Introduction to the Theory of Computation

Set 10 — Complexity (1)

Complexity of Algorithms

Among *decidable* problems, we still have **levels of difficulty** for algorithms

We may consider an algorithm more difficult for a variety of reasons

- Takes longer to execute
- Requires more memory to execute

Time complexity: Given an algorithm and an input string, how long will the algorithm take to execute?

Example

```
INSERTION-SORT(A)
For j = 2 to length(A)
    key := A[j]
    i := j-1
    While i > 0 and A[i] > key
        A[i+1] := A[i]
        i := i-1
    A[i+1] := key
```

Trace

4 6 1 8 5 3

- Start by looking at 6 & compare to 4
- 4 ≤ 6
- Next look at 1

1 4 6 8 5 3

- 8 is okay
- Move 5 then move 3

1 4 5 6 8 3

1 3 4 5 6 8

```
INSERTION-SORT(A)
For j = 2 to length(A)
    key := A[j]
    i := j-1
    While i > 0 and A[i] > key
        A[i+1] := A[i]
        i := i-1
    A[i+1] := key
```

How long does insertion sort take?

Two loops

Outer loop executed (n-1) times

n = length(A)

Inner loop executed up to j times

Total time is at most $\sum_{1\le k<n} (4+\sum_{1\le i<k} 3)$

$\sum_{1\le k<n} 4+\sum_{1\le k<n} 3k$

$4(n-1) + 3[(n+1)n/2-1] =$

$4n- 4 + 1.5 n^2 + 1.5 n - 2 =$

$1.5 n^2 + 5.5 n - 6$

```
INSERTION-SORT(A)
For j = 2 to length(A)
    key := A[j]
    i := j-1
    While i > 0 and A[i] > key
        A[i+1] := A[i]
        i := i-1
    A[i+1] := key
```

Big-O Notation

In general, the time complexity will be a sum of terms that is dominated by one term

- For example, $n^2 + 2n - 3$ is dominated by the n^2 term

Time complexity is most concerned with behavior for large n

- We disregard all terms except for the dominating term
- $n^2 + 2n - 3 = O(n^2)$

Asymptotic Upper Bound

Definition: Let f and g be two functions from $\mathbb{N}$ (natural numbers) to $\mathbb{R}^+$ (positive real numbers).

Then $f(n)=O(g(n))$ if positive integers c and n_0 exist such that for every $n \geq n_0$, $f(n) \leq c \times g(n)$.

In this case, we say that g(n) is an *upper bound* for f(n).

Example

$3n^4 + 5n^2 - 4 = O(\quad)$

$3n^4 + 5n^2 - 4 \leq 4n^4$ for every $n \geq 2$ since

$3n^4 + 5n^2 - 4 \leq 4n^4$

$\Rightarrow n^4 - 5n^2 + 4 \geq 0$

$\Rightarrow (n^2 - 4)(n^2 - 1) \geq 0$ *clearly holds for all* $n \geq 2$

For polynomials, we can drop everything except n^k, where k is the largest exponent

Big-O Notation and Logarithms

Recall $\log_b n = \log_x n / \log_x b$

- $\log_b n = O(\log_x n)$ for every $x > 0$
- With big-O notation, the base of the logarithm is unimportant!

$5n^5 \log_3 n - 3n^2 \log_2 \log_2 n = O(n^5 \log n)$

Mathematics with Big-O Notation

If $f(n) = O(n^3) + O(n)$, then $f(n) = O(n^3)$

Can simply select the largest term

What does $f(n) = 3^{O(n)}$ mean?

$3^{O(n)} \geq 3^{cn}$ for some constant c

How about $O(1)$?

$O(1) \geq c$ for some constant c

Constant time

Exponentials

What about $f(n) = 2^{O(\log n)}$?

$n = 2^{\log_2 n}$ [identity]

$n^c = 2^{c \log_2 n}$ [identity]

$n^c = 2^{O(\log n)}$ [upper bound of n^c for some c]

$2^{O(\log n)} = n^{O(1)}$ [equivalent upper bound]

An algorithm takes polynomial time if its complexity is $O(n^k)$ for some $k>0$

An algorithm takes exponential time if its complexity is $O(a^{n^i})$, where $a \geq 2$, $i>0$

Small-o Notation

Definition: Let f and g be two functions from $\mathbb{N}$ to $\mathbb{R}^+$.

Then $f(n)=o(g(n))$

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

That is, for *any* positive real number c, a number n_0 exists such that

for every $n \geq n_0$, $f(n) < c \times g(n)$

Big-O *versus* Small-o

If f(n) = o(g(n)) then f(n) = O(g(n)), but the reverse is not always true

Big-O is like “less than or equal” while small-o is like “strictly less than”

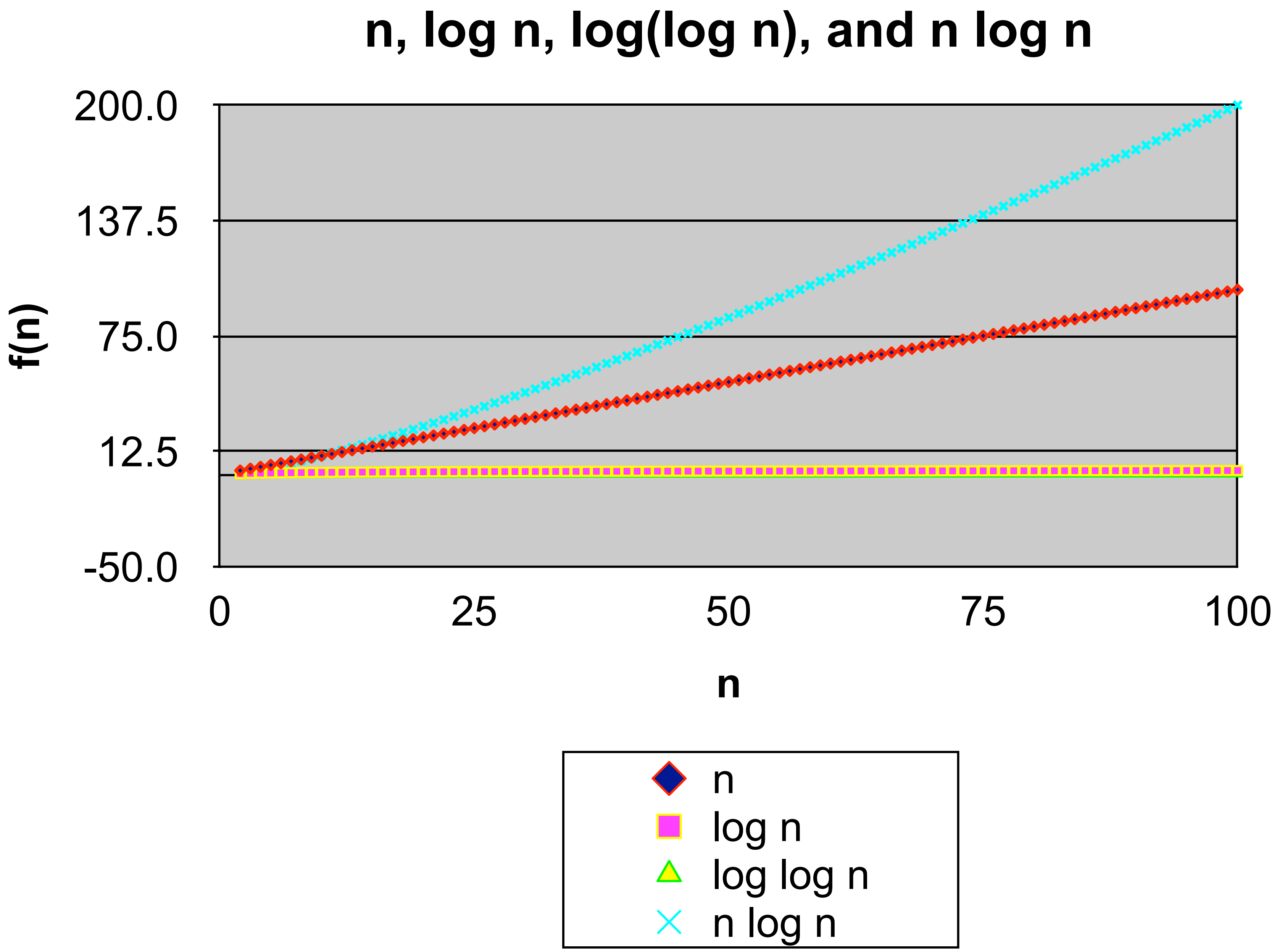
- Example
  - f(n) = O(f(n)) for every function f
  - f(n) ≠ o(f(n)) for every function f

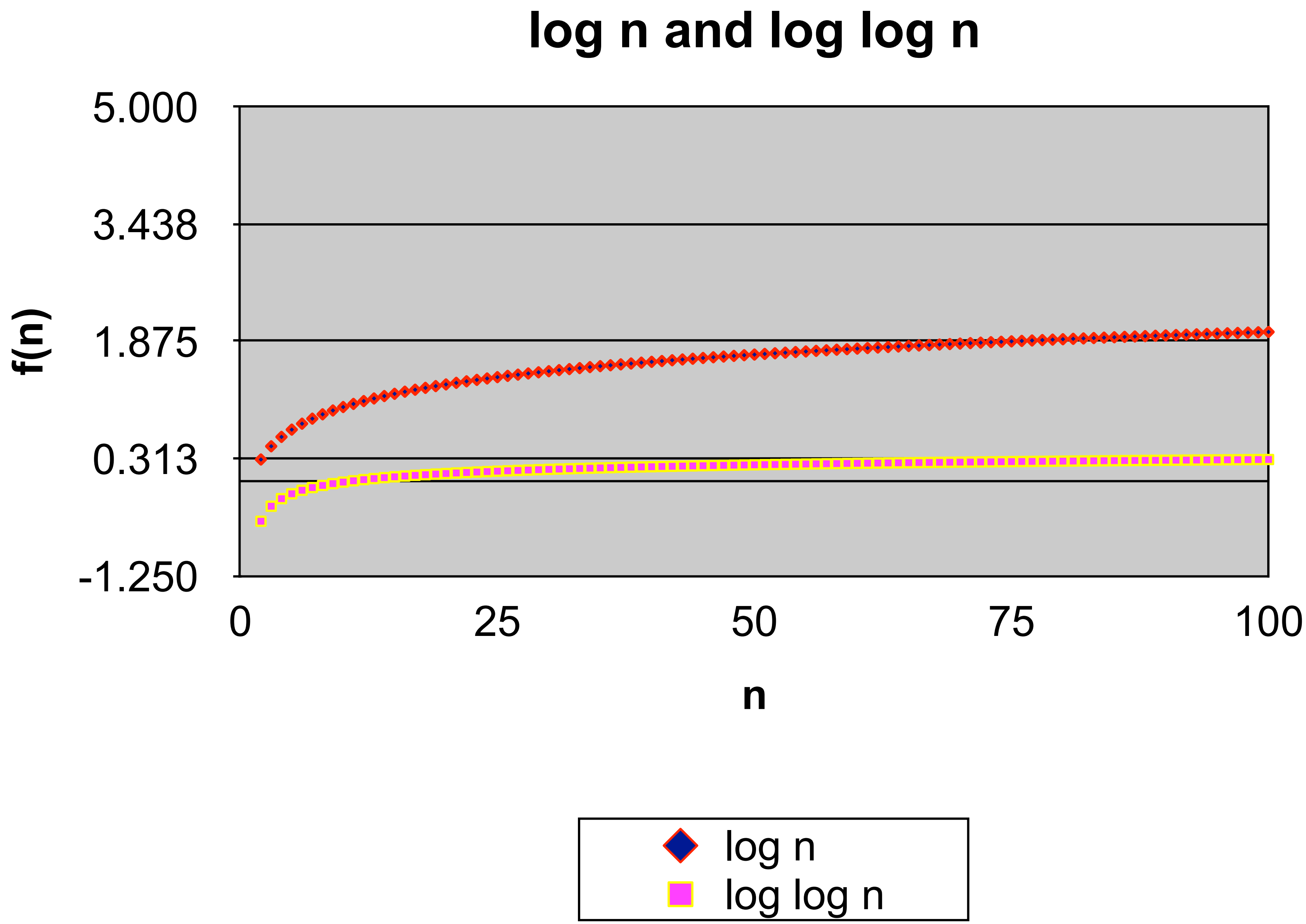
Some Identities

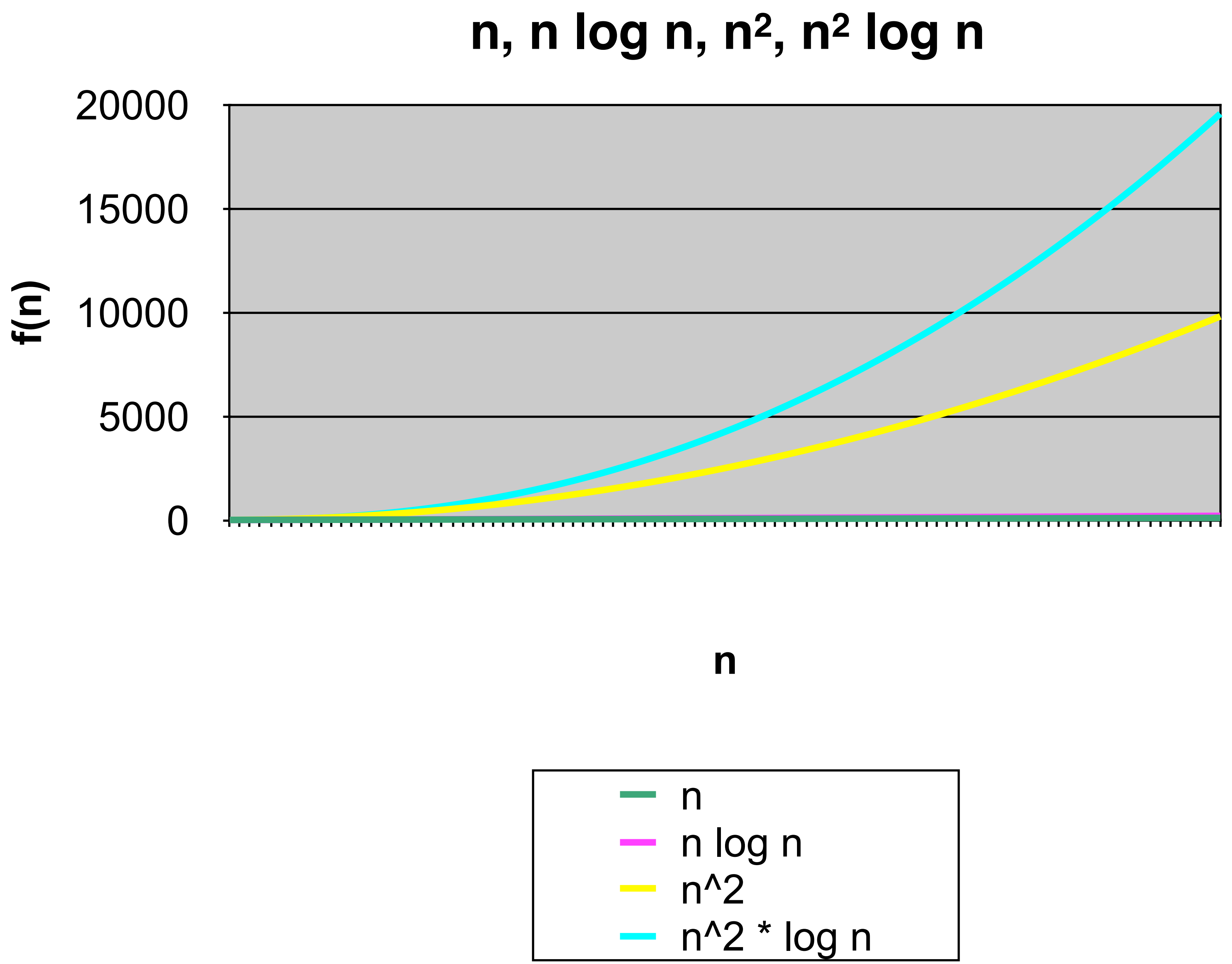
$n^i = o(n^k)$ for every $i < k$

$\log n = o(n)$

$\log \log n = o(\log n)$


n, log n, log(log n), and n log n
f(n)
200.0
137.5
75.0
12.5
-50.0
0
25
50
75
100
n
◆ n
■ log n
△ log log n
× n log n


log n and log log n
5.000
3.438
1.875
0.313
-1.250
f(n)
0
25
50
75
100
n
log n
log log n


n, n log n, n², n² log n
20000
15000
10000
5000
0
f(n)
n
n
n log n
n^2
n^2 * log n

Small-o *vs.* Big-O

Small-o is strictly less than

Big-O is less than or equal to

For any function f, is f(n) = o(f(n))?

- No … never!

For any function f, is f(n) = O(f(n))?

- Yes … always!

Analyzing Algorithms

We examine an algorithm to determine how long it will take to halt on an input of length n

- The amount of time to complete is called the algorithm's complexity class

Definition: Let t:$\mathbb{N} \rightarrow \mathbb{N}$ be a function.

The time complexity class, TIME(t(n)), is

TIME(t(n)) = { L I L is a language decided by an O(t(n))-time algorithm }

Example

Earlier, we saw that insertion sort takes $1.5\ n^2 + 5.5\ n - 6$ time

Insertion sort is in the time complexity class $O(n^2)$

Insertion sort is also in the time complexity class $O(n^k)$ for any $k > 2$

Importance of Model

The complexity of algorithms is a function of the length of the input

This length may vary depending on assumptions about our data and other model assumptions

Another Example

Finding minimum element in a set

Amount of time depends on the structure of the input

If set is a sorted array?

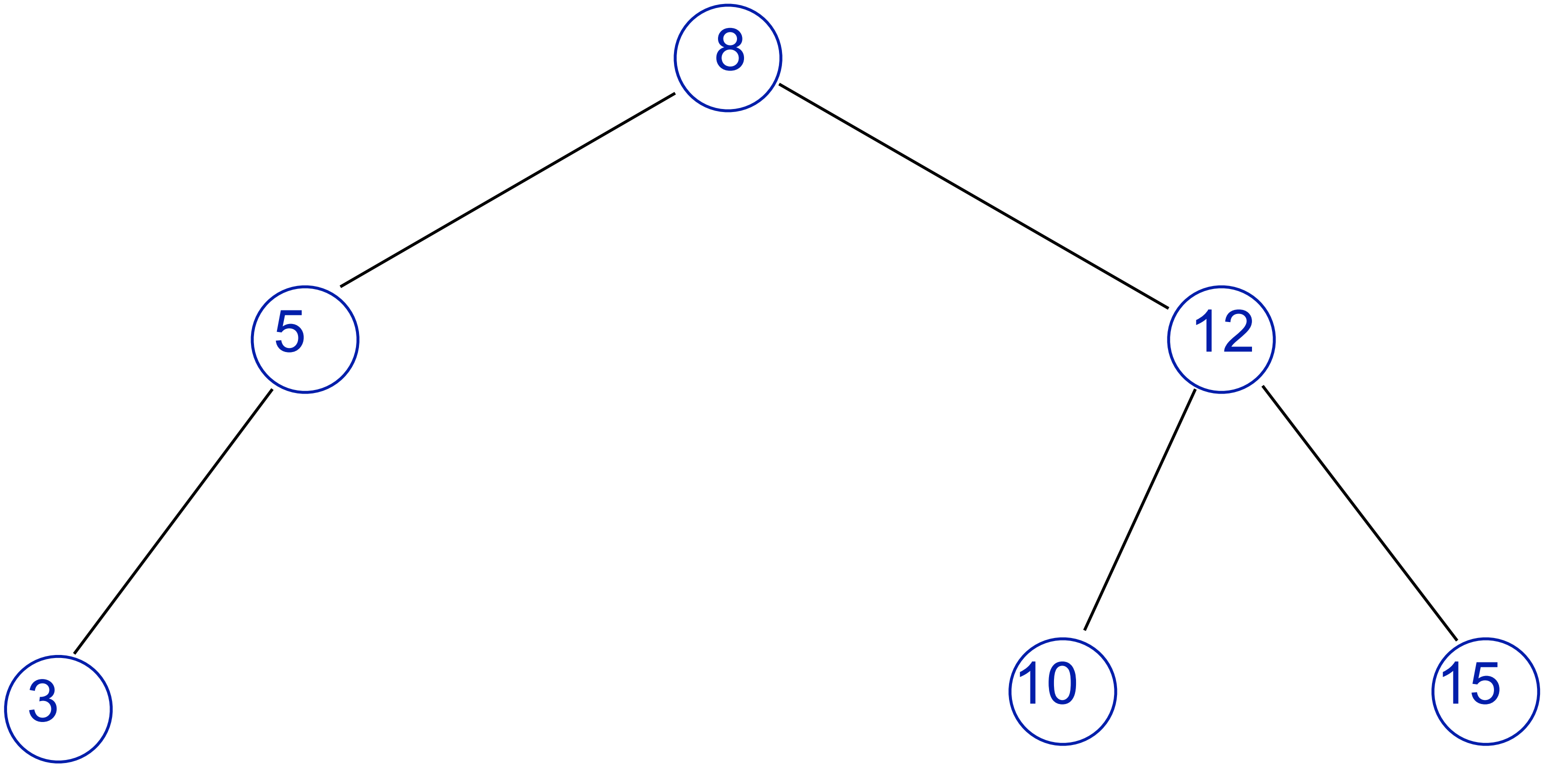
- O(1)

If set is an unsorted array?

- O(n)

If set is a balanced sorted tree?

Sorted Tree



Sorted Tree Examined

Finding minimum involves selecting left child until you reach a leaf

- Number of steps = depth of tree

Since the tree is balanced, the depth of the tree is O(log n)

What if the tree was not balanced?

Size of Input: Important Consideration

The running time is measured in terms of the size of the input

- If we increase the input size can that make the problem *seem* more efficient
- For example, represent integers in unary instead of binary

We consider only reasonable encodings

- The space used to encode the integer value v must be O(log v)

Unary *vs.* Binary Encoding

In unary encoding, the value 13 is encoded 1111111111111

- Length of encoding of value v is v

In binary encoding, the value 13 is encoded 1101

- Length of encoding of value v is $\lfloor \log_2 v \rfloor + 1$

Why does encoding matter?

Assume an algorithm takes as its input an integer of value v

What is the time complexity of an algorithm if it takes integer input with value v and executes for v steps?

- Recall time complexity is a function of the **length** of the input

If encoding is in unary, the complexity is O(n)

If encoding is binary, the complexity is $O(2^n)$

Example

An important problem in cryptography is prime factorization

- Most encryptions rely on the fact that prime factorization takes a long time (exponential in the length of the input)

Clearly, we can find the prime factorization of v by checking whether each integer smaller than v divides it

Prime Factorization

```
PRIME_FACTOR(v)
  w = v
  factors = ∅
  for i = 2 to v
      do while i divides w
           w = w / i
           factors = factors ∪ {i}
      enddo
      if w = 1 break
  next
```

factors is a multiset & will contain prime factors

Worst case execution time is v
(occurs when v is prime)

Complexity of Prime Factorization

The algorithm has complexity O(v)

- v is the value of the input

A trickster could claim they have a linear algorithm simply by changing the encoding of the input

- If the input is unary, then the factorization is linear in the length of the input
    - But this is cheating!

Enforcing reasonable encodings keeps this trick from occurring