COMP 250: Mid-term Exam Review

Lecture 19

Jérôme Waldispühl
School of Computer Science
McGill University
Review - Recursive algorithms

• To write a recursive algorithm:
  – Find how the problem can be broken up in one or more smaller problems of the same nature
  – Remember the base case!

• Usually, better running times are obtained when the size of the subproblems are approximately equal
  – $power(a,n) = a * power(a,n-1) \Rightarrow O(n)$
  – $power(a,n) = (power(a,n/2))^2 \Rightarrow O(log\ n)$

• Fibonacci, BinarySearch, MergeSort...
Merge Sort

Divide

4 3 2 1

4 3

2 1

4

3

2

1

Merge

1 2 3 4
Recursive algorithms & inductions

• You can see an algorithm as a function.
• To prove the correctness of a recursive algorithm, we can use a similar technique to the one used to prove mathematical inductions.
• But we should also prove that our algorithm terminates!
Induction proofs

• To prove that a proposition $P(n)$ holds for all $n \geq a$:
  – **Base case:**
    Prove that $P(a)$ holds
  – **Induction step on $n$:**
    Induction Hypothesis: Assume $P(n)$ holds
    Prove that I.H. implies that $P(n+1)$ holds
Generalized induction proofs

To prove that a proposition $P(n)$ holds for all $n \geq a$:

- **Base case:**
  Prove that $P(a)$, $P(a+1)$... (as many as needed) hold

- **Induction step on $n$:**
  *Induction Hypothesis:* Assume $P(k)$ holds for all $k \leq n$
  Prove that induction hypothesis implies that $P(n+1)$ holds
Induction proofs: Example

Claim: for all positive integers $k, n$, we have $(1 + k)^n \geq 1 + kn$.

Proof:
• Base case: $n=1$, $(1 + k)^1 \geq 1 + k$
• Induction hypothesis: for all $k$ and a given $n$, we have we have $(1 + k)^n \geq 1 + kn$
• Induction step (show that then it is true for $n+1$):
  
  $(1+k)^{n+1} = (1+k)(1+k)^n$
  
  $\geq (1+k)(1+kn)$
  
  $= 1 + kn + k + k^2n$
  
  $= 1 + k(n+1) + (1+k)$
  
  $\geq 1 + k(n+1)$
Proving algorithms

AF(int x) {
    f = 2;
    while ( x > 1 ) {
        if ( x % f == 0 ) {
            System.out.println(f);
            x = x / f;
        } else {
            f++;
        }
    }
}

FM(int[] A, int l, int r) {
    if (l<r) {
        m=[(l+r)/2]
        return max(
            FM(A,l,m),
            FM(A,m+1,r));
    } else { 
        return A[l];
    }
}
Proving algorithms

```java
FM(int[] A, int l, int r) {
    if (l<r) {
        m=⌊(l+r)/2⌋
        return max(FM(A,l,m),FM(A,m+1,r));
    } else { return A[l]; }
}
```

- Pre-condition: $0 \leq l \leq r \leq A.length$
- Post-condition: return max value stored in $A[l:r]$
- Base case: the max of an array of size 1 is the value stored.
- Induction Hypothesis: for all $l$ and $r$ such that $|r-l| \leq n$, $FM(A,l,r)$ returns the max in $A[l:r]$
- Induction step: for all $|r-l|=n+1$, then
  \[ FM(A,l,r) = \max(FM(A,l,m),FM(A,m+1,r)) \]
Proving algorithms

```
FM(int[] A, int l, int r) {
    if (l<r) {
        m=⌊(l+r)/2⌋
        return max(FM(A,l,m),FM(A,m+1,r));
    } else { return A[l]; }
}
```

- Induction Hypothesis: for all l and r such that |r-l|≤n, FM(A,l,r) returns the max in A[l:r]
- Induction step: Assume |r-l|=n+1.
  FM(A,l,r) = max(FM(A,l,m),FM(A,m+1,r))
  (I.H.) FM(A,l,m) & FM(A,m+1,r) return max of A[l,m] & A[m+1,r]
  Since max of A[l,r] is in A[l,m] or A[m+1,r], then FM(A,l,r) returns max of A[l:r]
- Termination: At every recursive call, |r-l| decreases strictly until |r-l|=0, which is the base case.
Loop Invariants

AF(int x) {
    f = 2;
    while ( x > 1 ) {
        if ( x % f == 0 ) {
            System.out.println(f);
            x = x / f;
        } else { f++; }
    }
}

Invariant:

x = the number that remains to be factor and we have removed all factors < f
Loop invariant

• **Initialization**: $x$ contains all factors

• **Maintenance**: At the beginning of the loop we removed all factors $< f$. After one iteration of the loop, if $f$ is a factor of $x$, then we remove it and do not increase $f$. Otherwise, we can safely increment $f$ and $x$ is such that we removed all factors $\leq f$.

• **Termination**: The number that remains to be factored $= 1$ and we have removed all factors including the largest factor (i.e.: we have found all the factors)
Discussion

• Difficulty is to identify a good loop invariant. The proof is usually simpler (initialization, maintenance, termination).

• A loop invariant can be any predicate as long as it helps to prove that the loop helps to solve the problem.

• With recursive algorithm, identifying the property we want to prove is easier because it is the post-condition.
# Invariant vs. Induction

<table>
<thead>
<tr>
<th>Iterative algorithm</th>
<th>Recursive algorithm</th>
</tr>
</thead>
<tbody>
<tr>
<td>Loop invariant</td>
<td>⇔</td>
</tr>
<tr>
<td>Initialization</td>
<td>⇔</td>
</tr>
<tr>
<td>Maintenance</td>
<td>⇔</td>
</tr>
<tr>
<td>Termination</td>
<td>⇔</td>
</tr>
</tbody>
</table>

> ⇔

Induction hypothesis
Base case
Induction step
Termination
Running time

- Primitive operations
  - Running time is constant, indep. of problem size
    - Assigning a value to a variable
    - Calling a method; returning from a method
    - Arithmetic operations, comparisons
    - Indexing into an array
    - Following object reference
    - Conditionals
- Running time ≡ Number of primitive operations
- Loops: Sum the running time of each iteration
- findMin, insertionSort
Recurrences

- For recursive algorithms, we express the running time $T(n)$ for an input of size $n$ as a function of $T(a)$ for some $a < n$.

- Example:
  - Binary search: $T(n) = T(n/2) + a$
  - MergeSort: $T(n) = 2 \cdot T(n/2) + c \cdot n$
Solving recurrences

- Solving recurrence $\equiv$ give explicit formula for $T(n)$
- Substitution method:
  - Replace occurrences of $T()$ by their value
  - Repeat until pattern emerges
- Prove by induction that guess is correct
Example

Solution of $T(n) = 2T(n/2) + cn$, $T(1)=0$ for $n \geq 0$ a power of 2?

$T(n) = 2 \left( 2T(n/4) + cn/2 \right) + cn$

$= 2^2 T(n/4) + 2 cn$

$= 2^2 \left( 2T(n/8) + cn/4 \right) + 2 cn$

$= 2^3 T(n/8) + 3 cn$

$\vdots$

$= 2^k T(n/2^k) + k cn$

We have $n=2^k$, thus $k=\log_2(n)$ and we replace $k$ in $T(n)$.

$T(n) = 2^{\log(n)} T(1) + \log_2(n) \cdot cn = cn \log(n)$

Then prove by induction...
Proof by induction

**Proposition.** If $T(n)$ satisfies the following recurrence, then $T(n) = n \log_2 n$.

$$T(n) = \begin{cases} 
0 & \text{if } n = 1 \\
2 \cdot T(n/2) + n & \text{otherwise}
\end{cases}$$

**Pf 2.** [by induction on $n$]

- **Base case:** when $n = 1$, $T(1) = 0$.
- **Inductive hypothesis:** assume $T(n) = n \log_2 n$.
- **Goal:** show that $T(2n) = 2n \log_2 (2n)$.

$$T(2n) = 2 \cdot T(n) + 2n = 2n \log_2 n + 2n = 2n (\log_2 (2n) - 1) + 2n = 2n \log_2 (2n). \quad \blacksquare$$
Big-Oh notation

g(n) is $O(f(n))$ iff there exist constants $c$ and $n_0$ such that $g(n) \leq c f(n)$ for all $n \geq n_0$

- $f(n)$ is $\Omega(g(n))$ iff $g(n)$ is $O(f(n))$
- $f(n)$ is $\Theta(g(n))$ iff $f(n)$ is $O(g(n))$ and $f(n)$ is $\Omega(g(n))$
Example

Show that $\text{Fib}(n)$ is $O(2^n)$

Proof: We will show by induction that for all $n \geq 0$, $\text{Fib}(n) < 2^n$

• Base case: $n=0$, $\text{Fib}(0)=0<2^0=1$; $n=1$, $\text{Fib}(1)=1<2^1=2$

• Induction hypothesis: for all $k \leq n$ $\text{Fib}(k)<2^k$. $\text{Fib}(n+1)$?

• Induction step:

\[
\text{Fib}(n+1) = \text{Fib}(n) + \text{Fib}(n-1) \\
< 2^n + 2^{n-1} \\
< 2^n + 2^n = 2^{n+1}
\]

Using the previous definition, we set $c=1$ and $n_0=0$, then $\text{Fib}(n)$ is $O(2^n)$. 
Example 2

- Previously, we used the recursive formula $T(n)=2T(n/2)+n$, with $T(1)=0$, to characterize the number of operations of MergeSort.

- We showed that $T(n) = n \log_2(n)$

- Then, MergeSort runs in $O(n \log_2(n))$
Big-Oh notation

Hierarchy of big-Oh classes
Big-Oh notation

Simplification rules

- If \( f_1(n) \in O(g(n)) \) and \( f_2(n) \in O(g(n)) \) then
  \[
  f_1(n) + f_2(n) \in O(g(n))
  \]

- If \( f_1(n) \in O(g(n)) \) then \( k.f_1(n) \in O(g(n)) \)

- If \( f_1(n) \in O(g(n)) \) and \( f_2(n) \in O(h(n)) \) then
  \[
  f_1(n).f_2(n) \in O(g(n).h(n))
  \]

...
Log identities

- $\log(ab) = \log(a) + \log(b)$
- $\log(a^n) = n \log(a)$
- $\log_a(n) = \log_b(n) / \log_b(a)$
- $a^{\log_b(n)} = n^{\log_b(a)}$
Running time of a For loop

```c
for (i=1; i<N; i=i*2) { ... }
```

*What is the running time of this loop?*

Value of i after k iterations: $2^k$

We have $i < N \Rightarrow 2^k < N \Rightarrow k < \log_2(N)$.

There is less than $\log_2(N)$ iterations.

The running time of this loop is $O(\log(n))$.

#operations: $O(\log(n))\ O(1) = O(\log(n))$
Abstract Data Types

• Implementation: Arrays or linked-lists

• Basic operations
  – getFirst(), get(n), getLast()
  – removeFirst(), removeLast(), remove(o)
  – addFirst(o), addLast(o), add(o)
  – empty(), size()

• Advantages and disadvantages over arrays

• Stack, Queues, deques, rotating arrays
Operations on deques with Array

- **Queue()** {
  L = new Array[N];
  head = tail = -1;
}

- **isEmpty()** {
  if ((head == -1) && (tail == -1)) return true;
  else return false;
}

- **isFull()** {
  if ((head - tail % N) == 1) return true;
  else return false;
}
Operations on deques with Array

• Enqueue(o) throw Exception {
  if ( isFull() ) { throw new Exception(“Full stack”) } 
  if ( isEmpty() ) { head = tail = 0; } 
  else { tail = ( ( tail + 1 ) % N ); } 
  L[tail] = o; 
}

• Dequeue() {
  if ( isEmpty() ) { throw new Exception(“Empty stack”) } 
  Object o = L[head]; 
  if (((head – tail) % N) = 1) { head = tail = -1; } 
  else { head = ( ( head + 1 ) % N ); } 
  return o; 
}