Evaluating Time Complexity of Your Code

Dr. R. A. Angryk

OUTLINE

- What is your code?
- How can we evaluate it?
- Why are we interested in the analysis of time complexity?
- Running Time Analysis Examples
- Best-case, Worst-case & Average-case Complexity
- Multiple-Variable Input Size
- Asymptotic Behavior of Functions
- More examples
- Questions

What is "the code"?

1. *Computational procedure (a sequence of steps)*
2. *Takes some value(s) (called input)*
3. *Produces some value(s) (called output)*

How can we analyze time complexity of the code?

To analyze the code we build a computational model, which makes certain simplifying assumptions, e.g.:

- Any number (integer, real), character, ... uses one "unit" of space
- All primitive operations take constant time
- … and that we have a Random Access Machine

Computational models

Random Access Machine

- One processor, sequential execution
- Basic data types
- Basic operations
  - Arithmetic operations: +, –, *, /, !, ,
  - Logical operations: AND, OR, NOT
  - Array indexing: A[x], where x is a variable
  - if-then-else
  - while-loops, for-loops
  - procedure calls
  - Recursive calls use infinite size of stack
  - Etc.

Parallel multi-processor access model (*PRAM) – maybe in Grad School…* ☺

What are we interested in?

- Consumption of the computational resources
  - CPU time (running time)
  - memory usage (space)
  - messages sent along the network (bandwidth)

  Usually we focus on the running time!
- Resource consumption differs depending on the size of the input → Specify resource consumption (e.g. running time) as a function of the inputs size
- Resource consumption may differ greatly for inputs of the same size, depending on their structure (highly unsorted, almost sorted, …)

Is it worth it?

- Computers are getting faster…
- Memory gets cheaper…
- Is it worth it?
  - Computers may be very fast, but not infinitely!
  - Memory may be very cheap, but not free!
  - There are always competitors on the market!
  - Advanced technologies involve even more complex algorithms
  - This is what truly separates skilled programmers from novices

I want to show you that…

- Programs for solving the same problem can differ dramatically in their efficiency.
- This difference might be much more significant than the differences generated by new hardware or software.

Comparison of two sorting algorithms (1)

Sort an array of ONE MILLION NUMBERS using:

1. **Insertion sort**
2. **Merge sort**

Ad. 1. Best programmer & machine language & one billion instructions per second computer → 1GHz=10^9Hz

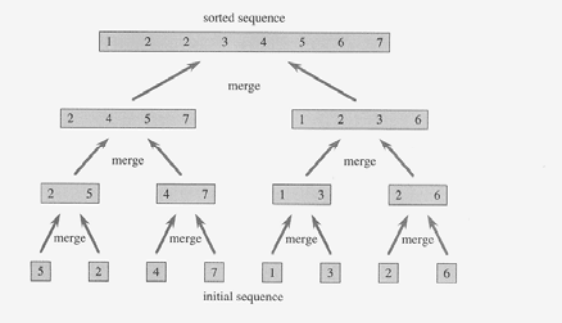
Ad. 2. Bad programmer & high-level language & ten million instructions per second computer → 10MHz=10^7Hz

THE FIRST MACHINE IS 100 TIMES FASTER ($10^9/10^7$)

INSERTION SORT

similar to sorting cards

Example of MERGE-SORT



Comparison of two sorting algorithms (2)

	INSERTION SORT	*MERGE SORT*
Running time formula	$c \cdot n^2$	$c \cdot n \cdot lg_2 n$
TASK: Sort an array of 1 mln numbers (n)	10^6	10^6
Constant factor (c) (e.g. # of CPU ticks per 1 operation of comparison)	great programmer machine language 2	bad programmer high-lev. language 50
Machine (# ticks per second)	10^9/1s =1GHz	10^7/1s =10MHz
Total # of ticks to sort it	$2 \cdot (10^6)^2$	$50 \cdot 10^6 \cdot lg_2 10^6$
Divided by the number of ticks in one second	$2 \cdot (10^6)^2 / 10^9$	$50 \cdot 10^6 \cdot lg_2 10^6 / 10^7$
Gives the RUNNING TIME:	= 2 000 s	≈ 100 s
To sort 10 milion of numbers	2.3 days	20 minutes

What do we really care about?

- All algorithms are fast for small inputs
- LARGE inputs is a different story…
- Asymptotic analysis:
  - Input size is very large (going to infinity!)
  - Ignore lower-order terms and constant coefficients
  - Only the highest order item taken under consideration – e.g in order of growth:

f(1) ↗ f(lg *n*) ↗ f(*n*) ↗ f(n^2) ↗ f(n^3) ↗ … ↗ f(2^n) ↗ …

Analyzing Code (1)

- For a while from now, we will focus on analyzing time complexity of a code; thus, when using term "complexity" I mean time complexity unless I specify otherwise.
- Running time analysis estimates the time required of a code as a function of the input size.
- Usages:
  - To estimate growth rate as input grows.
  - To be able to choose between alternative programs (or procedures).

Analyzing Code (2)

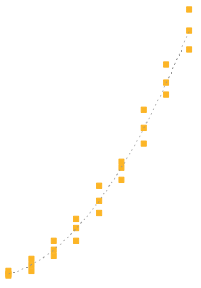
- We measure the complexity of a code by identifying a basic operation and then counting how many times the code performs that basic operation for an input of a given size (n)
- The number of the basic operations may differ for the same size of input
- Also, we need to be careful what we take for the measure of the input size (e.g. element of an array vs. HDD block)
  - We will focus on "simple data types" in this class, forget that I mentioned HDD blocks ☺

How can we analyze and compare running times of different programs?

- **Experimental Measurements**
- **Theoretical Analysis**
- Given 2 different codes for a single problem, we need to define a model for comparing two runtimes
- In general, we will compare for "large" n
  - scalability is of primary importance -
    - How does the code perform on progressively larger data?
  - Theoretical Analysis examines the limit as $n \to \infty$

Experimental Measurements

- Write a program implementing the algorithm
- Run the program with inputs of varying size and composition
- Use a method like System.currentTimeMillis() to get an accurate measure of the actual running time
- Plot the results



Limitations of Experiments

- It is necessary to implement every algorithm we want to compare, which may be difficult.
- Results may not be indicative of the running time on other inputs not included in the experiment.
- In order to compare two algorithms, exactly the same hardware and software environments must be used

Theoretical Analysis

- Lets us use a high-level description of the code instead of an actual implementation (now we compare our code against other approaches, without implementing all of them)
- Characterizes running time as a function of the input size, *n*.
- Takes into account all possible inputs
- Allows us to evaluate the speed of an algorithm independent of the hardware/software environment

Analysis and measurements

- Experimental performance measurements (e.g. execution time): machine dependent.
- Theoretical performance analysis: machine independent.
- How do we analyze a program independent of a machine?
  - Counting the number of steps.

General Rules for Counting Steps (1)

- Rule 1: FOR loops

  The running time of a for loop is at most the running time of the statements inside the for loop (including tests) times the number of iterations.
- Rule 2: Nested loops

  Analyze these inside out. The total running time of a statement inside a group of nested loops is is at most the running time of the statement multiplied by the product of the sizes of all the loops. *E.g:*

```
for (i=0; i < n; i++)
    for (j=0; j < n; j++)
        k++;
```

Running time: $T(N) = O(N^2)$

General Rules for Counting Steps (2)

- Rule 3: Consecutive statements

  These just add (which means that the maximum is the one that counts)

```
E.g: for (i=0; i < n; i++)
         a[i] = 0;                    } Takes O(N)
     for (i=0; i < n; i++)
         for (j=0; j < n; j++)        } Takes O(N^2)
             a[i] += a[j] + i + j;
```

Running Time: $T(N) = O(N) + O(N^2) = O(N^2)$

General Rules for Counting Steps (3)

- Rule 4: IF/ELSE

  For the fragment:

```
if (test condition)
        S1
else
        S2
```

The running time of an if/else statement is never more than the running time of the test plus the larger of the running times of S1 and S2.

Three kinds of running time

- The worst-case running time: the longest running time for *any* input of size *n*.
- The average running time: the average running time over *all* possible inputs of size *n*.
- The best-case running time: the shortest running time for *any* input of size *n*.

Best vs. Worst Case

- Best Case: When the situation is ideal.

E.g. : Already sorted input for a sorting program.

Not an interesting case to study (naive approach).

Lower bound on running time.

- Worst case: When situation is worst.

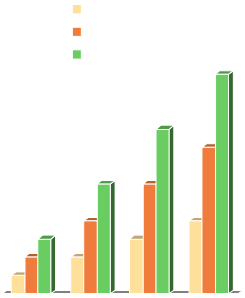
E.g.: Reverse sorted input for a sorting program.

Interesting to study since then we can say that no matter what the input is, our program will never take any longer.

Upper bound on running time for any input.

Average-case Running Time

- Average case: Any random input. Often roughly as bad as worst case. Problem with this case is that it may not be apparent what constitutes an "average" input for a particular problem.

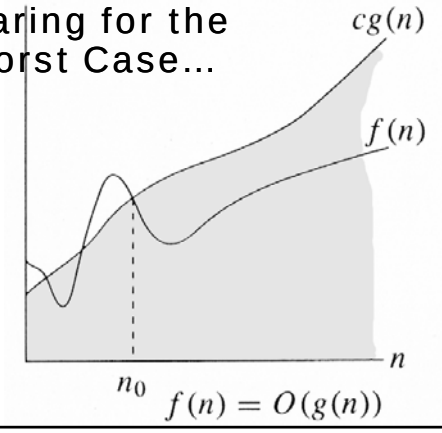


Why to analyze the Worst Case

We usually concentrate on finding the worst-cast running time.

- The worst-case running time of a program is an upper bound on the running time for any input. Knowing it gives us a guarantee that the code will never take any longer.
- For some programs/tasks, the worst case occurs fairly often (e.g. we would not sort the sorted data!).
- The average case is often roughly as bad as the worst case, e.g., insertion sort.

Preparing for the Worst Case…



Multiple-Variable Input Size

- Sometimes, the input size, referred so far as n, becomes a function of two or more variables. E.g. when sorting lists of size n with repeated elements, the input size can be defined as a function of n and a number of m distinct elements in the list, where 1<=m<=n.
- In such cases, best-case, worst-case, and average-case complexities are functions of multiple variables.
  - Often, to simplify our investigation we may assume n>>m, or n==m.

Growth of Functions

- A way to describe behavior of functions in the limit, as the size of the input increases without bound (goes to infinity).
- We are studying asymptotic efficiency, that is, we look at the running time of program when the input sizes are large enough to make only the order of growth of the running time relevant.
- Describe growth of functions.
- Focus on what's important by abstracting away all low-order terms and constant factors.
- Changing the hardware/ software environment
  - Affects $T(n)$ by a constant factor, but
  - Does not alter the growth rate of $T(n)$

Seven Growth Functions

- **Seven functions that often appear in time complexity analysis:**
  - **Constant ≈ *1***
  - **Logarithmic ≈ log *n***
  - **Linear ≈ *n***
  - **Log Linear ≈ *n* log *n***
  - **Quadratic ≈ n^2**
  - **Cubic ≈ n^3**
  - **Exponential ≈ 2^n**
  - **Factorial ≈ *n!***

The Constant Function

- **f(n) = c for some fixed constant c.**
- **The growth rate is independent of the input size.**
- **Most fundamental constant function is g(n) = 1**
- **f(n) = c can be written as**

  f(n) = cg(n)
- **Characterizes the number of steps needed to do a basic operation on a computer.**

The Linear and Quadratic Functions

- **Linear Function**
  - **f(n) = n**
  - **For example comparing a number x to each element of an array of size n will require n comparisons.**
- **Quadratic Function**
  - **f(n) = n^2**
  - **May appear when there are nested loops in programs**

The Cubic functions and other polynomials

- **Cubic functions**
  - **f(n) = n^3**
- **Polynomials**
  - **f(n) = $a_0 + a_1n + a_2n^2 + \ldots\ldots + a_dn^d$**
  - **d is the degree of the polynomial**
  - **$a_0, a_1 \ldots\ldots a_d$ are called coefficients.**

The Exponential Function

- **f(n) = b^n**
- **b is the base**
- **n is the exponent**
- **For example if we have a loop that starts by performing one operation and then doubles the number of operations performed in the nth iteration is 2^n.**
- **Exponent rules:**
  - **$(b^a)^c = b^{ac}$**
  - **$b^a b^c = b^{a+c}$**
  - **$b^a / b^c = b^{a-c}$**

The Logarithm Function

- **f(n) = $\log_b n$**
- **b is the base**
- **x = $\log_b n$ if and only if $b^x = n$**
- **Logarithm Rules**
  - **$\log_b ac = \log_b a + \log_b c$**
  - **$\text{Log}_b a/c = \log_b a - \log_b c$**
  - **$\log_b a^c = c\log_b a$**
  - **$\log_b a = (\log_d a)/ \log_d b$**
  - **$b^{\log_d a} = a^{\log_d b}$**

The N-Log-N Function

- f(n) = nlogn
- Function grows little faster than linear function and a lot slower than the quadratic function.

Growth rates Compared

	n=1	n=2	n=4	n=8	n=16	n=32
1	1	1	1	1	1	1
logn	0	1	2	3	4	5
n	1	2	4	8	16	32
nlogn	0	2	8	24	64	160
n^2	1	4	16	64	256	1024
n^3	1	8	64	512	4096	32768
2^n	2	4	16	235	65536	4294967296

Asymptotic Notation

- Although we can sometimes determine the exact running time of a code, the extra precision is not usually worth the effort of computing it.
- For large enough inputs, the multiplicative constants and lower order terms of an exact running time are dominated by the effects of the input size itself.

Simplifications

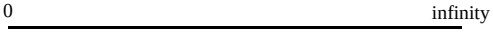
- Goal: to simplify analysis by getting rid of unneeded information
- Ignore constants
  - $4n^2 - 3n \log n + 17.5 n - 43 n^{⅔} + 75$ becomes
  - $n^2 – n \log n + n - n^{⅔} + 1$
- Asymptotic Efficiency - Expressed using only the highest-order term in the expression for the exact running time.
  - We want to say it in a formal way:
    $n^2 – n \log n + n - n^{⅔} + 1$ becomes n^2
- Instead of exact running time, we say: $O(n^2)$.

Why ignore constants?

- RAM model introduces errors in constants
  - Do all instructions take equal time?
  - Specific implementation (hardware, code optimizations) can speed up an algorithm by constant factors
  - We want to understand how effective a code is independent of these factors
- Simplification of analysis
  - Much easier to analyze if we focus only on n^2 rather than worrying about $3.7 n^2$ or $3.9 n^2$

Asymptotic Analysis

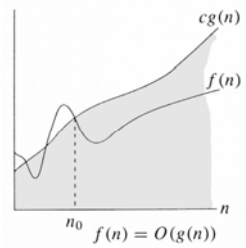
- Running time of our program as a function of input size *n* for large *n*.
- We focus on the infinite set of large n ignoring small values of n. It describes behavior of a function in the upper limit
- Usually, the code that is asymptotically more efficient will be the best choice for all but very small inputs.



O-notation

For function $g(n)$, we define $O(g(n))$, big-O of n, as the set:

$O(g(n))$ = {$f(n)$:
$\exists$ positive constants c and n_0,
such that $\forall n \geq n_0$,
we have $0 \leq f(n) \leq cg(n)$ }



Intuitively: Set of all functions whose *rate of growth* is the same as or lower than that of $g(n)$.

$g(n)$ is an *asymptotic upper bound* for $f(n)$.

Big-O

$$2n^2 = O(n^2)$$
$$1{,}000{,}000\ n^2 + 150{,}000 = O(n^2)$$
$$5n^2 + 7n + 20 = O(n^2)$$
$$2n^3 + 2 \neq O(n^2)$$
$$n^{2.1} \neq O(n^2)$$

Exercises on Big-*O* (1)

- **Prove that:** $20n^2 + 2n + 5 = O(n^2)$
- **Let c = 21 and n_0 = 4**
- **$21n^2 > 20n^2 + 2n + 5$ for all $n > 4$**
  $n^2 > 2n + 5$ for all $n > 4$
  TRUE

Exercises on Big-*O* (2) Big-*O* = Asymptotic Upper Bound

- **Show that $3n^2+2n+5 = O(n^2)$**
- **Let c = 10**

$10\ n^2 = 3n^2 + 2n^2 + 5n^2 \geq 3n^2 + 2n + 5$ for $n \geq 1$

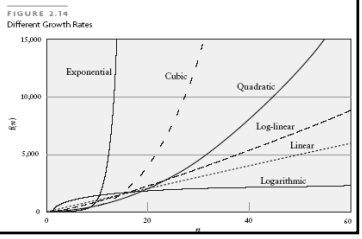
$c = 10$, $n_0 = 1$

Usage

- **We usually use the simplest formula when we use the O-notation.**
  - **We write**
    - $3n^2+2n+5 = O(n^2)$
  - **The followings are theoretically correct but we don't usually use them (what's the point?)**
    - $3n^2+2n+5 = O(3n^2+2n+5)$
    - $3n^2+2n+5 = O(n^2+n)$
    - $3n^2+2n+5 = O(3n^2)$

O-notation – what do you think?

- **f1(n) = 10 n + 25 n^2**
- **f2(n) = 20 n log n + 5 n**
- **f3(n) = 12000 n log n + 0.05 n^2**



Efficiency of your Code

- **Consider:**

```
for (int i = 0; i < n - 1; i++) {
    for (int j = i + 1; j < n; j++) {
        3 simple statements
    }
}
```

- **First time through outer loop, inner loop is executed n-1 times; next time n-2, and the last time once.**
- **So we have**
  - **T(*n*) = 3(*n* – 1) + 3(*n* – 2) + 3(*n* – 3)+ … + 3 or**
  - **T(*n*) = 3(*n* – 1 + *n* – 2 + *n* – 3 + … + 1)**
  - **(*n* – 1 + *n* – 2 + *n* – 3 + … + 1) is (1 + … + *n* – 3 + *n* – 2 + *n* – 1)**

Efficiency of your Code

- **We can reduce the expression (1 + … + *n* – 3 + *n* – 2 + *n* – 1) to:**
  - $\frac{n\,(n-1)}{2}$
- **So, T(*n*) = 1.5n^2 – 1.5*n***
- **This polynomial is zero when *n* is 1. For values greater than 1, 1.5n^2 is always greater than 1.5n^2 – 1.5*n***
- **Therefore, we can use 1 for n_0 and 1.5 for *c* to conclude that T(*n*) is O(n^2)**

Algorithms with Recursive Calls

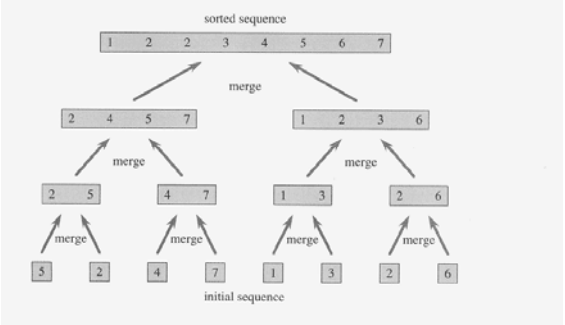
- **Running times of algorithms with** Recursive calls **can be described using recurrences**
- **A** recurrence **is an equation or inequality that describes a function in terms of its value on smaller inputs.**
- **For divide-and-conquer algorithms:**

$$T(n) = \begin{cases} \text{solving\_trivial\_problem} & \text{if } n = 1 \\ \text{num\_pieces } T(n/\text{subproblem\_size\_factor}) + \text{dividing} + \text{combining} & \text{if } n > 1 \end{cases}$$

- **Example: Merge Sort**

$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1 \\ 2T(n/2) + \Theta(n) & \text{if } n > 1 \end{cases}$$

Example of MERGE-SORT



Repeated Substitution Method

- **Expanding the recurrence by substitution and noticing a pattern**
- **<u>The procedure is straightforward:</u>**
  - **Substitute**
  - **Expand**
  - **Substitute**
  - **Expand**
  - **…**
  - **Observe a pattern and write how your expression looks after the *i*-th substitution**
  - **Find out what the value of *i* (e.g., lg*n*) should be to get the base case of the recurrence (say *T*(1))**
  - **Insert the value of *T*(1) and the expression of *i* into your expression**

Repeated Substitution

- **Let's find the running time of the merge sort (let's assume that $n=2^b$, for some *b*).**

$$T(n) = \begin{cases} 1 & \text{if } n = 1 \\ 2T(n/2) + n & \text{if } n > 1 \end{cases}$$

$$\begin{aligned} T(n) &= 2T(n/2) + n && \text{substitute} \\ &= 2\left(2T(n/4) + n/2\right) + n && \text{expand} \\ &= 2^2 T(n/4) + 2n && \text{substitute} \\ &= 2^2(2T(n/8) + n/4) + 2n && \text{expand} \\ &= 2^3 T(n/8) + 3n && \text{observe the pattern} \\ T(n) &= 2^i T(n/2^i) + in \\ &= 2^{\lg n} T(n/n) + n\lg n = n + n\lg n \end{aligned}$$

$2^i = num\_pieces = n \rightarrow 2^i = n \rightarrow i = \log_2 n$

$2^{\log_2 n} = n$

$T(\frac{n}{n}) = const$

Recursion-tree method

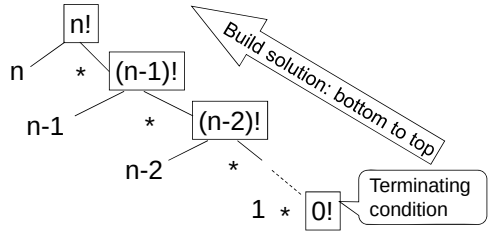
The recursion-tree method converts the recurrence into a tree:

- **Each node represents the cost incurred at various levels of recursion**
- **Sum up the costs of all levels**

- A recursion tree models the costs (time) of a recursive execution of an algorithm.
- The recursion tree method is good for generating guesses for the substitution method.
- The recursion-tree method can be unreliable.
- The recursion-tree method promotes intuition, however.

What is a Recursion Tree

- **A convenient way to visualize what happens during recursion**



Example of recursion tree

Solve $T(n) = T(n/4) + T(n/2) + n^2$:

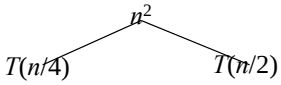
Example of recursion tree

Solve $T(n) = T(n/4) + T(n/2) + n^2$:

$T(n)$

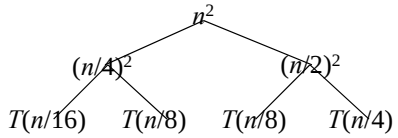
Example of recursion tree

Solve $T(n) = T(n/4) + T(n/2) + n^2$:



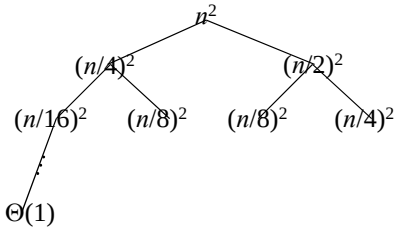
Example of recursion tree

Solve $T(n) = T(n/4) + T(n/2) + n^2$:



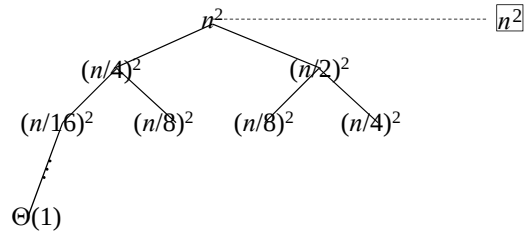
Example of recursion tree

Solve $T(n) = T(n/4) + T(n/2) + n^2$:



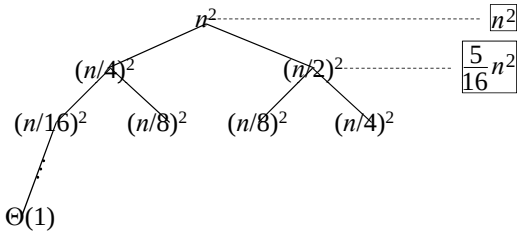
Example of recursion tree

Solve $T(n) = T(n/4) + T(n/2) + n^2$:



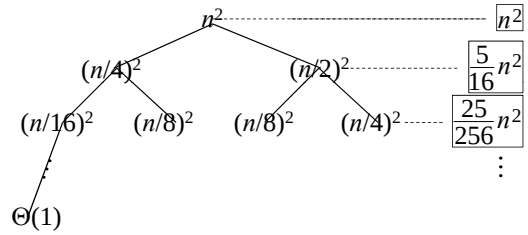
Example of recursion tree

Solve $T(n) = T(n/4) + T(n/2) + n^2$:



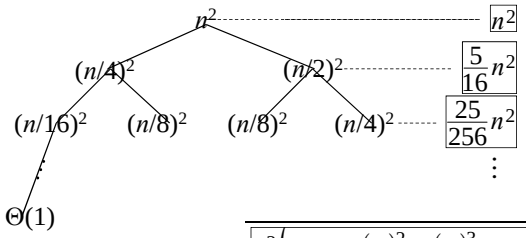
Example of recursion tree

Solve $T(n) = T(n/4) + T(n/2) + n^2$:



Example of recursion tree

Solve $T(n) = T(n/4) + T(n/2) + n^2$:



Total $= \boxed{n^2\left(1 + \frac{5}{16} + \left(\frac{5}{16}\right)^2 + \left(\frac{5}{16}\right)^3 + \cdots\right)}$

$= \Theta(n^2)$ *geometric series*

Appendix: geometric series

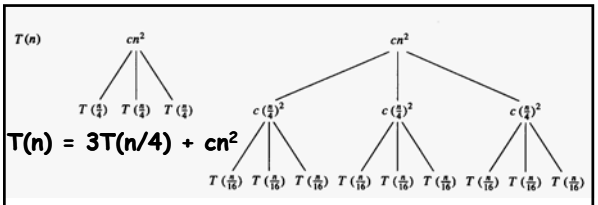
$$\boxed{1 + x + x^2 + \cdots + x^n = \frac{1 - x^{n+1}}{1 - x}} \quad \text{for } x \neq 1$$

$$\boxed{1 + x + x^2 + \cdots = \frac{1}{1 - x}} \quad \text{for } |x| < 1$$

Appendix 2: Useful math to remember...

Logarithms

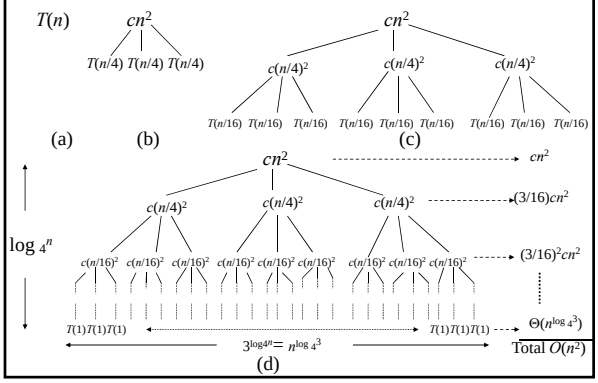
- $\log_c(ab) = \log_c a + \log_c b$
- $\log_b a^n = n \log_b a$
- $\log_b a^n = n \log_b a$
- $\log_b(1/a) = -\log_b a$
- $\log_b a = 1 / \log_a b$
- $b = a^{\log_d b}$
- $b^{\log_d a} = a^{\log_d b}$



T(n) = 3T(n/4) + cn²

- Subproblem size at level i is: $n/4^i$ for i=0,1,…, $\log_4 n$
- The total height of the tree is $\log_4 n+1$
- Subproblem size hits 1 when $1 = n/4^i \Rightarrow i = \log_4 n$
- Cost of a single node at level i = $c(n/4^i)^2$
- Number of nodes at level i = 3^i
- Last level has $3^{\log_4 n} = n^{\log_4 3}$ nodes, each with T(1)

Recursion Tree for $T(n)=3T(n/4)+cn^2$



Total Running Time for $T(n)=3T(n/4)+cn^2$

$T(n)=cn^2+(3/16)\,cn^2+(3/16)^2\,cn^2+$

$\cdots +(3/16)^{\log_4 n-1}\,cn^2+\Theta(n^{\log_4 3})$

$=(1+3/16+(3/16)^2+\cdots+(3/16)^{\log_4 n-2}+(3/16)^{\log_4 n-1})\,cn^2+\Theta(n^{\log_4 3})$

$<(1+3/16+(3/16)^2+\cdots+\cdots+(3/16)^{\infty})\,cn^2+\Theta(n^{\log_4 3})$

note the trick:

$$T(n)=\sum_{i=0}^{\log_4 n-1}\left(\frac{3}{16}\right)^i cn^2+\Theta\left(n^{\log_4 3}\right)\le\sum_{i=0}^{\infty}\left(\frac{3}{16}\right)^i cn^2+\Theta\left(n^{\log_4 3}\right)=\frac{1}{1-\frac{3}{16}}cn^2+\Theta\left(n^{\log_4 3}\right)=O(n^2)$$

This is still just a guess, needs to be proved!

T(n) = 3T(n/4) + cn² – cont.

Now it is the time to prove our guess by induction

- Guess: $T(n) = O(n^2)$
  - Induction goal: $T(n) \le dn^2$, for some d and $n \ge n_0$
  - Induction hypothesis: $T(n/4) \le d\,(n/4)^2$
- Proof of induction goal:

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

  $\le 3d\,(n/4)^2 + cn^2$

  $= (3/16)\,d\,n^2 + cn^2 = [(3/16)\,d + c]\,n^2$

  $\le d\,n^2$ iff: $(3/16)\,d + c \le d$

  $d \ge (16/13)c$

- We found a constant, therefore: $T(n) = O(n^2)$

Examples of Recurrences

- **T(n) = T(n-1) + n** $\Theta(n^2)$
  - Recursive algorithm that loops through the input to eliminate one item
- **Insertion sort:** find the place of the first element in the sorted list, and recursively call with one element less:

  $T(n) = T(n-1) + O(n)$ → $O(n^2)$
- **Sequential search**: see if the first element is the one we are looking for, and if not, recursively call with one element less:

  $T(n) = T(n-1) + O(1)$ → $O(n)$
- **Factorial:** multiply n by $(n-1)!$

  $T(n) = T(n-1) + O(1)$ → $O(n)$

Examples of Recurrences

- **T(n) = T(n/2) + c** **Θ(lgn)**
  - Recursive algorithm that halves the input in one step
- **Binary search:** see if the root of the tree is the one we are looking for, and if not, recursively call with either the left or right subtree, which has half the elements

  $T(n) = T(n/2) + O(1)$ → $O(\lg n)$

- **T(n) = T(n/2) + n** **Θ(n)**
  - Recursive algorithm that halves the input but must examine every item in the input

Examples of Recurrences

- **T(n) = 2T(n/2) + 1** **Θ(n)**
  - Recursive algorithm that splits the input into 2 halves and does a constant amount of other work
- **Binary tree traversal**: visit all the nodes of a tree by recursively visiting the nodes of the left and right tree:

  $T(n) = 2T(n/2) + O(1)$ → $O(n)$

The master method

- **Based on the master theorem.**
- **"Cookbook" approach for solving recurrences of the form**

  $T(n) = aT(n/b) + f(n)$
  - $a \geq 1$, $b > 1$ are constants.
  - $f(n)$ is asymptotically positive.
  - n/b may not be an integer, but we ignore floors and ceilings.
- **Requires memorization of three cases.**

The master theorem (simple version)

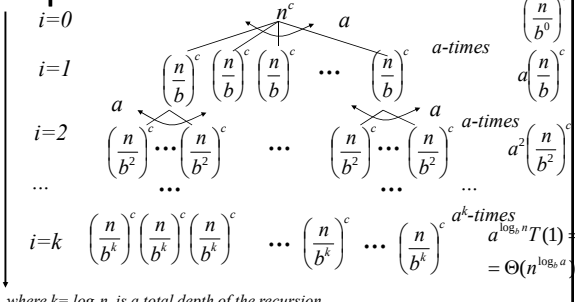
Let $a \geq 1, b > 1, c \geq 0$ be constants and integers, let $f(n)$ be a function of the form n^c, and let T(n) be defined on non-negative integers, denoted by n, by the following recurrence:

$$T(n) = aT(n/b) + n^c$$

Then

1. $T(n) = \Theta(n^c)$ when $a/b^c < 1$ (i.e. $\log_b a < c$)
2. $T(n) = \Theta(n^c \log_b n)$ when $a/b^c = 1$ ($\log_b a = c$)
3. $T(n) = \Theta(n^{\log_b a})$ when $a/b^c > 1$ (i.e. $\log_b a > c$, or simply $a > b^c$)

Recursion-tree for the equation



$$a^{\log_b n} T(1) = \Theta(n^{\log_b a})$$

where $k = \log_b n$, is a total depth of the recursion

$$T(n) = \Theta(n^{\log_b a}) + \sum_{i=0}^{\log_b n - 1} a^i \left(\frac{n}{b^i}\right)^c$$

Recursive substitution for the equation

$T(n) = aT(n/b) + n^c$

$T(n/b) = aT(n/b^2) + (n/b)^c$

$T(n/b^2) = aT(n/b^3) + (n/b^2)^c$

$T(n/b^4) = aT(n/b^5) + (n/b^4)^c$

Now substitute:

$T(n) = aT(n/b) + n^c$

$= a[aT(n/b^2) + (n/b)^c] + n^c$

$= a[a[aT(n/b^3) + (n/b^2)^c] + (n/b)^c] + n^c$

$= a^k T(n/b^k) + n^c[1 + a(1/b)^c + a^2(1/b^2)^c + \ldots a^{k-1}(1/b^{k-1})^c]$

$= a^k T(n/b^k) + \sum_{i=0}^{k-1} a^i \left(\frac{n}{b^i}\right)^c$

if $k = \log_b n$, is a total depth of the recursion than $\frac{n}{b^k} = \frac{n}{b^{\log_b n}} = \frac{n}{n^{\log_b b}} = 1$ *and*

$$T(n) = a^{\log_b n} T(1) + \sum_{i=0}^{\log_b n - 1} a^i \left(\frac{n}{b^i}\right)^c = n^{\log_b a} T(1) + \sum_{i=0}^{\log_b n - 1} a^i \left(\frac{n}{b^i}\right)^c = \Theta(n^{\log_b a}) + \sum_{i=0}^{\log_b n - 1} a^i \left(\frac{n}{b^i}\right)^c$$

Questions?

Reasoning about Programs: Assertions and Loop Invariants

- **Assertions: logical statements about a program that are claimed to be true; generally written as a comment**
- **Preconditions and postconditions are assertions**
- **A loop invariant is an assertion**
  - **Helps prove that a loop meets it specification**
  - **True before loop begins, at the beginning of each repetition of the loop body, and just after loop exit**

Assertions and Loop Invariants Example

```
int i = 0;
// invariant:  for all k, such that 0 <= k < i, x[k] != target
while (i < x.length) {
    if (x[i] == target)
        return i;   // target found at i
    i++;            // Test next element
}

// assert: for all k, such that 0 <= k < i, x[k] != target
//         and i >= x.length
return -1;       // target not found
```

Testing

- **Test drivers and stubs are tools used in testing**
  - **Test drivers exercise a method or class**
  - **Stubs stand in for called methods**

Next Time

- **Homework: Read Appendix C in your textbook!**
- **Batch-Processing**
- **Request-Response Programming**
- **Event-Oriented Programming**