CS 5800 Solutions to Assignment-2

Ajeya Kempegowda

February 2019

1. Consider the following recursive algorithm for listing the $n-$bit strings of a twiddling list. If $n = 1$, the list is 0,1. If $n > 1$, first take a twiddling list of $(n1)$ bit strings, and place a 0 in front of each string. Then, take a second copy of the twiddling list of $(n1)$ bit strings, place a 1 in front of each string, reverse the order of the strings and place it after the first list. So, for example, for $n = 2$, the list is $00, 01, 11, 10$, and for $n = 3$, we get $000, 001, 011, 010, 110, 111, 101, 100$.

   - **Prove by induction on n that every n-bit string appears exactly once in the list generated by the algorithm.**

     Proof. Let $B(n)$ be the set of n-bit binary strings. From the definition $B(1) = 0, 1$ and for order $n = 3$, the over lines denotes the change that creates the next string.

     $$B(3) = 00\overline{0}, 0\overline{0}1, 01\overline{1}, \overline{0}10, 11\overline{0}, 1\overline{1}1, \overline{1}01, 100 \tag{1}$$

     We can express the above recursively as B(1) = 0, 1 for $n > 1$

     $$B(n) = 0 \cdot B(n-1), 1 \cdot flipped(B(n-1)) \tag{2}$$

     In other words, $B(n)$ can be created by alternately suffixing 0 then 1, and 1 then 0, to successive strings in $B(n-1)$. Let bit_i be the operation that complements the ith bit of a binary string. That is, $bit_i(b) = b_n \cdots b_{i+1}\overline{b_i}b_{i1} \cdots b_1$
     The proof is by induction on n; is $B(n = 1) = 0, 1$ is the base case for the induction. Further, the algorithm begins with an initial object and an ordered list of operations, and then repeatedly creates a new object by applying the first possible operation to the most recently created object. Therefore, we can interpret the generation of n-bit strings as a simple greedy algorithm.

     $$B(n = 1) = 0, 1 = Greedy_{Bit_n}(0^1) \tag{3}$$

     Inductively we can write this as,

$$Greedy_{Bit_{n-1}}(0^{n-1}) = b_1, b_2 \cdots, b_{2^{n-1}} = B(n-1) \tag{4}$$

In particular, $b_1 = 0^{n-1}$ and $b_2 = 0^{n-2}1$. The first four strings generated by $Greedy_{Bit_n}(0^n)$ are $0^n, 0^{n-1}1, 0^{n-2}11, 0^{n-2}10 = b1 \cdot 0, b1 \cdot 1, b2 \cdot 1, b2 \cdot 0$.

Suppose $Greedy_{Bit_n}(0^n)$ begins for some fixed $1 \leq i < 2^{k1}$

$$b_1 \cdot 0, b_1 \cdot 1, b_2 \cdot 1, b_2 \cdot 0, \cdots, b_{2k-1} \cdot 0, b_{2k-1} \cdot 1, b_{2k} \cdot 1, b_{2k} \cdot 0 \tag{5}$$

The algorithm cannot apply bit_n to the last string in (4) since $b_{2k} \cdot 0 = b_{2k} \cdot 1$ is the second-last string in (4). Therefore, the algorithm can only apply bit_j for some j ¡ n. Thus, the next string (if any) generated by the algorithm will end with 0. Since, $Greedy_{Bit_{n-1}}(0^{n-1})$ begins by generating $b_1 0, b_2, \cdots, b_{2k}, b_{2k+1}$ and we know that $Greedy_{Bit_n}(0^n)$ behaves similarly.

Thus $Greedy_{Bit_n}(0^n)$ follows $b_{2k} \cdot 0$ by generating $b_{2k+1} \cdot 0$. Furthermore, the string generated after $b_{2k+1} \cdot 0$ is $b_{2k+1} \cdot 1$ since bit_n is the highest priority. Therefore, (4) is true when n+1 replaces n. Further, (4) is true for $i = 2^{k-1}$. Therefore, we can conclude that $Greedy_{Bit_n}(0^n)$ and B(k) share the same recursive structure by (2) and (4), which completes the induction.

- **Express the worst-case time complexity of the algorithm above as a recurrence relation, and solve the recurrence to obtain a tight-bound on its worst-case time complexity**

  Proof The time complexity of the above algorithm can be expressed as a recurrence relation as

$$T(n) = 2^n + 2T(n-1) \tag{6}$$

  Further, when $n = 0, 1, ..\infty$

  $T(n)$ assumes the following values:

  $T(1) = 2^1 + 2T(0) = 2$,

  $T(22) = 2^2 + 2T(1) = 8$,

  $T(3) = 2^3 + 2T(2) = 24$,

  the above can be generalized as $\theta(n2^n)$

  Proof using Induction:

  Base case: When n=1; $1 * 2^1$=2

From the principles of Induction, let's assumes T(n) is true for some k $\forall$k>$2T(k) = 2T(k-1) + 2^k$(7)

Now, to prove the boundary condition that it hols true for k+1;

$$T(k+1) = 2T(k+1-1) + 2^{k+1} \tag{8}$$

$$2T(k) + 2^{k+1} \tag{9}$$

$$2(2T(k-1) + 2^k) + 2^{k+1} \tag{10}$$

$$4T(k-1) + 2^{k+2} \tag{11}$$

$$4(T(k-1) + 2^k) \tag{12}$$

which resembles our original T(n). Hence we complete Induction.

Recurrences Drawing the Skyline

You are given the exact locations and shapes of several rectangular buildings in the city, in two dimensions. We assume that the bottoms of these buildings lie on a straight line. Building B_i is represented by the triple (L_i, H_i, R_i), where L_i and R_i denote the left and right coordinates of the building, and H_i denotes its height. For example, consider the following input:
(1,11, 5), (2, 6, 7), (3, 13, 9), (12, 7, 16), (14,3,25), (19,16,22), (23, 13, 29), and (24,4,28)

A *skyline* is a list of x coordinates and the heights connecting them arranged in order from left to right. The skyline for the example above is represented as: (1, 11, 3, 13, 9, 0, 12, 7, 16, 3, 19,16, 22, 3, 23, 13, 29, 0).

- **Design an $O(n \log_2 n)$ algorithm to determine the two-dimensional skyline of a set of n input buildings, eliminating hidden lines to produce the skyline**.

  Answer: The solution can be implemented using divide and conquer algorithm. Divide the the set of building into set of $n/2$ buildings and recursively find the skyline and finally merge them in linear time.

  Looking at the required solution, it's intuitive to think the solution revolves around the left co-ordinate and the height. We can build a logic around the L, H co-ordinates. Let A, A' be two skylines to be merged and R is the resultant output skyline. Let A_x, A_h be two arrays obtained by splitting A's L and H co-ordinates respectively. Compare the values in A, A', the current height is set to the Max(current(A_h), previous(A'_h))

Algorithm 1 GetSkyline

```
procedure GetSkyline(A[1 ··· N])
    if size(A) = 1 then
        return A
    end if
    A_x, A_h = Split[A]
    A_x = GetSkyline[A_x]
    A_h = GetSkyline[A_h]
    result = SkylineMerge[A_x, A_h]
    return result
end procedure
```

Algorithm 2 Split function

```
procedure SPLIT(A[1 ··· N])
    A_x = A[x − coordinate]
    A_h = A[height − coordinate]
    return A_x, A_h
end procedure
```

Algorithm 3 SkylineMerge

```
procedure SKYLINEMERGE(A, A′)
    current = 1
    prev =1
    i =1
    while current ≤ length(A) or prev ≤ length(A′) do
        x ← min(A_x[current], A′_x[prev])
        if A_x[current] ¡ A′_x[prev] then
            max ← max(A_x[current], A′_x[prev − 1])
            current++
        else
            if A_x[current] ¿ A′_x[prev] then
                max ← max(A_h[current − 1], A′_h[prev])
                prev++
            else
                max ← max(A_h[current], A′_h[prev])
                prev++
                current++
                if max ≠ res_h[i − 1] then
                    res_x[i] ← x
                    res_h[i] ← max
                    i++
                end if
            end if
        end if
    end while
end procedure
```

- **Prove the correctness and time complexity of your algorithm.**

  The correctness of the algorithm can be shown in way that after every iteration of the loop $res_h[i-1]$ contains height and whose final value is not determined. Further, current and prev has the least values initially confined by the constraints that x-coordinates A_x, A'_x are both greater than or equal to $res_h[i-1]$. Finally, A_x, A'_x both represent x-coordinates at which the height of their respective subskylines change. $res_h[i-1]$ will be retained until $x \geq min(A_x[current], A'_x[prev])$ and the value changes when this condition is met. if $A_x[current], A'_x[prev]$ then it's clear that new height should be the max value of $A_x[current], A'_x[prev]$.

  Time complexity:

  Splitting up of the input array into separate arrays of heights and x-coordinates taken $O(n)$. The merge sort is proven to have a complexity of $O(nlogn)$. Further, the merging of the skylines as aforementioned happens in linear time - $O(n)$. Therefore, the overall time complexity proves to be $O(nlogn)$

.

Finding the Majority Element An array $A[1..n]$ is said to have a *majority element* if more than half of its entries are the same. We would like to determine whether a given array A has a majority element, and if so, find the element. We assume that the elements of the array are not necessarily from some ordered domain like the integers, so there can be no comparisons of the form is $A[i] > A[j]$; only questions of the form "is $A[i] = A[j]$ can be answered.

- **Design an algorithm to find the majority element of a set of n elements, if it exists, in $\theta(nlogn)$ time. Prove the correctness of your algorithm**

  Proof. Solving the problem in $\theta(nlogn)$ time. Divide array A into two subsets, A_l and A_r. Then, A has a majority element m $\Leftrightarrow$ m appears more than n/2 times in A ***or*** n appears more than n/4 times in either A_l or A_r (or both); where n is the length of the array. After perform a linear time equality operation to decide whether it is possible to find a majority element.

  The correctness of this algorithm simply follows from the fact that if a majority exists in the array, it must exist in either the left or right halves as well. We then collect these majorities and check the entire array to verify that they are in fact majorities. Further, when the size of the array is 1, then it is the majority element. Therefore, the initial condition holds

- **Prove the worst-case time complexity of your algorithm using induction or other means**

  Count function computes the number of times an element appear in the input array $A[1, 2 \cdots N]$. Therefore two function calls of Count would be $O(n)$. Count is the linear time equality operation.

Algorithm 4 Majority element algorithm

```
1: procedure MAJORITY(A[1...n])
2:     if n = 1 then                                  ▷ This is the majority element
3:         return A[1]
4:     end if
5:     j=n/2
6:     l_sub = Majority(A[1 ··· j])                   ▷ T(n) = 2T(n/2)
7:     r_sub = Majority(A[j + 1 ··· n])
8:     if l_sub = r_sub then
9:         return l_sub
10:    end if
11:    l_count = Count(A[1...n], l_sub)               ▷ To get the frequency of the element
12:    r_count = Count(A[1...n], r_sub)
13:    if l_count > j+1 then
14:        return l_sub
15:    else if r_count > j+1 then
16:        return r_sub
17:    else
18:        return No Majority element
19:    end if
20: end procedure
```

Algorithm 5 Count the occurrences/frequency

```
1: procedure COUNT(A[[1...n]], input)
2:     counter = 0
3:     for element in A do                            ▷ O(n)
4:         if input = element then
5:             counter++
6:         end if
7:     end for
8:     return counter
9: end procedure
```

Further, the input is split and recursively called using Majority function. Therefore, the time complexity would be of the form $O(nlogn)$

The worst time complexity would be of the form $O(n) + O(nlogn) = O(nlogn)$

- **Can you find the majority element in linear time? If so, describe your algorithm and establish its time complexity**

Algorithm 6 Majority in Linear

```
procedure LinearMajority(A[[1...n]])
    if n = 2 then
        if a[1] = a[2] then
            return a[1]
        else
            return No majority element
        end if
    end if
    for i = 1 to n do
        if a[i] = a[i+1] then
            temp array ← append a[i] to temp array
        end if
    end for
    return LinearMajority(temp array)
end procedure
```

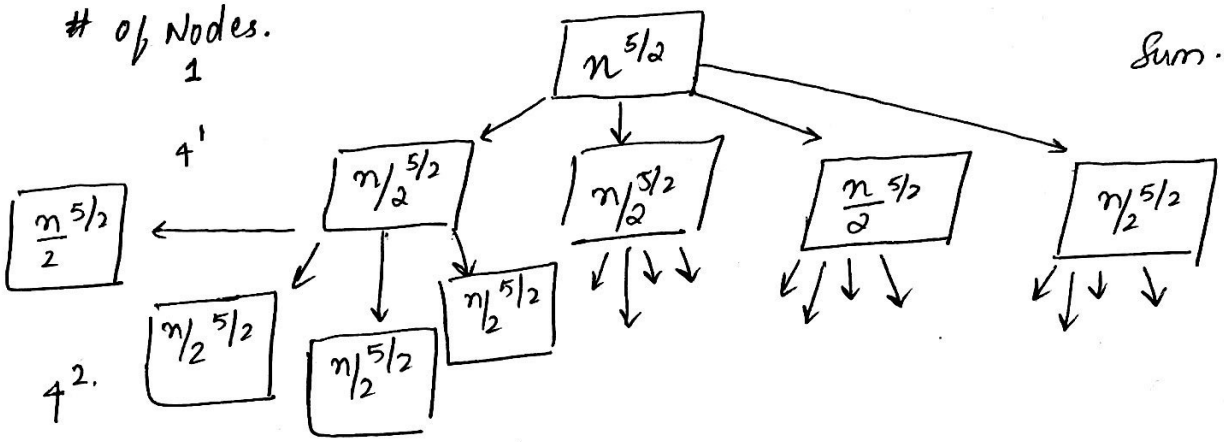
Recurrence relation for the algorithm is given as follows

$$T(n) = 2T(n/2) + O(n) \tag{13}$$

Complexity of the function LinearMajority is O(n).

2(a) $T(n) = 4T(n/2) + n^{5/2}$.

using the concept of Recurrence Tree



$(n/2^k)^{5/2} \approx T(1)$.

$\Rightarrow \frac{n}{2^k} = 1$.

$\Leftrightarrow k = \log_2 n$.

$\therefore T(n) = 4^{\log_2 n} T(1) + \sum_{k}^{(k-1)} 4^k (n/2^k)^{5/2}$.

solving by GP series

$S(n) = \frac{1-r^n}{1-r}$.

$\therefore n^{5/2} \sum_{i}^{(k-1)} 4^i (1/2^k)^{5/2}$

$\Leftrightarrow n^{5/2} \left[\frac{1-4(1/2)^{5/2k}}{1-4(1/2)^{5/2}} \right]$

$= n^{5/2} \left[\frac{1-2^{2\log_2 n} \cdot 2^{(-5/2)\log_2 n}}{1-2^2(2^{-5/2})} \right]$

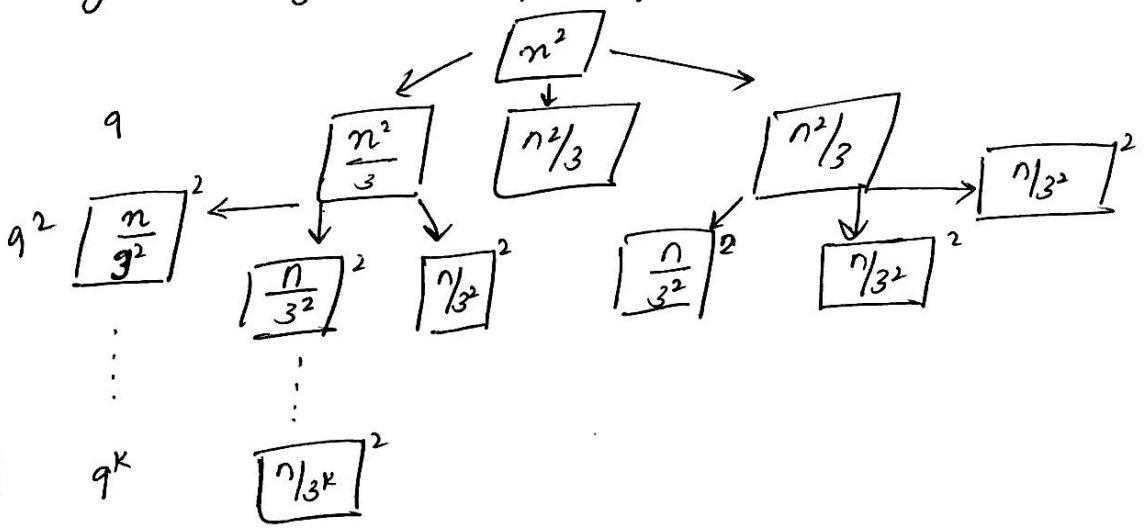
$= n^{5/2} \left[\frac{1-n^2 \cdot n^{-5/2}}{1-1/\sqrt{2}} \right]$

$= n^{5/2} \left(\frac{\sqrt{2}(\sqrt{n}-1)}{\sqrt{n}(\sqrt{2}-1)} \right)$

$= n^2 + \frac{(\sqrt{2}n^{5/2} - \sqrt{2}n^2)}{\sqrt{2}-1} = \Theta(n^{5/2})$

2(b) $T(n) = 9T(n/3) + n^2$.

Again, using the concepts of Recurrence tree.



sum : n^2

$9(n/3)^2 = n^2$

$9^2(n/3^2)^2 = n^2$

$9^k(n/3^k)^2 = n^2$

$\Leftrightarrow \quad n/3^k = T(1)$

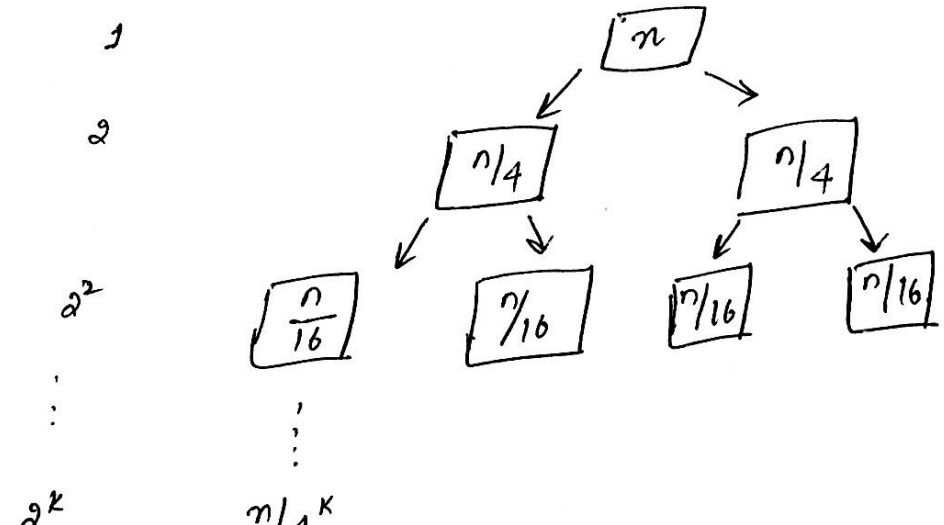
$n/3^k = 1 \quad \Leftrightarrow \quad k = \log_3 n.$

$T(n) = 9^{\log_3 n} T(1) + \sum_{i}^{(k-1)} n^2$

$= 9^{\log_3 n} + n^2 \cdot k \quad \Rightarrow \quad n^2 + n^2 \log_3 n.$

$= \theta(n^2 \log_3 n).$

2(c) $T(n) = 2T(n/4) + n.$



sum : n

$2(n/4) = n/2$

$4(n/16) = n/4.$

$2^k(n/4^k) = n/2^k.$

$$\frac{n}{4^k} = T(1) = 1 \quad \Rightarrow \quad n = \log_4 n.$$

$$T(n) = 2^{\log_4 n} T(1) + n \sum_{i=0}^{k-1} (2^{-k})$$

clearly $\sum 2^{-k}$ is a GP ; sum $= a\left(\frac{1-r^n}{1-r}\right)$.

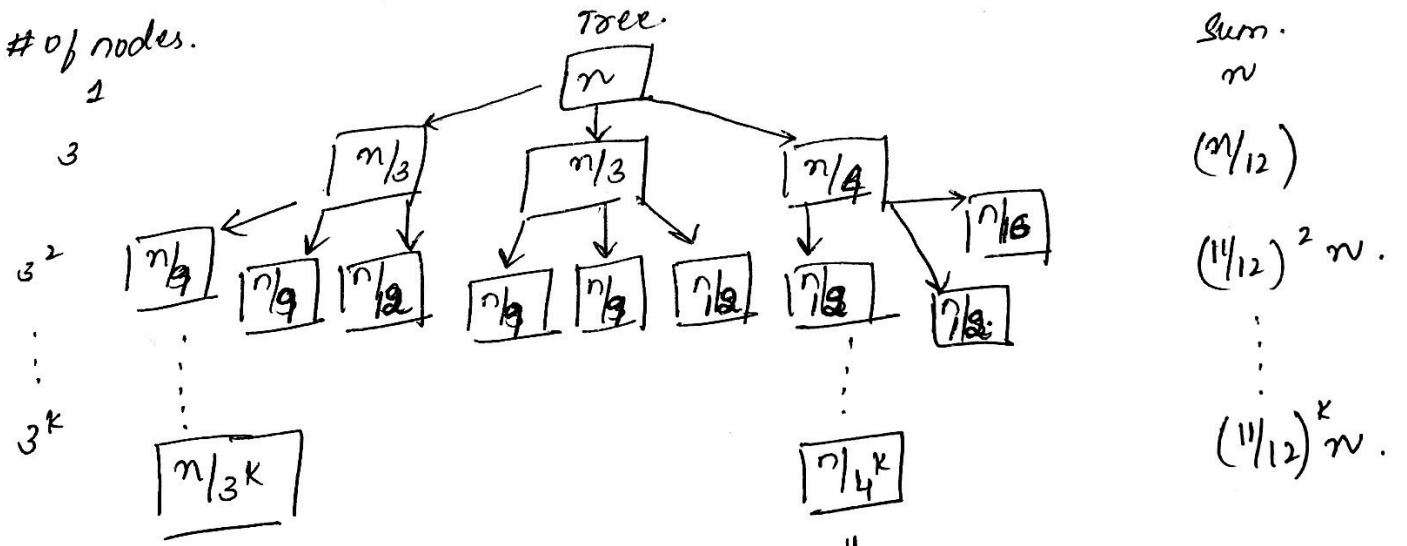
$$T(n) = \sqrt{n} + n\left(\frac{1-2^{-k}}{1-2^{-1}}\right)$$

$$= \sqrt{n} + n\cdot\left(\frac{1-1/2\log_4 n}{1-1/2}\right)$$

$$= \sqrt{n} + 2n\,(1-1/\sqrt{n})$$

$$= \sqrt{n} + 2\sqrt{n}\,(\sqrt{n}-1) \Rightarrow 2n-\sqrt{n} = \theta(n).$$

$$2(d)\,T(n) = 2T(n/3) + T(n/4) + n.$$



$$\frac{n}{3^k} = T(1)$$

$$\Rightarrow k = \log_3 n$$

$$\frac{n}{4^k} = T(1).$$

$$k = \log_4 n.$$

$$T(n) = 2^k T(1) + T(1) + n \cdot \sum_{i=0}^{(k-1)} (11/12)^k$$

$2^{\log_3 n} + 1 + n \cdot \left(1 + 11/12 + (11/12)^2 + \ldots \infty\right)$.

$2^{\log_3 n} + 1 + n \left(1 + \langle \text{close to zero} \rangle\right)$

↳ As the ratio b/w the terms in the GP is very less.

$2^{\log_3 n} + 1 + n$

$= \Theta(n)$