Algorithms

Analysis of Algorithms

Ming-Hsien Tsai

Academia Sinica
National Taiwan University

Based on: U. Manber. Introduction to Algorithms - A Creative Approach. Addison-Wesley, 1989.
Also based on slides made by Professor Yih-Kuen Tsay.

Introduction

- The purpose of ***algorithm analysis*** is to predict the behavior (running time, space requirement, etc.) of an algorithm without implementing it on a specific computer

- As the exact behavior of an algorithm is hard to predict, the analysis is usually an ***approximation***:
  - Relative to the input size (usually denoted by n): input possibilities too enormous to elaborate
  - Asymptotic: should care more about larger inputs
  - Worst-Case: easier to do, often representative

- Such an approximation is usually good enough for ***comparing*** different algorithms for the same problem

Complexity

- Theoretically, "***complexity*** of an algorithm" is a more precise term for "approximate behavior of an algorithm"

- Two most important measures of complexity:
  - ***Time Complexity***: an upper bound on the number of steps that the algorithm performs
  - ***Space Complexity***: an upper bound on the amount of temporary storage required for running the algorithm (excluding the input, the output, and the program itself)

- We will focus on time complexity

Comparing Running Time

- How do we compare the following running times?

1. $100n$

2. $2n^2 + 50$

3. $100n^{1.8}$

Comparing Running Time

- How do we compare the following running times?

	n = 1
1. $100n$	100
2. $2n^2 + 50$	52
3. $100n^{1.8}$	100

Comparing Running Time

- How do we compare the following running times?

	n = 1	**n = 2**
1. $100n$	100	200
2. $2n^2 + 50$	52	58
3. $100n^{1.8}$	100	348

Comparing Running Time

- How do we compare the following running times?

	n = 1	**n = 2**	**n = 100**
1. $100n$	100	200	10,000
2. $2n^2 + 50$	52	58	20,050
3. $100n^{1.8}$	100	348	398,107

Comparing Running Time

- How do we compare the following running times?

	n = 1	**n = 2**	**n = 100**	**n = 320,000,000**
1. $100n$	100	200	10,000	32,000,000,000
2. $2n^2 + 50$	52	58	20,050	204,800,000,000,000,050
3. $100n^{1.8}$	100	348	398,107	203,830,871,323,390,784

Comparing Running Time (cont'd)

- We will study an approach (the O notation) that allows us to ignore constant factors and concentrate on the behavior as n goes to infinity

- For most algorithms, the constants in the expressions of their running times tend to be small

1. $100n \quad \Rightarrow n$

2. $2n^2 + 50 \quad \Rightarrow n^2$

3. $100n^{1.8} \quad \Rightarrow n^{1.8}$

The O Notation

- A function g(n) is O(f(n)) for another function f(n) if there exist constants c and N such that, for all $n \geq N$, $g(n) \leq cf(n)$

- The O notation allows us to ignore constants conveniently

- The function g(n) may be substantially less than cf(n); the O notation bounds it only from above

The O Notation (cont'd)

- A function g(n) is O(f(n)) for another function f(n) if there exist constants c and N such that, for all $n \geq N$, $g(n) \leq cf(n)$

$5n^2 + 15 = O(n^2)$

(for all $n \geq ?$, $5n^2 + 15 \leq ?n^2$)

$5n^2 + 15 = O(n^3)$

$(n + 1)^2 = n^2 + O(n)$

$T(n) = 3n^2 + O(n)$

The O Notation (cont'd)

The O Notation (cont'd)

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

The O Notation (cont'd)

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

$O(n^2) = O(n)$?

The O Notation (cont'd)

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

$O(n^2) = O(n)$?

$n = O(n^2)$

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

$n = n^2$?

The O Notation (cont'd)

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

$O(n^2) = O(n)$?

$n = O(n^2)$

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

$n = n^2$?

The equality is one-way

You may write

$5n^2 + 15 = O(n^2)$

as

$5n^2 + 15 \leq O(n^2)$

or

$5n^2 + 15 \in O(n^2)$

The O Notation (cont'd)

- No need to specify the base of a logarithm
  - $\log_2 n = \log_{10} n / \log_{10} 2 = (1 / \log_{10} 2) \log_{10} n$
  - For example, we can just write O(log n)
- O(1) denotes a constant

Properties of O

- We can add and multiply with O
- However, we cannot subtract or divide with O

Theorem (3.2)

If $f(n) = O(s(n))$ and $g(n) = O(r(n))$, then

- $f(n) + g(n) = O(s(n) + r(n))$, and
- $f(n) \cdot g(n) = O(s(n) \cdot r(n))$

Properties of O

- We can add and multiply with O
- However, we cannot subtract or divide with O

Theorem (3.2)

If $f(n) = O(s(n))$ and $g(n) = O(r(n))$, then

- $f(n) + g(n) = O(s(n) + r(n))$, and
- $f(n) \cdot g(n) = O(s(n) \cdot r(n))$

$2n = O(n)$, $n = O(n)$, and $2n - n = n \neq O(n - n)$

$n^2 = O(n^2)$, $n = O(n^2)$, and $n^2 / n = n \neq O(n^2 / n^2)$

Polynomial vs. Exponential

- A function f(n) is ***monotonically growing*** if $n_1 \geq n_2$ implies that $f(n_1) \geq f(n_2)$

- An exponential function grows at least as fast as a polynomial function does

Theorem (3.1)

For all constants c > 0 and a > 1, and for all monotonically growing functions f(n), $(f(n))^c = O(a^{f(n)})$

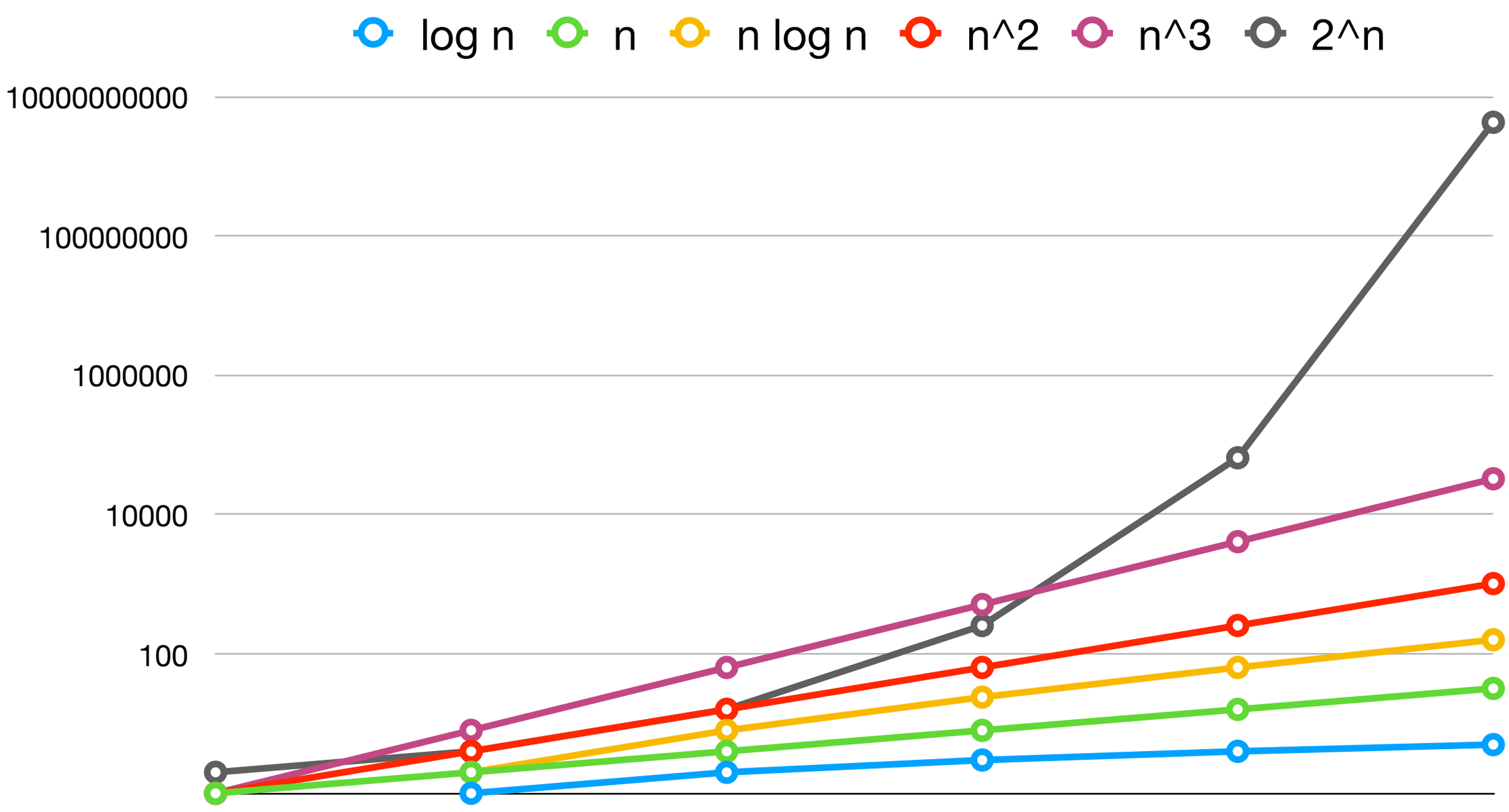
$$\text{Take } n \text{ as } f(n),\ n^c = O(a^n)$$
$$\text{Take } \log_a n \text{ as } f(n),\ (\log_a n)^c = O(a^{\log_a n}) = O(n)$$

Speed of Growth

log n	n	n log n	n^2	n^3	2^n
0	1	0	1	1	2
1	2	2	4	8	4
2	4	8	16	64	16
3	8	24	64	512	256
4	16	64	256	4,096	65,536
5	32	160	1,024	32,768	4,294,967,296

Source: [E. Horowitz et al. 1998]

Speed of Growth (cont'd)



Speed of Growth (cont'd)

	$time_1$	$time_2$	$time_3$	$time_4$
running times	**1000 steps/sec**	**2000 steps/sec**	**4000 steps/sec**	**8000 steps/sec**
$\log_2 n$	0.010	0.005	0.003	0.001
n	1	0.5	0.25	0.125
$n \log_2 n$	10	5	2.5	1.25
$n^{1.5}$	32	16	8	4
n^2	1000	500	250	125
n^3	1000000	500000	250000	125000
1.1^n	10^{39}	10^{39}	10^{38}	10^{38}

Running times (int seconds) under different assumptions (n = 1000)

Source: [Manber 1989]

O, o, Ω, and Θ

- Let T(n) be the number of steps required to solve a given problem for input size n

- We say that T(n) = Ω(g(n)) or the problem has a lower bound of Ω(g(n)) if there exist constants c and N such that, for all $n \geq N$, $T(n) \geq cg(n)$

- If a certain function f(n) satisfies both f(n) = O(g(n)) and f(n) = Ω(g(n)), then we say that f(n) = Θ(g(n))

$$5n^2 + 15 = O(n^2)$$

$$5n^2 + 15 = \Omega(n^2)?$$

O, o, Ω, and Θ (cont'd)

- We say that f(n) = o(g(n)) if $\lim_{n \to \infty} \frac{f(n)}{g(n)} = 0$
- Equivalently, f(n) = o(g(n)) if for every positive constant c, there exists a constant N such that for all n ≥ N, |f(n)| ≤ cg(n)

$2n = o(n^2)$

$1 / n = o(1)$

Polynomial vs. Exponential (cont'd)

- An exponential function grows faster than a polynomial function does

Theorem (3.3)

For all constants c > 0 and a > 1, and for all monotonically growing functions f(n), $(f(n))^c = o(a^{f(n)})$

Take n as $f(n)$, $n^c = o(a^n)$

Take $\log_a n$ as $f(n)$, $(\log_a n)^c = o(a^{\log_a n}) = o(n)$

Complexity Analysis

```
// Sort the given run of array A[] using array B[] as a source.
// iBegin is inclusive; iEnd is exclusive (A[iEnd] is not in the set).
TopDownSplitMerge(B[], iBegin, iEnd, A[])
{
    if (iEnd - iBegin < 2)                      // if run size == 1
        return;                                 //   consider it sorted
    // split the run longer than 1 item into halves
    iMiddle = (iEnd + iBegin) / 2;              // iMiddle = mid point
    // recursively sort both runs from array A[] into B[]
    TopDownSplitMerge(A, iBegin,  iMiddle, B);  // sort the left  run
    TopDownSplitMerge(A, iMiddle,    iEnd, B);  // sort the right run
    // merge the resulting runs from array B[] into A[]
    TopDownMerge(B, iBegin, iMiddle, iEnd, A);
}
```

Source: https://en.wikipedia.org/wiki/Merge_sort

Complexity Analysis

```
// Sort the given run of array A[] using array B[] as a source.
// iBegin is inclusive; iEnd is exclusive (A[iEnd] is not in the set).
TopDownSplitMerge(B[], iBegin, iEnd, A[])
{
    if (iEnd - iBegin < 2)                       // if run size == 1
```

What is the worst-case complexity?

```
    TopDownSplitMerge(A, iBegin,  iMiddle, B);  // sort the left  run
    TopDownSplitMerge(A, iMiddle,    iEnd, B);  // sort the right run
    // merge the resulting runs from array B[] into A[]
    TopDownMerge(B, iBegin, iMiddle, iEnd, A);
}
```

Source: https://en.wikipedia.org/wiki/Merge_sort

Sums

- Techniques for summing expressions are essential for complexity analysis

- For example, given that we know

$$S_0(n) = \sum_{i=1}^{n} 1 = n$$

and

$$S_1(n) = \sum_{i=1}^{n} i = 1 + 2 + 3 + \cdots + n = \frac{n(n+1)}{2},$$

we want to compute the sum

$$S_2(n) = \sum_{i=1}^{n} i^2 = 1^2 + 2^2 + 3^2 + \cdots + n^2$$

Sums (cont'd)

From

$$(i + 1)^3 = i^3 + 3i^2 + 3i + 1,$$

we have

$$(i + 1)^3 - i^3 = 3i^2 + 3i + 1$$

$$2^3 - 1^3 = 3 \times 1^2 + 3 \times 1 + 1$$
$$3^3 - 2^3 = 3 \times 2^2 + 3 \times 2 + 1$$
$$4^3 - 3^3 = 3 \times 3^2 + 3 \times 3 + 1$$
$$\ldots \ldots \ldots$$
$$(n + 1)^3 - n^3 = 3 \times n^2 + 3 \times n + 1$$

Sums (cont'd)

$$2^3 - 1^3 = 3 \times 1^2 + 3 \times 1 + 1$$

$$3^3 - 2^3 = 3 \times 2^2 + 3 \times 2 + 1$$

$$4^3 - 3^3 = 3 \times 3^2 + 3 \times 3 + 1$$

$$\ldots \quad \ldots \quad \ldots$$

$$(n + 1)^3 - n^3 = 3 \times n^2 + 3 \times n + 1$$

Sums (cont'd)

$$S_2(n)$$

$$2^3 - 1^3 = 3 \times 1^2 + 3 \times 1 + 1$$

$$3^3 - 2^3 = 3 \times 2^2 + 3 \times 2 + 1$$

$$4^3 - 3^3 = 3 \times 3^2 + 3 \times 3 + 1$$

$$\ldots \quad \ldots \quad \ldots$$

$$(n+1)^3 - n^3 = 3 \times n^2 + 3 \times n + 1$$

Sums (cont'd)

$$
\begin{array}{rcl}
 & & \qquad\qquad S_2(n) \qquad\quad S_1(n) \\
2^3 - 1^3 &=& 3 \times \boxed{1^2} + 3 \times \boxed{1} + 1 \\
3^3 - 2^3 &=& 3 \times \boxed{2^2} + 3 \times \boxed{2} + 1 \\
4^3 - 3^3 &=& 3 \times \boxed{3^2} + 3 \times \boxed{3} + 1 \\
 & & \cdots \quad \cdots \quad \cdots \\
(n+1)^3 - n^3 &=& 3 \times \boxed{n^2} + 3 \times \boxed{n} + 1
\end{array}
$$

Sums (cont'd)

$$
\begin{array}{rcl}
 & & \qquad\qquad S_2(n) \qquad\quad S_1(n) \quad S_0(n) \\
2^3 - 1^3 &=& 3 \times \boxed{1^2} + 3 \times \boxed{1} + \boxed{1} \\
3^3 - 2^3 &=& 3 \times \boxed{2^2} + 3 \times \boxed{2} + \boxed{1} \\
4^3 - 3^3 &=& 3 \times \boxed{3^2} + 3 \times \boxed{3} + \boxed{1} \\
 & & \cdots \quad \cdots \quad \cdots \\
(n+1)^3 - n^3 &=& 3 \times \boxed{n^2} + 3 \times \boxed{n} + \boxed{1}
\end{array}
$$

Sums (cont'd)

$$
\begin{array}{rcl}
 & & \qquad\qquad S_2(n) \qquad\quad S_1(n) \quad S_0(n) \\
2^3 - 1^3 &=& 3 \times \boxed{1^2} + 3 \times \boxed{1} + \boxed{1} \\
3^3 - 2^3 &=& 3 \times \boxed{2^2} + 3 \times \boxed{2} + \boxed{1} \\
4^3 - 3^3 &=& 3 \times \boxed{3^2} + 3 \times \boxed{3} + \boxed{1} \\
 & & \cdots \quad \cdots \quad \cdots \\
(n+1)^3 - n^3 &=& 3 \times \boxed{n^2} + 3 \times \boxed{n} + \boxed{1}
\end{array}
$$

Sums (cont'd)

$$
\begin{array}{rcl}
 & & S_2(n) \quad S_1(n) \quad S_0(n) \\
2^3 - 1^3 & = & 3 \times \boxed{1^2} + 3 \times \boxed{1} + \boxed{1} \\
3^3 - 2^3 & = & 3 \times \boxed{2^2} + 3 \times \boxed{2} + \boxed{1} \\
4^3 - 3^3 & = & 3 \times \boxed{3^2} + 3 \times \boxed{3} + \boxed{1} \\
 & \ldots & \ldots \quad \ldots \\
(n+1)^3 - n^3 & = & 3 \times \boxed{n^2} + 3 \times \boxed{n} + \boxed{1}
\end{array}
$$

Sums (cont'd)

$$
\begin{array}{rcl}
 & & \overbrace{}^{S_2(n)}\ \overbrace{}^{S_1(n)}\ \overbrace{}^{S_0(n)} \\
2^3 - 1^3 &=& 3 \times 1^2 + 3 \times 1 + 1 \\
3^3 - 2^3 &=& 3 \times 2^2 + 3 \times 2 + 1 \\
4^3 - 3^3 &=& 3 \times 3^2 + 3 \times 3 + 1 \\
 & \cdots\ \cdots\ \cdots & \\
(n+1)^3 - n^3 &=& 3 \times n^2 + 3 \times n + 1
\end{array}
$$

Sums (cont'd)

$$
\begin{array}{rcl}
 & & S_2(n) \quad S_1(n) \quad S_0(n) \\
2^3 - 1^3 & = & 3 \times \boxed{1^2} + 3 \times \boxed{1} + \boxed{1} \\
3^3 - 2^3 & = & 3 \times \boxed{2^2} + 3 \times \boxed{2} + \boxed{1} \\
4^3 - 3^3 & = & 3 \times \boxed{3^2} + 3 \times \boxed{3} + \boxed{1} \\
 & \cdots & \cdots \quad \cdots \quad \cdots \\
(n+1)^3 - n^3 & = & 3 \times \boxed{n^2} + 3 \times \boxed{n} + \boxed{1}
\end{array}
$$

Sums (cont'd)

$$
\begin{array}{rcl}
 & & S_2(n) \quad S_1(n) \quad S_0(n) \\
2^3 - 1^3 & = & 3 \times 1^2 + 3 \times 1 + 1 \\
3^3 - 2^3 & = & 3 \times 2^2 + 3 \times 2 + 1 \\
4^3 - 3^3 & = & 3 \times 3^2 + 3 \times 3 + 1 \\
 & \ldots\ldots\ldots & \\
(n+1)^3 - n^3 & = & 3 \times n^2 + 3 \times n + 1
\end{array}
$$

$$(n+1)^3 - 1 = 3 \times S_2(n) + 3 \times S_1(n) + S_0(n)$$

Sums (cont'd)

$$S_2(n) \quad S_1(n) \quad S_0(n)$$

$$2^3 - 1^3 = 3 \times 1^2 + 3 \times 1 + 1$$
$$3^3 - 2^3 = 3 \times 2^2 + 3 \times 2 + 1$$
$$4^3 - 3^3 = 3 \times 3^2 + 3 \times 3 + 1$$
$$\ldots \quad \ldots \quad \ldots$$
$$(n+1)^3 - n^3 = 3 \times n^2 + 3 \times n + 1$$

$$(n+1)^3 - 1 = 3 \times S_2(n) + 3 \times S_1(n) + S_0(n)$$

$$S_2(n) = ((n+1)^3 - 1 - 3S_1(n) - S_0(n)) / 3$$
$$= n(n+1)(2n+1) / 6$$

Sums (cont'd)

$$1^3 + 2^3 + \cdots + (n-1)^3 + n^3 + (n+1)^3$$

$$(0+1)^3 + (1+1)^3 + (2+1)^3 + \cdots + (n-1+1)^3 + (n+1)^3$$

- $S_2(n)$ can be derived by ***shifting and canceling*** terms of two sums
- This generalizes to $S_k(n)$, for $k > 2$
- Similar shifting and canceling techniques apply to other kinds of sums.

Exercise: compute the sum $F(n) = 1 + 2 + 4 + \cdots + 2^n$

Recurrence Relations

- A ***recurrence relation*** is a way to define a function by an expression involving the same function

- The Fibonacci numbers, for example, can be defined as follows:

$$\begin{cases} F(1) = 1 \\ F(2) = 1 \\ F(n) = F(n - 2) + F(n - 1) \end{cases}$$

- We need k - 2 steps to compute F(k)

- It is more convenient to have an explicit (or ***closed-form***) expression

- To obtain the explicit expression is called ***solving*** the recurrence relation

Guessing and Proving an Upper Bound

Recurrence relation: $\begin{cases} T(2) = 1 \\ T(2n) \leq 2T(n) + 2n - 1 \end{cases}$ (n is a power of 2)

Guess: $T(n) = O(n)$

Proof:

Base case: $T(2) \leq 2$

Inductive step: $T(2n) \leq 2T(n) + 2n - 1$

$\leq 2\ O(n) + 2n - 1$

$= O(n)$

Guessing and Proving an Upper Bound

Recurrence relation: $\begin{cases} T(2) = 1 \\ T(2n) \leq 2T(n) + 2n - 1 \end{cases}$ (n is a power of 2)

Guess: T(n) = O(n)

Proof:

Base case: $T(2) \leq 2$

Inductive step: $T(2n) \leq 2T(n) + 2n - 1$

$\leq 2\ O(n) + 2n - 1$

$= O(n)$

Guessing and Proving an Upper Bound

Recurrence relation: $\begin{cases} T(2) = 1 \\ T(2n) \leq 2T(n) + 2n - 1 \end{cases}$ (n is a power of 2)

Guess: T(n) = O(n)

Proof:

Base case: $T(2) \leq 2$

Inductive step: $T(2n) \leq 2T(n) + 2n - 1$

$\leq 2\ O(n) + 2n - 1$

$= O(n)$

Actually T(n) is not O(n)

Guessing and Proving an Upper Bound (cont'd)

Recurrence relation: $\begin{cases} T(2) = 1 \\ T(2n) \leq 2T(n) + 2n - 1 \end{cases}$ (n is a power of 2)

Guess: $T(n) = O(n^2)$

Proof (by showing $T(n) \leq n^2$):

Base case: $T(2) \leq 2^2$

Inductive step: $T(2n) \leq 2T(n) + 2n - 1$

$\leq 2n^2 + 2n - 1$

$< (2n)^2$

Guessing and Proving an Upper Bound (cont'd)

Recurrence relation: $\begin{cases} T(2) = 1 \\ T(2n) \leq 2T(n) + 2n - 1 \end{cases}$ (n is a power of 2)

Guess: $T(n) = O(n^2)$

Proof (by showing $T(n) \leq n^2$):

Base case: $T(2) \leq 2^2$

Inductive step: $T(2n) \leq 2T(n) + 2n - 1$

$\leq 2n^2 + 2n - 1$

$< (2n)^2$

Can we try a number between n and n^2?

Guessing and Proving an Upper Bound (cont'd)

Recurrence relation: $\begin{cases} T(2) = 1 \\ T(2n) \leq 2T(n) + 2n - 1 \end{cases}$ (n is a power of 2)

Guess: $T(n) = O(n \log n)$

Proof (by showing $T(n) \leq n \log n$):

Base case: $T(2) \leq 2 \log 2$

Inductive step: $T(2n) \leq 2T(n) + 2n - 1$

$\leq 2 (n \log n) + 2n - 1$

$= 2n \log n + 2n \log 2 - 1$

$\leq 2n (\log n + \log 2)$

$= 2n \log 2n$

Solving the Fibonacci Relation

- We will study two techniques for solving the Fibonacci relation
  - One uses the characteristic equation
  - The other uses generating functions
- These techniques may be generalized to handle recurrence relations of the following form for a constant k

$$F(n) = b_1 F(n - 1) + b_2 F(n - 2) + \cdots + b_k F(n - k)$$

Using the Characteristic Equation

- F(n) nearly doubles every time and should be an exponential function
- Try $F(n) = a^n$ where a is a (positive) constant
- What is a?
- By $F(n) = F(n-1) + F(n-2)$, we have $a^n = a^{n-1} + a^{n-2}$
- Thus, $a^2 = a + 1$
  - The two solutions are $a_1 = (1 + \sqrt{5}) / 2$ and $a_2 = (1 - \sqrt{5}) / 2$

Using the Characteristic Equation (cont'd)

- Now a_1^n and a_2^n solves the recurrence relation
  - Any linear combination of a_1^n and a_2^n solves the recurrence relation
- So the general solution is $c_1(a_1)^n + c_2(a_2)^n$
  - What are the value of c_1 and c_2?
  - The values of c_1 and c_2 need to satisfy the base cases F(1) and F(2)

Using the Characteristic Equation (cont'd)

- $F(1) = c_1(a_1)^1 + c_2(a_2)^1 = 1$
- $F(2) = c_1(a_1)^2 + c_2(a_2)^2 = 1$
- By solving the above two equations, we have $c_1 = 1 / \sqrt{5}$ and $c_2 = - 1 / \sqrt{5}$
- Therefore the Fibonacci relation is

$$F(n) = \frac{1}{\sqrt{5}}(\frac{1+\sqrt{5}}{2})^n - \frac{1}{\sqrt{5}}(\frac{1-\sqrt{5}}{2})^n$$

Using Generating Functions

- ***Generating functions*** provide a systematic, effective means for representing and manipulating infinite sequences (of numbers)

- We use them here to derive a closed-form representation of the Fibonacci numbers

- Below are two basic generating functions:

gen. func.	power series	generated sequence
$1 / (1 - z)$	$1 + z + z^2 + \cdots + z^n + \cdots$	$1, 1, 1, \ldots, 1, \ldots$
$c / (1 - az)$	$c + caz + ca^2z^2 + \cdots + ca^nz^n + \cdots$	$c, ca, ca^2, \ldots, ca^n, \ldots$

Using Generating Functions (cont'd)

- Let $G(z) = 0 + F_1z + F_2z^2 + F_3z^3 + \cdots + F_nz^n + \cdots$ (a generating function for the Fibonacci numbers; F(n) is written as F_n here)

$$G(z) = F_1z + F_2z^2 + F_3z^3 + \cdots + F_nz^n + F_{n+1}z^{n+1} + \cdots$$
$$zG(z) = F_1z^2 + F_2z^3 + F_3z^4 + \cdots + F_nz^{n+1} + F_{n+1}z^{n+2} + \cdots$$
$$z^2G(z) = F_1z^3 + F_2z^4 + F_3z^5 + \cdots + F_nz^{n+2} + F_{n+1}z^{n+3} + \cdots$$

$$\begin{cases} F(1) = 1 \\ F(2) = 1 \\ F(n) = F(n - 2) + F(n - 1) \end{cases}$$

Using Generating Functions (cont'd)

- Let $G(z) = 0 + F_1z + F_2z^2 + F_3z^3 + \cdots + F_nz^n + \cdots$ (a generating function for the Fibonacci numbers; F(n) is written as F_n here)

$$G(z) = F_1z + F_2z^2 + F_3z^3 + \cdots + F_nz^n + F_{n+1}z^{n+1} + \cdots$$
$$zG(z) = F_1z^2 + F_2z^3 + F_3z^4 + \cdots + F_nz^{n+1} + F_{n+1}z^{n+2} + \cdots$$
$$z^2G(z) = F_1z^3 + F_2z^4 + F_3z^5 + \cdots + F_nz^{n+2} + F_{n+1}z^{n+3} + \cdots$$

$$\begin{cases} F(1) = 1 \\ F(2) = 1 \\ F(n) = F(n-2) + F(n-1) \end{cases}$$

Using Generating Functions (cont'd)

- Let $G(z) = 0 + F_1z + F_2z^2 + F_3z^3 + \cdots + F_nz^n + \cdots$ (a generating function for the Fibonacci numbers; F(n) is written as F_n here)

$$G(z) = F_1z + F_2z^2 + F_3z^3 + \cdots + F_nz^n + F_{n+1}z^{n+1} + \cdots$$
$$zG(z) = F_1z^2 + F_2z^3 + F_3z^4 + \cdots + F_nz^{n+1} + F_{n+1}z^{n+2} + \cdots$$
$$z^2G(z) = F_1z^3 + F_2z^4 + F_3z^5 + \cdots + F_nz^{n+2} + F_{n+1}z^{n+3} + \cdots$$

$$\begin{cases} F(1) = 1 \\ F(2) = 1 \\ F(n) = F(n-2) + F(n-1) \end{cases}$$

Using Generating Functions (cont'd)

- Let $G(z) = 0 + F_1z + F_2z^2 + F_3z^3 + \cdots + F_nz^n + \cdots$ (a generating function for the Fibonacci numbers; F(n) is written as F_n here)

$$G(z) = F_1z + F_2z^2 + F_3z^3 + \cdots + F_nz^n + F_{n+1}z^{n+1} + \cdots$$
$$zG(z) = F_1z^2 + F_2z^3 + F_3z^4 + \cdots + F_nz^{n+1} + F_{n+1}z^{n+2} + \cdots$$
$$z^2G(z) = F_1z^3 + F_2z^4 + F_3z^5 + \cdots + F_nz^{n+2} + F_{n+1}z^{n+3} + \cdots$$

$$(1 - z - z^2)\, G(z) = z$$

$$\begin{cases} F(1) = 1 \\ F(2) = 1 \\ F(n) = F(n-2) + F(n-1) \end{cases}$$

Using Generating Functions (cont'd)

- Let $G(z) = 0 + F_1z + F_2z^2 + F_3z^3 + \cdots + F_nz^n + \cdots$ (a generating function for the Fibonacci numbers; F(n) is written as F_n here)

$$G(z) = F_1z + F_2z^2 + F_3z^3 + \cdots + F_nz^n + F_{n+1}z^{n+1} + \cdots$$
$$zG(z) = F_1z^2 + F_2z^3 + F_3z^4 + \cdots + F_nz^{n+1} + F_{n+1}z^{n+2} + \cdots$$
$$z^2G(z) = F_1z^3 + F_2z^4 + F_3z^5 + \cdots + F_nz^{n+2} + F_{n+1}z^{n+3} + \cdots$$

$(1 - z - z^2)\, G(z) = z$

$$G(z) = \frac{z}{1-z-z^2} \left(= \frac{z}{(1-\frac{1+\sqrt{5}}{2}z)(1-\frac{1-\sqrt{5}}{2}z)}\right)$$
$$= \frac{\frac{1}{\sqrt{5}}}{1-\frac{1+\sqrt{5}}{2}z} + \frac{-\frac{1}{\sqrt{5}}}{1-\frac{1-\sqrt{5}}{2}z}$$

Using Generating Functions (cont'd)

$$G(z) = 0 + F_1 z + F_2 z^2 + F_3 z^3 + \cdots + F_n z^n + \cdots$$

$$G(z) = \frac{\frac{1}{\sqrt{5}}}{1 - \frac{1+\sqrt{5}}{2} z} + \frac{-\frac{1}{\sqrt{5}}}{1 - \frac{1-\sqrt{5}}{2} z}$$

gen. func.	power series	generated sequence
c / (1 - az)	$c + caz + ca^2z^2 + \cdots + ca^nz^n + \cdots$	c, ca, ca^2, …, ca^n, …

$$\text{Therefore, } F_n = \frac{1}{\sqrt{5}}\left(\frac{1+\sqrt{5}}{2}\right)^n - \frac{1}{\sqrt{5}}\left(\frac{1-\sqrt{5}}{2}\right)^n$$

Divide and Conquer Relations

The running time T(n) of a divide-and-conquer algorithm satisfies

$$T(n) = a\,T(n / b) + cn^k$$

where

- a is the number of subproblems,
- n/b is the size of each subproblem, and
- cn^k is the time spent on dividing the problem and combining the solutions

Divide and Conquer Relations (cont'd)

$$T(n) = a\,T(n / b) + cn^k$$

Assume, for simplicity, $n = b^m$ (such that n / b is an integer)

$$\begin{aligned}
T(n) &= aT(\tfrac{n}{b}) + cn^k \\
&= a(aT(\tfrac{n}{b^2}) + c(\tfrac{n}{b})^k) + cn^k \\
&= a(a(aT(\tfrac{n}{b^3}) + c(\tfrac{n}{b^2})^k) + c(\tfrac{n}{b})^k) + cn^k \\
&\cdots \\
&= a(a(\cdots(aT(\tfrac{n}{b^m}) + c(\tfrac{n}{b^{m-1}})^k) + \cdots) + c(\tfrac{n}{b})^k) + cn^k
\end{aligned}$$

Assume $T(1) = c$

$$\begin{aligned}
T(n) &= ca^m + ca^{m-1}b^k + ca^{m-2}b^{2k} + \cdots + cb^{mk} \\
&= c\sum_{i=0}^{m} a^{m-i}b^{ik} \\
&= ca^m \sum_{i=0}^{m} (\tfrac{b^k}{a})^i
\end{aligned}$$

Divide and Conquer Relations (cont'd)

$$T(n) = ca^m \sum_{i=0}^{m} (\frac{b^k}{a})^i$$

Case 1: $a > b^k$

The factor of the geometric series is less than 1, so the series converges to a constant even if m goes to infinity

Therefore, $T(n) = O(a^m)$

Since $n = b^m$, $m = \log_b n$

$$T(n) = O(a^m) = O(a^{log_b n}) = O(n^{log_b a})$$

Divide and Conquer Relations (cont'd)

$$T(n) = ca^m \sum_{i=0}^{m} (\frac{b^k}{a})^i$$

Case 2: $a = b^k$

The factor of the geometric series is 1

Therefore, $T(n) = O(a^m m)$

Notice that $a = b^k$ implies that $\log_b a = k$ and $m = O(\log n)$

$$T(n) = O(n^k \log n)$$

Divide and Conquer Relations (cont'd)

$$T(n) = ca^m \sum_{i=0}^{m} (\frac{b^k}{a})^i$$

Case 3: $a < b^k$

The factor of the geometric series is greater than 1

Denote b^k / a by F

$$T(n) = ca^m \frac{F^{m+1} - 1}{F - 1} = O(a^m F^m) = O((b^k)^m) = O(n^k)$$

Divide and Conquer Relations (cont'd)

$$T(n) = ca^m \sum_{i=0}^{m} (\frac{b^k}{a})^i$$

If $a_n = ca_{n-1}$, where $c \neq 1$ is a constant, then

$$a_1 + a_2 + a_3 + \cdots + a_n = a_1 \frac{c^n - 1}{c - 1}$$

Denote b^k / a by F

$$T(n) = ca^m \frac{F^{m+1} - 1}{F - 1} = O(a^m F^m) = O((b^k)^m) = O(n^k)$$

Divide and Conquer Relations (cont'd)

$$T(n) = ca^m \sum_{i=0}^{m} (\frac{b^k}{a})^i$$

Case 3: $a < b^k$

The factor of the geometric series is greater than 1

Denote b^k / a by F

$$T(n) = ca^m \frac{F^{m+1} - 1}{F - 1} = O(a^m F^m) = O((b^k)^m) = O(n^k)$$

Divide and Conquer Relations (cont'd)

Theorem (3.4)

The solution of the recurrence relation $T(n) = aT(n/b) + cn^k$, where a and b are integer constants, $a \geq 1$, $b \geq 2$, and c and k are positive constants, is

$$T(n) = \begin{cases} O(n^{\log_b a}) & \text{if } a > b^k \\ O(n^k \log n) & \text{if } a = b^k \\ O(n^k) & \text{if } a < b^k \end{cases}$$

Due to its generality and usefulness, the theorem has conventionally been referred to as "the master theorem"

Recurrence Relations with Full History

- A ***full-history recurrence relation*** is one that depends on all the previous values of the function, not just on a few of them

- Examples:

$$T(n) = c + \sum_{i=1}^{n-1} T(i) \text{ where } c \text{ is a constant and } T(1) \text{ is given}$$

$$T(n) = n - 1 + \frac{2}{n} \sum_{i=1}^{n-1} T(i) \text{ (for } n \geq 2), T(1) = 0$$

Recurrence Relations with Full History (cont'd)

$$T(n) = c + \sum_{i=1}^{n-1} T(i) \text{ where } c \text{ is a constant and } T(1) \text{ is given}$$

$$T(n) - T(n-1) = (c + \sum_{i=1}^{n-1} T(i)) - (c + \sum_{i=1}^{n-2} T(i)) = T(n-1)$$

Hence, T(n) = 2T(n - 1) (which holds only for n ≥ 3)

So, we get

$$\begin{cases} T(2) &= c + T(1) \\ T(n) &= 2T(n-1) \quad \text{if } n \geq 3 \end{cases}$$

Finally, $T(n + 1) = (c + T(1))2^{n-1}$, for n ≥ 2

Recurrence Relations with Full History (cont'd)

$$T(n) = n - 1 + \frac{2}{n}\sum_{i=1}^{n-1} T(i) \text{ (for } n \geq 2), T(1) = 0$$

$$\begin{array}{rcl} nT(n) & = & n(n-1) + 2\sum_{i=1}^{n-1} T(i) \\ (n+1)T(n+1) & = & (n+1)n + 2\sum_{i=1}^{n} T(i) \end{array}$$

$$\begin{array}{rcl} (n+1)T(n+1) - nT(n) & = & (n+1)n - n(n-1) + 2T(n) \\ & = & 2n + 2T(n) \end{array}$$

which implies

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

Recurrence Relations with Full History (cont'd)

$$T(n) = n - 1 + \frac{2}{n}\sum_{i=1}^{n-1} T(i) \text{ (for } n \geq 2), T(1) = 0$$

Further simplification: $T(n+1) \leq \frac{n+2}{n+1}T(n) + 2$

$$\begin{aligned}
T(n) &\leq 2 + \tfrac{n+1}{n}(2 + \tfrac{n}{n-1}(2 + \tfrac{n-1}{n-2}(\cdots(2 + \tfrac{4}{3}T(2))\cdots))) \\
&\leq 2(1 + \tfrac{n+1}{n} + \tfrac{n+1}{n}\tfrac{n}{n-1} + \tfrac{n+1}{n}\tfrac{n}{n-1}\tfrac{n-1}{n-2} + \cdots + (\tfrac{n+1}{n}\tfrac{n}{n-1}\cdots\tfrac{4}{3})) \\
&\leq 2(n+1)(\tfrac{1}{n+1} + \tfrac{1}{n} + \tfrac{1}{n-1} + \cdots + \tfrac{1}{3}) \\
&\leq 2 + 2(n+1)(\tfrac{1}{n} + \tfrac{1}{n-1} + \cdots + 1) \\
&= O(n \log n)
\end{aligned}$$

Recurrence Relations with Full History (cont'd)

$$T(n) = n - 1 + \frac{2}{n}\sum_{i=1}^{n-1} T(i) \text{ (for } n \geq 2), T(1) = 0$$

Further simplification: $T(n+1) \leq \frac{n+2}{n+1}T(n) + 2$

$H(n) = 1 + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{n}$ is the Harmonic series.

$H(n) = \ln n + \gamma + O(\frac{1}{n})$

$T($ $))$

$$\begin{aligned} &\leq \quad 2 + 2(n+1)(\tfrac{1}{n} + \tfrac{1}{n-1} + \cdots + 1) \\ &= \quad O(n \log n) \end{aligned}$$

Useful Facts - Arithmetic Series

$$1 + 2 + 3 + \cdots + n = \frac{n(n+1)}{2}$$

$$a_1 + a_2 + a_3 + \cdots + a_n = \frac{n(a_n + a_1)}{2}$$
if $a_n = a_{n-1} + c$, where c is a constant

Useful Facts - Geometric Series

$$1 + 2 + 4 + \cdots + 2^n = 2^{n+1} - 1$$

$$a_1 + a_2 + a_3 + \cdots + a_n = a_1 \frac{c^n - 1}{c - 1}$$
$$\text{if } a_n = ca_{n-1}, \text{ where } c \neq 1 \text{ is a constant}$$

$$\sum_{i=1}^{\infty} a_i = \frac{a_1}{1 - c} \text{ if } 0 < c < 1$$

Useful Facts - Sum of Squares & Harmonic Series

Sum of Squares

$$\sum_{i=1}^{n} i^2 = \frac{n(n+1)(2n+1)}{6}$$

Harmonic Series:

$$H_n = \sum_{k=1}^{n} \frac{1}{k} = \ln n + \gamma + O(\frac{1}{n})$$

$$\text{where } \gamma = 0.577.. \text{ is Euler's constant}$$

Useful Facts - Basic Rules Involving Logarithms

$$\log_b a = \frac{1}{\log_a b}$$

$$\log_a x = \frac{\log_b x}{\log_b a}$$

$$b^{\log_b x} = x$$

$$b^{\log_a x} = x^{\log_a b}$$

Useful Facts - Sum of Logarithms & Stirling's Approximation

Sum of logarithms:

$$\begin{aligned}\sum_{i=1}^{n} \lfloor \log_2 i \rfloor &= (n+1)\lfloor \log_2 n \rfloor - 2^{\lfloor \log_2 n \rfloor + 1} + 2 \\ &= \Theta(n \log n)\end{aligned}$$

Stirling's approximation:

$$n! = \sqrt{2\pi n}\left(\frac{n}{e}\right)^n (1 + O(\frac{1}{n}))$$

which implies $\log_2(n!) = \Theta(n \log n)$

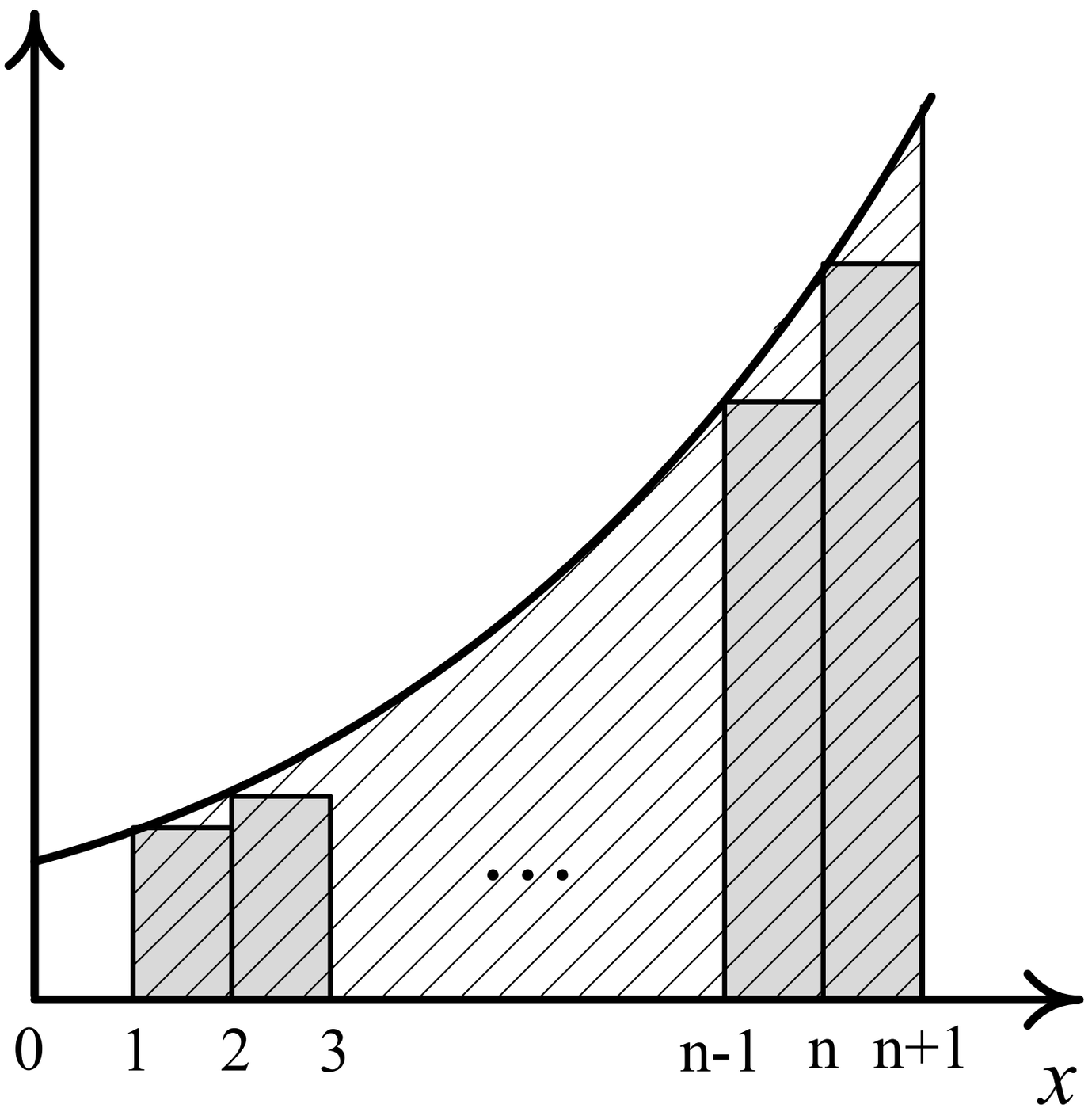
Useful Facts - Bounding a Summation by an Integral

- If f(x) is a monotonically increasing continuous function, then

$$\sum_{i=1}^{n} f(i) \leq \int_{x=1}^{x=n+1} f(x)dx$$

Useful Facts - Bounding a Summation by an Integral

- If f(x) is nction, then



Exercise

- What's the time complexity of the following program?

```
for (int i = 0, i < n, i++) {
    constant time operations
}
```

Exercise

- What's the time complexity of the following program?

```
for (int i = 0, i < n, i++) {
    for (int j = 0, j < n, j++) {
        constant time operations
    }
}
```

Exercise

- What's the time complexity of the following program?

```
for (int i = n, i > 0, i = i / 2) {
    for (int j = i; j < n; j = j++) {
        constant time operations
    }
}
```

Complexity Analysis (cont'd)

```
// Sort the given run of array A[] using array B[] as a source.
// iBegin is inclusive; iEnd is exclusive (A[iEnd] is not in the set).
TopDownSplitMerge(B[], iBegin, iEnd, A[])
{
    if (iEnd - iBegin < 2)                      // if run size == 1
        return;                                 //   consider it sorted
    // split the run longer than 1 item into halves
    iMiddle = (iEnd + iBegin) / 2;              // iMiddle = mid point
    // recursively sort both runs from array A[] into B[]
    TopDownSplitMerge(A, iBegin,  iMiddle, B);  // sort the left  run
    TopDownSplitMerge(A, iMiddle,    iEnd, B);  // sort the right run
    // merge the resulting runs from array B[] into A[]
    TopDownMerge(B, iBegin, iMiddle, iEnd, A);
}
```

Complexity Analysis (cont'd)

- Let T(n) denote the running time of TopDownSplitMerge for a list of n objects

- Then T(n) must satisfy the recurrence relation $T(n) = 2T(n/2) + n$

Complexity Analysis (cont'd)

- Let T(n) denote the running time of TopDownSplitMerge for a list of n objects

- Then T(n) must satisfy the recurrence relation $T(n) = 2T(n/2) + n$

Theorem (3.4)

The solution of the recurrence relation $T(n) = aT(n/b) + cn^k$, where a and b are integer constants, $a \geq 1$, $b \geq 2$, and c and k are positive constants, is

$$T(n) = \begin{cases} O(n^{\log_b a}) & \text{if } a > b^k \\ O(n^k \log n) & \text{if } a = b^k \\ O(n^k) & \text{if } a < b^k \end{cases}$$

Complexity Analysis (cont'd)

- Let T(n) denote the running time of TopDownSplitMerge for a list of n objects
- Then T(n) must satisfy the recurrence relation $T(n) = 2T(n/2) + n$

Theorem (3.4)

The solution of the recurrence relation $T(n) = aT(n/b) + cn^k$, where a and b are integer constants, $a \geq 1$, $b \geq 2$, and c and k are positive constants, is

$$T(n) = \begin{cases} O(n^{\log_b a}) & \text{if } a > b^k \\ O(n^k \log n) & \text{if } a = b^k \\ O(n^k) & \text{if } a < b^k \end{cases}$$

Complexity Analysis (cont'd)

- Let T(n) denote the running time of TopDownSplitMerge for a list of n objects
- Then T(n) must satisfy the recurrence relation $T(n) = 2T(n/2) + n$

Theorem (3.4)

The solution of the recurrence relation $T(n) = aT(n/b) + cn^k$, where a and b are integer constants, $a \geq 1$, $b \geq 2$, and c and k are positive constants, is

$$T(n) = \begin{cases} O(n^{\log_b a}) & \text{if } a > b^k \\ O(n^k \log n) & \text{if } a = b^k \\ O(n^k) & \text{if } a < b^k \end{cases}$$

$a = 2, b = 2, c = 1, k = 1$

Complexity Analysis (cont'd)

- Let T(n) denote the running time of TopDownSplitMerge for a list of n objects
- Then T(n) must satisfy the recurrence relation $T(n) = 2T(n/2) + n$

Theorem (3.4)

The solution of the recurrence relation $T(n) = aT(n/b) + cn^k$, where a and b are integer constants, $a \geq 1$, $b \geq 2$, and c and k are positive constants, is

$$T(n) = \begin{cases} O(n^{\log_b a}) & \text{if } a > b^k \\ O(n^k \log n) & \text{if } a = b^k \\ O(n^k) & \text{if } a < b^k \end{cases}$$

$a = 2, b = 2, c = 1, k = 1$

$a = b^k$

Complexity Analysis (cont'd)

- Let T(n) denote the running time of TopDownSplitMerge for a list of n objects
- Then T(n) must satisfy the recurrence relation $T(n) = 2T(n/2) + n$

Theorem (3.4)

The solution of the recurrence relation $T(n) = aT(n/b) + cn^k$, where a and b are integer constants, $a \geq 1$, $b \geq 2$, and c and k are positive constants, is

$$T(n) = \begin{cases} O(n^{\log_b a}) & \text{if } a > b^k \\ O(n^k \log n) & \text{if } a = b^k \\ O(n^k) & \text{if } a < b^k \end{cases}$$

$a = 2$, $b = 2$, $c = 1$, $k = 1$

$a = b^k$

Therefore, $T(n) = O(n \log n)$