

# Outline

MCS 260 L-29

2 November 2007

## Complexity and Cost

Measuring Complexity: big-oh  
complexity classes

### Complexity and Cost

Measuring Complexity:  
big-oh  
complexity classes

## Cost of Algorithms

timing Python programs  
counting flops: floating-point operations

### Cost of Algorithms

timing Python programs  
counting flops:  
floating-point operations

## Maple's cost function

### Maple's cost function

## Summary + Assignments

### Summary + Assignments

MCS 260 Lecture 29  
Introduction to Computer Science  
Jan Vershelde, 2 November 2007

# Complexity and Cost

of problems and algorithms

**Complexity measures the hardness of a problem.**

Cost is a property of an algorithm to solve a problem.

Efficiency concerns use of

**space** for intermediate and final results;

**time** for arithmetic, communication, management.

Depending on the type of inputs, one distinguishes between worst case, best case, and average case.

Importance for software development:

1. complexity coincides with cost of the best algorithm;
2. cost analysis of programs reveals its bottleneck.

Applications: public key cryptography; tuning algorithms.

## Complexity and Cost

Measuring Complexity:  
big-oh  
complexity classes

## Cost of Algorithms

timing Python programs  
counting flops:  
floating-point operations

## Maple's cost function

## Summary + Assignments

# Complexity and Cost

of problems and algorithms

Complexity measures the hardness of a problem.  
Cost is a property of an algorithm to solve a problem.

Efficiency concerns use of

space for intermediate and final results;

time for arithmetic, communication, management.

Depending on the type of inputs, one distinguishes between worst case, best case, and average case.

Importance for software development:

1. complexity coincides with cost of the best algorithm;
2. cost analysis of programs reveals its bottleneck.

Applications: public key cryptography; tuning algorithms.

## Complexity and Cost

Measuring Complexity:  
big-oh  
complexity classes

## Cost of Algorithms

timing Python programs  
counting flops:  
floating-point operations

## Maple's cost function

## Summary + Assignments

# Complexity and Cost

of problems and algorithms

Complexity measures the hardness of a problem.  
Cost is a property of an algorithm to solve a problem.

Efficiency concerns use of

**space** for intermediate and final results;

**time** for arithmetic, communication, management.

Depending on the type of inputs, one distinguishes between worst case, best case, and average case.

Importance for software development:

1. complexity coincides with cost of the best algorithm;
2. cost analysis of programs reveals its bottleneck.

Applications: public key cryptography; tuning algorithms.

## Complexity and Cost

Measuring Complexity:  
big-oh  
complexity classes

## Cost of Algorithms

timing Python programs  
counting flops:  
floating-point operations

## Maple's cost function

## Summary + Assignments

# Complexity and Cost

of problems and algorithms

Complexity measures the hardness of a problem.  
Cost is a property of an algorithm to solve a problem.

Efficiency concerns use of

**space** for intermediate and final results;

**time** for arithmetic, communication, management.

Depending on the type of inputs, one distinguishes between worst case, best case, and average case.

Importance for software development:

1. complexity coincides with cost of the best algorithm;
2. cost analysis of programs reveals its bottleneck.

Applications: public key cryptography; tuning algorithms.

## Complexity and Cost

Measuring Complexity:  
big-oh  
complexity classes

## Cost of Algorithms

timing Python programs  
counting flops:  
floating-point operations

## Maple's cost function

## Summary + Assignments

# Complexity and Cost

of problems and algorithms

Complexity measures the hardness of a problem.  
Cost is a property of an algorithm to solve a problem.

Efficiency concerns use of

**space** for intermediate and final results;

**time** for arithmetic, communication, management.

Depending on the type of inputs, one distinguishes between worst case, best case, and average case.

Importance for software development:

1. complexity coincides with cost of the best algorithm;
2. cost analysis of programs reveals its bottleneck.

Applications: public key cryptography; tuning algorithms.

## Complexity and Cost

Measuring Complexity:  
big-oh  
complexity classes

## Cost of Algorithms

timing Python programs  
counting flops:  
floating-point operations

## Maple's cost function

## Summary + Assignments

# Complexity and Cost

of problems and algorithms

Complexity measures the hardness of a problem.  
Cost is a property of an algorithm to solve a problem.

Efficiency concerns use of

**space** for intermediate and final results;

**time** for arithmetic, communication, management.

Depending on the type of inputs, one distinguishes between worst case, best case, and average case.

Importance for software development:

1. complexity coincides with cost of the best algorithm;
2. cost analysis of programs reveals its bottleneck.

Applications: public key cryptography; tuning algorithms.

## Complexity and Cost

Measuring Complexity:  
big-oh  
complexity classes

## Cost of Algorithms

timing Python programs  
counting flops:  
floating-point operations

## Maple's cost function

## Summary + Assignments

# Complexity and Cost

of problems and algorithms

Complexity measures the hardness of a problem.  
Cost is a property of an algorithm to solve a problem.

Efficiency concerns use of

**space** for intermediate and final results;

**time** for arithmetic, communication, management.

Depending on the type of inputs, one distinguishes between worst case, best case, and average case.

Importance for software development:

1. complexity coincides with cost of the best algorithm;
2. cost analysis of programs reveals its bottleneck.

Applications: public key cryptography; tuning algorithms.

## Complexity and Cost

Measuring Complexity:  
big-oh  
complexity classes

## Cost of Algorithms

timing Python programs  
counting flops:  
floating-point operations

## Maple's cost function

## Summary + Assignments

# Complexity and Cost

of problems and algorithms

Complexity measures the hardness of a problem.  
Cost is a property of an algorithm to solve a problem.

Efficiency concerns use of

**space** for intermediate and final results;

**time** for arithmetic, communication, management.

Depending on the type of inputs, one distinguishes between worst case, best case, and average case.

Importance for software development:

1. complexity coincides with cost of the best algorithm;
2. cost analysis of programs reveals its bottleneck.

Applications: public key cryptography; tuning algorithms.

## Complexity and Cost

Measuring Complexity:  
big-oh  
complexity classes

## Cost of Algorithms

timing Python programs  
counting flops:  
floating-point operations

## Maple's cost function

## Summary + Assignments

## Complexity and Cost

Measuring Complexity: big-oh  
complexity classes

### Complexity and Cost

Measuring Complexity:  
big-oh  
complexity classes

## Cost of Algorithms

timing Python programs  
counting flops: floating-point operations

### Cost of Algorithms

timing Python programs  
counting flops:  
floating-point operations

## Maple's cost function

### Maple's cost function

## Summary + Assignments

### Summary + Assignments

# The big-oh Notation

to measure complexity

Let  $n$  be the dimension of the problem,  
e.g.:  $n$  is number of elements to add, sort, etc...

A function  $f(n)$  is  $O(g(n))$  (we say:  $f$  is of order  $g$ )  
if there exists a positive constant  $c$  (*independent of  $n$* ):  
 $f(n) \leq cg(n)$ , for sufficiently large  $n$ .

Big-oh defines the order of complexity, some examples:

- ▶  $f$  is  $O(\log(n))$ : logarithmic in  $n$ ;
- ▶  $f$  is  $O(n)$ : linear in  $n$ ;
- ▶  $f$  is  $O(n^2)$ : quadratic in  $n$ ;
- ▶  $f$  is  $O(2^n)$ : exponential in  $n$ .

Complexity and  
Cost

Measuring Complexity:  
big-oh

complexity classes

Cost of Algorithms

timing Python programs  
counting flops:  
floating-point operations

Maple's cost  
function

Summary +  
Assignments

# The big-oh Notation

to measure complexity

Let  $n$  be the dimension of the problem,  
e.g.:  $n$  is number of elements to add, sort, etc...

A function  $f(n)$  is  $O(g(n))$  (we say:  $f$  is of order  $g$ )

if there exists a positive constant  $c$  (*independent of  $n$* ):

$f(n) \leq cg(n)$ , for sufficiently large  $n$ .

Big-oh defines the order of complexity, some examples:

- ▶  $f$  is  $O(\log(n))$ : logarithmic in  $n$ ;
- ▶  $f$  is  $O(n)$ : linear in  $n$ ;
- ▶  $f$  is  $O(n^2)$ : quadratic in  $n$ ;
- ▶  $f$  is  $O(2^n)$ : exponential in  $n$ .

Complexity and  
Cost

Measuring Complexity:  
big-oh

complexity classes

Cost of Algorithms

timing Python programs  
counting flops:  
floating-point operations

Maple's cost  
function

Summary +  
Assignments

# The big-oh Notation

to measure complexity

Let  $n$  be the dimension of the problem,  
e.g.:  $n$  is number of elements to add, sort, etc...

A function  $f(n)$  is  $O(g(n))$  (we say:  $f$  is of order  $g$ )  
if there exists a positive constant  $c$  (*independent of  $n$* ):  
 $f(n) \leq cg(n)$ , for sufficiently large  $n$ .

Big-oh defines the order of complexity, some examples:

- ▶  $f$  is  $O(\log(n))$ : logarithmic in  $n$ ;
- ▶  $f$  is  $O(n)$ : linear in  $n$ ;
- ▶  $f$  is  $O(n^2)$ : quadratic in  $n$ ;
- ▶  $f$  is  $O(2^n)$ : exponential in  $n$ .

Complexity and  
Cost

Measuring Complexity:  
big-oh  
complexity classes

Cost of Algorithms

timing Python programs  
counting flops:  
floating-point operations

Maple's cost  
function

Summary +  
Assignments

# The big-oh Notation

to measure complexity

Let  $n$  be the dimension of the problem,  
e.g.:  $n$  is number of elements to add, sort, etc...

A function  $f(n)$  is  $O(g(n))$  (we say:  $f$  is of order  $g$ )  
if there exists a positive constant  $c$  (*independent of  $n$* ):  
 $f(n) \leq cg(n)$ , for sufficiently large  $n$ .

Big-oh defines the order of complexity, some examples:

- ▶  $f$  is  $O(\log(n))$ : logarithmic in  $n$ ;
- ▶  $f$  is  $O(n)$ : linear in  $n$ ;
- ▶  $f$  is  $O(n^2)$ : quadratic in  $n$ ;
- ▶  $f$  is  $O(2^n)$ : exponential in  $n$ .

Complexity and  
Cost

Measuring Complexity:  
big-oh  
complexity classes

Cost of Algorithms

timing Python programs  
counting flops:  
floating-point operations

Maple's cost  
function

Summary +  
Assignments

# The big-oh Notation

to measure complexity

Let  $n$  be the dimension of the problem,  
e.g.:  $n$  is number of elements to add, sort, etc...

A function  $f(n)$  is  $O(g(n))$  (we say:  $f$  is of order  $g$ )  
if there exists a positive constant  $c$  (*independent of  $n$* ):  
 $f(n) \leq cg(n)$ , for sufficiently large  $n$ .

Big-oh defines the order of complexity, some examples:

- ▶  $f$  is  $O(\log(n))$ : logarithmic in  $n$ ;
- ▶  $f$  is  $O(n)$ : linear in  $n$ ;
- ▶  $f$  is  $O(n^2)$ : quadratic in  $n$ ;
- ▶  $f$  is  $O(2^n)$ : exponential in  $n$ .

Complexity and  
Cost

Measuring Complexity:  
big-oh  
complexity classes

Cost of Algorithms

timing Python programs  
counting flops:  
floating-point operations

Maple's cost  
function

Summary +  
Assignments

# The big-oh Notation

to measure complexity

Let  $n$  be the dimension of the problem,  
e.g.:  $n$  is number of elements to add, sort, etc...

A function  $f(n)$  is  $O(g(n))$  (we say:  $f$  is of order  $g$ )  
if there exists a positive constant  $c$  (*independent of  $n$* ):  
 $f(n) \leq cg(n)$ , for sufficiently large  $n$ .

Big-oh defines the order of complexity, some examples:

- ▶  $f$  is  $O(\log(n))$ : logarithmic in  $n$ ;
- ▶  $f$  is  $O(n)$ : linear in  $n$ ;
- ▶  $f$  is  $O(n^2)$ : quadratic in  $n$ ;
- ▶  $f$  is  $O(2^n)$ : exponential in  $n$ .

Complexity and  
Cost

Measuring Complexity:  
big-oh  
complexity classes

Cost of Algorithms

timing Python programs  
counting flops:  
floating-point operations

Maple's cost  
function

Summary +  
Assignments

# The big-oh Notation

to measure complexity

Let  $n$  be the dimension of the problem,  
e.g.:  $n$  is number of elements to add, sort, etc...

A function  $f(n)$  is  $O(g(n))$  (we say:  $f$  is of order  $g$ )  
if there exists a positive constant  $c$  (*independent of  $n$* ):  
 $f(n) \leq cg(n)$ , for sufficiently large  $n$ .

Big-oh defines the order of complexity, some examples:

- ▶  $f$  is  $O(\log(n))$ : logarithmic in  $n$ ;
- ▶  $f$  is  $O(n)$ : linear in  $n$ ;
- ▶  $f$  is  $O(n^2)$ : quadratic in  $n$ ;
- ▶  $f$  is  $O(2^n)$ : exponential in  $n$ .

Complexity and  
Cost

Measuring Complexity:  
big-oh  
complexity classes

Cost of Algorithms

timing Python programs  
counting flops:  
floating-point operations

Maple's cost  
function

Summary +  
Assignments

# The big-oh Notation

to measure complexity

Let  $n$  be the dimension of the problem,  
e.g.:  $n$  is number of elements to add, sort, etc...

A function  $f(n)$  is  $O(g(n))$  (we say:  $f$  is of order  $g$ )  
if there exists a positive constant  $c$  (*independent of  $n$* ):  
 $f(n) \leq cg(n)$ , for sufficiently large  $n$ .

Big-oh defines the order of complexity, some examples:

- ▶  $f$  is  $O(\log(n))$ : logarithmic in  $n$ ;
- ▶  $f$  is  $O(n)$ : linear in  $n$ ;
- ▶  $f$  is  $O(n^2)$ : quadratic in  $n$ ;
- ▶  $f$  is  $O(2^n)$ : exponential in  $n$ .

Complexity and  
Cost

Measuring Complexity:  
big-oh

complexity classes

Cost of Algorithms

timing Python programs  
counting flops:  
floating-point operations

Maple's cost  
function

Summary +  
Assignments

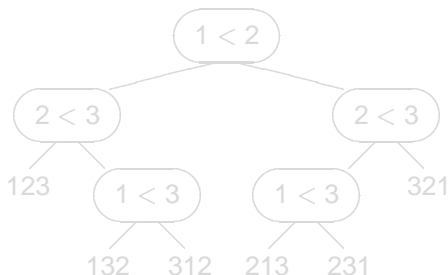
# Complexity of Sorting

independent of algorithm used

Minimal number of comparisons to sort  $n$  numbers?

#permutations equals  $n! = n \cdot (n - 1) \cdots 2 \cdot 1$ .

A sort computes a permutation to order the list.



$S(n)$  = minimal #comparisons. From the tree:  $n! \leq 2^{S(n)}$ .

Stirling:  $n! \approx \sqrt{2\pi n} \frac{n^n}{e^n} \Rightarrow O(\log(n!)) = O(n \log(n))$ .

A lower bound on sorting complexity:  $O(n \log(n))$ .

Complexity and Cost

Measuring Complexity:  
big-oh  
complexity classes

Cost of Algorithms

timing Python programs  
counting flops:  
floating-point operations

Maple's cost function

Summary + Assignments

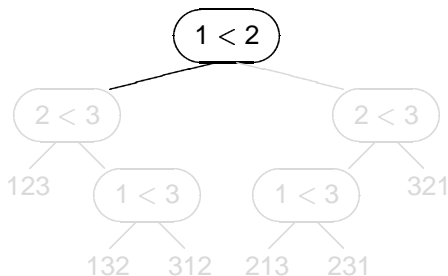
# Complexity of Sorting

independent of algorithm used

Minimal number of comparisons to sort  $n$  numbers?

#permutations equals  $n! = n \cdot (n - 1) \cdots 2 \cdot 1$ .

A sort computes a permutation to order the list.



$S(n)$  = minimal #comparisons. From the tree:  $n! \leq 2^{S(n)}$ .

Stirling:  $n! \approx \sqrt{2\pi n} \frac{n^n}{e^n} \Rightarrow O(\log(n!)) = O(n \log(n))$ .

A lower bound on sorting complexity:  $O(n \log(n))$ .

Complexity and Cost

Measuring Complexity:  
big-oh  
complexity classes

Cost of Algorithms

timing Python programs  
counting flops:  
floating-point operations

Maple's cost function

Summary + Assignments

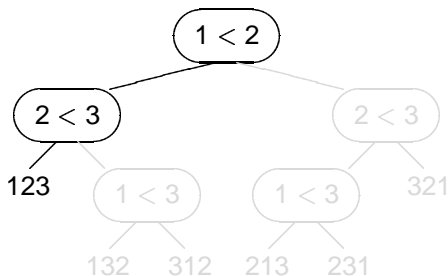
# Complexity of Sorting

independent of algorithm used

Minimal number of comparisons to sort  $n$  numbers?

#permutations equals  $n! = n \cdot (n - 1) \cdots 2 \cdot 1$ .

A sort computes a permutation to order the list.



$S(n)$  = minimal #comparisons. From the tree:  $n! \leq 2^{S(n)}$ .  
 Stirling:  $n! \approx \sqrt{2\pi n} \frac{n^n}{e^n} \Rightarrow O(\log(n!)) = O(n \log(n))$ .

A lower bound on sorting complexity:  $O(n \log(n))$ .

Complexity and Cost

Measuring Complexity:  
 big-oh  
 complexity classes

Cost of Algorithms

timing Python programs  
 counting flops:  
 floating-point operations

Maple's cost function

Summary + Assignments

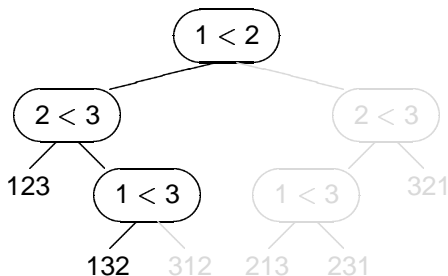
# Complexity of Sorting

independent of algorithm used

Minimal number of comparisons to sort  $n$  numbers?

#permutations equals  $n! = n \cdot (n - 1) \cdots 2 \cdot 1$ .

A sort computes a permutation to order the list.



$S(n)$  = minimal #comparisons. From the tree:  $n! \leq 2^{S(n)}$ .

Stirling:  $n! \approx \sqrt{2\pi n} \frac{n^n}{e^n} \Rightarrow O(\log(n!)) = O(n \log(n))$ .

A lower bound on sorting complexity:  $O(n \log(n))$ .

Complexity and Cost

Measuring Complexity:  
big-oh  
complexity classes

Cost of Algorithms

timing Python programs  
counting flops:  
floating-point operations

Maple's cost function

Summary + Assignments

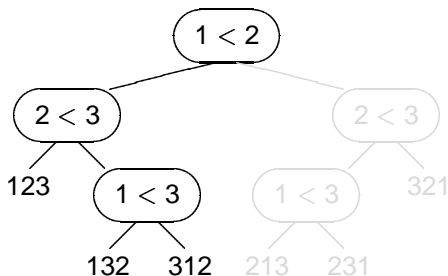
# Complexity of Sorting

independent of algorithm used

Minimal number of comparisons to sort  $n$  numbers?

#permutations equals  $n! = n \cdot (n - 1) \cdots 2 \cdot 1$ .

A sort computes a permutation to order the list.



$S(n)$  = minimal #comparisons. From the tree:  $n! \leq 2^{S(n)}$ .

Stirling:  $n! \approx \sqrt{2\pi n} \frac{n^n}{e^n} \Rightarrow O(\log(n!)) = O(n \log(n))$ .

A lower bound on sorting complexity:  $O(n \log(n))$ .

Complexity and Cost

Measuring Complexity:  
big-oh  
complexity classes

Cost of Algorithms

timing Python programs  
counting flops:  
floating-point operations

Maple's cost function

Summary + Assignments

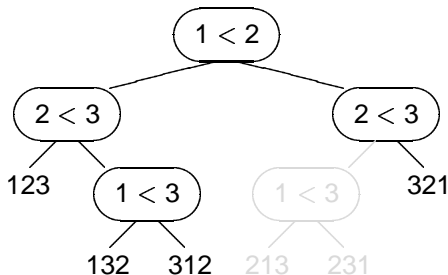
# Complexity of Sorting

independent of algorithm used

Minimal number of comparisons to sort  $n$  numbers?

#permutations equals  $n! = n \cdot (n - 1) \cdots 2 \cdot 1$ .

A sort computes a permutation to order the list.



$S(n)$  = minimal #comparisons. From the tree:  $n! \leq 2^{S(n)}$ .

Stirling:  $n! \approx \sqrt{2\pi n} \frac{n^n}{e^n} \Rightarrow O(\log(n!)) = O(n \log(n))$ .

A lower bound on sorting complexity:  $O(n \log(n))$ .

Complexity and Cost

Measuring Complexity:  
big-oh  
complexity classes

Cost of Algorithms

timing Python programs  
counting flops:  
floating-point operations

Maple's cost function

Summary + Assignments

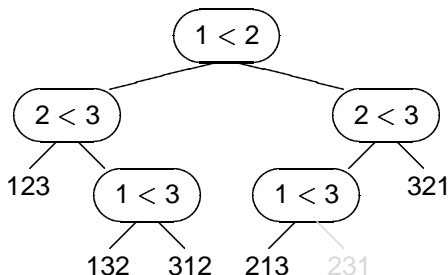
# Complexity of Sorting

independent of algorithm used

Minimal number of comparisons to sort  $n$  numbers?

#permutations equals  $n! = n \cdot (n - 1) \cdots 2 \cdot 1$ .

A sort computes a permutation to order the list.



$S(n)$  = minimal #comparisons. From the tree:  $n! \leq 2^{S(n)}$ .  
 Stirling:  $n! \approx \sqrt{2\pi n} \frac{n^n}{e^n} \Rightarrow O(\log(n!)) = O(n \log(n))$ .

A lower bound on sorting complexity:  $O(n \log(n))$ .

Complexity and Cost

Measuring Complexity:  
big-oh  
complexity classes

Cost of Algorithms

timing Python programs  
counting flops:  
floating-point operations

Maple's cost function

Summary + Assignments

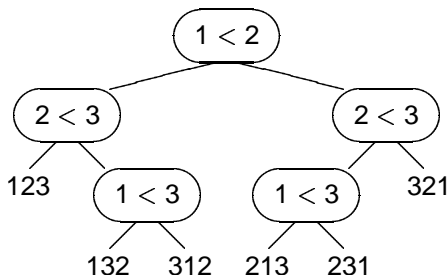
# Complexity of Sorting

independent of algorithm used

Minimal number of comparisons to sort  $n$  numbers?

#permutations equals  $n! = n \cdot (n - 1) \cdots 2 \cdot 1$ .

A sort computes a permutation to order the list.



$S(n)$  = minimal #comparisons. From the tree:  $n! \leq 2^{S(n)}$ .

Stirling:  $n! \approx \sqrt{2\pi n} \frac{n^n}{e^n} \Rightarrow O(\log(n!)) = O(n \log(n))$ .

A lower bound on sorting complexity:  $O(n \log(n))$ .

Complexity and Cost

Measuring Complexity:  
big-oh  
complexity classes

Cost of Algorithms

timing Python programs  
counting flops:  
floating-point operations

Maple's cost function

Summary + Assignments

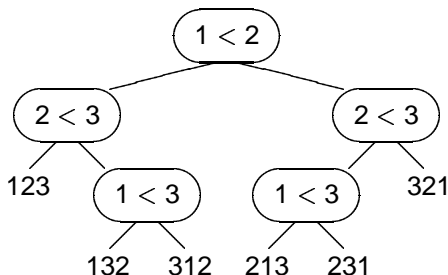
# Complexity of Sorting

independent of algorithm used

Minimal number of comparisons to sort  $n$  numbers?

#permutations equals  $n! = n \cdot (n - 1) \cdots 2 \cdot 1$ .

A sort computes a permutation to order the list.



$S(n)$  = minimal #comparisons. From the tree:  $n! \leq 2^{S(n)}$ .

Stirling:  $n! \approx \sqrt{2\pi n} \frac{n^n}{e^n} \Rightarrow O(\log(n!)) = O(n \log(n))$ .

A lower bound on sorting complexity:  $O(n \log(n))$ .

Complexity and Cost

Measuring Complexity:  
big-oh  
complexity classes

Cost of Algorithms

timing Python programs  
counting flops:  
floating-point operations

Maple's cost function

Summary + Assignments

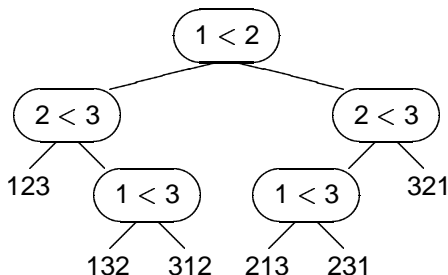
# Complexity of Sorting

independent of algorithm used

Minimal number of comparisons to sort  $n$  numbers?

#permutations equals  $n! = n \cdot (n - 1) \cdots 2 \cdot 1$ .

A sort computes a permutation to order the list.



$S(n)$  = minimal #comparisons. From the tree:  $n! \leq 2^{S(n)}$ .

Stirling:  $n! \approx \sqrt{2\pi n} \frac{n^n}{e^n} \Rightarrow O(\log(n!)) = O(n \log(n))$ .

A lower bound on sorting complexity:  $O(n \log(n))$ .

Complexity and Cost

Measuring Complexity:  
big-oh  
complexity classes

Cost of Algorithms

timing Python programs  
counting flops:  
floating-point operations

Maple's cost function

Summary + Assignments

## Complexity and Cost

Measuring Complexity: big-oh  
complexity classes

### Complexity and Cost

Measuring Complexity:  
big-oh

complexity classes

## Cost of Algorithms

timing Python programs  
counting flops: floating-point operations

### Cost of Algorithms

timing Python programs  
counting flops:  
floating-point operations

## Maple's cost function

### Maple's cost function

## Summary + Assignments

### Summary + Assignments

# Complexity Classes

We distinguish three big classes of complexity:

## P polynomial time

The problem can be solved in  $O(f(n))$ , where  $f(n)$  is a polynomial in  $n$ . Example: evaluate a polynomial.

## NP nondeterministic polynomial time

A solution to the problem can be verified in polynomial time. Example: root finding.

## #P counting problems

How many solutions does a problem have?

Example: determine #roots to nonlinear system.

Two problems belong to the same class if we can transform input/output in polynomial time.

How to win \$1,000,000: is  $P = NP$ ?

The halting problem is: *Given a program and a finite input, decide whether it will terminate.* **undecidable!**

## Complexity and Cost

Measuring Complexity:  
big-oh

complexity classes

## Cost of Algorithms

timing Python programs  
counting flops:  
floating-point operations

## Maple's cost function

## Summary + Assignments

# Complexity Classes

We distinguish three big classes of complexity:

**P** polynomial time

The problem can be solved in  $O(f(n))$ , where  $f(n)$  is a polynomial in  $n$ . Example: evaluate a polynomial.

**NP** nondeterministic polynomial time

A solution to the problem can be verified in polynomial time. Example: root finding.

**#P** counting problems

How many solutions does a problem have?

Example: determine #roots to nonlinear system.

Two problems belong to the same class if we can transform input/output in polynomial time.

How to win \$1,000,000: is  $P = NP$ ?

The halting problem is: *Given a program and a finite input, decide whether it will terminate.* **undecidable!**

Complexity and Cost

Measuring Complexity:  
big-oh  
complexity classes

Cost of Algorithms

timing Python programs  
counting flops:  
floating-point operations

Maple's cost function

Summary + Assignments

# Complexity Classes

We distinguish three big classes of complexity:

**P** polynomial time

The problem can be solved in  $O(f(n))$ , where  $f(n)$  is a polynomial in  $n$ . Example: evaluate a polynomial.

**NP** nondeterministic polynomial time

A solution to the problem can be verified in polynomial time. Example: root finding.

**#P** counting problems

How many solutions does a problem have?

Example: determine #roots to nonlinear system.

Two problems belong to the same class if we can transform input/output in polynomial time.

How to win \$1,000,000: is  $P = NP$ ?

The halting problem is: *Given a program and a finite input, decide whether it will terminate.* **undecidable!**

Complexity and Cost

Measuring Complexity:  
big-oh  
complexity classes

Cost of Algorithms

timing Python programs  
counting flops:  
floating-point operations

Maple's cost function

Summary + Assignments

# Complexity Classes

We distinguish three big classes of complexity:

**P** polynomial time

The problem can be solved in  $O(f(n))$ , where  $f(n)$  is a polynomial in  $n$ . Example: evaluate a polynomial.

**NP** nondeterministic polynomial time

A solution to the problem can be verified in polynomial time. Example: root finding.

**#P** counting problems

How many solutions does a problem have?

Example: determine #roots to nonlinear system.

Two problems belong to the same class if we can transform input/output in polynomial time.

How to win \$1,000,000: is  $P = NP$ ?

The halting problem is: *Given a program and a finite input, decide whether it will terminate.* **undecidable!**

## Complexity and Cost

Measuring Complexity:  
big-oh

complexity classes

## Cost of Algorithms

timing Python programs  
counting flops:  
floating-point operations

## Maple's cost function

## Summary + Assignments

# Complexity Classes

We distinguish three big classes of complexity:

**P** polynomial time

The problem can be solved in  $O(f(n))$ , where  $f(n)$  is a polynomial in  $n$ . Example: evaluate a polynomial.

**NP** nondeterministic polynomial time

A solution to the problem can be verified in polynomial time. Example: root finding.

**#P** counting problems

How many solutions does a problem have?

Example: determine #roots to nonlinear system.

Two problems belong to the same class if we can transform input/output in polynomial time.

How to win \$1,000,000: is  $P = NP$ ?

The halting problem is: *Given a program and a finite input, decide whether it will terminate.* **undecidable!**

Complexity and Cost

Measuring Complexity:  
big-oh  
complexity classes

Cost of Algorithms

timing Python programs  
counting flops:  
floating-point operations

Maple's cost function

Summary + Assignments

# Complexity Classes

We distinguish three big classes of complexity:

**P** polynomial time

The problem can be solved in  $O(f(n))$ , where  $f(n)$  is a polynomial in  $n$ . Example: evaluate a polynomial.

**NP** nondeterministic polynomial time

A solution to the problem can be verified in polynomial time. Example: root finding.

**#P** counting problems

How many solutions does a problem have?

Example: determine #roots to nonlinear system.

Two problems belong to the same class if we can transform input/output in polynomial time.

How to win \$1,000,000: is  $P = NP$ ?

The halting problem is: *Given a program and a finite input, decide whether it will terminate.* **undecidable!**

Complexity and Cost

Measuring Complexity:  
big-oh  
complexity classes

Cost of Algorithms

timing Python programs  
counting flops:  
floating-point operations

Maple's cost function

Summary + Assignments

# Complexity Classes

We distinguish three big classes of complexity:

**P** polynomial time

The problem can be solved in  $O(f(n))$ , where  $f(n)$  is a polynomial in  $n$ . Example: evaluate a polynomial.

**NP** nondeterministic polynomial time

A solution to the problem can be verified in polynomial time. Example: root finding.

**#P** counting problems

How many solutions does a problem have?

Example: determine #roots to nonlinear system.

Two problems belong to the same class if we can transform input/output in polynomial time.

How to win \$1,000,000: is  $P = NP$ ?

The halting problem is: *Given a program and a finite input, decide whether it will terminate.* **undecidable!**

Complexity and Cost

Measuring Complexity:  
big-oh

complexity classes

Cost of Algorithms

timing Python programs  
counting flops:  
floating-point operations

Maple's cost function

Summary + Assignments

# Complexity Classes

We distinguish three big classes of complexity:

**P** polynomial time

The problem can be solved in  $O(f(n))$ , where  $f(n)$  is a polynomial in  $n$ . Example: evaluate a polynomial.

**NP** nondeterministic polynomial time

A solution to the problem can be verified in polynomial time. Example: root finding.

**#P** counting problems

How many solutions does a problem have?

Example: determine #roots to nonlinear system.

Two problems belong to the same class if we can transform input/output in polynomial time.

How to win \$1,000,000: is  $P = NP$ ?

The halting problem is: *Given a program and a finite input, decide whether it will terminate.* **undecidable!**

Complexity and Cost

Measuring Complexity:  
big-oh  
complexity classes

Cost of Algorithms

timing Python programs  
counting flops:  
floating-point operations

Maple's cost function

Summary + Assignments

# Complexity Classes

We distinguish three big classes of complexity:

**P** polynomial time

The problem can be solved in  $O(f(n))$ , where  $f(n)$  is a polynomial in  $n$ . Example: evaluate a polynomial.

**NP** nondeterministic polynomial time

A solution to the problem can be verified in polynomial time. Example: root finding.

**#P** counting problems

How many solutions does a problem have?

Example: determine #roots to nonlinear system.

Two problems belong to the same class if we can transform input/output in polynomial time.

How to win \$1,000,000: is  $P = NP$ ?

The halting problem is: *Given a program and a finite input, decide whether it will terminate.* **undecidable!**

Complexity and Cost

Measuring Complexity:  
big-oh  
complexity classes

Cost of Algorithms

timing Python programs  
counting flops:  
floating-point operations

Maple's cost function

Summary + Assignments

# Complexity Classes

We distinguish three big classes of complexity:

**P** polynomial time

The problem can be solved in  $O(f(n))$ , where  $f(n)$  is a polynomial in  $n$ . Example: evaluate a polynomial.

**NP** nondeterministic polynomial time

A solution to the problem can be verified in polynomial time. Example: root finding.

**#P** counting problems

How many solutions does a problem have?

Example: determine #roots to nonlinear system.

Two problems belong to the same class if we can transform input/output in polynomial time.

How to win \$1,000,000: is  $P = NP$ ?

The halting problem is: *Given a program and a finite input, decide whether it will terminate.* **undecidable!**

Complexity and Cost

Measuring Complexity:  
big-oh  
complexity classes

Cost of Algorithms

timing Python programs  
counting flops:  
floating-point operations

Maple's cost function

Summary + Assignments

# Complexity Classes

We distinguish three big classes of complexity:

**P** polynomial time

The problem can be solved in  $O(f(n))$ , where  $f(n)$  is a polynomial in  $n$ . Example: evaluate a polynomial.

**NP** nondeterministic polynomial time

A solution to the problem can be verified in polynomial time. Example: root finding.

**#P** counting problems

How many solutions does a problem have?

Example: determine #roots to nonlinear system.

Two problems belong to the same class if we can transform input/output in polynomial time.

How to win \$1,000,000: is  $P = NP$ ?

The halting problem is: *Given a program and a finite input, decide whether it will terminate.* **undecidable!**

Complexity and Cost

Measuring Complexity:  
big-oh  
complexity classes

Cost of Algorithms

timing Python programs  
counting flops:  
floating-point operations

Maple's cost function

Summary + Assignments

# Complexity Classes

We distinguish three big classes of complexity:

**P** polynomial time

The problem can be solved in  $O(f(n))$ , where  $f(n)$  is a polynomial in  $n$ . Example: evaluate a polynomial.

**NP** nondeterministic polynomial time

A solution to the problem can be verified in polynomial time. Example: root finding.

**#P** counting problems

How many solutions does a problem have?

Example: determine #roots to nonlinear system.

Two problems belong to the same class if we can transform input/output in polynomial time.

How to win \$1,000,000: is  $P = NP$ ?

The halting problem is: *Given a program and a finite input, decide whether it will terminate.* **undecidable!**

Complexity and Cost

Measuring Complexity:  
big-oh  
complexity classes

Cost of Algorithms

timing Python programs  
counting flops:  
floating-point operations

Maple's cost function

Summary + Assignments

# Complexity Classes

We distinguish three big classes of complexity:

**P** polynomial time

The problem can be solved in  $O(f(n))$ , where  $f(n)$  is a polynomial in  $n$ . Example: evaluate a polynomial.

**NP** nondeterministic polynomial time

A solution to the problem can be verified in polynomial time. Example: root finding.

**#P** counting problems

How many solutions does a problem have?

Example: determine #roots to nonlinear system.

Two problems belong to the same class if we can transform input/output in polynomial time.

How to win \$1,000,000: is  $P = NP$ ?

The halting problem is: *Given a program and a finite input, decide whether it will terminate.* **undecidable!**

Complexity and Cost

Measuring Complexity:  
big-oh  
complexity classes

Cost of Algorithms

timing Python programs  
counting flops:  
floating-point operations

Maple's cost function

Summary + Assignments

# Outline

MCS 260 L-29

2 November 2007

## Complexity and Cost

Measuring Complexity: big-oh  
complexity classes

### Complexity and Cost

Measuring Complexity:  
big-oh  
complexity classes

## Cost of Algorithms

timing Python programs  
counting flops: floating-point operations

### Cost of Algorithms

timing Python programs  
counting flops:  
floating-point operations

## Maple's cost function

### Maple's cost function

## Summary + Assignments

### Summary + Assignments

# Cost of Algorithms

measuring efficiency and optimality

In our context, an algorithm = a Python program.

Static cost analysis (analyze source code):

1. count the number of arithmetical operations;
2. estimate the size of the used memory;
3. identify resource intensive tasks.

Dynamic cost analysis (time the program):

1. apply Unix command `time`;
2. use module `time` inside Python code;
3. count floating-point operations (flops).

Pushing a program to its limits is a *stress test*.

## Complexity and Cost

Measuring Complexity:  
big-oh  
complexity classes

## Cost of Algorithms

timing Python programs  
counting flops:  
floating-point operations

## Maple's cost function

## Summary + Assignments

# Cost of Algorithms

measuring efficiency and optimality

In our context, an algorithm = a Python program.

Static cost analysis (analyze source code):

1. count the number of arithmetical operations;
2. estimate the size of the used memory;
3. identify resource intensive tasks.

Dynamic cost analysis (time the program):

1. apply Unix command `time`;
2. use module `time` inside Python code;
3. count floating-point operations (flops).

Pushing a program to its limits is a *stress test*.

## Complexity and Cost

Measuring Complexity:  
big-oh  
complexity classes

## Cost of Algorithms

timing Python programs  
counting flops:  
floating-point operations

## Maple's cost function

## Summary + Assignments

# Cost of Algorithms

measuring efficiency and optimality

In our context, an algorithm = a Python program.

Static cost analysis (analyze source code):

1. count the number of arithmetical operations;
2. estimate the size of the used memory;
3. identify resource intensive tasks.

Dynamic cost analysis (time the program):

1. apply Unix command `time`;
2. use module `time` inside Python code;
3. count floating-point operations (flops).

Pushing a program to its limits is a *stress test*.

Complexity and  
Cost

Measuring Complexity:  
big-oh  
complexity classes

Cost of Algorithms

timing Python programs  
counting flops:  
floating-point operations

Maple's cost  
function

Summary +  
Assignments

# Cost of Algorithms

measuring efficiency and optimality

In our context, an algorithm = a Python program.

Static cost analysis (analyze source code):

1. count the number of arithmetical operations;
2. estimate the size of the used memory;
3. identify resource intensive tasks.

Dynamic cost analysis (time the program):

1. apply Unix command `time`;
2. use module `time` inside Python code;
3. count floating-point operations (flops).

Pushing a program to its limits is a *stress test*.

## Complexity and Cost

Measuring Complexity:  
big-oh  
complexity classes

## Cost of Algorithms

timing Python programs  
counting flops:  
floating-point operations

## Maple's cost function

## Summary + Assignments

# Cost of Algorithms

measuring efficiency and optimality

In our context, an algorithm = a Python program.

Static cost analysis (analyze source code):

1. count the number of arithmetical operations;
2. estimate the size of the used memory;
3. identify resource intensive tasks.

Dynamic cost analysis (time the program):

1. apply Unix command `time`;
2. use module `time` inside Python code;
3. count floating-point operations (flops).

Pushing a program to its limits is a *stress test*.

## Complexity and Cost

Measuring Complexity:  
big-oh  
complexity classes

## Cost of Algorithms

timing Python programs  
counting flops:  
floating-point operations

## Maple's cost function

## Summary + Assignments

# Cost of Algorithms

measuring efficiency and optimality

In our context, an algorithm = a Python program.

Static cost analysis (analyze source code):

1. count the number of arithmetical operations;
2. estimate the size of the used memory;
3. identify resource intensive tasks.

Dynamic cost analysis (time the program):

1. apply Unix command `time`;
2. use module `time` inside Python code;
3. count floating-point operations (flops).

Pushing a program to its limits is a *stress test*.

## Complexity and Cost

Measuring Complexity:  
big-oh  
complexity classes

## Cost of Algorithms

timing Python programs  
counting flops:  
floating-point operations

## Maple's cost function

## Summary + Assignments

# Cost of Algorithms

measuring efficiency and optimality

In our context, an algorithm = a Python program.

Static cost analysis (analyze source code):

1. count the number of arithmetical operations;
2. estimate the size of the used memory;
3. identify resource intensive tasks.

Dynamic cost analysis (time the program):

1. apply Unix command `time`;
2. use module `time` inside Python code;
3. count floating-point operations (flops).

Pushing a program to its limits is a *stress test*.

## Complexity and Cost

Measuring Complexity:  
big-oh  
complexity classes

## Cost of Algorithms

timing Python programs  
counting flops:  
floating-point operations

## Maple's cost function

## Summary + Assignments

# Cost of Algorithms

measuring efficiency and optimality

In our context, an algorithm = a Python program.

Static cost analysis (analyze source code):

1. count the number of arithmetical operations;
2. estimate the size of the used memory;
3. identify resource intensive tasks.

Dynamic cost analysis (time the program):

1. apply Unix command `time`;
2. use module `time` inside Python code;
3. count floating-point operations (flops).

Pushing a program to its limits is a *stress test*.

## Complexity and Cost

Measuring Complexity:  
big-oh  
complexity classes

## Cost of Algorithms

timing Python programs  
counting flops:  
floating-point operations

## Maple's cost function

## Summary + Assignments

# Timing Programs

time on unix like systems

The Unix `time` command runs as

```
$ time < program name >
```

and returns three times: real, user, and system time.

The real time is the wall time. User time is also called CPU time, and system time measures the overhead.

```
$ time python timetosort.py 100000
sorting 100000 floats took 0.350 seconds
```

```
real      0m1.639s
user      0m1.530s
sys       0m0.030s
```

The last three lines are the output of `time`.

The time reported by the program is specifically for `sort`.

Complexity and Cost

Measuring Complexity:  
big-oh  
complexity classes

Cost of Algorithms

timing Python programs  
counting flops:  
floating-point operations

Maple's cost function

Summary + Assignments

# Timing Programs

time on unix like systems

The Unix `time` command runs as

```
$ time < program name >
```

and returns three times: real, user, and system time.

The real time is the wall time. User time is also called CPU time, and system time measures the overhead.

```
$ time python timetosort.py 100000
sorting 100000 floats took 0.350 seconds
```

```
real      0m1.639s
user      0m1.530s
sys       0m0.030s
```

The last three lines are the output of `time`.

The time reported by the program is specifically for `sort`.

Complexity and Cost

Measuring Complexity:  
big-oh  
complexity classes

Cost of Algorithms

timing Python programs  
counting flops:  
floating-point operations

Maple's cost function

Summary + Assignments

# Timing Programs

time on unix like systems

The Unix `time` command runs as

```
$ time < program name >
```

and returns three times: real, user, and system time.

The real time is the wall time. User time is also called CPU time, and system time measures the overhead.

```
$ time python timetosort.py 100000
sorting 100000 floats took 0.350 seconds
```

```
real      0m1.639s
user      0m1.530s
sys       0m0.030s
```

The last three lines are the output of `time`.

The time reported by the program is specifically for `sort`.

Complexity and Cost

Measuring Complexity:  
big-oh  
complexity classes

Cost of Algorithms

timing Python programs  
counting flops:  
floating-point operations

Maple's cost function

Summary + Assignments

# Timing Programs

time on unix like systems

The Unix `time` command runs as

```
$ time < program name >
```

and returns three times: real, user, and system time.

The real time is the wall time. User time is also called CPU time, and system time measures the overhead.

```
$ time python timetosort.py 100000
sorting 100000 floats took 0.350 seconds
```

```
real      0m1.639s
user      0m1.530s
sys       0m0.030s
```

The last three lines are the output of `time`.

The time reported by the program is specifically for `sort`.

[Complexity and Cost](#)

Measuring Complexity:  
big-oh  
complexity classes

[Cost of Algorithms](#)

timing Python programs  
counting flops:  
floating-point operations

[Maple's cost function](#)

[Summary + Assignments](#)

# Using the time Module

to measure performance of Python code

To find the bottleneck in a program, one can measure the time spent executing a piece of code.

General template:

```
import time
< initialization >
start_time = time.clock()
< code to be timed >
stop_time = time.clock()
elapsed = stop_time - start_time
print 'time: %.3f seconds ' % elapsed
```

Applications:

1. find where code spend most of its time;
2. compare various algorithms experimentally.

Complexity and Cost

Measuring Complexity:  
big-oh  
complexity classes

Cost of Algorithms

timing Python programs  
counting flops:  
floating-point operations

Maple's cost function

Summary + Assignments

# Using the time Module

to measure performance of Python code

To find the bottleneck in a program, one can measure the time spent executing a piece of code.

General template:

```
import time
< initialization >
start_time = time.clock()
< code to be timed >
stop_time = time.clock()
elapsed = stop_time - start_time
print 'time: %.3f seconds ' % elapsed
```

Applications:

1. find where code spend most of its time;
2. compare various algorithms experimentally.

Complexity and Cost

Measuring Complexity:  
big-oh  
complexity classes

Cost of Algorithms

timing Python programs  
counting flops:  
floating-point operations

Maple's cost function

Summary + Assignments

# Using the time Module

to measure performance of Python code

To find the bottleneck in a program, one can measure the time spent executing a piece of code.

General template:

```
import time
< initialization >
start_time = time.clock()
< code to be timed >
stop_time = time.clock()
elapsed = stop_time - start_time
print 'time: %.3f seconds ' % elapsed
```

Applications:

1. find where code spend most of its time;
2. compare various algorithms experimentally.

Complexity and Cost

Measuring Complexity:  
big-oh  
complexity classes

Cost of Algorithms

timing Python programs  
counting flops:  
floating-point operations

Maple's cost function

Summary + Assignments

# Using the time Module

to measure performance of Python code

To find the bottleneck in a program, one can measure the time spent executing a piece of code.

General template:

```
import time
< initialization >
start_time = time.clock()
< code to be timed >
stop_time = time.clock()
elapsed = stop_time - start_time
print 'time: %.3f seconds ' % elapsed
```

Applications:

1. find where code spend most of its time;
2. compare various algorithms experimentally.

Complexity and Cost

Measuring Complexity:  
big-oh  
complexity classes

Cost of Algorithms

timing Python programs  
counting flops:  
floating-point operations

Maple's cost function

Summary + Assignments

# Timing the sort

file `timetosort.py`

The start of the program:

```
# L-29 MCS 260 Fri 2 Nov 2007 : timetosort
#
# This program illustrates how to time the
# sort of lists. The optional command line
# argument is the length of the list to sort.
```

```
import time
import sys
from random import gauss

if len(sys.argv) < 2:
    n = input('Give #elements : ')
else:
    n = int(sys.argv[1])
```

`sys.argv[1]` we capture a command line argument:  
the 100000 in `$ python timetosort 100000`.

[Complexity and Cost](#)

Measuring Complexity:  
big-oh  
complexity classes

[Cost of Algorithms](#)

timing Python programs  
counting flops:  
floating-point operations

[Maple's cost function](#)

[Summary + Assignments](#)

There tasks:

1. generate a list of  $n$  random floats
2. apply `L.sort` between `time.clock()`'s
3. print the elapsed time

The code continued:

```
r = range(0,n)
L = map(lambda i: gauss(0,1),r)
start_time = time.clock()
L.sort()
stop_time = time.clock()
elapsed = stop_time - start_time
print 'sorting %d floats' % n \
      + ' took %.3f seconds ' % elapsed
```

[Complexity and Cost](#)

Measuring Complexity:  
big-oh  
complexity classes

[Cost of Algorithms](#)

timing Python programs  
counting flops:  
floating-point operations

[Maple's cost function](#)

[Summary + Assignments](#)

# timetosort continued

There tasks:

1. generate a list of  $n$  random floats
2. apply `L.sort` between `time.clock()`'s
3. print the elapsed time

The code continued:

```
r = range(0,n)
L = map(lambda i: gauss(0,1),r)
start_time = time.clock()
L.sort()
stop_time = time.clock()
elapsed = stop_time - start_time
print 'sorting %d floats' % n \
      + ' took %.3f seconds ' % elapsed
```

[Complexity and Cost](#)

Measuring Complexity:  
big-oh  
complexity classes

[Cost of Algorithms](#)

timing Python programs  
counting flops:  
floating-point operations

[Maple's cost function](#)

[Summary + Assignments](#)

# timetosort continued

There tasks:

1. generate a list of  $n$  random floats
2. apply `L.sort` between `time.clock()`'s
3. print the elapsed time

The code continued:

```
r = range(0,n)
L = map(lambda i: gauss(0,1),r)
start_time = time.clock()
L.sort()
stop_time = time.clock()
elapsed = stop_time - start_time
print 'sorting %d floats' % n \
      + ' took %.3f seconds ' % elapsed
```

[Complexity and Cost](#)

Measuring Complexity:  
big-oh  
complexity classes

[Cost of Algorithms](#)

timing Python programs  
counting flops:  
floating-point operations

[Maple's cost function](#)

[Summary + Assignments](#)

There tasks:

1. generate a list of  $n$  random floats
2. apply `L.sort` between `time.clock()`'s
3. print the elapsed time

The code continued:

```
r = range(0,n)
L = map(lambda i: gauss(0,1),r)
start_time = time.clock()
L.sort()
stop_time = time.clock()
elapsed = stop_time - start_time
print 'sorting %d floats' % n \
      + ' took %.3f seconds ' % elapsed
```

[Complexity and Cost](#)

Measuring Complexity:  
big-oh  
complexity classes

[Cost of Algorithms](#)

timing Python programs  
counting flops:  
floating-point operations

[Maple's cost function](#)

[Summary + Assignments](#)

# timetosort continued

There tasks:

1. generate a list of  $n$  random floats
2. apply `L.sort` between `time.clock()`'s
3. print the elapsed time

The code continued:

```
r = range(0,n)
L = map(lambda i: gauss(0,1),r)
start_time = time.clock()
L.sort()
stop_time = time.clock()
elapsed = stop_time - start_time
print 'sorting %d floats' % n \
      + ' took %.3f seconds ' % elapsed
```

[Complexity and Cost](#)

Measuring Complexity:  
big-oh  
complexity classes

[Cost of Algorithms](#)

timing Python programs  
counting flops:  
floating-point operations

[Maple's cost function](#)

[Summary + Assignments](#)

There tasks:

1. generate a list of  $n$  random floats
2. apply `L.sort` between `time.clock()`'s
3. print the elapsed time

The code continued:

```
r = range(0,n)
L = map(lambda i: gauss(0,1),r)
start_time = time.clock()
L.sort()
stop_time = time.clock()
elapsed = stop_time - start_time
print 'sorting %d floats' % n \
      + ' took %.3f seconds ' % elapsed
```

[Complexity and Cost](#)

Measuring Complexity:  
big-oh  
complexity classes

[Cost of Algorithms](#)

timing Python programs  
counting flops:  
floating-point operations

[Maple's cost function](#)

[Summary + Assignments](#)

# Running the Timings

showing the edited output

## Running on a not so new PowerBook G4:

```

sorting 100000 floats took 0.350 seconds
real 0m1.639s  user 0m1.530s  sys  0m0.030s

sorting 200000 floats took 0.790 seconds
real 0m3.264s  user 0m3.130s  sys  0m0.060s

sorting 400000 floats took 1.770 seconds
real 0m6.591s  user 0m6.310s  sys  0m0.140s

sorting 800000 floats took 3.980 seconds
real 0m13.689s user 0m13.080s  sys  0m0.250s

sorting 1600000 floats took 8.900 seconds
real 0m30.954s user 0m27.360s  sys  0m0.540s

sorting 3200000 floats took 19.660 seconds
real 1m0.447s  user 0m56.980s  sys  0m0.980s

```

[Complexity and Cost](#)

Measuring Complexity:  
big-oh  
complexity classes

[Cost of Algorithms](#)

timing Python programs  
counting flops:  
floating-point operations

[Maple's cost function](#)

[Summary + Assignments](#)

# Running the Timings

showing the edited output

## Running on a not so new PowerBook G4:

```

sorting 100000 floats took 0.350 seconds
real 0m1.639s  user 0m1.530s  sys  0m0.030s
sorting 200000 floats took 0.790 seconds
real 0m3.264s  user 0m3.130s  sys  0m0.060s

sorting 400000 floats took 1.770 seconds
real 0m6.591s  user 0m6.310s  sys  0m0.140s

sorting 800000 floats took 3.980 seconds
real 0m13.689s user 0m13.080s  sys  0m0.250s

sorting 1600000 floats took 8.900 seconds
real 0m30.954s user 0m27.360s  sys  0m0.540s

sorting 3200000 floats took 19.660 seconds
real 1m0.447s  user 0m56.980s  sys  0m0.980s

```

[Complexity and Cost](#)

Measuring Complexity:  
big-oh  
complexity classes

[Cost of Algorithms](#)

timing Python programs  
counting flops:  
floating-point operations

[Maple's cost function](#)

[Summary + Assignments](#)

# Running the Timings

showing the edited output

## Running on a not so new PowerBook G4:

```

sorting 100000 floats took 0.350 seconds
real 0m1.639s  user 0m1.530s  sys  0m0.030s

sorting 200000 floats took 0.790 seconds
real 0m3.264s  user 0m3.130s  sys  0m0.060s

sorting 400000 floats took 1.770 seconds
real 0m6.591s  user 0m6.310s  sys  0m0.140s

sorting 800000 floats took 3.980 seconds
real 0m13.689s user 0m13.080s  sys  0m0.250s

sorting 1600000 floats took 8.900 seconds
real 0m30.954s user 0m27.360s  sys  0m0.540s

sorting 3200000 floats took 19.660 seconds
real 1m0.447s  user 0m56.980s  sys  0m0.980s

```

[Complexity and Cost](#)

Measuring Complexity:  
big-oh  
complexity classes

[Cost of Algorithms](#)

timing Python programs  
counting flops:  
floating-point operations

[Maple's cost function](#)

[Summary + Assignments](#)

# Running the Timings

showing the edited output

## Running on a not so new PowerBook G4:

```

sorting 100000 floats took 0.350 seconds
real 0m1.639s user 0m1.530s sys 0m0.030s
sorting 200000 floats took 0.790 seconds
real 0m3.264s user 0m3.130s sys 0m0.060s
sorting 400000 floats took 1.770 seconds
real 0m6.591s user 0m6.310s sys 0m0.140s
sorting 800000 floats took 3.980 seconds
real 0m13.689s user 0m13.080s sys 0m0.250s
sorting 1600000 floats took 8.900 seconds
real 0m30.954s user 0m27.360s sys 0m0.540s
sorting 3200000 floats took 19.660 seconds
real 1m0.447s user 0m56.980s sys 0m0.980s

```

[Complexity and Cost](#)

Measuring Complexity:  
big-oh  
complexity classes

[Cost of Algorithms](#)

timing Python programs  
counting flops:  
floating-point operations

[Maple's cost function](#)

[Summary + Assignments](#)

# Running the Timings

showing the edited output

## Running on a not so new PowerBook G4:

```
sorting 100000 floats took 0.350 seconds
real 0m1.639s user 0m1.530s sys 0m0.030s
sorting 200000 floats took 0.790 seconds
real 0m3.264s user 0m3.130s sys 0m0.060s
sorting 400000 floats took 1.770 seconds
real 0m6.591s user 0m6.310s sys 0m0.140s
sorting 800000 floats took 3.980 seconds
real 0m13.689s user 0m13.080s sys 0m0.250s
sorting 1600000 floats took 8.900 seconds
real 0m30.954s user 0m27.360s sys 0m0.540s
sorting 3200000 floats took 19.660 seconds
real 1m0.447s user 0m56.980s sys 0m0.980s
```

[Complexity and Cost](#)

Measuring Complexity:  
big-oh  
complexity classes

[Cost of Algorithms](#)

timing Python programs  
counting flops:  
floating-point operations

[Maple's cost function](#)

[Summary + Assignments](#)

# Running the Timings

showing the edited output

## Running on a not so new PowerBook G4:

```
sorting 100000 floats took 0.350 seconds
real 0m1.639s user 0m1.530s sys 0m0.030s
sorting 200000 floats took 0.790 seconds
real 0m3.264s user 0m3.130s sys 0m0.060s
sorting 400000 floats took 1.770 seconds
real 0m6.591s user 0m6.310s sys 0m0.140s
sorting 800000 floats took 3.980 seconds
real 0m13.689s user 0m13.080s sys 0m0.250s
sorting 1600000 floats took 8.900 seconds
real 0m30.954s user 0m27.360s sys 0m0.540s
sorting 3200000 floats took 19.660 seconds
real 1m0.447s user 0m56.980s sys 0m0.980s
```

[Complexity and Cost](#)

Measuring Complexity:  
big-oh  
complexity classes

[Cost of Algorithms](#)

timing Python programs  
counting flops:  
floating-point operations

[Maple's cost function](#)

[Summary + Assignments](#)

# Outline

MCS 260 L-29

2 November 2007

## Complexity and Cost

Measuring Complexity: big-oh  
complexity classes

### Complexity and Cost

Measuring Complexity:  
big-oh  
complexity classes

## Cost of Algorithms

timing Python programs  
counting flops: floating-point operations

### Cost of Algorithms

timing Python programs  
counting flops:  
floating-point operations

### Maple's cost function

Maple's cost function

### Summary + Assignments

Summary + Assignments

# Counting Flops

## floating-point operations

A flop is short for floating-point operation.  
In scientific computation, the cost analysis is often measured in flops.

Note: before version 6, MATLAB had a `flops` command.

Using Object Oriented Programming:

1. `class FlopFloat` inherits from `float`;
2. every object stores its `#flops`:  
these are the flops used to compute the number;
3. the overloaded arithmetical operators  
count also the flops for each result.

### Complexity and Cost

Measuring Complexity:  
big-oh  
complexity classes

### Cost of Algorithms

timing Python programs  
counting flops:  
floating-point operations

### Maple's cost function

### Summary + Assignments

# Counting Flops

## floating-point operations

A flop is short for floating-point operation.

In scientific computation, the cost analysis is often measured in flops.

Note: before version 6, MATLAB had a `flops` command.

Using Object Oriented Programming:

1. `class FlopFloat` inherits from `float`;
2. every object stores its `#flops`:  
these are the flops used to compute the number;
3. the overloaded arithmetical operators  
count also the flops for each result.

### Complexity and Cost

Measuring Complexity:  
big-oh  
complexity classes

### Cost of Algorithms

timing Python programs  
counting flops:  
floating-point operations

### Maple's cost function

### Summary + Assignments

# Counting Flops

## floating-point operations

A flop is short for floating-point operation.

In scientific computation, the cost analysis is often measured in flops.

Note: before version 6, MATLAB had a `flops` command.

Using Object Oriented Programming:

1. `class FlopFloat` inherits from `float`;
2. every object stores its `#flops`:  
these are the flops used to compute the number;
3. the overloaded arithmetical operators  
count also the flops for each result.

### Complexity and Cost

Measuring Complexity:  
big-oh  
complexity classes

### Cost of Algorithms

timing Python programs  
counting flops:  
floating-point operations

### Maple's cost function

### Summary + Assignments

# Counting Flops

## floating-point operations

A flop is short for floating-point operation.

In scientific computation, the cost analysis is often measured in flops.

Note: before version 6, MATLAB had a `flops` command.

Using Object Oriented Programming:

1. `class FlopFloat` inherits from `float`;
2. every object stores its `#flops`:  
these are the flops used to compute the number;
3. the overloaded arithmetical operators  
count also the flops for each result.

### Complexity and Cost

Measuring Complexity:  
big-oh  
complexity classes

### Cost of Algorithms

timing Python programs  
counting flops:  
floating-point operations

### Maple's cost function

### Summary + Assignments

# The class FlopFloat

MCS 260 L-29

2 November 2007

```
# L-29 MCS 260 Fri 2 Nov 2007 : flopfloats
#
# To analyze the cost of an algorithm,
# we count the number of operations.
```

```
class FlopFloat(float):
    """
    An object of the class FlopFloat records
    the number of floating-point operations
    executed to compute the float.
    """

    def __init__(self, f=0.0, n=0):
        "constructor for a flopfloat"
        self.float = f
        self.flops = n
```

Complexity and Cost

Measuring Complexity:  
big-oh  
complexity classes

Cost of Algorithms

timing Python programs  
counting flops:  
floating-point operations

Maple's cost function

Summary + Assignments

# The class FlopFloat

MCS 260 L-29

2 November 2007

```
# L-29 MCS 260 Fri 2 Nov 2007 : flopfloats
#
# To analyze the cost of an algorithm,
# we count the number of operations.
```

```
class FlopFloat(float):
    """
    An object of the class FlopFloat records
    the number of floating-point operations
    executed to compute the float.
    """

    def __init__(self, f=0.0, n=0):
        "constructor for a flopfloat"
        self.float = f
        self.flops = n
```

Complexity and  
Cost

Measuring Complexity:  
big-oh  
complexity classes

Cost of Algorithms

timing Python programs  
counting flops:  
floating-point operations

Maple's cost  
function

Summary +  
Assignments

# Overloading Arithmetical Operators

for example, the addition

The other may be an ordinary float:

```
def __add__(self, *other):
    "returns the result of the addition"
    if isinstance(other[0], FlopFloat):
        sum = FlopFloat(self.float + other[0].float)
        sum.flops = self.flops + other[0].flops + 1
    else: # treat other just as ordinary number
        sum = FlopFloat(self.float + other[0])
        sum.flops = self.flops + 1
    return sum
```

Other arithmetical operations are defined similarly.

We store the class definition in the file `flopfloats.py` and import it as a module.

Complexity and Cost

Measuring Complexity:  
big-oh  
complexity classes

Cost of Algorithms

timing Python programs  
counting flops:  
floating-point operations

Maple's cost  
function

Summary +  
Assignments

# Overloading Arithmetical Operators

for example, the addition

The other may be an ordinary float:

```
def __add__(self, *other):
    "returns the result of the addition"
    if isinstance(other[0], FlopFloat):
        sum = FlopFloat(self.float + other[0].float)
        sum.flops = self.flops + other[0].flops + 1
    else: # treat other just as ordinary number
        sum = FlopFloat(self.float + other[0])
        sum.flops = self.flops + 1
    return sum
```

Other arithmetical operations are defined similarly.

We store the class definition in the file `flopfloats.py` and import it as a module.

Complexity and Cost

Measuring Complexity:  
big-oh  
complexity classes

Cost of Algorithms

timing Python programs  
counting flops:  
floating-point operations

Maple's cost  
function

Summary +  
Assignments

# A Simple Test of Use

summing numbers in `flopsum.py`

```
# L-29 MCS 260 Fri 2 Nov 2007 : flopsum
#
# We use FlopFloats to count the number
# of operations when summing n floats.

from flopfloats import *
from random import gauss
print 'counting flops in a sum' + \
      ' of n floating-point numbers'
n = input('Give n : ')
sum = FlopFloat()
for i in range(0,n):
    r = FlopFloat(gauss(0,1))
    sum = sum + r
print 'sum = ' + str(sum) + \
      ' #flops is %d' % sum.flops
```

## Complexity and Cost

Measuring Complexity:  
big-oh  
complexity classes

## Cost of Algorithms

timing Python programs  
counting flops:  
floating-point operations

## Maple's cost function

## Summary + Assignments

# A Simple Test of Use

summing numbers in `flopsum.py`

```
# L-29 MCS 260 Fri 2 Nov 2007 : flopsum
#
# We use FlopFloats to count the number
# of operations when summing n floats.

from flopfloats import *
from random import gauss
print 'counting flops in a sum' + \
      ' of n floating-point numbers'
n = input('Give n : ')
sum = FlopFloat()
for i in range(0,n):
    r = FlopFloat(gauss(0,1))
    sum = sum + r
print 'sum = ' + str(sum) + \
      ' #flops is %d' % sum.flops
```

## Complexity and Cost

Measuring Complexity:  
big-oh  
complexity classes

## Cost of Algorithms

timing Python programs  
counting flops:  
floating-point operations

## Maple's cost function

## Summary + Assignments

# A Simple Test of Use

summing numbers in `flopsum.py`

```
# L-29 MCS 260 Fri 2 Nov 2007 : flopsum
#
# We use FlopFloats to count the number
# of operations when summing n floats.

from flopfloats import *
from random import gauss
print 'counting flops in a sum' + \
      ' of n floating-point numbers'
n = input('Give n : ')
sum = FlopFloat()
for i in range(0,n):
    r = FlopFloat(gauss(0,1))
    sum = sum + r
print 'sum = ' + str(sum) + \
      ' #flops is %d' % sum.flops
```

## Complexity and Cost

Measuring Complexity:  
big-oh  
complexity classes

## Cost of Algorithms

timing Python programs  
counting flops:  
floating-point operations

## Maple's cost function

## Summary + Assignments

# A Simple Test of Use

summing numbers in `flopsum.py`

```
# L-29 MCS 260 Fri 2 Nov 2007 : flopsum
#
# We use FlopFloats to count the number
# of operations when summing n floats.

from flopfloats import *
from random import gauss
print 'counting flops in a sum' + \
      ' of n floating-point numbers'
n = input('Give n : ')
sum = FlopFloat()
for i in range(0,n):
    r = FlopFloat(gauss(0,1))
    sum = sum + r
print 'sum = ' + str(sum) + \
      ' #flops is %d' % sum.flops
```

MCS 260 L-29

2 November 2007

Complexity and  
Cost

Measuring Complexity:  
big-oh  
complexity classes

Cost of Algorithms

timing Python programs  
counting flops:  
floating-point operations

Maple's cost  
function

Summary +  
Assignments

# Running flopsum

At the command prompt \$:

```
$ python flopsum.py
counting flops in a sum of n floating-point numbers
Give n : 100
sum = -2.8354e+00 #flops is 100
```

It works!

A less trivial application:

```
# L-29 MCS 260 Fri 2 Nov 2007 : floppoly
#
# We use FlopFloats to count the number
# of operations to evaluate a polynomial
# of degree d with random coefficients.
```

Complexity and Cost

Measuring Complexity:  
big-oh  
complexity classes

Cost of Algorithms  
timing Python programs

counting flops:  
floating-point operations

Maple's cost function

Summary + Assignments

# Running flopsum

At the command prompt \$:

```
$ python flopsum.py
counting flops in a sum of n floating-point numbers
Give n : 100
sum = -2.8354e+00 #flops is 100
```

It works!

A less trivial application:

```
# L-29 MCS 260 Fri 2 Nov 2007 : floppoly
#
# We use FlopFloats to count the number
# of operations to evaluate a polynomial
# of degree d with random coefficients.
```

# Evaluation of Polynomials

the program floppoly.py

```
from floppoly import *
from random import gauss

print '#flops to evaluate a polynomial'
d = input('Give degree : ')
x = FlopFloat(gauss(0,1))
sum = FlopFloat()
for i in range(0,d+1):
    r = FlopFloat(gauss(0,1))
    for j in range(0,i):
        r = r*x
    sum = sum + r
print 'evaluating a polynomial of degree '+ \
      '%d \ntakes %d flops' % (d,sum.flops)
```

## Complexity and Cost

Measuring Complexity:  
big-oh  
complexity classes

## Cost of Algorithms

timing Python programs  
counting flops:  
floating-point operations

## Maple's cost function

## Summary + Assignments

# Evaluation of Polynomials

the program floppoly.py

```
from floppoly import *
from random import gauss

print '#flops to evaluate a polynomial'
d = input('Give degree : ')
x = FlopFloat(gauss(0,1))
sum = FlopFloat()
for i in range(0,d+1):
    r = FlopFloat(gauss(0,1))
    for j in range(0,i):
        r = r*x
    sum = sum + r
print 'evaluating a polynomial of degree '+ \
      '%d \ntakes %d flops' % (d,sum.flops)
```

## Complexity and Cost

Measuring Complexity:  
big-oh  
complexity classes

## Cost of Algorithms

timing Python programs  
counting flops:  
floating-point operations

## Maple's cost function

## Summary + Assignments

# Evaluation of Polynomials

the program floppoly.py

```

from floppoly import *
from random import gauss

print '#flops to evaluate a polynomial'
d = input('Give degree : ')
x = FlopFloat(gauss(0,1))
sum = FlopFloat()
for i in range(0,d+1):
    r = FlopFloat(gauss(0,1))
    for j in range(0,i):
        r = r*x
    sum = sum + r
print 'evaluating a polynomial of degree '+ \
      '%d \ntakes %d flops' % (d,sum.flops)

```

## Complexity and Cost

Measuring Complexity:  
big-oh  
complexity classes

## Cost of Algorithms

timing Python programs  
counting flops:  
floating-point operations

## Maple's cost function

## Summary + Assignments

# Running floppoly

## Evaluating polynomials with random coefficients:

```
$ python floppoly.py
#flops to evaluate a polynomial
Give degree : 10
evaluating a polynomial of degree 10
takes 66 flops
```

```
$ python floppoly.py
#flops to evaluate a polynomial
Give degree : 20
evaluating a polynomial of degree 20
takes 231 flops
```

The #flops is not linear in the degree.

### Complexity and Cost

Measuring Complexity:  
big-oh  
complexity classes

### Cost of Algorithms

timing Python programs  
counting flops:  
floating-point operations

### Maple's cost function

### Summary + Assignments

# Running floppoly

## Evaluating polynomials with random coefficients:

```
$ python floppoly.py
#flops to evaluate a polynomial
Give degree : 10
evaluating a polynomial of degree 10
takes 66 flops
```

```
$ python floppoly.py
#flops to evaluate a polynomial
Give degree : 20
evaluating a polynomial of degree 20
takes 231 flops
```

The #flops is not linear in the degree.

### Complexity and Cost

Measuring Complexity:  
big-oh  
complexity classes

### Cost of Algorithms

timing Python programs  
counting flops:  
floating-point operations

### Maple's cost function

### Summary + Assignments

# Running floppoly

## Evaluating polynomials with random coefficients:

```
$ python floppoly.py
#flops to evaluate a polynomial
Give degree : 10
evaluating a polynomial of degree 10
takes 66 flops
```

```
$ python floppoly.py
#flops to evaluate a polynomial
Give degree : 20
evaluating a polynomial of degree 20
takes 231 flops
```

The #flops is not linear in the degree.

### Complexity and Cost

Measuring Complexity:  
big-oh  
complexity classes

### Cost of Algorithms

timing Python programs  
counting flops:  
floating-point operations

### Maple's cost function

### Summary + Assignments

# The cost function in Maple

Running Maple 11 at the command prompt:

```
|\^/|      Maple 11 (IBM INTEL LINUX)
._|\\|    |/_|. Copyright (c) Maplesoft, a division of Water
 \ MAPLE / All rights reserved. Maple is a trademark of Water
 <_____> Waterloo Maple Inc.
      |      Type ? for help.
> p := randpoly(x);
           5           4           3           2
p := -56 - 7 x  + 22 x  - 55 x  - 94 x  + 87 x

> codegen[cost](p);

           5 additions + 15 multiplications

> quit;
bytes used=562828, alloc=458668, time=0.03
```

Complexity and  
Cost

Measuring Complexity:  
big-oh

complexity classes

Cost of Algorithms

Tree Traversal Programs  
counting flops:  
floating-point operations

Maple's cost  
function

Summary +  
Assignments

# Summary + Assignments

See chapter 12 in *The Art & Craft of Computing*.

Assignments:

1. Examine the space complexity to sort  $n$  numbers.  
Expresses the memory use as a function of  $n$ .
2. If a (double) float occupies 8 bytes, how much space is needed to sort one million numbers? Find out how much internal memory your computer has. What is the largest list you could sort?
3. Modify the class `flopfloats.py` so that multiplications and divisions are counted separately from the additions and subtractions.
4. Run `floppy.py` for degrees  $d$  ranging from 2 to 20 and record the flops.
5. Look at the code for `floppy.py` and find a formula for its cost in function of  $d$ .

## Complexity and Cost

Measuring Complexity:  
big-oh  
complexity classes

## Cost of Algorithms

timing Python programs  
counting flops:  
floating-point operations

## Maple's cost function

## Summary + Assignments

# Problems to be Collected

MCS 260 L-29

2 November 2007

## Complexity and Cost

Measuring Complexity:  
big-oh  
complexity classes

## Cost of Algorithms

timing Python programs  
counting flops:  
floating-point operations

## Maple's cost function

## Summary + Assignments

**Homework collected on Monday 5 November:**  
exercise 2 of Lecture 26; exercises 1 (or 2), 3 and 4 of Lecture 27; and exercise 2 of Lecture 28.