CS61B, Spring 2003 **Discussion #7** **Amir Kamil**
UC Berkeley **3/10/03**

Topics: Efficiency, Hashing

1 Efficiency

1.1 Motivation

- Since there are often many possible algorithms or programs that compute the same results, we would like to use the one that is fastest.
- How do we decide how fast an algorithm is? Since knowing how fast an algorithm runs for a certain input does not reveal anything about how fast it runs on other inputs, we need another measure that tells us how fast it is for any input. A formula that relates input size to the running time of the algorithm satisfies this requirement.
- We also want to ignore machine dependent factors. If an algorithm takes two seconds on one machine for a given input, a trivial way to get it to run in one second is to use a machine that is twice as fast. There is a constant multiplicative factor relating the speed of an algorithm on one machine and its speed on another, which we will ignore.
- We are only interested in how fast an algorithm runs on large inputs, since even slow algorithms finish quickly on small inputs.

1.2 Asymptotic Analysis

Note: You do not have to know the material in this section in much detail, but it does provide some background for algorithm analysis.

- Rather than specifying the exact relation between an algorithm's input and its running time, we only specify how the running time scales as the input grows. For example, if the running time for an algorithm with input n is $4n^2$, we say that it's running time scales as n^2.
- Also rather than giving the exact relation, we are usually interested in limits on how fast or slow an algorithm is. So we define the following notation:
  1. We say that $f(n)$ is bounded above by $g(n)$ if for all $n > M$ and for some $K > 0$, $K \cdot |g(n)| \geq |f(n)|$. In words, $g(n)$ is an upper bound for $f(n)$ if some positive multiple of $|g(n)|$ is always greater than or equal to $|f(n)|$ after some arbitrary number M. Notice that this definition ignores both constant multiplicative factors and behavior for small inputs.
  2. Similarly, we say that $f(n)$ is bounded below by $g(n)$ if for all $n > M$ and for some $K > 0$, $K \cdot |g(n)| \leq |f(n)|$. In words, $g(n)$ is a lower bound for $f(n)$ if some positive multiple of $|g(n)|$ is always less than or equal to $|f(n)|$ after some arbitrary number M.
  3. We define a set of functions $O(g)$ such that $g(n)$ provides a lower bound for all functions in $O(g)$. In other words, $f(n) \in O(g)$ if $g(n)$ is a lower bound for $f(n)$.
  4. We define a set of functions $\Omega(g)$ such that $g(n)$ provides an lower bound for all functions in $\Omega(g)$. In other words, $f(n) \in \Omega(g)$ if $g(n)$ is a lower bound for $f(n)$.
  5. We define a set of functions $\Theta(g)$ such that $g(n)$ provides both an upper bound and a lower bound for all functions in $\Theta(g)$. In other words, $f(n) \in \Theta(g)$ if $g(n)$ is both an upper bound and a lower bound for $f(n)$.
- Now we can specify the speed of an algorithm by giving functions $g(n)$ and $h(n)$ such that its running time is in $O(g)$ and in $\Omega(h)$. If $g(n) = h(n)$, then its running time is in $\Theta(g)$.

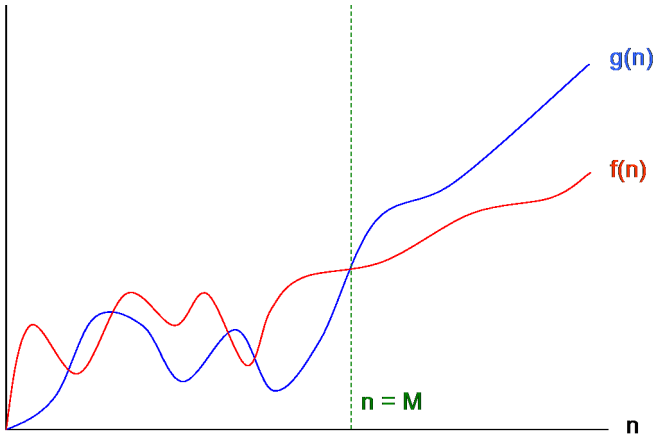

Figure 1: Functions $f(n)$ and $g(n)$ such that $f(n) \in O(g)$.

- Sometimes we only care about an upper bound on the running time of an algorithm, so we only give $g(n)$.
- Note that the above notation works for arbitrary functions $f(n)$ and $g(n)$. Using it to give running times of an algorithm is only a specific case of its usage. To familiarize ourselves with the notation, let's do some examples on arbitrary functions.

  1. Let $f(n) = 3n + 2$. Let $g(n) = n$. Then $f(n) \in \Theta(g)$ since $4 \cdot g(n) \geq f(n)$ for all $n > 1$ and $2 \cdot g(n) \leq f(n)$ for all $n > 1$.
  2. Let $f(n) = n^2$ and $g(n) = 1000n$. Then $f(n) \in \Omega(g)$ since $g(n) \leq f(n)$ for all $n > 1000$. Note that $f(n) \notin O(g)$ since you can't find K and M such that $K \cdot g(n) \geq f(n)$ for all $n > M$ (try it!).
  3. In the figure 1, $g(n) \in \Omega(f)$, and $f(n) \in O(g)$.
  4. Let $f(n) = log_2 n$ and $g(n) = log_9 n$. Is $f(n)$ in $O(g)$?
     Recall the identity:
     $$log_a b = \frac{log_c b}{log_c a}$$
     Thus, $f(n) = C \cdot g(n)$, $C = log_2 9$, and $f(n) \in \Theta(g)$.
  5. Let $f(n) = 1.01^n$ and $g(n) = n^2$. Is $f(n)$ in $O(g)$?
     Sometimes it helps to take the logarithms of both functions to decide which one is bigger:
     $$\begin{gathered} 1.01^n \ ? \ n^2 \\ \lg(1.01^n) \ ? \ \lg(n^2) \\ n \cdot \lg 1.01 \ ? \ 2 \cdot \lg n \end{gathered}$$
     Since $C_1 \cdot n \notin O(C_2 \cdot \lg n)$, $f(n) \notin O(g)$.
  6. Let $f(n) = 3^n$ and $g(n) = 2^n$. Is $f(n)$ in $O(g)$?
     We can try taking logarithms again:
     $$\begin{gathered} 3^n \ ? \ 2^n \\ \lg(3^n)) \ ? \ \lg(2^n)) \\ n \cdot \lg 3 \ ? \ n \cdot \lg 2 \end{gathered}$$
     We might think that since $C_1 \cdot n \in O(C_2 \cdot n)$, $f(n) \in O(g)$. However, be careful! When in doubt, check the definition. Can you find K and M such that $K \cdot 2^n \geq 3^n$ for all $n > M$? You will find that you can't, so in actuality, $f(n) \notin O(g)$.
     There is actually a simpler way to answer this question without actually finding K and M. It is the case that
     $$\lim_{n \to \infty} \frac{f(n)}{g(n)} < \infty \iff f(n) \in O(g(n)).$$

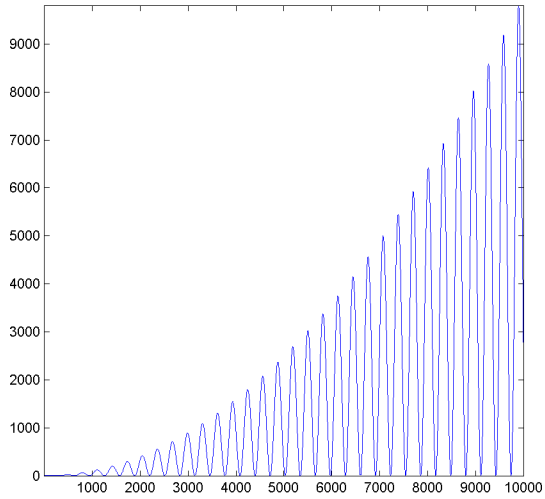

Figure 2: Graph of $f(n) = n^2 \cdot \sin^2 n + 0.001$.

Since $\lim_{n\to\infty} \frac{3^n}{2^n} = \lim_{n\to\infty} (\frac{3}{2})^n = \infty$, $3^n \notin O(2^n)$.

7. Is there a function $f(n) > 0$ such that $f(n) \notin O(n)$ and $f(n) \notin \Omega(n)$?
   The function in figure 2, $f(n) = n^2 \cdot \sin^2 n + 0.001$ satisfies the above criteria.

1.3 Algorithm Analysis

1.3.1 Iterative Algorithms

The running time of iterative algorithms is straightforward to compute. Let $f_1(n)$ be the time it takes one iteration of the algorithm to run, and let $f_2(n)$ be the number of iterations. Then the running time of the algorithm is $O(f_1 \cdot f_2)$. Examples:

1. ```
   int increment(int n) {
     return n + 1;
   }
   ```
   This function has one subexpression that takes constant time to execute and executes only once. So it runs in $O(1)$.

2. ```
   int factorial(int n) {
     for (int i = n; i > 0; i++) {
       n *= i;
     }
     return n;
   }
   ```
   This function has a subexpression `n *= i` that takes constant time to execute, and this subexpression is executed n times. So $f_1(n) = 1$, $f_2(n) = n$, and the function runs in $O(n)$ time.

3. ```
   int foo(int n) {
     int x;
     for (int i = 0; i < n; i++) {
       x += i;
     }
     for (int i = 1; i < n/2; i++) {
       x *= i;
     }
   ```
   ```

```
  return x;
}
```

In this case, there are two loops. The first runs in $O(n)$, and the second in $O(\frac{n}{2}) = O(n)$, so the total running time is in $O(n)$.

4.
```
int bar(int n) {
  int x;
  for (int i = 0; i < n; i++) {
    for (int j = i; j < n; j++) {
      x += 1;
    }
  }
  return x;
}
```

This function has one subexpression, the inner loop, that executes n times. What is the running time of the subexpression? The subexpression has one subexpression of its own that executes $n-i$ times and is constant. So the running time of the inner loop is in $O(n-i)$. Now the problem with determining the running time of the function is that i varies. But we can make estimates, as long as the estimates are greater than the actual value, so let's assume that the running time of the inner loop is n. Now the inner loop executes n times, so the total running time is in $O(n^2)$.

5.
```
int baz(int k, int n) {
  int res = 0;
  for (int i = 0; i < k; i++) {
    res += (k - i) * k;
  }
  for (int i = 0; i < n; i++) {
    res -= (n - i) * i;
  }
  return res;
}
```

This functions has two loops, the first of which is in $O(k)$ and the second of which is in $O(n)$. We don't know which of the two loops is faster, since it depends on the relative sizes of k and n, so we can only say that the function runs in $O(k+n)$. It is also possible to say that the function runs in $O(max(k,n))$ since we can give an upper bound on the faster loop by assuming it runs in the same amount of time as the slower loop. Note that in general, it is not possible to give the running time of a multiple input function in terms of only one of its inputs.

6.
```
int foobar(int k, int n) {
  int res = 0;
  for (int i = 0; i < k; i++) {
    for (int j = 0; j < n; j++) {
      res += (k - i) * j;
    }
  }
  return res;
}
```

The inner loop runs in $O(n)$ time, and the outer loop iterates k times, so the running time of this function is in $O(k \cdot n)$.

1.3.2 Recursive Algorithms

Recursive algorithms are somewhat harder to analyze than iterative algorithms. They usually require inductive analysis. We start at the base case and work our way up higher inputs until we see a pattern. One

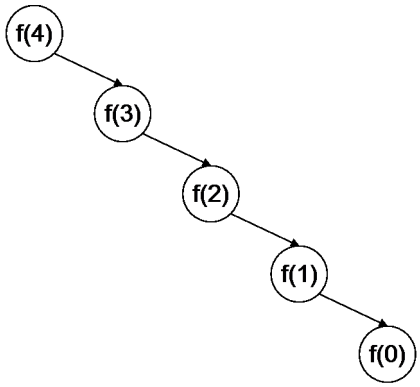

Figure 3: Tree of recursive calls for `factorial(4)`.

way that helps is to draw a tree of the recursive calls, with each call as a node and an edge between the caller and the callee. We then count how many nodes are in the tree as a function of the input. Then the running time of the algorithm is the number of nodes in the tree times the amount of time each call takes (not including the recursive calls each each call makes). Examples:

1. ```
   int factorial2(int n) {
     if (n == 0) {
       return 1;
     } else {
       return n * factorial2(n - 1);
     }
   }
   ```

   We draw a tree of the recursive calls in figure 3. About n recursive calls are made, and each call takes constant time, so the running time of `factorial()` is in $O(n)$.

2. ```
   int fibonacci(int n) {
     if (n == 0 || n == 1) {
       return 1;
     } else {
       return fibonacci(n - 1) + fibonacci(n - 2);
     }
   }
   ```

   Again we draw a tree of the recursive calls in figure 4. The tree is a nearly complete binary tree, so it has about 2^n nodes in it. So the running time of this function is in $O(2^n)$.

2 Hashing

Searching for an element in an unsorted array or linked list takes linear time, which is a little slow. Binary search trees (which we haven't done yet) provide logarithmic search times on average, but we would like to do even better. For integer keyed items, we can trivially get constant search time by using an array, with one position for each possible key. However, with 2^{32} possible keys, there's no way we could create an array large enough.

Instead, we could get almost constant search time if we create an array twice as large as our data set, and store a value k at position $k \, mod \, m$, where m is the size of the array. To search for a value k, we again compute k mod m and then check the corresponding array position. We would expect an even distribution of values, so we expect one or two values per array position. To retain this expectation, we may need to resize the array as we insert values. We want to keep the *load factor*, $\frac{k}{m}$, at about 0.5. Resizing potentially

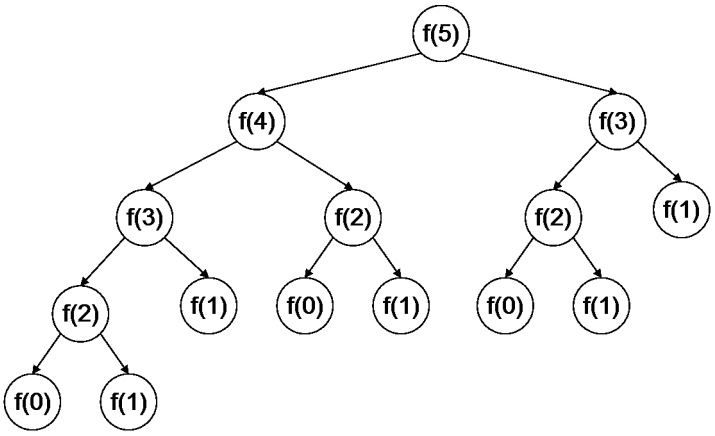

Figure 4: Tree of recursive calls for `fibonacci(5)`.

requires us to move all values, but the total amortized cost of insertion is still constant. Searching is also constant in this scheme.

There is the problem of multiple values mapping to the same array position, or *bucket*. This situation is called a *collision*. There are multiple ways of resolving collisions. One way is to place a value that maps to an occupied bucket to the next empty bucket. Another is to have linked lists at each bucket, and store values in the lists. This is the solution we'll use.

This is fine for integer values, but what about arbitrary objects? We first compute an integer corresponding to the object, called a *hash code*, and use that to map the objects to positions. There are two conditions on a hash code. Equal objects (as defined by the `equals()` method) must have equal hash codes, and the hash code for an object cannot change while it is in a *hash table*. Then, assuming the hash code can be computed quickly and distributes objects evenly in a table, this gives us constant-time searching for arbitrary objects.

How should we compute a hash code on an object? Many objects can be represented as k-tuples of integers. For example, an object corresponding to a point in two-dimensional space can be represented by a double of integers, converting the floating point x and y values bitwise to integers. A good hash code for a k-tuple $(x_0, x_1, ..., x_{k-1})$ is

$$x_0 \cdot a^{n-1} + x_1 \cdot a^{n-2} + ... + x_{n-2} \cdot a + x_{n-1}$$

where a is a constant. Choosing a prime number minimizes the chance of a collision.

Our hash code formula works even if k is unbounded, such as in strings. Java's `String` class uses the following formula:

```
public int hashCode() {
  int code = 0;
  for (int i = 0; i < length(); i++) {
    code = 31 * code + charAt(i);
  }
  return code;
}
```

This is our formula, with $a = 31$ and $x_i =$`charAt(i)`.

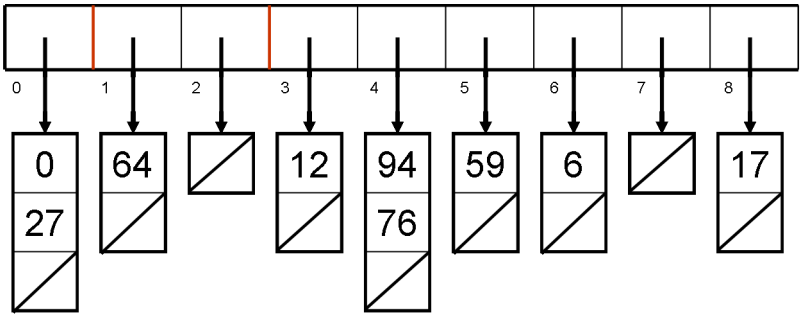

Figure 5: A fast integer lookup table, with a load factor of 1.