Colorado State University

CS 320 Fall 2023
Solving recurrences for Divide & Conquer

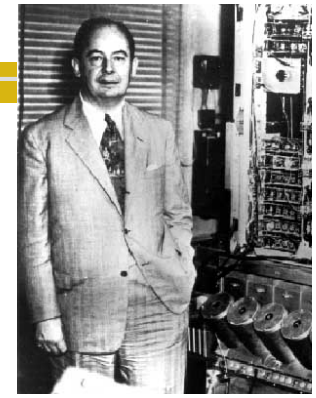
Sanjay Rajopadhye
Colorado State University

Divide & Conquer

- Break up the problem into (multiple, smaller) parts
- Solve each of the parts recursively
- Combine the solution of each of the parts into a solution of the original problem

First example: Merge sort

- Divide the array into two halves
- Recursively sort each half
- Merge the two sorted halves



John von Neumann (1945)

Analysis

Divide $O(1)$

Merge $O(n)$

What about the recursive calls?

$2\,T\left(\frac{n}{2}\right)$

A	L	G	O	R	I	T	H	M	S

A	L	G	O	R

I	T	H	M	S

A	G	L	O	R

H	I	M	S	T

A	G	H	I	L	M	O	R	S	T

Complexity of merge

- Time: $O(n)$
- Space: $O(n)$
  - Often with two arrays of length n
  - Can you do (a constant factor) better?

Recurrence relations

- A recurrence relation for a sequence, $\{a_n\}$ is and equation that expresses a_n in terms of one or more of the previous elements of the sequence, $a_1, a_2, \ldots\ a_{n-1}$
- A special kind of recursive function
- There may be *base cases*, and the equation hold for $n \geq n_0$ for some constant n_0
  - Example: $a_n = 2a_{n-1} + 1$ and $a_1 = 1$
  - After setting up the recurrence relation, we *solve* it

Recurrence relation for Merge-sort

- *Define* the number of comparisons to sort an input of length n as: $T(n)$
- Use the *structure* of the D&C algorithm to define an equation/relation for $T(n)$

$$T(n) \leq \begin{cases} c & \text{if } n = 1 \\ T\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + T\left(\left\lceil \frac{n}{2} \right\rceil\right) + cn & \text{otherwise} \end{cases}$$

Solving the Recurrence

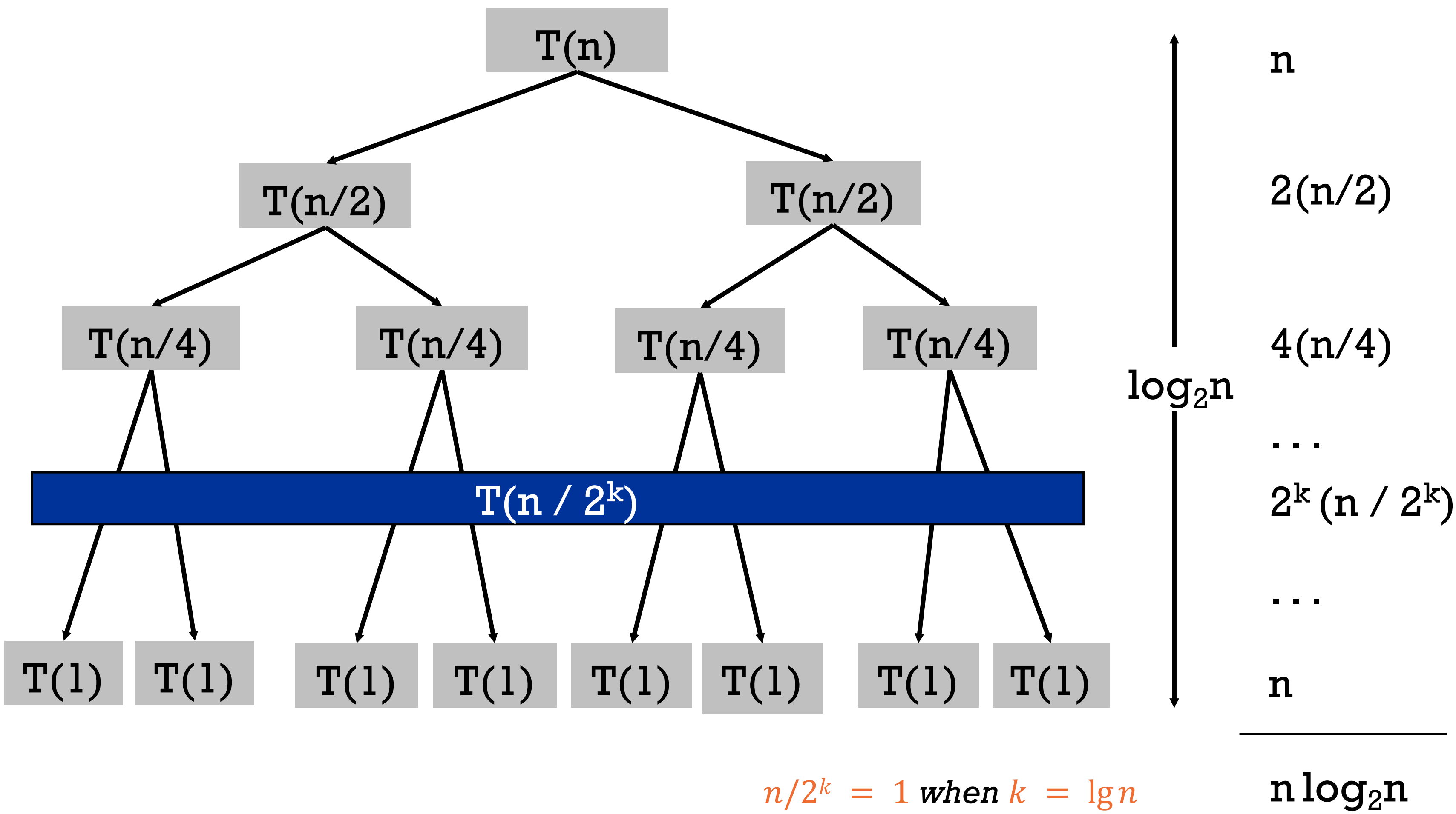
$$T(n) = \begin{cases} c & \text{if } n = 1 \\ 2T\left(\frac{n}{2}\right) + cn & \text{otherwise} \end{cases}$$

- Solution (*closed form*):

$$T(n) = \Theta(n \log n)$$

- Number of techniques
  - Unrolling the recurrence
  - Repeated substitution
  - See a pattern, guess (i.e., make a hypothesis), and then, prove by induction

Unroll $T(n) = \begin{cases} c & \text{if } n = 1 \\ 2T\left(\frac{n}{2}\right) + cn & \text{otherwise} \end{cases}$



$n/2^k = 1$ *when* $k = \lg n$

Seeing the pattern

- What is the "*label*" of each node?
- When does the label become "*small enough*" (base case)
- How many levels in the tree? [Hint: use the above two]
- How many nodes at each level?
- What is the "*contribution*" of each node?
- What is the contribution of *each level*?
- How many *leaves*?
- *Contribution of the leaves* (different from contribution of other levels)

Repeated substitution for $T(n) = \begin{cases} c & \text{if } n = 1 \\ 2T\left(\frac{n}{2}\right) + cn & \text{otherwise} \end{cases}$

- **Claim:** $T(n) = cn \log_2 n$

$$
\begin{aligned}
T(n) &= 2T(n/2) && + cn \\
&= 4T(n/4) && + cn + 2cn/2 \\
&= 8T(n/8) && + cn + cn + 4cn/4 \\
&\cdots \\
&= 2^{\log_2 n} T(1) && + \underbrace{cn + \cdots + cn}_{\log_2 n} \\
&= O(n \log_2 n)
\end{aligned}
$$

This reaches $T(1)$ when
$n = 2^{\lg n}$
by definition of $\lg n$

Binary search

```
function BS(x, start, end)
  if (end <= start)
    return A[start]
  mid = (end + start)/2
  if A[mid] < x
    return BS(x, mid, end)
  return BS(x, start, mid-1)
```

- What is the recurrence?
- Apply repeated substitution (on doc cam or exercise)

Find max in an unsorted array

Algorithm:

- Base case n=1
- Otherwise: find the max of the two halves, and return the max of that

```
function FM(start, end)
  if (end = start)
    return A[start]
  mid = (end + start)/2
  return max( FM(start, mid-1), FM(mid, end) )
```

Find max in an unsorted array

Recurrence: base case: $T(1) = 0$

Otherwise: $T(n) = 2T\left(\frac{n}{2}\right) + 1$

$$\begin{aligned}
&= 4T\left(\frac{n}{4}\right) + 2 + 1 \\
&= 8T\left(\frac{n}{8}\right) + 4 + 2 + 1 \\
&\vdots \\
&= 2^k T\left(\frac{n}{2^k}\right) + 2^{k-1} + 2^{k-2} + \cdots 2^0 \\
&= 2^k T\left(\frac{n}{2^k}\right) + 2 \cdot 2^{k-1} - 1 \\
&= 2^k T\left(\frac{n}{2^k}\right) + 2^k - 1
\end{aligned}$$

Bae case is reached when $2^k = n$, i.e., $k = \lg n$, So

$T(n) = 0 + 2^{\lg n} - 1 = n - 1$

Another example

```
function foo(A, B) // the size of A is n
  if (n == 1):
  return fuzz(A, B) // base case, fuzz is
constant time

// Process A to build two parts, A0 and A1 of
size n/2 each

C0 = foo (A0, B)
C1 = foo (A0, B)
return buzz(C0, C1) // buzz is O(n^2)
```

General Divide & Conquer

```
function foo(parameters) // the size of A is n
  if (n <= b):                // base case
      return fuzz(A, B)  // constant time
// Divide input into a parts, each of size n/b
  divide()
// Make a calls to
  foo(new parameters) // size is n/b
  return combine(r_1, …, r_a)
// Complexity of divide and combine is O(n^d)
```

Master Theorem

- Let $a \geq 1, b > 1, n = b^k$ and $T(n)$ be given by

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

- The solution of the recurrence is

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

Merge-sort by master theorem

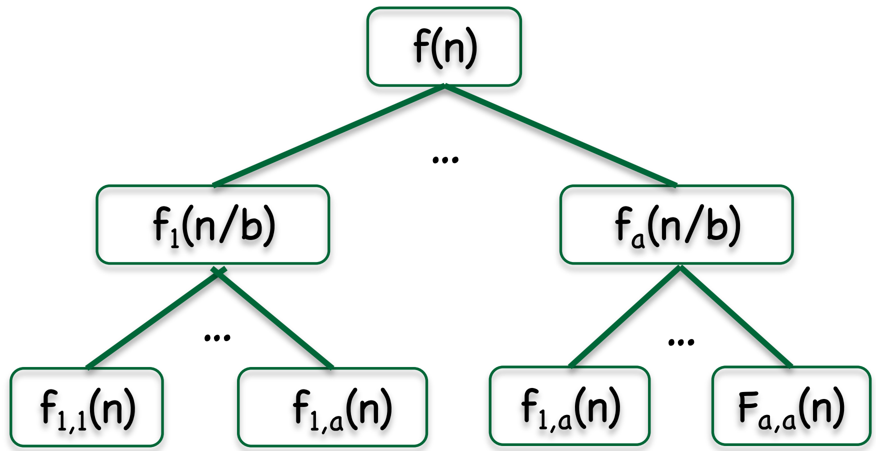
- $a = 2, b = 2, d = 1$
- So, $a = 2$, and $b^d = 2$

… and the solution is

$$T(n) = O(n^d \log n) = O(n \log n)$$

Divide & Conquer call tree

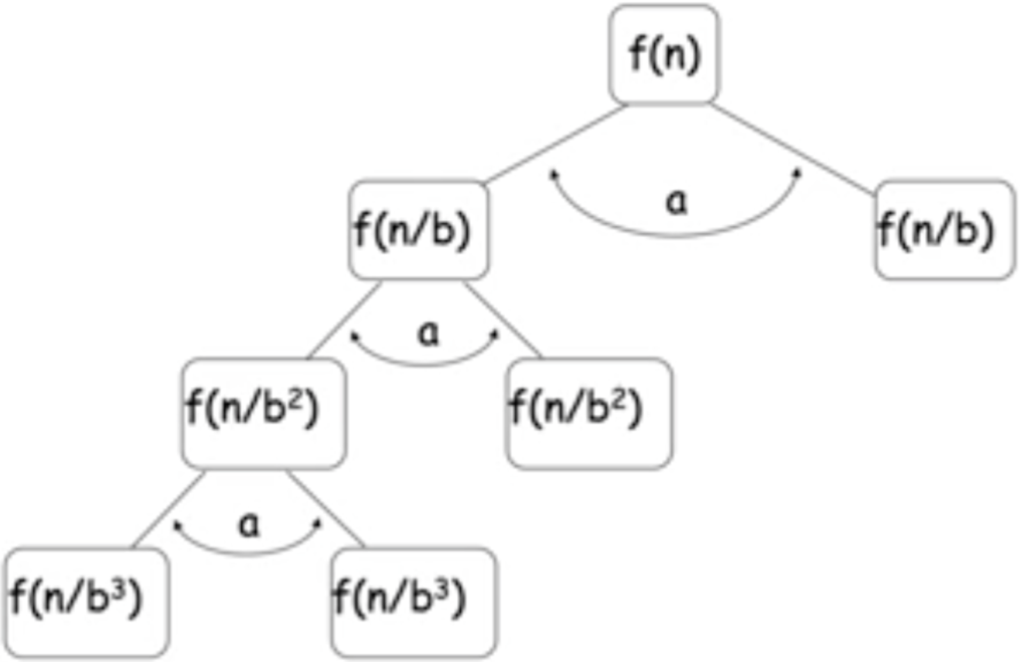
```
Function foo(A) //size n
  if (n <= b) return (base(A)
A₁ … Aₐ = divide() // size n/b
// Recurse
  C₁ = foo(A₁)
  ⋮
  Cₐ = foo(Aₐ)
return combine(C₁, …, Cₐ)
```



- Base is constant time
- Divide and combine takes $O(n^d)$

$$f(n) = af(n/b) + n^d$$

$f(1) = c$ ← does not play a role, as we only care about O



→ n^d

→ $a(n/b)^d$ or $(a/b^d)\, n^d$

→ $(a/\, b^d)^2\, n^d$

→ $(a/\, b^d)^3\, n^d$

Level i: a^i calls of $f(n/d^i)$ → $(a/\, b^d)^i\, n^d$

+

Stops when $n/b^i = 1$ i.e. $i = \log_b n$

$$n^d \sum_{i=0}^{\log_b n} (a/b^d)^i$$

12

Three Cases for r = (a/b^d)

Geometric series: $\sum_{i=0}^{k} r^i = \frac{r^{k+1}-1}{r-1}$ Here r = (a/b^d)

1. $r < 1$ e.g. $r = \frac{1}{2}$ $1+1/2+1/4+\ldots < 2$ for any k
2. $r = 1$ $\sum_{i=0}^{k} 1^i = k+1 = O(k)$
3. $r > 1$ e.g. $r = 2$ $1+2+4+2^k = 2^{k+1}-1 = O(2^k)$

13

The three cases in practice

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

r = 1 a=2, b=2, d=1 $r = a/b^d = 1$ $n^1 \sum_{i=0}^{\log n} 1^i = n(\log n + 1)$

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

$T(n) = 2T(n/2) + 1$ // e.g. recursive max in array size n:

if n=1, then the element is the max.

r > 1 else divide array in 2 halves, find max of each and choose max of the two

a=2, b=2, d=0 $r = a/b^d = 2$ $n^0 \sum_{i=0}^{\log n} 2^i = (2^{\log n) + 1} - 1)/(2-1) = (2n-1)/1$

$T(n) = O(n)$

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

r < 1 a=2, b=2, d=2 $r = - a/b^d = 1/2$ $n^2 \sum_{i=0}^{\log n} (\frac{1}{2})^i = n^2 (1 + 1/2 + 1/4 + \ldots) < 2n^2$

$T(n) = O(n^2)$

Draw trees for these and do the analysis, as in slides 9, 10, 11

14

Towers of Hanoi

- Move all disks to third peg, without ever placing a larger disk on a smaller one.
- What's the recurrence relation? $a_n = 2a_{n-1} + 1$ with the base case that $a_1 = 1$
- Let's solve by repeated substitution
  - Plug in the definition
  - Do the algebra to collect all the non-recursive expressions together
  - Identify a pattern
  - Determine how many times the pattern occurs until we hit the base case

Hanoi by repeated substitution

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

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

$$= 4T(n-2) + 2 + 1$$

$$= 4(2T(n-3)+1) + 2 + 1$$

$$= 8T(n-3) + 4 + 2 + 1$$

- What is the label and how is it changing?
- What about the other terms?
- When do we hit the base case?

Hanoi by repeated substitution

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

$$\begin{aligned}
&= 2(2T(n-2)+1)+1 \\
&= 4T(n-2)+2+1 \\
&= 4(2T(n-3)+1)+2+1 \\
&= 8T(n-3)+4+2+1 \\
&\vdots \\
&= 2^i T(n-i) + \sum_{j=0}^{i-1} 2^j
\end{aligned}$$

- When does the label become 1?
- When $i = n - 1$ So our solution is

Colorado State University

Hanoi by repeated substitution

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

$$= \sum_{j=0}^{n-1} 2^j = 2^n - 1 = \Theta(2^n)$$

- This is a geometric series
- The Master Theorem does not apply