

RECURRENCES

Overview of Today's Lecture

- Recurrence equations: a tool for analyzing algorithms
- Applications of recurrence equations

Two-Step Strategy for Analyzing Algorithms

1. Characterize running-time (space, etc.) of algorithm by a recurrence equation.
2. Solve the recurrence equation, expressing answer in asymptotic notation.

Recurrence Equations

A **recurrence equation** is just a recursive function definition. It defines a function at one input in terms of its value on smaller inputs.

We use recurrence equations to characterize the running time of algorithms. $T(n)$ typically stands for the running-time of a given algorithm on an input of size n for a particular case (worst-case by default, but could be average-case or best case).

Recurrence equations for divide-and-conquer algorithms typically have the form:

General Case ($n \geq 1$)

$$T(n) = [\text{\# of subproblems}] T([\text{size of each subproblem}]) + [\text{cost of divide\&glue}]$$

Base Case ($n < 1$)

$$T(n) = 0$$

Because the base case is always the same, it is usually omitted when defining $T(n)$.

Deriving Recurrence Equations: Examples

1. Insert an element into a sorted list of length n .	$T(n) = \underline{\quad\quad} T(\underline{\quad\quad\quad}) + \underline{\quad\quad\quad}$
2. Insertion-sort a list of length n .	$T(n) = \underline{\quad\quad} T(\underline{\quad\quad\quad}) + \underline{\quad\quad\quad}$
3. Binary search on an array of n elements.	$T(n) = \underline{\quad\quad} T(\underline{\quad\quad\quad}) + \underline{\quad\quad\quad}$
4. Find the maximum value in a balanced binary tree (not a BST) with n numeric leaves.	$T(n) = \underline{\quad\quad} T(\underline{\quad\quad\quad}) + \underline{\quad\quad\quad}$
5. Merge-sort a list of n elements.	$T(n) = \underline{\quad\quad} T(\underline{\quad\quad\quad}) + \underline{\quad\quad\quad}$
6. Create a BST for a list of n elements, assuming that the BST remains balanced after each insertion.	$T(n) = \underline{\quad\quad} T(\underline{\quad\quad\quad}) + \underline{\quad\quad\quad}$
7. Given a balanced binary tree and an element x , multiply each node n on the path to x by the sum of the elements in the subtree rooted at n .	$T(n) = \underline{\quad\quad} T(\underline{\quad\quad\quad}) + \underline{\quad\quad\quad}$
8. Towers of Hanoi	$T(n) = \underline{\quad\quad} T(\underline{\quad\quad\quad}) + \underline{\quad\quad\quad}$

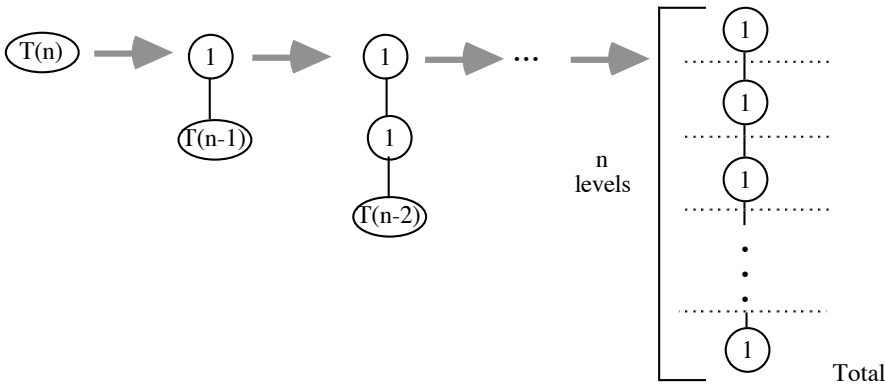
Solving Recurrence Equations: The Recursion-Tree Method

We will use the **recursion-tree method** for solving recurrence equations (CS231 covers other approaches):

1. Construct a recursion tree by unwinding the recurrence equation.
2. Determine the cost of the entire tree by summing the costs of the nodes.

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

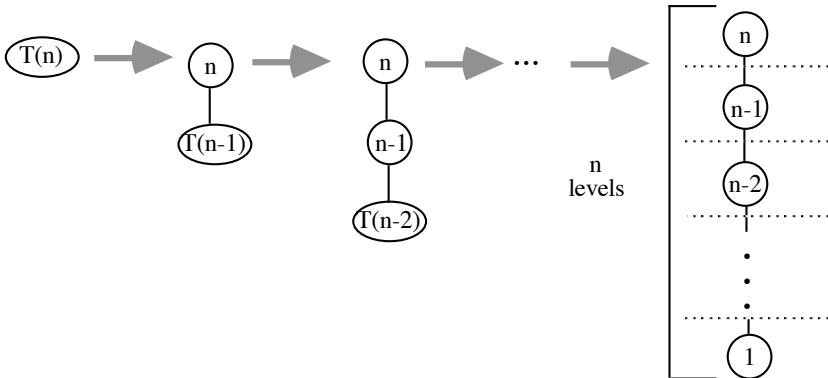
Also derivable: $T(n - 1) = T(n - 2) + 1$
 $T(n - 2) = T(n - 3) + 1$
 etc.



cost of nodes = number of nodes = n

Recurrence 2: $T(n) = T(n - 1) + n$

Also derivable: $T(n - 1) = T(n - 2) + (n - 1)$
 $T(n - 2) = T(n - 3) + (n - 2)$
 etc.

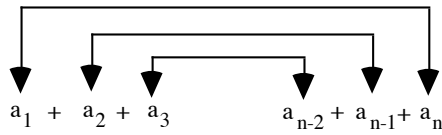


Total cost of nodes = $1 + 2 + 3 + \dots + (n - 2) + (n - 1) + n = \sum_{k=1}^n k = \{\text{Arithmetic series; see below}\}$

Arithmetic Series

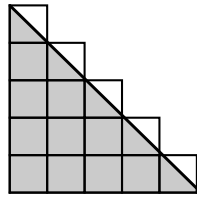
A series is arithmetic if $a_k = c + a_{(k-1)}$.

Trick: Sum corresponding pairs of numbers:



$$\sum_{k=1}^n a_k = \frac{n}{2}(a_1 + a_n)$$

Examples:



- $1 + 2 + 3 + \dots + n =$
- Sum of first n elements of series $7 + 10 + 13 + 16 + \dots =$

Analyzing Insertion Sort

U	N	S	O	R	T	E	D
U	N	S	O	R	T	E	D
N	U	S	O	R	T	E	D
N	S	U	O	R	T	E	D
N	O	S	U	R	T	E	D
N	O	R	S	U	T	E	D
N	O	R	S	T	U	E	D
E	N	O	R	S	T	U	D
D	E	N	O	R	S	T	U

Idea: On the i th step of the algorithm, insert $A[i]$ into the sorted segment $A[1..i-1]$.

Invariant: After the i th step of the algorithm, $A[1..i]$ is sorted.

Case	Recurrence	Solution
<i>Worst</i>		
<i>Best</i>		
<i>Average</i>		

Matching Theoretical Results with Performance Numbers for Insertion Sort

Timings for insertion sort on Java vectors (in milliseconds)

n	Sorted	Reverse Sorted	Random
1000	1	105	43
2000	0	350	173
4000	1	1413	706
8000	4	5707	2850
16000	7	27769	12416
32000	14	145265	59143

Analyzing Selection Sort

U	N	S	O	R	T	E	D
D	N	S	O	R	T	E	U
D	E	S	O	R	T	N	U
D	E	N	O	R	T	S	U
D	E	N	O	R	T	S	U
D	E	N	O	R	T	S	U
D	E	N	O	R	T	S	U
D	E	N	O	R	S	T	U
D	E	N	O	R	S	T	U
D	E	N	O	R	S	T	U

Idea: On the i th step of the algorithm, store into $A[i]$ the smallest element of $A[i..n]$.

Invariant: After step i , the elements $A[1..i]$ are in their final sorted positions.

Case	Recurrence	Solution
<i>Worst</i>		
<i>Best</i>		
<i>Average</i>		

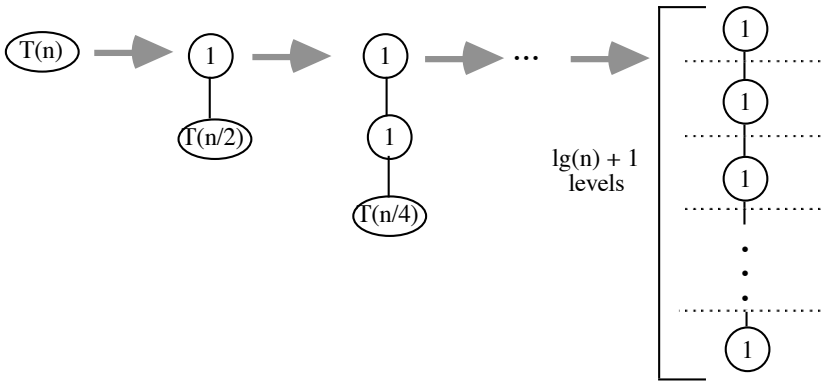
Performance Numbers for Selection Sort

Timings for selection sort on Java vectors (in milliseconds)

n	Sorted	Reverse Sorted	Random
1000	84	81	81
2000	314	320	331
4000	1254	1279	1785
8000	6777	7373	7263
16000	35752	36209	37659
32000	174791	166806	173995

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

Also derivable: $T(n/2) = T(n/4) + 1$
 $T(n/4) = T(n/8) + 1$
 etc.



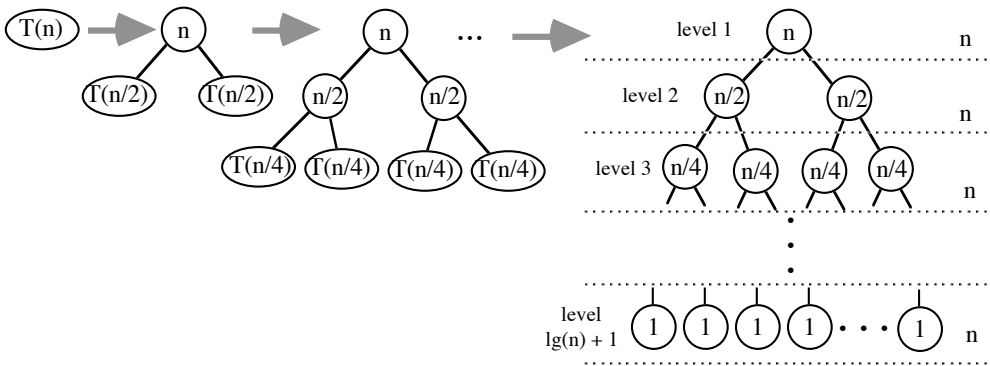
Total cost of nodes = number of nodes = $\lg(n) + 1 = \Theta(\lg(n))$

(In this handout, we use $\lg(n)$ to stand for $\log_2(n)$.)

Information Services
Comment: (We start with one $T(n)$ node and generate a number of subnodes equal to the number of times n can be successively divided by 2. The latter quantity is $\lg(n)$; including the 1 generated by the original $T(n)$ node gives $\lg(n) + 1$ nodes.)

Recurrence 4: $T(n) = 2T(n/2) + n$

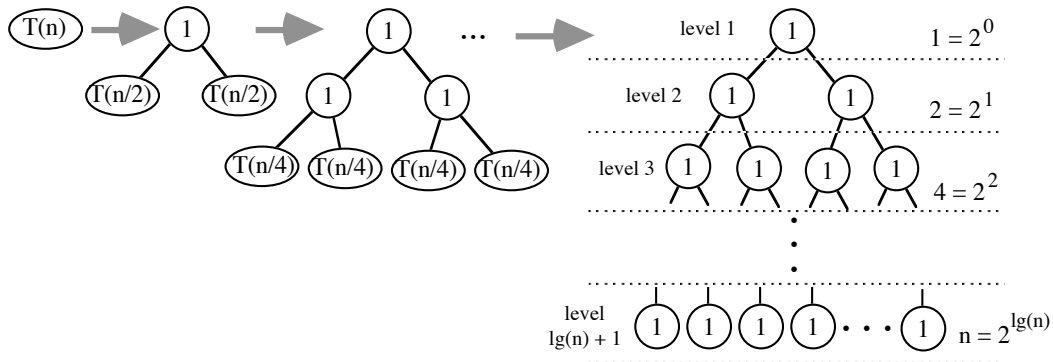
Also derivable: $T(n/2) = 2T(n/4) + n/2$
 $T(n/4) = 2T(n/8) + n/4$



Total cost = $(n)(\text{number of levels}) = n(\lg(n) + 1) = \Theta(n \cdot \lg(n))$

Recurrence 5: $T(n) = 2T(n/2) + 1$

Also derivable: $T(n/2) = 2T(n/4) + 1$
 $T(n/4) = 2T(n/8) + 1$
 etc.



Total cost = number of nodes
 = sum of nodes at each level
 = $1 + 2 + 4 + 8 + \dots + 2^{\lg(n)}$ {Geometric series! See below.}

$$= \sum_{k=0}^{\lg(n)} 2^k =$$

Geometric Series

A series is geometric if $a_k = c \cdot a_{(k-1)}$.

Let $S(n)$ stand for $\sum_{k=0}^n a_0 c^k = a_0 + a_0 c + a_0 c^2 + a_0 c^3 + \dots + a_0 c^n$

(Note carefully: $S(n)$ has $n+1$ terms, *not* n terms!)

Notice the following:

$$\begin{aligned} cS(n) &= a_0 c + a_0 c^2 + a_0 c^3 + \dots + a_0 c^n + a_0 c^{n+1} \\ - S(n) &= a_0 + a_0 c + a_0 c^2 + a_0 c^3 + \dots + a_0 c^n \end{aligned}$$

$$(c - 1) S(n) = a_0 c^{n+1} - a_0 = a_0(c^{n+1} - 1)$$

$$S(n) = \frac{a_0(c^{n+1} - 1)}{(c - 1)}$$

If $0 < c < 1$ and $n \rightarrow \infty$, the above formula can be rewritten as:

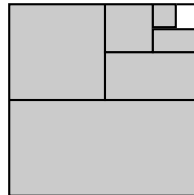
$$\lim_{n \rightarrow \infty} S(n) = \frac{a_0}{(1 - c)}, \text{ if } 0 < c < 1$$

Examples:

$$1 + 2 + 4 + 8 + \dots + 2^n =$$

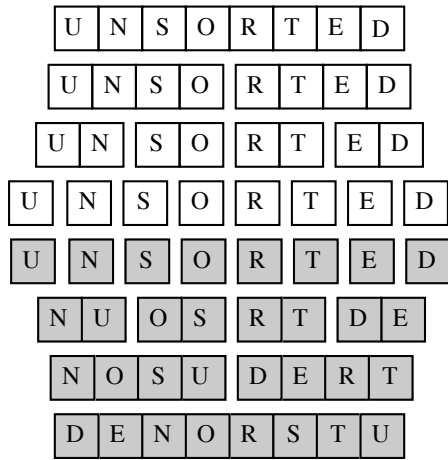
$$1 + 2 + 4 + 8 + \dots + 2^{\lg(n)} =$$

$$1/2 + 1/4 + 1/8 + \dots =$$



$$1/3 + 1/9 + 1/27 + \dots = \quad \{\text{Paper tearing example}\}$$

Analyzing Merge Sort



Idea: Recursively sort subarrays and then merge them into a single sorted array.

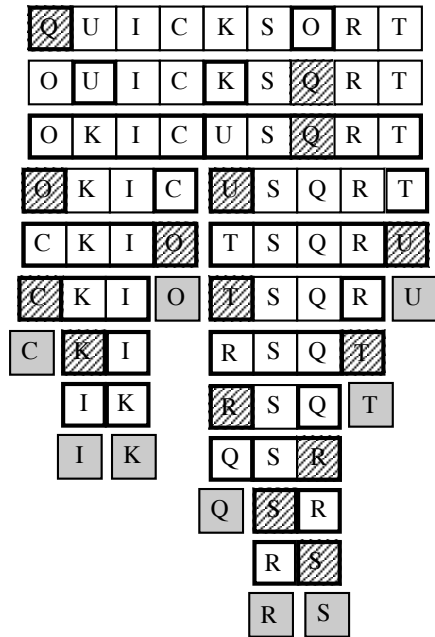
Case	Recurrence	Solution
<i>Worst</i>		
<i>Best</i>		
<i>Average</i>		

Performance Numbers for Merge Sort

Timings for merge sort on Java vectors (in milliseconds)

n	Sorted	Reverse Sorted	Random
1000	23	4	6
2000	11	10	12
4000	23	24	27
8000	51	49	60
16000	110	113	130
32000	242	250	287

Analyzing Quick Sort



Idea:

Choose $A[0]$ as the *pivot*.

Step 1: Partition A into two subarrays by moving elements in A:

1. *lesser*: all elements < pivot
2. *greater*: all elements \geq pivot

The *lesser* subarray should occupy the left portion of A and *greater* should occupy the right portion of A.

One way to partition is to use two fingers moving inward starting at opposite ends of arrays and swapping when left element is \geq pivot and right element is < pivot.

Case	Recurrence	Solution
<i>Worst</i>		
<i>Best</i>		
<i>Average</i>		

Performance Numbers for Quick Sort

Timings for quick sort on Java vectors (in milliseconds)

n	Sorted	Reverse Sorted	Random
1000	48	45	2
2000	170	171	5
4000	680	685	12
8000	2801	2739	26
16000	15246	12062	58
32000	79110	52943	130

Recurrence 6: $T(n) = T(n-1) + \lg(n)$

Also derivable: $T(n-1) = T(n-2) + \lg(n-1)$
 $T(n-2) = T(n-3) + \lg(n-2)$

Picture goes here:

$$\begin{aligned} T(n) &= \lg(n) + \lg(n-1) + \lg(n-2) \dots + \lg(2) + \lg(1) \\ &= \lg(n \cdot (n-1) \cdot (n-2) \cdot \dots \cdot 3 \cdot 2 \cdot 1) \quad \{Because \lg(a) + \lg(b) = \lg(a \cdot b)\} \\ &= \lg(n!) \end{aligned}$$

$\Theta(\lg(n!)) \approx \Theta(n \lg(n))$ by Stirling's approximation

Analyzing Tree Sort

Picture goes here

Idea: Insert A's elements one-by-one into a binary search tree. When done, read elements back into A via an in-order traversal.

Case	Recurrence	Solution
<i>Worst</i>		
<i>Best</i>		
<i>Average</i>		

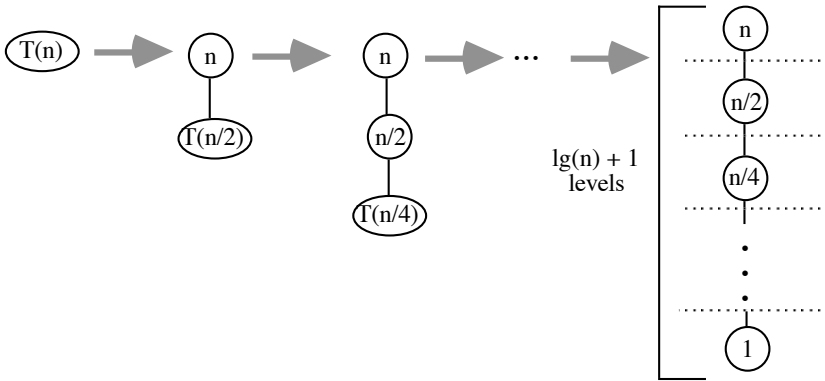
Performance Numbers for Tree Sort

Timings for tree sort on Java vectors (in milliseconds):

n	Sorted	Reverse Sorted	Random
1000	46	37	32
2000	108	121	31
4000	471	481	64
8000	3176	3757	147
16000	24947	31008	391
32000	102832	117779	718

Recurrence 7: $T(n) = T(n/2) + n$

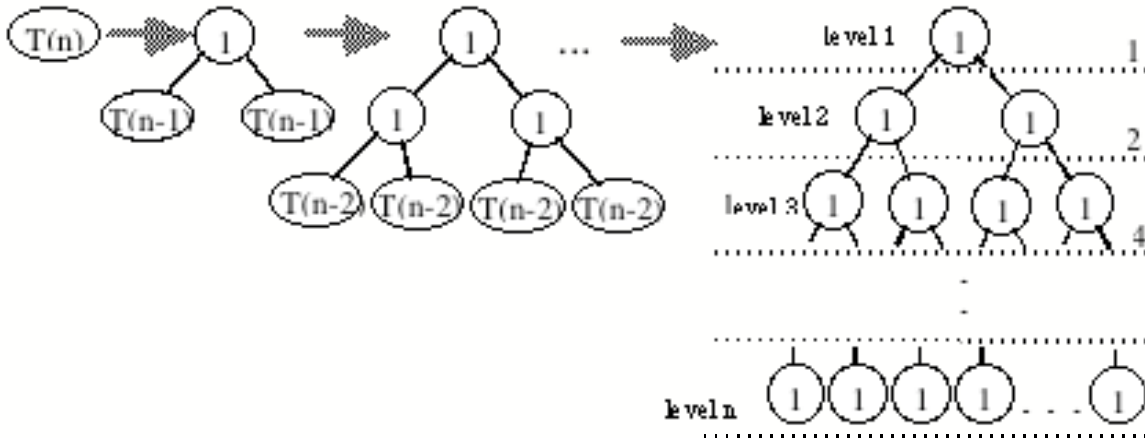
Also derivable: $T(n/2) = T(n/4) + n/2$
 $T(n/4) = T(n/8) + n/4$



Total cost = $n + n/2 + n/4 + \dots + n/2^{\lg(n)} = \sum_{k=0}^{\lg(n)} \frac{n}{2^k} < \sum_{k=0}^{\infty} \frac{n}{2^k} =$

Recurrence 8: $T(n) = 2T(n - 1) + 1$

Also derivable: $T(n-1) = 2T(n-2) + 1$
 $T(n-2) = 2T(n-3) + 1$



Total cost = $1 + 2 + 4 + \dots + 2^{n-1} =$

Summary

The following table summarizes and generalizes the above examples. Equivalence classes of functions are arranged in the table from "biggest" to "smallest". That is, if function f appears above function g in the table, then g is $O(f)$ but f is not $O(g)$ (or, equivalently, f is $\Omega(g)$ but g is not $\Omega(f)$). It is worth noting that there are many more equivalence classes than are listed in the table, but the ones in the table are the ones most commonly encountered in this course.

Equivalence class of functions	Name	Typical Recurrence Equations
$\Theta(3^n)$	exponential (base = 3)	$T(n) = 3 \cdot T(n - a) + b, a > 0, b > 0$
$\Theta(2^n)$	exponential (base = 2)	$T(n) = 2 \cdot T(n - a) + b, a > 0, b > 0$
$\Theta(n^2)$	quadratic	$T(n) = T(n - a) + b \cdot n, a > 0, b > 0$
$\Theta(n \log(n))$	$n \log(n)$	$T(n) = k \cdot T((n-a)/k) + b \cdot n, k > 1, a > 1, b > 1$ or $T(n) = T(n-a) + \log(n), a > 1$ (base of log is irrelevant)
$\Theta(n)$	linear	$T(n) = T(n - a) + b, n > 0, a > 0, b > 0$ or $T(n) = k \cdot T((n-a)/k) + b, k > 1, a > 0, b > 0$ or $T(n) = T((n-a)/k) + b \cdot n, k > 1, a > 1, b > 1$
$\Theta(\log(n))$	logarithmic (base of log is irrelevant)	$T(n) = T(n/k) + a, k > 1, a > 0$
$\Theta(1)$	constant	$T(n) = a$