + (c_2 + c_3)n$
Q: What are the values of c_i?

Best/Worst/Average Case Analysis

- **Best case**: A[1] is the largest element.
- **Worst case**: elements are sorted in increasing order
- **Average case**: *?* Depends on the input characteristics

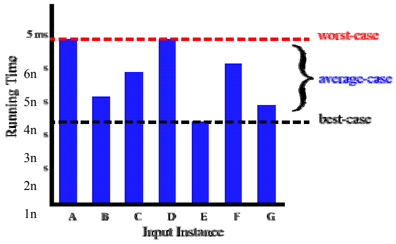
Q: What do we use?

A: Worst case or Average-case is usually used:

- Worst-case is an upper-bound; in certain application domains (e.g., air traffic control, surgery) knowing the **worst-case** time complexity is of crucial importance
- Finding the **average case** can be very difficult; needs knowledge of input distribution.
- Best-case is not very useful.

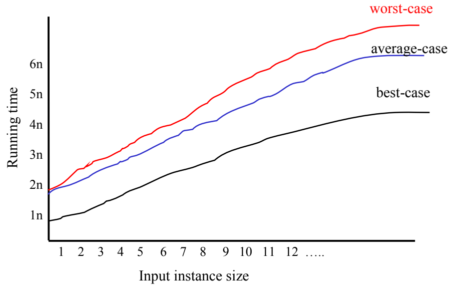
Best/Worst/Average Case (2)

- For a specific size of input *n*, investigate running times for different input instances:



Best/Worst/Average Case (3)

For inputs of all sizes:



Asymptotic notation : Intuition

Running time bound: $c_1 + c_5 – c_3 – c_4 + (c_2 + c_3 + c_4)n$
What are the values of c_i? machine-dependent

A simpler expression: $c_5 + c_6 n$ **[still complex].**

Q: Can we throw away the lower order terms?
A: Yes, if we do not worry about constants, and there exist constants c_7, c_8 such that $c_7 n \leq c_5 + c_6 n \leq c_8 n$, then we say that the running time is $\theta(n)$.

Need some mathematics to formalize this (LATER).

Q: Are we interested in small n or large?
A: Assume interested in large n – cleaner theory, usually realistic. Remember the assumption when interpreting results!

What does asymptotic analysis not predict?

- Exact run times
- Comparison for small instances
- Small differences in performance

So far…

1. Covered basics of algorithm correctness, analysis (Ch. 1 of the text).
2. Next: Another example of algorithm correctness and analysis (Ch 2). More about asymptotic notation (Ch. 3).

Note: Some slides in this lecture are adopted from Jeff Edmonds' slides.

Asymptotic notation - continued

Will do the relevant math later. For now, the intuition is:
1. O() is used for upper bounds "grows slower than"
2. Ω() used for lower bounds "grows faster than"
3. Θ() used for denoting matching upper and lower bounds. "grows as fast as"
   These are bounds on running time, not for the problem

The thumbrules for getting the running time are
1. Throw away all terms other than the most significant one -- Calculus may be needed
   e.g.: which is greater: n log n or $n^{1.001}$?
2. Throw away the constant factor.
3. The expression is Θ() of whatever's left.
   Asymptotic optimality – expression inside Θ() best possible.

A Harder Problem

INPUT: A[1..n] - an array of integers, k, 1 ≤k ≤length(A)
OUTPUT: an element m of A such that m is the k^{th} largest element in A.

Think for a minute

Brute Force: Find the maximum, remove it. Repeat k-1 times. Find maximum.

Q: How good is this algorithm?
A: Depends on k! Can show that the running time is Θ(nk). If k=1, asymptotically optimal.
Also true for any constant k.
If k = log n, running time is Θ(n log n). Is this good?
If k = n/2 (MEDIAN), running time is $\Theta(n^2)$.
Definitely bad! Can sort in O(n log n)!

Q: Is there a better algorithm? YES!

Space complexity

INPUT: n distinct integers such that n = 2^m-1, each integer k satisfies 0 ≤k ≤ 2^m-1.
OUTPUT: a number j, 0 ≤ j ≤ 2^m-1, such that j is not contained in the input.

Brute Force 1: Sort the numbers.
Analysis: Θ(n log n) time, Θ(n log n) space.

Brute Force 2: Use a table of size n, "tick off" each number as it is read.
Analysis: θ (n) time, θ (n) space.

Q: Can the running time be improved? No (why?)
Q: Can the space complexity be improved? YES!

Think for a minute

Space complexity – contd.

INPUT: n distinct integers such that n = 2^m-1, each integer k satisfies 0 ≤k ≤ 2^m-1.
OUTPUT: a number j, 0 ≤ j ≤ 2^m-1, such that j is not contained in the input.

Observation:

```
  000
  001
  010
  011
  100
  101
  110
+ 111
--------------
  000
```

Keep a running bitwise sum (XOR) of the inputs. The final sum is the integer missing.

Q: How do we prove this?

When you see a new problem, ask…

1. Is it similar/identical/equivalent to an existing problem?
2. Has the problem been solved?
3. If a solution exists, is the solution the best possible?
   May be a hard question :
   Can answer NO by presenting a better algorithm.
   To answer YES need to prove that NO algorithm can do better!
   How do you reason about all possible algorithms?
   (there is an infinite set of correct algorithms)
4. If no solution exists, and it seems hard to design an efficient algorithm, is it ***intractable***?

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.

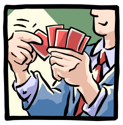
Loop invariants

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

"We maintain a subset of elements sorted within a list. The remaining elements are off to the side somewhere. Initially, think of the first element in the array as a sorted list of length one. One at a time, we take one of the elements that is off to the side and we insert it into the sorted list where it belongs. This gives a sorted list that is one element longer than it was before. When the last element has been inserted, the array is completely sorted."



English descriptions:

- Easy, intuitive.
- Often imprecise, may leave out critical details.

Insertion sort

```
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
```

Can you understand
The algorithm?
I would not know
this is insertion sort!

Moral: document code!

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:

<PreCond> & <code> ⇒ <PostCond>

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

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 each **for** loop, 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 each **for** loop, 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 k^{th} position so that *A*[*k-1*] ≤ *A*[*k*] ≤ *A*[*k*+1].

A[*1…j-1*] sorted + *A*[*j*] ⟶ *A*[*1…j*] 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 each **for** loop, 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"*☺

Analysis of Insertion Sort

Let's compute the **running time** as a function of the **input size**

	cost	times
`for j←2 to n`	c_1	n
`  do key←A[j]`	c_2	n-1
`      Insert A[j] into the sorted sequence A[1..j-1]`	0	n-1
`      i←j-1`	c_3	n-1
`      while i>0 and A[i]>key`	c_4	$\sum_{j=2}^{n} t_j$
`        do A[i+1]←A[i]`	c_5	$\sum_{j=2}^{n} (t_j - 1)$
`           i ← i-1`	c_6	$\sum_{j=2}^{n} (t_j - 1)$
`      A[i+1] ← key`	c_7	n-1

Analysis of Insertion Sort – contd.

- **Best case**: elements already sorted → t_j*=1*, running time = *Θ(n)*, i.e., *linear* time.
- **Worst case**: elements are sorted in inverse order → t_j*=j*, running time = *Θ*(n^2)*,* i.e., *quadratic* time
- **Average case**: t_j*=j/2,* running time = *Θ*(n^2)*,* i.e., *quadratic* time

- We analyzed insertion sort, and it has worst case running time $An^2 + Bn + C$, where $A = (c_5+c_6+c_7)/2$ etc.
- Q1: How useful are the details in this result?
- Q2: How can we simplify the expression?

Back to asymptotics……

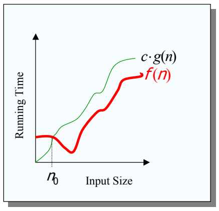
We will now look more formally at the process of simplifying running times and other measures of complexity.

Asymptotic analysis

- Goal: to simplify analysis of running time by getting rid of "details", which may be affected by specific implementation and hardware
  - like "rounding": $1{,}000{,}001 \approx 1{,}000{,}000$
  - $3n^2 \approx n^2$
- Capturing the essence: how the running time of an algorithm increases with the size of the input *in the limit*.
  - Asymptotically more efficient algorithms are best for all but small inputs

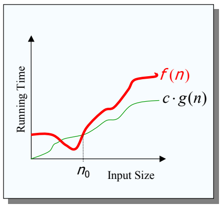
Asymptotic notation

- The "big-Oh" *O*-Notation
  - asymptotic upper bound
  - *f(n)* ∊ *O(g(n))*, if there exists constants *c* and n_0, *s.t.* $\mathbf{f(n)} \le \mathbf{c\ g(n)}$ for $n \ge n_0$
  - *f(n)* and *g(n)* are functions over non-negative integers
- Used for *worst-case* analysis



Asymptotic notation – contd.

- The "big-Omega" Ω–Notation
  - asymptotic lower bound
  - *f(n)* ∊ *Ω(g(n))* if there exists constants *c* and n_0, *s.t.* $\mathbf{c\ g(n)} \le \mathbf{f(n)}$ for $n \ge n_0$
- Used to describe *best-case* running times or lower bounds of algorithmic problems
  - E.g., lower-bound of searching in an unsorted array is *Ω(n)*.

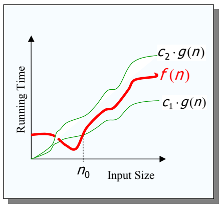


Asymptotic notation – contd.

- Simple Rule: Drop lower order terms and constant factors.
  - $50\ n \log n \in O(n \log n)$
  - $7n - 3 \in O(n)$
  - $8n^2 \log n + 5n^2 + n \in O(n^2 \log n)$
- Note: Even though 50 *n* log *n* ∊ O(n^5), we usually try to express a O() expression using as small an order as possible

Asymptotic notation – contd.

- The "big-Theta" Θ–Notation
  - asymptoticly tight bound
  - *f(n)* ∊ *Θ(g(n))* if there exists constants c_1, c_2, and n_0, *s.t.* $\mathbf{c_1\ g(n)} \le \mathbf{f(n)} \le \mathbf{c_2\ g(n)}$ for $n \ge n_0$
- *f(n)* ∊ *Θ(g(n))* if and only if *f(n)* ∊ *O(g(n))* and *f(n)* ∊ *Ω(g(n))*
- *O(f(n))* is often misused instead of *Θ(f(n))*



Until now…

1. Started formal definitions of asymptotic notation – Ch. 3 of the text.
2. Next, we will continue with iterative algorithms
3. Then, we move to recursive algorithms.

Asymptotic notation – contd.

- Two more asymptotic notations
  - "Little-Oh" notation *f(n)=o(g(n))* non-tight analogue of Big-Oh
    - For every *c*, there should exist n_0, s.t. $\mathbf{f(n)} \le \mathbf{c\ g(n)}$ for $n \ge n_0$
    - Used for **comparisons** of running times. If *f(n)* ∊ *o(g(n))*, it is said that *g(n) dominates f(n)*.
    - More useful defn:

$$\boxed{\lim_{n \to \infty} \frac{f(n)}{g(n)} = 0}$$

  - "Little-omega" notation *f(n)* ∊ *ω(g(n))* non-tight analogue of Big-Omega

Asymptotic notation – contd.

- (VERY CRUDE) Analogy with real numbers
  - $f(n) = O(g(n)) \cong f \le g$
  - $f(n) = \Omega(g(n)) \cong f \ge g$
  - $f(n) = \Theta(g(n)) \cong f = g$
  - $f(n) = o(g(n)) \cong f < g$
  - $f(n) = \omega(g(n)) \cong f > g$
- Abuse of notation: *f(n) = O(g(n))* actually means $f(n) \in O(g(n))$.

Points to ponder and lessons

Common uses:

$\Theta(1)$ – *constant.*
$n^{\Theta(1)}$ – *polynomial*
$2^{\Theta(n)}$ – *exponential*

Be careful!
$n^{\Theta(1)} \neq \Theta(n^1)$
$2^{\Theta(n)} \neq \Theta(2^n)$

- When is asymptotic analysis useful?
- When is it NOT useful?

Many, many abuses of asymptotic notation in Computer Science literature.

Lesson: Always remember the implicit assumptions…

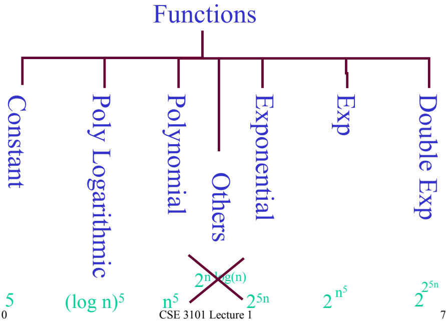
Comparison of Running Times

Running Time	Maximum problem size (n)		
	1 second	1 minute	1 hour
$400n$	2500	150000	9000000
$20n \log n$	4096	166666	7826087
$2n^2$	707	5477	42426
n^4	31	88	244
2^n	19	25	31

Classifying functions

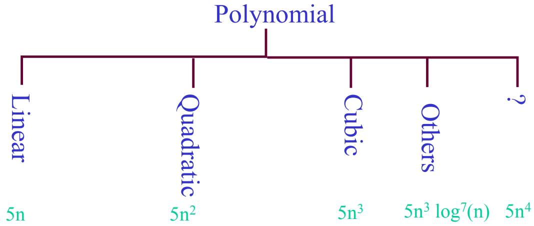
$T(n)$	10	100	1,000	10,000
$\log n$	3	6	9	13
$n^{1/2}$	3	10	31	100
n	10	100	1,000	10,000
$n \log n$	30	600	9,000	130,000
n^2	100	10,000	10^6	10^8
n^3	1,000	10^6	10^9	10^{12}
2^n	1,024	10^{30}	10^{300}	10^{3000}

Hierarchy of functions



Classifying Polynomials

Dominant term is of the form n^c



Logarithmic functions

- $\log_{10}n$ = # digits to write n
- $\log_2 n$ = # bits to write n
  $= 3.32 \log_{10} n$

Differ only by a multiplicative constant.

- $\log(n^{1000}) = 1000 \log(n)$

Poly Logarithmic (a.k.a. polylog)

$(\log n)^5 = \log^5 n$

Crucial asymptotic facts

Logarithmic << Polynomial

$\log^{1000} n \ll n^{0.001}$ For sufficiently large n

Linear << Quadratic

$10000\, n \ll 0.0001\, n^2$ For sufficiently large n

Polynomial << Exponential

$n^{1000} \ll 2^{0.001\, n}$ For sufficiently large n

Are constant functions constant?

The running time of the algorithm is a “constant”
It does not depend **significantly**
on the size of the input.

Write $\theta(1)$.

Yes • 5

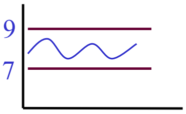
Yes • 1,000,000,000,000

Yes • 0.0000000000001

No • -5

No • 0

Yes • $8 + \sin(n)$



Lie in between

Polynomial Functions

Quadratic

- n^2
- $0.001\, n^2$
- $1000\, n^2$
- $5n^2 + 3000n + 2\log n$

Lie in between

Polynomial

- n^c
- $n^{0.0001}$
- n^{10000}
- $5n^2 + 8n + 2\log n$
- $5n^2 \log n$
- $5n^{2.5}$

Lie in between

Exponential functions

- 2^n
- $2^{0.0001\, n}$
- $2^{10000\, n}$
- $8^n \quad = 2^{3n}$
- $2^n / n^{100} \quad > 2^{0.5n}$
- $2^n \cdot n^{100} \quad < 2^{2n}$

$$2^{0.5n} > n^{100}$$
$$2^n = 2^{0.5n} \cdot 2^{0.5n} > n^{100} \cdot 2^{0.5n}$$
$$2^n / n^{100} > 2^{0.5n}$$

Proving asymptotic expressions

Use definitions!

e.g. $f(n) = 3n^2 + 7n + 8 = \theta(n^2)$

$f(n) \in \Theta(g(n))$ if there exists constants c_1, c_2, and n_0, s.t.
$c_1\, g(n) \le f(n) \le c_2\, g(n)$ for $n \ge n_0$

Here $g(n) = n^2$

One direction ($f(n) = \Omega(g(n))$ is easy

$c_1\, g(n) \le f(n)$ holds for $c_1 = 3$ and $n \ge 0$

The other direction ($f(n) = O(g(n))$ needs more care

$f(n) \le c_2\, g(n)$ holds for $c_2 = 18$ and $n \ge 1$ (CHECK!)

So $n_0 = 1$

Proving asymptotic expressions – contd.

Caveats!

1. constants c_1, c_2 MUST BE POSITIVE.

2. Could have chosen $c_2 = 3 + \varepsilon$ for any $\varepsilon > 0$. WHY?

-- because $7n + 8 \leq \varepsilon n^2$ for $n \geq n_0$ for some sufficiently large n_0. Usually, the smaller the ε you choose, the harder it is to find n_0. So choosing a large ε is easier.

3. Order of quantifiers

$\exists c_1\ c_2\ \exists n_0\ \forall\ n \geq n_{0,}\ c_1 g(n) \leq f(n) \leq c_2 g(n)$

vs

$\exists n_0\ \forall\ n \geq n_0\ \exists c_1\ c_{2,}\ c_1 g(n) \leq f(n) \leq c_2 g(n)$

-- allows a different c_1 and c_2 for each n. Can choose $c_2 = 1/n$!! So we can “prove” $n^3 = \Theta\ (n^2)$.

Why polynomial vs exponential?

Philosophical/Mathematical reason – polynomials have different properties, grow much slower; mathematically natural distinction.

Practical reasons

1. almost every algorithm ever designed and every algorithm considered practical are very low degree polynomials with reasonable constants.

2. a large class of natural, practical problems seem to allow only exponential time algorithms. Most experts believe that there do not exist any polynomial time algorithms for any of these; i.e. P ≠ NP.

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

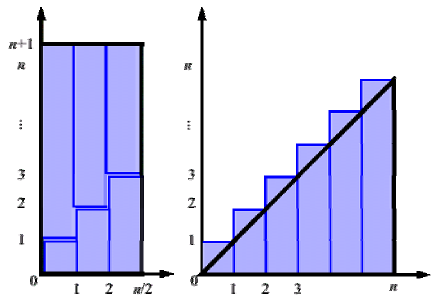
- Geometric progression
  - given an integer n_0 and a real number $0 < a \neq 1$

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

  - geometric progressions exhibit exponential growth
- Arithmetic progression

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

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

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

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

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

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

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: $\sum kx^k$

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

- 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 j^{th} 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

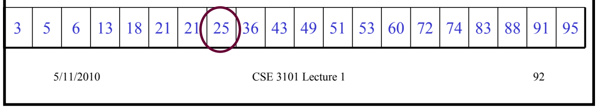
Narrowed the search space, e.g. Binary search

- 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

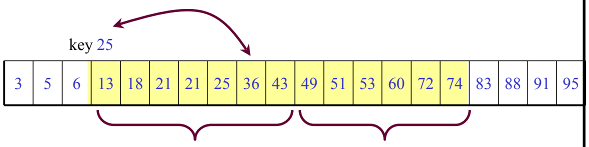
- Cut sublist in half.
- Determine which half the key would be in.
- Keep that half.

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.



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

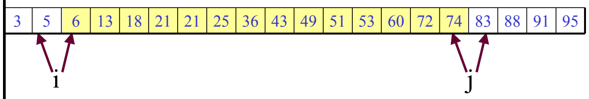
If key ≤ mid,
then key is in
left half.

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



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

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

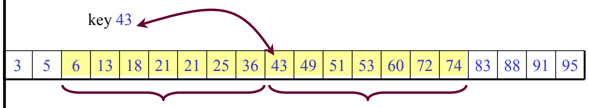
- If the sublist has even length, which element is taken to be mid?

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].



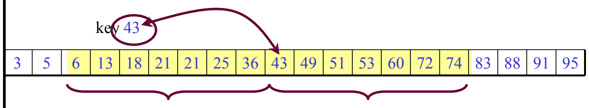
3	5	6	13	18	21	21	25	36	43	49	51	53	60	72	74	83	88	91	95

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].



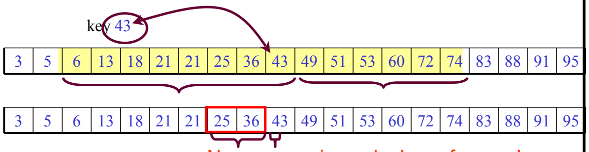
3	5	6	13	18	21	21	25	36	43	49	51	53	60	72	74	83	88	91	95

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].



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

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

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 usually 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 (⟨L(1..n), key⟩)
⟨pre−cond⟩: ⟨L(1..n), key⟩ is a sorted list and key is an element.
⟨post−cond⟩: 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
        ⟨loop−invariant⟩: If the key is contained in L(1..n), then
            the key is contained in the sublist L(i..j).

        exit when j ≤ i
        mid = ⌊(i+j)/2⌋
        if(key ≤ 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
```

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)

<x,y> ⇒ <x-y,y>
⇒ <x-2y,y>
⇒ <x-3y,y>
⇒ <x-4y,y>
⇒ <x-iy,y>
⇒ <x rem y,y>
= <x mod y,y>
⇒ <y,x mod y>

But x mod 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>

Little progress
Lots of progress

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

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?

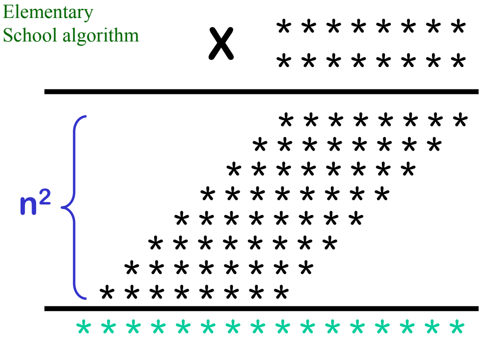
Gauss' idea

- $m_1 = ac$
- $m_2 = bd$
- $A_1 = m_1 – m_2 = ac-bd$
- $m_3 = (a+b)(c+d) = ac + ad + bc + bd$
- $A_2 = m_3 – m_1 – m_2 = ad+bc$
- **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

Q: Is there a faster algorithm?

A: YES! Use divide-and-conquer.

Divide and Conquer

Intuition:

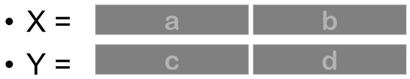
•**DIVIDE** my instance to the problem into smaller instances to the same problem.

•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



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

```
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

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

Technique 1: Guess and verify

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

Holds for n=1

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

$= 2n^2 - n$

Time complexity of MULT

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

Technique 2: Expand recursion

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

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

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

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

$= \ldots\ldots\ldots$

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

GUESS

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

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

$= 2\,n^2 - n$ [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  e$2^n$ + (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 $2^k = n$

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

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

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

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

Not just 25% savings!

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

Multiplication Algorithms

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