Divide and Conquer

What is divide and Conquer?

- It is a class of algorithmic techniques in which input is broken into several parts, solves the problem in each part recursively and then combines the solutions of these sub problems to get the solution of the original problem.
- Analysing the divide and conquer algorithm requires solving recurrence relations
- We will start with simple algorithms which we have already studied.

The Merge sort algorithm

- It sorts a list of numbers by first dividing them into two haves
- Each half is sorted separately by recursion
- Both results are combined to get an overall solution

Note: In recursion, a base case is required

Base case for merge sort: We can take the base case as two elements where two elements are compared and put in order or only one element also.

We can also take base case as one element.

Merge sort: Divide and Conquer

- **Divide** the n-element sequence to be sorted into two subsequences of n/2 element each
- **Conquer:** Sort the two subsequences recursively **using merge sort**
- **Combine:** merge the two sorted subsequences to produce the sorted answer

Merge Sort

```
Function Mergesort(A,left,right,B) // sort from left to right into B
    if(right-left ==1)
                B[0] = A[left]
     if (right-left >1)
                mid = (left+right)/2
Mergesort(A,left,mid,L)
Mergesort(A,mid,right,R)
Merge(L, mid-left,R,right-mid,B)
```

Merge operation

Function Merge(A,m,B,n,C) //merge A[0…m-1] and B[0..n-1] into C[0…m+n-1]

i=0; j=0; k=0;

While(k<m+n)

if(j==n or A[i] <=B[j])

  C[k] = A[i] i++,k++;

 if(i==m or A[i] >b[j]) c[k] = B[j]; j++, k++;

Analysis of Merge Sort

$T(1) = 1$

$T(n) = 2T(n/2) + n$

$= 2[2T(n/4) + n/2] + n$

$= 2^2 T(n/2^2) + 2n$

……

$= 2^j T(n/2^j) + jn =$

it will reach to base case when $n/2^j = 1$, it will be 1 when $j = \log_2 n$

So $T(n) = n \log_2 n$

Building recurrence relation for Merge sort

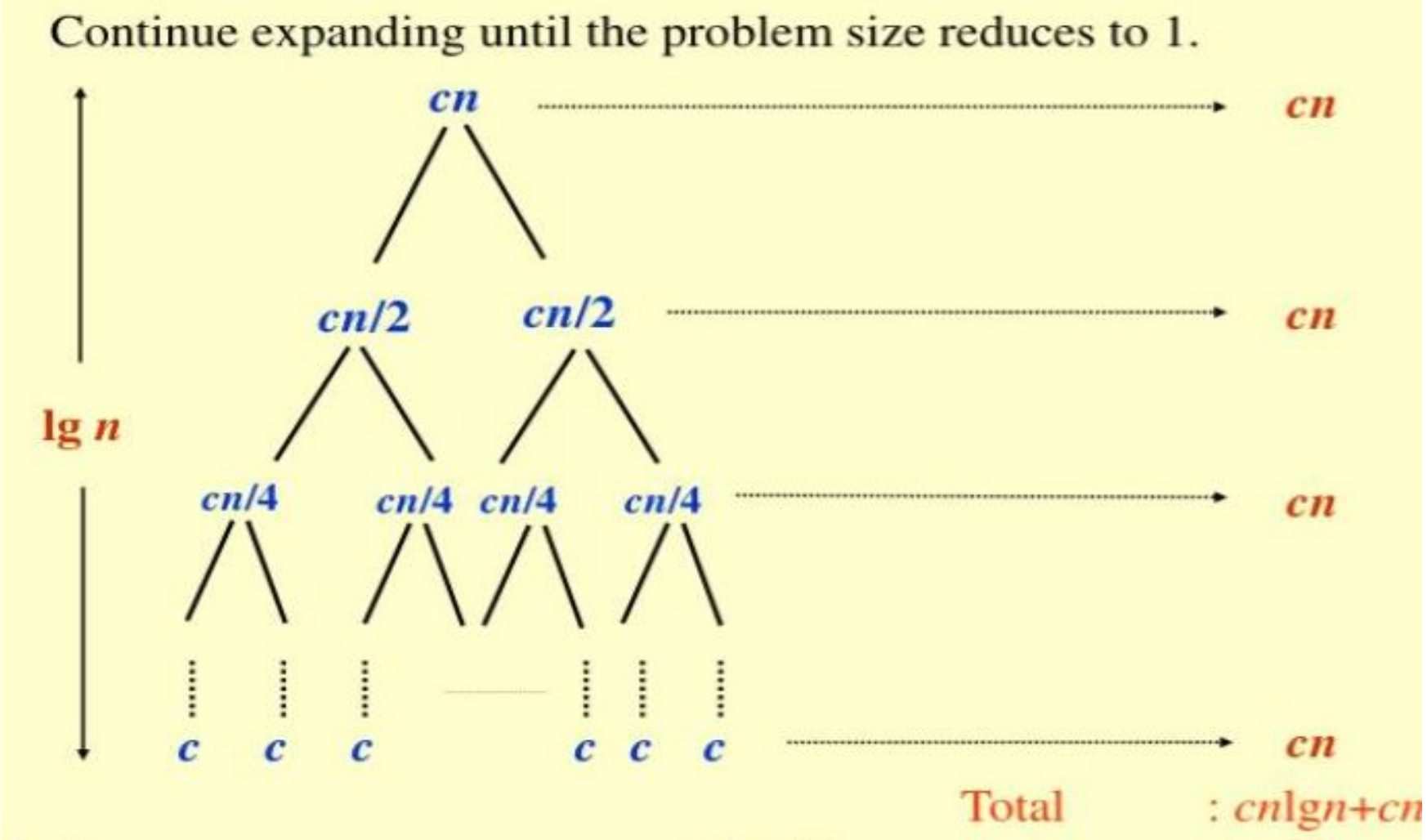
- Suppose T(n) denotes the worst case complexity for size n
- For simplicity let us assume n is even
- Assume, algorithm takes O(n) time to divide the input into two parts
- Now we have two problems of size n/2
- Worst case complexity of n/2 is T(n/2), so total time = 2 T(n/2)
- If it takes O(n) time to combine two sorted arrays then we can write that

$T(n) <= 2T(n/2) + cn$ for some constant where $n>=2$ and $T(2) <=c$

Approaches to solving recurrence relations

1. Unroll the recurrence, identify the pattern and then sum up over all level until it reaches to base case.
2. Start with a guess solution, substitute into the relation and check whether it satisfies or not

Unrolling…


Continue expanding until the problem size reduces to 1.
cn
cn
cn/2
cn/2
cn
lg n
cn/4
cn/4
cn/4
cn/4
cn
c
c
c
c
c
c
cn
Total
: cnlgn+cn

Start with a guess solution

- Suppose we assume with a solution n log2n
- We can prove that n log2n satisfies $T(n) <= 2T(n/2) +cn$

Quick sort

- What is the concept behind Quicksort?
- Does it also use divide and conquer strategy?
- How is it different from Mergesort?
- What is the complexity of Quicksort?

Example

24 60 85 1 20 15 11 100 75 (array X)

- N = 9, lb = 0, ub =8, down=0, up =8, pivot element a = 24
- We will start incrementing down and stop at down =1 because 60 >24
- We will start decrementing up and stop at up =6 because 11<=24
- Interchange X[1] by X[6]– 24 11 85 1 20 15 60 100 75
- incrementing down and stop at down =2 because 85 >24
- decrementing up and stop at up = 5 because 15<=24
- Interchange X[2] by X[5]– 24 11 15 1 20 85 60 100 75

Example

24 60 85 1 20 15 11 100 75 (array X) Original array

24 11 15 1 20 85 60 100 75 (now after making two interchanges) (down =2; up= 5)

- incrementing down and stop at down =5 because 85 >24
- decrementing up and stop at up = 4 because 20<=24
- Since up<=down, interchange x[up] with x[lb]:

  20 11 15 1 24 **85 60 100 75**

- Above array is partitioned in two subarrays X[0] to X[3] and X[5] to X[8]
- Now we will apply partition algorithm on these two arrays taking ***X[0] pivot element for first array*** and taking ***X[5] pivot element for second array***

Applying partition on subarrays

- Array1: ***20 11 15 1*** (lb =0 ub = 3, a= 20, down =0 up = 3)
- Array2: ***85 60 100 75*** (lb =5 ub = 8, a =85, down =5 up = 8)
- Array1: 20 11 15 1 (lb =0 ub = 3, a= 20, down =0 up = 3)
- Incrementing down… … = 3
- Decrementing up.. , will not be decremented. So up = 3
- Hence x[up] will be exchanged with x[lb] and we will have:
- [1 11 15] [20]
- Array2: 85 60 100 75 (lb =5 ub = 8, a =85, down =5 up = 8)
- Incrementing down… … = 7 and up will not be decremented. So up = 8
- Interchanging x[down] and x[up]: 85 60 75 100, next iteration down =8, up=7, interchanging x[up] and x[lb]: [75 60] 85 [100]

Partition function

```
Void partition(int x[],int lb,int ub, int *pj)
{ int a, down, up, temp;
 a = x[lb]; up =ub;down = lb;
While(down<up) {while(x[down] <=a && down <ub) down++;
                        while(x[up] >a) up--;
                        if(down <up)   swap(x[down],x[up])
                      }
                  x[lb] = x[up]; x[up] =a; *pj = up;
}
```

Quick sort algorithm

```
 if(lb>=ub) return;
Partition(x,lb,ub,j);
Quick(x, lb, j-1);
Quick(x,j+1,ub);
```

Quicksort Sorting efficiency..

- We know that sorting efficiency depends upon the arrangement of input data and input data may be already sorted, nearly sorted, completely random, in reverse order or nearly reverse.
- Examine Quick sort:
- **How many passes of partition will be there if:**

1. The arrangement of data is such that **pivot element** partitions the array into two almost equal arrays
2. Pivot element does not partition equally or almost equally

Quick sort complexity

- ***Best case*** (Pivot divides the array into two equal subarrays)
- $O(n\log_2 n)$

$n + 2*n/2 \quad +4 *n/4 \quad +\ldots\ldots\ldots n*(n/n)$

$= n +n + \ldots \; n$ (m terms) $= n\log_2 n$

$T(n) = 2T(n/2) +n$

- Worst case (Data is sorted or in reverse order)
- $O(n^2)$

$T(n) = T(n-1) + n$

$=n + (n-1) + (n-2) +\ldots. 1 = n(n-1)/2$

- Average case $O(n\log_e n)$ (Random data)
- Requires more space due to recursion and depends upon the number of nested recursive calls and the size of stack.

Few ways to choose pivot elements…

- Several choices have been found to improve the efficiency of quicksort by creating nearly balanced subfiles
- First Technique: Take the median of first, last and middle element i.e median of x[lb], x[ub] and x[(lb+ub)/2] this median of three value is generally closer to the median of the file/subfile being partitioned
- Second Technique: Take first pivot as median and then subsequently use mean as pivot of each subfile.
- Other techniques: a. Use the middle element of file as pivot element

  b. Choose a random element

Finding Max,Min using Divide and Conquer

- Divide the list into small groups.
- Then find max and min of each group.
- The max/min of result must be one of maxs and mins of the groups.
- e.g. A = [5, 7, 1, 4, 10, 6]
- A1 = [5, 7, 1], max(A1) = 7, min(A1) = 1
- A2 = [4, 10, 6], max(A2) = 10, min(A2) = 4
- So the min and max of A is min(1, 4) and max(7, 10)
- i.e 1 and 10.

Finding Max and Min of an array (Simple algo)

- Algorithm MaxMin(A : list, n, max, min : integer)

```
{ max = A[1]; min = A[1];
for(i = 2;i <= n;i + +)
{ if(A[i] > max)max = A[i];
 if(A[i] < min)min = A[i]; }
 }
```

Time complexity: T(n) = 2(n − 1)

Example

- e.g. A = [5, 7, 1, 4, 10, 6]
- A1 = [5, 7, 1], max(A1) = 7, min(A1) = 1
- A2 = [4, 10, 6], max(A2) = 10, min(A2) = 4
- So the min and max of A is min(1, 4) and max(7, 10)
- i.e 1 and 10.

Divide and Conquer code

- Algorithm DCMaxMin(list A, i, j, fmax, fmin)

(i,j are parameters set as $1 \leq i \leq j \leq n$, the algorithm)

```
{  if(i == j){fmax = A[i]; fmin = A[i]}
if(i == j − 1){ if(A[i] > A[j]){fmin = A[j]; fmax = A[i]}
else{fmin = A[i]; fmax = A[j]; }
else {mid = (i + j)/2;
 DCMaxMin(listA : i, mid, gmax, gmin)
 DCMaxM in(listA : mid + 1, j, hmax, hmin)
fmax = max(gmax, hmax);
 fmin = min(gmin, hmin); } }
```

Complexity of MaxMin

Simple method : O(n)

Divide and Conquer: $T(n) = 2T(n/2) + 2$

Is this Recurrence relation correct?

What is the complexity of this?

- End of Second lecture of Divide and Conquer

Matrix multiplication

- What is the complexity of multiplying two matrices of size nxn

```
for(i=0;n-1;i++)
   for(j=0;n-1;j++)
      c[i,j] = 0;
      for(k=0;n-1;k++)
       c[i,j] = c[i,j] + a[i,k]* b[k,j];
return c;
```

Using Divide and Conquer

- Let us partition each of A, B and C into n/2 x n/2 matrices
- Each matrix will have four partitions

$$A = \begin{bmatrix} A11 & A12 \\ A21 & A22 \end{bmatrix}$$

$$B = \begin{bmatrix} B11 & B12 \\ B21 & B22 \end{bmatrix}$$

$$C = \begin{bmatrix} C11 & C12 \\ C21 & C22 \end{bmatrix}$$

Cont..

Since C = A.B

$C_{11} = A_{11}.B_{11} + A_{12}.B_{21}$

$C_{12} = A_{11}.B_{12} + A_{12}.B_{22}$

$C_{21} = A_{21}.B_{11} + A_{22}.B_{21}$

$C_{22} = A_{21}.B_{12} + A_{22}.B_{22}$

Each of these four equations specify two multiplications of n/2 x n/2 matrices and the addition of n/2 x n/2 products

Square matrix multiplication using divide and conquer

- If n==1: $c_{11} = a_{1\,1} * b_{1\,1}$

else:

Partition A,B and C as in equations….

$C_{1\,1}$ = square-matrix-multiply-recursive($A_{1\,1}$, $B_{1\,1}$)

+ square-matrix-multiply-recursive($A_{1\,2}$, $B_{2\,1}$)

$C_{1\,2}$ = square-matrix-multiply-recursive($A_{1\,1}$, $B_{1\,2}$)

+ square-matrix-multiply-recursive($A_{1\,2}$, $B_{2\,12}$)

……. Return C

Recurrence relation

$T(1) = 1$ or $\theta(1)$

$T(n) = 8T(n/2) + 4 *$ time to add $n^2/4$ entries + partitioning time

$= 8T(n/2) + \theta(n^2) + \theta(1)$

$= 8T(n/2) + cn^2$

$T(n/2) = 8T(n/4) + c(n/2)^2$

$= 8T(n/4) + cn^2/2^2$

$T(n/4) = 8T(n/8) + c(n/4)^2$

$= 8T(n/8) + cn^2/4^2$

$T(n) = 8T(n/2) + cn^2$

$= 8(8T(n/4) + cn^2/2^2) + cn^2$

$= 8^2 T(n/4) + 2cn^2 + cn^2$

$= 8^2 (8T(n/8) + cn^2/4^2) + 2cn^2 + cn^2$

$= 8^3 T(n/8) + 8^2 cn^2/4^2 + 2cn^2 + cn^2$

$= 8^3 T(n/8) + 2^2 cn^2 + 2cn^2 + cn^2$

At $n = 2^k$, $T(n) = 8^k T(1) + 2^{k-1} cn^2 + \ldots. 2^2 cn^2 + 2cn^2 + cn^2$

$T(n) = 8^k + \theta(n^2)$

$T(n) = O(8^k) = O(n^3)$

Strassen method

- The Strassen algorithm performs 7 multiplications of n/2 x n/2 matrices but it adds more matrix additions but only a constant number of additions.
- It has following steps:

1. Divide the input matrices A and B and the output matrix into n/2 x n/2 submatrices
2. Define 7 matrices M1….M7 as shown in the next slide
3. Using these 7 matrices we can compute C_{11} ….. C_{22}

Matrices M_1 .....M_7

$$M_1 = (A_{11} + A_{22})(B_{11} + B_{22})$$
$$M_2 = (A_{21} + A_{22})B_{11}$$
$$M_3 = A_{11}(B_{12} - B_{22})$$
$$M_4 = A_{22}(B_{21} - B_{11})$$
$$M_5 = (A_{11} + A_{12})B_{22}$$
$$M_6 = (A_{21} - A_{11})(B_{11} + B_{12})$$
$$M_7 = (A_{12} - A_{22})(B_{21} + B_{22})$$

Observations…..

1. While calculating M_1 …..M_7 we observe that there are 7 matrix multiplications and 10 matrix additions of size n/2 x n/2

cont...

$$C_{11} = M_1 + M_4 - M_5 + M_7$$

$$C_{12} = M_3 + M_5$$

$$C_{21} = M_2 + M_4$$

$$C_{22} = M_1 - M_2 + M3 + M_6$$

$T(n) = 7T(n/2) + C(\text{no. of additions}) * \text{time to add } n^2/4 \text{ entries} + \text{partitioning time}$

$T(n) = 7T(n/2) + cn^2 \quad \text{if } n > 1$

$= \theta(1) \text{ if } n = 1$

$T(n) = O(7^k) = O(n^{2.81})$

Solving Recurrence relations

Different Techniques to solve recurrence relations

SUBSTITUTION METHOD: The substitution method comprises of 3 steps:

- Guess the form of the solution
- Verify by induction
- Solve for constants
- The guessed solution is substituted for the function. This method is powerful, but we have to guess the form of the answer in order to apply it.

e.g. Let us take : Recurrence equation: $T(n)=4T(n/2)+n$

Cont..

Step 1: Guessing: $T(n)=4T(n/2)+n$

How to guess? $T(n)=4T(n/2)$ gives a hint…

($F(n)=4f(n/2)$; $F(2n)=4f(n)$; $F(n)=n^2$ So, $T(n)$ is atleast order of n^2)

Guess: $T(n)=O(n^3)$

Step 2: verify the induction

Assume $T(k)<=ck^3$;

$T(n)=4T(n/2)+n$ => it is $<=4c(n/2)^3+n$ $<=cn^3/2+n$ $<=cn^3-(cn^3/2-n)$
$T(n)<=cn^3$ as $(cn^3/2-n)$ is always positive

So what we assumed was true. $T(n)=O(n^3)$

Cont..

- **Step 3**: solve for constants
- $Cn^3 /2-n>=0$ for $n>=1$ $c>=2$

- Now suppose we guess that $T(n)=O(n^2)$ which is tight upper bound
- Assume, $T(k)<=ck^2$ so prove $T(n)<=cn^2$
- $T(n)=4T(n/2)+n$; after substituting ..$4c(n/2)^2 +n = cn^2 +n$
- So, T(n) will never be less than cn^2 . But if we take the assumption of $T(k)=c_1 k^2 -c_2k$, then we can find that $T(n) = O(n^2)$

ITERATIVE METHOD

2. Iterative method:

- $T(n)=2T(n/2)+n$

$\Rightarrow 2[2T(n/4) + n/2]+n$

$\Rightarrow 2^2\ T(n/4)+n+n \Rightarrow 2^2\ [2T(n/8)+ n/4]+2n \Rightarrow$

$\Rightarrow$ After k iterations ,$T(n)=2^k\ T(n/2^k)+kn$----------(1)

$\Rightarrow$Sub problem size is 1 after $n/2^k =1 \Rightarrow k=\log_2 n$

$\Rightarrow$So,after $\log_2 n$ iterations ,the sub-problem size will be 1.

$\Rightarrow$ So,when $k=\log_2 n$ is put in equation 1 we get : $T(n)=nT(1)+n\log n = nc+n\log n$ taking $c=T(1)$, hence $O(n\log_2 n)$

RECURSION TREE METHOD

3.BY RECURSION TREE METHOD:

- In a recursion tree, each node represents the cost of a single sub-problem
- We sum the cost within each level of the tree to obtain a set of per level cost, and
- then we sum all the per level cost to determine the total cost of all levels of recursion .
- It is good for generating guesses for the substitution method.
- The recursion-tree method can be unreliable, just like any method that uses ellipses (…).
- The recursion-tree method promotes intuition, however.

Example of recursion tree

Solve $T(n) = T(n/4) + T(n/2) + n^2$:

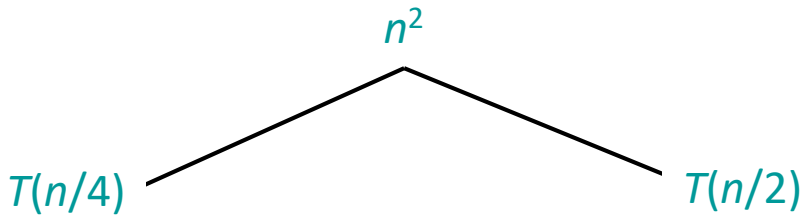
Example of recursion tree

Solve $T(n) = T(n/4) + T(n/2) + n^2$:

$T(n)$

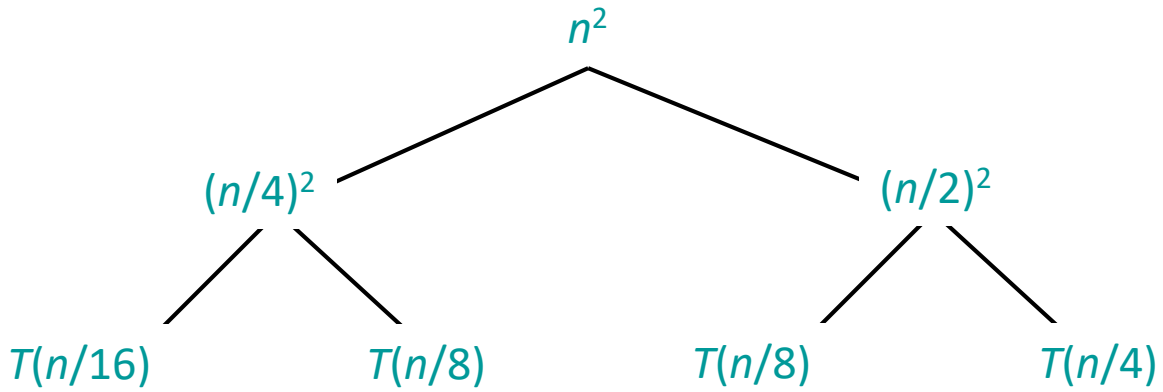
Example of recursion tree

Solve $T(n) = T(n/4) + T(n/2) + n^2$:



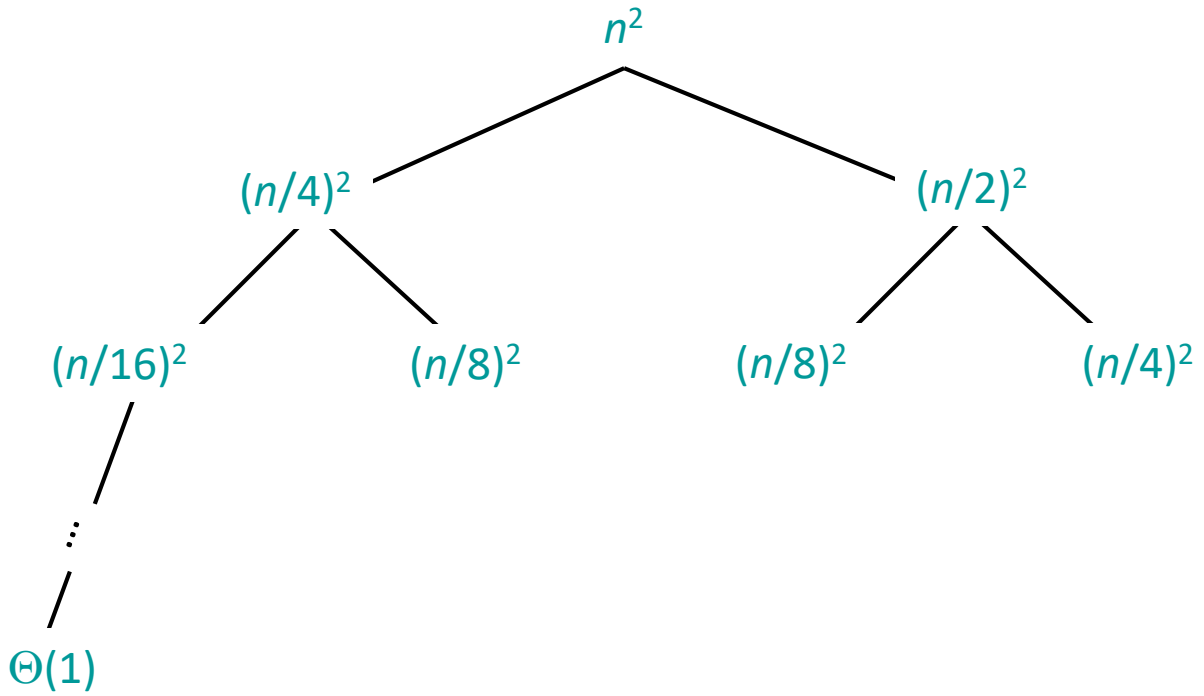
Example of recursion tree

Solve $T(n) = T(n/4) + T(n/2) + n^2$:



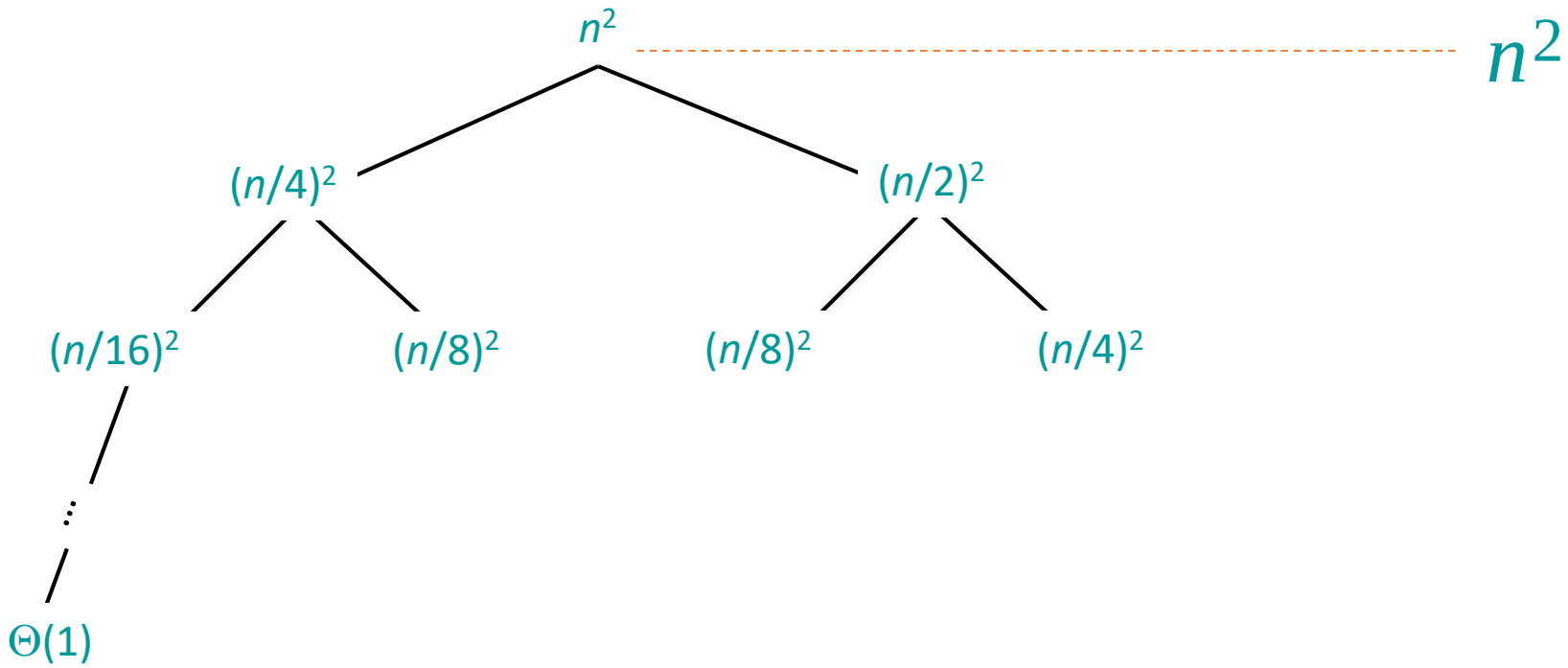
Example of recursion tree

Solve $T(n) = T(n/4) + T(n/2) + n^2$:



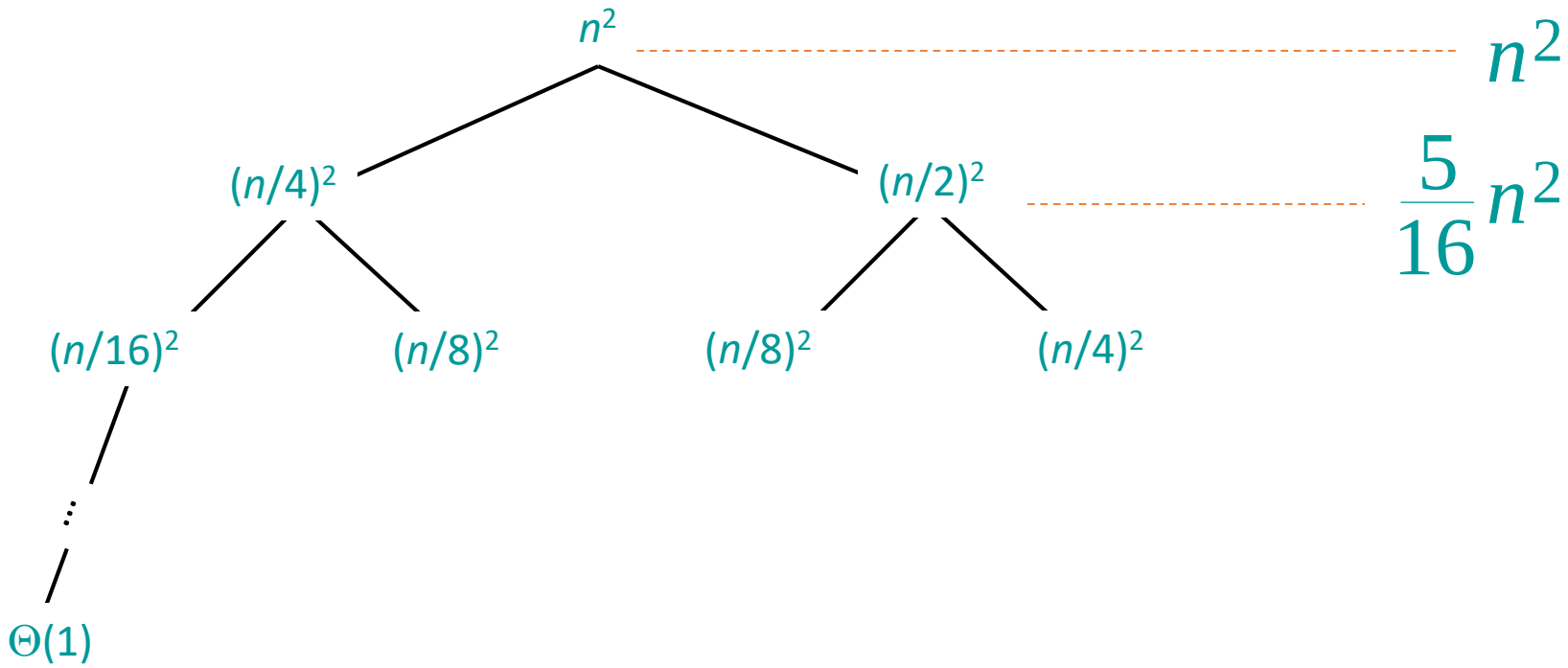
Example of recursion tree

Solve $T(n) = T(n/4) + T(n/2) + n^2$:



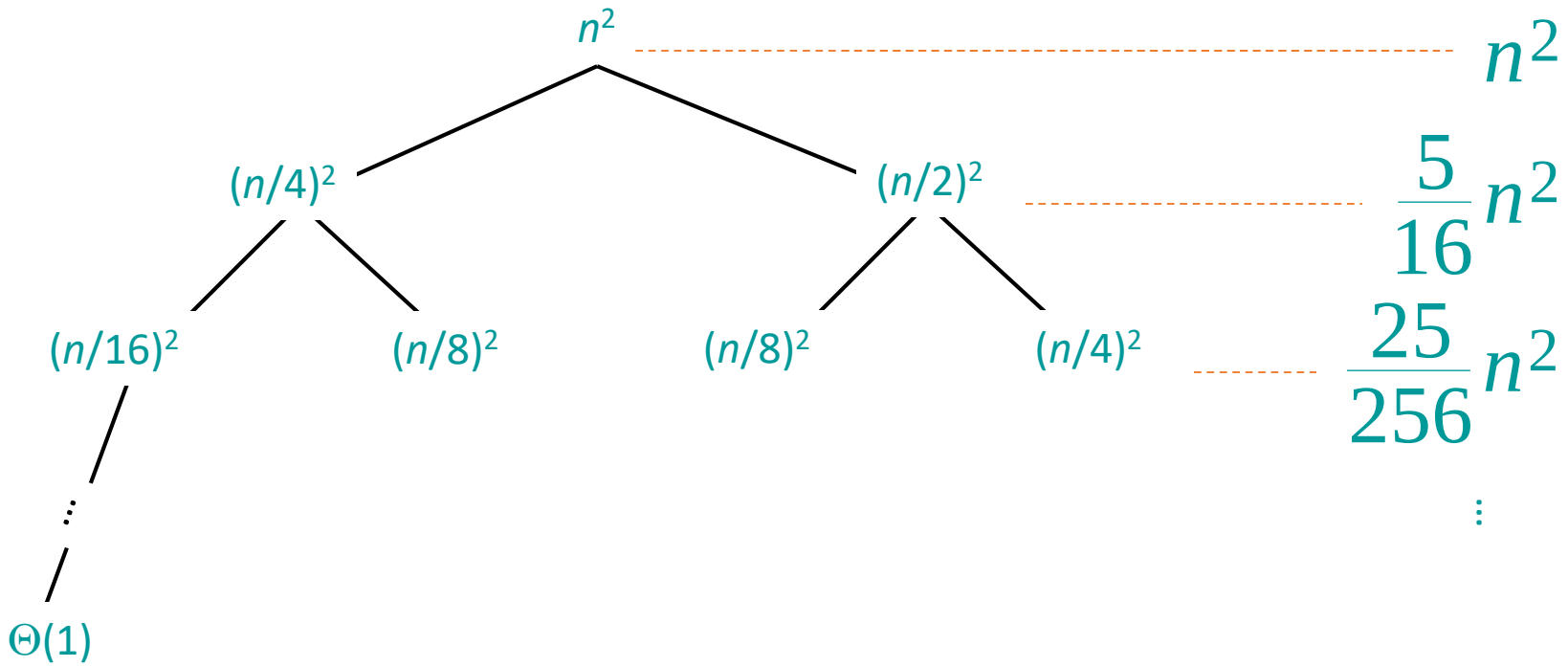
Example of recursion tree

Solve $T(n) = T(n/4) + T(n/2) + n^2$:



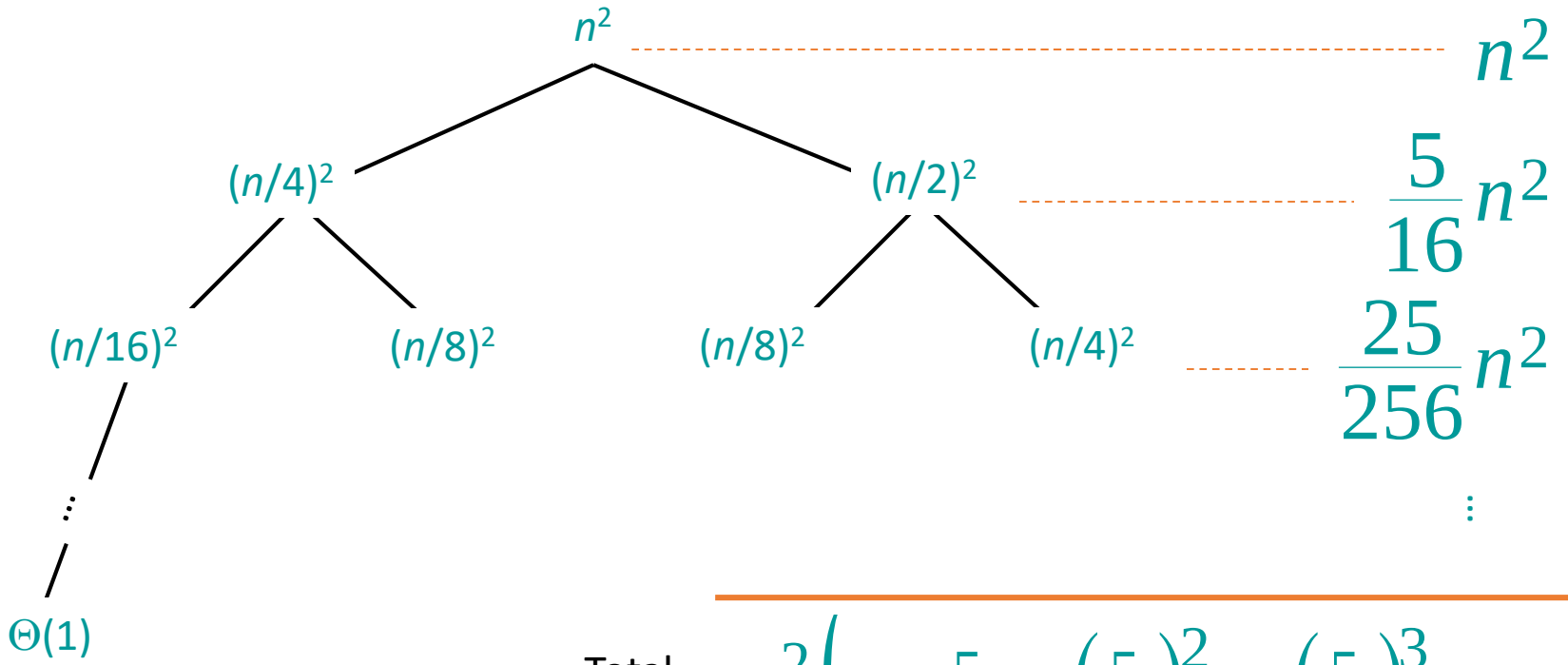
Example of recursion tree

Solve $T(n) = T(n/4) + T(n/2) + n^2$:



Example of recursion tree

Solve $T(n) = T(n/4) + T(n/2) + n^2$:



Total $= n^2\left(1 + \frac{5}{16} + \left(\frac{5}{16}\right)^2 + \left(\frac{5}{16}\right)^3 + \cdots\right)$

$= \Theta(n^2)$ *geometric series*

geometric series

$$1 + x + x^2 + \cdots + x^n = \frac{1 - x^{n+1}}{1 - x} \quad \text{for } x \neq 1$$

$$1 + x + x^2 + \cdots = \frac{1}{1 - x} \quad \text{for } |x| < 1$$

The master method

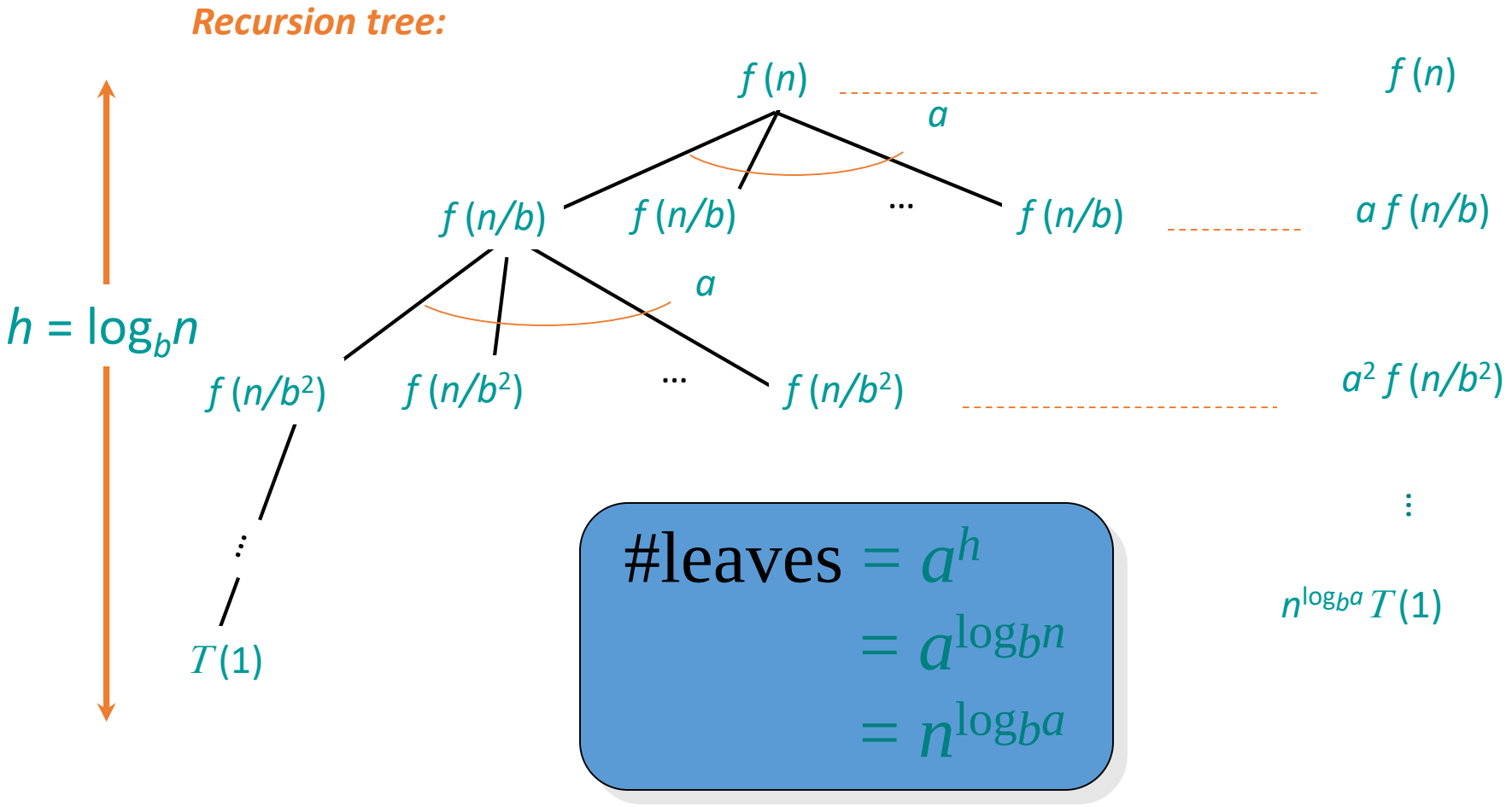
The master method applies to recurrences of the form

$$T(n) = a\,T(n/b) + f(n)\,,$$

where $a \geq 1$, $b > 1$, and f is asymptotically positive.

The concept of Master theorem…

Idea of master theorem



Three common cases

Compare $f(n)$ with $n^{\log_b a}$:

1. $f(n) = O(n^{\log_b a - \varepsilon})$ for some constant $\varepsilon > 0$.
   - $f(n)$ grows polynomially slower than $n^{\log_b a}$ (by an n^{ε} factor).

   Solution: $T(n) = \Theta(n^{\log_b a})$.

Cont..

Compare $f(n)$ with $n^{\log_b a}$:

2. $f(n) = \Theta(n^{\log_b a} \lg^k n)$ for some constant $k \geq 0$.
   - $f(n)$ and $n^{\log_b a}$ grow at similar rates.

   Solution: $T(n) = \Theta(n^{\log_b a} \lg^{k+1} n)$.

Cont..

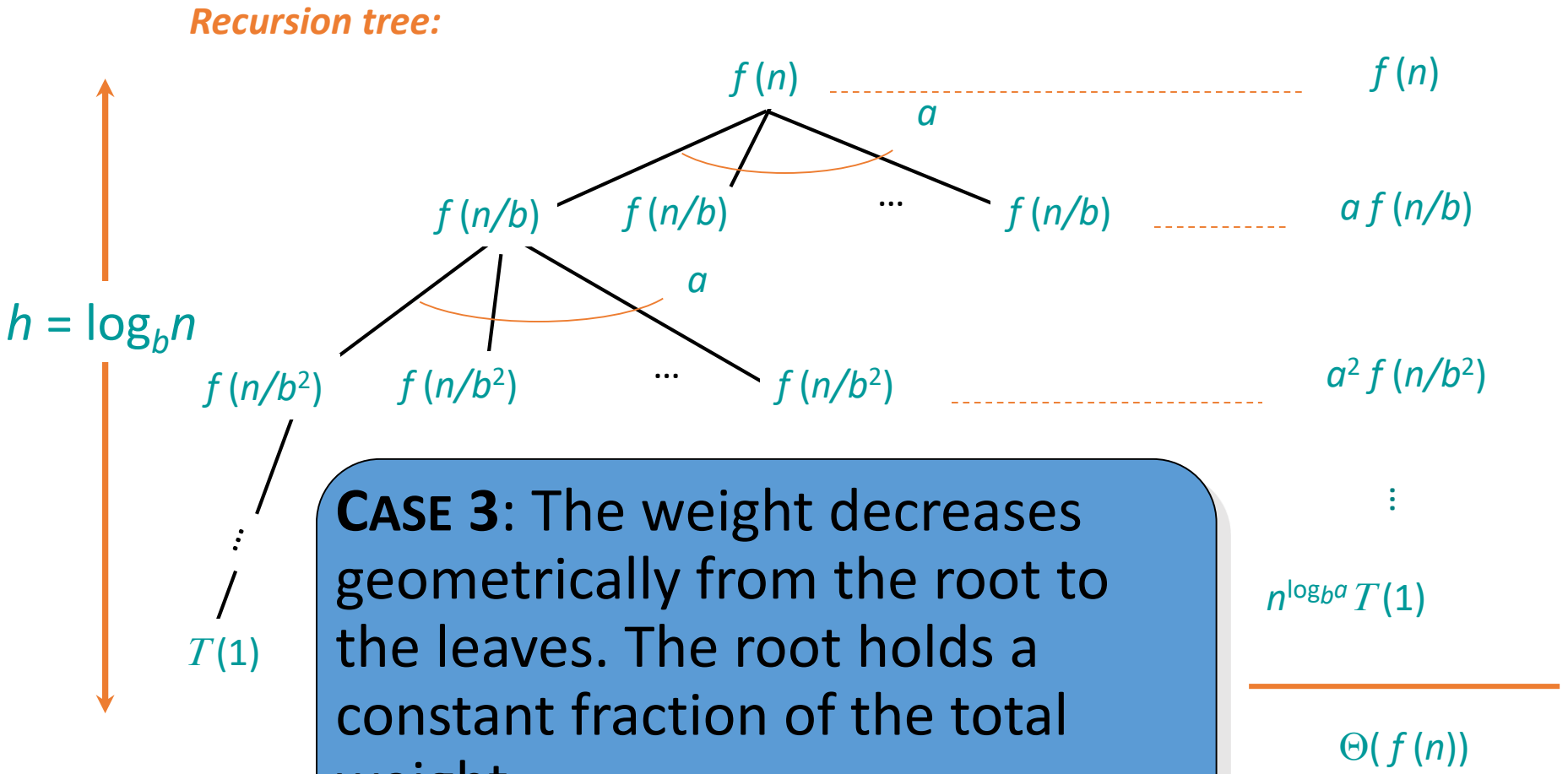
Compare $f(n)$ with $n^{\log_b a}$:

3. $f(n) = \Omega(n^{\log_b a + \varepsilon})$ for some constant $\varepsilon > 0$.
   - $f(n)$ grows polynomially faster than $n^{\log_b a}$ (by an n^{ε} factor),

   and $f(n)$ satisfies the ***regularity condition*** that $a\,f(n/b) \le c\,f(n)$ for some constant $c < 1$.

   Solution: $T(n) = \Theta(f(n))$.

Idea of master theorem



CASE 3: The weight decreases geometrically from the root to the leaves. The root holds a constant fraction of the total weight.

Examples

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

$a = 4, b = 2 \Rightarrow n^{\log_b a} = n^2$; $f(n) = n$.

CASE 1: $f(n) = O(n^{2-\varepsilon})$ for $\varepsilon = 1$.

$\therefore\ T(n) = \Theta(n^2)$.

Ex. $T(n) = 4T(n/2) + n^2$

$a = 4, b = 2 \Rightarrow n^{\log_b a} = n^2$; $f(n) = n^2$.

CASE 2: $f(n) = \Theta(n^2 \lg^0 n)$, that is, $k = 0$.

$\therefore\ T(n) = \Theta(n^2 \lg n)$.

Examples

Ex. $T(n) = 4T(n/2) + n^3$

$a = 4$, $b = 2 \Rightarrow n^{\log_b a} = n^2$; $f(n) = n^3$.

CASE 3: $f(n) = \Omega(n^{2+\varepsilon})$ for $\varepsilon = 1$

and $4(cn/2)^3 \le cn^3$ (reg. cond.) for $c = 1/2$.

$\therefore\ T(n) = \Theta(n^3)$.

Ex. $T(n) = 4T(n/2) + n^2/\lg n$

$a = 4$, $b = 2 \Rightarrow n^{\log_b a} = n^2$; $f(n) = n^2/\lg n$.

Master method does not apply. In particular, for every constant $\varepsilon > 0$, we have $n^\varepsilon = \omega(\lg n)$.

Find closest pair of points given a set of points

- The problem statement: Given n points in the plane, find the pair that is closest together.
- Let us denote the set of points by P = {p1,..., pn}, where pi has coordinates (xi, yi); and for two points pi, pj ∈ P, we use d(pi, pj) to denote the standard Euclidean distance between them.
- Our goal is to find a pair of points pi, pj that minimizes d(pi, pj).

Brute force method:

Try every pair (pi,pj) and report minimum

Complexity: $O(n^2)$

Studying One Dimensional problem first..

- A point p is given by x coordinate xp only
- d(pi,pj) = |pj - pi|
- Given n points (p1,p2,…,pn), how to find closest pair
- Sort the points (Complexity — O(n log n))
- Compute minimum separation between adjacent points after sorting
- This allows calculating only two distances with neighbouring points for each point
- So for n points 2n distances and Complexity is :O(n)
- Overall complexity: O(nlogn)

2 dimensional, Using Divide and conquer

Divide and Conquer algorithm:

- Split set of points into two halves by vertical line
- Recursively compute closest pair in left and right half
- Also compute closest pairs across separating line
- How can we do this efficiently?

Cont…

- Suppose d= min(d1, d2) where d1 = minimum distance in left partition and d2 = minimum distance in right partition
- Now we have to examine the points across the separating line
- What are the candidate points (one from left and one from right) which can have distance less than d?
- All the points have to be within the distance d from the dividing line
- But do we have to consider all the points in that distance? (in the worst case all n/2 points may lie there)

- Actually for a point p in the band of left half, we can draw two squares of depth d, one below the point p along y axis and another above the point p along the y-axis
- Let us examine it for one point in the left band of separating line
- Suppose point is p_{left} and now with how many points in right band we need to calculate the distance for this point such that there are chances that the distance is less than d
- Let us draw a square of length d first
- Now in this square how many relevant points can be there…

Cont…

- Let us see how many points can be within each square:
- If we divide a square into 4 parts than the length of diagonal of each will be d/Sqrt(2) which is less than d
- So there can be atmost four points which will have distance more than d or equal to.
- Pigeonhole principle can be applied here. If there is any 5^{th} point in this square than it will lie in one of the smaller square of size n/2 only and then the distance between two points will be less than d which is contradiction to our assumption.