

Section 03: Solutions

Review Problems

1. Code Analysis

For each of the following code blocks, what is the worst-case runtime? Give a big- Θ bound.

```
(a) public IList<String> repeat(DoubleLinkedList<String> list, int n) {
    IList<String> result = new DoubleLinkedList<String>();
    for(String str : list) {
        for(int i = 0; i < n; i++) {
            result.add(str);
        }
    }
    return result;
}
```

Solution:

The runtime is $\Theta(nm)$, where m is the length of the input list and n is equal to the int n parameter.

One thing to note here is that unlike many of the methods we've analyzed before, we can't quite describe the runtime of this algorithm using just a single variable: we need two, one for each loop.

```
(b) public void foo(int n) {
    for (int i = 0; i < n; i++) {
        for (int j = 5; j < i; j++) {
            System.out.println("Hello!");
        }

        for (int j = i; j >= 0; j -= 2) {
            System.out.println("Hello!");
        }
    }
}
```

Solution:

The inner loop executes about $i - 5 + i/2$ operations per loop. So we execute about

$$\sum_{i=0}^{n-1} i - 5 + i/2 = \frac{3}{2} \sum_{i=0}^{n-1} i - \sum_{i=0}^{n-1} 5 = \frac{3}{2} * \frac{(0 + n - 1) * n}{2} - 5n = \frac{3n(n - 1)}{4} - 5n$$

which means the runtime is $\Theta(n^2)$.

```
(c) public int num(int n){
    if (n < 10) {
        return n;
    } else if (n < 1000) {
        return num(n - 2);
    } else {
        return num(n / 2);
    }
}
```

Solution:

The answer is $\Theta(\log(n))$.

One thing to note is that the second case effectively has no impact on the runtime. That second case occurs only for $n < 1000$ – when discussing asymptotic analysis, we only care what happens with the runtime as n grows large.

```
(d) public int foo(int n) {
    if (n <= 0) {
        return 3;
    }
    int x = foo(n - 1);
    System.out.println("hello");
    x += foo(n - 1);
    return x;
}
```

Solution:

The answer is $\Theta(2^n)$.

In order to determine that this is exponential, let's start by considering the following recurrence:

$$T(n) = \begin{cases} 1 & \text{If } n = 0 \\ 2T(n - 1) + 1 & \text{Otherwise} \end{cases}$$

While we could unfold this to get an exact closed form, we can approximate the final asymptotic behavior by taking a step back and thinking on a higher level what this is doing.

Basically, what happens is we take the work done by $T(n - 1)$ and multiply it by 2. If we ignore the +1 constant work done in the recursive case, the net effect is that we multiply 2 approximately n times. This simplifies to 2^n .

2. Binary Search Trees

- (a) Write a method `validate` to validate a BST. Although the basic algorithm can be converted to any data structure and work in any format, if it helps, you may write this method for the `IntTree` class:

```
public class IntTree {
    private IntTreeNode overallRoot;

    // constructors and other methods omitted for clarity

    private class IntTreeNode {
        public int data;
        public IntTreeNode left;
        public IntTreeNode right;

        // constructors omitted for clarity
    }
}
```

Solution:

```
public boolean validate() {
    return validate(overallRoot, Integer.MIN_VALUE, Integer.MAX_VALUE);
}

private boolean validate(IntTreeNode root, int min, int max) {
    if (root == null) {
        return true;
    } else if (root.data > max || root.data < min) {
        return false;
    } else {
        return validate(root.left, min, root.data - 1) &&
            validate (root.right, root.data + 1, max);
    }
}
```

Section Problems

3. Modeling recursive functions

(a) Consider the following method.

```
public static int f(int n) {
    if (n == 0) {
        return 0;
    }

    int result = 0;
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < i; j++) {
            result++;
        }
    }
    return 5 * f(n / 2) + 3 * result + 2 * f(n / 2);
}
```

- (i) Give a recurrence formula for the running time of this code. It's OK to provide a \mathcal{O} for the non-recursive terms (for example if the running time is $A(n) = 4A(n/3) + 25n$, you need to get the 4 and the 3 right but you don't have to worry about getting the 25 right). Just show us how you got there.

Hint: You may need to use Gauss's summation identity (see the last page).

Solution:

$$T(n) = \begin{cases} 1 & \text{When } n = 0 \\ \frac{n(n-1)}{2} + 2T(n/2) & \text{Otherwise} \end{cases}$$

The runtime for the two loops is $\sum_{i=0}^{n-1} \sum_{j=0}^{i-1} 1 = \sum_{i=0}^{n-1} i = \frac{n(n-1)}{2}$.

- (ii) Find a recurrence $W(n)$ modeling the returned integer output of $f(n)$.

Solution:

$$W(n) = \begin{cases} 0 & \text{When } n = 0 \\ \frac{3n(n-1)}{2} + 7W(n/2) & \text{Otherwise} \end{cases}$$

After the loop, result is $\sum_{i=0}^{n-1} \sum_{j=0}^{i-1} 1 = \sum_{i=0}^{n-1} i = \frac{n(n-1)}{2}$. And we use $3 * \text{result}$.

(b) Consider the following method.

```
public static int g(n) {
    if (n <= 1) {
        return 1000;
    }
    if (g(n / 3) > 5) {
        for (int i = 0; i < n; i++) {
            System.out.println("Hello");
        }
        return 5 * g(n / 3);
    } else {
        for (int i = 0; i < n * n; i++) {
            System.out.println("World");
        }
        return 4 * g(n / 3);
    }
}
```

(i) Find a recurrence $S(n)$ modeling the worst-case runtime of $g(n)$.

Solution:

$$S(n) = \begin{cases} 1 & \text{When } n \leq 1 \\ 2S(n/3) + n & \text{Otherwise} \end{cases}$$

Important: note that the if statement contains a recursive call that must be evaluated for $n > 1$.

(ii) Find a recurrence $X(n)$ modeling the returned integer output of $g(n)$.

Solution:

$$X(n) = \begin{cases} 1000 & \text{When } n \leq 1 \\ 5T(n/3) & \text{Otherwise} \end{cases}$$

(iii) Find a recurrence $P(n)$ modeling the printed output of $g(n)$.

Solution:

$$P(n) = 2P(n/3) + n$$

(c) Consider the following set of recursive methods.

```

public int test(int n) {
    IDictionary<Integer, Integer> dict = new AvlDictionary<>();
    populate(n, dict);
    int counter = 0;
    for (int i = 0; i < n; i++) {
        counter += dict.get(i);
    }
    return counter;
}

private void populate(int k, IDictionary<Integer, Integer> dict) {
    if (k == 0) {
        dict.put(0, k);
    } else {
        for (int i = 0; i < k; i++) {
            dict.put(i, i);
        }
        populate(k / 2, dict);
    }
}

```

(i) Write a mathematical function representing the *worst-case runtime* of test.

You should write two functions, one for the runtime of test and one for the runtime of populate.

Solution:

The runtime of the populate method is:

$$P(k) = \begin{cases} \log(N) & \text{When } k = 0 \\ k \log(N) + P(k/2) & \text{Otherwise} \end{cases}$$

Here, N is the maximum possible value of n .

The runtime of the test method is then $R(n) = P(n) + n \log(n)$.

4. Master Theorem

For each of the recurrences below, use the Master Theorem to find the big- Θ of the closed form or explain why Master Theorem doesn't apply. (See the last page for the definition of Master Theorem.)

(a) $T(n) = \begin{cases} 18 & \text{if } n \leq 5 \\ 3T(n/4) + n^2 & \text{otherwise} \end{cases}$ **Solution:**

This is the correct form for Master Theorem. We want to compare $\log_4(3)$ to 2. $\log_4(3)$ is between 0 and 1 (since $4^0 < 3 < 4^1$), so $\log_4(3) < 2$. We're thus in the case where the answer is $\Theta(n^2)$.

(b) $T(n) = \begin{cases} 1 & \text{if } n \leq 1 \\ 9T(n/3) + n^2 & \text{otherwise} \end{cases}$ **Solution:**

We want to compare $\log_3(9)$ to 2. $\log_3(9)$ is 2 (since $3^2 = 9$) since the two things we're comparing are equal, we have $\Theta(n^2 \log n)$ as our final answer.

(c) $T(n) = \begin{cases} 1 & \text{if } n \leq 1 \\ \log(n)T(n/2) + n & \text{otherwise} \end{cases}$ **Solution:**

This recurrence is not in the right form to use the Master Theorem. The coefficient of $T(n/2)$ needs to be a constant, not a function of n .

(d) $T(n) = \begin{cases} 1 & \text{if } n \leq 19 \\ 4T(n/3) + n & \text{otherwise} \end{cases}$ **Solution:**

We want to compare $\log_3(4)$ to 1. $\log_3(4)$ is between 1 and 2 (since $3^1 < 4 < 3^2$), so $\log_3(4) > 1$. In this case, the Master Theorem says our result is $\Theta(n^{\log_3(4)})$

(e) $T(n) = \begin{cases} 5 & \text{if } n \leq 24 \\ 2T(n-2) + 5n^3 & \text{otherwise} \end{cases}$ **Solution:**

This recurrence is not in the right form to use Master Theorem. It's only applicable if we are dividing the input size, not if we're subtracting from it.

5. Tree method walk-through

Consider the following recurrence: $T(n) = \begin{cases} 1 & \text{if } n \leq 1 \\ 3T(n/6) + 2n & \text{otherwise} \end{cases}$

We want to find an *exact* closed form of this equation by using the tree method. Suppose we draw out the total work done by this method as a tree, as discussed in lecture. Let n be the initial input to T .

(a) What is the size of the input at level i (as in class, call the root level 0)?

Solution:

We divide by 6 at each level, so the input size is $n/6^i$.

(b) What is the work done by a single node at non-base-case level i ?

Solution:

The non-recursive work done at recursive levels is given by the $+2n$ in the recurrence equation. So each node at (non-base-case) level i does $2 \times$ the input for that level. Using our answer from part a) we get that each node does $2n/6^i$ work.

(c) What is the number of nodes at level i ?

Note: let $i = 0$ indicate the level corresponding to the root node. So, when $i = 0$, your expression should be equal to 1.

Solution:

Each (non-base-case) node produces 3 more nodes, so at level i we have 3^i nodes.

(d) What is the total work at the i^{th} recursive level?

Solution:

Combining our last two parts: $3^i \cdot \frac{2n}{6^i} = \frac{2n}{2^i}$. You could simplify this further to be $\frac{n}{2^{i-1}}$ but we will leave it as is for now to make simplifying to a closed form easier later.

(e) What is the last level of the tree?

Solution:

We hit our base case when $n/6^i = 1$, which is at level $i = \log_6(n)$.

(f) What is the work done in the base case?

Solution:

From previous parts, there are $3^{\log_6(n)}$ nodes at that level, from the recurrence each does 1 unit of work, so we get $1 \cdot 3^{\log_6(n)}$ work.

(g) Combine your answers from previous parts to get an expression for the total work.

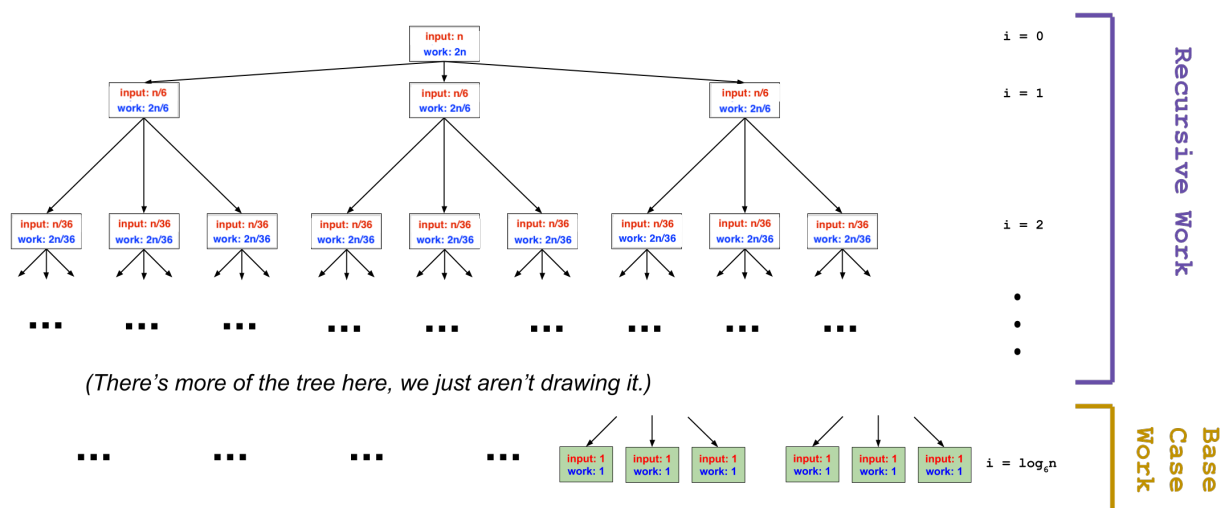
Solution:

We know the expression for the work done at each recursive level and the base case level. Combining these we have:

$$\sum_{i=0}^{\log_6(n)-1} \frac{2n}{2^i} + 3^{\log_6(n)}$$

Side note: Yes, it says $\log_6(n) - 1$ and not $\log_6(n)$. This is because the last level ($\log_6(n)$) is the base case level and has a different formula.

Here's a drawing of of the tree. Note that we distinguish the **input** to a node (red) and the **work** done by a node (blue) since these can be different when we're applying the tree method.



(h) Simplify to a closed form.

Note: you do not need to simplify your answer, once you found the closed form. Hint: You should use the finite geometric series identity somewhere while finding a closed form.

Solution:

We combine all the pieces and simplify:

$$T(n) = \sum_{i=0}^{\log_6(n)-1} \frac{2n}{2^i} + 3^{\log_6(n)}$$

Begin by taking the constant $2n$ outside of the summation as per the identity for factoring out a constant from the cheatsheet at the end of this document.

$$= 2n \sum_{i=0}^{\log_6(n)-1} \left(\frac{1}{2}\right)^i + 3^{\log_6(n)}$$

We'll apply two identities: to the summation we apply the finite geometric series identity; and to the base-case work, we apply the power of a log identity. We get:

$$T(n) = 2n \cdot \frac{\left(\frac{1}{2}\right)^{\log_6(n)} - 1}{\frac{1}{2} - 1} + n^{\log_6(3)}$$

You don't have to simplify further, but if you were to simplify, you would get:

$$\begin{aligned} T(n) &= 2n \cdot \frac{\left(\frac{1}{2}\right)^{\log_6(n)} - 1}{1/2 - 1} + n^{\log_6(3)} \\ &= -4n \left(n^{\log_6(1/2)} - 1 \right) + n^{\log_6(3)} \\ &= -4n \left(n^{\log_6(3/6)} - 1 \right) + n^{\log_6(3)} \\ &= -4n \left(n^{\log_6(3) - \log_6(6)} - 1 \right) + n^{\log_6(3)} \\ &= -4n \left(\frac{n^{\log_6(3)}}{n} - 1 \right) + n^{\log_6(3)} \\ &= -4n^{\log_6(3)} + 2n + n^{\log_6(3)} \\ &= 4n - n^{\log_6(3)} \end{aligned}$$

(i) Use the master theorem to find a big- Θ bound of $T(n)$. (See the last page for the definition of Master Theorem.)

Solution:

We check that $\log_6(3) < 1$, so we get $T(n) \in \Theta(n)$, which is consistent with our answer in the last part.

6. More tree method recurrences

For each of the following recurrences, find their closed form using the tree method. Then, check your answer using the master method (if applicable). It may be a useful guide to use the steps from section 4 of this handout to help you with all the parts of solving a recurrence problem fully.

$$(a) T(n) = \begin{cases} 1 & \text{if } n = 1 \\ T(n/2) + 3 & \text{otherwise} \end{cases}$$

Solution:

Given this recurrence, we know...

- Input size at level i : $n/2^i$
- Number of nodes on level i : $1^i = 1$
- Total work done per level: $1 \cdot 3 = 3$
- Last level of the tree: $\log_2(n)$
- Total work done in base case: $1 \cdot 1^{\log_3(n)} = 1$

So we get the expression:

$$\left(\sum_{i=0}^{\log_2(n)-1} 3 \right) + 1$$

Using the summation of a constant identity, we get:

$$3 \log_2(n) + 1$$

We can apply the master theorem here. Note that $\log_b(a) = \log_2(1) = 0 = c$, which means that $T(n) \in \Theta(n^c \log(n))$ which is $T(n) \in \Theta(n^0 \log(n))$ which further simplifies to $T(n) \in \Theta(\log(n))$.

This agrees with our simplified form.

$$(b) S(q) = \begin{cases} 1 & \text{if } q = 1 \\ 2S(q-1) + 1 & \text{otherwise} \end{cases}$$

Solution:

Given this recurrence, we know...

- Size of input at level i is $q - i$
- Number of nodes on level i : 2^i
- Total work done at (recursive) level i : $2^i \cdot 1$
- Last level of the tree: $q - 1$
- Total work done in base case: $1 \cdot 2^{q-1}$

Note that these expressions look a little different from the ones we've seen up above. This is because we aren't *dividing* our terms by some constant factor – instead, we're *subtracting* them.

So we get the expression:

$$\left(\sum_{i=0}^{q-1-1} 2^i \right) + 2^{q-1}$$

We apply the finite geometric series to get:

$$\frac{2^{q-1} - 1}{2 - 1} + 2^{q-1}$$

If we wanted to simplify, we'd get:

$$2^q - 1$$

Note that we may NOT apply the master theorem here – our original recurrence doesn't match the form given in the theorem.

$$(c) T(n) = \begin{cases} 1 & \text{if } n = 1 \\ 8T(n/2) + 4n^2 & \text{otherwise} \end{cases}$$

Solution:

Given this recurrence, we know...

- Size of input at level i is $n/2^i$
- Number of nodes on level i : 8^i
- Total work done per (recursive) level: $8^i \cdot 4 \left(\frac{n}{2^i}\right)^2 = 8^i \cdot 4 \cdot \frac{n^2}{4^i}$
- Last level of the tree: When $n/2^i = 1$, i.e. $\log_2(n)$
- Total work done in base case: $1 \cdot 8^{\log_2(n)}$

So we get the expression:

$$\left(\sum_{i=0}^{\log_2(n)-1} 8^i \cdot 4 \cdot \frac{n^2}{4^i} \right) + 8^{\log_2(n)}$$

We can simplify by pulling the $4n^2$ out of the summation:

$$4n^2 \left(\sum_{i=0}^{\log_2(n)-1} \frac{8^i}{4^i} \right) + 8^{\log_2(n)}$$

This further simplifies to:

$$4n^2 \left(\sum_{i=0}^{\log_2(n)-1} 2^i \right) + 8^{\log_2(n)}$$

After applying the finite geometric series identity, we get:

$$4n^2 \cdot \frac{2^{\log_2(n)} - 1}{2 - 1} + 8^{\log_2(n)}$$

This is a closed form so we could stop, but if we want a tidy solution, we can continue simplifying:

$$\begin{aligned} T(n) &= 4n^2 \cdot \frac{2^{\log_2(n)} - 1}{2 - 1} + 8^{\log_2(n)} \\ &= 4n^2 \cdot (2^{\log_2(n)} - 1) + 8^{\log_2(n)} \\ &= 4n^2 \cdot (n^{\log_2(2)} - 1) + n^{\log_2(8)} \\ &= 4n^2 \cdot (n - 1) + n^3 \\ &= 5n^3 - 4n^2 \end{aligned}$$

We can apply the master theorem here. Note that $\log_b(a) = \log_2(8) = 3 > 2 = c$, which means that $T(n) \in \Theta(n^{\log_b(a)})$ which is $T(n) \in \Theta(n^{\log_2 8})$ which in turn simplifies to $T(n) \in \Theta(n^3)$.

This agrees with our simplified form.

Food for Thought

7. TreeMap implemented as a Binary Search Tree

Consider the following method, which is a part of a Binary Search Tree implementation of a TreeMap class.

```
public V find(K key) {
    return find(this.root, key);
}

private V find(Node<K, V> current, K key) {
    if (current == null) {
        return null;
    }
    if (current.key.equals(key)) {
        return current.value;
    }
    if (current.key.compareTo(key) > 0) {
        return find(current.left, key);
    } else {
        return find(current.right, key);
    }
}
```

- (a) We want to analyze the runtime of our `find(x)` method in the best possible case and the worst possible case. What does our tree look like in the best possible case? In the worst possible case?

Solution:

In the best possible case, our tree will be completely balanced. In the worst possible case, it will be completely unbalanced, resembling a linked list.

- (b) Write a recurrence to represent the worst-case runtime for `find(x)` in terms of n , the number of elements contained within our tree. Then, provide a Θ bound.

Solution:

The recurrence representing the worst-case runtime of `find(x)` is:

$$T_w(n) = \begin{cases} 1 & \text{when } n = 0 \\ 1 + T(n - 1) & \text{otherwise} \end{cases}$$

That is, every time we recurse, we are able to eliminate only one node from the span of possibilities we must consider. This is possible in case the tree is absolutely unbalanced (think of a tree that looks like a linked list).

This recurrence is in $\Theta(n)$.

- (c) Assuming we have an optimally structured tree, write a recurrence for the runtime of `find(x)` (again in terms of n). Then, provide a Θ bound.

Solution:

The recurrence representing the best-case runtime of `find(x)` is:

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

That is, every time we recurse, we are able to eliminate about half of the nodes we must consider.

This recurrence is in $\Theta(\log(n))$.

8. Maximum Difference Between Node and Ancestor

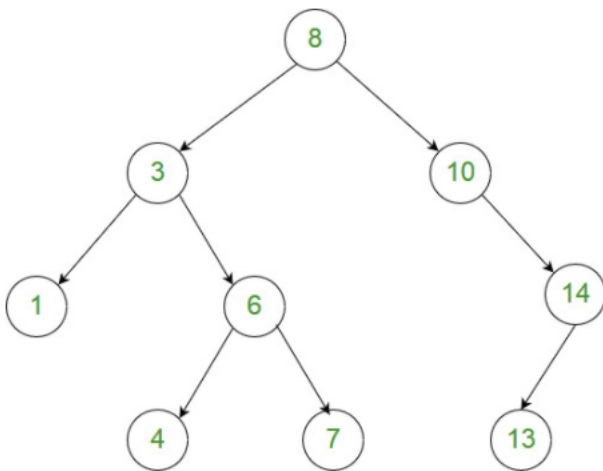
- (a) Let a `TreeNode` be defined as the following:

```
public class TreeNode {
    int val;
    TreeNode left;
    TreeNode right;
    TreeNode() {}
    TreeNode(int val) { this.val = val; }
    TreeNode(int val, TreeNode left, TreeNode right) {
        this.val = val;
        this.left = left;
        this.right = right;
    }
}
```

Given the root of a binary tree, find the maximum value V for which there exist different nodes A and B where $V = |A.val - B.val|$ and A is an ancestor of B .

A node A is an ancestor of B if either any child of A is equal to B , or any child of A is an ancestor of B .

Consider the following example:



Input: [8, 3, 10, 1, 6, null, 14, null, null, 4, 7, 13]

Output: 7

Explanation: We have various ancestor-node differences, some of which are given below:

$$\begin{aligned} |8 - 3| &= 5 \\ |3 - 7| &= 4 \\ |8 - 1| &= 7 \\ |10 - 13| &= 3 \end{aligned}$$

Among all possible differences, the maximum value of 7 is obtained by $|8 - 1| = 7$.

Solution:

First, recall that this is a **binary tree** and NOT a binary search tree. This means that the tree has no underlying ordering which makes a difference in the solution. The fact that the binary tree is not ordered means we must check all of the nodes to find the maximum difference between ancestors.

At a high level what we want to figure out is the global minimum and global maximum value at each **subtree** (not for the overall tree, because the result is concerned with nodes that need to be ancestors) and to calculate the difference to update our global result variable.

Thus, a possible implementation might look like:

```
class Solution {
    int res = Integer.MIN_VALUE;
    public int maxAncestorDiff(TreeNode root) {
        dfs(root, root.val, root.val);
        return res;
    }
    private void dfs(TreeNode node, int min, int max) {
        if (node == null) return;
        min = Math.min(node.val, min);
        max = Math.max(node.val, max);
        res = Math.max(res, max - min);
        dfs(node.left, min, max);
        dfs(node.right, min, max);
    }
}
```

Time complexity: $O(n)$ where n is the number of nodes in the tree. It is not a binary search tree so we need to traverse every single node to find maximum and minimum.

Space complexity: $O(1)$ since we are not using any extra data structures for storage

Master Theorem

For recurrences in this form, where a, b, c, e are constants:

$$T(n) = \begin{cases} d & \text{if } n \leq \text{some constant} \\ aT(n/b) + e \cdot n^c & \text{otherwise} \end{cases} \quad T(n) \text{ is } \begin{cases} \Theta(n^c) & \text{if } \log_b(a) < c \\ \Theta(n^c \log n) & \text{if } \log_b(a) = c \\ \Theta(n^{\log_b(a)}) & \text{if } \log_b(a) > c \end{cases}$$

Useful summation identities

Splitting a sum

$$\sum_{i=a}^b (x + y) = \sum_{i=a}^b x + \sum_{i=a}^b y$$

Adjusting summation bounds

$$\sum_{i=a}^b f(x) = \sum_{i=0}^b f(x) - \sum_{i=0}^{a-1} f(x)$$

Factoring out a constant

$$\sum_{i=a}^b cf(i) = c \sum_{i=a}^b f(i)$$

Summation of a constant

$$\sum_{i=0}^{n-1} c = \underbrace{c + c + \dots + c}_{n \text{ times}} = cn$$

Note: this rule is a special case of the rule on the left

Gauss's identity

$$\sum_{i=0}^{n-1} i = 0 + 1 + 2 + \dots + n - 1 = \frac{n(n-1)}{2}$$

Finite geometric series

$$\sum_{i=0}^{n-1} x^i = 1 + x + x^2 + \dots + x^{n-1} = \frac{x^n - 1}{x - 1}$$

Infinite geometric series

$$\sum_{i=0}^{\infty} x^i = 1 + x + x^2 + \dots = \frac{1}{1-x}$$

Note: applicable only when $-1 < x < 1$

Useful Log Rules

Power of a log identity

$$a^{\log_b c} = c^{\log_b a}$$

Product rule

$$\log_c(a * b) = \log_c a + \log_c b$$

Quotient rule

$$\log_c(a/b) = \log_c a - \log_c b$$

Power rule

$$\log_c(a^b) = b * \log_c a$$

Change of base formula

$$\log_b a = (\log_c a) / (\log_c b)$$