LE/EECS 3101 – Course Notes

Sample Version

Completed Edition:

Chapter 1 – Analyzing Algorithm Correctness

Course Material

Motivations for Analyzing Algorithm Correctness

- We want to be able to show that our algorithm is 100% correct with **formal reasoning**.

Defining "Correct"

- A program is correct if and only if the following implication is true:
    - $\boldsymbol{precondition \rightarrow postcondition}$
- To prove the implication, we must:
    1) Assume the precondition
    2) Argue that the postcondition is true after the execution of the code.

Correctness Proofs

- The proof structure via, "2)" depends on the type of program.
    - **Simple Sequential Code**
        - Trace the program line-by-line.
    - **Code with If-Statements**
        - Check all possible paths of execution and **trace line-by-line for each path**.
    - Code with Loops
        - Cannot be traced line-by-line.
        - Must make use of more resources, particularly, a **loop invariant.**

Loop Invariant

- A loop invariant is a **predicate** with variables being parameters.
    - They give **a relationship between variables**.

Loop Invariant Requirements in Loops

1) **Base Case**
    a. The invariant must hold **prior** to the first iteration.
2) **Inductive Step**
    a. Assuming that the **invariant and loop guard are both true**, the invariant must remain true after one arbitrary loop iteration.
3) **Conclusion**
    a. The loop invariant is true at the ends of all iterations of the loop, by principle of induction.

Developing a Complete "Correctness Proof" with Loop Invariants

1) Partial Correctness

   a. Prove some carefully-chosen loop invariant.

   b. Use the **loop invariant** and the **negation of the loop guard** to show that the **postcondition** (after the loop) is **true**.

2) Termination

   a. Show that the loop **terminates** (i.e., Not an infinite loop)

How to Pick a Good Loop Invariant?

- **Recall:** Loop invariants are a **formalization of your intuition.** Use your intuition about what the code does.

1) Loop invariants are **incremental**. They assert something that's true, every time the loop runs, at every iteration.

   a. In a list, they usually say something about the, ***"part of the list that's been processed so far."***

2) In proofs, use only the:

   a. Invariant

   b. Loop Condition Being True (Only if you're inside the loop…)

   c. Pre-Condition

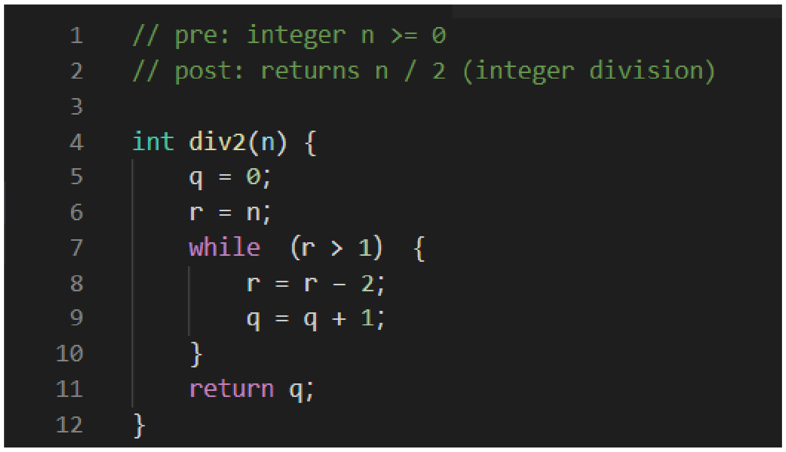
   d. Math

   Anything not inside [a, d] should be written into the invariant itself.

3) Look for any patterns. Visual aids are often very useful in finding a good loop invariant. Also, you may want to run the code and keep track of the iteration specific variable changes.

Recitation Problems

1. Given the following function, $div2(int\ n)$, provide a proof of correctness given the pre and post condition.



```
// pre: integer n >= 0
// post: returns n / 2 (integer division)

int div2(n) {
    q = 0;
    r = n;
    while  (r > 1)  {
        r = r - 2;
        q = q + 1;
    }
    return q;
}
```

The following correctness proof will have the, “Code with Loops” structure.

Step 1 ➔ First, we obtain an intuition on what the code supposedly does.

$\underline{\boldsymbol{Iteration}}$	$\underline{\boldsymbol{q}}$	$\underline{\boldsymbol{r}}$
0	0	n
1	1	$n-2$
2	2	$n-4$
3	3	$n-6$
4	4	$n-8$

When the input, n is even, $r = 0$ upon termination.

Conversely, $r = 1$ when n is odd.

Thus, with the intuition on the relationship among variables, we assert the following loop invariant:

- $Inv(q,r) = \begin{cases} (1): q = \frac{n-r}{2} \\ (2): \quad r \geq 0 \end{cases}$

How the First Invariant was Formed

- $r = n - 2q$
    - $2q = n - r$
        - $q = \frac{n-r}{2}$

Base Case

$q_0 = 0$

$r_0 = n$

1) $\boldsymbol{q_0 = \frac{n-r_0}{2}}$

   $$0 = \frac{n-n}{2}$$
   $$0 = \frac{0}{2}$$
   $$0 = 0$$

   Thus, invariant #1 holds in the base case.

2) $\boldsymbol{r_0 \geq 0}$
   $n \geq 0$ is true by the precondition.
   Thus, invariant #2 holds in the base case.

Inductive Step

Before Iteration: q_0, r_0

Inductive Hypothesis: $q_0 = \frac{n-r_0}{2}\ and\ r_0 \geq 0$

After Iteration: q_1, r_1

Proving The First Loop Invariant

$q_1 = q_0 + 1\ [Line\ \#9]$

$$q_1 = \frac{n-r_0}{2} + 1\ [I.H]$$

$$q_1 = \frac{n-r_0+2}{2}$$

$$q_1 = \frac{n-(r_0-2)}{2}$$

$$q_1 = \frac{n-r_1}{2}\ [Line\ \#8]$$

Proving the Second Loop Invariant

$r_1 = r_0 - 2$ $[Line\ \#8]$

We know that $r_1 \geq 0$ will hold, because r_0 is strictly greater than 1 by the loop guard (line #7, $r > 1$).

Proving the Postcondition (After the Loop Terminates)

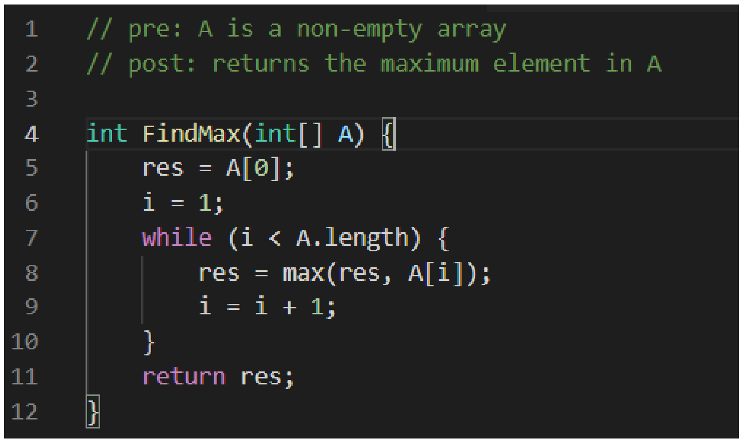
The negation of the loop guard (that is, on line #7) is $r \leq 1$.

For $r = 1$ and $q = \frac{n-1}{2}$, we see that q returns the integer division, $\frac{n}{2}$. Thus, the two invariants hold at the end of the post-condition.

Proving Termination

r will decrease by 2 every iteration, therefore it will eventually recede to or past 1. This will make the loop guard, $r > 1$ be untrue, which will thus terminate the function $div2(n)$.

2. Given the following function, $FindMax(int[]\ \ A)$, provide a proof of correctness given the pre and post condition.



```
1   // pre: A is a non-empty array
2   // post: returns the maximum element in A
3
4   int FindMax(int[] A) {
5       res = A[0];
6       i = 1;
7       while (i < A.length) {
8           res = max(res, A[i]);
9           i = i + 1;
10      }
11      return res;
12  }
```

The following correctness proof will have the, "Code with Loops" structure.

Based on our intuition of what the code does, we assert the loop invariant:

- $$Inv(q,r) = \begin{cases} (1): & i \leq A.length \\ (2): & res = \max(A[0..i-1]) \end{cases}$$

Note: We subtract 1 in invariant #2's end index, because at the end of each iteration, the value of i is incremented by 1 (line #9).

Base Case

$res_0 = A[0]$

$i_0 = 1$

1) $\boldsymbol{i_0 \leq A.length}$

   $0 \leq A.length$ is true by the precondition that the input list is non-empty.

2) $res_0 = \max(A[0..i_0 - 1])$
   $res_0 = \max\ (A[0..1 - 1])$
   $= \max\ (A[0..0])$
   $= \max\ (A[0])$

Inductive Step

Before Iteration: i_0, res_0

Inductive Hypothesis: $i_0 \leq A.length\ and\ res_0 = \max\ (A[0..i_0 - 1])$

After Iteration: i_1, res_1

Proving The First Loop Invariant

$i_1 = i_0 + 1\ [Line\ \#9]$

$i_1 \leq A.length$ will hold, because i_0 is strictly less than $A.length$ by the loop guard, $i < A.length$.

Proving the Second Loop Invariant

$res_1 = \max(res_0, A[i_0])\ \ [Line\ \#8]$

$res_1 = \max(A[0..i_0 - 1], A[i_0])\ [I.H]$

$res_1 = \max\ (A[0..i_0])$

$res_1 = \max(A[0..i_1 - 1])\ \ [Line\ \#9]$

Proving the Postcondition (After the Loop Terminates)

The negation of the loop guard (line #7) is $i \geq A.length$. For $i = A.length$ and

$res = \max(A[0..i-1])$, we see that res will be the largest value in A.

Proving Termination

"i" will increase by 1 every iteration, therefore it will eventually exceed $A.length$. This will make the loop guard false, showing that the program must terminate.

How to Pick a Good Loop Invariant?

- **Recall:** Loop invariants are a **formalization of your intuition.** Use your intuition about what the code does.

1) Loop invariants are **incremental**. They assert something that's true, every time the loop runs, at every iteration.
    a. In a list, they usually say something about the, "part of the list that's been processed so far."
2) In proofs, use only the:
    a. Invariant
    b. Loop Condition Being True
       (Only if you're inside the loop…)
    c. Pre-Condition
    d. Math

    Anything not inside [a, d] should be written into the invariant itself.

3) Look for any patterns. Visual aids are often very useful in finding a good loop invariant. Also, you may want to run the code and keep track of the iteration specific variable changes.

Chapter 2 – Analyzing Algorithm Complexity

Chapter 2 – Prerequisites Summary / Cheat Sheet

Logarithm Laws

1. $\log_a(m \times n) = \log_a(m) + \log_a(n)$
2. $\log_a\left(\frac{m}{n}\right) = \log_a(m) - \log_a(n)$
3. $\log_a(m^n) = nlog_a(m)$
4. $\log_a 1 = 0$
5. $\log_a a = 1$
6. $a^{\log_a(m)} = m$
7. $\log_a(x) = \frac{\log_b(x)}{\log_b(a)}$

Expectation

- The running time is, **"distributed"** between the best and the worst.
- Represents the **expectation of the running time** which is **distributed between the best and worst-case scenario.**

- Let t_n be a **random variable** whose possible values are between $[best, worst]$. Then, we can formulate the known equation:

$$\boldsymbol{E[t_n] = \sum_{t=best}^{worst} \mathrm{t} \times \mathrm{Pr}(t_n = t)}$$

Note: We need to be given the **probability distribution** of the inputs in order to solve for the $\Pr\,(t_n = t)$

Arithmetic Series

- $\sum_{k=0}^{\infty}(a + kd) = \frac{n}{2}(2a + (n-1)d)$

Geometric Series

- $a + ar + ar^2 + ar^3 + \cdots + ar^{n-1} + \cdots = \sum_{n=1}^{\infty} ar^{n-1}, a \neq 0$
- $\sum_{n=1}^{\infty} ar^{n-1} = \frac{a(1-r^n)}{(1-r)}$

Lower Bound on Series (Ratio Test)

- Given $\sum_{i=1}^{\log_2(n)} f(x) \leq \sum_{i=1}^{\infty} f(x)$, then: $\lim_{n\to\infty}\left(\frac{a_{n+1}}{a_n}\right) = L < 1$

Course Material

Introduction

- We will define **complexity** as the quantity of resources required by an algorithm, measured as a function of the input size.
- There are two types of complexities we can reasonably measure:
    - **Time Complexity**
        - Number of steps ("running time") executed by an algorithm
    - **Space Complexity**
        - Number of units of space required by an algorithm.
            - Number of elements in a list
            - Number of nodes in a tree / graph

Running Time

- Could be measured by:
    - Counting the number of times **all lines** are executed.
    - Counting the number of times **some lines** are executed.
- In the running time, we don't really care about the number of steps. We care about **how the number of steps grows as the size of the input grows:**
    - Constant factors do not matter when it comes to growth
    - Only the highest-order terms matter

Review: The Growth Rates of Functions

1. n^n
2. 2^n
3. n^3
4. n^2
5. $nlog_2(n)$
6. n
7. $\sqrt{n}$
8. $\log_2(n)$
9. 1

Worst-Case Running Time

- The case with the **longest** running time.
    - $T(n) = \max\{t_{(x)}: x\ is\ an\ input\ of\ size\ n\}$

Best-Case Running Time

- The case with the **shortest** running time.
    - $T(n) = \min\{t_{(x)}: x\ is\ an\ input\ of\ size\ n\}$

Average-Case Running Time

- The running time is, **"distributed"** between the best and the worst.

- Represents the **expectation of the running time** which is **distributed between the best and worst-case scenario.**

- Let t_n be a **random variable** whose possible values are between $[best, worst]$. Then, we can formulate the known equation:

  $$\boldsymbol{E[t_n] = \sum_{t=best}^{worst} \mathrm{t} \times \mathrm{Pr}(t_n = t)}$$

  Note: We need to be given the **probability distribution** of the inputs in order to solve for the Pr $(t_n = t)$

Asymptotic Notations

1. $O(f(n))$ is the asymptotic **upper-bound**.
   a. The set of functions that grows no faster than $f(n)$
2. $\Omega(f(n))$ is the asymptotic **lower-bound**.
   a. The set of functions that grows no slower than $f(n)$
3. $\Theta(f(n))$ is the asymptotic **tight-bound**
   a. This set of functions grows no faster and no slower than $f(n)$.
   b. Both $O(f(n))$ and $\Omega(f(n))$

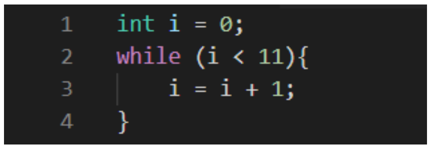
- Do note that both O and Ω can be used to:
  - Upper / Lower Bound the worst-case
  - Upper / Lower Bound the best-case
  - Upper / Lower bound the average-case

- **Using Big-O**
  - **Worst-Case**
    - $\forall x \in Input(n) \leq cf(n), c > 0$
  - **Best-Case**
    - $\exists x \in Input(n) \leq cf(n), c > 0$
- **Using Big-Omega**
  - **Worst-Case**
    - $\exists x \in Input(n) \geq cf(n), c > 0$
  - **Best-Case**
    - $\forall x \in Input(n) \geq cf(n), c > 0$

Recitation Problems

1. Determine the time-complexity of the following code fragments. Justify your answer. [EASY]

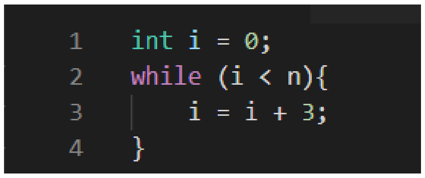
Problem #1



```
int i = 0;
while (i < 11){
    i = i + 1;
}
```

Notice that the loop itself doesn't depend on an input size. Thus, we can see that it runs in a constant amount of time.

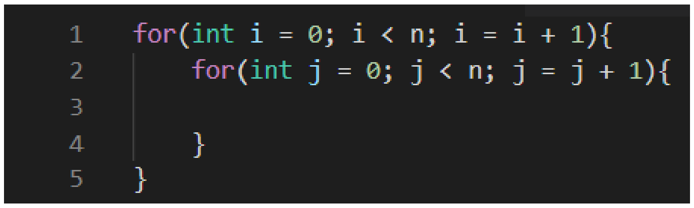
Problem #2



```
int i = 0;
while (i < n){
    i = i + 3;
}
```

We can see that $f(n) = \frac{n}{3}$. Thus, $O(f(n)) = O(n)$ which is our time complexity.

Problem #3

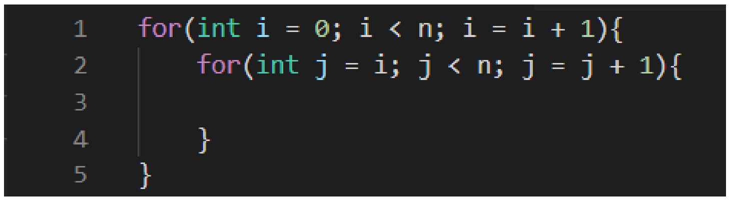


```
for(int i = 0; i < n; i = i + 1){
    for(int j = 0; j < n; j = j + 1){

    }
}
```

The time complexity of the inner and outer loop isolated by themselves is $O(n)$. Thus, together, the time complexity of the whole algorithm is equal to the outer loop multiplied by the outer loop, thus being equal to $O(n^2)$.

Problem #4



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

    }
}
```

For a moment, let's just focus on the inner-loop. Since variable, "i" goes from $[0, n)$, the amount of looping is directly determined by what, "i" is. As a result, we can see that:

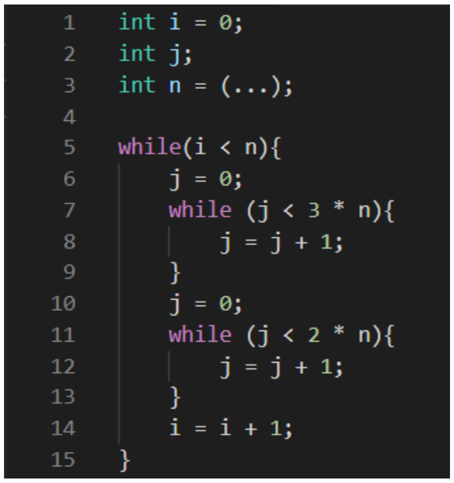
Value of *i*	**Work Done**
1	$(n - 1)$
2	$(n - 2)$

(…)

Very quickly, we can see the pattern that $\sum_{i=1}^{m} n = n + (n-1) + (n-2) + \cdots + 3 + 2 + 1$. Thus, the inner loop, which is dependent on the outer loop, has a time complexity of $O\left(\frac{n(n+1)}{2}\right) = O\left(\frac{n^2}{2} + \frac{n}{2}\right) = O(n^2)$.

Thus, the time complexity of the whole algorithm can easily be seen to equal $O(n^2)$.

Problem #5



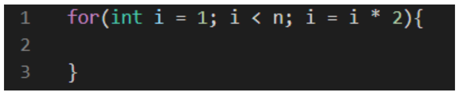
```
int i = 0;
int j;
int n = (...);

while(i < n){
    j = 0;
    while (j < 3 * n){
        j = j + 1;
    }
    j = 0;
    while (j < 2 * n){
        j = j + 1;
    }
    i = i + 1;
}
```

The general approach to such types of problems is to multiply loops on different levels, while adding those that are on the similar level.

Therefore, the time complexity of the following algorithm can be seen to be $O\big(n \times (3n + 2n)\big)$ which simplifies to $O(n^2)$.

Problem #6



```
for(int i = 1; i < n; i = i * 2){

}
```

Problems such as these may not seem immediately clear as to how to determine the time complexity. For these, the best recommended solution is to trace the algorithm and look for a pattern.

Notice here that the value of i changes accordingly:

1. $i = 1$
2. $i = 1 \times 2$
3. $i = 1 \times 2 \times 2$
4. $i = 1 \times 2 \times 2 \times 2$
   $(\ldots)$

   2^k

The exit condition of the following loop occurs when $i \geq n$.

The value of $i = 2^k$. Thus, we can see that $2^k \geq n$ is also the equivalent exit condition.

Notice here that:

$2^k \geq n$

$\log_2(2^k) \geq \log_2(n)$

$k \geq \log_2(n)$

Therefore, the time complexity of the following algorithm can be seen to be $O(\log_2(n))$.

2) Determine the time-complexity of the following code fragments. Justify your answer. [TUTORIALS]

Problem #1: Determine the worst-case running time of the $BigOh(int\ n)$ function.

```
int BigOh(int n) {

    int res = 0;
    i = 0;
    while (i < n * n) {
        if (i % 3 == 0) {
            j = 1;
            while (j < n) {
                res = res + j;
                j = j + 18;
            }
            i = i + 31;
        }
        else {
            j = 1;
            while (j < n * n) {
                res = res + j;
                j = j * 7;
            }
            i = i + 7;
        }
    }
    return res;
}
```

Analyzing the outer-most loop, $while(i < n * n)\{\ldots\}$, we see that i is always updated by a factor of 7. Thus, this particular loop will have a worst-case running time of $O\left(\frac{n^2}{7}\right) \approx O(n^2)$.

Analyzing the First Inner Loop

- $while\ (j < n)\{\ldots j = j + 18\}$
- This loop will have a worst-case running time of $O\left(\frac{n}{18}\right) = O(n)$.

Analyzing the Second Inner Loop

- $while(j < n * n \ldots j = j * 7\}$
- We can see that the loop will terminate when:
  - $7^k = n^2$

Side Work

$k = \log_7(n^2)$

$= 2\log_7(n)$

Using the Change of Base, we get:

$\log_7(n) = \dfrac{\log_2(n)}{\log_2(7)}$

$k = \log_2(n)$

Thus, the worst-case running time of the algorithm is equal to:
$O(outer_{loop} \times worst_{inner}) = O(n^2 \times n) = O(n^3)$

<u>Problem #2:</u>

A and B are the values of the two uniform, six-side dice, rolled independently. We would like to measure the runtime using the number of times the print line is executed.

What's the average-case runtime?
On average, how much do you expect to gain / lose in terms of your money?

```
int TwoDice(int A, int B) {

    int money = -3; // pay $3 to play
    if (A == B) {
        for (int i = 0; i < 10; i++) {
            print("cha-ching");
            money = money + 1;
        }
    }
    else {
        print("cha-ching");
        money = money + 1;
    }
    return money;
}
```

In the worst-case scenario, the program will print "cha-ching" 10 times. This refers to the case where $A == B$.

In the best-case scenario, the program will print "cha-ching" 1 time. This refers to the case where $A \neq B$.

We can also come to the conclusion that $t\ Pr(t_n = t) = \begin{cases} 10\ \times \left(\frac{6}{36} = \frac{1}{6}\right) \\ 1\ \times \left(1 - \frac{6}{36} = \frac{30}{36} = \frac{5}{6}\right) \end{cases}$

$$E[t_n] = \sum_{t=best}^{worst} \mathrm{t} \times \Pr(t_n = t)$$

$$= \sum_{t=1}^{10} \mathrm{t} \times \Pr(t_n = t)$$

$$= 10\left(\frac{1}{6}\right) + 1\left(\frac{5}{6}\right)$$

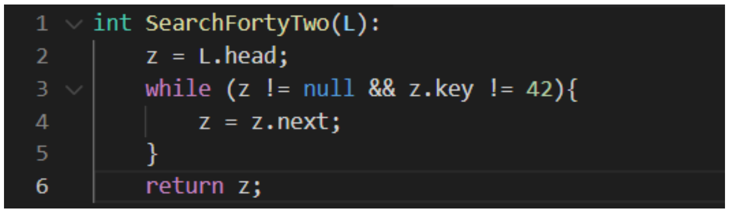
$$= \frac{5}{2}$$

$$= 2.5$$

Therefore, on average, we'll expect to see the print() statement 2.5 times. This is the average-case runtime.

It costs \$3 to play, and we expect to get \$2.50 back in the average case. Thus, on average, you will lose \$0.50, by virtue of $-3 + 2.5$.

Problem #3: Determine the average-case running time of the $SearchFortyTwo(L)$ function.



```
int SearchFortyTwo(L):
    z = L.head;
    while (z != null && z.key != 42){
        z = z.next;
    }
    return z;
```

For each key in the linked list, we pick an integer between 1 and 100 (inclusive), independently, uniformly at random. Compute the average-case running time of this algorithm.

Step 1 – The Worst & Best-Case Scenarios

In the best-case, 42 is found at the head of L. Thus, it incurs a cost of 1.
In the worst-case, 42 is not found in the list, L. Thus, it incurs a cost of $n + 1$.

Therefore,

$$E[t_n] = \sum_{t=1}^{n+1} \mathrm{t} \times \Pr(t_n = t)$$

Step 2 – Solving for the $\boldsymbol{Pr(t_n = t)}$

Recall: For each key in the linked list, we pick an integer between $[1,100]$, uniformly at random.

If we let the random variable t represent the probability of the position t being equal to 42, then we can infer the following information that:

$\Pr(t_n = 1) = 0.01$
$\Pr(t_n = 2) = 0.99 \times 0.01$
$\Pr(t_n = 3) = (0.99)^2 \times 0.01$
$\dots$
$\Pr(t_n = n) = (0.99)^{n-1} \times 0.01$

Also, if $42 \notin L$, then we have $\Pr(t_n = n + 1) = (0.99)^n$

Thus, we can sufficiently summarize our findings of the random variable t:

$$\Pr(t_n = t) = \begin{cases} (0.99)^{t-1} \times 0.01, & 1 \le t \le n \\ (0.99)^n, & t = n + 1 \end{cases}$$

Step 3 – Computing the Average-Case Running Time [Summation]

$$E[t_n] = \sum_{t=1}^{n+1} t \times \Pr(t_n = t)$$

$$= \sum_{t=1}^{n} t \times (0.99)^{t-1} \times (0.01) \quad + \quad (n+1)(0.99)^n$$

$$= (n+1)(0.99)^n + \mathbf{0.01} \sum_{\boldsymbol{t=1}}^{\boldsymbol{n}} \boldsymbol{t} \times (\mathbf{0.99})^{\boldsymbol{t-1}}$$

We must break down the highlighted green sum, using basic summation tricks. Below is the following computation.

$$\boldsymbol{S_n} = \boldsymbol{S} = \sum_{\boldsymbol{t=1}}^{\boldsymbol{n}} \boldsymbol{t} \times (\mathbf{0.99})^{\boldsymbol{t-1}} = \mathbf{1} + (\mathbf{2} \times \mathbf{0.99}) + \left(\mathbf{3} \times \mathbf{0.99^2}\right) + \cdots + \left(\boldsymbol{n} \times \mathbf{0.99}^{\boldsymbol{n-1}}\right)$$

$$rS_n = 0.99S = \sum_{t=1}^{n} t \times 0.99^t = (1 \times 0.99) + (2 \times 0.99^2) + \cdots + (n-1) \times (0.99^{n-1}) + n \times 0.99^n$$

Taking the difference of these two, equations, we get:

$$S_n - rS_n = 0.01S$$

$$= 1 + (1 \times 0.99) + 0.99^2 + \cdots + 0.99^{n-1} - (n \times 0.99^n)$$

$$0.01S_n = \sum_{i=0}^{n-1} (0.99^i) \quad - \quad n \times 0.99^n$$

$$= \frac{1 - 0.99^n}{1 - 0.99}\ (By\ the\ sum\ of\ a\ geometric\ series\ formula) - (n \times 0.99^n)$$

$$= 100 - 100(0.99^n) - (n \times 0.99^n)$$

$$= 100 - 0.99^n(100 + n)$$

Now, we plug this back into our preceding part to finish off the said exercise.

$$E[t_n] = \sum_{t=1}^{n+1} t \times \Pr(t_n = t)$$

$$= 0.01 \sum_{t=1}^{n} t \times (0.99)^{t-1} \quad + \quad (n+1)(0.99)^n$$

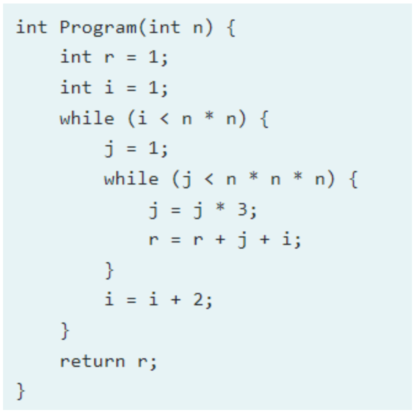
$$= (n+1)(0.99)^n + \mathbf{100 - 0.99^n(100 + n)}$$

$$= (n \times 0.99^n) + (0.99^n) + 100 - (100 \times 0.99^n) - (n \times 0.99^n)$$

$$= 100 - 99(0.99^n)$$

3) Determine the time-complexity of the following code fragments. Justify your answer. [HARD]

Problem #4: What is the worst-case running time of the function, $Program(int\ n)$.



```
int Program(int n) {
    int r = 1;
    int i = 1;
    while (i < n * n) {
        j = 1;
        while (j < n * n * n) {
            j = j * 3;
            r = r + j + i;
        }
        i = i + 2;
    }
    return r;
}
```

Outer Loop

- We can see that the outer loop, $while(i < n * n)\{ \dots i = i + 2\}$ will have a worst-case running time of $O\left(\frac{n^2}{2}\right) = O(n^2)$

Inner Loop

Notice here that the value of j changes accordingly:

1. $j = 1$
2. $i = 1 \times 3$
3. $i = 1 \times 3 \times 3$
4. $i = 1 \times 3 \times 3 \times 3$

   $(\dots)$

   3^k

The exit condition of the following loop occurs when $j \geq n^3$.

The value of $j = 3^k$. Thus, we can see that $3^k \geq n^3$ is also the equivalent exit condition.

Notice here that:

$3^k \geq n^3$

$log_3(3^k) \geq \log_3(n^3)$

$k \geq 3\log_3(n)$

Using the Change of Base, we can see that:

$$\log_3(n) = \frac{\log_2(n)}{\log_2(3)}$$

Therefore, the time complexity of the following algorithm can be seen to be $O(\mathrm{n}^2 \log_2(n))$.

Chapter 3 – Divide and Conquer; Recurrence

Course Material

Divide-and-Conquer

- In divide-and-conquer, we solve a problem **recursively**, applying three steps at each level of the recursion:
    1. **Divide** the problem into a number of subproblems that are **smaller instances of the same problem.**
    2. **Conquer** the subproblems by solving them **recursively.** If the subproblem sizes are small enough, just solve the subproblems in a straightforward manner.
    3. **Combine** the solutions to the subproblem into the solution for the original problem.

Recurrences

- Recurrences give us a natural way to **characterize the running times of divide-and-conquer algorithms**.
- A recurrence is an equation or inequality that describes a function in terms of its value on smaller inputs.
- Solving a recurrence gives us an **asymptotic Θ/O bound** on a given divide-and-conquer algorithm.

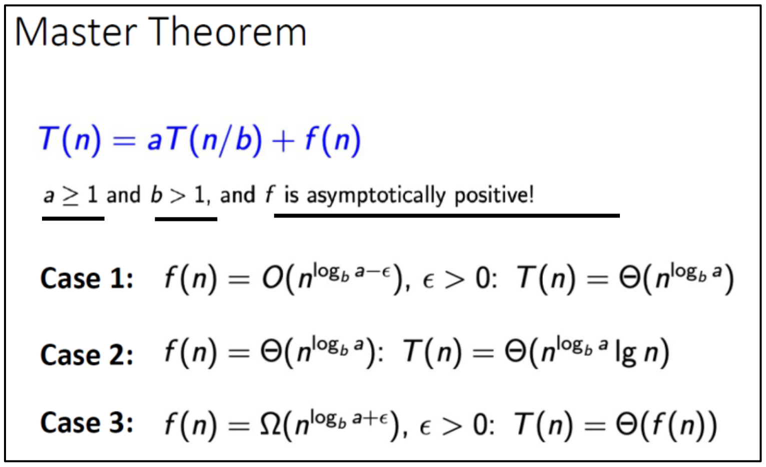
Typical Form of the Running Time of Divide-and-Conquer Algorithms

- $T(n) = aT\left(\frac{n}{b}\right) + f(n)$
  - a: The number of recursive calls (or, the number of subproblems.)
  - b: The rate at which the subproblem decreases.
  - $f(n)$: The runtime of the non-recursive parts of the algorithm (i.e., The divide and combine times.)

Methods for Solving a Recurrence Relation

1. **Substitution Method**
   a. We guess a bound and then use mathematical induction to prove the guess correct.
2. **Recursion Tree**
   a. Converts the recurrence into a tree whose **nodes represent the cost incurred at various levels of the recursion.** We use techniques for bounding summations to solve the recurrence.
   b. The root of the recursion tree represents the number of operations done, not including those of the recursion step.
3. **Master Theorem**
   a. Useful, however, it's not always applicable on all types of recurrence relations. It solves a recurrence relation, given a specific said known structure.

Master Theorem



Master Theorem

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

$\underline{a \geq 1}$ and $\underline{b > 1}$, and $\underline{f \text{ is asymptotically positive!}}$

Case 1: $f(n) = O(n^{\log_b a - \epsilon})$, $\epsilon > 0$: $T(n) = \Theta(n^{\log_b a})$

Case 2: $f(n) = \Theta(n^{\log_b a})$: $T(n) = \Theta(n^{\log_b a} \lg n)$

Case 3: $f(n) = \Omega(n^{\log_b a + \epsilon})$, $\epsilon > 0$: $T(n) = \Theta(f(n))$

Using the Master Theorem

1. Make sure that you can actually use the Master Theorem. This is to say that you should make sure that your choices of $a, b, and\ f(n)$ are acceptable arguments for this algorithm.
2. Start by calculating $\log_b(a)$:
   a. Apply Case 1 when: $f(n) < n^{\log_b(a)}$
   b. Apply Case 2 when: $f(n) = n^{\log_b(a)}$
   c. Apply Case 3 when: $f(n) > n^{\log_b(a)}$

Divide-and-Conquer with Master Theorem

- The combination of the two gives you the ability to very quickly iterate between **algorithm design and runtime analysis.**

Master Theorem: Caveats

- We ignore floors and ceilings when dividing the problem size, because it doesn't matter for the asymptotic bounds.
- We assume $T(1)$ is constant. (Which, it should be for real algorithms.)

Understanding the Recursion Tree

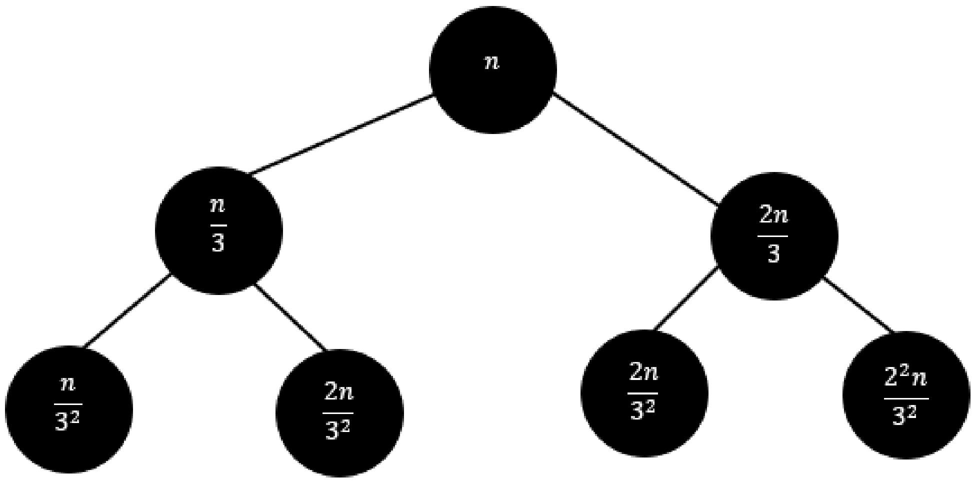
- The Recursion Tree is a useful tool for analyzing the runtime of a recursive problem **when the Master Theorem is not applicable.**
  - Each node represents the cost of a single subproblem somewhere in the set of recursive function invocations.

Recursion Tree Algorithm Specifics

1. We sum the cost within each level of the tree to obtain a set of per-level costs.
2. We sum all the per-level costs to determine the total cost of all levels of recursion.
3. We use the outputs generated to make a good guess for the recurrence relation solution. We can verify the solution by using the aforementioned substitution algorithm too.

Recitation Problems

1. Use the recursive tree method to get a tighter big-Oh bound on the recurrence:
   $T(n) = T\left(\frac{n}{3}\right) + T\left(\frac{2n}{3}\right) + n$



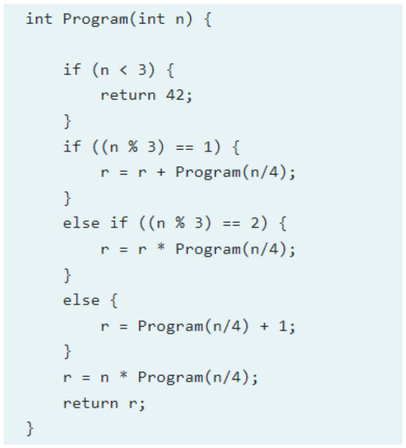
Observations

At each row, there is a total of n work being done.
We can see that the left-most path has a height of $\log_3(n)$
We can see that the right-most path has a height of $\log_{\frac{3}{2}}(n)$

Since we know that $\log_{\frac{3}{2}}(n) > \log_3(n)$, because logarithms with smaller bases grow faster, the tighter big-Oh bound on the recurrence relation can be calculated by:

$$Total\ Amount\ of\ Work\ Done = \sum Work\ per\ level \times \#(levels)$$
$$= nlog_{\frac{3}{2}}(n)$$
$$\approx nlog_2(n) \text{ (Omitting the change of base step line(s))}$$

2. What is the worst-case runtime of the following pseudocode?



```
int Program(int n) {

    if (n < 3) {
        return 42;
    }
    if ((n % 3) == 1) {
        r = r + Program(n/4);
    }
    else if ((n % 3) == 2) {
        r = r * Program(n/4);
    }
    else {
        r = Program(n/4) + 1;
    }
    r = n * Program(n/4);
    return r;
}
```

Given the following recursive algorithm, we can clearly see that there are two recursive calls being made. One is made in the if-else blocks, and the other prior to returning the known variable, r.

The subproblems decrease by a rate of 4 as well throughout the two recursive calls being made. The amount of non-recursive work being done at each call is constant.

That is, we are using basic arithmetic.

Seeing the structure of this recurrence relation, the Master Theorem can be used to find the worst-case runtime.

For $a = 2, b = 4$,

$$\log_4(2) = 0.5$$
$$n^{\log_4(2)} = \sqrt{n} > O(1)$$

Thus, by case 1 of the Master Theorem, we can see that $f(n) = O(\sqrt{n})$

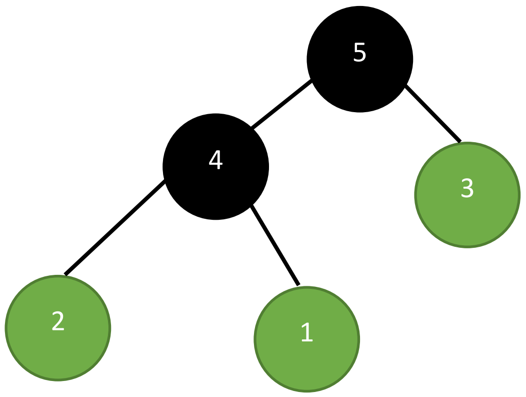
Chapter [4 + 5] – Sorting

Recitation Problems

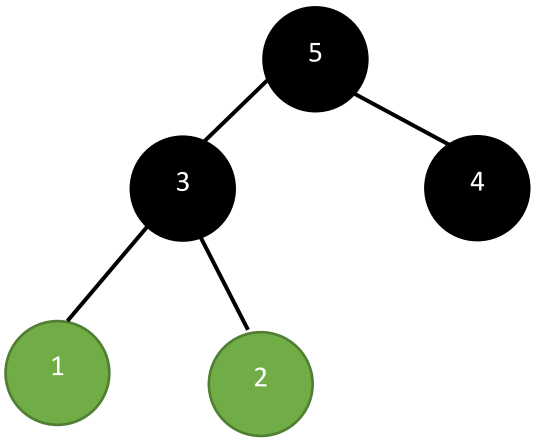
1. Calculate the number of all possible arrays that store a binary max-heap with keys: 1, 2, 3, 4, 5.

   Structure Cases

   Case 1



   Case 2



Notice that in case 1, the nodes $\{1,2,3\}$ can all be swapped with each other, maintaining a binary max heap. Thus, the number of possible arrays can be calculated as $\#(sol_{c_1}) = 3! = 6$

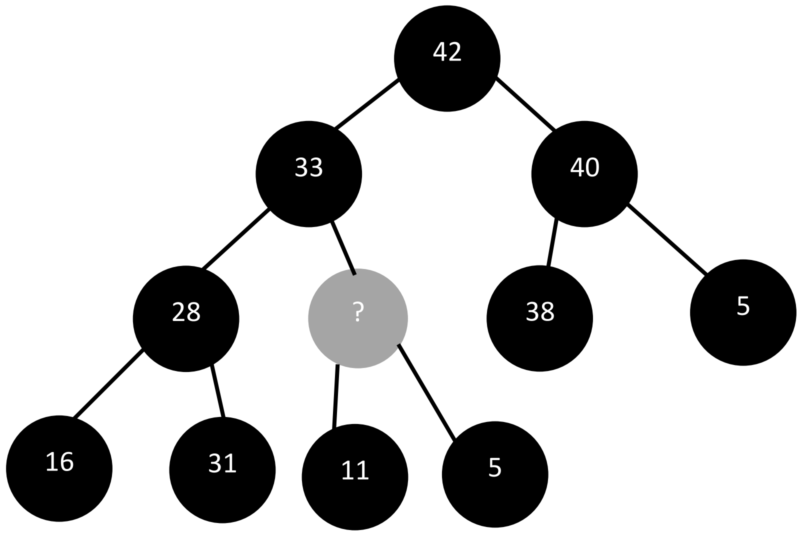
Notice that in case 2, the nodes $\{1, 2\}$ can be swapped with each other, maintaining a binary max heap. Thus, the number of possible arrays can be calculated as $\#(sol_{c_2}) = 2! = 2$

Thus, in total, there are $6 + 2 = 8$ possible arrays that store a binary max-heap with keys 1,2,3,4,5.

2. Fill in the blank in the following array so that it is a valid binary max-heap. Choose all numbers that apply.

   [42, 33, 40, 28, _____, 35, 38, 5, 16, 31, 11]

   <u>Potential Answers:</u> [18, 39, 32, 24, 29, 30, 34]



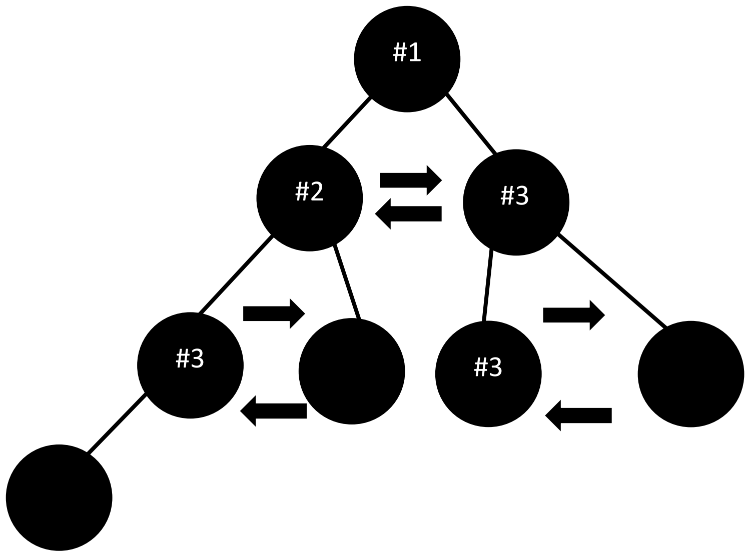
The potential candidate solution to fill the unknown node value must be restricted via:
$\mathbf{11 \leq ? \leq 33}$

Thus, from the candidate answers, the one possible answer must be 32.

3. Consider an array of size 8 that stores a binary max-heap. The indices of the array start from 0. Below, choose ALL possible indices that could have the third-largest element of the array. **Assume all elements are distinct.**

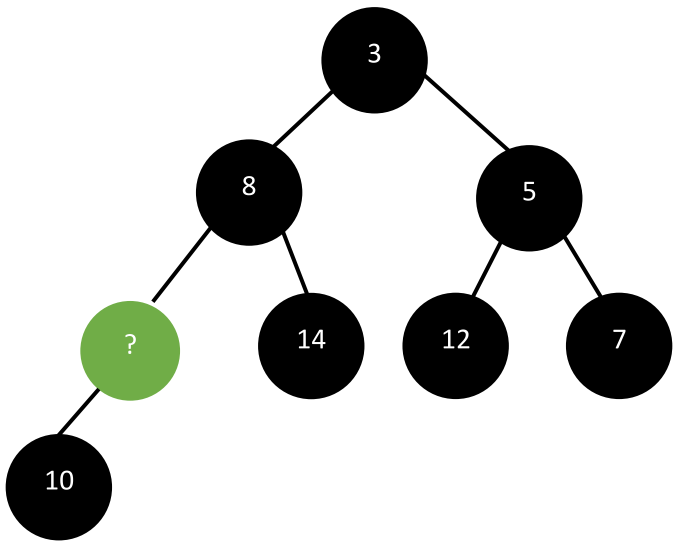
   Potential Answers: [0, 1, 2, 3, 4, 5, 6, 7]

   The third-largest element of the array could be in the indices [1, 2, 3, 4, 5, 6]



4. Fill in the blank in the following array with an integer value so that the array stores a valid binary min-heap. The value you enter must NOT be the same as any of the existing values in the array. Enter your answer in the box below.

   [3, 8, 5, BLANK, 14, 12, 7, 10]



The value of the unknown node must be constrained via: $8 \leq ? \leq 10$. Thus, it must be equal to 9 as the input.

5. How many leaf nodes are there in a binary heap with 321 nodes in total? Enter the number in the box below.

   $$\#(leaves) = \left\lceil \frac{321}{2} \right\rceil = 161 \; such \; leaf \; nodes$$