Exam 1 Review

Topics For Exam 1

Topics	Reading
Introduction	1.1-1.3
Induction and Loop invariants	1.4-1.7, GT 1.3
Elementary Algorithmics	Chapter 2
Asymptotic Notation	Chapter 3
Algorithm Analysis - Analyzing control structures - Worst-case and Average-case - Amortized analysis	4.1-4.6
Solving Recurrences	4.7
Exam 1	

Induction Proof

- Mastering
  - First and second principles of induction
  - Given a mathematical equation, know how to prove it by induction
    - Example: prove by induction that $\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$
- Exposure
  - Constructive induction

Invariant

To prove some statement S about a loop is correct, define in terms of a series of smaller statement $S_0, S_1 \ldots, S_k$ where:

- The initial claim, S_0, is true before the loop begins.
- If S_{i-1} is true before iteration i begins, then one can show that S_i will be true after iteration i is over or at the beginning of loop $i+1$.
- The final statement, S_k, implies the statement S that we wish to justify as being true.

This is essentially an induction proof. The proof is for a loop iterating from 1 to k. It's trivial to expand this argument to other loop bounds.

Loop Invariant: Example

- Prove the following loop find the max(a[0], …, a[n-1])

```
int max(int a[n])
{
   int max = a[0];
   int i;

   for (i=1; i<=n-1; i++)
     if (max < a[i])
         max = a[i];

    return max;
}
```

Elementary Algorithmics

- Given a problem
  - What's an instance
  - Instance size
- What does efficiency mean?
  - Time

Average and worst-case analysis

- How to compare two algorithms
  - Worst case, average, best-case
- Worst case
  - Appropriate for an algorithm whose response time is critical
- Average
  - For an algorithm which is to be used many times on many different instances
  - Harder to analyze, need to know the distribution of the instances
- Best case

Elementary Operation

- An elementary operation is one whose execution time can be bounded above by a constant depending only on the particular implementation—the machine, the programming language, etc.

- Example
  - X = Sum{A[i] | 1 <= i <= n}
  - Fibonacci sequence, addition may not be an elementary operation

Asymptotic Notation

- What does "the order of" mean
- Big O, Ω, and Θ notations
- Properties of asymptotic notation
- Limit rule
- Duality rule
- Smooth and b-smooth

Asymptotic notations

- Know the definitions of big O, Ω, and Θ notations
  - Example: what does $O(n^2)$ mean?
- Know how to prove whether a function is in big O, Ω, or Θ based on definition
  - Example
    - Prove that if $f(n) \in O(g(n))$ then $g(n) \in \Omega(f(n))$

Maximum, Duality and Limit rules

- Know to prove asymptotic relationship using the rules
  - Example
    - Show that $O((n+1)^2) = O(n^2)$

The Maximum rule

- Let $f, g : N \to R^{\geq 0}$,

  then $O(f(n) + g(n)) = O(\max(f(n), g(n)))$

- Examples
  - $O(12n^3\text{-}5n\text{+}n\log n\text{+}36) = O(n^3)$

- The maximum rule let us ignore lower-order terms

The Duality Rule

$t(n) \in \Omega(f(n))$

iff

$f(n) \in O(t(n))$

Example: $\sqrt{n} \in \Omega(\log n)$

We can apply, similarly, the limit rule, the maximum rule, and the threshold rule for Ω using the duality rule

The Limit Rule

- Let $f, g : N \to R^{\geq 0}$, then

1. If $\lim_{n\to\infty} \frac{f(n)}{g(n)} \in R^+$ then $f(n) \in O(g(n))$ and $g(n) \in O(f(n))$
2. If $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$ then $f(n) \in O(g(n))$ and $g(n) \notin O(f(n))$
3. If $\lim_{n\to\infty} \frac{f(n)}{g(n)} = +\infty$ then $f(n) \notin O(g(n))$ and $g(n) \in O(f(n))$

Proof: use the definition of limit and big-O

The Limit Rule

- Let $f, g : N \to R^{\geq 0}$, then

1. If $\lim_{n\to\infty} \frac{f(n)}{g(n)} \in R^+$ then $f(n) \in \Theta(g(n))$
2. If $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$ then $f(n) \in O(g(n))$ but $f(n) \notin \Theta(g(n))$
3. If $\lim_{n\to\infty} \frac{f(n)}{g(n)} = +\infty$ then $f(n) \in \Omega(g(n))$ but $f(n) \notin \Theta(g(n))$

Semantics of big-O and Ω

- When we say an algorithm takes worst-case time *t(n)* ∈ O(*f(n)*), then there exist a real constant *c* such that *c*f(n)* is an upper bound for any instances of size of sufficiently large *n*
- When we say an algorithm takes worst-case time *t(n)* ∈ Ω(*f(n)*), then there exist a real constant *d* such that there exists at least one instance of size *n* whose execution time >= *d*f(n)*, for any sufficiently large *n*
- Example
  - Is it possible an algorithm takes worst-case time O(*n*) and Ω(*nlog n*)?

Practice Problems

anAlgorithm(int n)
{
  // if (x) is an elementary
  // operation
  if (x) {
    some work done
    by n^2 elementary
    operations;
  } else {
    some work done
    by n^3 elementary
    operations;
  }
}

- True or false
  - The algorithm takes time in $O(n^2)$ F
  - The algorithm takes time in $\Omega(n^2)$ T
  - The algorithm takes time in $O(n^3)$ T
  - The algorithm takes time in $\Omega(n^3)$ F
  - The algorithm takes time in $\Theta(n^3)$ F
  - The algorithm takes time in $\Theta(n^2)$ F
  - The algorithm takes worst case time in $O(n^3)$ T
  - The algorithm takes worst case time in $\Omega(n^3)$ T
  - The algorithm takes worst case time in $\Theta(n^3)$ T
  - The algorithm takes best case time in $\Omega(n^3)$ F

Smooth

- Know the definition of smooth and how to prove if a function is smooth or not
  - Example: what does b-smooth mean?
  - Prove that n^2 is smooth
- A function $f : N \to R^{\geq 0}$ is *eventually nondecreasing* if there exists an integer threshold n_0 such that $f(n) \leq f(n+1)$ for all $n \geq n_0$
- Function f is *b-smooth* (b is an integer >1) if it is eventually nondecreasing and it satisfies condition $f(bn) \in O(f(n))$
- A function is *smooth* if it is b-smooth for every integer b>=2
- **Theorem**: If a function is b-smooth for any b>=2, it is smooth

Exposure: Smoothness rule

- Let $f : N \to R^{\geq 0}$ be a smooth function and let $t : N \to R^{\geq 0}$ be an eventually nondecreasing function. Then $t(n) \in \Theta(f(n))$ whenever $t(n) \in \Theta(f(n) \mid n \text{ is power of } b)$
- The rule holds for O and Ω

Analysis of Algorithms

- Mastering
  - Analyzing control structures
    - Sequencing
    - For loops
    - While and repeat loops
    - Recursive calls
  - Finding and using a barometer
  - Average case analysis
- Exposure
  - Amortized analysis

Control structures: sequences

- P is an algorithm that consists of two fragments, P1 and P2

```
P
{
   P1;
   P2;
}
```

- P1 takes time t1 and P2 takes times t2
- The sequencing rule asserts P takes time $t=t1+t2 \in \Theta(\max(t1,t2))$.

Control structures: sequences

- P is an algorithm that consists of two fragments, P1 and P2

```
P
{
   P1;
   P2;
}
```

- P1 takes time t1 and P2 takes times t2
- The sequencing rule asserts P takes time $t=t1+t2 \in \Theta(\max(t1,t2))$.

For loops

```
for (i=0; i<m; i++) {
  P(i);
}
```

- Case 1: P(i) takes time *t* independent of i and n, then the loop takes time *O(mt)* if m>0.
- Case 2: P(i) takes time *t(i),* the loop takes time $\sum_{i=0}^{m-1} t(i)$

Example: analyzing the following nests

```
for (i=0; i<n; i++) {
  for (j=0; j<n; j++)
    constant work
}
```

```
for (i=1; i<n; i++) {
  for (j=0; j<i; j++)
    constant work
}
```

```
for (i=1; i<n; i++) {
  for (j=0; j<i*i; j++)
    constant work
}
```

```
for (i=1; i<n; i++) {
  for (j=0; j<i; j++)
    constant work

  for (k=0; k<i*i; k++)
    constant work
}
```

“while” and “repeat” loops

- The bounds may not be explicit as in the for loops
- Careful about the inner loops
  - Is it a function of the variables in outer loops?
- Analyze the following two algorithms

```
int example1(int n)
{
  while (n>0) {
      work in constant;
      n = n/3;
  }
}
```

```
int example2(int n)
{
  while (n>0) {
      for (i=0; i<n; i++) {
         work in constant;
      }
      n = n/3;
  }
}
```

Recursive calls

Typically we can come out a recurrence equation to mimics the control flow.

```
double fibRecursive(int n)
{
 double ret;
 if (n<2)
   ret = (double)n;
 else
   ret = fibRecursive(n-1)+fibRecursive(n-2);
 return ret;
}
```

$$T(n) = \begin{cases} a & \text{if } n = 0 \text{ or } 1 \\ T(n-1)+T(n-2)+h(n) & \text{otherwise} \end{cases}$$

Using a Barometer

- A ***barometer*** instruction is one that is executed at least as often as any other instruction in the algorithm
- We can then count the number of times that the barometer instruction get executed
  - Provided that the time taken by each instruction is bounded by a constant, the time taken by the entire algorithm is in the exact order of the number of times the barometer instruction is executed

Average Case Analysis

- We need to know instance distribution
  - Given the instance distribution, know how to calculate the average cost

$$\text{average cost} = \sum_{i=1}^{m} p_i c_i$$

- Sometimes we make ideal assumption that all instances of any given sizes are equally distributed