This is a course about **algorithmics**, the **mathematical study of algorithms**.

How and why should we study algorithms?

- **Real-World Code?**: Use science and engineering on actual problems. *Important, but not our topic here.*
- **Abstract Problems and Solutions?**: Formulate common problems mathematically, design algorithms for them, prove theorems about the resources needed by each algorithm. *The basic activity of this course.*
- **Computational Paradigms?**: Identify design principles common to problems in many domains. *The organizational structure of this course.*

What should this course give you to use as a working computer scientist?

- Specific solutions to specific problems, with specific costs
- A toolkit of appproaches to new problems
- A deeper understanding of what can be done, and what can be done quickly.

Two Slides From CMPSCI 601

Our real-world target is *digital computation*. Our first abstraction is say that we have an *input*, a collection of bits, and want an *output*, often a single bit. The key questions are then:

- How is the input organized?
- What computational operations are allowed?
- Do we have internal memory, and how much?

An answer to these questions gives us a formal model of computation.

Some Formal Models of Computation:

- **Boolean:** Input bits are undifferentiated, we can use boolean operations (AND, OR, NOT) and store the results. We also express properties of the input using propositional logic.
- **Formal Language Theory:** The input bits are arranged in a *string* of letters. We work with one letter at a time. Defining the internal memory gives us models such as the *finite-state machine*, *pushdown automaton*, or *Turing machine*.
- **First-Order Logic:** The input bits form a *structure* made up of *relations*. We express properties of the input using first-order logic (e.g., quantifiers $\forall$ and $\exists$).
- **Recursive Function Theory:** Input bits are formed into non-negative integers, on which we define functions starting from arithmetic operations.
- **Abstract RAM:** The input and internal memory are formed into words in registers, and operations mimic those of real-world sequential computers.

The Basic Question: How many steps might it take, in the **worst case**, to solve problem X for an input of size n? (Sometimes we will also think about the **average case**.)

This is the **time complexity function** for problem X, called $T_X(n)$ or $T(n)$ if X is clear from context.

How to measure the answer? Not just "with what units", but "with what mathematics"?

- We don't care exactly what a basic step consists of.
- We're interested mostly in the *growth* of $T(n)$ as n gets large. So we *ignore small values* of n and *ignore constant multiplicative factors*.

Thus we use *asymptotic notation* for functions.

Big-O and Related Notation:

Let f and g be functions from **N** (the non-negative integers) to **R**, the real numbers.

- $f = O(g)$ ("f is big-O of g") means

$$\exists c : \exists n_0 : \forall n \geq n_0 : f(n) < cg(n)$$

- $f = o(g)$ ("f is little-O of g") means

$$\forall c > 0 : \exists n_0 : \forall n \geq n_0 : f(n) < cg(n)$$

- $f = \Omega(g)$ ("f is big-Omega of g") means

$$\exists c : \exists n_0 : \forall n \geq n_0 : f(n) \geq cg(n)$$

- $f = \omega(g)$ ("f is little-Omega of g") means

$$\forall c > 0 : \exists n_0 : \forall n \geq n_0 : f(n) > c(f(n)$$

- $f = \Theta(g)$ ("f is Theta of g") means

$$f = O(g) \wedge f = \Omega(g)$$

Basic Facts: Θ is an equivalence relation, arithmetic works on classes with additional rules such as $(f = O(g)) \rightarrow (f + g = O(g))$. etc.

Common Equivalence Classes: $\Theta(1)$, $\Theta(\log n)$, $\Theta(n)$, $\Theta(n \log n)$, $\Theta(n^2)$, $\Theta(n^3)$, $\Theta(2^n)$, $\Theta(n!)$, etc.

A problem X is in the class **P** if $\exists k : T_X(n) = O(n^k)$.

Roughly speaking, the problems in **P** are those whose running times scale reasonably with n. But there are example of problems in **P** where:

- The best known exponent is large ($O(n^{12})$ for prime-testing)
- An exponent has been proved to exist but is unknown, or the best known one is inconceivably large
- There is a known algorithm with $O(n^2)$ time, but the best known constant in the big-O is inconceivably large

But within the realm of problems we look at, identifying **P** with "feasible" will make sense. And if a problem is not in **P**, it does *not* scale well though we may have a solution for useful small values of n.

Texts:

[A:] Micah Adler, Lecture notes for this course, available at Collective Copies, Amherst.

[CLRS:] Cormen, Leiserson, Rivest, and Stein, *Introduction to Algorithms*, McGraw-Hill (suggested only)

[KT]: Kleinberg and Tardos, *Algorithm Design*, Addison-Wesley (suggested only)

Prerequisites: Mathematical maturity: reason abstractly, understand and write proofs, use big-O notation. In UMass terms CMPSCI 250 absolutely needed; CMPSCI 311 strongly suggested, CMPSCI 401 helpful. Considerable use of basic probability, some calculus.

Graded Work:

- Five or six problem sets (30% of grade)
- Midterm (30% of grade), Wed 26 October, 6:30-9:00 p.m., Herter 231
- Final (40% of grade), during exam period

Cooperation: Students should talk to each other and help each other; but **write up solutions on your own, in your own words.** Sharing or copying a solution could result in a grade of F for the course, even on the first offense. If a significant part of one of your solutions is due to someone else, or something you've read then **you must acknowledge your source!** When the grader then looks at the source, it should be clear from your writeup that you've understood anything you've taken from it. A good heuristic is not to have the source in front of you when you write up.

The syllabus is now up on the course web site:

- `http://www.cs.umass.edu/~barring/cs611`

Lectures will follow the Adler notes fairly closely, and my slides will be available on the web site.

High-Level Outline of the Course:

- Preliminaries, Divide and Conquer (3 lectures)
- Greedy Algorithms and Matroids (4 lectures)
- Shortest Paths and Network Flow (4 lectures)
- Randomized Algorithms (4 lectures)
- NP-Completeness and Approximation Algorithms (7 lectures)
- Linear Programming (3 lectures)

Divide and Conquer Paradigm: *Split* problem into pieces, *solve* subproblems using recursive calls, *integrate* the solutions. Need a *base case* solution. Prove correctness by *mathematical induction*.

Sorting Problem: Input an array of n objects of a type that can be compared. Output the same objects in sorted order according to these comparisons.

Mergesort Algorithm: Split list arbitrarily into two equal-size sublists. Sort each list recursively. Merge the sorted sublists by comparing the first elements of each list and copying the smaller into the output list (deleting it from its old list). Base case of sorting a one-element list is a no-op.

How long does this take?

We form a *recurrence relation* that gives an inductive definition of the function $T(n)$. **Assume that n is a power of two** (see HW#1).

- If $n \leq 1$, $T(n) = O(1)$
- Otherwise, for even n, $T(n) = 2T(n/2) + T_M(n)$

We first need to analyze $T_M(n)$, the time to *merge* the two sorted lists. It is pretty easy to see that this is $\Theta(n)$, because we do $\Theta(1)$ steps to deal with each of the n items. So our second clause becomes:

$T(n) = 2T(n/2) + \Theta(n)$.

We will now look at a general method to solve recurrences of this type.

The Mergesort analysis has given us the recurrence:

$T(n) = 2T(n/2) + \Theta(n)$,

with base case $T(n) = \Theta(1)$ for $n \leq 1$.

Whenever we use divide-and-conquer, splitting the input into a pieces each of size n/b, merging them in $\Theta(n^\alpha)$ time, and solving the base case in constant time, we get a recurrence of this form:

$T(n) = aT(n/b) + \Theta(n^\alpha)$, base case $T(n) = \Theta(1)$ for $n \leq c$

The *Master Theorem* gives us a general solution for $T(n)$, in big-O terms:

- Let $\beta = \log_b(a)$
- If $\alpha > \beta$, then $T(n) = \Theta(n^\alpha)$
- If $\alpha = \beta$, then $T(n) = \Theta(n^\alpha \log n)$
- If $\alpha < \beta$, then $T(n) = \Theta(n^\beta)$

Proof of the Master Theorem:

Use the rule to expand $T(n)$ repeatedly:

$$\begin{aligned} T(n) &= aT(n/b) + \Theta(n^\alpha) \\ T(n) &= a^2T(n/b^2) + a\Theta((n/b)^\alpha) + \Theta(n^\alpha) \\ T(n) &= a^3T(n/b^3) + a^2\Theta((n/b^2)^\alpha) + a\Theta((n/b)^\alpha + \Theta(n^\alpha) \end{aligned}$$

When does this stop? When the argument of T, which is n/b^i, falls into the base case. For simplicity, let's assume that n is a power of b, so that eventually $n/b^i = 1$ and thus $T(n/b^i) = \Theta(1)$. This gives us:

$$T(n) = a^i\Theta(1) + a^{i-1}\Theta(b^\alpha) + \ldots + a\Theta((n/b)^\alpha) + \Theta(n^\alpha)$$

$$T(n) = a^i\Theta(1) + a^{i-1}\Theta(b^\alpha) + \ldots + a\Theta((n/b)^\alpha) + \Theta(n^\alpha)$$

We may factor out the Θ's (why?), and then note that to get each term from the following one, we multiply by a and divide by b^α. So defining r to be a/b^α, we get:

$$T(n) = \Theta(n^\alpha(r^i + r^{i-1} + \ldots + r + 1))$$

(Because $n^\alpha r^i = n^\alpha(a^\alpha/b^{i\alpha})$ and $b^i = n$.)

By the rule for arithmetic progressions (as long as $r \neq 1$), this is:

$$T(n) = \Theta(n^\alpha)(\tfrac{1-r^{i+1}}{1-r})$$

We need to analyze this sum based on the relative size of α and β.

$T(n) = \Theta(n^\alpha(r^i + r^{i-1} + \ldots + r + 1))$

$T(n) = \Theta(n^\alpha)(\frac{1-r^{i+1}}{1-r})$

Case I: If $\alpha > \beta$, then $r = a/b^\alpha < a/b^\beta = a/b^{\log_b a} = a/a = 1$. Then the geometric sum is $\Theta(1)$ (some constant) and $T(n) = \Theta(n^\alpha)$.

Case II: If $\alpha = \beta$, then $r = a/b^\alpha = a/b^\beta = a/a = 1$. The terms in the geometric sum are all the same (each equal to 1) so we just have to count them. There are $i + 1$ of them, and $i = \log_b(n) = \Theta(\log n)$ since b is a constant. So $T(n) = \Theta(n^\alpha \log n)$.

Case III: If $\alpha < \beta$, then $r > 1$ and $\frac{1-r^{i+1}}{1-r} = \Theta(r^i)$. Since $r = a/b^\alpha$, $r^i = a^i/b^{i\alpha} = a^i/n^\alpha$. So $n^\alpha r^i = a^i = (b^{\log_b a})^i = b^{\beta i} = (b^i)^\beta = n^\beta$.

We have proved the Master Theorem. In the case of Mergesort, $a = b = 2$ because we solve two subproblems of half the size. We have $\alpha = 1$ because the time to merge is linear. Since $\beta = \log_b a = \log_2 2 = 1$, $\alpha = \beta$ and we are in Case II. So the running time of Mergesort is $\Theta(n^\alpha \log n) = \Theta(n \log n)$.