Homework 2

Jiaxin Li[1] and Sunjun Gu[2]

[1]jli2274@wisc.edu

[2]sgu59@wisc.edu

1 Question 1

1.1 (a)

Algorithm 1 1(a)

Input: An integer n, $n \geq 0$, where 10^n is the number to present in binary
Output: The binary representation of 10^n;

```
0: procedure PowTen(n)
1: if n = 0 then
2:    return 1_2
3: else if n = 1 then
4:    return 1010_2
5: else
6:    m ← ⌊n/2⌋
7:    c ← PowTen(m)
8:    s ← Integer − Multiplication(c, c)
9:    if n is even then
10:      return s
11:   else
12:      s1 ← s left shifted by 3 positions
13:      s2 ← s left shifted by 1 positions
14:      return (s1 + s2)
15:   end if
16: end if
16: end procedure=0
```

∗**Integer − Multiplication** is the same with algorithm in the lecture notes with the same name.
where **Integer − Multiplication**(**a**, **b**)

Input:Two numbers a and b, in binary, two n-bit integers.

Output: The number a·b, in binary

Runtime: $O(n^{log_2 3})$

Proof of Correctness :
The proposition here P(n) is that this algorithm can return the correct binary representation of 10^n given the input integer n ($n \geq 0$) which is the power exponent of the decimal number 10^n.
Base case : When $n = 0$, the decimal number is 1; $n = 1$, the decimal number is 10. Check the Decimal and Binary Comparison Table, $1_{10} = 1_2, 10_{10} = 1010_2$. Therefore, P(0), P(1) is correct.

Induction Step : By strong induction, suppose $P(0), P(1), \ldots, P(k), (k \geq 0, k \in Z)$ is correct. Such that given a decimal number 10^k with the power exponent k, the algorithm can return the correct binary representation of 10^k.
To prove the correctness of $P(k+1)$:

The given decimal number 10^{k+1} has the power exponent $k+1$.

$k \geq 1 \rightarrow k+1 \geq 2 \rightarrow n \geq 2$

so $line5 \sim line14$ will be executed.

Line 5: $m = \lfloor \frac{n}{2} \rfloor = \lfloor \frac{k+1}{2} \rfloor$

Line 6: Because $k \geq 1, k \in Z \rightarrow 2k \geq k+1 \rightarrow k \geq \frac{k+1}{2}$, and $\lfloor \frac{k+1}{2} \rfloor$ is applied the floor function. Therefore, $0 \leq m = \lfloor \frac{k+1}{2} \rfloor \leq \frac{k+1}{2} \leq k$ and $P(m)$ is correct under the previous assumptions. $c = P(m)$ is the binary representation of 10^m.

Line 7: Apply the Integer-Multiplication algorithm on c, get the number $c \cdot c$ in binary, marking as s.

Line 10 $\sim$ 11: When n is even, namely $k+1$ is even, we can write $10^{k+1} = 10^{\frac{k+1}{2}} \cdot 10^{\frac{k+1}{2}} = 10^m \cdot 10^m = 10^{m+m}$, that is because when $k+1$ is even$m = \lfloor \frac{k+1}{2} \rfloor = \frac{k+1}{2}$,$m+m = \lfloor \frac{k+1}{2} \rfloor + \lfloor \frac{k+1}{2} \rfloor = \frac{k+1}{2} + \frac{k+1}{2} = k+1$; Therefore, we could see the binary representation of 10^{k+1} as the binary representation of $10^m \cdot 10^m$. From line 6, we get the binary representation of 10^m, $c = P(m)$. From line 7, we get the binary integer multiplication of c and c, $s = Integer - Multiplication(c, c)$. Therefore, s is the binary representation of 10^{k+1}.

Line 12 $\sim$ 14: When n is odd, namely $k+1$ is odd, we can write $10^{k+1} = 10^{\frac{k}{2}} \cdot 10^{\frac{k}{2}} \cdot 10^1 = 10^m \cdot 10^m \cdot 10^1 = 10^{m+m+1}$, that is because $m = \lfloor \frac{k+1}{2} \rfloor = \frac{k}{2}, 10^{m+m+1} = 10^{k+1}$. Therefore, we could see the binary representation of 10^{k+1} as the binary representation of $10^m \cdot 10^m \cdot (8+2)$. From line 6, we get the binary representation of 10^m, $c = P(m)$. From line 7, we get the binary integer multiplication of c and c, $s = Integer - Multiplication(c, c)$. Check the Decimal and Binary Comparison Table, $8_{10} = 1000_2$, $2_{10} = 10_2$.Therefore, the binary representation of 10^{k+1} is $s \cdot (1000_2 + 10_2) = s \cdot 1000_2 + s \cdot 10_2$. The value $s \cdot 1000_2$ could be get by s left shifted by 3 positions and stored as s1 and the value $s \cdot 10_2$ could be get by s left shifted by 1 positions and stored as s2, add s1 and s2 up. This is the binary representation of 10_{k+1} when n is odd.

Therefore, the binary representation of 10^{k+1} would be returned correctly by the algorithm.
Overall,$P(0), P(1), P(2), \ldots, P(k) \rightarrow P(k+1)$.
By mathematical inductive principle, P(n) is true for all $n \geq 0$.

Proof of Time Complexity :
We use the recursion tree to organize the calculation. Since the recursive case of the implementation makes one recursive calls (line 7), the recursion tree is a line.

In the worst recursive case, any given execution of PowTen makes one recursive call(line 7),

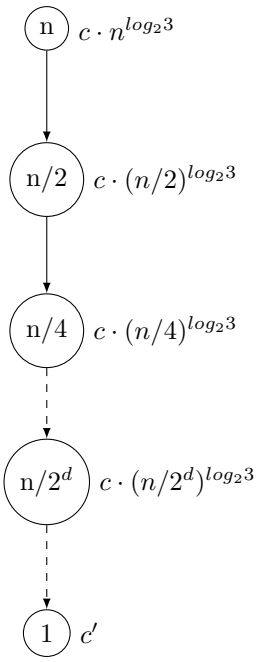

Figure 1: Depth: $log_2 n$

one call to Integer- Multiplication(line 8), bit shift work(line 12), and O(1) additional work(other lines). As discussed in class, Integer- Multiplication takes O($n^{log_2 3}$) time in the input exponent index , bit shift takes O(n) linear time; it follows that each level of the recursion tree does a combined work. And The recursive calls of PowTen decreases the input into equal-size halves from n,n/2,n/4,...,n/2^d, ..., 1.($d\,is\,the\,depth\,of\,the\,tree$).

The common ratio of the geometric sequence formed by the running time of each layer is $\frac{1}{2^{log_2 3}} = \frac{1}{3} < 1$, therefore the geometric series will converge to a constant. After computation, the total running time of PowTen is O($n^{log_2 3}$).
The mathematical computation as following:

$$
\begin{aligned}
Time &= c \cdot n^{log_2 3} + c \cdot (n/2)^{log_2 3} + ...c \cdot (n/2^d)^{log_2 3} + ... + c' \\
&= \sum_{d=0}^{log_2 n} c \cdot (\frac{n}{2^d})^{log_2 3} \\
&= c \cdot n^{log_2 3} \sum_{d=0}^{log_2 n} (\frac{1}{2^{log_2 3}})^d \\
&= c \cdot n^{log_2 3} \sum_{d=0}^{log_2 n} (\frac{1}{3})^d \\
&\leq c \cdot n^{log_2 3} \sum_{d=0}^{\infty} (\frac{1}{3})^d \\
&= c \cdot n^{log_2 3} (\frac{3}{2})
\end{aligned}
$$

The operations that are performed locally in the recursive case and in the base case are different. The constants hidden in the O are different. We use c as the constant in the recursive case, and c' in the base case. However, as it is constant, it would have little impact on our total runtime, we could ignore it. And ignore other relative constants, finally runtime is O($n^{log_2 3}$).

1.2 (b)

Algorithm 2 1(b)

Input: An n-digit decimal number X, where n is an integer and $n \geq 1$.
Output: The binary representation of the n-digit decimal number X;

```
0: procedure DecToBin(X)
1: if n = 1 then
2:    return X_2,from table
3: else
4:    R ← Lowest ⌈n/2⌉ digits of X
5:    L ← The remainder digits of X
6:    R' ← DecToBin(R)
7:    L' ← DecToBin(L)
8:    P' ← PowTen(⌈n/2⌉)
9:    return Integer − Multiplication(L', P') + R'
10: end if
10: end procedure=0
```

∗**Integer − Multiplication** is the same with algorithm in the lecture notes with the same name. where **Integer − Multiplication**(**a**, **b**)

Input:Two numbers a and b, in binary, two n-bit integers.

Output: The number a·b, in binary

Runtime: $O(n^{log_2 3})$

$*$**PowTen** is the same with algorithm in 1(a) with the same name.

where **PowTen**(**a**, **b**)

Input:An integer n,n0, where 10^n is the number to present in binary

Output: The binary representation of 10^n

Runtime: $O(n^{log_2 3})$

Proof of Correctness :

The proposition here P(n) is that this algorithm can return the correct binary representation of n-digit decimal number X ($n \geq 1$).

Base case : When $n = 1$, the binary representation of X could be correctly checked from the Decimal and Binary Comparison Table. Therefore, P(1) is correct.

Induction Step : By strong induction, suppose $P(1), P(2), \ldots, P(k), (k \geq 1, k \in Z)$ is correct. Such that given a decimal number X with n digits, the algorithm can return the correct binary representation of X.

To prove the correctness of $P(k+1)$:

The given decimal number X has $k+1$ digits.

$k \geq \rightarrow k+1 = n \geq 2$

so $line4 \sim line9$ will be executed.

Line 4: Divide the n-digit decimal number X into Low digit part and high digit part. Store the lowest $\lceil \frac{k+1}{2} \rceil$ digits of X in R

Line 5: Store the high digit part, namely the remainder digits of X in L.

Line 6: Because $k+1 \geq 2 \rightarrow \frac{k+1}{2} \geq 1 \rightarrow \lceil \frac{k+1}{2} \rceil \geq 1$, and $k \geq 1 \rightarrow 2k \geq k+1 \rightarrow \frac{k+1}{2} \leq k \rightarrow \lceil \frac{k+1}{2} \rceil \leq k$.Therefore, DecToBin(R) is correct under the previous assumption for $\lceil \frac{k+1}{2} \rceil$ digits. And Store the binary representation of R by the recursively call on DecToBin on R', R' = DecToBin(R)

Line 7: Because L is the left part of (k+1)-digit X and by line 6 statement, R must have digits no smaller than 1 and no bigger than k, that is, the left part L must have digits between (k+1)-k=1 and (k+1)-1=k. Therefore, by the previous assumption, DecToBin(L) is correctly return the binary representation of L, store it on L',L' = DecToBin(L)

Line 8: By line 4 and 5, we could write X as $X = L \cdot 10^{\lceil \frac{k+1}{2} \rceil} + R$. Therefore, the binary representation of X is $L_2 \cdot 10^{\lceil \frac{k+1}{2} \rceil}{}_2 + R_2$. By the correctness of the previous assumptions, we could get L_2 and R_2 and now we need $10^{\lceil \frac{k+1}{2} \rceil}{}_2$. For the binary representation of 10_n, we could call the PowTen algorithm on 1(a).Store it on P', $P' = PowTen(10^{\lceil \frac{k+1}{2} \rceil})$

Line 9: By the correctness of the previous assumptions, we write $X_2 = L_2 \cdot 10^{\lceil \frac{k+1}{2} \rceil}{}_2 + R_2 = L' \cdot P' + R'$. The binary integer multiplication could apply the lecture's Integer- Multiplication algorithm on L' and P'. Finally, $Integer - Multiplication(L', P') + R'$ is the binary representation of k+1 digits decimal number X

Therefore, the binary representation of k+1 digits decimal number X would be returned correctly by the algorithm.

Overall,$P(1), P(2), \ldots, P(k) \rightarrow P(k+1)$.

By mathematical inductive principle, P(n) is true for all $n \geq 1$.

Proof of Time Complexity :
We use the recursion tree to organize the calculation. Since the recursive case of the implementation makes two recursive calls (line 6,7), the recursion tree is a binary tree.

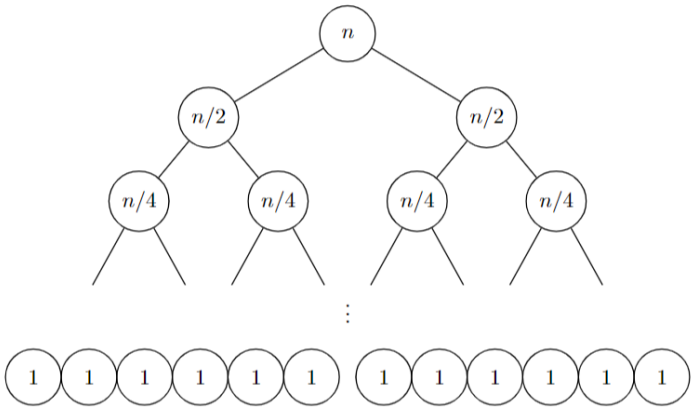

Figure 2: recursion tree with depth: $log_2 n$
For the recursion tree, it has depth(d) of $log_2 n$,and work of each layer from top to bottom is $c \cdot n^{log_2 3}$, $2 \cdot c \cdot \frac{n}{2}^{log_2 3}$,...,$2^d \cdot c \cdot \frac{n}{2^d}^{log_2 3}$,...,$c'$

For runtime of each line, Line 1,2 take O(1) time to work; Line 4,5 take O(n) time to store the low hand high part of X;Line 6,7 take 2 recursive calls on the low and high part of X. Line 8,9 call the PowTen and Integer-Multiplication algorithm which both take O($n^{log_2 3}$) to work as discussed in class.In the worst recursive case, any given execution of DecToBin makes 2 recursive call(line 6,7), one call to store the low and high part of X(line 4,5), one call on PowTen algorithm(line 8),one call on Integer-Multiplication algorithm(line 9) and O(1) additional work; it follows that each level of the recursion tree does a combined work. And The recursive calls of DecToBin divide the input into equal-size halves two sub-input from n,n/2,n/4,...,n/2^d, ..., 1($d\,is\,the\,depth\,of\,the\,tree$).

The common ratio of the geometric sequence formed by the running time of each layer is $\frac{2}{2^{log_2 3}} = \frac{2}{3} < 1$, therefore the geometric series will converge to a constant. After computation, the total running time of DecToBin is O($n^{log_2 3}$).
The mathematical computation as following:

$$
\begin{aligned}
Time &= c \cdot n^{log_2 3} + 2 \cdot c \cdot (n/2)^{log_2 3} + ...2^d \cdot c \cdot (n/2^d)^{log_2 3} + ... + c' \\
&= \sum_{d=0}^{log_2 n - 1} 2^d \cdot c \cdot (\frac{n}{2^d})^{log_2 3} \\
&= c \cdot n^{log_2 3} \sum_{d=0}^{log_2 n - 1} 2^d \cdot (\frac{1}{2^{log_2 3}})^d \\
&= c \cdot n^{log_2 3} \sum_{d=0}^{log_2 (n-1)} 2^d \cdot (\frac{1}{3})^d \\
&\leq c \cdot n^{log_2 3} \sum_{d=0}^{\infty} (\frac{2}{3})^d \\
&= 3 \cdot c \cdot n^{log_2 3}
\end{aligned}
$$

The operations that are performed locally in the recursive case and in the base case are different. The constants hidden in the O are different. We use c as the constant in the recursive case, and c' in the base case. However, as it is constant, it would have little impact on total runtime, we could ignore it. And ignore other relative constants, finally runtime is O($n^{log_2 3}$).

2 Question 3

The algorithm is on page 8, and the recursion tree used in proof is at the end of the file.

Proof of correctness:

The proposition here **P(n)** is that the MaxValue algorithm can return correct maximum total value that can be achieved by a subset of the given **n** items with their values **V** and weights **WT**, when the weight limit W is given and all items with value greater than the minimum V of the subset will be included in this subset.

Base Case: This algorithm has 2 base cases

(1) when $n = 0$, there is not any item to put in the knapsack. So no matter what value W (weight limit) is, the maximum total value is 0. It is correct that the algorithm returns 0. P(0) is correct. (Line $2 \sim 3$)

(2) when $n = 1$, there is only 1 item in this knapsack.

If this item's weight exceeds the weight limitation, then none item can be put in this knapsack. It is correct that the algorithm returns 0. (Line $7 \sim 8$)

If this item's weight is smaller than the weight limitation, then this item should be put in this knapsack. So the maximum total value is exactly the value of this item. It is correct that the algorithm returns this item's value as max value. (Line $4 \sim 6$)

Overall, P(1) is correct.

Algorithm 3 MaxValue

Input: **X[1,...n]**, an array of nodes **(V,WT)**, in each node there is a field of item's value **V** and a field of item's weight **WT**; **W**, the maximum weight the knapsack can hold

Output: **m**, the maximum total value that can be achieved by a subset of the items

```
0: procedure MaxValue(X,W)
1: n ← the length of X
2: if n = 0 then
3:    return 0
4: else if n = 1 then
5:    if (X[1] → WT) ≤ W then
6:       return X[1] → V
7:    else
8:       return 0
9:    end if
10: else
11:    valueArray ← newarray
12:    for i = 1 → n do
13:       Put (X[i] → V) at the end of valueArray
14:    end for
15:    pivot ← (SELECT(valueArray, ⌊(n+1)/2⌋)
16:    L, R ← newarrays
17:    sumR ← 0
18:    valueR ← 0
19:    pivotItem ← NULL
20:    for i = 1 → n do
21:       if (X[i] → V) < pivot then
22:          Put X[i] at the end of L
23:       else if (X[i] → V) > pivot then
24:          Put X[i] at the end of R
25:          sumR = sumR + (X[i] → WT)
26:          valueR = valueR + (X[i] → V)
27:       else
28:          pivotItem ← X[i]
29:       end if
30:    end for
31:    if sumR < W then
32:       Put pivotItem at the end of L
33:       m ← valueR + MaxValue(L, W − sumR)
34:    else if sumR > W then
35:       m ← MaxValue(R, W)
36:    else
37:       m ← valueR
38:    end if
39:    return m
40: end if
40: end procedure=0
```

Inductive Step: Here we use **Strong Induction** instead of general mathematical induction. Suppose P(j) is correct for $j = 0, 1, 2, ..., k; (k \geq 1)$. Such that given **less or equal to k** items and weight limit **W** , the algorithm can return the correct max value under the restriction in the context. To prove the correctness of P(k+1):

In (k+1)th situation, we have $n = k + 1$ items.

$k \geq 1 \Rightarrow n = k + 1 \geq 2$

So in this situation, Line10 $\sim$ 40 will be executed.

Line 11 $\sim$ 14: These lines iterate each item to get the value of the item and store these k+1 values in a new array called valueArray.

Line 15: Use SELECT algorithm introduced in lecture notes to get the $\lfloor \frac{n+1}{2} \rfloor$th smallest value in valueArray, (when $n \geq 2$, $1 \leq \lfloor \frac{n+1}{2} \rfloor < n$). This value is assigned to *pivot*

Line 20 $\sim$ 29: By iterating the array X, these lines split those items in X into 3 parts, items with value **strictly** greater than pivot will be stored in **R**, which is an array of notes; items with value **strictly** less than pivot will be stored in **L**, which is also an array of notes. And the item with value equals to pivot will be recorded in a variable called **pivotItem**

If n is even:

$\lfloor \frac{n+1}{2} \rfloor = \frac{n}{2}$, then L has $\frac{n}{2} - 1$ items, R has $\frac{n}{2}$ items.

If n is odd:

$\lfloor \frac{n+1}{2} \rfloor = \frac{n+1}{2}$, then L has $\frac{n-1}{2}$ items, R has $\frac{n-1}{2}$ items.

By spliting the array, we also get the sum of the values in R ($valueR$) and the sum of the weights in R ($sumR$).

Because of the additional restriction in this problem, the subset that achieve the maximum value could only be:

(1) A subset of R (including R itself or $\emptyset$)

(2) Or the $R \cup S$, where $S \subseteq (L \cup pivotItem)$

If it is situation (1), then the problem is reduced to find a subset in **R** instead of the whole list of items X. In such situation, the sum of the weights in R is greater or equal to the weight limit. Line34 $\sim$ 35 or 36 $\sim$ 37 will be executed:

Line 34 $\sim$ 35: To find the max total value in R instead of X, the procedure MaxValue is recursively called on R and the same weight limit W. We have already proved that the length of R will be $\frac{n}{2}$ (n is even) or $\frac{n-1}{2}$ (n is odd). In both cases, the length of R will be strictly less than $n = k+1$. So the length of R ($|R|$) must be an integer that belongs to [1,k].

So Under the assumption of strong induction: P($|R|$) must be true. In other words, MaxValue(R,W) in line 34 could return the correct max total value in R. Therefore, P(k+1) is true in this circumstance.

Line 36 $\sim$ 37: This is a special situation, if the sum of weights in R is exactly the weight limit, then there won't be any more space for additional items, and reducing any item will cause the total value is no longer the max total value. So we can directly return the sum of the values in R. P(k+1) is true in this circumstance.

If it is situation (2), then all of the items in R should be included in our final solution. And we

also need to find a subset from the union of L and pivotItem to get the maximum total value. We have already know the sum of values in R, so this problem will be reduced to find a subset in the union of L and pivotItem. i.e. recursively call MaxValue on updated L with updated weight limit (W-sumR). Line 31 $\sim$ 33 will be excuted:

Line 31 $\sim$ 33: We have already proved that the length of L ($|L|$) is $\frac{n}{2} - 1$ (n is even) or $\frac{n-1}{2}$ (n is odd), after appending pivot item at the end of L, the length become $\frac{n}{2}$ (n is even) or $\frac{n+1}{2}$ (n is odd).

1. n is even:

$n \geq 2, \frac{n}{2} < n$

2. n is odd:

$n \geq 3, \frac{n+1}{2} < n$

so $|L| < n = k + 1$

Under the assumption of strong induction: P($|L|$) must be true. In other words, MaxValue(L,W-sumR) in line 33 could return the correct max total value in L. Therefore, P(k+1) is true in this circumstance.

Overall, P(j),j=0,1,2,...,k $\rightarrow$ P(k+1)

By strong induction, P(n) is true for all $n \geq 0$

Proof of Time Complexity:

We use the recursion tree to organize the calculation. The recursive case of the implementation makes at most 1 recursive call at line 33 or line 35, so the resulting recursion tree is just a line, as shown below. The operations that are **local** to a given call are those in:

(1) Getting the length of X (line 1) and getting each item's value to form a valueArray in a for loop (Line 12 $\sim$ 14), $c \cdot n$ operations are executed here, so the time complexity in these lines is **O(n)**

(2) Select a pivot which is aroudly at the middle of the array by using SELECT algorithm introduced in the lecture notes. The time complexity of this algorithm has already been proved to be **O(n)** in the lecture notes (Line 15).

(3) Split the array X into 3 parts: L, pivotItem and R (line 20 $\sim$ 29). This for loop also consume linear time, **O(n)**.

(4) Those operations in the other lines spend **constant time**.

Above all, the local operations consume **linear time**.

To further analyze the structure of the recursion tree:

Between every 2 adjacent level, the number of the items is aroundly **halved**.

To prove this, we need some conclusions that have already been proved in the previous part **Proof of Correctness**.

From that part, we know the size of R is $\frac{n}{2}$ (n is even) or $\frac{n-1}{2}$ (n is odd), and the size of L (after appending pivotItem into L) is $\frac{n}{2}$ (n is even) or $\frac{n+1}{2}$ (n is odd).

Let n_{d+1} be the size of the next level, and $n_{d+1} = \alpha \cdot n_d$

When n_d is even, $n_{d+1} = \frac{n_d}{2} \Rightarrow \alpha = \frac{1}{2}$.

When n_d is odd, $n_{d+1} = \frac{n_d}{2} \pm \frac{1}{2} \leq \frac{n_d}{2} + \frac{1}{2} \Rightarrow \alpha \leq \frac{1}{2} + \frac{1}{2n_d}$

Because $n_d = 1$ is the base case, so we only analyze $n_d \geq 3$ here:

$n_d \geq 3 \Rightarrow \alpha \leq \frac{1}{2} + \frac{1}{2n_d} \leq \frac{5}{6}$

So the local operation number will be as shown in the figure:

Aggregate those terms in each level:

$T = c \cdot n + c \cdot \alpha_1 n + c \cdot \alpha_1\alpha_2 n + ... + c \cdot \alpha_1 ... \alpha_d n$

$\alpha \leq \frac{5}{6} \Rightarrow T \leq c \cdot ((\frac{5}{6})^0 n + (\frac{5}{6})^1 n + (\frac{5}{6})^2 n + ... + (\frac{5}{6})^d n) = cn \sum_0^d (\frac{5}{6})^d$

Because $\frac{5}{6} < 1$, this is a geometric sequence that converges to $c \cdot 6n$ when d approaches infinity.

So $T \leq c \cdot 6n, O(T) = O(n)$

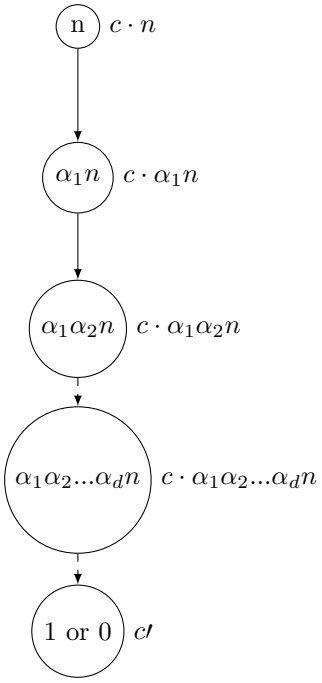

Figure 3: Depth: $log_2 n$