Midterm Review

Topics For Midterm

1. Analyzing Algorithms (Chapter 2)
2. Growth of Functions (Chapter 3)
3. Recurrence (Chapter 4)
4. Heapsort (Chapter 6)

Study guide

- Study the homework and quiz questions
- Go through the lecture notes or at least the review slides

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

Machine Model and Elementary (Primitive) Operation

- Assuming RAM (random-access machine) model
  - Instructions and costs are well-defined
  - Realistic
  - No concurrent operations

- An elementary (primitive) 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.

Asymptotic Notation

- What does “the order of” mean
- Big O, Ω, Θ, o, ω notations
- Properties of asymptotic notation
- Limit rule

Asymptotic notations

- Know the definitions of big O, Ω, Θ, o and ω notations
  - Example: what does $O(n^2)$ mean?
- Know how to prove whether a function is in big O, Ω, and Θ based on definition
  - Example
    - Prove that if $f(n) = O(g(n))$ then $g(n) = \Omega(f(n))$
    - Prove $3n+5 = \Theta(n)$ using the definition of Θ

Definition of big O

$$O(g(n)) = \{f(n) \mid (\exists c \in R^+, n_0 \in N)(\forall n \geq n_0)[0 \leq f(n) \leq cg(n)]\}$$

- Typically used for *asymptotic upper bound*
- Remember the order of growth below

$$O(\lg n) \subset O(n^c) \subset O(n^c \lg n) \subset O(n^{c+\varepsilon} \lg n) \subset O(d^n) \qquad c, \varepsilon > 0, d > 1$$

Definition of Ω

$$\Omega(g(n)) = \{f(n) \mid (\exists c \in R^+, n_0 \in N)(\forall n \geq n_0)[f(n) \geq cg(n) \geq 0]\}$$

- Ω is typically used to describe *asymptotic lower bound*
  - For example, insertion sort take time in Ω(n)
- Ω for algorithm complexity
  - We use it to give the lower bounds on the intrinsic difficulty of solving problems
  - Example, any comparison-based sorting algorithm takes time Ω(*nlogn*)

The Θ notation

Definition:

$$\Theta(g(n)) = \{f(n) \mid (\exists c_1, c_2 \in R^+, n_0 \in N)(\forall n \geq n_0)[0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)]\}$$

Equivalent to: $\Theta(f(n)) = O(f(n)) \cap \Omega(f(n))$

- Used to describe *asymptotically tight bound*
- Example: selection sort take time in $\Theta(n^2)$

Definition of o and ω

- Definition

$$o(g(n)) = \{f(n) \mid (\forall c \in R^+, \exists n_0 \in N, \forall n \geq n_0)[0 \leq f(n) < cg(n)]\}$$

$$\omega(g(n)) = \{f(n) \mid (\forall c \in R^+, \exists n_0 \in N, \forall n \geq n_0)[f(n) > cg(n) \geq 0]\}$$

- Denote upper/lower bounds that are not asymptotically tight
- Example $1000n \in o(n^2); \quad 1000n^2 \notin o(n^2)$

$$1000n^2 \in \omega(n); \quad 1000n^2 \notin \omega(n^2)$$

- Properties

$$f(n) \in o(g(n)) \Leftrightarrow \lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$$

$$f(n) \in \omega(g(n)) \Leftrightarrow \lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty$$

Maximum and Limit rules

- Know to prove asymptotic relationship using the rules
  - Example
    - Show that $O((n+1)^2) = O(n^2)$
    - Show that $\lg^2 n \in O(n^{0.5})$

The Maximum rule

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

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

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

- The maximum rule let us ignore lower-order terms

The Limit Rule

- Let $f, g : N \rightarrow R^{\geq 0}$, 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 $f(n) \notin \Theta(g(n))$

Relational Properties

- Transtivity: *O, o, Ω, ω,Θ*
- Reflexity: *O, Ω, Θ*
- Symmetry: $f(n) = \Theta(g(n)) \Leftrightarrow g(n) \in \Theta(f(n))$
- Transpose symmetry (Duality)

$$f(n) = O(g(n)) \Leftrightarrow g(n) \in \Omega(f(n))$$

$$f(n) = o(g(n)) \Leftrightarrow g(n) \in \omega(f(n))$$

- Analogy

$$f(n) \in O(g(n)) \approx a \leq b$$

$$f(n) \in \Omega(g(n)) \approx a \geq b$$

$$f(n) \in \Theta(g(n)) \approx a = b$$

$$f(n) \in o(g(n)) \approx a < b$$

$$f(n) \in \omega(g(n)) \approx a > b$$

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 $t(n) = \Omega(f(n))$, 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 $\Omega(n\log n)$?

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

Analysis of Algorithms

Analyzing control structures

- Sequencing
- For loops
- While and repeat loops
- Recursive calls

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 $t=t1+t2 = \Theta(\max(t1,t2))$.

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;
}
```

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

Solving Recurrence

- Know how to solve a recurrence using recursion tree and verify the solution using the substitution method
- Know how to use the simplified version of the Master theorem

Heaps

- Know the definition
  - What is the heap property?
- Given a node, know how to calculate its parent and children
- Know how each heap method work
  - Can write and analyze these algorithms
  - Given an example heap, demonstrate how these algorithms work
  - Design a new similar heap related algorithm

Some important properties of heaps

- Given a node *T[i]*
  - It's parent is *T[i/2]*, if *i>1*.
  - It's left child is *T[2*i]*, if *2*i<=n*.
  - It's right child is *T[2*i+1]*, if *2*i+1<=n*.

- The height of a heap containing n nodes is $\lfloor \lg n \rfloor$

Methods of class MaxHeap

- heapify(int i);
- increaseKey(int i, int key);
- maximum();
- extractMax();
- insert(int key);
- buildHeap();
- heapSort();