Computational Optimization
ISE 407

Lecture 6

Dr. Ted Ralphs

Reading for this Lecture

- Aho, Hopcroft, and Ullman, Chapter 1
- Miller and Boxer, Chapters 1 and 5
- "Engineering Cache Oblivious Sorting Algorithms", K. Vinther.

Computational Complexity

- What is computational complexity?
  - With respect to a *mathematical problem*, complexity theory provides a rigorous framework for assessing difficulty (absolute and relative).
  - With respect to different *algorithms* for the same problem, complexity theory provides a framework for assessing and comparing efficiency.
- To rigorously define the meaning of *efficient*, we need a basic set assumptions called the *model of computation*.

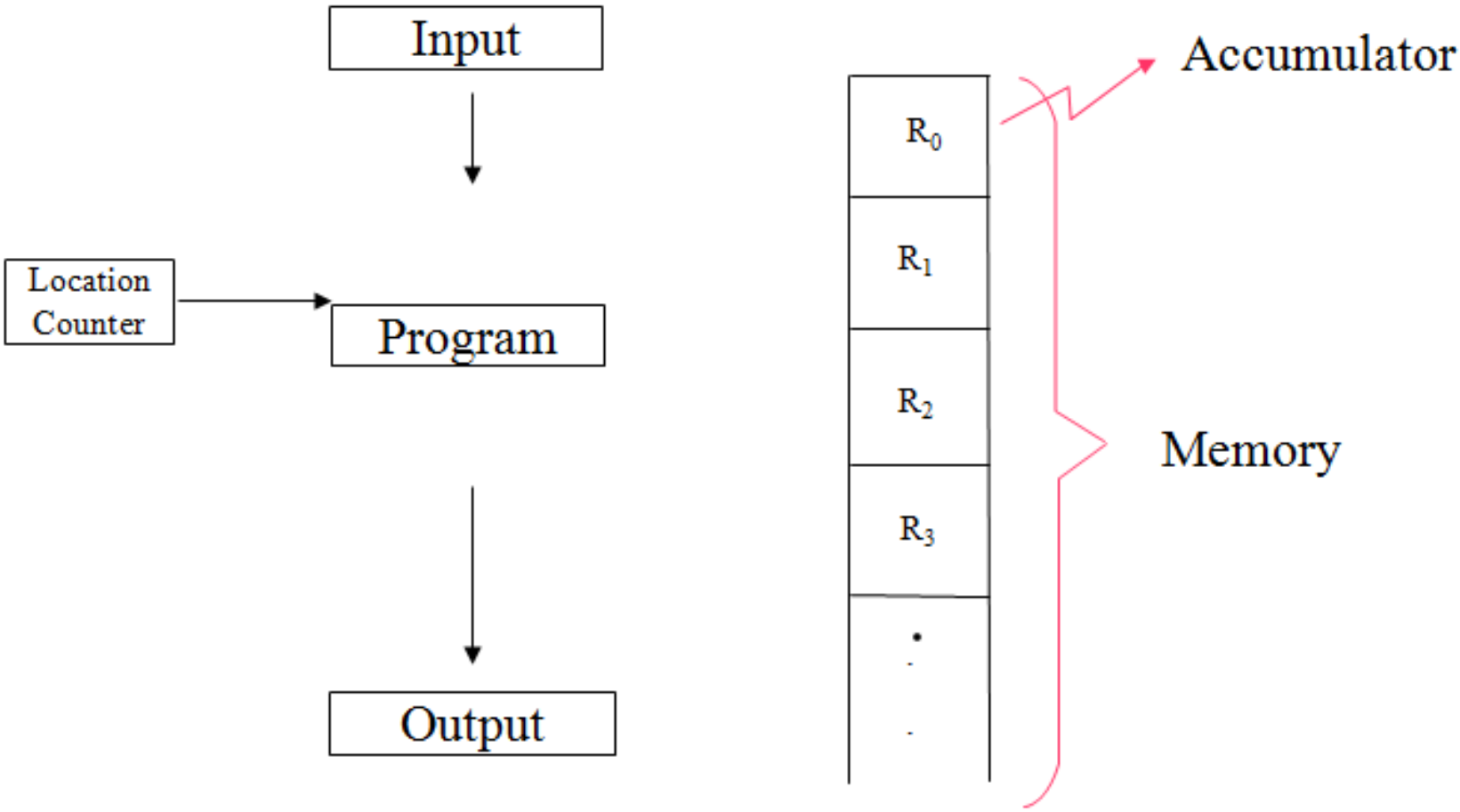
Models of Computation

- The model of computation is a conceptual model of how a computer works on which we can base a formal analysis of algorithms.
- We would like our model to come as close as possible to representing the realities discussed in the first four lectures.
- Unfortunately, it is not possible for a useful conceptual model to take into account all practical details.
- The model must be simple enough to allow proof of certain mathematical results based on assumptions of the model.
- Well-known models do not take into account details of the hardware like the memory hierarchy.

Turing Machines

- The complexity framework we use today is based on the concept of a *Turing machine* proposed by A.M. Turing.
- It was originally intended to allow rigorous specification of the steps of an algorithm in terms of simple, atomic operations.
- A Turing machine essentially specifies an *algorithm* or, more specifically, a *program*, but the concept was invented before computers existed.
- Corresponding to any given algorithm, there is a Turing machine that takes an input and produces an output through a sequence of steps.
- The specific sequence might be different for different inputs, i.e., we have a notion of *conditional branching*.
- Turing later conceived of something like what we think of as a *computer*, which was able to load a "program" into its memory.
- This became known as a *universal Turing machine*.
- These concepts heavily influenced the inventions that lead to modern computer architectures.

The RAM Model of a Computer



Execution in the RAM Model

- The RAM model of a computer is mathematically similar to a Turing machine, but described in terms more similar to a modern computer.
- In each time step, one instruction is executed.
- The instructions are similar to those in a machine language.
- Each instruction consists of an *operation* and an *address* (more like an argument).
- The input to the operation can either be the location of the next instruction, a memory location, or a constant operand (integer).
- The accumulator is where the result of any operation is placed.
- The input to the RAM program is read sequentially and the output is written sequentially.

Assumptions of the RAM model

- The program is a sequence of (optionally) labeled instructions.
- The program is not stored in memory and hence cannot modify itself.
- The instance is small enough to fit in the memory.
- All numbers fit into one computer "word" ($\Leftarrow$ Important).
  - This means that we don't model the details of floating point operations.
  - Essentially, all operations are performed in full rational arithmetic.
- Any memory location can be accessed in one unit of time (there no concept of registers or cache).
- In the simplest variant, all fundamental operations can be performed in one unit of time.
- This is what is known as a "unit cost model."

RAM Instructions

- The exact set of instructions can vary, but would be something like the following.

Operation Code	Address	Operation Code	Address
LOAD	operand	DIV	operand
STORE	operand	READ	operand
ADD	operand	WRITE	operand
SUB	operand	JUMP	label
MULT	operand	JGTZ	label
HALT		JZERO	label

- The operand can be one of the following
  - $= i$, indicating the integer itself.
  - A non-negative integer i, indicating the contents of register i
  - $*i$, indicating the contents of register j, where j is the contents of register i (indirect addressing)

Interpretation

- Let c be the memory map, so that $c(i)$ is the contents of register i.
- then the *value* of the operand is
  - $v(= i) = i$
  - $v(i) = c(i)$
  - $v(*i) = c(c(i))$
- Instructions can be interpreted as follows.
  - `LOAD a` $\Rightarrow c(0) \leftarrow v(a)$
  - `STORE i` $\Rightarrow c(i) \leftarrow c(0)$
  - `ADD a` $\Rightarrow c(0) \leftarrow c(0) + v(a)$
  - `READ i` $\Rightarrow c(i) \leftarrow$ current input symbol
  - `JGTZ b` $\Rightarrow$ Go to instruction b if $c(0) > 0$.
  - ...

Alternative Models of Computation

- The models of computation we've discussed here are overly simplistic and do not take into account such details as the cache.
- There are more sophisticated models, but these are difficult to work with and the details matter more.
- Time will tell whether more sophisticated theoretical analysis proves fruitful.
- There are also related frameworks that consider *computability*, such as the λ-calculus and recursion theory.
- The original Church-Turing thesis addressed the equivalence of Turing-computability and the notions yielded by these other frameworks.
- They were eventually shown to be formally equivalent.

Using the Model

- To assess an algorithm using the RAM model, we analyze an implementation of the algorithm specified in the RAM model.
- The goal is typically to count the number of basic steps needed to execute the algorithm over a set of instances.
- In some cases, we also assess the memory usage.
- The number of basic steps required to solve a given instance of a problem is called the *running time* for that instance.
- The running time can be different for different models of computation.
- For the types of analysis we'll do, the details will not usually matter, at least for sequential algorithms.
- In most cases, we don't actually need to write a detailed description of the algorithm in the RAM model.

Chruch-Turing Thesis

Proposition 1. *(Complexity-theoretic) Church-Turing Thesis: All "realistic" models of computation are polynomially equivalent.*

- The above result is also known as the *feasibility thesis* and can be more formally stated as

  "A probabilistic Turing machine can efficiently simulate any realistic model of computation."

- Accordingly, the unit cost model would not be considered "realistic."
- For example, computing the determinant of an $n \times n$ matrix takes $\approx n^3$ steps in the unit cost model.
- It has exponential running time in the log cost model because the sizes of the numbers arising in the calculation can gow exponentially.
- In practice, we thus use the log cost model in most cases.

Complexity of an Algorithm

- To now formally define the complexity of an algorithm, we need a summary measure of the number of time steps required to execute it.
- The summarization is done over a (usually infinite) set of instances.
- We have previously discussed several possible summary measures.
  - Worst case
  - Average case
  - Best case
- For practical reasons, worst-case is the typical choice.
- When the set of instances over which summarization is occurring is large, it is not useful to summarize over all instances.
- Instead we derive a function of some property that serves to divide the instances into subsets.
- What should the property be?

The Size of an Instance

- The "size" of the input is a natural universal property that clearly affects efficiency, since we mst at least read in the data.
- The *size* of an instance is defined formally to be the amount of memory required to store a description of the instance.
- The size may be different, depending on assumptions of our model of computation.
  - In the basic unit cost model, the size of the instance is the number of distinct input parameters.
  - The log cost model is more realistic and assumes $\log n$ bits are required to store an integer n.
- In optimization, we often need to explicitly account for the *magnitude* of the input data and.

Encoding Length

- In the log cost model, the number of bits required to store a number n is called its *encoding length*, denoted $\langle n \rangle$.
- Although it is tempting to think that the unit cost model suffices for most practical cases, this is less true than one might think.
- Even if the numbers in the input have small encoding length, numbers that arise in intermediate computations must also be accounted for.
- It's also important to realize that bounded encoding length limits not only how big a number can be, but also how *small*.
- Thus, limiting encoding length limits *accuracy*.
- This aligns with the practical reality that floating point numbers do have a limited encoding length.
- To break free of this restriction, one must implement non-primitive numerical types, such as the `Rational` type in Julia.

Recall: Equivalence of Optimization and Separation

- Recall this result from IE 418. Separation Problem: Given a polyhedron $\mathcal{Q} \subseteq \mathbb{R}^n$ and $x^* \in \mathbb{R}^n$, determine whether $x^* \in \mathcal{P}$ and if not, determine (π, π_0), a valid inequality for $\mathcal{Q}$ such that $\pi x^* > \pi_0$.

  Optimization Problem: Given a polyhedron $\mathcal{Q}$, and a cost vector $c \in \mathbb{R}^n$, determine x^* such that $cx^* = \max\{cx : x \in \mathcal{Q}\}$.

  Theorem 1. *For a family of rational polyhedra $\mathcal{Q}(n, T)$ whose input length is polynomial in n and $\log T$, there is a polynomial-time reduction of the linear programming problem over the family to the separation problem over the family. Conversely, there is a polynomial-time reduction of the separation problem to the linear programming problem.*

- The parameter n represents the dimension of the space.
- The parameter T represents the largest numerator or denominator of any coordinate of an extreme point of $\mathcal{Q}$ (the *vertex complexity*).

Connection to Encoding Length

- Note that the definition of *vertex complexity* was not given in terms of the encoding length of a full description of $\mathcal{P}$.
- This is because $\mathcal{P}$ may be an implicitly defined polyhedron whose description is never fully constructed.
- What *is* explicitly constructed are the components of the description (extreme points and facet-defining inqeualities).
- The importance of the vertex (and facet) complexity in the analysis is primarily that they provide bounds on the norms of *these* vectors.
- The ability to derive such bounds is a crucial element in the overall framework they present.

Results on Encoding Length

- The following are relevant results from Grötschel, Lovász and Schrijver.

Proposition 2. *(Grötschel, Lovász and Schrijver 1993)*

1. *(1.3.3) For any* $r \in \mathbb{Q}$, $2^{-\langle r \rangle + 2} \leq |r| \leq 2^{\langle r \rangle - 1} - 1$.
2. *(1.3.3) For any* $x \in \mathbb{Q}^n$, $\|x\|_2 \leq \|x\|_1 < 2^{\langle x \rangle - n}$.
3. *(6.2.8) If* $\mathcal{P}$ *is a polyhedron with vertex complexity at most* ν *and* $(a, b) \in \mathbb{Z}^{n+1}$ *is such that*

$$a^\top x \leq b + 2^{-\nu - 1}$$

for all $x \in \mathcal{P}$*, then* (a, b) *is also valid for* $\mathcal{P}$.

- In other words, the facet complexity and the encoding lengths of the vectors involved specify a "granularity".

Running Time Function of an Algorithm

- We now have all the pieces to formally define a notion of complexity.
- The formal *(time) complexity* is defined to be the worst-case running time expressed as a *function* of the *size* of the instance.
- This function is called the *running time function* of the algorithm.
- The *space complexity*, on the other hand, is the number of registers required to execute the algorithm.

Asymptotic Analysis

- We now develop a formal framework for comparing algorithms based on their running time functions.
- Unless otherwise specified, we will assume all functions map $\mathbb{N}_+$ to $\mathbb{R}_+$.
- Our usual function names will be f, g, and T.
- We will also assume that n is a variable denoting the input size that takes on values in $\mathbb{N}_+$.
- We will also use m as a variable taking on values in $\mathbb{N}_+$.
- We will use a, b, and c to denote constants.
- Generally, all variables and constants will take on values in $\mathbb{N}_+$.
- Although it is common practice, I will try not to refer to a function by the notation "$f(n)$" because $f(n)$ is a value, not a function.
  - Correct: "f is a polynomial function."
  - Incorrect: "$f(n)$ is a polynomial function."

Asymptotic Analysis

- Question: Why are we *really* interested in the theoretical running times of algorithms?
- Answer: To compare different algorithm for solving the same problem.
- We are interested in performance for large input sizes.
- For this purpose, we need only compare the *asymptotic growth rates* of the running times.
  - Consider algorithm A with running time given by f and algorithm B with running time given by g.
  - We are interested in knowing

$$L = \lim_{n \to \infty} \frac{f(n)}{g(n)}$$

  - What are the four possibilities?

Θ Notation

- We now define the set

$$\Theta(g) = \{f : \exists c_1, c_2, n_0 > 0 \text{ such that } c_1 g(n) \leq f(n) \leq c_2 g(n)\ \forall n \geq n_0\}$$

- If $f \in \Theta(g)$, then we say that f and g *grow at the same rate* or that they are *of the same order*.
- Note that

$$f \in \Theta(g) \Leftrightarrow g \in \Theta(f)$$

- We also know that if $\lim_{n\to\infty} \frac{f(n)}{g(n)} = c$ for some constant c, then $f \in \Theta(g)$.
- If the limit doesn't exist, we don't know.

Big-O Notation

- We now define the set

$$O(g) = \{f : \exists c, n_0 > 0 \text{ such that } 0 \leq f(n) \leq cg(n) \forall n \geq n_0\}$$

- If $f \in O(g)$, then we say that "f is big-O of g" or that g *grows at least as fast as* f.
- Note that if $f \in O(g)$, then either $f \in \Theta(g)$ or $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$
- Some other notation
  - $f \in \Omega(g) \Leftrightarrow g \in O(f)$.
  - $f \in o(g) \Leftrightarrow f \in O(g) \setminus \Theta(g) \Leftrightarrow \lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$.
  - $f \in \omega(g) \Leftrightarrow g \in o(f) \Leftrightarrow \lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty$.

Example

- Recall the polynomial evaluation example from last class.
- Let's show that if $f(n) = \frac{1}{2}(n^2 + 3n)$, then $f \in \Theta(n^2)$.

Comparing Functions

- The notation we have just defines gives us a way of ordering functions.
- We can can interpret
  - $f \in O(g)$ as "$f \leq g$,"
  - $f \in \Omega(g)$ as "$f \geq g$,"
  - $f \in o(g)$ as "$f < g$,"
  - $f \in \omega(g)$ as "$f > g$," and
  - $f \in \Theta(g)$ as "$f = g$."
- This gives us a method for comparing algorithms based on their running times.
- Note that most of the relational properties of real numbers (transitivity, reflexivity, symmetry) work here also.
- However, not every pair of functions is comparable.

Commonly Occurring Functions

- Polynomials: All polynomials f of degree k are in $\Theta(n^k)$.
- Exponentials
  - A function in which n appears as an exponent on a constant is an *exponential function*, i.e., 2^n.
  - For all positive constants a and b, $\lim_{n\to\infty} \frac{n^a}{b^n} = 0$.
  - This means that exponential functions always grow faster than polynomials.
- Logarithms
  - Logarithms of different bases differ only by a constant multiple, so they all grow at the same rate.
  - A *polylogarithmic* function is a function in $\Theta(lg^k)$.
  - Polylogarithmic functions always grow more slowly than polynomials.
- Factorials: Factorial functions grow more quickly than exponentials, but are in $o(n^n)$.

Simple Example

- Consider the following simple problem:

  Input: $x \in \mathbb{R}$ and $n \in \mathbb{Z}$.

  Output: x^n.

- What is the most straightforward algorithm for solving this problem?
- What is its running time function?
- Is there a *better* algorithm?

A More Efficient Algorithm

Let's assume that $n = 2^m$ for $m \in \mathbb{Z}$. We can use repeated squaring:

```
def pow(x, n):
    for i in range(log(n, 2)):
        x *= x
```

Questions:

- Is this algorithm correct?
- What is its running time function?
- What do we mean by *efficiency*?
- Why don't we call it *speed*?
- How do we modify it for the general case?

Example: Calculating Fibonacci Numbers

Consider the following function for calculating the n^{th} Fibonacci number derived from its definition.

```
def fibonacci1(n):
    if n == 1 or n == 2:
        return 1
    else:
        return fibonacci1(n-1) + fibonacci1(n-2)
```

- The correctness of this function does not really need to be proven formally, but to illustrate, we could prove it using induction.
- A formal inductive proof requires
  - a base case; and
  - an inductive hypothesis
- What are they in this case?
- How do we know the algorithm terminates?

Example: Calculating Fibonacci Numbers

- Let's try to determine how long it will take to calculate the n^{th} Fibonacci number using the algorithm on the previous slide.
- Here is a small routine that returns the execution time of a function for calculating a fibonacci number.

```
def timing(f, n):
    print 'Calculating fibonacci number', n
    start = time()
    f(n)
    return (time()-start)

>>> print timing(fibonacci1, 10)
0.00299978256226
>>> print timing(fibonacci1, 30)
31.0210001469
```

- What happened with the second function call??
- Why is this function apparently so inefficient??
- What is its running time function?

Improving the Implementation

- Although recursion provides a natural way for mathematically defining the Fibonacci function, this does not lead to an efficient algorithm.
- How do we improve the function?
- What is the reason for the inefficiency?

Calculating Fibonacci Numbers: Second Implementation

- Second Try: Store and reuse intermediate results.

```
def fibonacci2(n):
    f = [0, 1, 1]
    for i in range(3, n+1):
        f.append(f[i-1] + f[i-2])
    return f[n]
```

- What is the running time function now?
- Are there any downsides of this implementation?

Calculating Fibonacci Numbers: Third Implementation

- Third Try: Only store the intermediate results that are needed.

```
def fibonacci3(n):
    a, b = 0, 1
    for i in range(n):
        a, b = b, a+b
    return a
```

Deriving Running Time Function Empirically

- Let's derive the running time functions of the two implementations empirically in this case.
- Note that this is an unusual case where the input is just a single integer, so the worst case and average case are essentially the same.
- Hence, the theoretical and empirical should match.

```
algos = [fibonacci1, fibonacci3]
symbols = ['bs', 'rs']
symboldict = dict(zip(algos, symbols))
actual = {}
for a in algos:
    actual[a] = []
    for i in range(1, 30):
        actual[a].append(timing(a, i))
# create plots
for a in algos:
    plt.plot(range(1, 30), actual[a], symboldict[a])
plt.show()
```

Plotting the Data

- To plot the data, we use `matplotlib` .
- Plotting the results of the code on the previous slide, we get the following.

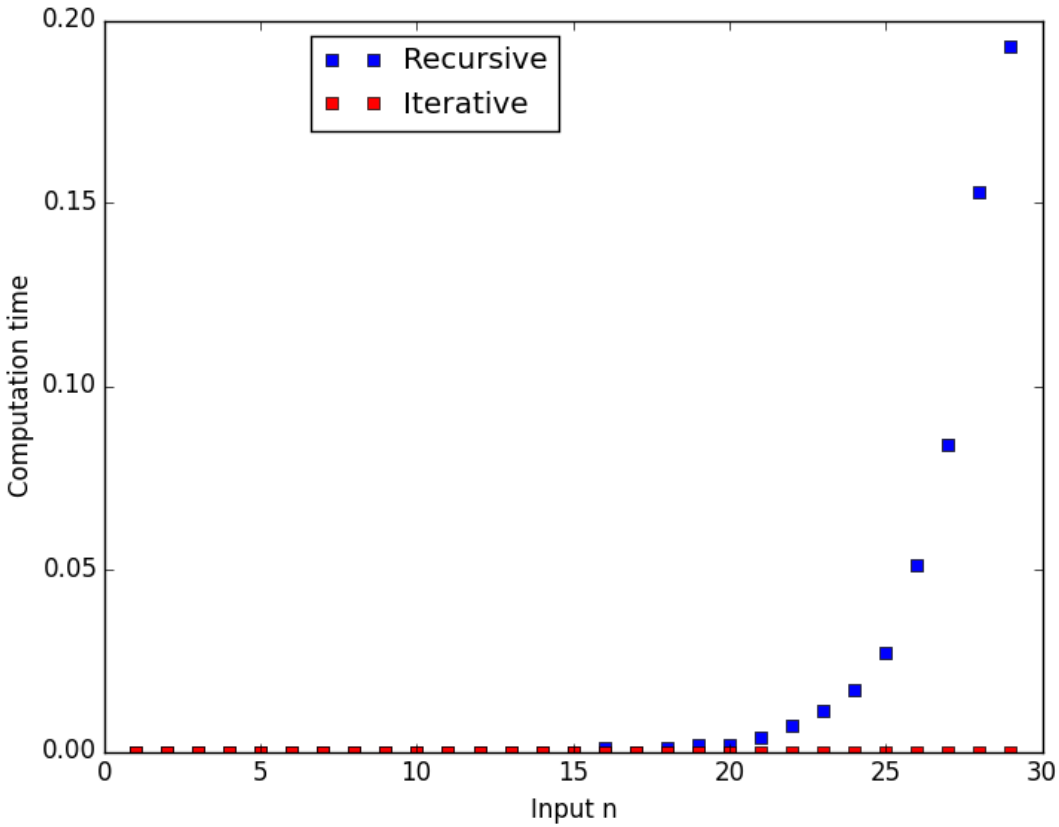


Figure 1: Running times of recursive versus iterative algorithms

Comparing Theortical and Empirical

- Now we compare our theoretical prediction to the empirical data from earlier.
- What are the "units" of measurement?
- To put the numbers on the same scale, we need to either determine the hardware constant or count the number of "representative operations".

```
theoretical = []
actual = []
n = range(1, 30)
for i in range(1, 30):
    actual.append(timing(a, i))
    theoretical.append(fibonacci(i))
# figure out the constant factor to put times on the same scale
scale = actual[algos[0]][-1]/theoretical[-1]
theoretical = [theoretical[i]*scale for i in range(len(n))]
plt.plot(n, actual, 'bs')
plt.plot(n, theoretical, 'ys')
plt.show()
```

Plotting the Data

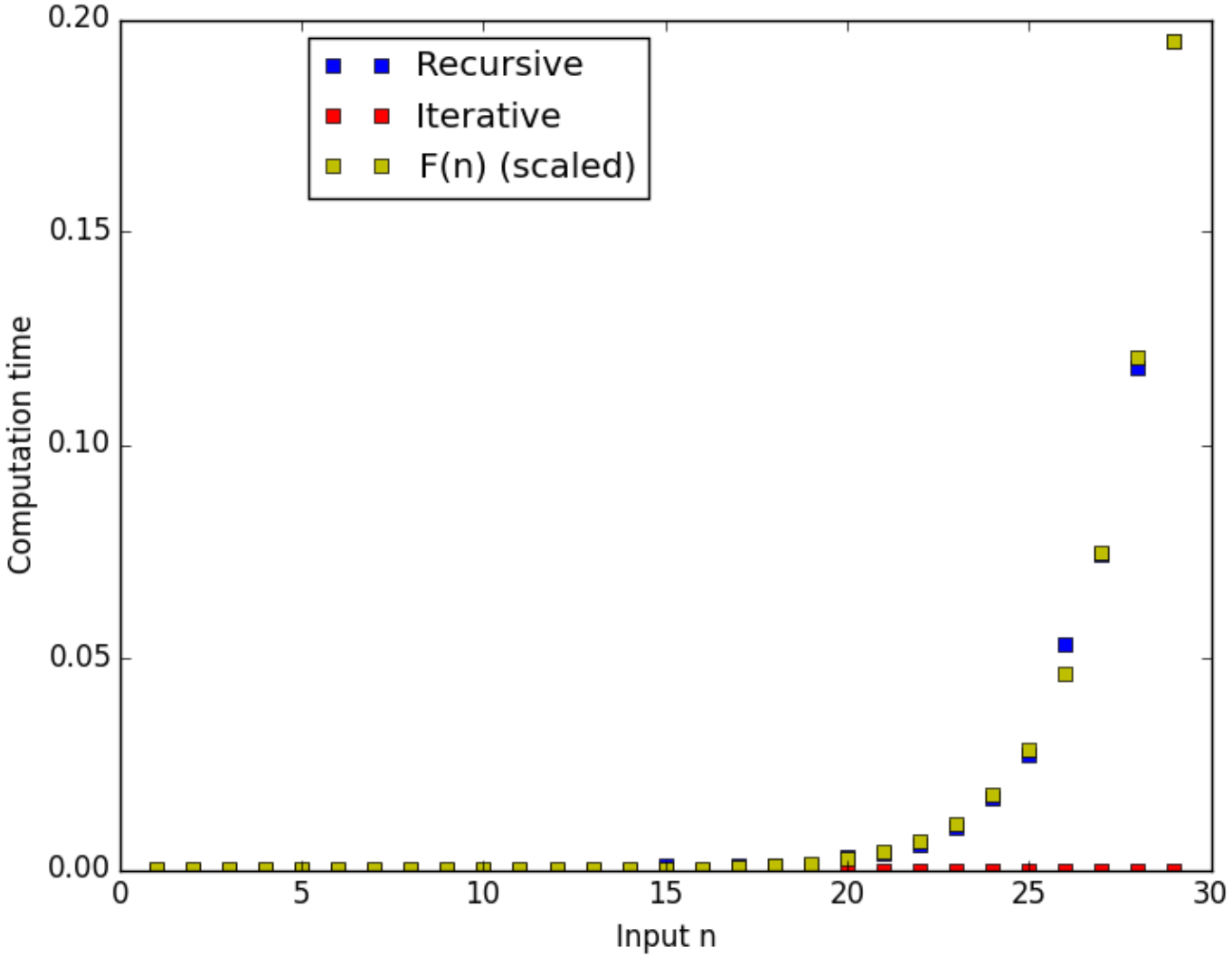


Figure 2: Running times of recursive versus iterative algorithms

Assessing

- As expected, there is a pretty good match between theoretical and empirical, but we shouldn't expect this in general.
- Empirical data usually represents some kind of "average" case, whereas our theoretical analysis will usually be worst case.
- Note also that the "input size" for the Fibonacci function is actual $\log n$, not n.
- It is important to observe that although our simple model of computation ignores constants and low-order terms, it is still useful.
- The types of optimizations discussed in the first block mainly affect constant terms in the empirical running time.
- Theoretical analysis is focused on improvements in high-order terms, which has a much higher impact.
- It is only after we make such "macro" improvements that it is appropriate to focus on more fine-grained optimizations.

Uses and Abuses of big-O Notation

- The uses of the notation for asymptotic can be confusing since they are used for different purposes and also sometimes misused.
- Meaning of $O(f)$
  - When applied to the running time an algorithm
    - Usually means that the true running time (which may not be known) is $O(f)$ (bounded above by f).
    - If the bound is tight, then technically, we should say the running time function is $\Theta(f)$.
    - However, in some cases, people mean that the running time for a single instance is $O(f)$, which doesn't tell you whether the bound is tight.
  - When applied to the complexity of a problem, it usually means that the running time function of the best-known algorithm is $O(f)$ (but there may be a better algorithm)

Uses and Abuses of Θ and Ω Notation

- Meaning of $\Theta(f)$
  - When applied to the running time of an algorithm
    - Usually means that f represents a worst-case bound on the true running time function that is tight.
    - Can mean that the running time is *always* $\Theta(f)$ for any input.
  - When applied to the complexity of a problem, usually means that the precise complexity is known.
- $\Omega(f)$ is typically used to indicate that there can't exist an algorithm with running better than f.
- The meaning is usually clear from context.

Limitations of Asymptotic Analysis

- Ignores constant factors
  - These are nearly impossible to model
  - Example:

```
for (i = 0; i < 10; i++)
   write i;

for (i = 9; i >= 0; i--)
   write i;
```

- Generally speaking, these models do not take the minor details of implementation into account.
- Small problem sizes
- Worst case vs. average case