Introduction to Algorithms

Kyuseok Shim
Electrical and Computer Engineering
Seoul National University

Administrative Information

- Lecturer: Kyuseok Shim
  - Office: 302-531
  - E.Mail: shim@kdd.snu.ac.kr
  - Home page: http://kdd.snu.ac.kr/~shim
- TA
  - 서장혁 jhseo@kdd.snu.ac.kr
  - 구한준 hjkoo@kdd.snu.ac.kr
  - Office: 302-516-2
  - Office hour: Contact via E-mail
- Course Home Page
  - eTL Homepage
  - 강의 슬라이드 download

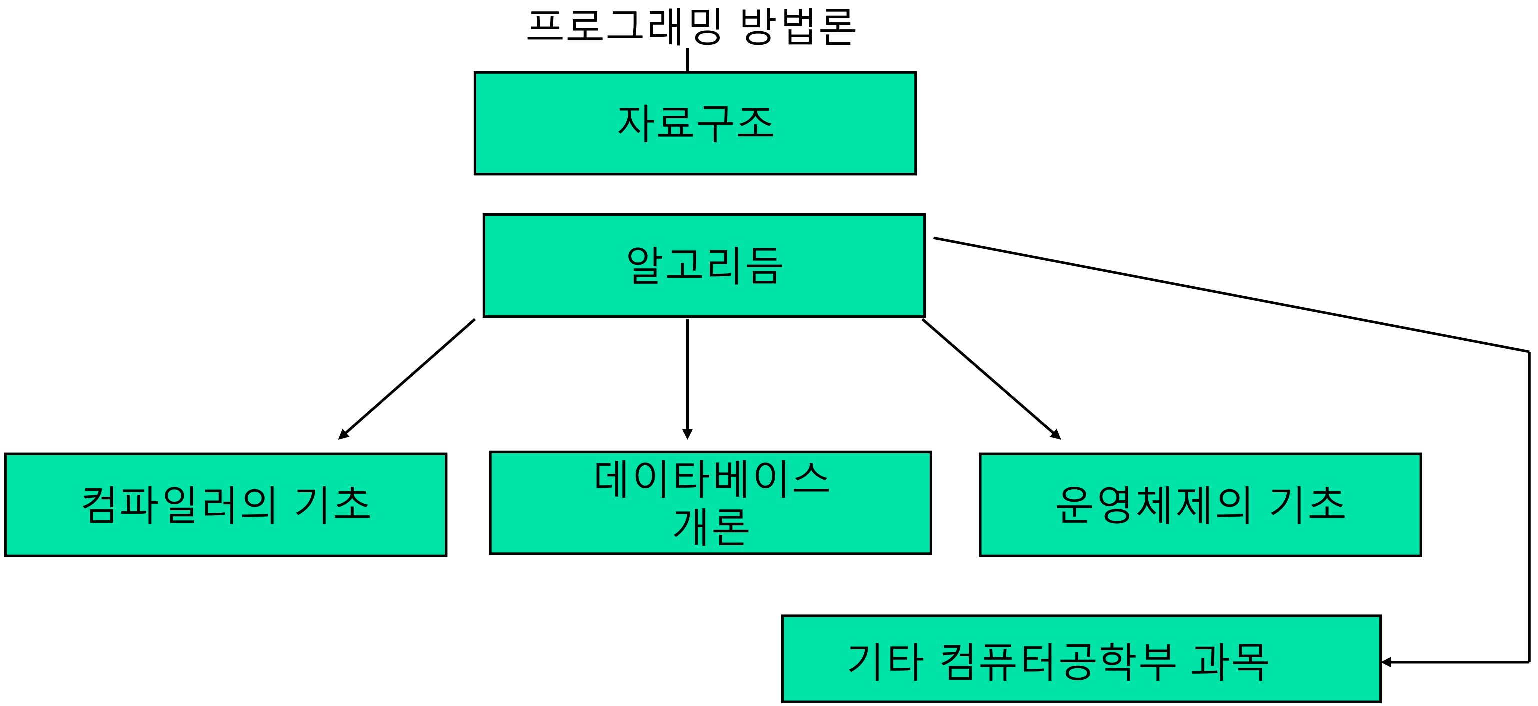
Administrative Information

- Programming Languages for Programming Assignments
  - C++
- Prerequisites
  - Recommended: 전기공학부 프로그래밍 방법론과 자료구조
- How to Succeed in this course:
  - Practice solving many problems both with pencils and computers
  - Make sure to allocate at least one day per week to this course.
  - Ask many questions in class – Don't get lost! If you think you get lost, try to catch up with the help of TAs.

Textbook

- Title: Introduction to Algorithms (third edition)
- Authors: Thomas T. Cormen , Charles E. Leiserson , Ronald L. Rivest, Clifford Stein
- Publisher: MIT Press, Cambridge, MA

Core Software Curriculum



Chapter 2
Getting Started

Outline

- This chapter familiarize you with the framework we shall use throughout the lecture to think about the design and analysis of algorithms.
- We begin by examining the insertion sort algorithm to solve the sorting problem, we then argue that it correctly sorts, and we analyze its running time.
- We next introduce the divide-and-conquer approach to the design of algorithms, use it to develop an algorithm called merge sort, and analyze the merge sort's running time.

Sorting Problem

- Input:
  - A sequence of n numbers $<a_1, a_2, \ldots, a_n>$
- Output:
  - A reordering $<a'_1, a'_2, \ldots, a'_n>$ of the input sequence such that $a'_1 \leq a'_2 \leq \ldots \leq a'_n$
- There are many sorting algorithms
  - Insertion sort
  - Merge sort
  - Quick sort

Insertion Sort

- It uses an incremental approach!
- For a sequence of n numbers A[1..n], it consists of $n-1$ passes.
- For pass j = 2 through n
  - Use the fact that the elements in $A[1..j-1]$ are already known to be in sorted order.
  - Ensures that the elements in A$[1..j]$ are in sorted order.

Insertion Sort

```
INSERTION-SORT(A)
for j = 2 to A.length
    key = A[j]
    // Insert A[j] into the sorted sequence 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
```

Insertion Sort

j = 2

index	1	2	3	4	5	6	7	8
value	3	2	8	6	1	7	4	5

sorted

Insertion Sort

j = 3

index	1	2	3	4	5	6	7	8
value	2	3	8	6	1	7	4	5

sorted

Insertion Sort

j = 4

index	1	2	3	4	5	6	7	8
value	2	3	8	6	1	7	4	5

sorted

Insertion Sort

j = 5

index	1	2	3	4	5	6	7	8
value	2	3	6	8	1	7	4	5

sorted

Insertion Sort

j = 5

index	1	2	3	4	5	6	7	8
value	2	3	6	8	1	7	4	5

sorted

Insertion Sort

j = 6

index	1	2	3	4	5	6	7	8
value	1	2	3	6	8	7	4	5

sorted

Insertion Sort

j = 7

index	1	2	3	4	5	6	7	8
value	1	2	3	6	7	8	4	5

sorted

Insertion Sort

j = 8

index	1	2	3	4	5	6	7	8
value	1	2	3	4	6	7	8	5

sorted

Insertion Sort

j = 8

index	1	2	3	4	5	6	7	8
value	1	2	3	4	5	6	7	8

sorted

Loop Invariants and the Correctness Proof

- We use loop invariants to help us understand why an algorithm is correct.
- We must show three things about a loop. invariant:
  - Initialization: It is true prior to the first iteration of the loop.
  - Maintenance: If it is true before an iteration of the loop, it remains true before the next iteration.
  - Termination: When the loop terminates, the invariant gives us a useful property that helps show that the algorithm is correct.

Insertion Sort

- Loop invariant
  - At the start of each iteration of the for-loop of lines 1-8, the subarray $A[1 \dots j-1]$ consists of the elements originally in $A[1 \dots j-1]$ but in ascending order.
- Initialization:
  - When $\mathrm{j} = 2$, $A[1 \dots j-1]$ consists of just the single element $A[1]$ which is the original one in $A[1]$. Moreover, the subarray is sorted. Thus, loop invariant holds prior to the first iteration of the loop.

Insertion Sort

- Loop invariant
  - At the start of each iteration of the for-loop of lines 1-8, the subarray $A[1 \dots j-1]$ consists of the elements originally in $A[1 \dots j-1]$ but in ascending order.
- Maintenance:
  - The body of outer for-loop works by moving $A[j-1]$, $A[j-2]$, $A[j-3]$, and so on by one position to the right until the proper position for $A[j]$ is found (lines4-7), at which point the value of $A[j]$ is inserted (line 8)
  - The subarray $A[1 \dots j]$ then consists of the elements originally in $A[1 \dots j]$, but in sorted order
  - Incrementing j for the next iteration of the for loop then preserves the loop invariant

Insertion Sort

- Loop invariant
  - At the start of each iteration of the for-loop of lines 1-8, the subarray $A[1 \dots j-1]$ consists of the elements originally in $A[1 \dots j-1]$ but in ascending order.
- Termination:
  - When the for loop terminates, we have j = n+1
  - Substituting n+1 for j in the wording of loop invariant, we have that $A[1 \dots n]$ consists of the elements originally in $A[1 \dots n]$, but in ascending order. Hence, the entire array is sorted, which means that the algorithm is correct.

Divide and Conquer

- We solve a problem recursively by applying three steps at each level of the recursion:
  - Divide the problem into a number of subproblems that are smaller instances of the same problem.
  - Conquer the subproblem by solving them recursively.
    - If the problem sizes are small enough (i.e. we have gotten down to the base case), solve the subproblem in a straightforward manner
  - Combine the solutions to the subproblems into the solution for the original problem.

Merge Sort

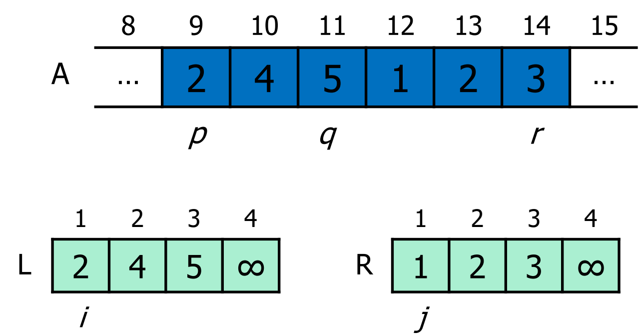
- Merge Sort algorithm closely follows the divide-and-conquer paradigm. Intuitively, it operates as follows.
  - Divide: Divide the n-element sequence to be sorted into two subsequences of n=2 elements each.
  - Conquer: Sort the two subsequences recursively using merge sort.
  - Combine: Merge the two sorted subsequences to produce the sorted answer.
- The recursion “bottoms out” when the sequence to be sorted has length 1, in which case there is no work to be done, since every sequence of length 1 is already in sorted order

Merge Procedure

- The key operation of merge sort algorithm.
- The procedure assumes that that the subarrays A[p..q] and A[q+1..r] are in sorted order.
- It merges them to form a single sorted subarray that replaces the current subarray A[p..r].
- We merge by calling an auxiliary procedure MERGE(A, p, q, r) where A is an array and p, q, and r are indices into the array such that $p \leq q < r$.

Merge Procedure

```
MERGE(A, p, q, r)
n1 = q – p + 1
n2 = r – q
let L[1 ... n1+1] and R[1 ... n2+1] be new arrays
for i = 1 to n1
        L[i] = A[p+i-1]
for j = 1 to n2
        R[j] = A[q+j]
L[n1+1] = ∞
R[n2+1] = ∞
i = 1
j = 1
for k = p to r
        if L[i] ≤ R[j]
                A[k] = L[i]
                i = i + 1
        else A[k] = R[j]
                j = j + 1
```



Operations of Merge(A, 9, 11, 14)

	8	9	10	11	12	13	14	15
A	…	2	4	5	1	2	3	…
		k						

	1	2	3	4
L	2	4	5	∞
	i			

	1	2	3	4
R	1	2	3	∞
	j			

Operations of Merge(A, 9, 11, 14)

	8	9	10	11	12	13	14	15
A	…	1	4	5	1	2	3	…
			k					

	1	2	3	4
L	2	4	5	∞
	i			

	1	2	3	4
R	1	2	3	∞
		j		

Operations of Merge(A, 9, 11, 14)

	8	9	10	11	12	13	14	15
A	…	1	2	5	1	2	3	…
				k				

	1	2	3	4
L	2	4	5	∞
		i		

	1	2	3	4
R	1	2	3	∞
		j		

Operations of Merge(A, 9, 11, 14)

	8	9	10	11	12	13	14	15
A	…	1	2	2	1	2	3	…
					k			

	1	2	3	4
L	2	4	5	∞
		i		

	1	2	3	4
R	1	2	3	∞
			j	

Operations of Merge(A, 9, 11, 14)

	8	9	10	11	12	13	14	15
A	…	1	2	2	3	2	3	…
						k		

	1	2	3	4
L	2	4	5	∞
		i		

	1	2	3	4
R	1	2	3	∞
				j

Operations of Merge(A, 9, 11, 14)

	8	9	10	11	12	13	14	15
A	…	1	2	2	3	4	3	…
							k	

	1	2	3	4
L	2	4	5	∞
			i	

	1	2	3	4
R	1	2	3	∞
				j

Operations of Merge(A, 9, 11, 14)

	8	9	10	11	12	13	14	15
A	…	1	2	2	3	4	5	…
							k	

	1	2	3	4
L	2	4	5	∞
				i

	1	2	3	4
R	1	2	3	∞
				j

Merge Procedure

```
MERGE(A, p, q, r)
n1 = q – p + 1
n2 = r – q
let L[1 ... n1+1] and R[1 ... n2+1] be new arrays
for i = 1 to n1
        L[i] = A[p+i-1]
for j = 1 to n2
        R[j] = A[q+j]
L[n1+1] = ∞
R[n2+1] = ∞
i = 1
j = 1
for k = p to r
        if L[i] ≤ R[j]
                A[k] = L[i]
                i = i + 1
        else A[k] = R[j]
                j = j + 1
```

Correctness of Merge Procedure

- Loop Invariant:
  - At the start of each iteration for the for-loop of lines 12-17, the subarray $A[p \ldots k-1]$ contains the $(k-p)$ smallest elements of $L[1 \ldots n_1+1]$ and $R[1 \ldots n_2+1]$, in ascending order.
  - Moreover, $L[i]$ and $R[j]$ are the smallest elements of their arrays that have not been copied back into A.

Correctness of Merge Procedure

- Loop Invariant:
  - At the start of each iteration for the for-loop of lines 12-17, the subarray $A[p \ldots k-1]$ contains the $(k-p)$ smallest elements of $L[1 \ldots n_1+1]$ and $R[1 \ldots n_2+1]$, in ascending order.
  - Moreover, $L[i]$ and $R[j]$ are the smallest elements of their arrays that have not been copied back into A.
- Initialization:
  - Prior to the first iteration of the loop, we have $k=p$, so that the subarray $A[p \ldots k-1]$ is empty.
  - The empty subarray contains the (k-p=0) smallest elements in L and R.
  - Since i=j=1, both $L[i]$ and $R[j]$ are the smallest elements of their arrays that have not been copied back into A.

Correctness of Merge Procedure

- Loop Invariant:
  - At the start of each iteration for the for-loop of lines 12-17, the subarray $A[p \dots k-1]$ contains the $(k-p)$ smallest elements of $L[1 \dots n_1+1]$ and $R[1 \dots n_2+1]$, in ascending order.
  - Moreover, $L[i]$ and $R[j]$ are the smallest elements of their arrays that have not been copied back into A.
- Maintenance:
  - When $L[i] \leq R[j]$,
    - $L[i]$ is the smallest element not yet copied back into A
    - Because $A[p \dots k-1]$ contains the $k-p$ smallest elements, after line 14 copies $L[i]$ into $A[k]$, the subarray $A[p \dots k]$ will contain the $k-p+1$ smallest elements.
    - Incrementing k and i (in line 15) reestablishes the loop invariant for the next iteration.
  - When $L[i] > R[j]$,
    - the lines 16-17 perform the appropriate action to maintain the loop invariant.

Correctness of Merge Procedure

- Loop Invariant:
  - At the start of each iteration for the for-loop of lines 12-17, the subarray $A[p \dots k-1]$ contains the $(k-p)$ smallest elements of $L[1 \dots n_1+1]$ and $R[1 \dots n_2+1]$, in ascending order.
  - Moreover, $L[i]$ and $R[j]$ are the smallest elements of their arrays that have not been copied back into A.
- Termination:
  - At termination, $k = r + 1$.
  - By the loop invariant, the subarray $A[p \dots k-1]$, which is $A[p \dots r]$, contains the $(k-p) = (r-p+1)$ smallest elements of $L[1 \dots n_1+1]$ and $R[1 \dots n_2+1]$, in sorted order.
  - The arrays L and R together contain $n_1 + n_2 + 2 = r - p + 3$ elements.
  - All but the two largest have been copied back into A, and these two largest elements are the sentinels.

Merge Sort

- Merge Sort algorithm operates as follows
  - Divide: The divide step just computes the middle of the subarray, which takes Θ(1) time
  - Conquer: We recursively solve two subproblems, each of size n/2, which contributes 2T(n/2) to the running time.
  - Combine: We have already noted that the MERGE procedure on an n-element subarray takes Θ(n) time
- Thus, the recurrence for the worst-case running time T(n) of merge sort is
  - $T(1)=1$
    $T(n)=2T(n/2)+n$

Merge Sort

MERGE-SORT(A,p,r)

```
if p < r
q = ⌊(p+r)/2⌋
    MERGE-SORT (A,p,q)
    MERGE-SORT (A,q+1,r)
    MERGE(A,p,q,r)
```

- Merge() is the procedure to merge two sorted lists.

Solving Recurrences

$T(1) = 1$

$T(n) = 2T\left(\frac{n}{2}\right) + n$ for n > 1

Let $n = 2^k$. Then,

$$T(n) = 2T\left(\frac{n}{2}\right) + n = 2\left(2T\left(\frac{n}{2^2}\right) + \frac{n}{2}\right) + n$$

Solving Recurrences

$T(1) = 1$

$T(n) = 2T\left(\frac{n}{2}\right) + n$ for n > 1

Let $n = 2^k$. Then,

$$T(n) = 2T\left(\frac{n}{2}\right) + n = 2\left(2T\left(\frac{n}{2^2}\right) + \frac{n}{2}\right) + n$$
$$= 2^2\, T\left(\frac{n}{2^2}\right) + 2n = 2^2\left(2T\left(\frac{n}{n^3}\right) + \frac{n}{2^2}\right) + 2n$$

Solving Recurrences

$T(1) = 1$

$T(n) = 2T\left(\frac{n}{2}\right) + n$ for n > 1

Let $n = 2^k$. Then,

$$
\begin{aligned}
T(n) &= 2T\left(\frac{n}{2}\right) + n = 2\left(2T\left(\frac{n}{2^2}\right) + \frac{n}{2}\right) + n \\
&= 2^2\, T\left(\frac{n}{2^2}\right) + 2n = 2^2\left(2T\left(\frac{n}{n^3}\right) + \frac{n}{2^2}\right) + 2n \\
&= 2^3 T\left(\frac{n}{2^3}\right) + 3n
\end{aligned}
$$

Solving Recurrences

$T(1) = 1$

$T(n) = 2T\left(\frac{n}{2}\right) + n$ for n > 1

Let $n = 2^k$. Then,

$$\begin{aligned} T(n) &= 2T\left(\frac{n}{2}\right) + n = 2\left(2T\left(\frac{n}{2^2}\right) + \frac{n}{2}\right) + n \\ &= 2^2\ T\left(\frac{n}{2^2}\right) + 2n = 2^2\left(2T\left(\frac{n}{n^3}\right) + \frac{n}{2^2}\right) + 2n \\ &= 2^3 T\left(\frac{n}{2^3}\right) + 3n \\ &\dots \\ &= 2^k T\left(\frac{n}{2^k}\right) + kn \end{aligned}$$

When $\frac{n}{2^k}$ = 1,
we have n = 2^k and k = lg n

Solving Recurrences

$T(1) = 1$

$T(n) = 2T\left(\frac{n}{2}\right) + n$ for n > 1

Let $n = 2^k$. Then,

$$\begin{aligned}
T(n) &= 2T\left(\frac{n}{2}\right) + n = 2\left(2T\left(\frac{n}{2^2}\right) + \frac{n}{2}\right) + n \\
&= 2^2\, T\left(\frac{n}{2^2}\right) + 2n = 2^2\left(2T\left(\frac{n}{n^3}\right) + \frac{n}{2^2}\right) + 2n \\
&= 2^3 T\left(\frac{n}{2^3}\right) + 3n \\
&\ldots \\
&= 2^k T\left(\frac{n}{2^k}\right) + kn \\
&= nT(1) + n \lg n
\end{aligned}$$

When $\frac{n}{2^k}$ = 1,
we have n = 2^k and k = lg n

Solving Recurrences

$T(1) = 1$

$T(n) = 2T\left(\frac{n}{2}\right) + n$ for n > 1

Let $n = 2^k$. Then,

$$
\begin{aligned}
T(n) &= 2T\left(\frac{n}{2}\right) + n = 2\left(2T\left(\frac{n}{2^2}\right) + \frac{n}{2}\right) + n \\
&= 2^2\, T\left(\frac{n}{2^2}\right) + 2n = 2^2\left(2T\left(\frac{n}{n^3}\right) + \frac{n}{2^2}\right) + 2n \\
&= 2^3 T\left(\frac{n}{2^3}\right) + 3n \\
&\ldots \\
&= 2^k T\left(\frac{n}{2^k}\right) + kn \\
&= nT(1) + n\lg n \\
&= n + n\lg n
\end{aligned}
$$

When $\frac{n}{2^k} = 1$,
we have n = 2^k and k = lg n

Operations of Merge Sort

85 24 63 45 17 31 96 50

Initial sequence

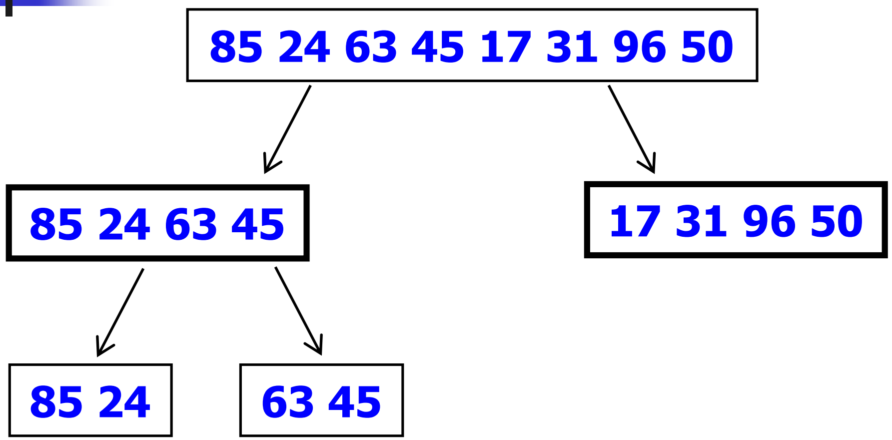
Operations of Merge Sort

85 24 63 45 17 31 96 50

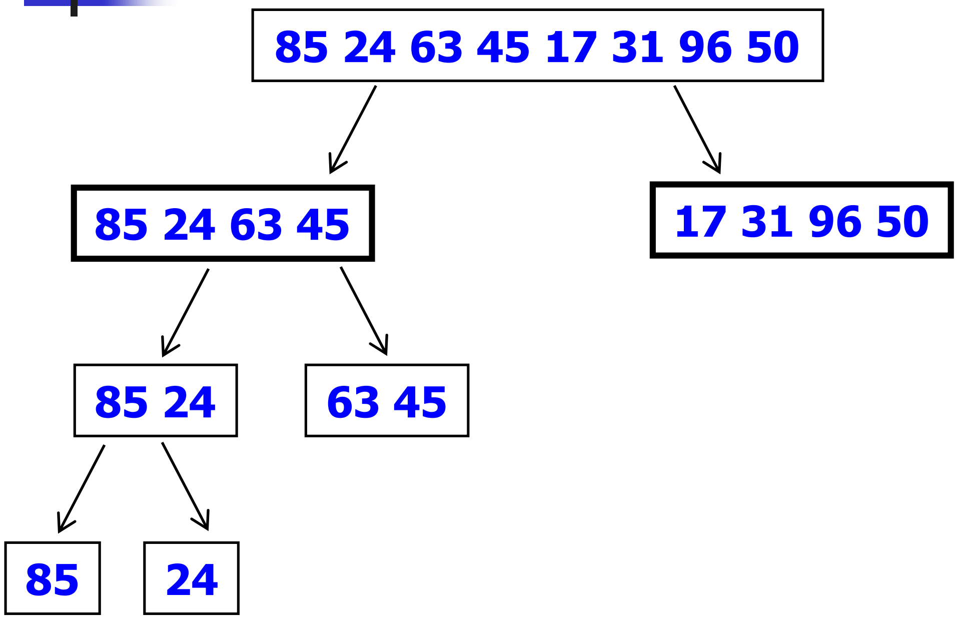
85 24 63 45

17 31 96 50

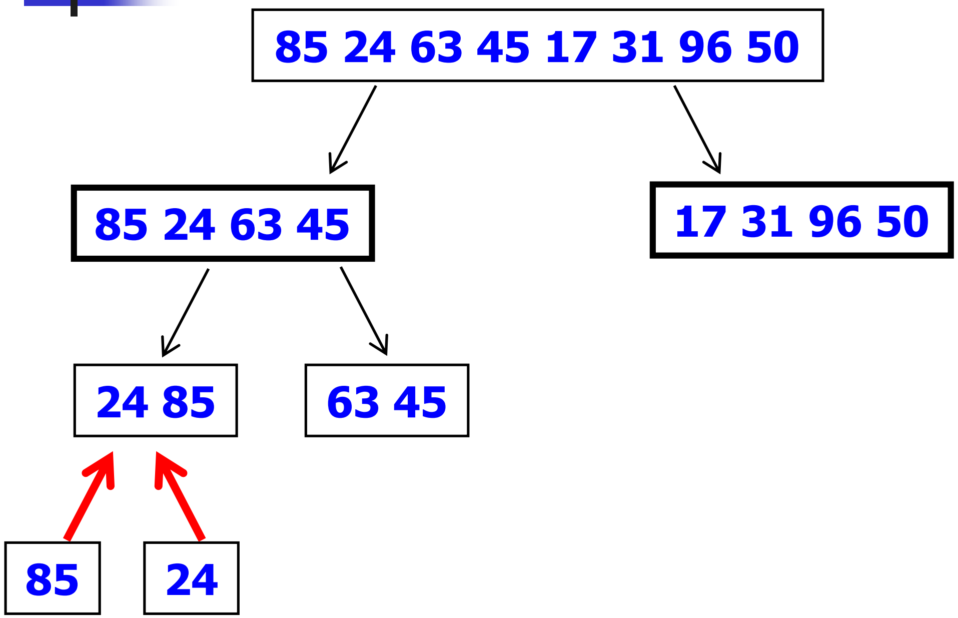
Operations of Merge Sort



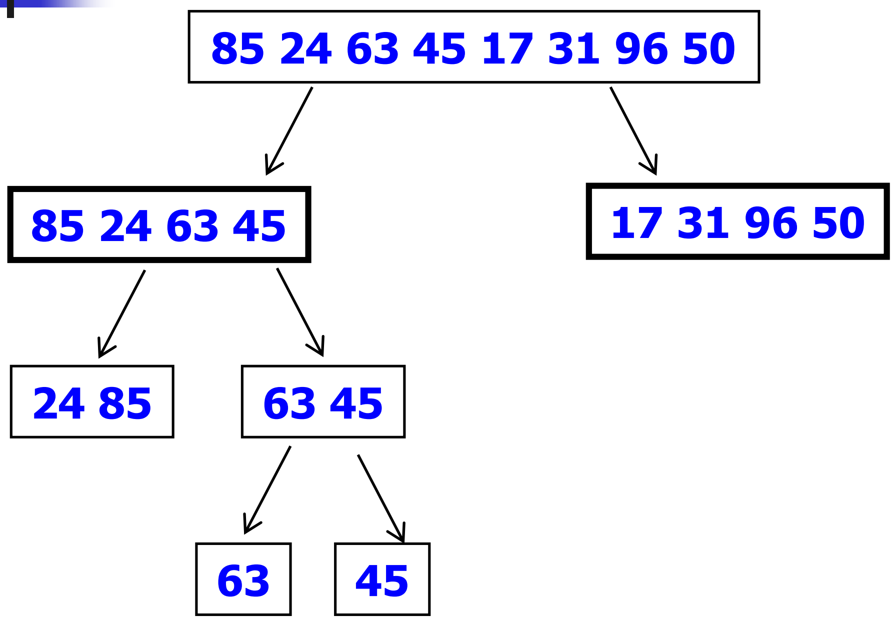
Operations of Merge Sort



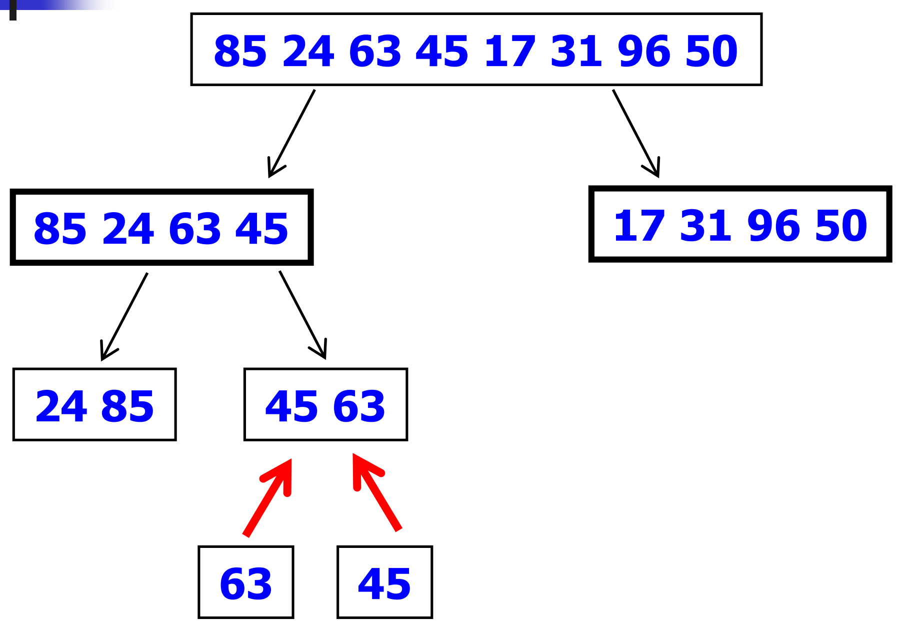
Operations of Merge Sort



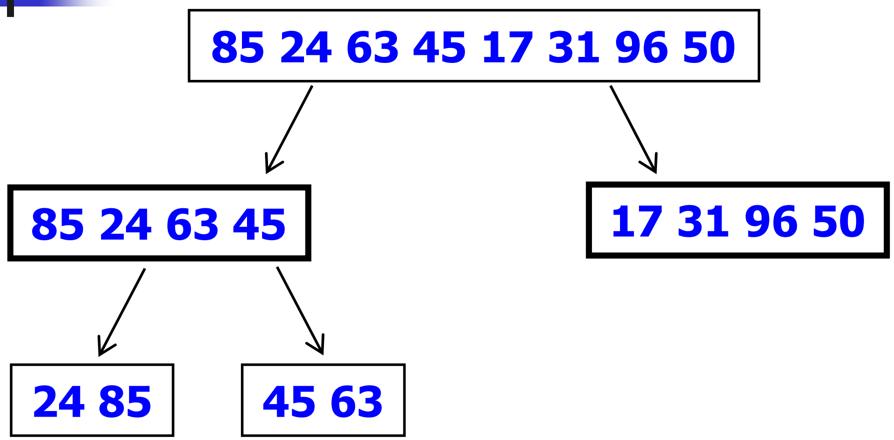
Operations of Merge Sort



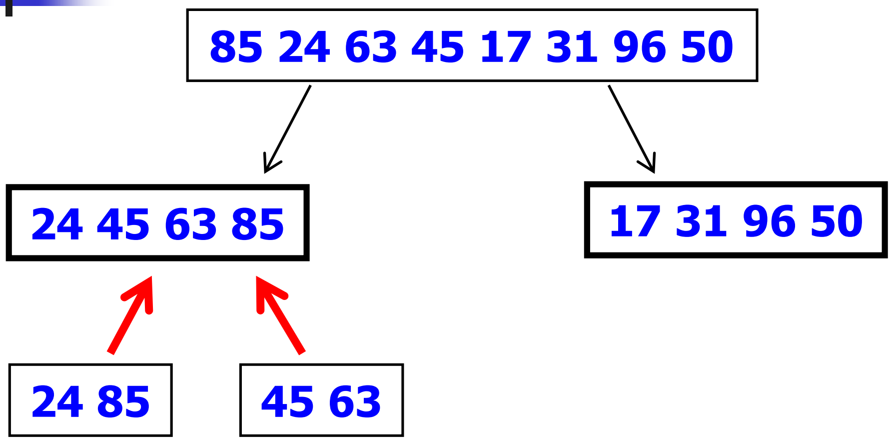
Operations of Merge Sort



Operations of Merge Sort



Operations of Merge Sort



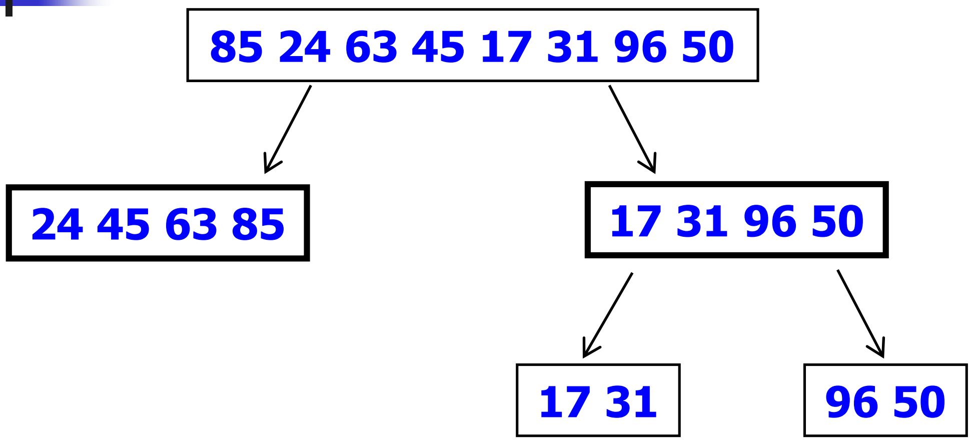
Operations of Merge Sort

85 24 63 45 17 31 96 50

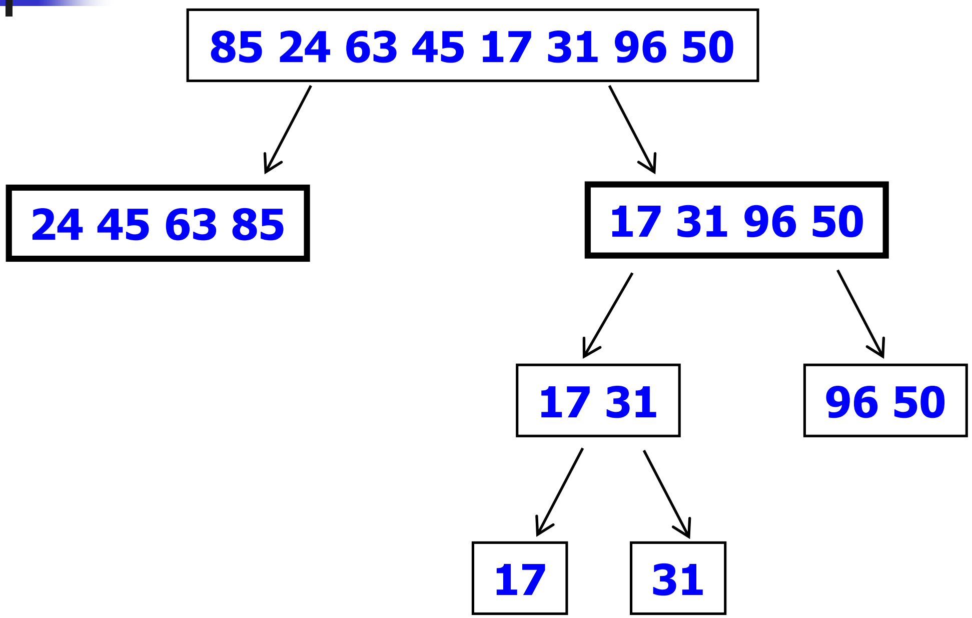
24 45 63 85

17 31 96 50

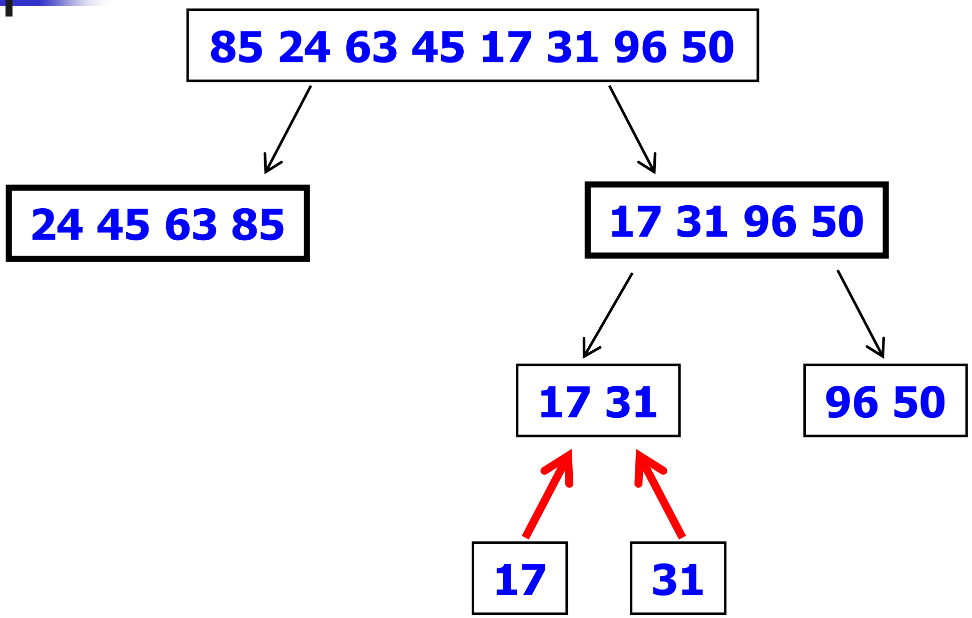
Operations of Merge Sort



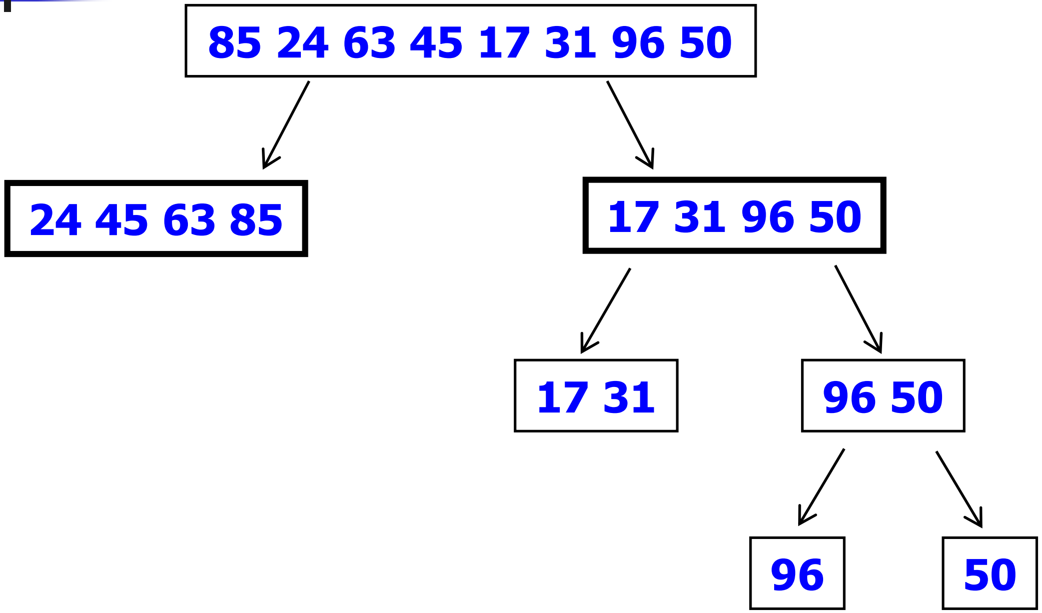
Operations of Merge Sort



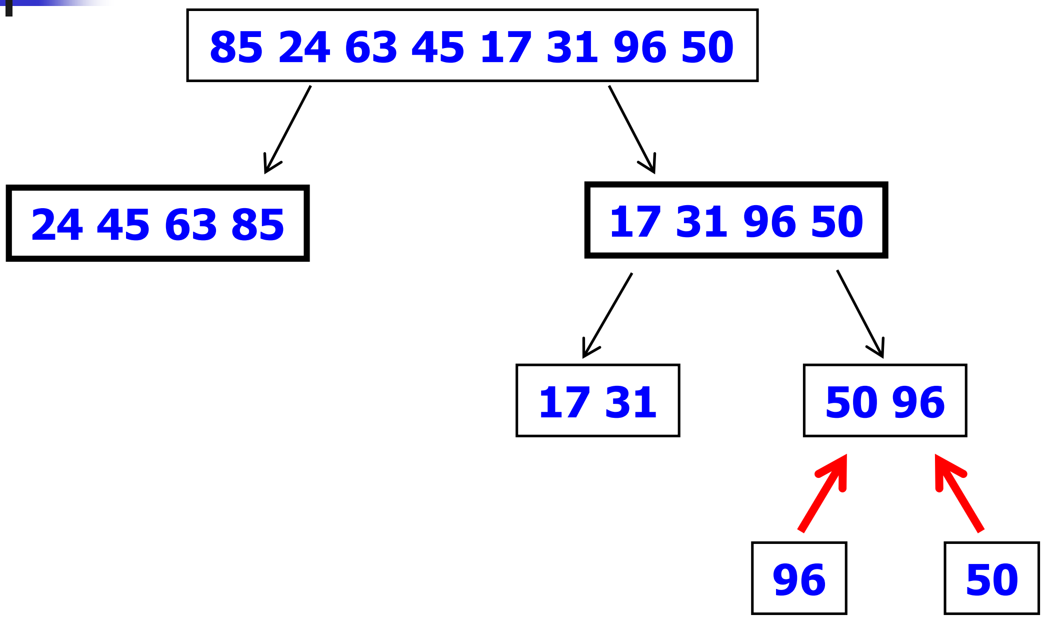
Operations of Merge Sort



Operations of Merge Sort



Operations of Merge Sort



Operations of Merge Sort

85 24 63 45 17 31 96 50

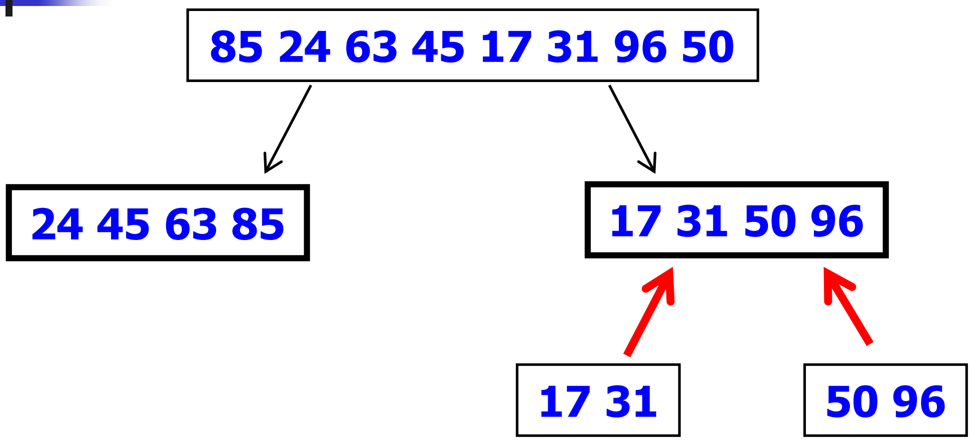
24 45 63 85

17 31 96 50

17 31

50 96

Operations of Merge Sort



Operations of Merge Sort

85 24 63 45 17 31 96 50

24 45 63 85

17 31 50 96

Operations of Merge Sort

17 24 31 45 50 63 85 96

24 45 63 85

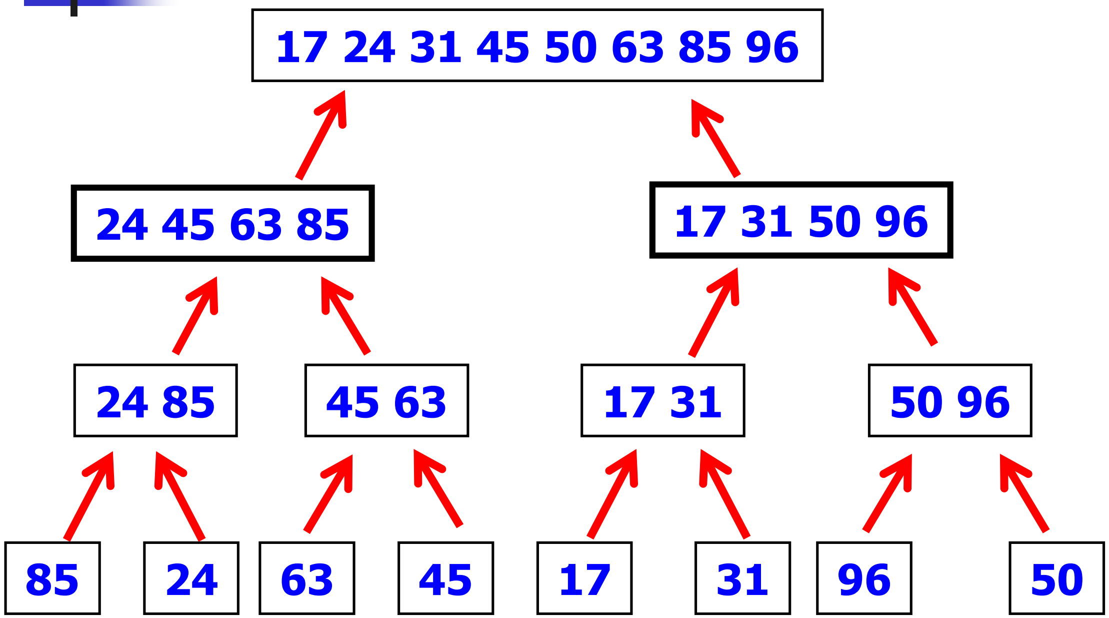
17 31 50 96

Operations of Merge Sort

17 24 31 45 50 63 85 96

Sorted sequence

Operations of Merge Sort



Any Question?

Introduction to Algorithms (Chapter3-4)

Kyuseok Shim

Electrical and Computer Engineering
Seoul National University

Chapter 3: Growth of Functions

Algorithm Analysis

- How much better is one curve than another one (answer: typically a lot, for large inputs)
- How do we decide which curve a particular algorithm lies on (answer: sometimes it's easy, sometimes it's hard).
- How to use this information to design better algorithms (answer: definitely).
- Can we predict how an algorithm will perform for large input sets, based on its performance for moderate input sets (answer: definitely).

Algorithm Analysis

- Running time of an algorithm almost always depends on the amount of input: More input means more time. Thus the running time, T, is a function of the amount of input, n, or $T(n) = f(n)$.
- The exact value of the function depends on
  - the speed of the host machine
  - the quality of the compiler and optimizer
  - the quality of the program that implements the algorithm
  - the basic fundamentals of the algorithm
- Typically, the last item is the most important.

Θ-notation

- For a given function g(n), we define
  - $\Theta(g(n)) = \{ f(n)$: there exist positive constants c_1, c_2, and n_0 such that $0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)$ for all $n \geq n_0 \}$.
- A function f(n) belongs to the set Θ(g(n)) if there is positive constants c_1 and c_2 such that it can be "sandwiched" between $c_1 g(n)$ and $c_2 g(n)$, for sufficiently large n.
- Because Θ(g(n)) is a set, we could write "$f(n) \in \Theta(g(n))$" to indicate that f(n) is a member of Θ(g(n)).
- Instead, we write "$f(n) = \Theta(g(n))$" to express the same notion.

Θ-notation

- For all values of n at and to the right of n_0, the value of f(n) lies at or above $c_1g(n)$ and at or below $c_2g(n)$.
- In other words, for all n ≥ n0, the function f(n) is equal to g(n) to within a constant factor.
- We say that g(n) is an asymptotically tight bound for f(n).
- The definition of Θ(g(n)) requires that every member f(n) ∈ Θ(g(n)) be asymptotically nonnegative, that is, that f(n) be nonnegative whenever n is sufficiently large.
- Consequently, the function g(n) itself must be asymptotically nonnegative, or else the set Θ(g(n)) is empty.
- We shall therefore assume that every function used within Θ-notation is asymptotically nonnegative.

Θ-notation

- We can throw away lower-order terms and ignore the leading coefficient of the highest-order term
- To justify $\frac{1}{2}n^2 - 3n = \Theta(n^2)$, we must determine positive constants c_1, c_2, and n_0 such that
  - $c_1 n^2 \leq \frac{1}{2}n^2 - 3n \leq c_2 n^2$ for all $n \geq n_0$.
- Dividing by n^2 yields
  - $c_1 \leq \frac{1}{2} - \frac{3}{n} \leq c_2$.
- The right-hand inequality hold for any value of $n \geq 1$ by choosing any constant $c_2 \geq \frac{1}{2}$.
- Likewise, the left-hand inequality hold for any value of $n \geq 7$ by choosing any constant $c_1 \leq \frac{1}{14}$.
- Thus, by choosing $c_1 = \frac{1}{14}$, $c_2 \geq \frac{1}{2}$, and $n_0 \geq 7$, we can verify that $\frac{1}{2}n^2 - 3n = \Theta(n^2)$.

Θ-notation

- $f(n)=\Theta(g(n))$ iff there exist positive constants c_1, c_2 and n_0 such that
  - $c_1 g(n) \le f(n) \le c_2 g(n)$ for all n, $n \ge n_0$
- $g(n)$ is both an upper and lower bound
- Examples
  - $f(n) = 3n+2=\Theta(n)$
  - $f(n) = 10n^2+4n+2=\Theta(n^2)$
  - $f(n) = 6\times 2^n+n^2=\Theta(2^n)$

O-notation

- Gives an upper bound on a function, to within a constant factor.
- $O(g(n))$ is pronounced "big-oh of g of n" or often just "oh of g of n".
- For a given function $g(n)$, we define

  - $O(g(n)) = \{ f(n)$: there exist positive constants c and n_0 such that $0 \leq f(n) \leq cg(n)$ for all $n \geq n_0 \}$.

- $f(n) = O(g(n))$ indicates that a function $f(n)$ is on or below $cg(n)$.
- Since Θ-notation is a stronger notion than O-notation, $f(n) = \Theta(g(n))$ implies $f(n) = O(g(n))$ (i.e., we have $\Theta(g(n)) \in O(g(n))$).
- We sometimes find O-notation informally describing asymptotically tight bounds (i.e., what we have defined using Θ-notation).

O-notation

- $f(n)=O(g(n))$ iff there exist positive constants c and n_0 such that
  - $f(n) \le cg(n)$ for all n, $n \ge n_0$
- Examples
  - $f(n) = 3n+3 = O(n)$ as $3n+3 \le 4n$ for $n \ge 3$
  - $f(n) = 3n+3 = O(n^2)$ as $3n+3 \le 3n^2$ for $n \ge 2$

O-notation

- Using O-notation, we can often describe the running time of an algorithm merely by inspecting the algorithm's overall structure.
- For example, the doubly nested loop structure of the insertion sort algorithm from Chapter 2 immediately yields an $O(n^2)$ upper bound on the worst-case running time:
  - the cost of each iteration of the inner loop is bounded from above by $O(1)$ (constant)
  - the indices i and j are both at most n
  - the inner loop is executed at most once for each of the n^2 pairs of values for i and j
- Since O-notation describes an upper bound, when we use it to bound the worst case running time of an algorithm, we have a bound on the running time of the algorithm on every input.
- Thus, the $O(n^2)$ bound on worst-case running time of insertion sort also applies to its running time on every input.
- The $\Theta(n^2)$ bound on the worst-case running time of insertion sort, however, does not imply a $\Theta(n^2)$ bound on the running time of insertion sort on *every* input.
- For example, we saw in Chapter 2 that when the input is already sorted, insertion sort runs in $\Theta(n)$ time.

Ω-notation

- Gives an upper bound on a function, to within a constant factor
- Ω(g(n)) is pronounced "big-omega of g of n" or often just "omega of g of n"
- For a given function g(n), we define
  - $\Omega(g(n)) = \{ f(n)$: there exist positive constants c and n_0 such that $0 \leq cg(n) \leq f(n)$ for all $n \geq n_0 \}$
- f(n) =Ω(g(n)) indicates that a function f(n) is on or above cg(n)
- Since Θ-notation is a stronger notion than O-notation, f(n) = Θ(g(n)) implies f(n) =Ω(g(n)) (i.e., we have Θ(g(n)) ∈ Ω(g(n)))
- We sometimes find Ω-notation informally describing asymptotically tight bounds (i.e., what we have defined using Θ-notation)

Ω-notation

- $f(n)=\Omega(g(n))$ iff there exist positive constants c and n_0 such that
  - $f(n) \geq cg(n)$ for all n, $n \geq n_0$
- g(n) is a lower bound
- Examples
  - $f(n) = 3n+2=\Omega(n)$
  - $f(n) = 10n^2+4n+2=\Omega(n^2)$

Asymptotic Notations

- Theorem 3.1
  - For any two functions $f(n)$ and $g(n)$, we have $f(n) = \Theta(g(n))$ if and only if
    $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$

- Thus, $an^2 + bn + c = \Theta(n^2)$ for any constants a, b, and c, where $a > 0$, immediately implies

$an^2 + bn + c = O(n^2)$ and $an^2 + bn + c = \Omega(n^2)$

Asymptotic Notations

- When asymptotic notation appears in a formula, we interpret it as standing for some anonymous function that we do not care to name
  - For example, the formula $2n^2 + 3n + 1 = 2n^2 + \Theta(n)$ means that $2n^2 + 3n + 1 = 2n^2 + f(n)$, where $f(n)$ is some function in the set $\Theta(n)$
  - In this case, we let $f(n)= 3n + 1$, which indeed is in $\Theta(n)$
- Using asymptotic notation in this manner can help eliminate inessential detail and clutter in an equation
  - For example, we can express the worst-case running time of an algorithm as the recurrence $T(n) = 2T(n/2) + \Theta(n)$
- If we are interested only in the asymptotic behavior of $T(n)$, there is no point in specifying all the lower-order terms exactly

Properties of Big-Oh

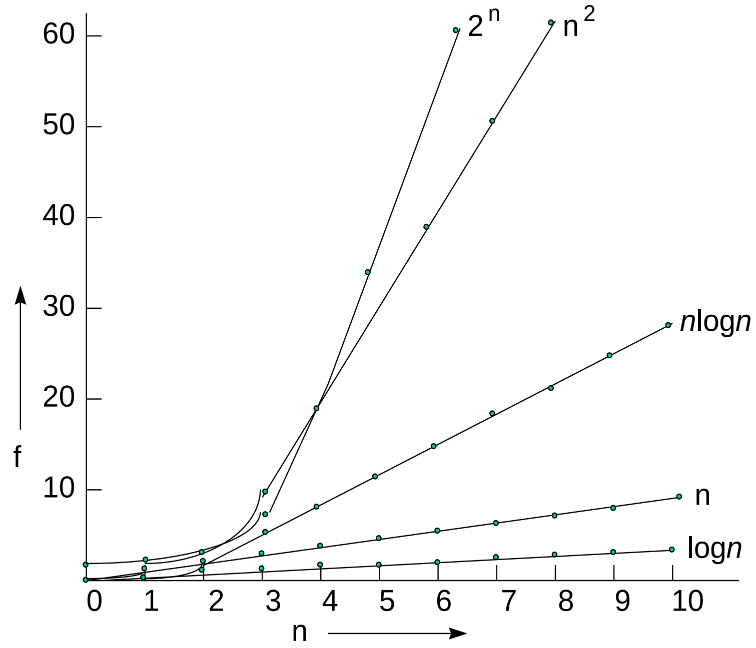
- If $T_1(n) = O(f(n))$ and $T_2(n) = O(g(n))$, then
  - $T_1(n) + T_2(n) = \max(O(f(n)), O(g(n))$
    - Lower-order terms are ignored
  - $T1(n) * T2(n) \ = O(f(n) * g(n))$
- $O(c * f(n)) = O(f(n))$ for some constant c
  - Constants are ignored!
- In reality, constants and lower-order terms may matter, especially when the input size is small.

Big-Oh

- Cubic: dominant term is some constant times n^3. We say $O(n^3)$.
- Quadratic: dominant term is some constant times n^2. We say $O(n^2)$.
- $O(n\ logn)$: dominant term is some constant times $n\ logn$.
- Linear: dominant term is some constant times n. We say $O(n)$.
- Example: $350n^2 + n + n^3$ is cubic.
- Big-Oh ignores leading constants.

Practical Complexities

- For large n, only programs of small complexity are feasible



Dominant Term Matters

- Suppose we estimate $350n^2 + n + n^3$ with n^3.
- For $n = 10000$:
  - Actual value is 1,003,500,010,000
  - Estimate is 1,000,000,000,000
  - Error in estimate is 0.35%, which is negligible.
- For large n, dominant term is usually indicative of algorithm's behavior.
- For small n, dominant term is not necessarily indicative of behavior, **BUT**, typically programs on small inputs run so fast we don't care anyway.

Running Time Calculation

- Summations for Loops

```
for  i = 1 to n do {
        . . . .
        . . . .
     }
                  (a)
```

```
for  i = 1 to n do {
     for j = 1 to n do {
          . . . . . . .
     }
}        (b)
```

If the loop of (a) takes $f(i)$ times, $T(n)=\sum_{i=1}^{n} f(i)$

If the loop of (b) takes $g(i,j)$ times, $T(n)=\sum_{i=1}^{n}\sum_{j=1}^{n} g(i,j)$

$\sum_{i=1}^{n} 1 = n$ **constant sum**

$\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$ **the linear sum**

$\sum_{i=1}^{n} c^i = \frac{c^{n+1}-1}{c-1}$, $c \neq 1$

Running Time Calculation

- Sequential and If-Then-Else Blocks

```
for i = 1 to n do {
    A[i] = 0;
}
for i = 1 to n do {
    for  j = 1 to n do {
      A[i]++;
    }
}
```

$$T(n) = O(n) + O(n^2) = O(n^2)$$

```
if (cond)
   S1
else
   S2
```

$$T(n) = max(T_{s1}(n), (T_{s2}(n))$$

Divide and Conquer

- We solve a problem recursively by applying three steps at each level of the recursion.
  - Divide the problem into a number of subproblems that are smaller instances of the same problem
  - Conquer the subproblem by solving them recursively
    - If the problem sizes are small enough (i.e. we have gotten down to the base case), solve the subproblem in a straightforward manner
  - Combine the solutions to the subproblems into the solution for the original problem

Recurrences

- A recurrence is an equation or inequality that describes a function in terms of its value on smaller inputs.
- Recurrences give us a natural way to characterize the running times of divide-and-conquer algorithms.
- Thus, they go hand in hand with the divide-and-conquer paradigm.

Recurrences

- The worst-case running time T(n) of the MERGE-SORT procedure is

Recurrences

- The worst-case running time T(n) of the MERGE-SORT procedure is
  - $T(1) = 1$
  - $T(n) = 2T\left(\frac{n}{2}\right) + n$ if n > 1

Recurrences

- The worst-case running time T(n) of the MERGE-SORT procedure is
  - $T(1) = 1$
  - $T(n) = 2T\left(\frac{n}{2}\right) + n$ if n > 1
- If a recursive algorithm divide subproblems into unequal sizes, such as a 2/3-to-1/3 split and combine steps takes linear time, such an algorithm give rise to the recurrence

Recurrences

- The worst-case running time T(n) of the MERGE-SORT procedure is
  - $T(1) = 1$
  - $T(n) = 2T\left(\frac{n}{2}\right) + n$ if n > 1
- If a recursive algorithm divide subproblems into unequal sizes, such as a 2/3-to-1/3 split and combine steps takes linear time, such an algorithm give rise to the recurrence
  - $T(n) = T\left(\frac{2n}{3}\right) + T\left(\frac{n}{3}\right) + \Theta(n)$

Recurrences

- If a recursive version of linear search linear search algorithm creates just one problem containing only one element fewer than the original problem, each recursive call would take constant time plus the time for the recursive calls it makes.
- Such an algorithm yields the recurrence
  - $T(n) = T(n-1) + \Theta(1)$

The methods for Solving Recurrences

- Brute-force method
- Substitution method
- Recursion tree method
- Master method

Inequality Recurrences

- $T(n) \leq 2T\left(\frac{n}{2}\right) + \Theta(n)$
  - Because such a recurrence states only an upper bound on T(n), we couch its solution using O-notation rather than Θ-notation
- $T(n) \geq 2T\left(\frac{n}{2}\right) + \Theta(n)$
  - Because the recurrence gives only a lower bound on T(n), we use Ω-notation in its solution

Technicalities in Recurrences

- In practice, we neglect certain technical details
  - If we call MERGE-SORT on n elements, when n is odd, we end up with subproblems of size $\left\lceil \frac{n}{2} \right\rceil$ and $\left\lfloor \frac{n}{2} \right\rfloor$.
  - Technically, the recurrence describing the worst-case running time of MERGE-SORT is
    - $T(1) = 1$
      $T(n) = T\left(\left\lceil \frac{n}{2} \right\rceil\right) + T\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + n$ for n > 1
  - For convenience, we omit floors, ceilings and statements of the boundary conditions of recurrences and assume that T(n) is constant for small n.

Brute-force Method

$T(1) = 1$

$T(n) = 2T\left(\frac{n}{2}\right) + n$ for n > 1

Let $n = 2^k$. Then,

$$
\begin{aligned}
T(n) &= 2T\left(\frac{n}{2}\right) + n = 2\left(2T\left(\frac{n}{2^2}\right) + \frac{n}{2}\right) + n \\
&= 2^2\, T\left(\frac{n}{2^2}\right) + 2n = 2^2\left(2T\left(\frac{n}{2^3}\right) + \frac{n}{2^2}\right) + 2n \\
&= 2^3 T\left(\frac{n}{2^3}\right) + 3n \\
&\dots \\
&= 2^k T\left(\frac{n}{2^k}\right) + kn \\
&= n + n \lg n
\end{aligned}
$$

When $\frac{n}{2^k} = 1$,
we have n = 2^k and k = lg n

Another Brute-force Method

$T(1) = 1$

$T(n) = 2T\left(\frac{n}{2}\right) + n$ for n > 1

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

$$\frac{T(n/2)}{n/2} = \frac{T(n/4)}{n/4} + 1$$

$$\frac{T(n/4)}{n/4} = \frac{T(n/8)}{n/8} + 1$$

… …

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

Another Brute-force Method

$T(1) = 1$

$T(n) = 2T\left(\frac{n}{2}\right) + n$ for n > 1

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

$$\frac{T(n/2)}{n/2} = \frac{T(n/4)}{n/4} + 1$$

$$\frac{T(n/4)}{n/4} = \frac{T(n/8)}{n/8} + 1$$

… …

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

Another Brute-force Method

$T(1) = 1$

$T(n) = 2T\left(\frac{n}{2}\right) + n$ for n > 1

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

$\frac{T(n/2)}{n/2} = \frac{T(n/4)}{n/4} + 1$

$\frac{T(n/4)}{n/4} = \frac{T(n/8)}{n/8} + 1$

…… ……

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

Another Brute-force Method

$$T(1) = 1$$

$$T(n) = 2T\left(\frac{n}{2}\right) + n \text{ for n > 1}$$

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

$$\frac{T(n/2)}{n/2} = \frac{T(n/4)}{n/4} + 1$$

$$\frac{T(n/4)}{n/4} = \frac{T(n/8)}{n/8} + 1$$

… …

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

Another Brute-force Method

$T(1) = 1$

$T(n) = 2T\left(\frac{n}{2}\right) + n$ for n > 1

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

$\frac{T(n/2)}{n/2} = \frac{T(n/4)}{n/4} + 1$ $\xrightarrow{\text{Divide by n}}$ $\frac{T(n)}{n} = \frac{T(1)}{1} + \lg n$

$\frac{T(n/4)}{n/4} = \frac{T(n/8)}{n/8} + 1$ Thus, $T(n) = n + n\log n = \Theta(n \lg n)$

……

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

Substitution Method

- Comprises two steps:
  - Guess the form of the solution
  - Use mathematical induction to find the constants and show that the solution works
- We can use the substitution method to establish either upper or lower bounds on a recurrence.

Substitution Method

$T(1) = 1$

$T(n) = 2T\left(\frac{n}{2}\right) + n$ for n > 1

- We guess that the solution is T(n) = O(n lg n).
- The substitution method requires us to prove that $T(n) \leq c\, n \lg n$ for an appropriate choice of the constant c > 0.
  - Base case: Examine later
  - Induction hypothesis: $T(m) \leq c\, m \lg m$ holds for all positive for $m < n$

Substitution Method

$T(1) = 1$

$T(n) = 2T\left(\frac{n}{2}\right) + n$ for n > 1

- We guess that the solution is T(n) = O(n lg n).
- The substitution method requires us to prove that $T(n) \leq c\, n \lg n$ for an appropriate choice of the constant c > 0.
  - Base case: Examine later
  - Induction hypothesis: $T(m) \leq c\, m \lg m$ holds for all positive for $m < n$
  - Induction step:

$$
\begin{aligned}
T(n) &= 2T(n/2) + n \\
&\leq 2c\left(\frac{n}{2}\right)\lg\left(\frac{n}{2}\right) + n \\
&= c\, n \lg\left(\frac{n}{2}\right) + n \\
&= c\, n \lg n - c\, n \lg 2 + n \\
&= c\, n \lg n - c\, n + n \\
&\leq c\, n\, lgn \quad \text{(for } c \geq 1\text{)}
\end{aligned}
$$

$\lg\left(\frac{\beta}{\alpha}\right) = \lg(\beta) - \lg(\alpha)$

Substitution Method

- Base case revisited
  - $T(1) \leq c\ 1\ \mathrm{lg} 1 = 0$ wrong!!
  - The base case of our induction proof fails to hold
- What should we do?
  - We can overcome this obstacle in proving an inductive hypothesis for a specific boundary condition with only a little more effort.
  - We are interested in asymptotic behavior.
  - Remove the difficult boundary condition from induction proof.
  - We do so by first observing that for n > 3, the recurrence does not depend directly on T(1).
  - Thus, we can replace T(1) by T(2) and T(3) as the base cases in the induction proof.
  - From the recurrence, we have $T(2) \leq c\ 2\ \lg 2$ and $T(3) \leq c\ 3\ \lg 3$.
  - Any choice of $c \geq 2$ suffices for the base cases of n=2 and n=3.
  - $T(n) \leq c\, n \log n$ for $c \geq 2$ and $n \geq 2$

Making a Good Guess

- If a recurrence is similar to one you have seen before, guessing a similar solution is reasonable
  - $T(n) = 2T(n/2+17) + n$
  - When n is large, the difference between n/2 and n/2+17 is not that large
  - Consequently, guess $T(n) = O(n \lg n)$ and prove by substitution method

Making a Good Guess

- Prove loose upper and lower bounds on the recurrence and then reduce the range of uncertainty
  - We might start with T(n) = $\Omega(n)$
  - We can prove T(n) = $O(n^2)$
  - Then, we can gradually lower the upper bound and raise the lower bound until we converge on the correct, asymptotically tight solution of T(n) = $\Theta(n \lg n)$

Substitution Method

- Sometimes, your correct guess still may fail to work out in the induction.
- The problem is frequently turns out to be that the inductive assumption is not strong enough to prove the detailed bound.
- If you revise the guess by subtracting a lower-order term when you hit such a snag, it may become okay.
  - $T(n) = T(n/2) + T(n/2) + 1$, show $T(n) \leq c\, n$
  - Base case: $T(1) = 1 \leq c$
  - Induction hypothesis: $T(m) \leq cm$ for $m < n$
  - Induction step:

Substitution Method

- Sometimes, your correct guess still may fail to work out in the induction.
- The problem is frequently turns out to be that the inductive assumption is not strong enough to prove the detailed bound.
- If you revise the guess by subtracting a lower-order term when you hit such a snag, it may become okay.
  - $T(n) = T(n/2) + T(n/2) + 1$, show $T(n) \leq c\, n$
  - Base case: $T(1) = 1 \leq c$
  - Induction hypothesis: $T(m) \ \leq \ cm$ for $m < n$
  - Induction step:
    - $T(n) = 2\, T(n/2) + 1 \leq cn + 1$ which does not imply $T(n) \leq cn$ for any choice of c

Substitution Method

- Sometimes, your correct guess still may fail to work out in the induction.
- The problem is frequently turns out to be that the inductive assumption is not strong enough to prove the detailed bound.
- If you revise the guess by subtracting a lower-order term when you hit such a snag, it may become okay.
  - $T(n) = T(n/2) + T(n/2) + 1$, show $T(n) \leq c\,n$
  - Base case: $T(1) = 1 \leq c$
  - Induction hypothesis: $T(m) \leq cm$ for $m < n$
  - Induction step:
    - $T(n) = 2\,T(n/2) + 1 \leq cn + 1$ which does not imply $T(n) \leq cn$ for any choice of c
    - Our guess is nearly right and we are off only by the constant 1, a lower-order term
    - Make a stronger induction hypothesis by subtracting a lower-order term from our previous guess: $T(n) \leq cn - b$
    - $T(n) \leq cn - 2b + 1 \leq cn - b$ as long as $b \geq 1$

Substitution Method

- Avoiding Pitfalls
  - $T(n) = 2T(n/2) + n$ and prove $T(n) \leq c\, n$
  - $T(n) \leq 2c(n/2) + n = cn + n$
    - Thus, $T(n) \leq cn$

Substitution Method

- Avoiding Pitfalls
  - $T(n) = 2T(n/2) + n$ and prove $T(n) \le c\,n$
  - $T(n) \le 2c(n/2) + n = cn + n$
    - Thus, $T(n) \le cn$
    - Wrong!! We should prove the exact form of the induction hypothesis, that is $T(n) \le cn$

Substitution Method

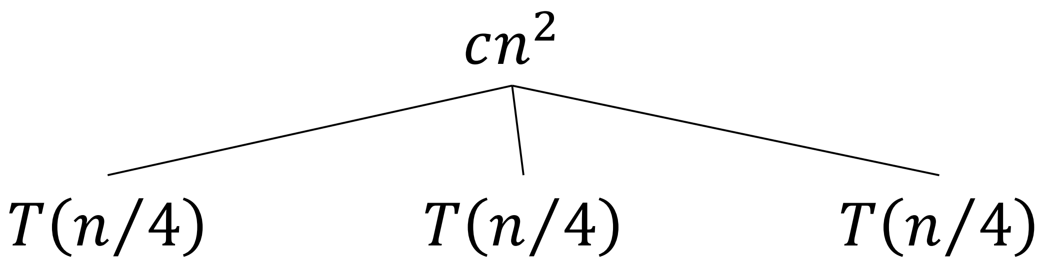
- Changing variables
  - $T(n) = 2T\left(n^{1/2}\right) + \lg n$
  - Rename $m = \lg n$ yields $T(2^m) = 2T\left(2^{m/2}\right) + m$.
  - Then, by renaming $S(m) = T(2^m)$, we get

  $S(m) = 2S(m/2) + m$.
  - Thus, we obtain $S(m) = O(m \log m)$.
  - By chainging back from S(m) to T(n), we obtain

    $T(n) = T(2^m) = S(m) = O(m \log m) = O(\lg n \lg (\lg n))$.

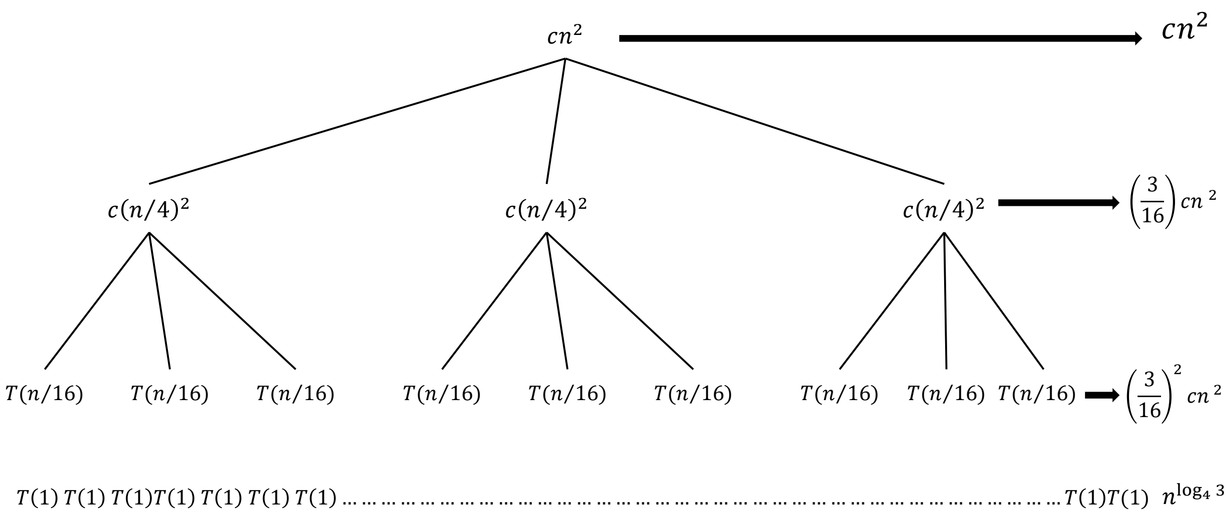
Recursion-tree Method

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



Recursion-tree Method

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



$$T(n) = \left((3/16)^{log_4 n} - 1\right)/\left((3/16) - 1\right)cn^2 + \Theta\left(n^{log_4 3}\right) = O(n^2)$$

Recursion-tree Method

- Remember

$$a + ar + ar^2 + \cdots + ar^{n-1} = \frac{a(r^n - 1)}{r - 1}$$

$$a + ar + ar^2 + \cdots + r^{n-1} + \cdots = \frac{a}{r - 1}$$

- $T(n) = cn^2 + \frac{3}{16}cn^2 + \left(\frac{3}{16}\right)^2 cn^2 + \cdots + \left(\frac{3}{16}\right)^{\log_4 n-1} cn^2 + \theta\left(n^{\log_4 3}\right)$

$$= \sum_{i=0}^{\log_4 n-1} \left(\frac{3}{16}\right)^i cn^2 + \theta\left(n^{\log_4 3}\right) = \frac{\left(\frac{3}{16}\right)^{\log_4 n} - 1}{\left(\frac{3}{16}\right) - 1} cn^2 + \theta\left(n^{\log_4 3}\right)$$

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

$$= \frac{1}{1-\left(\frac{3}{16}\right)} cn^2 + \theta\left(n^{\log_4 3}\right) = \frac{16}{13} cn^2 + \theta\left(n^{\log_4 3}\right) = O(n^2)$$

Master Method

- It is a cookbook method for solving recurrences of the form $T(n) = aT(n/b) + f(n)$ where a $\geq$ 1 and b > 1 are constants and f(n) is an asymptotically positive function.
- In each of three case, we compare the function f(n) with the function $n^{\log_b a}$.
- The larger of two functions determine the solution to the recurrence
  - (1) If $n^{\log_b a}$ is the larger, $T(n) = \Theta(n^{\log_b a})$
  - (2) If the two functions are the same size,
  $$T(n) = \Theta(n^{\log_b a} \lg n)$$
  - (3) If f(n) is the larger, $T(n) = \Theta(f(n))$

Master Method

- Theorem 4.1 (Master theorem)
  - Let $a \geq 1$ and $b > 1$ be constants
  - Let $f(n)$ be a function
  - 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 some 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 some constant $\varepsilon > 0$, and if $a\,f(n/b) \leq c\,f(n)$ for some constant $c < 1$ and all sufficiently large n, then $T(n) = \Theta(f(n))$

Master Method

$$T(n) = \ a\, T(n/b) + f(n)$$

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

- Consider $T(n) = 9T\left(\frac{n}{3}\right) + n$.
  - Since a = 9, b = 3,
    - f(n) = n
    - $n^{\log_b a} = n^{\log_3 9} = \Theta(n^2)$
  - Because f(n) = $O(n^{\log_3 9 - \varepsilon})$ with $\varepsilon = 1$, we can apply case (1) of the master theorem
  - Thus, $T(n) = \Theta(n^{\log_b a}) = \Theta(n^2)$

Master Method

$$T(n) = \ a\,T(n/b) + f(n)$$

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

- Consider $T(n) = T\left(\frac{2n}{3}\right) + 1$.
  - Since a = 1, b = 3/2,
    - f(n) = 1
    - $n^{\log_b a} = n^{\log_{3/2} 1} = n^0 = 1$
  - Because f(n) = $O(n^{\log_b a}) = \Theta(1)$, we can apply case (2) of the master theorem
  - Thus, $T(n) = \Theta(n^{\log_b a} \lg n) = \Theta(\lg n)$

Master Method

$$T(n) = \; a\,T(n/b) + f(n)$$

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

- Consider $T(n) = 3T\left(\frac{n}{4}\right) + n\lg n$.
  - Since a = 3, b = 4,
    - f(n) = n lg n
    - $n^{\log_b a} \; = n^{\log_4 3} = O(n^{0.793})$
  - Because f(n) =$\Omega(n^{\,\log_4 3+\varepsilon})$ with $\varepsilon \approx \; 0.2$, we can apply case (3) of the master theorem
  - Thus, $T(n) = \Theta\big(f(n)\big) = \Theta(n\lg n)$

Chapter 4:
Divide and Conquer

Binary Search

Static Searching

- Given an integer X and an array A, return the position of X in A or an indication that it is not present.
- If X occurs more than once, return any occurrence.
- If the array is not sorted, use a sequential search
  - Unsuccessful search: $O(N)$; every item is examined
  - Successful search:
    - Worst case: $O(N)$; every item is examined
    - Average case: $O(N)$; half the items are examined
- Can we do better if we know the array is sorted?

Binary Search

- Look in the middle
  - Case 1: If X is less than the item in the middle, look in the subarray to the left of the middle.
  - Case 2: If X is greater than the item in the middle, look in the subarray to the right of the middle.
  - Case 3: If X is equal to the item in the middle, we have a match.

Binary Search Algorithm

```
BINARY-SEARCH(A, low, high, X)
  if low > high
    return NOT_FOUND
  if low == high
    if A[low] == X
      return low
    else
      return NOT_FOUND
  else
    mid = ( low + high ) / 2
    if A[mid] == X
      return mid
    if A[mid] > X
      return BINARY-SEARCH(A, low, mid-1, X)
    else
      return BINARY-SEARCH(A, mid+1, high, X)
```

Worst Case Time Complexity

$T(1) = 1$

$T(n) = T\left(\frac{n}{2}\right) + 1$ for n > 1

Let $n = 2^k$. Then,

$$T(n) = T\left(\frac{n}{2}\right) + 1$$
$$= T\left(\frac{n}{2^2}\right) + 2$$
$$= T\left(\frac{n}{2^3}\right) + 3$$
$$\dots$$
$$= T\left(\frac{n}{2^k}\right) + k$$
$$= \Theta(\lg n)$$

When $\frac{n}{2^k} = 1$,
we have n = 2^k and k = lg n