Algorithm Analysis

CSE/IT 122 ~ Algorithms and Data Structures

Ex: $n < 2^n$

- Show that $n < 2^n$ for all positive integers
- Proof:
  - Basis Step: $1 < 2^1 = 2$, which is true
  - Inductive Step: Assume true for P(n). That is we assume $n < 2^n$ for some value n.
    - Need to show $(n + 1) < 2^{n+1}$
    - Add 1 to each side of $n < 2^n$
    - $N + 1 < 2^n + 1$ using the assumption for P(n)
    - Now $n + 1 < 2^n + 1 <= 2^n + 2^n = 2^n(1 + 1) = 2^n2 = 2^{n+1}$

Ex: n >= 4, n! > 2^n

- Show for n >= 4, n! > 2^n
- Proof:
  - Basis Step: n = 4, 4! = 24 > 2^4 = 16
  - Inductive Step: Assume for some n, n! > 2^n
    - Show (n + 1)! > 2^{n+1}
    - (n + 1)! = (n + 1) * n! > (n + 1) * 2^n
    - Now for n >= 4, n + 1 > 2 for all n
    - So (n + 1) * 2^n >= 2 * 2^n = 2^{n+1}

For You To Try: n^3 - n is divisible by 3

- Prove $n^3 - n$ is divisible by 3 whenever n is a positive integer

For You To Try: $n^3 - n$ is divisible by 3

- Show $n^3 - n$ is divisible by 3 whenever n is a positive integer
- Proof:
  - Basis Step: $1^3 - 1 = 0$ which is divisible by 3
  - Inductive Step: Assume $n^3 - n$ is divisible by 3
  - Need to show $(n + 1)^3 - (n + 1)$ is divisible by 3
  - $(n + 1)^3 - (n + 1) = n^3 + 3n^2 +3n + 1 - (n + 1)$
  - $= (n^3 - n) + 3(n^2 + n)$
  - First term is divisible by 3, and by inductive step, 2nd term is divisible by 3

Ex: Fibonacci

- ➔ Fibonacci numbers $F_{n+1} = F_n + F_{n-1}$ and $F_0 = 1$, $F_1 = 1$
- ➔ This produces the sequence 1,1,2,3,5,8,13,21,...
- ➔ Fibonacci sequences appear in all kinds of weird places.
  - For example, the length of strings you can form from the alphabet {0,1} with no consecutive 1’s in the string
  - Strings are:
    - The empty string ~ length 0, one of those
    - 1,0 ~ length 1, two of those
    - 00, 01, 10 ~ length 2, 3 of those
    - 000, 001, 010, 100, 101 ~ length 3, five of those
- ➔ To proof, simply take $n \to \infty, \frac{F_{n+1}}{F_n} = \phi$ and show that

$$F_n = \frac{1}{\sqrt{5}}[(\frac{1+\sqrt{5}}{2})^{n+1} - (\frac{1-\sqrt{5}}{2})^{n+1}]$$

Ex: A Perfect Binary Tree

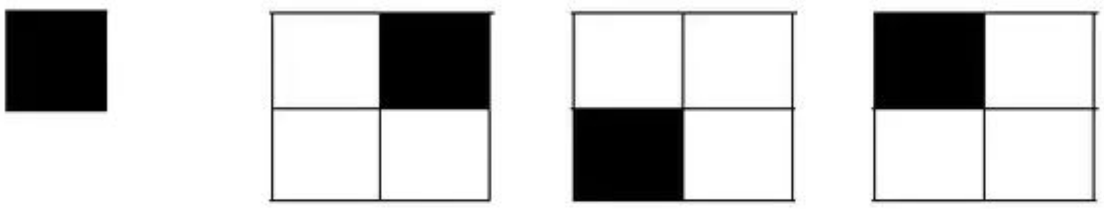
- Claim: A perfect binary tree (every child has two nodes except the leaves and every level is completely filled aka complete binary tree by some authors) with k levels has exactly $2^k - 1$ nodes. Root node is at level 1.
- Proof: Proof by induction on number of levels.

Ex: A Perfect Binary Tree

- Proof: Proof by induction on number of levels.
  - Basis Step: This is true for k = 1, since a complete binary tree with on level consists of a single node and $2^1 - 1 = 1$
  - Inductive Step: Suppose a complete binary tree with *k* levels has 2^k-1 nodes.
    - Show a complete k+1 binary tree with k+1 levels has $2^{k+1}-1$ nodes
    - A complete binary tree with k+1 levels consists of a root plus two trees with k levels
    - So by the induction hypothesis, the total number of nodes is
      - $1 + 2(2^k - 1) = 2^{k+1} - 1$

For You To Try: A Tiling Problem

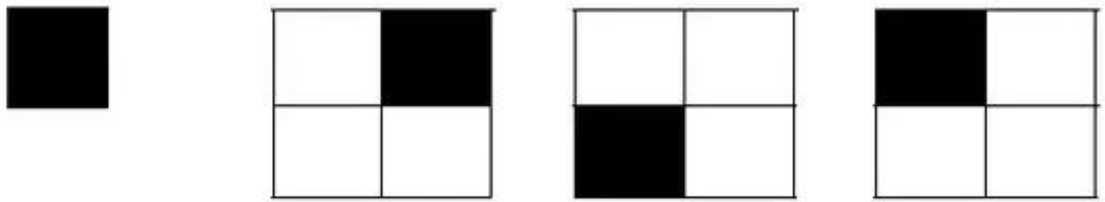
➔ Show that any 2^n x 2^n chessboard, where n is any positive integer, with one square removed can be tiled using L-shaped pieces. The L-shaped piece covers 3 squares at a time.



The black tile represents the removed square

A Tiling Problem: Solution

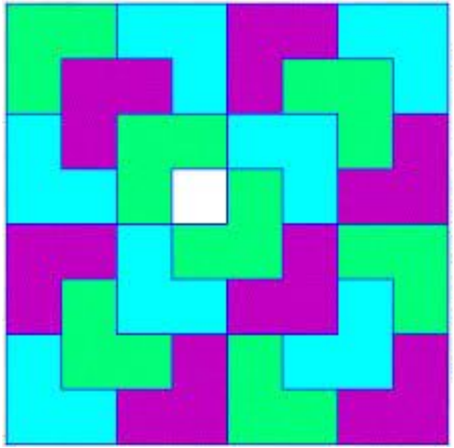
- Basis Step: P(1) is true, since any 2x2 chessboard can be tiled with an L-shaped piece. As seen in the picture below.



A Tiling Problem: Solution

- Inductive Step: Assume P(n) is true, any $2^n x 2^n$ chessboard can be tiled with L-shape pieces with one square removed from the chessboard
  - Have to show that it is true for P(n + 1) case given P(n) is true.
  - Take any $2^{n+1} x 2^{n+1}$ chessboard and remove a piece. A $2^{n+1} x 2^{n+1}$ chessboard contains 4 $2^n x 2^n$ chessboards as $2^{n+1} x 2^{n+1} = 2(2^n) = 4 x 2^n x 2^n$
  - The missing piece must be in one of the $2^n x 2^n$ chessboards. By assumption, this can be tiled.
  - For the other 3 $2^n x 2^n$ boards remove the piece that touches the center of the larger chessboard. All 3 boards can now be tiled and the remaining 3 pieces which forms an L-shape can also be tiled.

A Tiling Problem: Solution



Algorithms and Analysis

- Definition: An algorithm is a definite procedure for solving a problem using a finite number of steps.

Calculating the Sum of 1 to N

➔ First attempt with a for loop

```
long sum_forloop(long n)
{
    long i = 1;
    long sum = 0.0;

    for( ; i <= n; i++)
      sum += i;

    return sum;
```

Calculating the Sum of 1 to N

- ➔ How many times does this function execute?
- ➔ Estimate the running time of the program.
- ➔ We will use T(n) to denote the running time of a program.
  - T(n) does not have units
  - Can think of T(n) as the number of instructions executed on an idealized computer
- ➔ Why no units?
  - Run time of a program depends on:
    - Speed of the machine code is running on
    - Compiler used
    - Quality of code

```
long sum_forloop(long n)
{
    long i = 1;
    long sum = 0.0;

    for( ; i <= n; i++)
      sum += i;

    return sum;
```

RAM Model

➔ RAM ~ Random Access Machine
➔ Machine Independent algorithms
➔ RAM Model is a hypothetical computer where:
  - Simple operations, e.g. +,-,/,*,if, etc. take only **one** time step. We do not need to worry about units of the time step or exactly how long a single time step is. The time step has been normalized to 1.
  - Loops and function calls are not simple operations, but involve many single step operations. In general, the time it takes to run through a loop depends on the number of loop iterations.
  - Each memory access takes one time step. No difference if access is on the disk or cache. You have as much memory as you need.

RAM Model

- ➔ Measure running time by counting up the number of steps an algorithm takes to execute.
- ➔ If you assume the RAM model can do a given number of steps per second, can get an actual running time.
- ➔ RAM model is a simple model of an idealized computer
- ➔ Although simple, it helps describe algorithms in practice

Counting Steps

- Consecutive statements
  - Add running time of each statement
- Loops
  - Loops are not simple operations
  - Running time of the statements inside the loop multiplied by the number of iterations
  - The **for** loop depends on the input n. This is general principle. The running time of a program depends on the inputs to that program.

Ex: One for Loop

- Note: Fencepost errors ((n - 1) - 0 + 1) + 1
  - (n - 1) - 0 + 1 comes from the loop
  - The plus 1 comes from the last test.

	Cost	Number of Times
for(i = 0; i < n; i++)	c_1	$n + 1$
sum += i;	c_2	n

$$T(n) = c_1 \cdot (n + 1) + c_2 \cdot n$$

For Loop Example ~ Single Loop ~ You Try

1A)

```
p = 0
x = 2
for i = 2 to n
    p = (p + i) * x
```

1B)

```
for i = n to 4
    a = 3 * i
    b = 6 * i
```

2)

```
for i = 1 to floor(n/2)
    a = n - i
```

3)

```
for i = 0 to n
    b = b * i;
```

For Loop Example ~ Single Loop ~ You Try

- 1) for loop fencepost in () and + 1 for test:
  - (n-1-3+1)+1 = n-2
  - Inside loop n-3 times (just the fencepost portion)
  - Cost = c_1(n-2) + c_2(n-3)
- 1A)
  - p = 0 ~ 1 time, cost c_1
  - x = 2 ~ 1 time, cost c_2
  - for i = 2 to n ~ (n-2+1)+1, cost c_3
  - p = (p+1)*x ~ (n-2+1), cost c_4
  - Cost = c_1 + c_2 + c_3n + c_4(n-1)

For Loop Example ~ Single Loop ~ You Try

- 1B)
  - for i = n to 4 ~ (n-4+1)+1=n-2, cost c_1
  - a=3*i ~ (n-3), cost c_2
  - b=6*i ~ (n-3), cost c_3
  - Cost: $(n-2)*c_2 + (n-3)*(c_2+c_3)$

For Loop Example ~ Single Loop ~ You Try

2) Fencepost $\lfloor \frac{n}{2} \rfloor - 1 + 1$

plus 1 for the test

get for executes $c_1 \cdot \lfloor \frac{n}{2} \rfloor + 1$

and body of loop executes $\lfloor \frac{n}{2} \rfloor$ times

Add together

3) fencepost $n - 0 + 1 = n + 1$

plus 1 for the test, so executes $n + 2$ times

body of loop

Hwk 0 and Hwk 1 Available Now