Midterm Review

Topics For Midterm

Topics	Reading
Introduction	1
Induction and loop invariants	2.1
Asymptotic Notation	3.1-3.2
Algorithm Analysis - Analyzing control structures - Worst-case and Average-case - Amortized analysis	2.2 5.1-5.3 17.1-17.3
Solving Recurrences	4.1-4.3
Heap and Heap Sort	6
Binomial Heaps	19

Study guide

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

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 $\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$

Loop Invariants

- To prove some statement S about a loop is correct. Define S in terms of a series of smaller statements, S_0, S_1, …, S_k, where
  - The initial claim, S_0, is true before loop begins
    - Initialization (compared to induction basis)
  - If S_{i-1} is true before iteration i begins, then S_i will be true after iteration i is over
    - Maintenance (compared to induction step)
  - The final statement implies S
    - Termination (conclusion. This step is different from a typical induction proof)
- Mastering:
  - Given a loop invariant, know to prove its properties: initilization, maintenance, and termination

Loop Invariant: Example

- Prove the following loop find the max(a[0], …, a[n-1]) using the loop invariant
  - Si: max = max(a[0..i]).

```
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;
}
```

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 $f(n) \in O(g(n))$ then $g(n) \in \Omega(f(n))$
    - Prove $3n+5 \in \Theta(n)$ 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 Θ(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 $f, g : N \to R^{\geq 0}$, 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 $f(n) \notin \Theta(g(n))$

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

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

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

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

Amortized Analysis

```
for (i=0; i<n; i++) P;
or
…P1…P2….Pi………Pn…
```

- Operation P is called n times.
- Each call to P is not independent: its execution time depends on the previous calls.
- The "average" cost to P considers the average over successive calls.
  - Compared to the "average-case" analysis which considers the average over all instances based on their distribution

Three analyzing methods

- Required
  - Know how to apply Aggregate analysis
  - Given the amortized cost, know how to argue it using the accounting method
  - Given the potential function, know how to derive the amortized cost.

An aggregate analysis: binary counter

- For n consecutive operations
  - A[0] flips each time incrementCounter() is called
  - A[1] flips $\left\lfloor \frac{n}{2} \right\rfloor$ times
  - …
  - A[i] flips $\left\lfloor \frac{n}{2^i} \right\rfloor$ times
  - …
- Total flips is

$$\sum_{i=0}^{k-1} \left\lfloor \frac{n}{2^i} \right\rfloor \le \sum_{i=0}^{k-1} \frac{n}{2^i} = n\sum_{i=0}^{k-1} \frac{1}{2^i} < n\sum_{i=0}^{\infty} \frac{1}{2^i} = 2n \in O(n)$$

- Average flips per operation ≈ 2

Accounting for binary counter

- Assume amortized cost: 2
  - Allocate 2 dollars for each call
  - Associate the 2 dollars with the bit set
- Actual cost:
  - Spend one dollar when a bit is flipped (set or reset)
- Analysis
  - Each bit "1" gets 1 dollar credit associated with it
  - Pay the flipping cost of each bit using the credit
  - Balance = the number of 1's which is never negative

Potential functions

- A potential function describes the state of "cleanliness" before a process/operation executes.
  - A large value of the state means "dirtier": it denotes the amortized cost of the following processes
  - Let $\Phi(D_0)$ be the value of the initial state and $\Phi(D_i)$ be that of the state after the i^th call, the amortized time taken by the i-th call is $\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1})$
  - Let T_n denote the total time required for the n calls, and $\hat{T}_n$ be the total amortized time, we have

$$\hat{T}_n = T_n + \Phi(D_n) - \Phi(D_0)$$

  - $\hat{T}_n$ can be an upper bound for T_n when $\Phi(D_n) \ge \Phi(D_0)$

Potential method for Binary Counter

- We use the number of ones as potential function
- Then the amortized cost of adding one to the counter is
  - The counter value is even. The least significant bit (A[0]) is set and it adds one more 1. The amortized cost is 1+1=2.
  - All bits of the counter is 1. The loop executes k times and all k 1s change to 0s. The amortized cost is k+(0-k) = 0.
  - In other cases, assume the loop executes i times. It flips each of the rightmost i bits from 1 to 0, and set (i+1)-th bit from 0 to 1. The 1s decreases by i-1. The amortized cost if (i+1)-(i-1) = 2.

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

Recurrences

- The substitution method
- The recursion tree method
- The master method

The substitution method

- Guessing the form of the solution
- Using the mathematical induction to show that the solution works

The substitution method: an example

We'd like to solve $T(n) = 3T(\lfloor n/4 \rfloor) + n$.
We guess $T(n) \in O(n)$.
We prove by induction that there exists a constant c such that $T(n) \leq cn$ for sufficiently large n.

$$\begin{aligned} T(n) &= n + 3T(\lfloor n/4 \rfloor) \\ &\leq n + 3 * c * \lfloor n/4 \rfloor \\ &\leq (1 + 3c/4)n \\ &\leq cn, \ when\ c \geq 4 \end{aligned}$$

The recursion-tree method

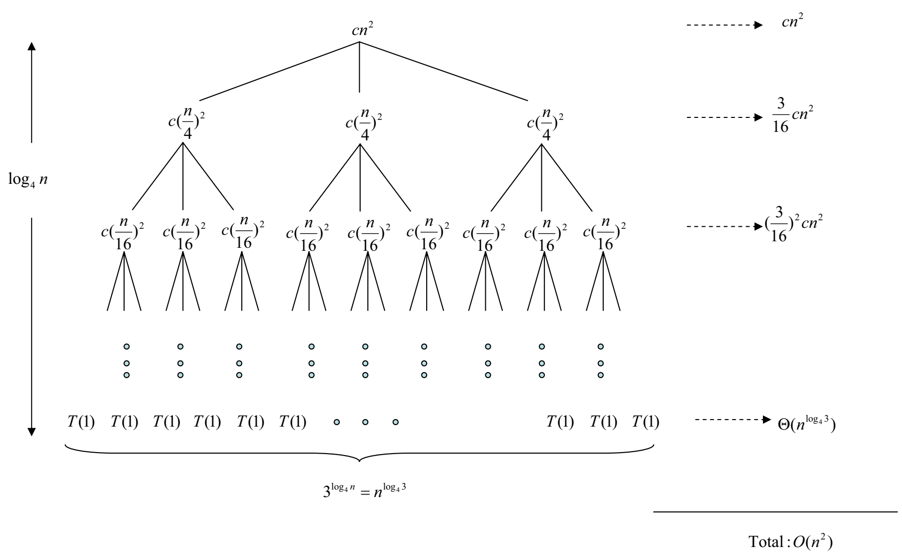
- The method
  - Draw a recursion tree where each node represents the cost of a single subproblem
  - Sum the cost of each level to get per-level cost
  - Sum all per-level costs to get the total cost
- Applications
  - Can be used to find a good guess. Complete by using the substitution method. Can be a bit sloppy when constructing the tree.
  - Can serve as a direct proof. Need to be strict when draw the tree.

The recursion-tree method: an example

We'd like to solve $T(n) = 3T(\lfloor n/4 \rfloor) + \Theta(n^2)$.
We instead draw a tree for $T(n) = 3T(n/4) + cn^2$.
Some sloppiness we use here

- assume n is an exact power of 4 to remove the floor function
- replace $\Theta(n^2)$ by cn^2

Constructing the recursion tree



The sum of per-level costs results below:

$$
\begin{aligned}
T(n) &= cn^2 + \frac{3}{16}cn^2 + (\frac{3}{16})^2 cn^2 + ... + (\frac{3}{16})^{log_4 n - 1} cn^2 + 3^{log_4 n}\Theta(1) \\
&= \sum_{i=0}^{log_4 n - 1} (\frac{3}{16})^i cn^2 + \Theta(n^{log_4 3}) \\
&< \sum_{i=0}^{\infty} (\frac{3}{16})^i cn^2 + \Theta(n^{log_4 3}) \\
&= \frac{1}{1 - 3/16} cn^2 + \Theta(n^{log_4 3}) \\
&= O(n^2)
\end{aligned}
$$

Asymptotic recurrences

Consider a function $T : N \rightarrow R^+$ such that

$$T(n) = lT(n/b) + f(n)$$

for all sufficiently large n, where $l \geq 1$ and $b \geq 2$ are constants, and $f(n) \in \Theta(n^k)$ for some $k \geq 0$. We conclude that

$$T(n) \in \begin{cases} \Theta(n^k) & if \quad k > log_b l \\ \Theta(n^k \log n) & if \quad k = log_b l \\ \Theta(n^{\log_b l}) & if \quad k < log_b l \end{cases}$$

Examples

$$T(n) = T(n/3) + 1.$$

$$T(n) = T(n/3) + n.$$

$$T(n) = 9T(n/3) + n.$$

$$T(n) = 3T(n/4) + nlgn.$$

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

Methods of class MaxHeap

```
Class MaxHeap {
  int A[];
  int n;

  public void heapify(int i);
  public void increaseKey(int i, int key);
  public int maximum();
  public int extractMax();
  public void insert(int key);
  public void buildHeap();
  public void heapSort();
}
```

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 $\lfloor \lg n \rfloor$

Binomial Heaps

- Know the definition of Binomial Trees and Binomial Heaps
- Understand the following algorithms

  (Can write and analyze these algorithms.
  Given an example binomial heap, demonstrate how these algorithms work.
  Design a new similar binomial heap related algorithm)
  - Unite two equal size binomial trees
  - Unite two binomial heaps
  - minimum()
  - extractMin()
  - insert()
  - decreaseKey()
  - deleteKey()

Unite two equal size binomial trees

```
BinomialTree uniteBinomialTrees(B1, B2){
   // B1, B2 are the same size: B1.degree = B2.degree
   if (B1.root().key < B2.root().key) {
      B.copy(B1);
      B.setDegree(B1.degree()+1);
      B2.root().setParent(B1.root());
      B2.root().setSibling(B1.child());
      B.setChild(B2);
   } else {
      // link in the other way
      …
   }
}
```

It takes a time in O(1).

Unite two binomial heaps

```
binomialHeapsUnion(H1, H2)
{
   while (simultaneously following the links in H1 and H2) {
      if there are three degree i trees{ // one from the carry-on
         merge two of them and set it as carry-on;
         add the remainder to H;
      } else if there are two degree i trees {
         merge the two trees;
         set it as carry on;
      } else if there is one degree i tree{
         add it to H;
      }
   }
   add the carry-on if exists to H.
}
```

Assume the result binomial heap contains n nodes. The construction can be done in $\lfloor \lg n \rfloor + 1$ stages. Time in O(log n)

minimum()

- Return the node pointed by the *min* pointer.
  - Cost O(1)
- Without the *min* pointer
  - Traverse the link to find the min
  - Cost O(lg n)

insert

```
insert(v, H)
{
   make a 1 node binomial tree B0;
   Build a binomial heap H0 that contains B0;
   merge H0 and H;
}
```

extractMin(): remove the minimum node

```
extractMin(H)
{
    take the min binomial tree B out (H/B);
    remove the root of B;
    join the subtrees of B into a new binomial heap H';
    unite H/B and H';
}
```

Cost: O(log n)

insert

```
insert(v, H)
{
   1. make a 1 node binomial tree B_0*;
   2. i = 0;
   3. while (1) {
         if (H include a B_i) {
           remove B_i from H;
           merge B_i* and B_i into a binomial tree B_{i+1}*;
           i++;
         } else
            break;
       }
   4. insert B_i* into the list of roots of H.
}
```

decreaseKey

```
public void decreaseKey(Node x, int key)
{
   Node  cur = x;
   Node  parent = x.parent;

   while (parent !=NULL && cur.key < parent.key){
         swap(cur.key, parent.key);
         cur = parent;
         parent = cur.parent;
   }
}
```

Cost?

deleteKey

```
public void deleteKey(Node x)
{
   decreaseKey(x, -∞);
   extractMin();
}
```

Cost?