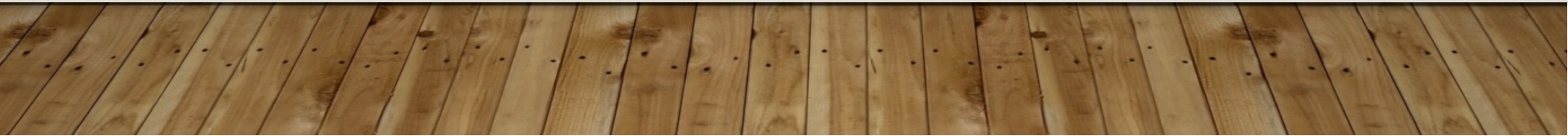
Computer Science/Mathematics 3804

Design and Analysis of Algorithms I



Description&Prerequisites

Course Description

An introduction to the design and analysis of algorithms.

Topics include: recurrence relations, sorting and searching, divide-and-conquer, dynamic programming, greedy algorithms, NP-completeness.

Prerequisites

COMP 2002 or COMP 2402, and either COMP 1805 or both of MATH 2007 and MATH 2108, or equivalents.

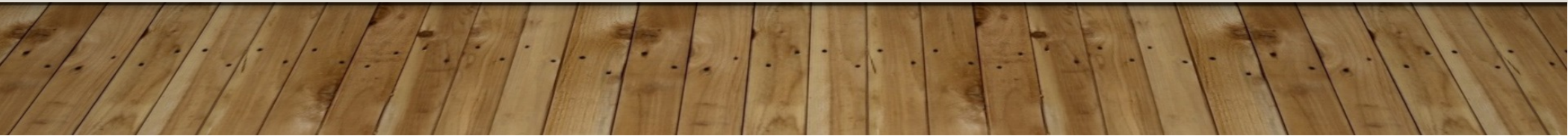
To help students achieve their individual objectives in this course, we will do an initial assessment test (anonymous) and provide students with three 30-minutes background review sessions. Each session will be offered three times for your convenience. They are not mandatory.

Assessment

Since 3804 traditionally has a large failure rate, we want to help by trying to see why this is the case.

This in-class assessment is for you to assess whether you have the right background information.

We then offer a review of key material.



Highly Recommended background prep, sessions

More details on Brightspace

Elementary Data Structures
Monday Sept. 12th 14:30-15:55

Algorithms and Complexity
Monday Sept. 19th 14:30-15:55

Counting and Recurrences
Monday Sept. 26th 14:30-15:55

Office Hours TAs				
TA	Office			
Stephen Lu		stephenluu@cmail.carleton.ca	no office hours	
Zoltan Kalnay	online	ZoltanKalnay@cmail.carleton.ca		
Heny Dave	online	HENYDAVE@cmail.carleton.ca		
Jag Zhang	online	JAGZHANG@cmail.carleton.ca		

Office Hours

Name	Time
Zoltan	Mondays 14:00-15:30
Heny	Tuesdays 10:00-11:30
Jag	Thursdays 12:00-13:30

Some changes might still occur please check Brightspace for announcements.

Office Hours:

Professor J.-R. Sack

Wednesdays 14:00-15:00 or by appointment

- PRIMARY:
- Introduction to Algorithms (3rd Edition) by
  - Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest and Clifford Stein (2009). ISBN 0-262-03384-4. Published by MIT Press.
  - (the authors are working on a fourth revision)

- alternate reference - not as detailed,
  but I base some discussion on material from
- Algorithms by
  - S. Dasgupta, C.H. Papadimitriou, and U.V.Vazirani, (2007) Published by McGrawHill may also available be available online

Tests/Assignments/Final

Assignments will be submitted online in pdf via Brightspace (details to be provided by me). Typed assignments are preferred. Figures may be drawn and scanned in.

It is your responsibility to ensure that the quality of the pdf is good so that the TAs can read them easily. Scanners are accessible at many locations on Campus.

Pictures taken with mobile devices are acceptable if they are of top quality.

Tests/Assignments/Final

Please note that tests and examinations in this course may use services provided by Scheduling and Examination Services. This will determine if the test/exams are in-person or electronically

Details on the test/exam will follow as soon as they are available.

Component	Weight	Due Dates (estimated – final dates as per announcements)
Assignment 1	10%	Sunday, October 2nd 23:59 online
Assignment 2	10%	Sunday, October 23trd 23:59 online
Test	20%	November 2nd
Assignment 3	10%	Sunday, November 13th 23:59 online
Assignment 4	10%	Sunday, December 4th 23:59 online
Final Exam	40%	by central scheduling

Academic Integrity/Plagiarism/...

Student Academic Integrity Policy

Every student should be familiar with the Carleton University student academic integrity policy. A student found in violation of academic integrity standards may be awarded penalties which range from a reprimand to receiving a grade of F in the course or even being expelled from the program or University. Some examples of offences are: plagiarism and unauthorized co-operation or collaboration. Information on this policy may be found in the Undergraduate Calendar.

Plagiarism

As defined by Senate, "plagiarism is presenting, whether intentional or not, the ideas, expression of ideas or work of others as one's own". Such reported offences will be reviewed by the office of the Dean of Science.

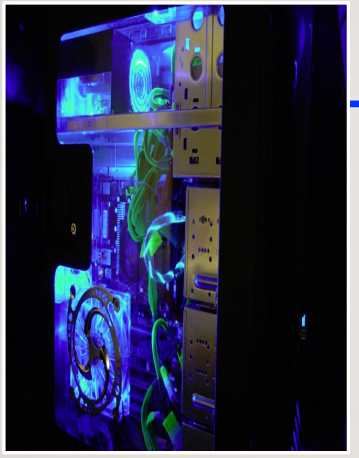
Unauthorized Co-operation or Collaboration

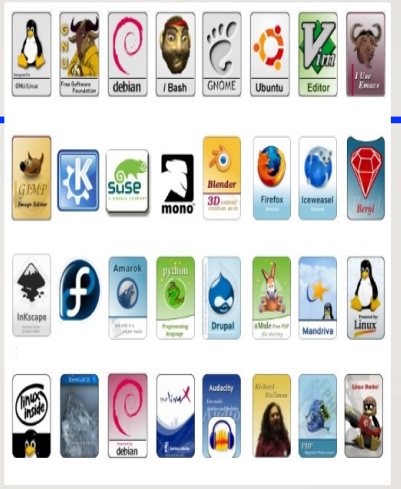
Senate policy states that "to ensure fairness and equity in assessment of term work, students shall not co-operate or collaborate in the completion of an academic assignment, in whole or in part, when the instructor has indicated that the assignment is to be completed on an individual basis".

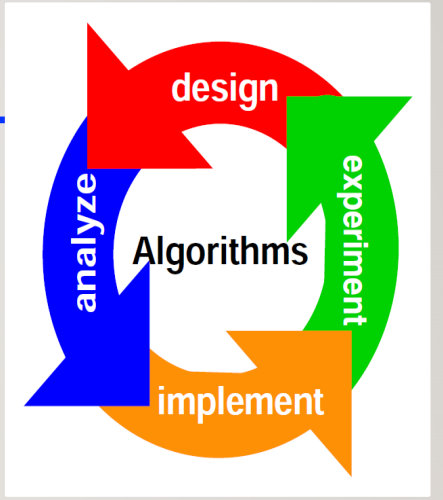
Other information

The course outline contains all other important items of information, please read it!

COMPUTER SCIENCE







Hardware

- Hardware
- Communication Infra-structure

Software

- Programs
- Software Infra-structure

Efficient algs+data str.

- Algorithms
- Data Structures

Introduction

"Algorithms change/d the world"

Precise instructions of how to add, subtract, multiply, divide, extract square roots, get digits of π,...

[Al Khwarizmi, AD 600, Baghdad]

Unambiguous, precise, mechanical, efficient and correct

Definition "Algorithm"

Informal:

An algorithm is any well-defined computational procedure that takes some value, or set of values, as input and produces some value, or set of values, as output.

An algorithm is thus a sequence of computational steps that transforms the input string into an output string.

Three questions to ask - algorithm

After we understand what it does or is supposed to do.

1. Is it correct?
2. How much time and/or space does it take, as a function of the input size?
3. Can we do better?

Correctness

Definition:

— An algorithm is correct if, for every input instance, it halts with the correct output.

— A correct algorithm solves the problem.

— An incorrect algorithm therefore:

- ... <see class>
- ... <see class>

Usefulness of incorrect algorithms

Sometimes, incorrect algorithms are useful or required

The algorithm

- must halt
- should give the correct result "mostly”
- Often algorithms may perform well in practice, but may fail sometimes.

Are they useful or desirable? Heuristics are examples. (see class for details)

Time and Space

We will measure time and space in "big oh”

Time:

— elementary operations count

Space usage:

Counting units of

— Internal memory and/or

— Disc -Tape

Establishing the Correctness of an Algorithm

By definition, it must: halt and return the correct result.

(1) We must show that it always halts; that part is usually easy.

(2) How to show that it returns the correct result?

That is sometimes interesting.

Pineapple

The number of spirals going in each direction is a Fibonacci Number.

Here, there are 13 spirals that turn clockwise and 21 curving counterclockwise.

In sunflowers, the number of clockwise and counterclockwise spirals will always be consecutive Fibonacci Numbers like 21 and 34 or 55 and 34.

A Sequence

- The first 21 numbers are:

  0, 1,1, 2, 3, 5, *8,* 13, 21, 34, 55, 89, 144, 233
  377, 610, 987,1597, 2584, 4181, 6765

What can we say about this sequence?

Many things

- Non-negative
- Increasing
- Rapidly increasing almost as fast as 2^n, in fact a good approximation is: $20^{0.694n}$
- If we call F_n the n^{th} number then
  $F_n = F_{n-2} + F_{n-1}$ where $n>1$ and
  $F_0 = 0$ and $F_1 = 1$.

Fibonacci Numbers

SCIENCEphotOLIBRARY

F_n is called the n^{th} Fibonacci number.

Fibonacci was living between 1170 and 1250.

Fibonacci Occurrences in Nature

Number of Petals	Flower
3 petals (or 2 sets of 3)	lily (usually in 2 sets of 3 for 6 total), iris
5 petals	buttercup, wild rose, larkspur, columbine (aquilegia), vinca
8 petals	delphinium, coreopsis
13 petals	ragwort, marigold, cineraria
21 petals	aster, black-eyed susan, chicory
34 petals	plantain, daisy, pyrethrum
55 petals	daisy, the asteraceae family
89 petals	daisy, the asteraceae family

0, 1, *1, 2, 3, 5, 8, 13, 21,* 34, 55, 89, 144,...

Algorithm I for Fibonacci

Function fib1(n)

```
if n = 0 return 0
if n = 1 return 1
return fib1(n-1) + fib1(n-2)
```

The Three Questions

1. Is the algorithm correct?
2. How much time does it take to compute?
3. Can we do better?

1. is easy, that it follows simply by definition of the Fibonacci sequence. But let us do this formally.

1. Is the algorithm correct?

1. It is a recursive algorithm. So, a proof by induction seems the best way.

   First observe that the algorithm always halts.

   base cases(I):

   if n = 0 the algorithms returns 0 & stops by

   definition, $F_0 = 0$, so: correct

   if n=1 the algorithm also returns 1 & stops by

   definition $F_1 = 1$ so: correct

1. Is the algorithm correct? cont'd

1. Assume therefore now that n>1 and that the algorithm has correctly computed fib1(0),…, fib1(n-1).

   Then, the algorithm returns

   "return fib1(n-1) + fib1(n-2)"

   This recursively calls fib1(n-l) and fib1(n-2) and adds these two values. By induction hypothesis, fib1(n-l) and fib1(n-2) have been correctly computed.

   By definition of F_n, $F_n = F_{n\_2} + F_{n-1}$ for n≥2. Thus, the algorithm is correct.

2. How much time does it take to compute?

- The time is a function of n, let us call it T(n)
- $T(n) \leqq 2$, for $n < 1$ why?
- For larger values of n, i.e., $n \geq 2$

$$T(n) = T(n-1) + T(n-2) + 3$$

so, we can see that $T(n) \geq F(n)$

bad news?!

to compute T(200) > F(200) > 2^{138} steps would be required which is huge.

E.g., the Japanese Fugaku can do 442.01 petaFlop. To compute F(200) would longer than the earth is expected to live.

1 quadrillion calculations per second is a Peta Flop.

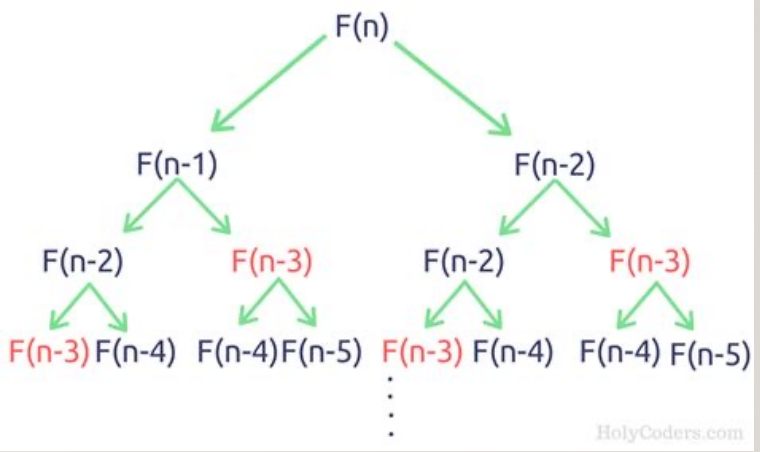
SO, CAN WE DO BETTER?

3. Can we do better?

Why is fib1 so bad?

Function fib1(n)

```
if n = 0 return 0
if n = 1return 1
return fib1(n-1) + fib1(n-2)
```



A SECOND ATTEMPT FIB2

Function fib2(n)

```
if n = 0 return 0
create an array f[0 .. n]
f[0] = 0, f[1] = 1
for i = 2 … n
        f[i] = f[i-1] + f[i-2]
return f[n]
```

Replace recursive calls
by array accesses.
Why recompute every time
when you can just do a look-up?

The three questions

1. Is the algorithm correct?
2. How much time does it take to compute?
3. Can we do better?

1. Is the algorithm correct?

   again, here the correctness follows from the above as the recursive call is replaced by an equivalent array access.

2. How much time does it take to compute?

Except of the small number of constant-time operations the main work is carried out in the loop.

— the loop body is one addition

— and is executed n-1 times

So, the total number of addition etc. operations taken by fib2 is linear in n. Are all operations O(1)**?**

Careful

We get that fib2 executes a linear number of additions. However, the numbers are huge.

Our standard model of analysis assumes that the numbers have a constant number of bits (32 or 64...) or at most log n (minimum required to store the number n).

Here the numbers are about 0.694n bits long!

Such numbers cannot be added in one step!

Adding 2 n-bit numbers takes time proportional to n. (bit operations) Such complications while rare, need to be carefully taken into consideration!

Conclusion

Lemma: The running time of fib2(n) has O(n) operations, some of which take O(n) bits to manipulate.

[This is more detailed than we usually argue in our "O" analysis because we have words which are longer than O(log n) bits.]

What is the running time to compute the Fibonacci then?

- There is a difference in required run-time between computing the n^{th} Fibonacci number only, or all Fibonacci numbers up to the n^{th}.

3. Can we do better? YES, by reduction to fast matrix multiplication (not discussed here) the n^{th} Fibonacci number can be computed fast.

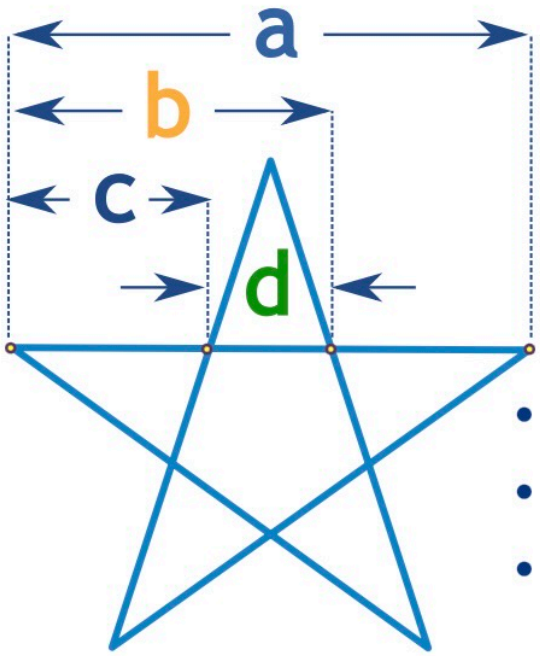
Golden ratio

- Two numbers a,b are said to be in the "golden ratio, Φ," if

$$\frac{a+b}{a} = \frac{a}{b} = \Phi$$

Many buildings and artworks have the Golden Ratio in them, such as the Parthenon in Greece, but it is not really known if it was designed that way.

Golden Ratio



Pentagram

No, not witchcraft! The pentagram is more famous as a magical or holy symbol. And it has the Golden Ratio in it:

- a/b = 1.618...
- b/c = 1.618...
- c/d = 1.618...

see https://www.mathsisfun.com/numbers/golden-ratio.html

Golden ratio (equivalent)

- Two numbers a,b are in the "golden ratio, Φ," if

$$\frac{a+b}{a} = 1 + \frac{b}{a}$$

So, since b/a=1/Φ we get: 1+ 1/Φ= Φ

CALCULATING THE VALUE

First **guess** its value, then do this calculation again and again:

- A) divide 1 by your value (=1/value)
- B) add 1
- C) now use *that* value and start again at A

With a calculator, just keep pressing "1/x", "+", "1", "=", around and around.

I started with 2 and got this:

value	1/value	1/value + 1
2	1/2 = 0.5	0.5 + 1 = **1.5**
1.5	1/1.5 = 0.666...	0.666... + 1 = **1.666...**
1.666...	1/1.666... = 0.6	0.6 + 1 = **1.6**
1.6	1/1.6 = 0.625	0.625 + 1 = **1.625**
1.625	1/1.625 = 0.6153...	0.6154... + 1 = **1.6153...**
1.6153...		

GOLDEN RATIO

Now solve $1 + 1/\Phi = \Phi$

$\Phi + 1 = \Phi^2$ rewritten as: $\Phi^2 - \Phi - 1 = 0$

By using the solutions to a quadratic formula,

$$ax^2 + bx + c \qquad x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

For $a = 1$, $b = -1$ and $c = -1$

we get:

$\Phi_1 = (1+\sqrt{5})/2$ and $\Phi_2 = (1-\sqrt{5})/2$

Because Φ is a ration between positive quantities, Φ is positive. $\Phi = 1.618033988\ldots$

GOLDEN RATIO AND FIBONACCI

- By induction, we can prove:

$$F_i = (\Phi_1^i - \Phi_2^i)/\sqrt{5}$$

- Note that $|\Phi_2^i|/\sqrt{5} < 1/\sqrt{5}$ (as $|\Phi_2|<1$)

  thus $|\Phi_2^i|/\sqrt{5} < ½$ and therefore:

- $F_i = \text{floor}(\Phi_1^i/\sqrt{5} + ½)$
- So: the i^{th} Fibonacci number is $\Phi_1^i/\sqrt{5}$ rounded to the nearest integer. Exponential growth!

Recall Big-0

Let f(n) and g(n) be functions from the positive integers to the reals. We say that
f(n) = O(g(n)) if there is a constant c>0 such that

f(n) < c*g(n).

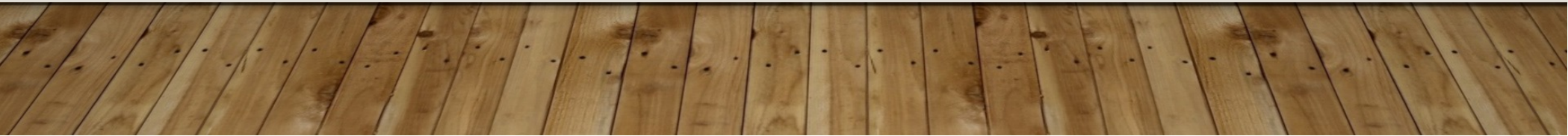
This means that f grows no faster than g.

Also, the constant c allows us to ignore small values of n.

Course requirement

I will assume that you are familiar with big-O.

if not, please ! please read up fast.



A COMMON STRATEGY: LOOP INVARIANT

- For algorithms that execute, e.g., a main loop (or a recurrence)
- Set up a **Loop invariant,** say L
- Say the algorithm executes a loop n times.
- **L(i):** statement about the algorithm is true before the i^{th} execution of the loop.
- **Base Case**: this is the base case, i.e., after initialization of the variables (if any), before entering the loop the first time. We need to show that for invariant is true for the base case.

Loop invariant continued

Termination: when the loop terminates, the truth of the invariant helps us establish the correctness of the algorithm.

Maintenance: prove that if L(i-1) is correct before the execution of the loop then L(i) is also true after the loop execution.

Insertion-Sort

```
INSERTION-SORT(A)
j:=2 // to make life easier to prove correctness //
for j:=2 to length of A
    key := A[j]
    // insert A[j] into A[1..j-1] //
    i:=j-1
    while (i>0 and A[i] > key)
        A[i+1] := A[i]
        i:=i -1
    A[i+1] := key
```

Example

- In class

Halting

The algorithm halts because it executes a finite number of statements a finite number of times, where each statement takes finite time to execute.

Setting up the invariant

- INVARIANT L[j]

At the start of each iteration of the **for** loop the subarray A[1...j-1] consists of the elements originally stored in A[1..j-1] but now sorted.

Notation: A[i..j] is a subarray of array A consisting of the elements A[i], A[i+1],..., A[j]

Correctness of Insertion Sort

Base Case:

Before the algorithm enters the loop,

what do we know?

We know that

- the input is stored an array
- j = 2

Base case cont'd

"At the start of each iteration of the **for** loop the subarray A[1...j-1] consists of the elements originally stored in A[1..j-1] but sorted."

Since j=2, we need to show that

"At the start of each iteration of the **for** loop the subarray A[1…1] consists of the elements originally stored in A[1..1] but now sorted."

A[1..1]=A[1].

- A[1] contains the element originally in A[1]
- A[1] by itself is clearly sorted

Termination

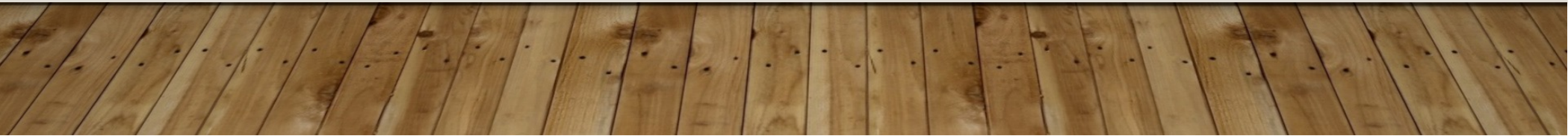
At **termination:**

If we have shown that L(j) is true then

"At the start of each iteration of the **for** loop the subarray A[1...j-1] consists of the elements originally stored in A[1..j-1], but now sorted."

when the loop terminates j > A.length, in fact j=A.length+1 as the loop-variable, j, is incremented by one each time.

So, the subarray A[1..A.length] is sorted and contains the elements originally stored in A. Thus the algorithm returns the correct result.



Loop maintenance

At the start of each iteration of the **for** loop the subarray A[1...j-1] consists of the elements originally stored in A[1..j-1] but now sorted.
Now, consider j to be incremented

Key := A[j]

Key and A[1..j-1] contain all elements of A[1..j]

Loop maintenance cont'd

i=j-1 // sets the limit of the subarray to be examined subsequently//

```
while (i>0 and A[i] > key)
    A[i+i] :=A[i]
    i:=i -1
A[i+1] := key
```

We will show separately that if A[1..i] is a sorted array and key is a value then after the execution of the algorithm (above) A[1..i+1] is a sorted array containing the values of A[1..i] and key.

Since i=j-1, by loop-invariant, we have that A[1..j-1] is a sorted array of the values originally stored in A[1..j-1]. Key = A[j] so, after the above, A[1..j] is a sorted array of the values originally stored in A[1..j]

PROVING THE CORRECTNESS OF THE WHILE-LOOP

Input: A[1..j-1] a sorted subarray of numbers and A[j] possibly in the wrong relative order w.r.t. A[1…j-1]

Output: A sorted array, A[1..j], of the numbers originally stored in A[1..j]

```
        key := A[j]      and      i:=j-1
    while ( i>0 and A[i] > key)
        A[i+1] := A[i]
        i:=i – 1
    A[i+1] := key
```

LOOP INVARIANT:

at the beginning of each iteration of the while-loop

- A[i+2],…,A[j] are each greater than key and
- A[i+2..j] contains the sorted values originally stored in A[i+1..j-1]
- A[1..i] is untouched by the while loop

CORRECTNESS WHILE-LOOP CONT'D

- **Base-case**
- "at the beginning of each iteration of the while-loop

(A) A[i+2],…,A[j] are each greater than key and

(B) A[i+2..j] contains the sorted values originally stored in A[i+1..j-1]

- (B) A[1..i] is untouched by the while loop"

- Before the first time through the while loop, i:=j-1
  thus i+2 = j+1 and since A[j+1] does not exist the statement (A) is true.
- Also (B) is true (A[1..i] is untouched) since we did enter the loop so far.

CORRECTNESS WHILE-LOOP CONT'D

Maintenance

- "at the beginning of each iteration of the while-loop
- A[i+2],…,A[j] are each greater than key and
- A[i+2..j] contains the sorted values originally stored in A[i+1..j-1]
- A[1..i] is untouched by the while loop"

Assume that the loop invariant is true and now we enter the while loop,

- Since A[i]>key we move A[i] to A[i+1] so A[i+1]> key
- After we decrement i thus, now A[i+2] >key (in addition to the other elements previously considered….A[j])
- The subarray of considered elements remains sorted
- Subarray A[1 ..i] remains untouched and has shrunk by 1.

Correctness while-loop cont'd

Termination

"at the end of final iteration of the while-loop A[i+2],...,A[j] are each greater than key and A[i+2..j] are the sorted values originally stored in A[i+1..j-l]

A[1..i] is untouched by the algorithm"

Two cases arise:

1) The loop condition is violated because i = 0 or
2) A[i] ≤ key

Assume **1) i.e., i=0** then since A[i+2 ... j] is the sorted array of values originally stored in A[i+1.. j-1], for i=0, this means A[2..j] is the sorted array of values from A[1...j-1] and A[2],...A[j] are all greater than key. The final assignment statement A[i+1]=key puts A[1]=key thus A[1...j] is the sorted array of all values originally stored in A[1...j].

Correctness while-loop cont'd

Termination

"at the end of final iteration of the while-loop A[i+2],...,A[j] are each greater than key and A[i+2..j] are the sorted values originally stored in A[i+l..j-1]

A[1..i] is untouched by the algorithm"

Now assume 2), i.e., A[i] ≤ key

By loop-invariant, A[i+2 ... j] is the sorted array of the values originally stored in A[i+1.. j-1]; all elements are > key.

A[1..i] is untouched since A[1 ...i] was sorted before and A[i] ≤ key, by transitivity, all elements in A[1...i] are sorted and ≤ key.

The final assignment puts key into its correct position, i.e., A[i+1] = key thus the entire array A[1..j] is now sorted.

2. Time and Space Complexity

The algorithm executes two nested loops, each loop is executed at most O(n) times. The inner loop has O(1) time complexity. Except for the inner loop, the outer loop is O(1).

Therefore, the total time complexity is $O(n^2)$.

Except for a constant number of additional storage locations, the only storage used is the input array; the algorithm is "in-place".

Thus, the space complexity is O(n).

3. Can we do better?

Yes, we already know that e.g., MergeSort is an O(n log n) algorithm using O(n) space.

I recall the algorithm and its analysis briefly as another example of how to prove correctness.

MergeSort

```
MERGE-SORT(A, p, r)
// sorts a subarray A[p..r] of numbers//
    If p<r
        q = floor{(p+r)/2}
        MERGE-SORT(A,p,q)
        MERGE-SORT(A,q+1,r)
        MERGE(A,p,q,r)
```

Algorithm MERGE

MERGE(A,p,q,r)

//Forms a sorted (sub)array A[p..r] from two sorted (sub)arrays A[p..q] and A[q+1..r]; L, R are two temporary arrays//

$n_1=q-p+1$; $n_2=r-(q+1)+1=r-q$

//sizes of the "input arrays"//

Copy A[p..q] into L[1.. n_1]; add L[n_1+1]=∞

Copy A[q+1..r] into R[1.. n_2]; add R[n_2+1]=∞

i=1; j =1

Algorithm MERGE cont'd

```
                    MERGE(A,p,q,r)
For k= p to r      //go through all elements //
    if L[i] ≤ R[j]
        A[k] = L[i]
        i = i+1
    else A[k] = R[k]
        j = j+1
```

First we analyze MERGE

1. Correctness

Good exercise (when you are done then look at the book for the solution)

2. Time Complexity

Let n= r-p+1

MERGE has three loops examining O(n) elements

the first two copy array elements into L and R, resp.

the last executes the loop r-p+1 = O(n) times and performs O(1) operations at a cost of O(1) each; so total: O(n)

First we analyze MERGE cont'd

2. Space Complexity

   there are three arrays of size O(n)

3. Can we do better?

   no – we need to examine all elements thus Ω(n) is a lower bound. (Lower bound means we cannot do better, i.e., no algorithm can be designed which performs the task in a smaller time bound.) Same for space (we must store the elements as input; the additional arrays are all of the same size which does not change the space complexity in "O").

Question

- What would happen if we were to attempt the MERGE procedure in the original array A?
- Any impact on the complexity?

Divide&Conquer Algorithm D&C

- Recall D&C
- Break problem into subproblems that are smaller instances of the same type of problem
- Recursively solve the subproblems
- Combine the answers to the subproblems into an answer to the original problem

Here: D&C MERGE-SORT

- Break problem into subproblems that are smaller instances of the same type of problem
  - "the subarrays: A[p..q] and A[q+1..r]"
- Recursively solve the subproblems
  - "MERGE-SORT(A,p,q) and MERGE-SORT(A,q+1,r)"
- Combine the answers to the subproblems into an answer to the original problem
  - "MERGE(A,p,q,r)"

1. Correctness of MERGE-SORT

There is no loop, but a straight recurrence. A proof by induction seems feasible.

Induction on the size of the array n=r-p+1.

Base Case: n=1 (i.e., r=p)

a single element array is sorted

Induction Hypothesis: assume that MERGE-SORT sorts any array of sizes 1,..., n-1, for n>2.

1. Correctness of MERGE-SORT cont'd

Show that it sorts any input array of size n

Since n>2, p<r thus the statements inside the "if-stement" are executed

With q, the array is split into two (roughly equal-sized) subarrays (A[p..q] and A[q+l..r])

Since there sizes are less than n, by induction and after calls to MERGE-SORT on them, they are correctly sorted.

The final step MERGE then sorts the two sorted subarrays (the correctness follows from the correctness of MERGE)

2. Complexity

Again let n = r-p+1

The space used is: O(n) because MERGE-SORT uses only a constant additional locations (note that the maintenance of the recurrence also takes some storage which is linear in n. This requires re-using of the additional array space used by MERGE! Otherwise, this is O(n log n)
MERGE has O(n) space usage as we said

2. Complexity cont'd

- Now, let us analyze the time complexity.
- It is given by this recurrence relation
- $T(n) = 2*T(n/2) + 0(n)$
  — Why? See class
- There are many ways to solve this recurrence.
- You will have seen some already before. I recall.

Solving the recurrence

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

a) By developing the recurrence:

Let us say it is exactly n not O(n) - makes notation easier

Otherwise, we can carry the constants through.

$T(n)=2T(n/2)+n=2[2T(n/4)+n/2]+n = 4T(n/4) + 2n$

$=2^2T(n/2^2) + 2*n = 2^2[2T(n/2^3)+n/2^2) + 2n$

$=2^3T(n/2^3) + 3*n = ...$

$= 2^iT(n/2^i) + i*n$, for all i=0,...,?

What is i maximally?

Solving recurrences cont'd

- i is maximally $\log_2 n$ because then $2^i = n$

$T(n) = 2^i T(n/2^i) + i*n = 2^{\log_2 n} T(n/2^{\log_2 n}) + (\log_2 n)*n$ since

$T(n/2^{\log_2 n}) = T(1) = 1$; $T(n) = 2^{\log_2 n} + (\log_2 n)*n = n + n*\log_2 n$

$= O(n*\log_2 n)$. We thus get:

Lemma: MERGE-SORT sorts n elements in O(n logn) time.

3. Can we do better?

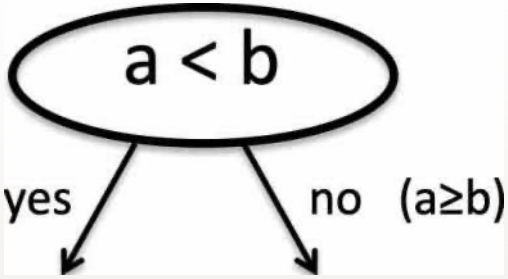
To answer this, we need to specify a precise model of computation. We will now count comparisons only. (In any sorting algorithm discussed, we compared elements to each other and then rearranged the array. So, this is natural to do. Not in all sorting algorithms mind you.)

We will show that no sorting algorithm can sort n arbitrary numbers in o(n log n) comparisons.

[this is little "oh"; informally, that means with fewer than n log n comparisons]

Comparisons

A comparison between two numbers a, b



Consider now sorting three numbers a,b,c

Sorting by comparisons

In class

comparison tree for sorting 3 elements

Sorting n elements counting comparisons only

Any comparison-based algorithm that sorts n elements has an execution that can be described by a comparison tree.

If we sort n elements any permutation of the n elements can appear as output thus as one of the leaves of the tree.

Thus the tree must have n! leaves.

Sorting n elements counting comparisons only - depth of tree

- The tree is binary.
- What is the depth of a binary tree on m nodes?

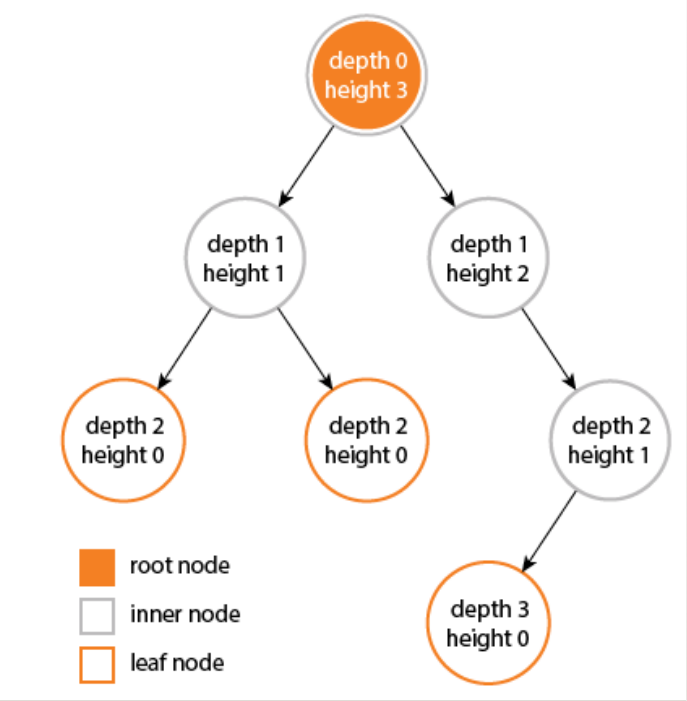
Depth: number of edges from the node to the root.

H**eight** of a node is the number of edges on the

longest path from the node to a leaf.

Height of a **tree** is the height of its root node, or equivalently,

the depth of its deepest node.



Sorting n elements counting comparisons only - depth of tree

- What is the depth of a binary tree on m nodes?

  Thus, the depth > log(n!)

  log(n!) > c*n log n, for some c>0

  Why? *In-class*

a better bound for n!

Stirling's approximation provides a more precise bound

Stirling's formula: $n! \approx \sqrt{2\pi n}\left(\frac{n}{e}\right)^n$

Cont'd

- The depth of the tree corresponds to the number of operation (of type comparison) we need to do.

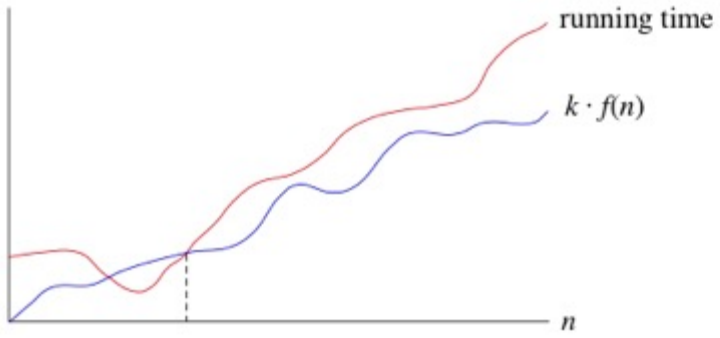
Theorem: Any comparison-based sorting algorithm has an Ω(n log n) lower bound to sort n elements.

Corollary: MERGE-SORT is thus an optimal comparison-based algorithm.

We need not look any further to get a better algorithm (in terms of "big oh").

Recall Ω - Big Omega!

If a running time is $\Omega(f(n))$, then for large enough n, the running time is at least $k \cdot f(n)$ for some constant k. Here's how to think of a running time that is $\Omega(f(n))$:



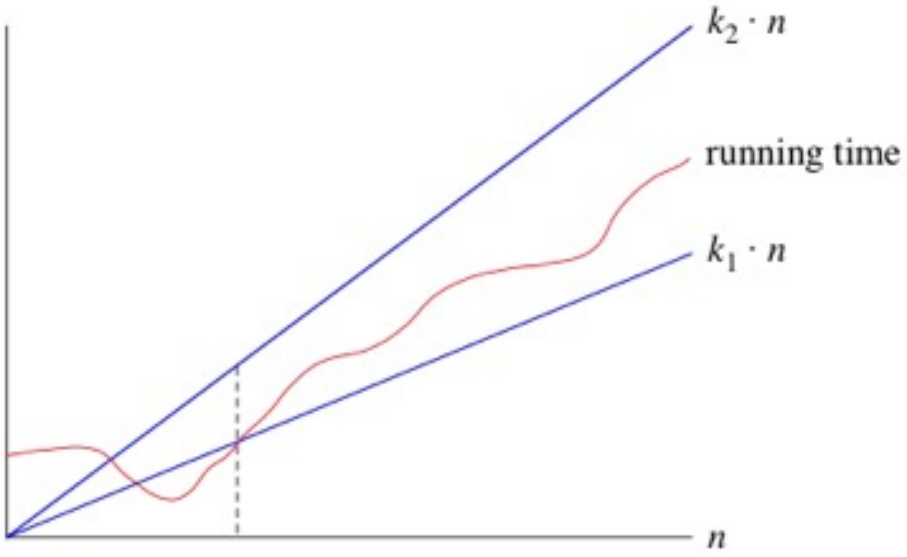
graph from:
https://www.khanacademy.org/computing/computer-science/algorithms/asymptotic-notation/a/big-big-omega-notation

f(n) = o(g(n)) means,

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

Theta() symbol: θ - example θ(n)

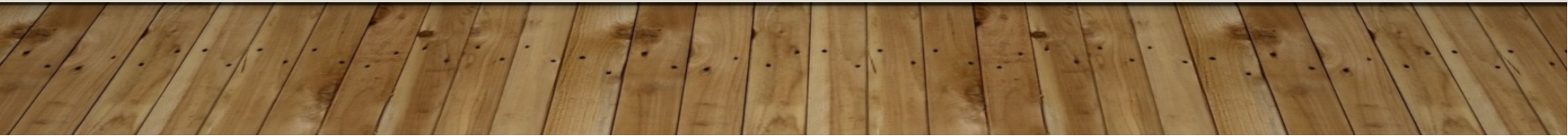
When we say that a particular running time is $\Theta(n)$, we're saying that once n gets large enough, the running time is at least $k_1 \cdot n$ and at most $k_2 \cdot n$ for some constants k_1 and k_2. Here's how to think of $\Theta(n)$:



Back to Recurrences

What is the complexity of binary search?

Brief recall in-class



2. Time complexity

Recurrence: Let n be the size of the array in which we carry out binary search.

$T(n) = T(n/2) + O(1)$

$T(n) = T(n/2^1) + 1*O(1) = T(n/4) + 2*O(1)$

$= T(n/2^2) + 2*O(1)$

$= T(n/8) + O(1) + O(1) + O(1)$

$= T(n/2^3) + 3*O(1)$... by induction

$= T(n/2^i) + i*O(1)$ for $i = 1, \ldots, \log_2 n$

$= T(n/2^{\log_2 n}) + \log_2 n * O(1) = O(\log_2 n)$

as $T(1) = O(1)$

Another example

(arises in the analysis of a multiplication algorithm - not important right now here) $T(n) = 3\,T(n/2) + n$

$= 3[3T(n/2^2) + n/2^1] + n$

$= 3^2T(n/2^2) + 3n/2^1 + n$ =... <by induction>

$= {}_3{}^{\log_2 n}O(1) + .. + (3/2)^i n + ... + (3/2)^1 n + n$

Note: ${}_3{}^{\log_2 n} = n^{\log_2 3}$

This is a geometric series -> $0(n^{\log_2 3}) = 0(n^{1.59})$

Rules for working with logarithms

Rule name	Rule
product rule	$\log_b(x \cdot y) = \log_b(x) + \log_b(y)$
quotient rule	$\log_b(x / y) = \log_b(x) - \log_b(y)$
power rule	$\log_b(x^y) = y \cdot \log_b(x)$
base switch rule	$\log_b(c) = 1 / \log_c(b)$
base change rule	$\log_b(x) = \log_c(x) / \log_c(b)$

Why is $3^{\log_2 n} = n^{\log_2 3}$?

- $3^{\log_2 n} = n^{\log_2 3}$
- Take the log base n on both sides:
- Obtain: $\log_2 n \log_n 3 = \log_2 3$
- Divide both sides by $\log_2 n$
- Obtain $\log_n 3 = \log_2 3 / \log_2 n$

- **base change rule** $\log_b(x) = \log_c(x) / \log_c(b)$
  - With b= n , x = 3 and c=2 the above is thus correct

Geometric Series

- For $x \neq 1$, the summation $\sum_{k=0}^{n} x^k = 1 + x + x^2 + \ldots + x^n$ is a geometric series and has value:

$$= \frac{x^{n+1} - 1}{x - 1}$$

- When $|x| < 1$, we get an infinite decreasing geometric series $\sum_{k=0}^{\infty} x^k$ converging to $1/(1-x)$.

Geometric Series

(see also textbook p. 1147)

What happens for $x = 1$?

$x = \frac{1}{2}$?

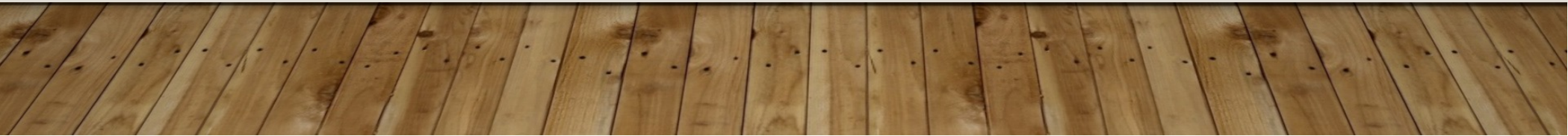
$x = 2$?

See class.

Solving Recurrences by Substitution

The previous method, also known as Iteration/Iterative of Unfolding Method, often works, but may be error-prone if you are not careful. Use the hint from class.

If you have a good guess of what the solution is then, the following method is faster to use.



Substitution/Guessing Method

1. Make a guess
2. Prove using induction

Example: $T(n) = T(n-1) + n$

1. Guess $T(n) \leq cn^2$
2. Prove using induction

   Assume that $T(k) \leq ck^2$ for $k < n$, for some constant $c>1$

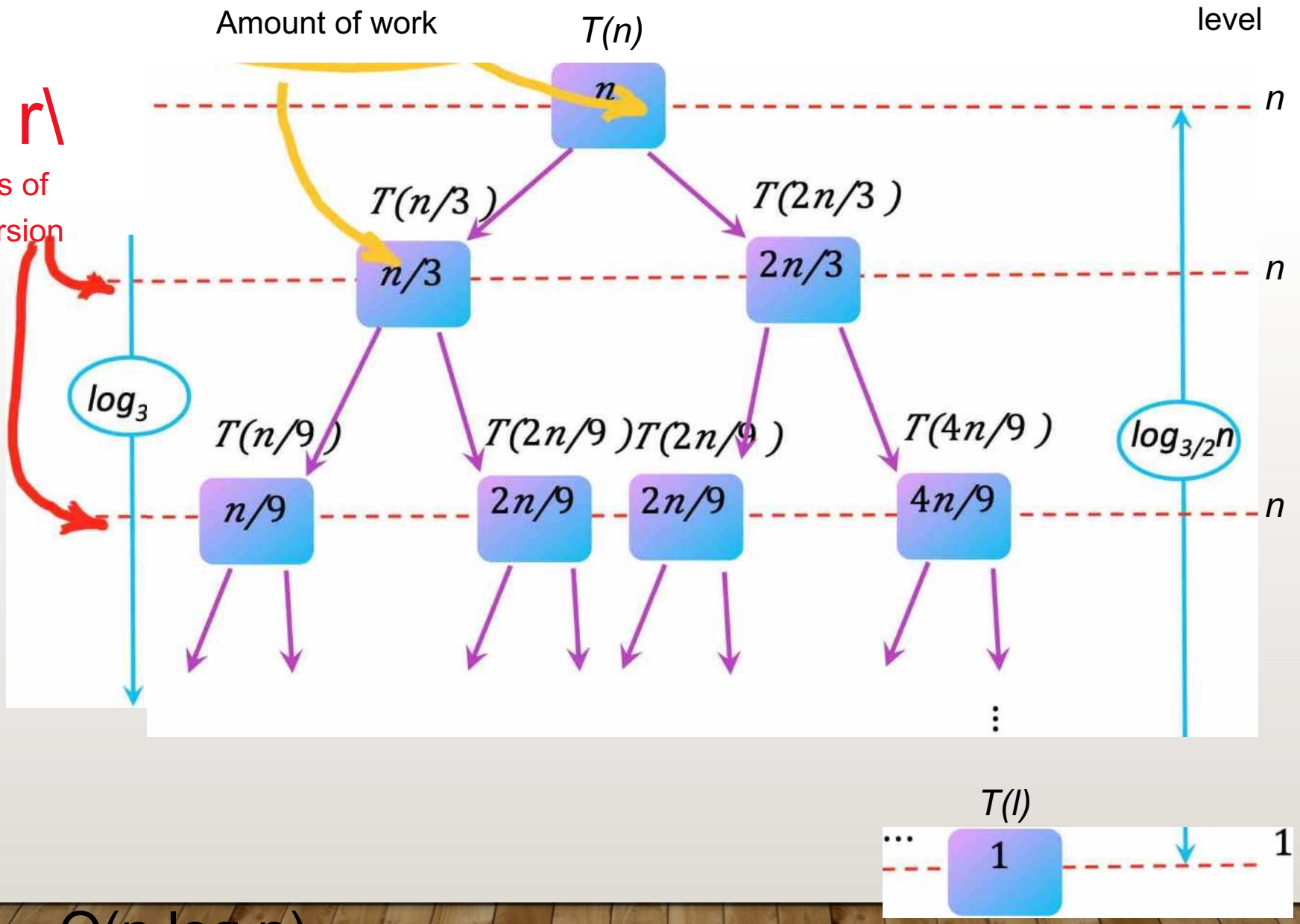
   $T(n) = T(n-1) + n \leq c(n-1)^2 + n \leq cn^2 - 2cn + c + n$

   $= cn^2 - c(2n-1) + n \leq cn^2$ for $c>1$ as $-c(2n-1) < n$

   (actually $c>1/2$ would do)

Recall Tutorial

Recursion Tree Method



O(n log n)

Master Theorem: slightly simplified

Simplified setting for many D&C algorithms

Let a=# of subproblems, *(n/b)* = size of the subproblems (assume ceiling function for *n/b)*

and n the total size if

if T(n) = $aT(n/b) + O(n^d)$, then

$$T(n) = \begin{cases} \text{A.} & O(n^d) \text{ if } d > \log_b a \\ \text{B.} & O(n^d \log n) \text{ if } d = \log_b a \\ \text{C.} & O(n^{\log_b a}) \text{ if } d < \log_b a \end{cases}$$

Proof Sketch

assume w.l.o.g. that $n = b^j$, for some *j*.

The work carried out on level *k* of the recurrence tree is $a^k * 0(n/b^k)^d$, where

number of subproblems — their sizes — work due to $0(n^d)$

$a^k * 0(n/b^k)^d = 0(n^d)(a/b^d)^k$ where $k=0,\ldots,\log_b n$ →

geometric series with ratio a/b^d

Proof sketch cont'd

Three cases arise:

(1) $a/b^d < 1$ -> series is determined by the first term

i.e., $O(n^d)$ *[work is decreasing as we go down the levels]*

(2) $a/b^d = 1$ -> all $O(\log n)$ terms are equal to $O(n^d)$

(3) $a/b^d > 1$ series (work per level) is increasing and its sum is determined by the last term $O(n^{\log_b a})$

3. cont'd

$$O(n^d\ (a/b^d)^{\log_b n}) = O(n^d(a^{\log_b n} / (b^{\log_b n})^d))$$

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

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

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

Example of uses of Master Theoreml

Binary Search

$$T(n) = T(n/2) + 0(1)$$

[again take n/2 to mean ceiling(n/2)]

What are a, *b,* and *d* for this example?

Binary search use of Master Theorem 1

a =1 $b = 2$, $d = 0$

now calculate $log_b a = \log_2 1 = 0$

We are therefore in Case B and thus get:

T(n) = $0(n^d log n)$ = 0(log n) as d=0

Binary search is therefore 0(log *n).*

Revisit MergeSort

- T(n) = $aT(n/b) + O(n^d)$, $a = 2$, b=2 , d = 1

$log_b a = log_2 2 = 1$ which equals d

A. $O(n^d)$ if $d > log_b a$

B. $O(n^d \log n)$ if $d = log_b a$ <- O(n log n) since d=1

C. $O(n^{\log_b a})$ if $d < log_b a$

Master Theorem I gives us rapidly O(n log n)

Solving Recurrences

Master theorem 2 - slightly more general

Let $a \geq 1$ and $b > 1$, left $f(n)$ be a function, and let $T(n)$ be defined on the nonnegative integers by the recurrence $T(n) = a*T(n/b) + f(n)$

1. If $f(n) = O(n^{\log_b a - \varepsilon})$ for constant $\varepsilon > 0$,
   then $T(n) = \Theta(n^{\log_b a})$

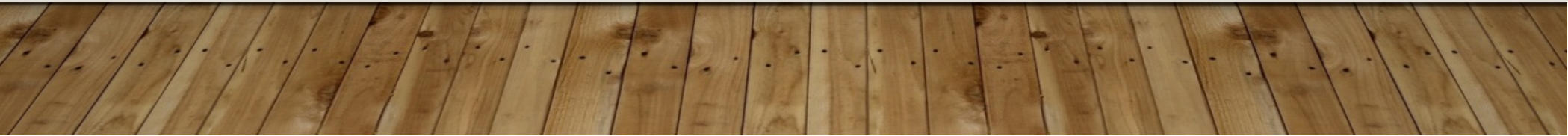
2. If $f(n) = \Theta(n^{\log_b a})$
   then $T(n) = \Theta(n^{\log_b a} \log n)$

3. If $f(n) = \Omega(n^{\log_b a + \varepsilon})$ for constant $\varepsilon > 0$, then if $a*f(n/b) < c*f(n)$ for some constant $c < 1$ and all sufficiently large n
   then $T(n) = \Theta(f(n))$

Note

this setting is more precise and also incorporates Θ bounds.

Working with the Master Theorem takes practice!



Example: $T(n) = 9T(n/3) + n$

What do you expect?

Apply Master Theorem with

$a=9$, $b=3$ and $f(n) = n$

$n^{\log_b a} = n^{\log_3 9} = \Theta(n^2)$

Since $f(n) = O(n^{\log_3 9-\varepsilon})$ with $\varepsilon = 1$, we are in Case 1. Therefore: $T(n) = \Theta(n^2)$

Example: $T(n) = T(2n/3) + 1$

What do you expect?

Apply Master Theorem with a=1, b=3/2

and f(n) = 1

$n^{\log_b a} = n^{\log_{3/2} 1} = n^0 = 1$

Since $f(n) = O(n^{\log_b a}) = \Theta(1)$, we are in

Case 2. therefore: $T(n) = \Theta(\log n)$

Example: $T(n) = 3\,T(n/4) + n\log n$

- What do you expect?
- Apply Master Theorem

  with a=3, b=4 and f(n) = n log n

  $n^{\log_b a} = n^{\log_4 3} = \Theta(n^{0.793})$

  Since $f(n) = \Omega(n^{\log_4 3+\varepsilon})$ with ε=0.2, we are in not quite yet in Case 3.

We need to show that a*f(n/b) < c f(n) for some constant

c <1 and all sufficiently large n.

Then, we will get: $T(n) = \Theta(f(n) = \Theta(n\log n)$

cont'd: $T(n) = 3\,T(n/4) + n\log n$

We need to show that af(n/b) < c f(n), for some constant c <1 and all sufficiently large n.

a*f(n/b) = 3*f(n/4) =

3*(n/4)log(n/4) ≤ (3/4)nlogn = (3/4) f(n), c=3/4

Therefore, we now get: T(n) = Θ(f(n) = Θ(nlog n)

Limitations of Master Theorem

Example: $T(n) = 2\,T(n/2) + n \log n$

- What do you expect?
- We cannot apply the Master Theorem.
- Attempt:

  with a=2, b=2 and f(n) = n logn

  $n^{\log_b a} = n^{\log_2 2} = \Theta(n)$

Case 3 does NOT apply.

"If $f(n) = \Omega(n^{\log_b a+\varepsilon})$ for constant $\varepsilon > 0$, "

$f(n) = n \log n$ and $n^{\log_b a+\varepsilon} = n^{1+\varepsilon}$.

Limitations of Master Theorem cont'd

we have $f(n) = n \log n$ and $n^{\log_b a+\varepsilon} = n^{1+\varepsilon}$

Is $n \log n = \Omega(n^{1+\varepsilon})$ or not?

Note: $n \log n / n = \log n$

$\Omega(n^{1+\varepsilon})/n = \Omega(n*n^{\varepsilon}/n) = \Omega(n^{\varepsilon})$

but $\log n$ is not $\Omega(n^{\varepsilon})$.

[$\Omega(n^{\varepsilon})$is exponential and $\log n$ only logarithmic.] So, the Master Theorem cannot be applied here.