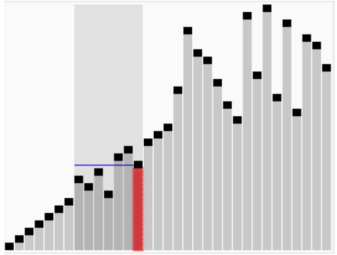


Quicksort
Lecture 8

COSC 242 – Algorithms and Data Structures

Today’s outline

1. Introduction to Quicksort
2. Partition
3. Worst case
4. Best case
5. Balanced partitioning

Today’s outline

1. Introduction to Quicksort
2. Partition
3. Worst case
4. Best case
5. Balanced partitioning

Mergsort limitations

Mergesort divides the input array A[*low..high*] into two pieces of similar size without looking at the contents.

We get pieces A[*low..mid*] and A[(*mid+1)..high*] by using the location $mid \leftarrow (low + high)/2$.

After sorting the two pieces, we have to merge them, costing us an extra *n* operations.

Can we arrange it so that we don't have to merge at the end? This is the strategy that Quicksort uses.

Key features of Quicksort

Key features

- Worst-case running time: $\theta(n^2)$
- Expected running time: $\theta(n \log_2 n)$
- Constants hidden in $\theta(n \log_2 n)$ are small.
- Sorts in place.

Quicksort is an efficient and widely used sorting algorithm.

Divide and conquer approach

Quicksort is based on a three-step process that uses divide-and-conquer.

Recall: divide-and-conquer splits a task into smaller pieces, and uses recursion to solve the smaller sub-problems. Once the base case is reached, it then recurses up, combining the mini-solutions.

Quicksort process

To sort the subarray $A[p..r]$:

Divide: Partition (rearrange) $A[p..r]$ into two (possibly empty) subarrays: $A[p..q-1]$ and $A[q+1..r]$. Each element of the first subarray $A[p..q-1] <= A[q]$. Each element of the second subarray $A[q+1..r] > A[q]$. As part of this partitioning process, identify index q. We call this the "pivot".

Conquer: Sort $A[p..q-1]$ and $A[q+1..r]$ by recursive calls to quicksort.

Combine: Combine the two subarrays. Since both are now sorted, no work is done.

Quicksort implementation


```
QUICKSORT (A, p, r):
1 if p < r
2     q = PARTITION(A, p, r)
3     QUICKSORT(A, p, q-1)      // First half, from p up to partition point q
4     QUICKSORT(A, q+1, r)      // Second half, from q+1 up to end r
```

Recall
A[p .. r]
p = first index
r = last index

Initial call is `QUICKSORT(A, 1, n)`

Need more help?

If at the end of today's notes you need some extra help, check out this Kahn academy class, co-written by the textbook's first author [1].

Today’s outline

1. Introduction to Quicksort
2. Partition
3. Worst case
4. Best case
5. Balanced partitioning

Partitioning

The quicksort algorithm is deceptively simple. However, the key insight lies in the partitioning process.

```
PARTITION(A, p, r)
  x = A[r]
  i = p - 1
  for j = p to r - 1
     if A[j] ≤ x
           i = i + 1
           swap A[i] with A[j]
  swap A[i+1] with A[r]
  return i + 1
```

PARTITION always selects an element x = $A[r]$ as a ***pivot*** element around which to partition the subarray $A[p..r]$.

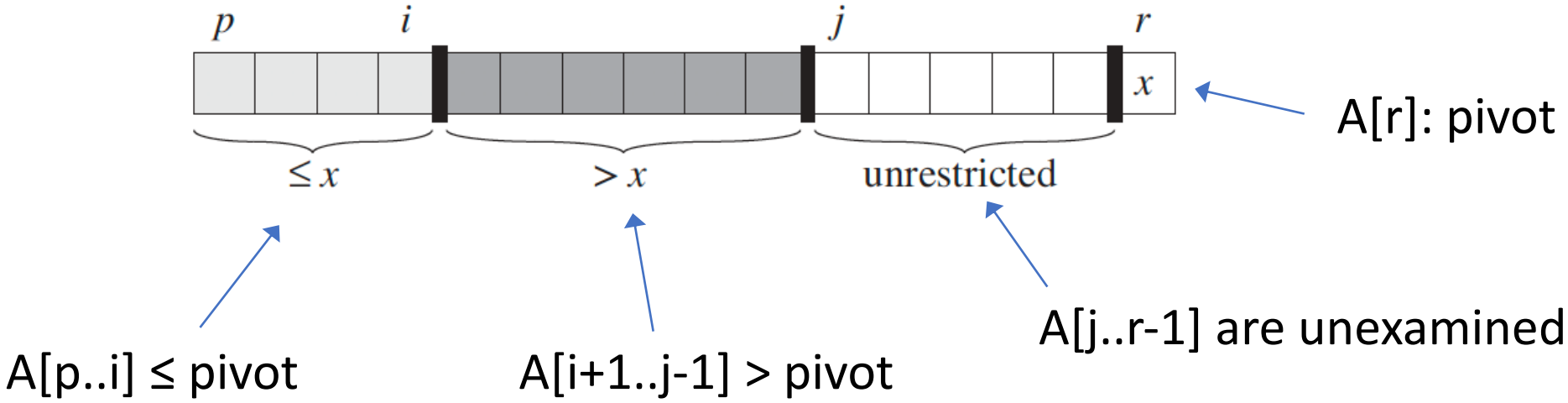
Visualising quicksort

Lightly shaded: First partition ≤ x, that is, ≤ A[r]

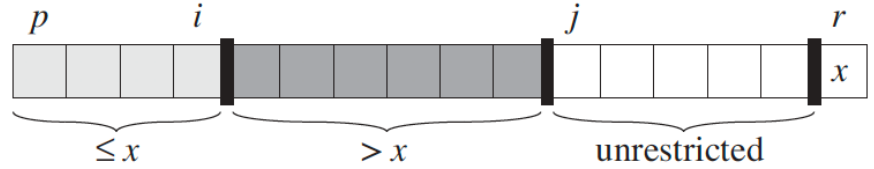
Heavily shaded: Second partition > x

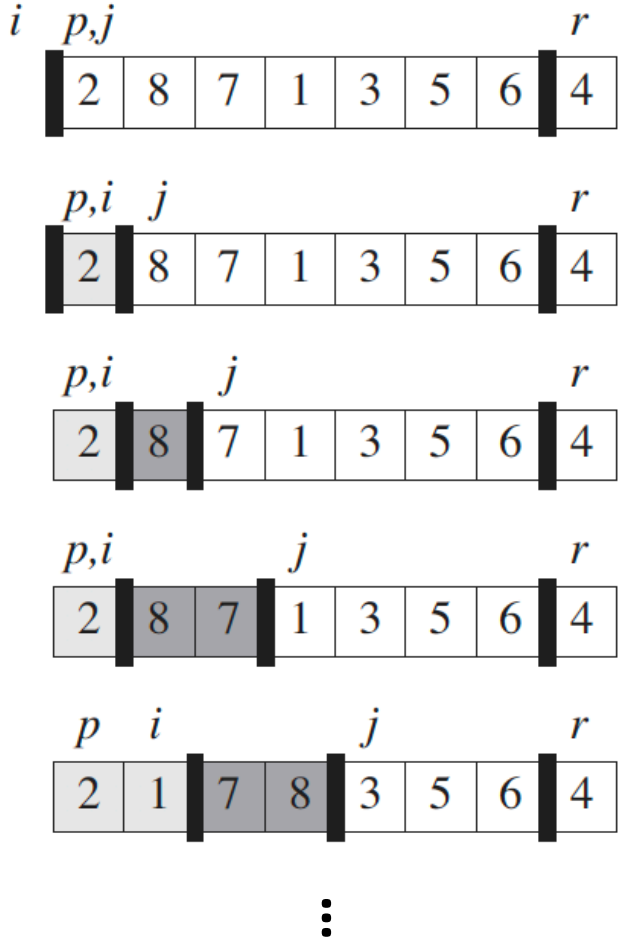
Unshaded: Elements not yet put in a partition

Unshaded r: The pivot x



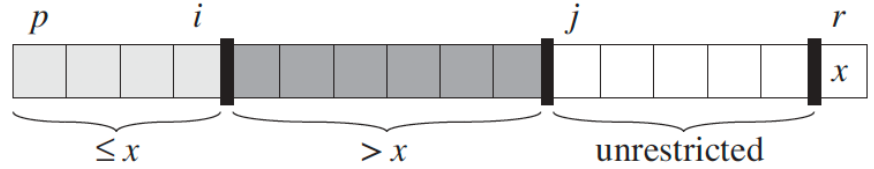
Partition in action

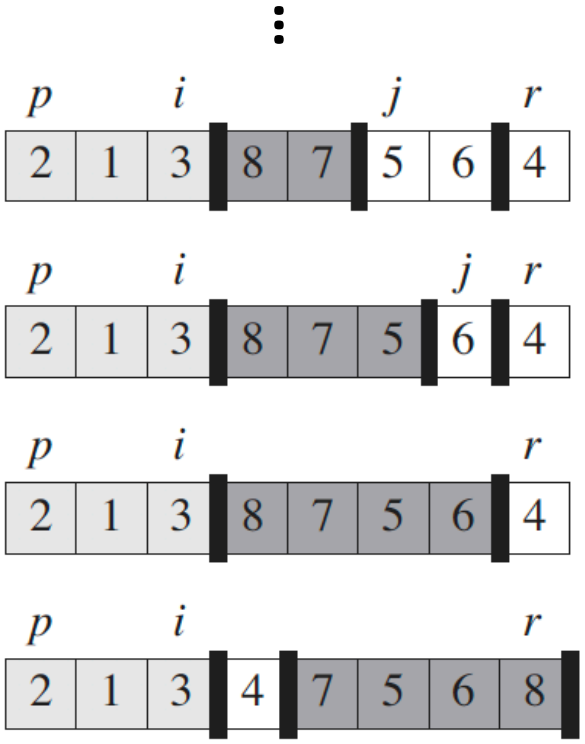





```
PARTITION(A, p, r)
   x = A[r]
   i = p - 1
   for j = p to r - 1
    if A[j] ≤ x
       i = i + 1
       swap A[i] with A[j]
   swap A[i+1] with A[r]
   return i + 1
```

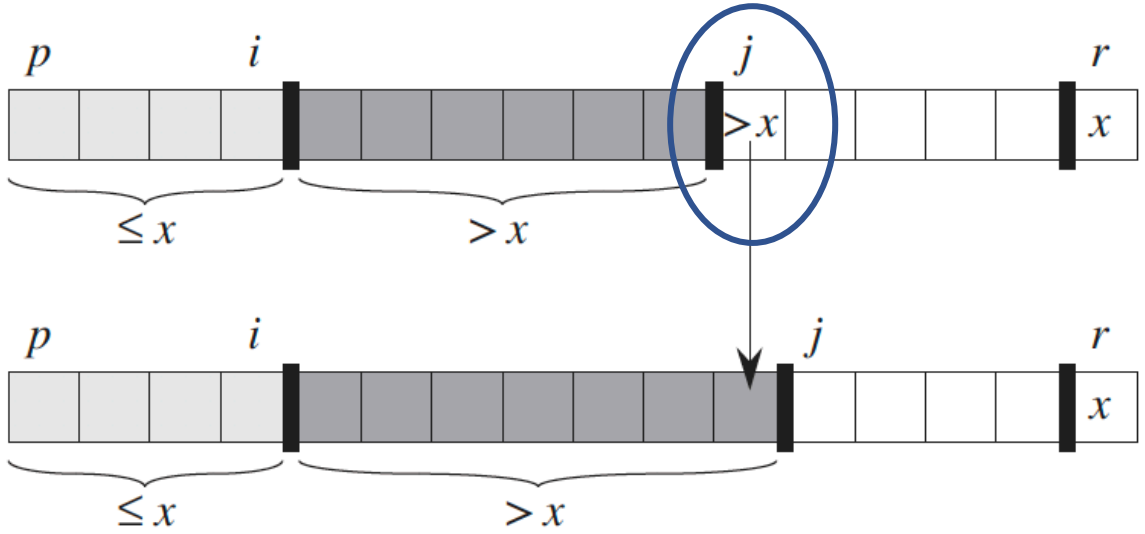
Partition in action






```
PARTITION(A, p, r)
   x = A[r]
   i = p - 1
   for j = p to r - 1
    if A[j] ≤ x
       i = i + 1
       swap A[i] with A[j]
   swap A[i+1] with A[r]
   return i + 1
```

Partition example 1



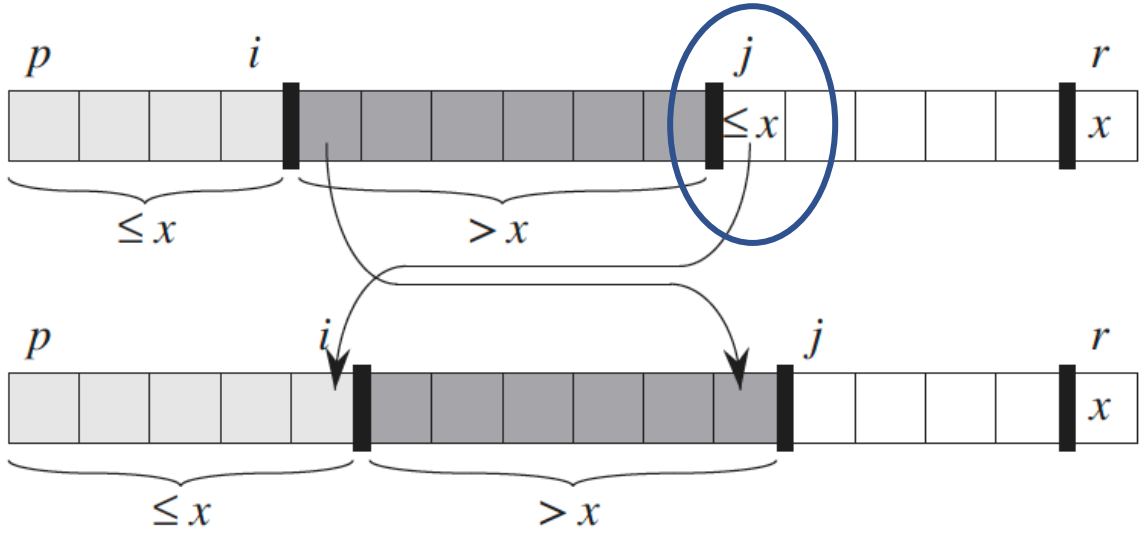

```
PARTITION(A, p, r)
   x = A[r]
   i = p - 1
   for j = p to r - 1
    if A[j] ≤ x
        i = i + 1
        swap A[i] with A[j]
   swap A[i+1] with A[r]
   return i + 1
```

If $A[j] > x$, then:

1. j is incremented

That is, the heavily shaded region grows by 1.

Partition example 2




```
PARTITION(A, p, r)
   x = A[r]
   i = p - 1
   for j = p to r - 1
    if A[j] ≤ x
        i = i + 1
        swap A[i] with A[j]
   swap A[i+1] with A[r]
   return i + 1
```

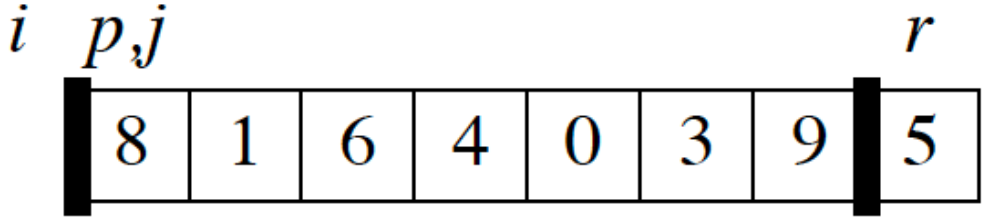
If $A[j] \leq x$, then:

1. index i is incremented
2. $A[i]$ and $A[j]$ are swapped
3. j is incremented

That is, the heavily shaded region grows by 1.

Class challenge 1

Apply partition to the following array:




```
PARTITION(A, p, r)
   x = A[r]
   i = p - 1
   for j = p to r - 1
    if A[j] ≤ x
        i = i + 1
        swap A[i] with A[j]
   swap A[i+1] with A[r]
   return i + 1
```

Partition analysis

```
PARTITION(A, p, r)
1.  x = A[r]
2.  i = p - 1
3.  for j = p to r - 1
4.      if A[j] ≤ x
5.          i = i + 1
6.          swap A[i] with A[j]
7.  swap A[i+1] with A[r]
8.  return i + 1
```

To analyse Partition, we can use a simple counting argument as we did for Merge:

- i starts at p-1
- j starts at p
- Each time through the for loop (line 3), j is incremented at least once (line 3).
- Only if our element is less than our pivot do we increment i (line 5), and swap elements (line 6).
- We then do our final swap of our pivot (line 7), and return the index of our pivot (line 8).

Partition analysis

O(n)

```
PARTITION(A, p, r)
1.  x = A[r]
2.  i = p - 1
3.  for j = p to r - 1
4.      if A[j] ≤ x
5.          i = i + 1
6.          swap A[i] with A[j]
7.  swap A[i+1] with A[r]
8.  return i + 1
```

The for loop body (starts line 3) executes r−1−p = Θ(n) times.

Why? In the worst case, every time the body of the **if** is executed, it takes constant time, or O(1). Lines 1, 2, and 7 are also constant time. However, this is the same running time if the array is already sorted.

Thus the running time is Θ(n).

Overview of quicksort analysis

The running time of quicksort depends on the partitioning of the subarrays:

- If the subarrays are balanced, then quicksort can run as fast as mergesort, $\boldsymbol{\Theta}(n \cdot \log n)$.
- If they are unbalanced, then quicksort can run as slowly as insertion sort, $\boldsymbol{\Theta}(n^2)$.

Today’s outline

1. Introduction to Quicksort
2. Partition
3. Worst case
4. Best case
5. Balanced partitioning

Quicksort: worst case analysis

For the worst case, suppose that we were really unlucky, and the subarray partitions are maximally unbalanced.

Specifically, the pivot is always either the **largest or smallest** element in the array.

Then one subarray will have 0 elements, while the other will contain n-1 elements; that is, all element except the pivot (hence n-1).

The two quicksort recursive calls will be on subarrays of **size = 0** and **size = n-1**.

```
QUICKSORT (A, p, r):
1 if p < r
2   q = PARTITION(A, p, r)
3   QUICKSORT(A, p, q-1)
4   QUICKSORT(A, q+1, r)
```

Quicksort: worst case analysis

```
QUICKSORT (A, p, r):
1 if p < r
2   q = PARTITION(A, p, r)
3   QUICKSORT(A, p, q-1)
4   QUICKSORT(A, q+1, r)
```

The time for partitioning is $\Theta(n)$.

A recursive call on the empty subarray just returns, so $T(0) = \Theta(1)$.

A recursive call on the full subarray is $T(n-1)$.

Recurrence for running time:

$$\begin{aligned} T(n) &= T(n-1) + T(0) + \Theta(n) \\ &= T(n-1) + \Theta(n) \end{aligned}$$

Solve the recurrence

Lets state our function:

Base case: $T(1) = 1$

Thing to solve: $T(n) = T(n-1) + n$

Solve the recurrence

Base case: $f(1) = 1$

Function: $f(n) = f(n-1) + \mathrm{n}$

Solution

k = 1 $f(n) = n + f(n-1)$

Workspace

Solve the recurrence

Base case: $f(1) = 1$

Function: $f(n) = f(n-1) + n$

Solution

k = 1 $f(n) = n + f(n-1)$

Lets expand this

Workspace

$f(n-1) =$

Lets take our f(n), and since we wish to expand f(n-1), we need to subtract 1 from every n.

Solve the recurrence

Base case: $f(1) = 1$

Function: $f(n) = f(n-1) + \mathrm{n}$

Solution

k = 1 $f(n) = n + f(n-1)$

Workspace

$f(n-1) = f((n-1)-1) + \mathrm{n} - 1$

$= f(n-2) + n - 1$

Solve the recurrence

Base case: $f(1) = 1$

Function: $f(n) = f(n-1) + \mathrm{n}$

Solution

k = 1 $f(n) = n + f(n-1)$

Workspace

$f(n-1) = f((n-1)-1) + \mathrm{n} - 1$

$= f(n-2) + n - 1$

Lets now plug our workspace f(n-1) expansion back into our solution side

Solve the recurrence

Base case: $f(1) = 1$

Function: $f(n) = f(n-1) + \mathrm{n}$

Solution

k = 1 $f(n) = n + f(n-1)$

$= n + [n - 1 + f(n-2)]$

Workspace

$f(n-1) = f((n-1)-1) + \mathrm{n} - 1$

$= f(n-2) + n - 1$

Solve the recurrence

Base case: $f(1) = 1$

Function: $f(n) = f(n-1) + \mathrm{n}$

Solution

k = 1 $f(n) = n + f(n-1)$

$= n + [n - 1 + f(n-2)]$

k = 2 $= n + (n-1) + f(n-2)$

Workspace

$f(n-1) = f((n-1)-1) + \mathrm{n} - 1$

$= f(n-2) + n - 1$

Solve the recurrence

Base case: $f(1) = 1$

Function: $f(n) = f(n-1) + \mathrm{n}$

Solution

k = 1 $f(n) = n + f(n-1)$

$= n + [n - 1 + f(n-2)]$

k = 2 $= n + (n-1) + f(n-2)$

Lets expand this

Workspace

$f(n-1) = f((n-1)-1) + \mathrm{n} - 1$

$= f(n-2) + n - 1$

$f(n-2) = f((n-2)-1) + \mathrm{n} - 2$

$= f(n-3) + n - 2$

Solve the recurrence

Base case: $f(1) = 1$

Function: $f(n) = f(n-1) + \mathrm{n}$

Solution

k = 1 $f(n) = n + f(n-1)$

$= n + [n - 1 + f(n-2)]$

k = 2 $= n + (n-1) + f(n-2)$

Workspace

$f(n-1) = f((n-1)-1) + \mathrm{n} - 1$

$= f(n-2) + n - 1$

$f(n-2) = f((n-2)-1) + \mathrm{n} - 2$

$= f(n-3) + n - 2$

Lets now plug our workspace f(n-2) expansion back into our solution side

Solve the recurrence

Base case: $f(1) = 1$

Function: $f(n) = f(n-1) + \mathrm{n}$

Solution

k = 1 $f(n) = n + f(n-1)$

$= n + [n - 1 + f(n-2)]$

k = 2 $= n + (n-1) + f(n-2)$

$= n + (n-1) + [(n-2) + f(n-3)]$

k = 3 $= n + (n-1) + (n-2) + f(n-3)$

Workspace

$f(n-1) = f((n-1)-1) + \mathrm{n} - 1$

$= f(n-2) + n - 1$

$f(n-2) = f((n-2)-1) + \mathrm{n} - 2$

$= f(n-3) + n - 2$

Solve the recurrence

Base case: $f(1) = 1$

Function: $f(n) = f(n-1) + \mathrm{n}$

Solution

k = 1 $f(n) = n + f(n-1)$

$= n + [n - 1 + f(n-2)]$

k = 2 $= n + (n-1) + f(n-2)$

$= n + (n-1) + [(n-2) + f(n-3)]$

k = 3 $= n + (n-1) + (n-2) + f(n-3)$

Lets expand this

Workspace

$f(n-1) = f((n-1)-1) + \mathrm{n} - 1$

$= f(n-2) + n - 1$

$f(n-2) = f((n-2)-1) + \mathrm{n} - 2$

$= f(n-3) + n - 2$

$f(n-3) = f((n-3)-1) + \mathrm{n} - 3$

$f(n-4) + n - 3$

Solve the recurrence

Base case: $f(1) = 1$

Function: $f(n) = f(n-1) + \mathrm{n}$

Solution

k = 1 $f(n) = n + f(n-1)$

$= n + [n - 1 + f(n-2)]$

k = 2 $= n + (n-1) + f(n-2)$

$= n + (n-1) + [(n-2) + f(n-3)]$

k = 3 $= n + (n-1) + (n-2) + f(n-3)$

Workspace

$f(n-1) = f((n-1)-1) + \mathrm{n} - 1$

$= f(n-2) + n - 1$

$f(n-2) = f((n-2)-1) + \mathrm{n} - 2$

$= f(n-3) + n - 2$

$f(n-3) = f((n-3)-1) + \mathrm{n} - 3$

$f(n-4) + n - 3$

Lets now plug our workspace f(n-3) expansion back into our solution side

Solve the recurrence

Base case: $f(1) = 1$

Function: $f(n) = f(n-1) + \mathrm{n}$

Solution

k = 1 $f(n) = n + f(n-1)$

$= n + [n - 1 + f(n-2)]$

k = 2 $= n + (n-1) + f(n-2)$

$= n + (n-1) + [(n-2) + f(n-3)]$

k = 3 $= n + (n-1) + (n-2) + f(n-3)$

$= n + (n-1) + (n-2) + [(n-3) + f(n-4)]$

k = 4 $= n + (n-1) + (n-2) + (n-3) + f(n-4)$

Workspace

$f(n-1) = f((n-1)-1) + \mathrm{n} - 1$

$= f(n-2) + n - 1$

$f(n-2) = f((n-2)-1) + \mathrm{n} - 2$

$= f(n-3) + n - 2$

$f(n-3) = f((n-3)-1) + \mathrm{n} - 3$

$f(n-4) + n - 3$

Solve the recurrence

$$
\begin{aligned}
f(n) &= n + f(n-1) \\
&= n + (n-1) + f(n-2) \\
&= n + (n-1) + (n-2) + f(n-3) \\
&= n + (n-1) + (n-2) + (n-3) + f(n-4) \\
&= n + (n-1) + (n-2) + (n-3) + \ldots + 3 + 2 + f(1)
\end{aligned}
$$

Solve the recurrence

$$
\begin{aligned}
f(n) &= n + f(n-1) \\
&= n + (n-1) + f(n-2) \\
&= n + (n-1) + (n-2) + f(n-3) \\
&= n + (n-1) + (n-2) + (n-3) + f(n-4) \\
&= n + (n-1) + (n-2) + (n-3) + \ldots + 3 + 2 + \cancel{f(1)}^{0}
\end{aligned}
$$

With 1 element subarray, cost is 0

Solve the recurrence

$$
\begin{aligned}
f(n) &= n + f(n-1) \\
&= n + (n-1) + f(n-2) \\
&= n + (n-1) + (n-2) + f(n-3) \\
&= n + (n-1) + (n-2) + (n-3) + f(n-4) \\
&= n + (n-1) + (n-2) + (n-3) + \ldots + 3 + 2 + 0 \\
&= (n + (n-1) + (n-2) + (n-3) + \ldots + 3 + 2 + 1) - 1
\end{aligned}
$$

Lets rewrite that 0 as 1 - 1

Solve the recurrence

$$
\begin{aligned}
f(n) &= n + f(n-1) \\
&= n + (n-1) + f(n-2) \\
&= n + (n-1) + (n-2) + f(n-3) \\
&= n + (n-1) + (n-2) + (n-3) + f(n-4) \\
&= n + (n-1) + (n-2) + (n-3) + \ldots + 3 + 2 + 0 \\
&= (n + (n-1) + (n-2) + (n-3) + \ldots + 3 + 2 + 1) - 1
\end{aligned}
$$

Hmm, doesn't this look like the sum of an arithmetic series?

Solve the recurrence

$$
\begin{aligned}
f(n) &= n + f(n-1) \\
&= n + (n-1) + f(n-2) \\
&= n + (n-1) + (n-2) + f(n-3) \\
&= n + (n-1) + (n-2) + (n-3) + f(n-4) \\
&= n + (n-1) + (n-2) + (n-3) + \ldots + 3 + 2 + 0 \\
&= (n + (n-1) + (n-2) + (n-3) + \ldots + 3 + 2 + 1) - 1 \\
&= n(n+1)/2 - 1
\end{aligned}
$$

Replaced with the closed-form solution

Solve the recurrence

$$
\begin{aligned}
f(n) &= n + f(n-1) \\
&= n + (n-1) + f(n-2) \\
&= n + (n-1) + (n-2) + f(n-3) \\
&= n + (n-1) + (n-2) + (n-3) + f(n-4) \\
&= n + (n-1) + (n-2) + (n-3) + \ldots + 3 + 2 + 0 \\
&= (n + (n-1) + (n-2) + (n-3) + \ldots + 3 + 2 + 1) - 1 \\
&= n(n+1)/2 - 1 \\
&\approx O(n^2)
\end{aligned}
$$

Worst case

So in the worst case Quicksort is $O(n^2)$. That's not good.

The worst case occurs exactly when Quicksort is run on an already sorted array. But insertion sort takes only O(n) in this case.

We can avoid the worst case by choosing a better pivot. The most common choices are: choose a random element as the pivot, or choose the median of 3 elements (e.g. first, middle, last) as the pivot.

Today’s outline

1. Introduction to Quicksort
2. Partition
3. Worst case
4. Best case
5. Balanced partitioning

Quicksort: best case analysis

The best case occurs when partition produces two subarrays, each of which is no more than $n/2$. That is, completely balanced subarrays every time. Our function here is:

Base case: $T(1) = 1$

Thing to solve: $T(n) = T(n/2) + T(n/2)$

This is the same equation as mergesort (phew!)

In the best case, Quicksort is $O(n \log_2 n + n)$

Today’s outline

1. Introduction to Quicksort
2. Partition
3. Worst case
4. Best case
5. Balanced partitioning

Balanced partitioning

That's the worst case and best case. But what about the **average case**?

The average-case running time of quicksort is much closer to the best case than to the worst case.

Consider a pivot that always gives us a 9-to-1 (90%/10%) split of the data (hmm, that sounds bad…). In that case, the function is:

$$T(n) = T(9n/10) + T(n/10) + \theta(n)$$

This is tricky to solve with the substitution method, so we will use a **recursion tree** instead.

Recursion tree

We use a recursion tree to help solve a recurrence equation.

This visual technique is analogous to using the substitution method, as the basic idea is the same:

Intuition

1. Keep expanding the recursive sub-calls until we reach a base case.
2. Identify the depth of our recursion tree
3. Identify the number of operations done at each level
4. Sum up the work across all levels

Our equations

Lets start by restating our equations:

Base case: $T(1) = 1$

Recurrence: $T(n) = T(9n/10) + T(n/10) + c \cdot n$

Here we've explicitly included the constant hidden in our $\boldsymbol{\Theta}$(n) term. As you will see, this is important in a recursion tree.

$T(1) = 1$

$T(n) = \underbrace{T(9n/10)}_{\text{Recursive 1}} + \underbrace{T(n/10)}_{\text{Recursive 2}} + \underbrace{cn}_{\text{Constant}}$

Tree depth	Tree	Row sum
0	n	

T(9/10) **T(1/10)**

At the first level of recursion, our algorithm does **n work plus two recursive calls.**

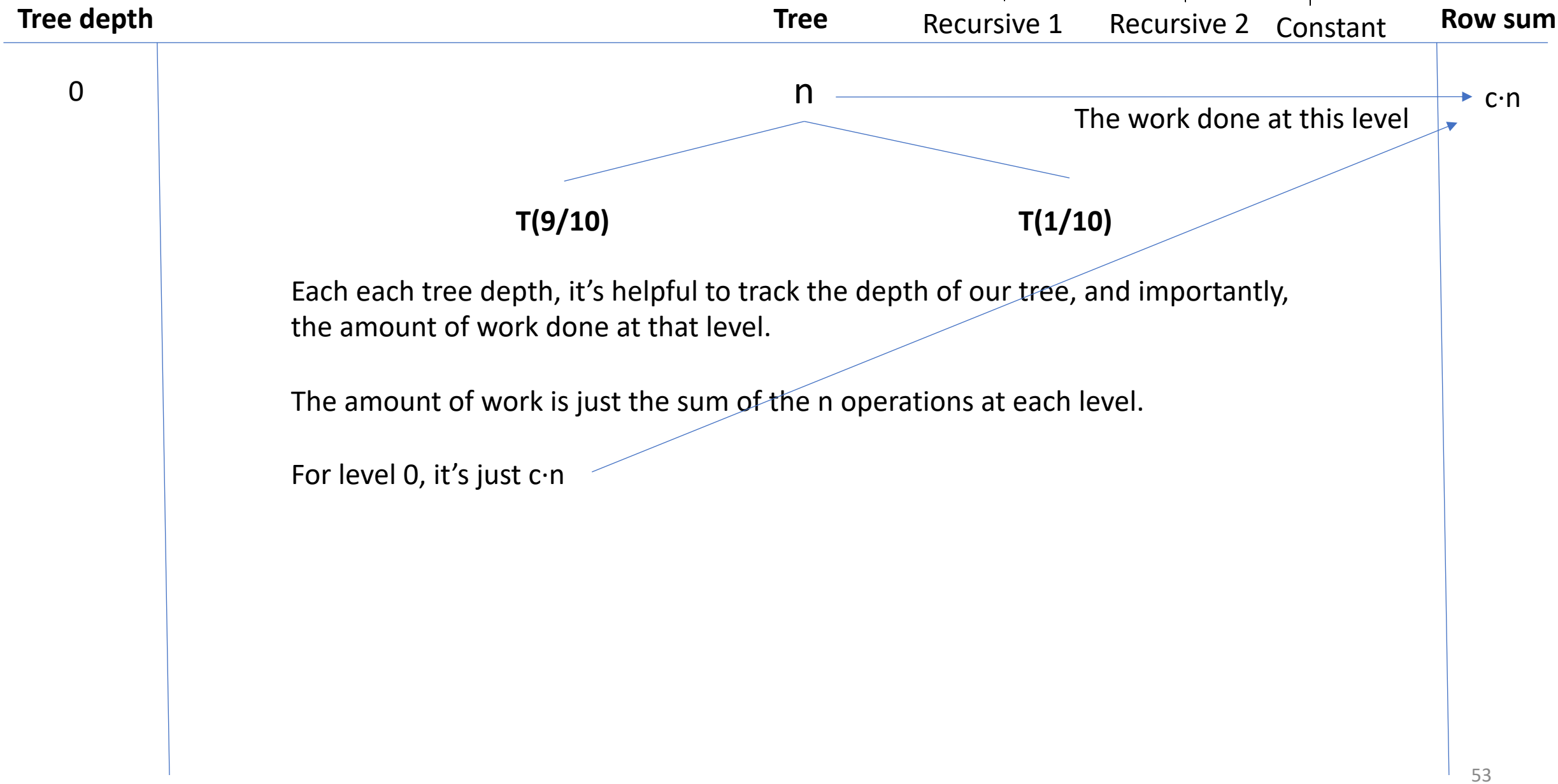
Those calls operate on arrays of size 9/10n and 1/10n of the original (size n).

That is, we will split the job into two parts, giving 90% to one recursive call, and 10% to the other. Each split will in turn take some amount of c·n.

Just like the substitution method, we now move to the next level of our recursion.

$T(1) = 1$

$T(n) = \underbrace{T(9n/10)}_{\text{Recursive 1}} + \underbrace{T(n/10)}_{\text{Recursive 2}} + \underbrace{cn}_{\text{Constant}}$



Each each tree depth, it's helpful to track the depth of our tree, and importantly, the amount of work done at that level.

The amount of work is just the sum of the n operations at each level.

For level 0, it's just c·n

$T(1) = 1$

$T(n) = T(9n/10) + T(n/10) + cn$

Recursive 1 Recursive 2 Constant

Tree depth **Tree** **Row sum**

0 n

T(9/10) **T(1/10)**

At depth 1, we calculated our recurrences by expanding n for ‘n’ in our equation.

$T(1) = 1$

$T(n) = T(9n/10) + T(n/10) + cn$

Recursive 1 Recursive 2 Constant

Tree depth **Tree** **Row sum**

0 n c·n

1 $\frac{9}{10}n$ $\frac{1}{10}n$

2

To determine the size of each recursive call, we inserted **T(9n/10)** and **T(1n/10)** for **n** into our equation.

$T(1) = 1$

$T(n) = \underbrace{T(9n/10)}_{\text{Recursive 1}} + \underbrace{T(n/10)}_{\text{Recursive 2}} + \underbrace{cn}_{\text{Constant}}$

Tree depth	Tree	Row sum
0	n	c·n
1	$\frac{9}{10}n$ + $\frac{1}{10}n$	c·n
2		

Calculate the work at depth 1

$T(1) = 1$

$T(n) = T(9n/10) + T(n/10) + cn$

Recursive 1 Recursive 2 Constant

Tree depth	Tree	Row sum
0	n	c·n
1	$\frac{9}{10}n$ $\frac{1}{10}n$	c·n
2	$T(9^2/10^2)$ $T(9/10^2)$	

For our recurrences at depth 2, we now expand T(9n/10) for the left side.

$T(1) = 1$

$T(n) = T(9n/10) + T(n/10) + cn$

Recursive 1 Recursive 2 Constant

Tree depth **Tree** **Row sum**

0 n c·n

1 $\frac{9}{10}n$ $\frac{1}{10}n$ c·n

2 **T(9²/10²)** **T(9/10²)** T(9/10²) T(1²/10²)

And T(1n/10) for our right side.

$T(1) = 1$

$T(n) = T(9n/10) + T(n/10) + cn$

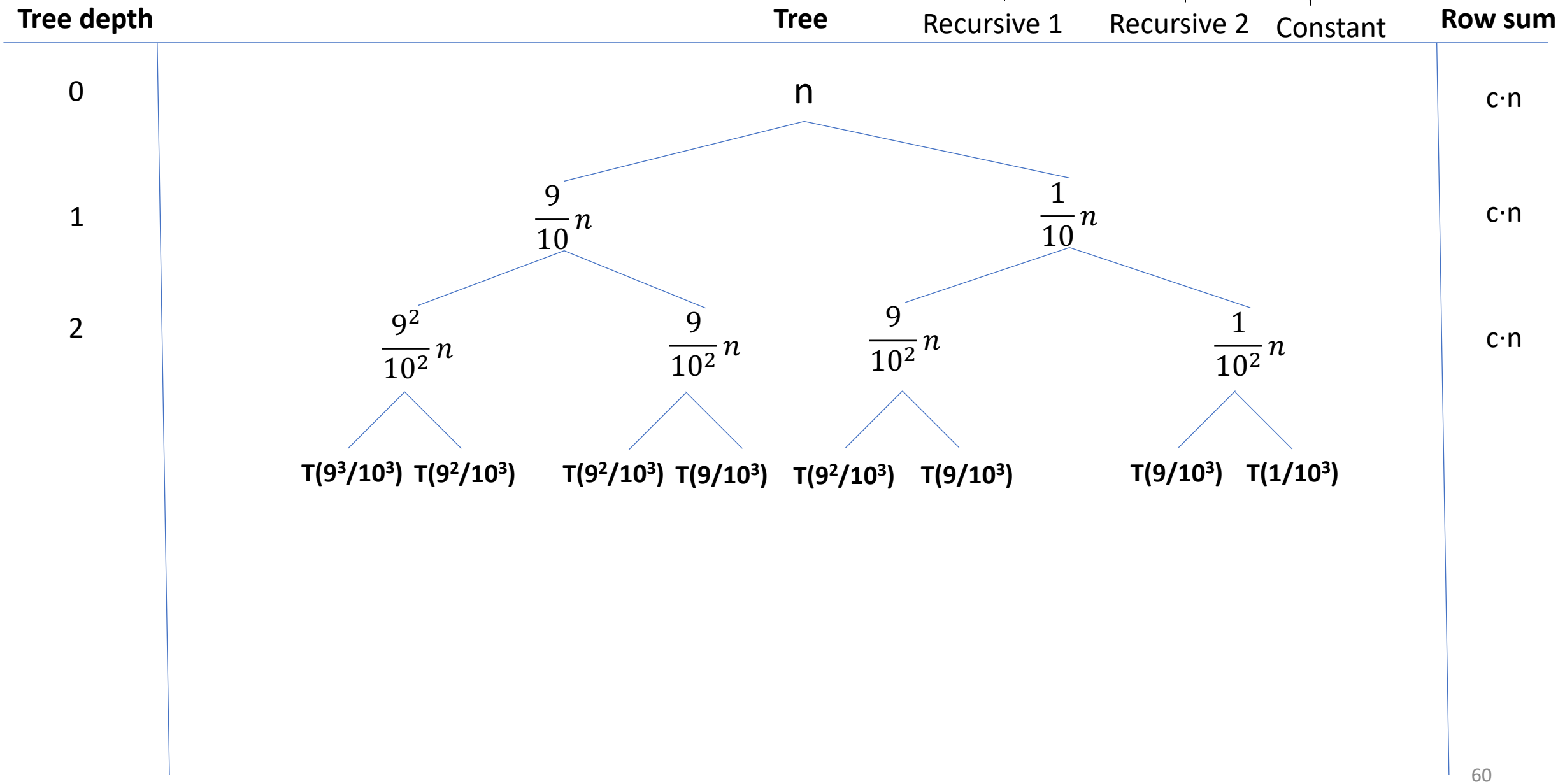
Recursive 1 Recursive 2 Constant

Tree depth	Tree	Row sum
0	n	c·n
1	$\frac{9}{10}n$ + $\frac{1}{10}n$	c·n
2	$\frac{9^2}{10^2}n$ + $\frac{9}{10^2}n$ + $\frac{9}{10^2}n$ + $\frac{1}{10^2}n$	c·n

Insert each of our recurrences into our constant work at each level.
Calculate the work at this depth.

$T(1) = 1$

$T(n) = \underbrace{T(9n/10)}_{\text{Recursive 1}} + \underbrace{T(n/10)}_{\text{Recursive 2}} + \underbrace{cn}_{\text{Constant}}$



$T(1) = 1$

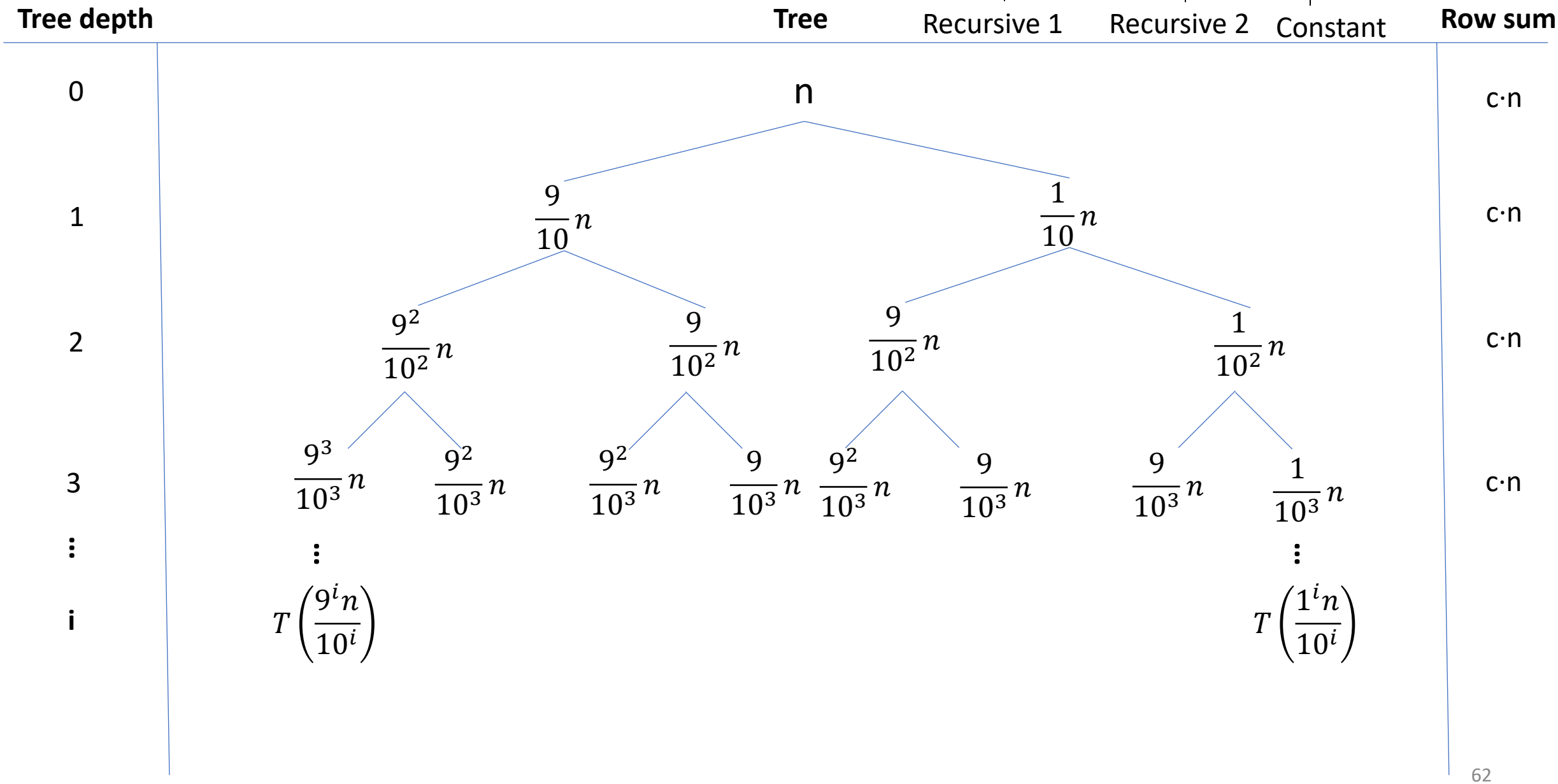
$T(n) = \underbrace{T(9n/10)}_{\text{Recursive 1}} + \underbrace{T(n/10)}_{\text{Recursive 2}} + \underbrace{cn}_{\text{Constant}}$

Tree depth	Tree	Row sum
0	n	c·n
1	$\frac{9}{10}n$ $\quad$ $\frac{1}{10}n$	c·n
2	$\frac{9^2}{10^2}n$ $\quad$ $\frac{9}{10^2}n$ $\quad$ $\frac{9}{10^2}n$ $\quad$ $\frac{1}{10^2}n$	c·n
3	$\frac{9^3}{10^3}n$ $\quad$ $\frac{9^2}{10^3}n$ $\quad$ $\frac{9^2}{10^3}n$ $\quad$ $\frac{9}{10^3}n$ $\quad$ $\frac{9^2}{10^3}n$ $\quad$ $\frac{9}{10^3}n$ $\quad$ $\frac{9}{10^3}n$ $\quad$ $\frac{1}{10^3}n$	c·n

We've now expanded 3 levels of our recursion tree.
This allows us to see a pattern forming.

$T(1) = 1$

$T(n) = \underbrace{T(9n/10)}_{\text{Recursive 1}} + \underbrace{T(n/10)}_{\text{Recursive 2}} + \underbrace{cn}_{\text{Constant}}$



$T(1) = 1$

$T(n) = \underbrace{T(9n/10)}_{\text{Recursive 1}} + \underbrace{T(n/10)}_{\text{Recursive 2}} + \underbrace{cn}_{\text{Constant}}$

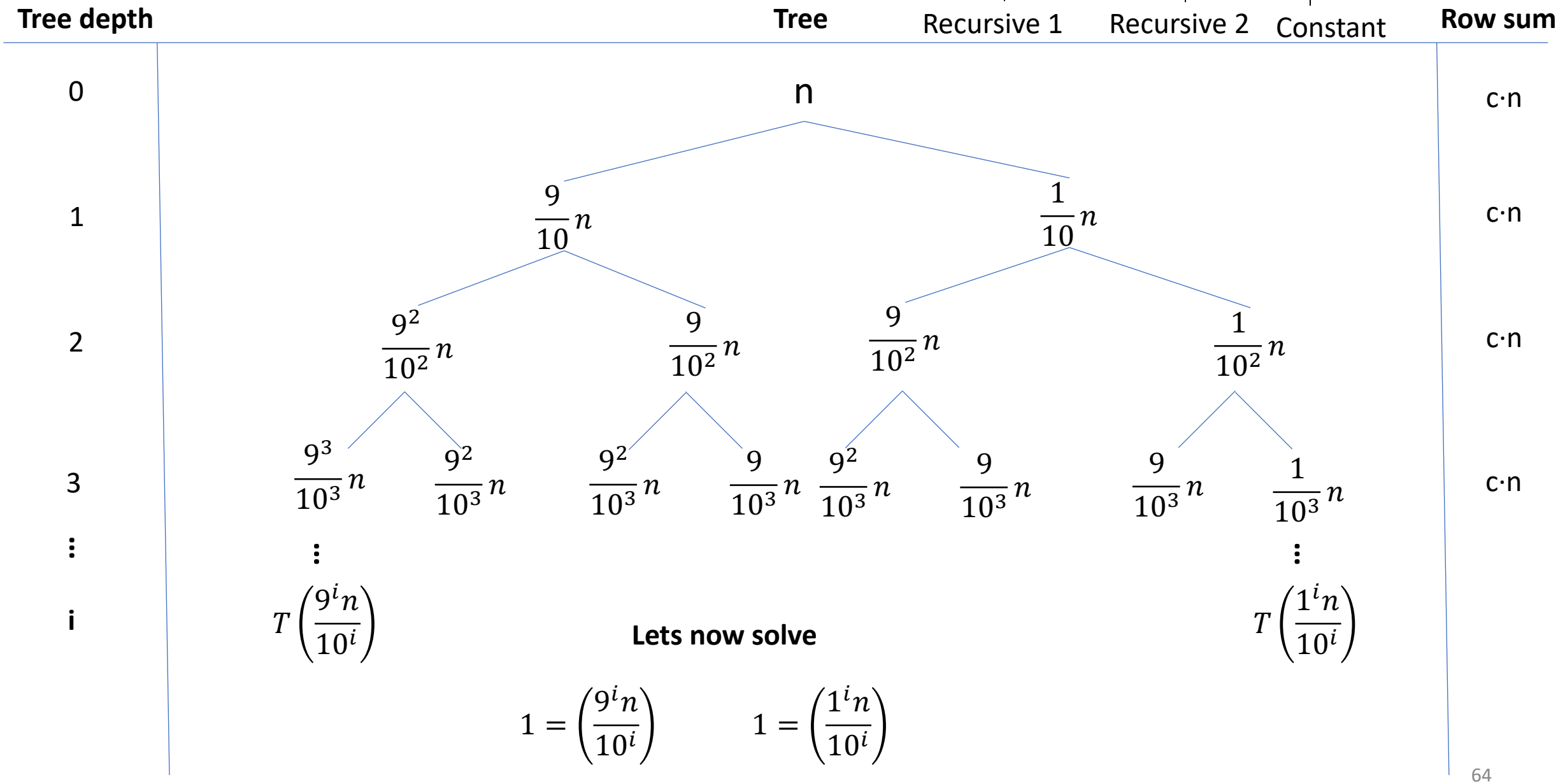
Tree depth	Tree	Row sum
0	n	c·n
1	$\frac{9}{10}n$ $\quad$ $\frac{1}{10}n$	c·n
2	$\frac{9^2}{10^2}n$ $\quad$ $\frac{9}{10^2}n$ $\quad$ $\frac{9}{10^2}n$ $\quad$ $\frac{1}{10^2}n$	c·n
3	$\frac{9^3}{10^3}n$ $\quad$ $\frac{9^2}{10^3}n$ $\quad$ $\frac{9^2}{10^3}n$ $\quad$ $\frac{9}{10^3}n$ $\quad$ $\frac{9^2}{10^3}n$ $\quad$ $\frac{9}{10^3}n$ $\quad$ $\frac{9}{10^3}n$ $\quad$ $\frac{1}{10^3}n$	c·n
⋮	⋮ ⋮	
i	$T\left(\frac{9^i n}{10^i}\right)$ $\quad$ $T\left(\frac{1^i n}{10^i}\right)$	

Great! So when do we stop? When we reach our base case T(1) = 1

Important Note: why do we have two conditions here? Because we're always splitting into 90% and 10%. Crucially, our 10% will reach "base case" much sooner than our 90%.

$T(1) = 1$

$T(n) = \underbrace{T(9n/10)}_{\text{Recursive 1}} + \underbrace{T(n/10)}_{\text{Recursive 2}} + \underbrace{cn}_{\text{Constant}}$

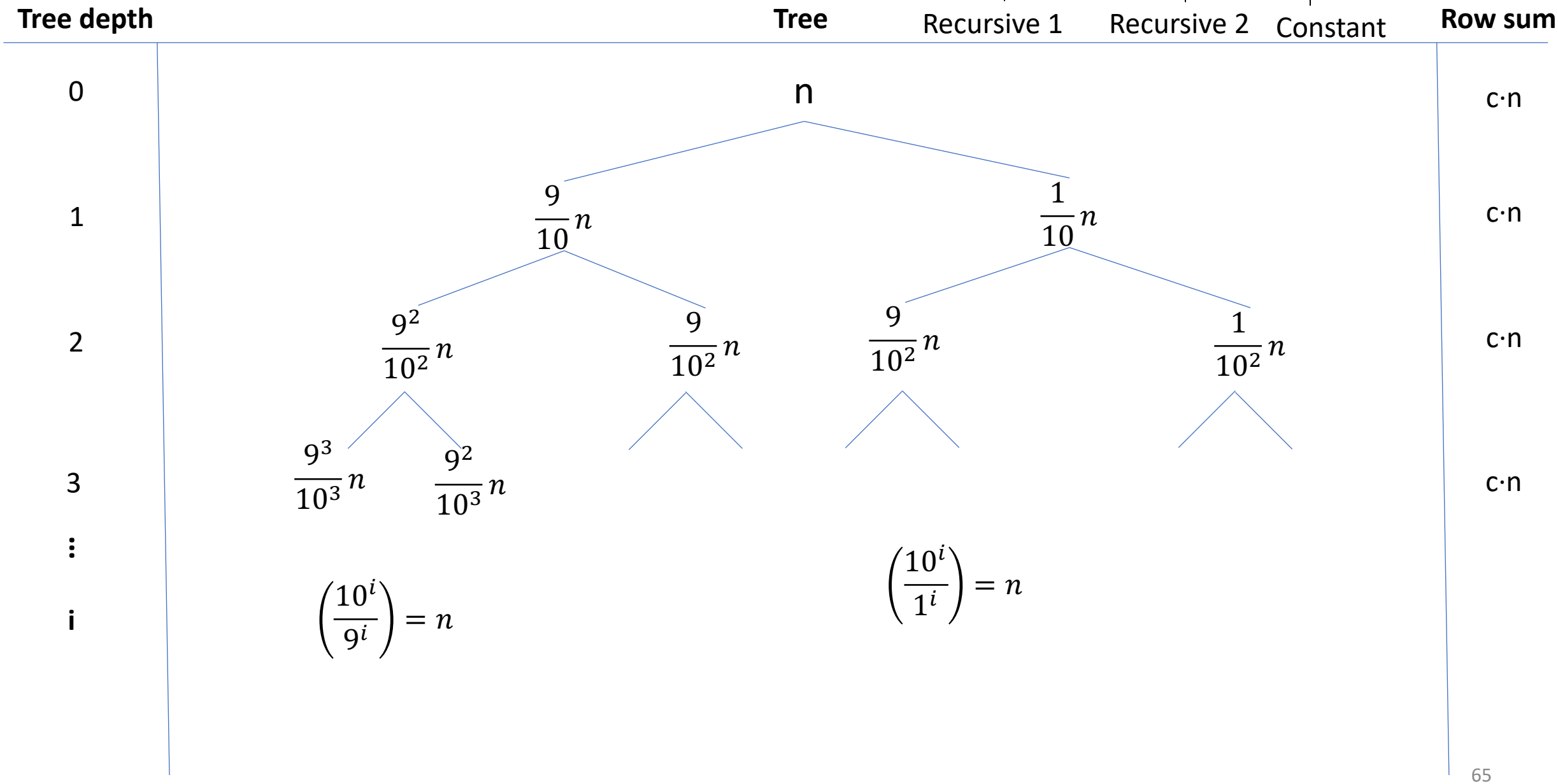


Lets now solve

$1 = \left(\frac{9^i n}{10^i}\right)$ $\quad$ $1 = \left(\frac{1^i n}{10^i}\right)$

$T(1) = 1$

$T(n) = \underbrace{T(9n/10)}_{\text{Recursive 1}} + \underbrace{T(n/10)}_{\text{Recursive 2}} + \underbrace{cn}_{\text{Constant}}$



$T(1) = 1$

$T(n) = \underbrace{T(9n/10)}_{\text{Recursive 1}} + \underbrace{T(n/10)}_{\text{Recursive 2}} + \underbrace{cn}_{\text{Constant}}$

Tree depth	Tree	Row sum
0	n	c·n
1	$\frac{9}{10}n$ $\quad$ $\frac{1}{10}n$	c·n
2	$\frac{9^2}{10^2}n$ $\quad$ $\frac{9}{10^2}n$ $\quad$ $\frac{9}{10^2}n$ $\quad$ $\frac{1}{10^2}n$	c·n
3	$\frac{9^3}{10^3}n$ $\quad$ $\frac{9^2}{10^3}n$	c·n
⋮		
i		

$$\left(\frac{10^i}{9^i}\right) = n$$

$$\left(\frac{10}{9}\right)^i = n$$

$$\left(\frac{10^i}{1^i}\right) = n$$

$$\left(\frac{10}{1}\right)^i = n$$

$T(1) = 1$

$T(n) = \underbrace{T(9n/10)}_{\text{Recursive 1}} + \underbrace{T(n/10)}_{\text{Recursive 2}} + \underbrace{cn}_{\text{Constant}}$

Tree depth	Tree	Row sum
0	n	c·n
1	$\frac{9}{10}n$ $\quad$ $\frac{1}{10}n$	c·n
2	$\frac{9^2}{10^2}n$ $\quad$ $\frac{9}{10^2}n$ $\quad$ $\frac{9}{10^2}n$ $\quad$ $\frac{1}{10^2}n$	c·n
3	$\frac{9^3}{10^3}n$ $\quad$ $\frac{9^2}{10^3}n$	c·n
⋮		
i		

$$\left(\frac{10^i}{9^i}\right) = n$$

$$\left(\frac{10}{9}\right)^i = n$$

$$\log_{\frac{10}{9}} n = i$$

$$\left(\frac{10^i}{1^i}\right) = n$$

$$\left(\frac{10}{1}\right)^i = n$$

$$\log_{10} n = i$$

$T(1) = 1$

$T(n) = \underbrace{T(9n/10)}_{\text{Recursive 1}} + \underbrace{T(n/10)}_{\text{Recursive 2}} + \underbrace{cn}_{\text{Constant}}$

Tree depth	Tree	Row sum
0	n	c·n
1	$\frac{9}{10}n$ $\quad$ $\frac{1}{10}n$	c·n
2	$\frac{9^2}{10^2}n$ $\quad$ $\frac{9}{10^2}n$ $\quad$ $\frac{9}{10^2}n$ $\quad$ $\frac{1}{10^2}n$	c·n
3	$\frac{9^3}{10^3}n$ $\quad$ $\frac{9^2}{10^3}n$	c·n
⋮		
i		

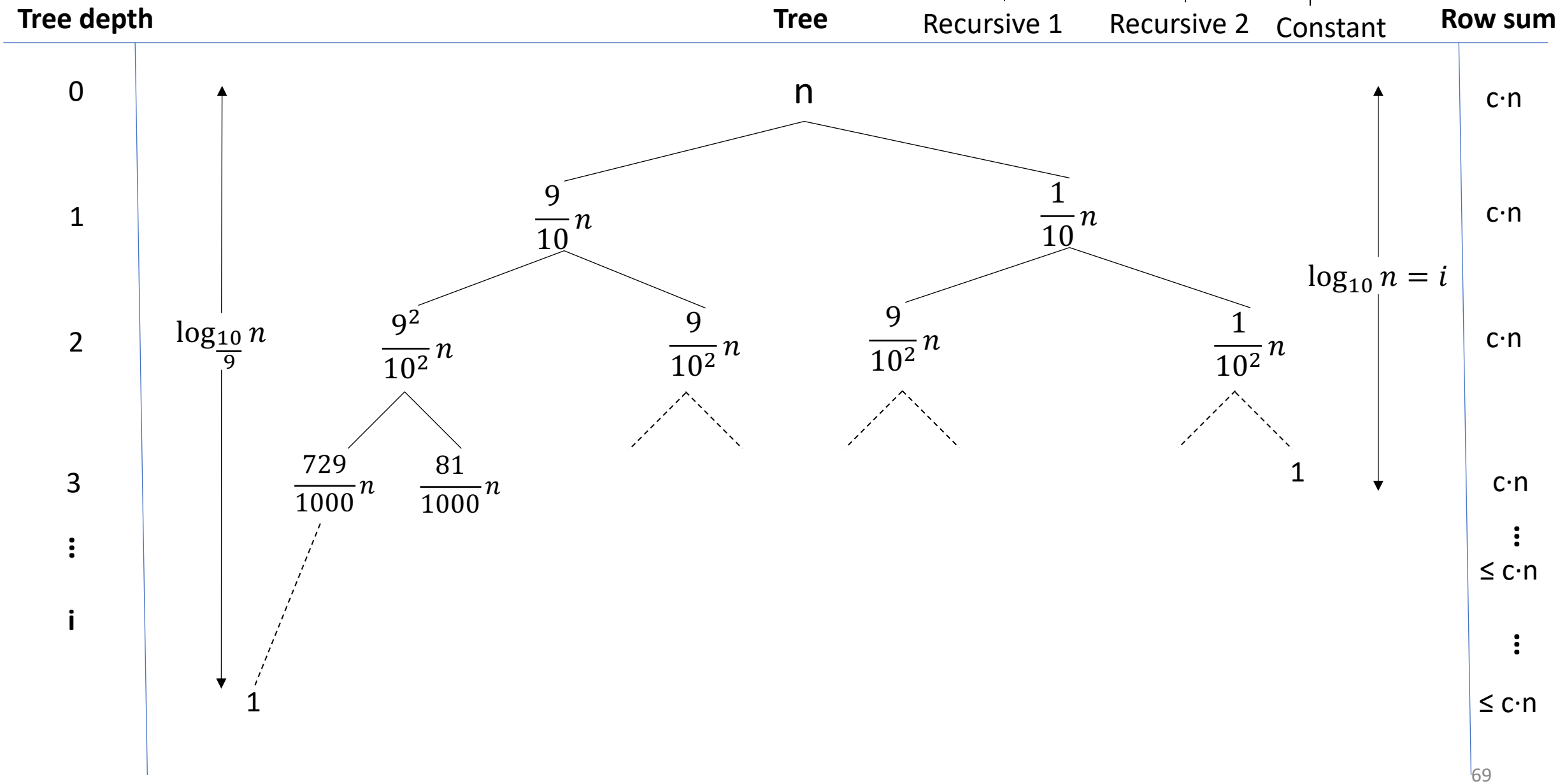
For our 90% splits, we reach base case at depth $\log_{\frac{10}{9}} n = i$

For our 10% splits, we reach base case at depth $\log_{10} n = i$

Lets update our tree to show these relationships…

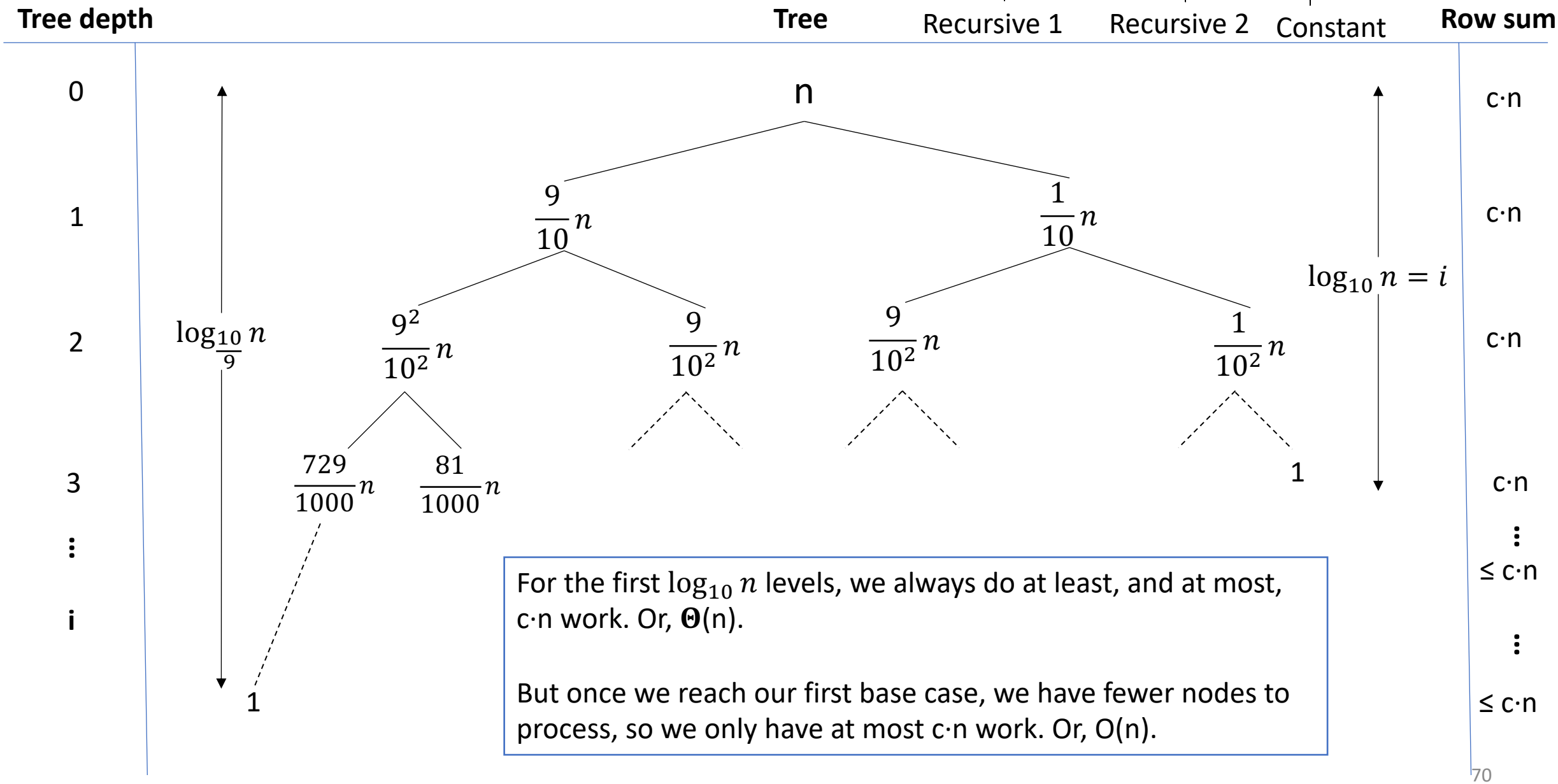
$T(1) = 1$

$T(n) = \underbrace{T(9n/10)}_{\text{Recursive 1}} + \underbrace{T(n/10)}_{\text{Recursive 2}} + \underbrace{cn}_{\text{Constant}}$



$T(1) = 1$

$T(n) = \underbrace{T(9n/10)}_{\text{Recursive 1}} + \underbrace{T(n/10)}_{\text{Recursive 2}} + \underbrace{cn}_{\text{Constant}}$

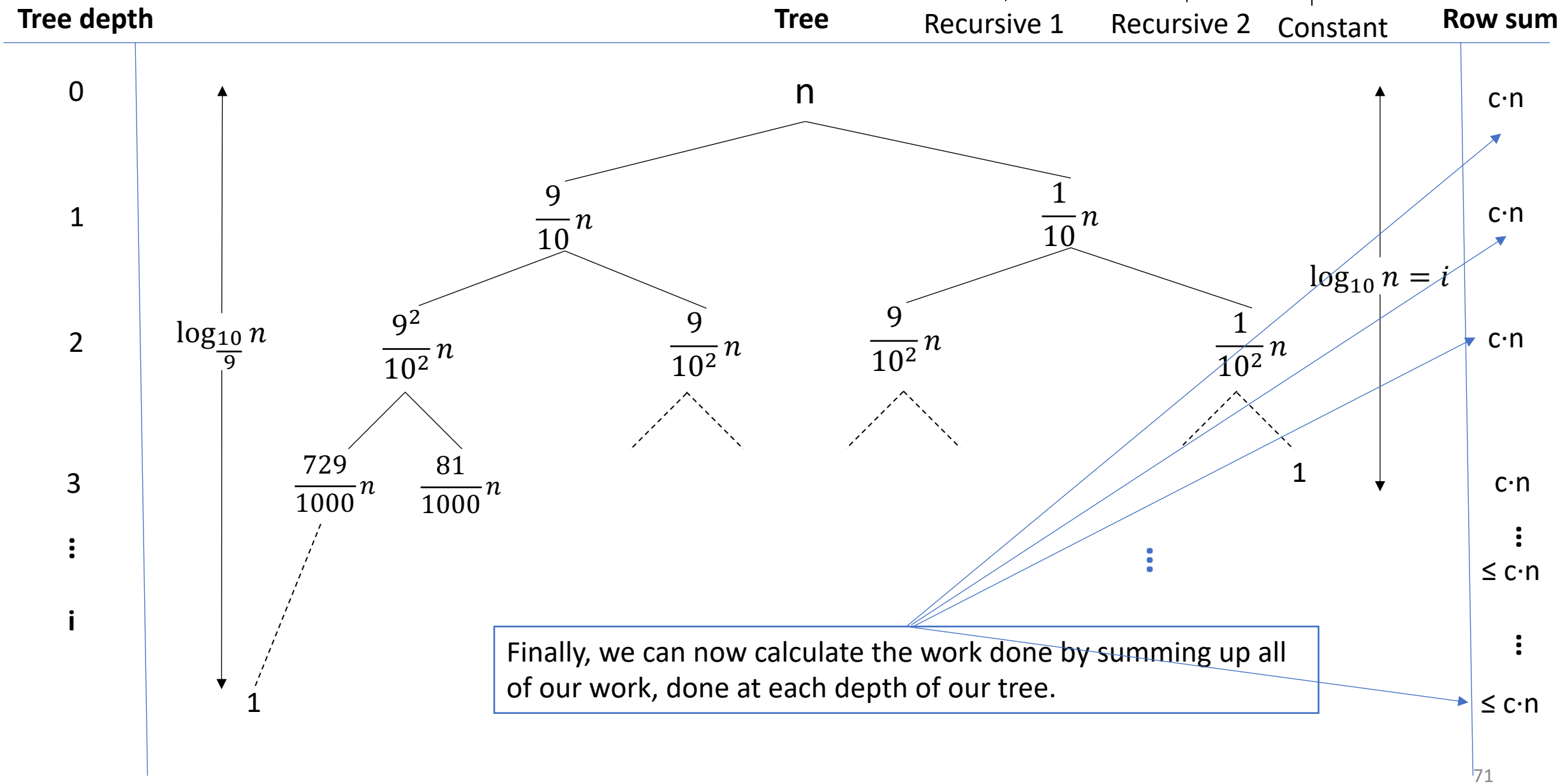


For the first $\log_{10} n$ levels, we always do at least, and at most, c·n work. Or, $\boldsymbol{\Theta}$(n).

But once we reach our first base case, we have fewer nodes to process, so we only have at most c·n work. Or, O(n).

$T(1) = 1$

$T(n) = \underbrace{T(9n/10)}_{\text{Recursive 1}} + \underbrace{T(n/10)}_{\text{Recursive 2}} + \underbrace{cn}_{\text{Constant}}$



Calculating our total work

We now sum all the work done at each tree depth to get the total cost of quicksort average case.

Since we have $\log_{\frac{10}{9}} n$ levels, each doing ‘n’ work, we have:

$O\left(n \times \log_{\frac{10}{9}} n\right)$ work.

But the base of our log is messy for asymptotic notation. Let use the following mathematical fact: $\log_a n = \frac{\log_b n}{\log_b a}$

Calculating our total work

Using this fact, our equation can be rewritten as:

$$\log_{\frac{10}{9}} n = \frac{\log_2 n}{\log_2 \frac{10}{9}} = c \cdot \log_2 n$$

Since $1/\log_2 \frac{10}{9}$ is a constant, we replace it with c. As long as this value is a constant, the log doesn't matter in asymptotic notation.

Again, we do 'n' work per level, so rewriting $\mathrm{O}\left(n \times \log_{\frac{10}{9}} n\right)$ we get:

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

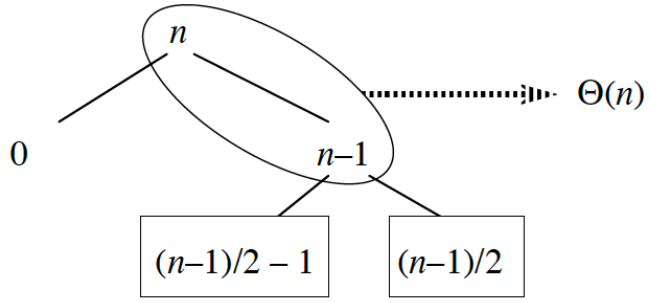
Intuition for average case

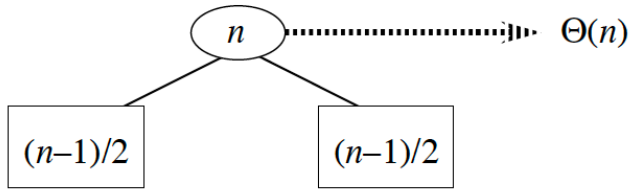
Splits in the recursion tree will not always be constant.

What about alternating a perfect split with a bad split?

In this case, the tree is at most twice as high as for all perfect splits, so still O(n log n + n)

Both figures result in O(n $\log_2$n) time, though the constant for the figure on the left is higher than that of the figure on the right.





Suggested reading

Chapter 7 covers quicksort. Today's material is covered in sections 7.1 and 7.2, and are a "must read".

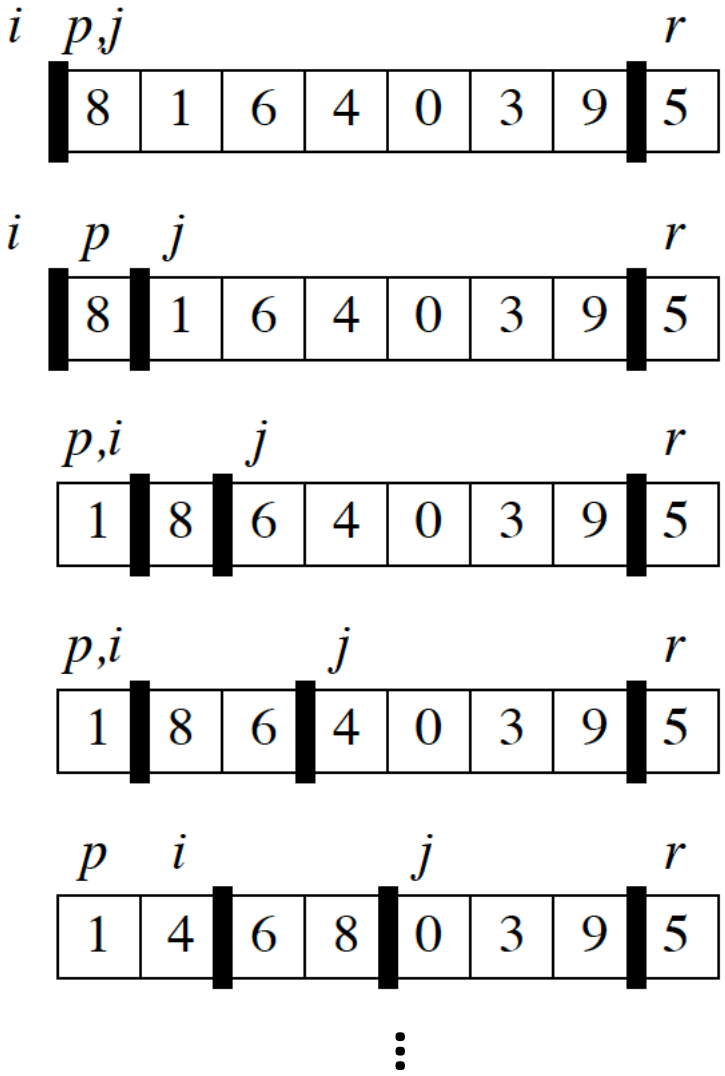
Recurrence trees[2] are covered in section 4.4.

References

1. The first author of the text has a gentle introduction to quicksort on Kahn academy.
2. Dr Bower's video of solving recurrence trees is also helpful.

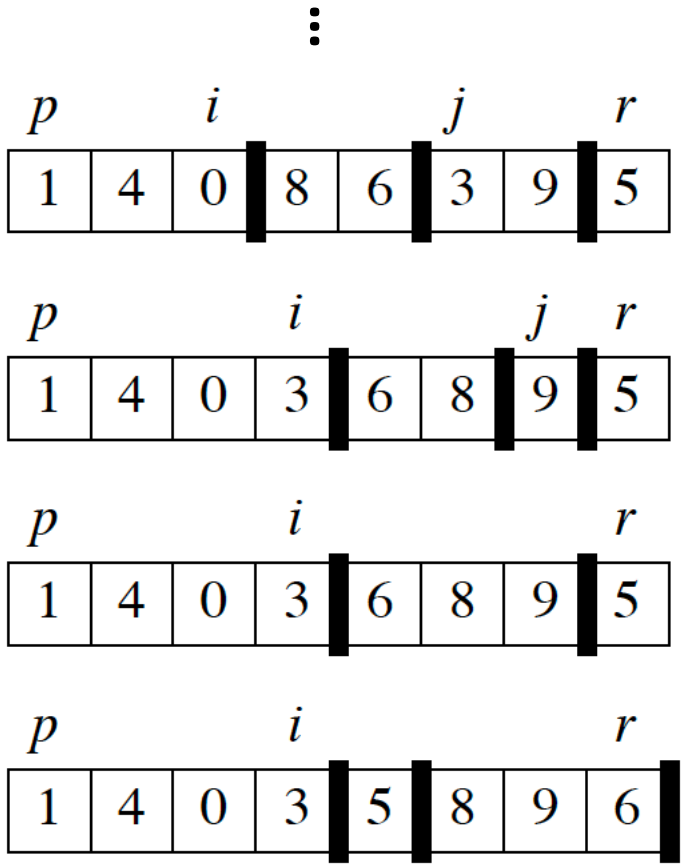
Solutions

Class challenge 1




```
PARTITION(A, p, r)
   x = A[r]
   i = p - 1
   for j = p to r - 1
    if A[j] ≤ x
        i = i + 1
        swap A[i] with A[j]
   swap A[i+1] with A[r]
   return i + 1
```

Class challenge 1




```
PARTITION(A, p, r)
   x = A[r]
   i = p - 1
   for j = p to r - 1
    if A[j] ≤ x
        i = i + 1
        swap A[i] with A[j]
   swap A[i+1] with A[r]
   return i + 1
```

For loop exits, no need for index *j*

Image attributions