References for Today’s Lecture

• Required reading
  – Sections 17.2-17.5

• References
  – AMO Sections 2.3
  – CLRS Section 22.1
Designing Algorithms

• In this class, we will discuss the development of algorithms that are both correct and efficient.

• How do we know if an algorithm is correct and what do we mean by efficient?
Proving Correctness

• **Correctness** of an algorithm must be proven mathematically.

• For the algorithms we’ll study, this will not be easy in some cases.

• Many algorithms simply require that the output satisfy some easily verifiable criterion and follow one of the following paradigms.
  
  – **Iterative**: The algorithm executes a loop until a termination condition is satisfied.
  
  – **Recursive**: Divide the problem into one or smaller instances of the same problem.

• In both cases, we must prove both that the algorithm terminates and that the result is correct by ensuring the criterion is satisfied.

  – Correctness of iterative algorithms is typically proven by showing that there is an *invariant* that holds true after each iteration.
  
  – Recursive algorithms are almost always proven by an induction argument.
Proving Correctness for Optimization Problems

- Correctness of an optimization algorithm requires more work.
- Typically, we will need to ensure that some particular optimality criteria are achieved.
- Proving that the algorithm terminates will involve showing that a certain progress towards achieving optimality is made on each step.
- These proofs are often not very obvious.
Example: Insertion Sort

A simple algorithm for sorting a list of numbers is insertion sort

def insertion_sort(l):
    for i in range(1, len(l)):
        save = l[i]
        j = i
        while j > 0 and l[j - 1] > save:
            l[j] = l[j - 1]
            j -= 1
        l[j] = save

Why is this algorithm correct? What is the invariant?
Example: Calculating Fibonacci Numbers

As an example of a recursive algorithm, consider the following function for calculating the $n^{th}$ Fibonacci number.

```python
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?
Algorithm Analysis

• We will be analyzing algorithms from both an empirical and a theoretical point of view.

• Theoretical analysis involves determining analytically the worst case number of basic operations required to execute an algorithms on a given class of graphs.

• Empirical analysis involves real-world performance of a given implementation over a given test set.

• This introduces many (irrelevant) factors that need to be controlled for in some way.
  – Test platform (hardware, language, compiler)
  – Measures of performance (what to compare)
  – Benchmark test set (what instances to test on)
  – Algorithmic parameters
  – Implementational details

• It is much less obvious how to perform a rigorous analysis in the presence of so many factors.

• Practical considerations prevent complete testing.
Measures of Performance

- For the time being, we focus on sequential algorithms.
- What is an appropriate measure of performance?
- What is the goal?
  - Compare two algorithms.
  - Improve the implementation of a single algorithm.
- Possible measures
  - Empirical running time (CPU time, wallclock)
  - Representative operation counts
Measuring Time

• There are three relevant measures of time taken by a process.
  – **User time** measures the amount of time (number of cycles taken by a process in “user mode.”
  – **System time** the time taken by the kernel executing on behalf of the process.
  – **Wallclock time** is the total “real” time taken to execute the process.

• Generally speaking, user time is the most relevant, though it ignores some important operations (I/O, etc.).

• Wallclock time should be used cautiously/sparingly, but may be necessary for assessment of parallel codes,
Representative Operation Counts

- In some cases, we may want to count operations, rather than time
  - Identify bottlenecks
  - Counterpart to theoretical analysis

- What operations should we count?
  - Profilers can count function calls and executions of individual lines of code to identify bottlenecks.
  - We may know a priori what operations we want to measure (example: API calls in a graph class).
Test Sets

• It is crucial to choose your test set well.
• The instances must be chosen carefully in order to allow proper conclusions to be drawn.
• We must pay close attention to their size, inherent difficulty, and other important structural properties.
• This is especially important if we are trying to distinguish among multiple algorithms.
Graphs for Testing

• Graph algorithms can perform much differently on graphs with different properties.

• Graphs that arise in applications are not generally “random.”

• The most important property is usually density, which is the ratio of the number of edges to the number of nodes.

• Most graphs that arise in practice are sparse.

• In performing various tests, it will be important to be able to generate random graph that perform like graphs that might arise in practice.

• How should we do this?
Generating Random Graphs: Unweighted

- Random edges
- Fixed degree
- Random degree
- k-neighbors
- Euclidean
Generating Random Graphs: Weighted

- Euclidean
- Random
Other Sources of Graphs

- Transaction graph
- Function call graph
- Social graph
- Interval graph
- Be Bruijn graph
Comparing Algorithms

- Given a performance measure and a test set, the question still arises how to decide which algorithm is “better.”

- We can do the comparison using some sort of summary statistic.
  - Arithmetic mean
  - Geometric mean
  - Variance

- These statistics hide information useful for comparison.
Accounting for Stochasticity

• In empirical analysis, we must take account of the fact that running times are inherently stochastic.

• If we are measuring wallclock time, this may vary substantially for seemingly identical executions.

• In the case of parallel processing, stochasticity may also arise due to asynchronism (order of operations).

• In such case, multiple identical runs may be used to estimate the affect of this randomness.

• If necessary, statistical analysis may be used to analyze the results, but this is beyond the scope of this course.
Performance Profiles

- Performance profiles allow comparison of algorithms across an entire test set without loss of information.
- They provide a visual summary of how algorithms compare on a performance measure across a test set.
Example: Calculating Fibonacci Numbers

• Let’s try to measure how long it takes to calculate the \(n^{th}\) Fibonacci number using the recursive algorithm.

• Here is a small routine that returns the execution time of a function for calculating a fibonacci number.

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

```bash
>>> print timing(fibonacci1, 10)
0.00299978256226
>>> print timing(fibonacci1, 30)
31.0210001469
```
Example: Calculating Fibonacci Numbers (cont’d)

- Notice that we are passing a function as an argument to another function.
- Since functions are just objects, we can put them on lists and pass them as argument, which is very useful.
- What happened with the second function call??
- Why is this function apparently so inefficient??
Calculating Fibonacci Numbers: Second Implementation

• **Second Try**: Store and reuse intermediate results.

```python
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]
```

• Are there any downsides of this implementation?
Calculating Fibonacci Numbers: Third Implementation

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

```python
def fibonacci3(n):
    a, b = 0, 1
    for i in range(n):
        a, b = b, a+b
    return a
```
Calculating Fibonacci Numbers: Comparing Recursive and Iterative

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

>>> print timing(fibonacci1, 10)
0.00299978256226
>>> print timing(fibonacci3, 10)
0.0

Why is the running time of fibonacci3 0?
Calculating Fibonacci Numbers (cont’d)

• The problem with the previous example was that the execution time of the function was so small, it could not be measured accurately.

• To overcome this problem, we can call the function repeatedly in a loop and then take an average.

```python
def timing(f, n, iterations = 1):
    print 'Calculating fibonacci number', n
    start = time()
    for i in range(iterations):
        f(n)
    return (time()-start)/iterations

>>> print timing(fibonacci1, 10, 1000)
0.00213199996948
>>> print timing(fibonacci3, 10, 1000)
3.61999988556e-05
```
Example: Calculating Fibonacci Numbers (cont’d)

• Note that the third argument to the function has a default value and is optional in calling the function.

• With this new function, we get a sensible measurement.

• Here, \texttt{fibonacci1()} is not obviously inefficient—we do not see this until we try larger numbers.
### Running Time as a Function of “Input Size”

- Typically, running times grow as the “amount” of data (number of inputs or magnitude of the inputs) grows.
- We are interested in knowing the general trend.
- Let’s do this in the fibonacci case.

```python
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, a full-featured package that provides graphing capabilities similar to Matlab.

• Plotting the results of the code on the previous slide, we get the following.

Figure 1: Running times of recursive versus iterative algorithms
Theoretical Analysis

• Can we derive the graph on the previous slide “theoretically”?

• In a basic theoretical analysis, we try to determine how many “steps” would be necessary to complete the algorithm.

• We assume that each “step” takes a constant amount of time, where the constant depends on the hardware.

• We might also be interested in other resources required for the algorithm, such as memory.

• What is the “theoretical” running time for each of the fibonacci algorithms?
  – Aside from the recursive calls, there are only roughly 2 “steps” in each function call.
  – The number of function calls is the $n^{th}$ Fibonacci number!
Theoretical Analysis

- Let's try to 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”

```python
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
More Complex Algorithms

• For most algorithms we will encounter, the analysis is not quite so straightforward.
  – We may not be able to derive the theoretical running time so easily.
  – The algorithm may behave very differently on different inputs.

• What do we want to know?
  – Best-case
  – Worst-case
  – Average-case

• It may depend on how much we know about the instances that will be encountered in practice or how risk-averse we are.
Example: Sorting

• Let's again consider the insertion sort algorithm.
  – How should we test it?
  – How about random instances?
  – Can we guess anything about the algorithm theoretically?
Insertion Sort: Simple Empirical Analysis

Generating random inputs of different sizes, we get the following empirical running time function.

Figure 3: Running time of insertion sort on randomly generated lists

What is your guess as to what function this is?
Insertion Sort: Theoretical Analysis

• What is the maximum number of steps the insertion sort algorithm can take?

• On what kinds of inputs is the worst-case behavior observed?

• What is the “best” case?

• On what kinds of inputs is this best case observed?

• Do you think that empirical analysis will tell us everything we need to know about this algorithm?
Operation Counts

• One way of avoiding the dependence on hardware is to count “representative operations”.

• What are the basic operations in a sorting algorithm?
  – Compare
  – Swap

• Most sorting algorithms consist of repetitions of these two basic operations.

• The number of these operations performed is a proxy for the empirical running time that is independent of hardware.
Figure 4: Operation counts for insertion sort on randomly generated lists
Obtaining Operation Counts

- One way to obtain operation counts is using a profiler.
- A profiler counts function calls and all reports the amount of time spent in each function in your program.

```python
>>> cProfile.run('insertion_sort_count(aList)', 'cprof.out')
>>> p = pstats.Stats('cprof.out')
>>> p.sort_stats('cumulative').print_stats(10)

<table>
<thead>
<tr>
<th>ncalls</th>
<th>tottime</th>
<th>percall</th>
<th>cumtime</th>
<th>percall</th>
<th>function</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1.011</td>
<td>1.011</td>
<td>3.815</td>
<td>3.815</td>
<td>insertion_sort</td>
</tr>
<tr>
<td>251040</td>
<td>0.507</td>
<td>0.000</td>
<td>0.507</td>
<td>0.000</td>
<td>shift_right</td>
</tr>
<tr>
<td>252027</td>
<td>0.393</td>
<td>0.000</td>
<td>0.393</td>
<td>0.000</td>
<td>compare</td>
</tr>
<tr>
<td>999</td>
<td>0.002</td>
<td>0.000</td>
<td>0.002</td>
<td>0.000</td>
<td>assign</td>
</tr>
</tbody>
</table>
```
Bottleneck Operations

- If an algorithm is not running as efficiently as we think it should, we may want to know where efforts to improve the algorithm would best be spent.

- Bottleneck analysis breaks an algorithm into parts (modules) and analyzes each part using the same analysis as we use for the whole.

- By determining the running times of individual modules, we can determine which part is the most crucial in improving the overall running time.

- To do this, we can make a graph showing the percentage of the running time taken by each module as a function of input size.

- This should make it obvious which module is the bottleneck.

- As we analyze more complex algorithms, we will do some of these kinds of analyses.
From Empirical to Theoretical

• Next, we will look at methods of doing theoretical analysis.

• These are much cleaner methods, in some sense, since many extraneous details of the test environment do not have to be considered.

• For sequential algorithms, theoretical analysis is often good enough for choosing between algorithms.

• It is less ideal with respect to tuning of implementational details.

• For parallel algorithms, theoretical analysis is far more problematic.

• The details not captured by the model of computation can matter much more.

• We will not consider parallel algorithms in this class.