Analysis of Algorithms

Running Time Calculations

With C language examples

The Model

- In order to analyze algorithms in our formal framework, we need a model of computation.
- We assume infinite memory. This won’t take into account effects like page faults.
- Our model has the standard repertoire of simple instructions, such as addition, multiplication, comparison, and assignment. We can:
  - Count the number of times all of these operations are performed, or
  - Decide which operation is the most expensive and count that instead.

The Model continued…

- The most important resource to analyze is the running time.
  - Although the compiler and computer affect these results, we will not model them here.
  - Instead we focus primarily on the algorithm (not necessarily the program) and the input to the algorithm. Typically the size of the input (N) is the main consideration.

The Model continued…

- We define two functions, $T_{avg}(N)$ and $T_{worst}(N)$ as the average and worst-case running time of the algorithm.
- The average running time is much harder to compute, let alone define.
  - For example, what is the "average" input to the algorithm? This is not always well defined.
- Generally the quantity required is the worst-case time, since this provides a bound for all input.
  - We'll concentrate on this, namely, a "Big-Oh" estimate.

A Simple Example

$$\sum_{i=1}^{N} i^3$$

```
int sum (int N)
{
   int partial_sum = 0;
   for (int i = 1; i <= N; i++)
           partial_sum = partial_sum +
                    (i * i * i);
   return partial_sum;
}
```

Time Units to Compute

----

1 for the assignment.

1 assignment, *N+1* tests, and *N* increments.

***N* loops of 4 units for an assignment, an addition, and two multiplications.**

1 for the return statement.

Total: $1+(1+N+1+N)+4N+1 = 6N+4 = O(N)$

Analysis too complex – there are ways to simplify this…

A Simpler Analysis

$$\sum_{i=1}^{N} i^3$$

```
int sum (int N)
{
    int partial_sum = 0;
    for (int i = 1; i <= N; i++)
            partial_sum = partial_sum +
                        (i * i * i);
    return partial_sum;
}
```

Time Units to Compute

N loops

Two multiplications per loop

Total: $2N = O(N)$

This time we focus on the multiplications, which are the most expensive operation. The Big-Oh is the same.

Another Example

```
void Compress()
{
    for (k = 1; k <= N; k++) {
      Q1[i][k] = (M[i]*Q[i][k] +
                      M[j]*Q[j][k]) /
                    (M[i]+M[j]);
      Q1[j][k] = 0.0;
      Q2[i][k] = Q1[i][k];
      Q2[j][k] = 0.0;
    }
}
```

Time Units to Compute

N loops

Two multiplications and one division.

Total: $3N = O(N)$

This time we focus on the multiplications and division, which are the most expensive operations.

Another Example

```
void checkZ()
{
    unsigned int i, j, temp;

    for (i = 1; i <= N; i++) {
      temp = 0;
      for (j = 0; j < M; j++) {
          temp = temp * Z[i][j];
      }
      if (temp != n) printf("Error\n");
    }
}
```

Time Units to Compute

N loops

M loops

One multiplication per loop

Total: $NM = O(NM)$

This time we focus on the multiplications, which are the most expensive operation. Note the nested loops.

General Rules

- The rest of the lecture will provide general rules for analyzing the running time of your code.
  - We will look at intuitive rules.
  - We will also look at more formal rules of how to translate the running time of your code into mathematical expressions.

General Rules: For Loops

- The running time of a for loop is at most the running time of the statements inside the for loop (including tests) times the number of iterations.

```
for (int i = 1; i <= N; i++) {
      sum = sum+i;
}
```

- The above example is *O(N).*

For Loops: Formally

- In general, a for loop translates to a summation. The index and bounds of the summation are the same as the index and bounds of the for loop.

```
for (int i = 1; i <= N; i++) {
      sum = sum+i;
}
```

$$\sum_{i=1}^{N} 1 = N$$

- Suppose we count the number of additions that are done. There is 1 addition per iteration of the loop, hence *N* additions in total.

General Rules: Nested Loops

- Analyze these inside out. The total running time of a statement inside a group of nested loops is the running time of the statement multiplied by the product of the sizes of all the loops.

```
for (int i = 1; i <= N; i++) {
      for (int j = 1; j <= M; j++) {
            sum = sum+i+j;
      }
}
```

- The above example is *O(MN).*

Nested Loops: Formally

- Nested for loops translate into multiple summations, one for each for loop.

```
for (int i = 1; i <= N; i++) {
    for (int j = 1; j <= M; j++) {
        sum = sum+i+j;
    }
}
```

$$\sum_{i=1}^{N}\sum_{j=1}^{M} 2 = \sum_{i=1}^{N} 2M = 2MN$$

- Again, count the number of additions. The outer (inner) summation is for the outer (inner) for loop.

General Rules: Consecutive Statements

- Consecutive statements: **These just add (which means that the maximum is the one that counts).**

```
for (int i = 1; i <= N; i++) {
    sum = sum+i;
}
for (int i = 1; i <= N; i++) {
    for (int j = 1; j <= N; j++) {
        sum = sum+i+j;
    }
}
```

- The above example is $O(N^2+N) = O(N^2)$.

Consecutive Statements: Formally

- Add the running times of the separate blocks of your code

```
for (int i = 1; i <= N; i++) {
    sum = sum+i;
}
for (int i = 1; i <= N; i++) {
    for (int j = 1; j <= N; j++) {
        sum = sum+i+j;
    }
}
```

$$\left[\sum_{i=1}^{N} 1\right] + \left[\sum_{i=1}^{N}\sum_{j=1}^{N} 2\right] = N + 2N^2$$

General Rules: Conditionals

- If (test) s1 else s2: The running time is never more than the running time of the test plus the larger of the running times of s1 and s2.

```
if (test == 1) {
    for (int i = 1; i <= N; i++) {
        sum = sum+i;
}}
else for (int i = 1; i <= N; i++) {
        for (int j = 1; j <= N; j++) {
            sum = sum+i+j;
}}
```

- The above example is $O(N^2)$.

Conditionals: Formally

- If (test) s1 else s2: Compute the maximum of the running time for s1 and s2.

```
if (test == 1) {
    for (int i = 1; i <= N; i++) {
        sum = sum+i;
}}
else for (int i = 1; i <= N; i++) {
        for (int j = 1; j <= N; j++) {
            sum = sum+i+j;
}}
```

$$\max\left(\sum_{i=1}^{N} 1, \sum_{i=1}^{N}\sum_{j=1}^{N} 2\right) =$$

$$\max\left(N, 2N^2\right) = 2N^2$$

General Rules: Conditionals + Loops

- If you have a conditional inside a loop, things can get more hairy:

```
for (int i = 1; i <= N*N; i++) {
      if (i%N == 0) {
            foo();
}     }
```

- Count the number of times that foo() is called. It **looks** $O(N^2)$, right? But it isn't. Why?

Conditionals + Loops: Formally

- The conditional can dramatically reduce the number of times the code is actually called.

```
for (int i = 1; i <= N*N; i++) {
    if (i%N == 0) {
        foo();
}   }
```

→

```
for (int i = N; i <= N*N; i = i+N) {
    foo();
}
```

Foo() is called only when *i* is a multiple of *N*, from *N* to *N*N*. So, really, this is only *N* times.

General Rules: Recursion

- Basic strategy: analyze from the inside (or deepest part) first and work outwards. If there are function calls, these must be analyzed first. This even works for recursive functions:

```
long factorial (int n) {
   if (n <= 1)
       return 1;
  else
      return n * factorial(n-1);
}
```

Time Units to Compute

----

1 for the multiplication statement.
What about the function call?

- This clearly **looks** like a linear-time algorithm, right? In other words, the function will be called recursively *N* times.

Recursion: Formally

- Recursive functions are described with “recurrence relations”. These are too difficult for us now – we’ll look at these formally later in the class. But, let’s examine some simpler ones now (like factorial):

```
long factorial (int n) {
    if (n <= 1)
        return 1;
   else
       return n * factorial(n-1);
}
```

$$T(N) = 1 + T(N-1) =$$
$$2 + T(N-2) =$$
$$3 + T(N-3) = \ldots = N$$

- Let the running time of *factorial(N) = T(N),* and count the number of multiplications that are done.

Another Recursive Example: Fibonacci

```
long F(int n) {
    if (n <= 1)
        return 1;
   else
       return F(n-1)+F(n-2);
}
```

Time Units to Compute

----

1 for the comparison.

1 for the addition.

What about the function calls?

- Let the running time of F$(N) = T(N)$. Then $T(N) = T(N-1) + T(N-2) + 2$. Can we give a lower bound on $T(N)$ from this? Note that $T(N) > T(N-1) + T(N-2)$.

F(0) = 1, F(1) = 1, F(2) = 2, F(3) = 3, F(4) = 5, F(5) = 8, F(6) = 13,…

Fibonacci Analysis

Let $F(N)$ be the Nth Fibonacci number.

We can prove that (1) $T(N) \geq F(N)$ and (2) $F(N) \geq (3/2)^N$. Thus $T(N) \geq (3/2)^N$, which means the running time grows exponentially. This is quite bad.

Recall Proof by Induction

- Proof by (strong) induction:
  - Show theorem true for trivial case(s). Then, assuming theorem true up to case N, show true for $N+1$. Thus true for all N.

Proof that T(N) >= F(N)

Base cases: $T(0) = 1 \geq F(0) = 1,$

$T(1) = 1 \geq F(1) = 1.$

$T(2) = 4 \geq F(2) = 2.$

We know that $T(N+1) > T(N) + T(N-1)$

and $F(N+1) = F(N) + F(N-1).$

Assume theorem holds for all $k, 1 \leq k \leq N$

Now prove for the $N+1$ case:

$T(N+1) > T(N) + T(N-1) \geq F(N) + F(N\text{-}1) = F(N+1)$

Proof that F(N) >= (3/2)^N

Base cases: $F(5) = 8 \geq (3/2)^5 = 7.6,$

$F(6) = 13 \geq (3/2)^6 = 11.4.$

Assume theorem holds for all $k, 1 \leq k \leq N.$

Now prove for the $N+1$ case:

$F(N+1) = F(N) + F(N\text{-}1) \geq (3/2)^N + (3/2)^{N-1} =$

$(3/2)^N (1 + (2/3)) = (3/2)^N (5/3) >$

$(3/2)^N (3/2) = (3/2)^{N+1}.$