Introduction to Divide and Conquer

Divide and Conquer

- Break a problem into similar subproblems
- Solve each subproblem recursively
- Combine

Above its weight class

- Divide and conquer is a very simple idea
- But it has far more than its share of the ``miraculous algorithms''
- Strassen Matrix Multiplication and improvements
- Karatsuba multiplication
- Fast Fourier Transform
- Linear time select

Merge sort: quick review

Merge(A[1..n], B[1..n]): linear time, combines two sorted lists

```
I:=1 ; J:=1
FOR k=1 TO 2n do:
    IF I > n THEN C[k]:= B[J]; J++
    ELSE IF J > n THEN C[k]:= A[I];I++
     ELSE IF A[I] > B[J] THEN C[k]:=B[J]; J++
        ELSE C[k]:=A[I];I++
 Return C
```

MergeSort (A[1..n])

```
IF n=1 return A[1]
ELSE Return Merge (MergeSort(A[1..n/2]),MergeSort (A[n/2+1…n])
```

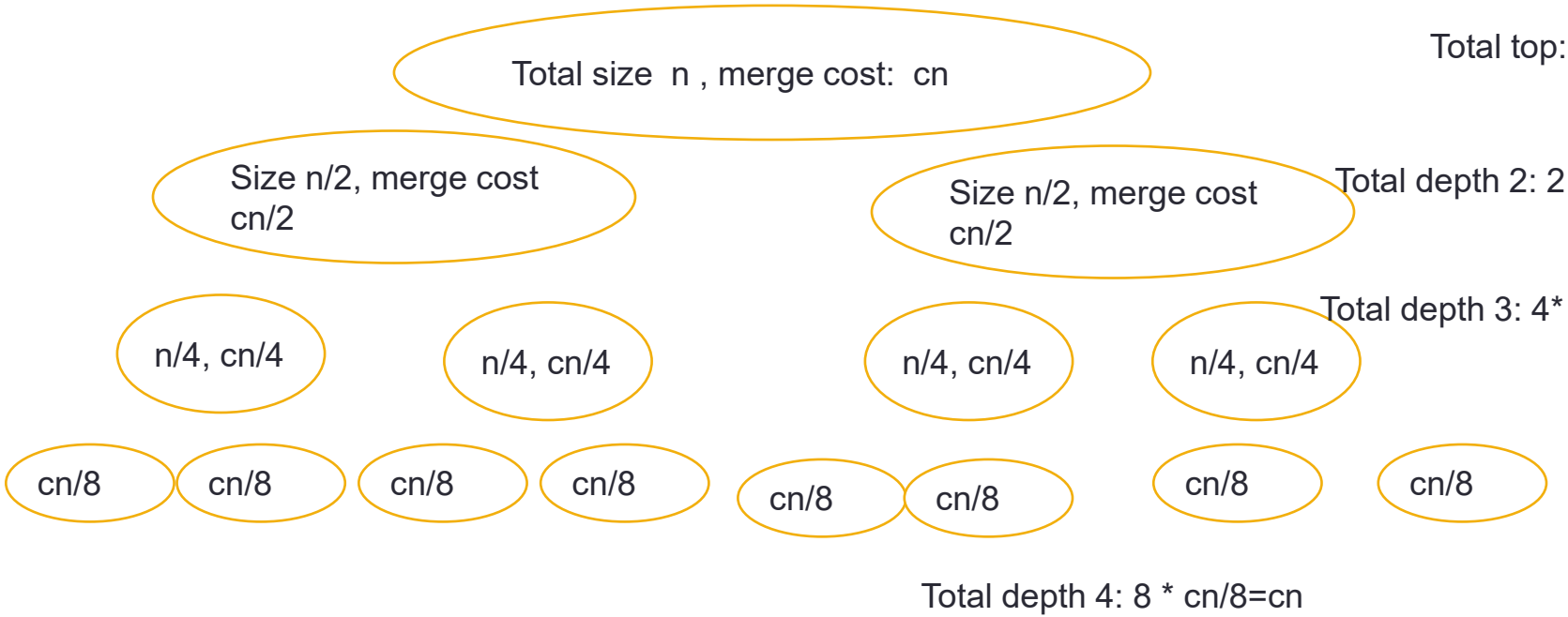
Time analysis? Start with recurrence, then solve.

Time analysis

- Given by recurrence: $T(n) = 2\,T(n/2) + O(n)$

- Can picture as tree of recursive calls, with Merge times at each

Tree of recursive calls



Time analysis cont

- Depth of tree= $\log_2 n$
- Each level, total work is O(n)
- Therefore, T(n) is O(n log n)

Multiplying Binary numbers

				1	1	0	1	
			×	1	0	1	1	
				1	1	0	1	(1101 times 1)
			1	1	0	1		(1101 times 1, shifted once)
		0	0	0	0			(1101 times 0, shifted twice)
+	1	1	0	1				(1101 times 1, shifted thrice)
1	0	0	0	1	1	1	1	(binary 143)

Divide and conquer multiply

- Say we want to multiply 10100110 and 10110011

- How can we divide the problem into sub-problems?
- Remember, we want much smaller sub-problems

Multiplying large binary numbers

- 10100110 = 166= 1010 * 2^4 + 0110 = 10*16 + 6
- 10110011 = 179= 1011*$2^4 + 0011 = 11$*16 + 3
- 10100110*10110011 = (10*16+6)(11*16+3)= 110*256 + 6*11*16 + 3*10*16 + 3*6

Multiplying Binary numbers (DC)

- Suppose we want to multiply two n-bit numbers together where n is a power of 2.
- One way we can do this is by splitting each number into their left and right halves which are each n/2 bits long

- x= 

- y= 

Multiplying Binary numbers (DC)

- Suppose we want to multiply two n-bit numbers together where n is a power of 2.
- One way we can do this is by splitting each number into their left and right halves which are each n/2 bits long

- $x = 2^{n/2}xL + xR$
- $y = 2^{n/2}yL + yR$

Multiplying Binary numbers (DC)

$$x = 2^{n/2}x_L + x_R$$
$$y = 2^{n/2}y_L + y_R$$

- $xy = \left(2^{\frac{n}{2}}x_L + x_R\right)\left(2^{\frac{n}{2}}y_L + y_R\right)$
- $xy = 2^n x_L y_L + 2^{\frac{n}{2}}(x_L y_R + x_R y_L) + x_R y_R$

Algorithm multiply

function **multiply** (x,y)

Input: n-bit integers x and y

Output: the product xy

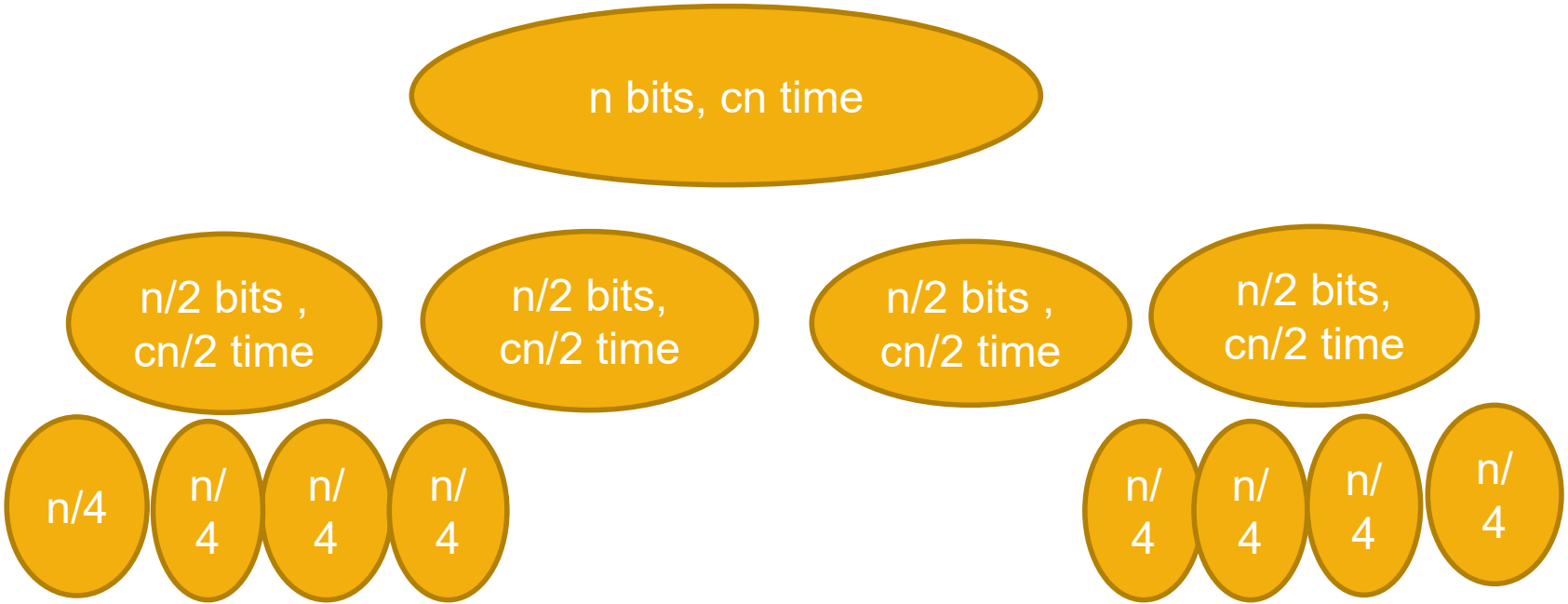
- If n=1: return xy
- x_L, x_R and y_L, y_R are the left- and right-most n/2 bits of x and y, respectively.
- $P_1 = \mathbf{multiply}(x_L, y_L)$
- $P_2 = \mathbf{multiply}(x_L, y_R)$
- $P_3 = \mathbf{multiply}(x_R, y_L)$
- $P_4 = \mathbf{multiply}(x_R, y_R)$
- return $P_1 * 2^n + (P_2 + P_3) * 2^{\frac{n}{2}} + P_4$

Algorithm

- Runtime
- Let T(n) be the runtime of the multiply algorithm.

- Then $T(n) = 4T\left(\frac{n}{2}\right) + O(n)$

Total time


n bits, cn time
n/2 bits , cn/2 time
n/2 bits, cn/2 time
n/2 bits , cn/2 time
n/2 bits, cn/2 time
n/4
n/ 4
n/ 4
n/ 4
n/ 4
n/ 4
n/ 4
n/ 4

Total

- One top level : cn
- Four depth 1: cn/2 *4
- 16 depth 2: cn/4 * 16
- 64 depth 3: cn/8 * 64

….

4^t $depth\ t : \frac{cn}{2^t} * 4^t = cn * 2^t$

Max level : t= log n, c1 $*4^{logn} = c * 2^{logn} * 2^{logn} = cn^2$

Total time

- cn (1+ 2 +4 +8+…2^{logn}) = $O(cn^2)$
- Because in a geometric series with ratio other than 1, largest term dominates order.

Multiplication

Andrey Kolmogorov 1903 - 1987

Anatoly Karatsuba 1937 - 2008

Insight: replace one (of the 4) multiplications by (linear time) subtraction

Multiplying Binomials

- if you want to multiply two binomials
- $(ax + b)(cx + d) = acx^2 + adx + bcx + bd$
- It requires ? 4 multiplications. ac, ad, bc, bd

Multiplying Binomials

- If you want to multiply two binomials
- $(ax + b)(cx + d) = acx^2 + (ad + bc)x + bd$
- It seems to require 4 multiplications: ac, ad, bc, bd
- If we assume that addition is cheap (has short runtime), then we can improve this by only doing 3 multiplications:

$$ac, bd, (a + b)(c + d)$$

"Magic" formula: $ad + bc = (a + b)(c + d) - ac - bd = (ac + bc + ad + bd) - ac - bd.$

Multiplying Binomials

- Reducing the number of multiplications from 4 to 3 may not seem very impressive when calculating asymptotics.

- If this was only a part of a bigger algorithm, it may be an improvement.

Algorithm multiply KS

function **multiplyKS** (x,y)

Input: n-bit integers x and y

Output: the product xy

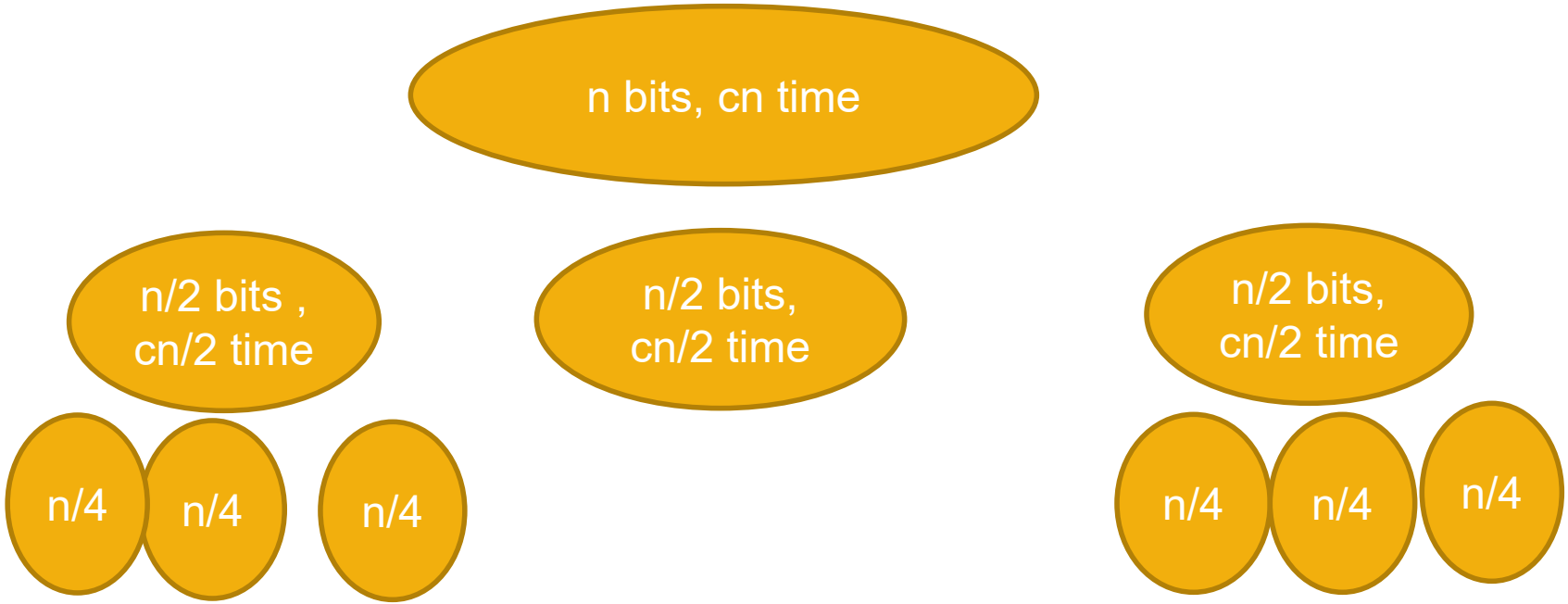
- If n=1: return xy
- x_L, x_R and y_L, y_R are the left- and right-most n/2 bits of x and y, respectively.
- $R_1 = \mathbf{multiplyKS}(x_L, y_L)$
- $R_2 = \mathbf{multiplyKS}(x_R, y_R)$
- $R_3 = \mathbf{multiplyKS}\big((x_L + x_R)(y_L + y_R)\big)$
- return $R_1 * 2^n + (R_3 - R_1 - R_2) * 2^{\frac{n}{2}} + R_2$

Algorithm multiplyKS

- Runtime
- Let T(n) be the runtime of the multiply algorithm.

- Then
- $T(n) = 3T\left(\frac{n}{2}\right) + O(n)$

Total time


n bits, cn time
n/2 bits , cn/2 time
n/2 bits, cn/2 time
n/2 bits, cn/2 time
n/4
n/4
n/4
n/4
n/4
n/4

3 vs 4

- Since we are pruning the tree recursively, replacing 4 recursive calls instead of 3 reduces the size of the tree more than a constant factor.

Total

- One top level : cn
- Four depth 1: cn/2 *3
- 9 depth 2: cn/4 * 9
- 27 depth 3: cn/8 * 27

….

$$depth\ t : \frac{cn}{2^t} * 3^t = cn * (1.5)^t$$

Max level : t= log n,

Total time

- cn (1+ 1.5 +2.25 +…$(1.5)^{\log n}$) = $O\left(3^{\log n}\right)$
- Because in a geometric series with ratio other than 1, largest term dominates order.

- But what is $3^{\log n}$?

Simplifying

- $3^{\log n} = \left(2^{\log 3}\right)^{\log n} = 2^{\{\log n * \log 3\}} = \left(2^{\{\log n\}}\right)^{\log 3} = n^{\log 3} = n^{1.58\ldots}$

- So total time is $O\left(n^{\log_2 3}\right)$ =O($n^{1.58\ldots}$)

Power of 3

- Say we want to compute 3^n in binary.
- Becomes a large number, between n and 2n bits to write down
- We shouldn't think of arithmetic as constant time, but we can use the improved multiplication algorithm as a sub-routine.

- What are some divide and conquer recursions for this problem?

Power of 3

- $3^n = 3 * 3^{n-1}$. But this won't be a very fast recursion, because problem stays nearly same size
- If $n\ is\ even, 3^n = 3^{n/2} * 3^{n/2}$
- If n is odd $3^n = 3 * 3^{(n-1)/2} * 3^{(n-1)/2}$

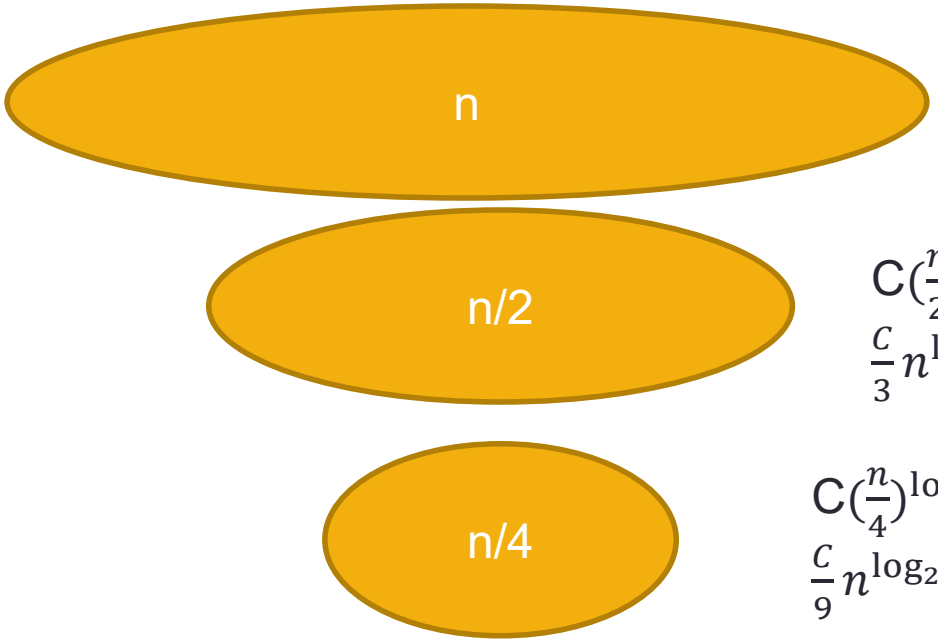
As algorithm

Powerof3(n)

- If n=0 return 1; If n=1, return ``11”
- R = Powerof3 (n div 2)
- If n is even:
  - return multiplyKS (R, R)
- Else:
  - return basicMult (3, multiplyKS(R,R))

- T(n) = ??

As tree: just path



$Cn^{\log_2 3}$

$C(\frac{n}{2})^{\log_2 3} = \frac{C}{3}n^{\log_2 3}$

$C(\frac{n}{4})^{\log_2 3} = \frac{C}{9}n^{\log_2 3}$

Work dominated by top level

- The total time at all recursion levels except the top is less than that at the top level.

- So total time is $O(n^{\log_2 3})$, same as one Karatsuba multiply

Commonalities and differences

In the three examples we've seen, the time at each level is a **geometric series**.

- Sometimes it is increasing (Karatsuba Multiplication)
- Sometimes it stays the same (mergesort)
- Sometimes it is decreasing (Power of 3)

Master Theorem

- How do you solve a recurrence of the form

$$T(n) = aT\left(\frac{n}{b}\right) + O\left(n^d\right)$$

We will use the master theorem.

The master theorem gives a general rule for which of those three cases we are in, and works out the results for each case, so we don't have to map out the tree and compute the sum each time.

Summation Lemma

Consider the summation

$$\sum_{k=0}^{n} r^k$$

It behaves differently for different values of r.

Summation Lemma

Consider the summation

$$\sum_{k=0}^{n} r^k$$

It behaves differently for different values of r.

If $r < 1$ then this sum converges. This means that the sum is bounded above by some constant c. Therefore

$$if\ \ r < 1\ \ then \sum_{k=0}^{n} r^k < c\ \ for\ \ all\ \ n\ \ so\ \ \sum_{k=0}^{n} r^k \in O(1)$$

Summation Lemma

Consider the summation

$$\sum_{k=0}^{n} r^k$$

It behaves differently for different values of r.

If $r = 1$ then this sum is just summing 1 over and over n times. Therefore

$$if \;\; r = 1 \;\; \sum_{k=0}^{n} r^k = \sum_{k=0}^{n} 1 = n + 1 \;\; \epsilon \;\; O(n)$$

Summation Lemma

Consider the summation

$$\sum_{k=0}^{n} r^k$$

It behaves differently for different values of r.

If $r > 1$ then this sum is exponential with base r.

$$if\ \ r > 1\ \sum_{k=0}^{n} r^k < cr^n\ for\ all\ n, \qquad so \sum_{k=0}^{n} r^k \in\ \ O(r^n) \qquad \left(c > \frac{r}{r-1}\right)$$

Summation Lemma

Consider the summation

$$\sum_{k=0}^{n} r^k$$

It behaves differently for different values of r.

$$\sum_{k=0}^{n} r^k \in \begin{cases} O(1) & if \ \ r < 1 \\ O(n) & if \ \ r = 1 \\ O(r^n) & if \ \ r > 1 \end{cases}$$

Master Theorem

Master Theorem: If $T(n) = aT(n/b) + O(n^d)$ for some constants $a > 0, b > 1, d \geq 0$,

Then

$$T(n) \in \begin{cases} O(n^d) & if \ a < b^d \\ O(n^d \log n) & if \ a = b^d \\ O(n^{\log_b a}) & if \ a > b^d \end{cases}$$

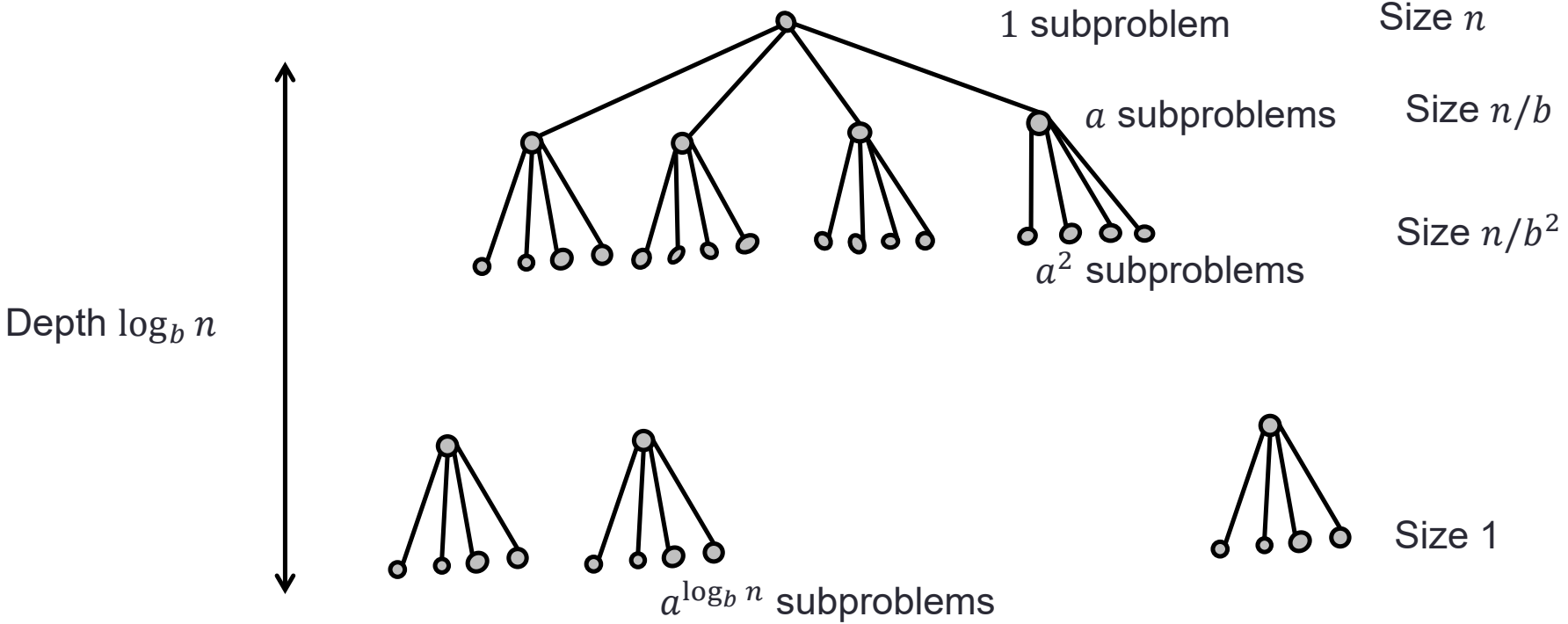
Top-heavy

Steady state

Bottom heavy

Master Theorem: Solving the recurrence

$$T(n) = aT(n/b) + O(n^d)$$



Master Theorem: Solving the recurrence

After k levels, there are a^k subproblems, each of size n/b^k.

So, during the kth level of recursion, the time complexity is
$O\left(\left(\frac{n}{b^k}\right)^d\right)a^k = O\left(a^k\left(\frac{n}{b^k}\right)^d\right) = O\left(n^d\left(\frac{a}{b^d}\right)^k\right)$.

Geometric series with r= a/b^d. Three cases based on comparing r to 1.

Master Theorem: Proof

$$T(n) = O\left(n^d \sum_{k=0}^{\log_b n} \left(\frac{a}{b^d}\right)^k\right)$$

Case 1: $a < b^d$

Then we have that $\frac{a}{b^d} < 1$ and the series converges to a constant so most of the work is done at top-level (e.g., Powerof3)

$$T(n) = O\left(n^d\right)$$

Master Theorem: Proof

$$T(n) = O\left(n^d \sum_{k=0}^{\log_b n} \left(\frac{a}{b^d}\right)^k\right)$$

Case 2: $a = b^d$

Then we have that $\frac{a}{b^d} = 1$ and so each of O(logn) levels contributes the same work total.

$T(n) = O\left(n^d \log_b n\right)$= O($n^d \log n / \log b$) = $O(n^d \log n)$. (Need b > 1, a constant)

Master Theorem: Proof

$$T(n) = O\left(n^d \sum_{k=0}^{\log_b n} \left(\frac{a}{b^d}\right)^k\right)$$

Case 2: $a > b^d$

The main work is done at the last level, the base cases. There are

$$a^{\log_b n} \text{ base cases, because we have an a} - \text{ary tree of height } \log_b n$$

Using the same algebraic manipulations we did for Multiplication,

$$a^{\log_b n} = \left(b^{\log_b a}\right)^{\log_b n} = \left(b^{\log_b n}\right)^{\log_b a} = n^{\log_b a}$$

Master Theorem

Theorem: If $T(n) = aT(n/b) + O(n^d)$ for some constants $a > 0, b > 1, d \geq 0$,

Then

$$T(n) \in \begin{cases} O(n^d) & if \ a < b^d \\ O(n^d \log n) & if \ a = b^d \\ O(n^{\log_b a}) & if \ a > b^d \end{cases}$$

Top-heavy

Steady-state

Bottom-heavy

Master Theorem Applied to Multiply

$$T(n) \epsilon \begin{cases} O(n^d) & if \ \ a < b^d \\ O(n^d \log n) & if \ \ a = b^d \\ O(n^{\log_b a}) & if \ \ a > b^d \end{cases}$$

The recursion for the runtime of Multiply is
T(n) = 4T(n/2) + cn

So we have that a=4, b=2, and d=1. In this case, $a > b^d$ so

$$T(n) \epsilon O(n^{\log_2 4}) = O(n^2)$$

Not any improvement of grade-school method.

Master Theorem Applied to MultiplyKS

$$T(n)\epsilon\begin{cases} O(n^d) & if\ \ a < b^d \\ O(n^d \log n) & if\ \ a = b^d \\ O(n^{\log_b a}) & if\ \ a > b^d \end{cases}$$

The recursion for the runtime of Multiply is
T(n) = 3T(n/2) + cn

So we have that a=3, b=2, and d=1. In this case, $a > b^d$ so

$$T(n)\epsilon O\left(n^{\log_2 3}\right) = O(n^{1.58})$$

An improvement on grade-school method!!!!!!

Poll: What is the fastest known integer multiplication time?

- Option A: $O\left(n^{\log 3}\right)$
- Option B: $O(n \log n \ (\log(\log n)\ )^2)$
- Option C: $O(n \log n \ 2\text{^}\{\log^* n\})$
- Option D: $O(n \log n)$
- Option E: $O(n)$

Poll: What is the fastest known integer multiplication time? All have/will be correct

- $O(n^{\log 3})$ Karatsuba, 1960
- $O(n \log n \ \log\log n\ )$ Schönhage and Strassen, 1971
- $O(n \log n \ 2\text{^}\{c \log^* n\})$ Fürer, 2007
- $O(n \log n)$ Harvey and van der Hoeven, 2019
- $O(n)$, you, tomorrow?