Test 2 Statistics

- High score: 99
- Distribution of raw scores:
  - 90’s: 18
  - 80’s: 26
  - 70’s: 31
  - 60’s: 18
  - 50’s: 16
  - < 50: 25
- Curve: if $x$ is your raw score then
  $$y = \frac{x - 48}{90 - 48}(90 - 60) + 60$$
  is your scaled score. This is a simple linear scaling that is based on mapping 86 to 90 and 48 to 60.
- We will briefly discuss the solutions during lecture. The solutions are posted on the web.
- One of the questions will be repeated on the final!
Review from Thursday

Recursion examples

- Merge sort
- Word search

Today’s Class

- Why is erasing from a vector slower than erasing from a list?
- Analyzing algorithms by counting operations.
- The order notation.
- Different orders of magnitude.
- Best-case, average-case and worst-case analysis.
- Simple applications of the order notation, including loops and nested-loops.
- Analyzing the initial problem.
Erasing Vectors vs. Lists

- Suppose we have to build a vector or a list of numbers and then, later, to call a function to erase some of the values. For example, suppose we had to erase the odd numbers.

- Here’s the erase function for vectors:

```cpp
void erase_odd( vector<int>& values )
{
    vector<int>::iterator p = values.begin();
    while ( p != values.end() )
    {
        if ( *p % 2 == 1 )
            p = values.erase( p );
        else
            ++p;
    }
}
```

(At this point we are ignoring the existence of the `remove_if` function.)

- The version for lists is identical, except for the substitution of the word `vector` for the word `list`.

- Suppose you ran this on a list or vector of 100,000 integers (about half of which are odd). Which function would be faster? Would it matter? Why?

Why Do We Analyze Functions and Algorithms?

- We want to do better than just implementing and testing every idea we have.

- We want to know why one algorithm is better than another.

- We want to know the best we can do. (This is often quite hard.)
How Do We Analyze Functions and Algorithms?

We will look at the pros and cons of several different algorithm analysis options:

1. Don’t do any analysis; just use the first algorithm you can think of that works.

2. Implement and time algorithms to choose the best.

3. Analyze algorithms by counting operations of different types and assign different weights to different types based on how long each takes.

4. Analyze algorithms by assuming each operation requires the same amount of time. Count the total number of operations, and then multiply this count by the average cost of an operation.

What Happens In Practice?

- 99% of the time: rough count similar to #4 as a function of the size of the data. Use order notation to simplify the resulting function and even to simplify the analysis that leads to the function.

- 1% of the time: implement and time.

Counting Example

Suppose foo is a vector of doubles, initialized with a sequence of values.

- Here is a simple algorithm to find the sum of the values in the vector:

```c
double sum = 0;
for ( int i=0; i<foo.size(); ++i )
    sum += foo[i];
```

- How do you count the total number of operations?

- Go ahead and try. Come up with a function describing the number of operations. Let n stand for the size of the vector.

- You are likely to come up with different answers. How do we resolve these differences?
**Order Notation**

The following discussion emphasizes intuition. That’s all we care about in CS II. For more details and more technical depth, see the Weiss text book.

- **Definition**

  Algorithm $A$ is order $f(n)$ — denoted $O(f(n))$ — if constants $k$ and $n_0$ exist such that $A$ requires no more than $k \times f(n)$ time units to solve a problem of size $n \geq n_0$.

- As a result, algorithms requiring $3n + 2$, $5n - 3$, $14 + 17n$ operations are all $O(n)$ (i.e. in applying the definition of order notation $f(n) = n$).

- Algorithms requiring $n^2/10 + 15n - 3$ and $10000 + 35n^2$ are all $O(n^2)$ (i.e. in applying the definition of order notation $f(n) = n^2$).

- Intuitively (and importantly), we determine the order by finding the asymptotically dominant term (function of $n$) and throw out the leading constant. This term could involve logarithmic or exponential functions of $n$.

- **Implications for analysis:**
  - We don’t need to quibble about small differences in the numbers of operations.
  - We also do not need to worry about the different costs of different types of operations.
  - We don’t produce an actual time. We just obtain a rough count of the number of operations. This count is used for comparison purposes.

- In practice, this makes analysis relatively simple, quick and (sometimes unfortunately) rough.
Common Orders of Magnitude

Here are the most commonly occurring orders of magnitude in algorithm analysis.

- $O(1)$: The number of operations is independent of the size of the problem.
- $O(\log n)$
- $O(n)$
- $O(n \log n)$
- $O(n^2)$
- $O(n^2 \log n)$
- $O(n^3)$
- $O(2^n)$

Significance of Orders of Magnitude

- On a computer that performs $10^8$ operations per second:
  - An algorithm that actually requires $15n \log n$ operations requires about 3 seconds on a problem of size $n = 1,000,000$, and 50 minutes on a problem of size $n = 100,000,000$.
  - An algorithm that actually requires $n^2$ operations requires about 3 hours on a problem of size $n = 1,000,000$, and 115 days on a problem of size $n = 100,000,000$.

- Thus, the leading constant of 15 on the $n \log n$ does not make a substantial difference. What matters is the $n^2$ vs. the $n \log n$.

- Moreover, in practice the leading constants usually do not vary by a factor of 15.

Back to Analysis: A Slightly Harder Example

- Here’s an algorithm to determine if the value stored in variable $x$ is also in a vector called $\text{foo}$
int loc=0;
bool found = false;
while ( !found && loc < foo.size() )
{
    if ( x == foo[loc] )
        found = true;
    else
        loc ++ ;
}
if ( found ) cout << "It is there!\n";

• Can you analyze it? What did you do about the if statement? What did you assume about where the value stored in x occurs in the vector (if at all)?

Best-Case, Average-Case and Worst-Case Analysis

• For a given fixed size vector, we might want to know:
  – The fewest number of operations (best case) that might occur.
  – The average number of operations (average case) that will occur.
  – The maximum number of operations (worst case) that can occur.

• The last is the most common. The first is rarely used.

• On the previous algorithm, the best case is $O(1)$, but the average case and worst case are both $O(n)$.

Approaching An Analysis Problem

• Decide the important variable (or variables) that determine the “size” of the problem.
  – For container classes this will generally be the number of values stored.

• Decide what to count. The order notation helps us here.
  – If each loop iteration does a fixed (or bounded) amount of work, then we only need to count the number of loop iterations.
  – We might also count specific operations, such as comparisons.

• Do the count, using order notation to describe the result.
Examples: Loops

• Version A:

```c
int count=0;
for ( int i=0; i<n; ++i )
    for ( int j=0; j<n; ++j )
        ++count;
```

• Version B:

```c
int count=0;
for ( int i=0; i<n; ++i )
    ++count;
for ( int j=0; j<n; ++j )
    ++count;
```

• Version c:

```c
int count=0;
for ( int i=0; i<n; ++i )
    for ( int j=i; j<n; ++j )
        ++count;
```

• How many operations in each?
Erasing from a Vector vs. Erasing from a List

- Recall the `erase_odd` function we started class with:

```c++
void erase_odd( vector<int>& values )
{
    vector<int>::iterator p = values.begin();
    while ( p != values.end() )
    {
        if ( *p % 2 == 1 )
            p = values.erase( p );
        else
            ++p;
    }
}
```

- Here's code that implements `erase` in a vector.

```c++
template <class T>
Vec<T> :: iterator Vec<T> :: erase( iterator p )
{
    for ( iterator q = p; q < data_end-1; ++q )
        *q = *(q+1);
    -- data_end;
    return p;
}
```

- Letting \( n \) be the size of the vector, what is the cost of each call to `erase`? What is the overall cost of `erase_odd`?

- The `erase` function for lists just cuts the item indicated by the iterator out of the chain. (We will see how this chain is built starting Thursday.)

- It is easy to see intuitively that this is an \( O(1) \) operation, independent of the size of the list!

**Parting Question**

Why doesn’t the `vector` container class have a `push_front` member function?