Lecture 10: Divide and Conquer via Sorting: Mergesort

CS 5002: Discrete Math

Adrienne Slaughter

Northeastern University

November 15, 2018

Agenda

- What is Sorting?
- Intro to Divide and Conquer
- A first Divide and Conquer problem: Mergesort: The Algorithm
- Analyzing runtime: Recursion Trees
- Proving correctness: Induction
- If time, more on Divide and Conquer

Section 1

A first problem: Sorting

Why sorting?

Learning sorts is like learning scales for musicians.

Applications of Sorting

It turns out that sorting makes a bunch of other problems really easy to solve:

- **Searching:** Binary search is great, but requires sorted data. Once data is sorted, it's easy to search.
- **Closest pair:** Given a set of m numbers, how do you find the pair of numbers that have the smallest difference between them?
- **Element uniqueness:** Are there any duplicates in a given set of n items? (A special case of the closest pair)
- **Frequency distribution:** Given a set of n items, which element occurs the largest number of times in the set? Note, this enables not just calculating frequencies, but can also support the question "How many times does item k occur?".
- **Selection:** What is the k^{th} largest number in an array? If the array is sorted, lookup is constant.

Sorting Orders

Collections can be sorted in different orders:

- An array $a[0 \ldots n]$ is in ***increasing order*** if $a[i] < a[j]$ for all $i < j$ where $0 \le i < j \le n$
  - $[1, 2, 3, 4, 5]$
- An array $a[0 \ldots n]$ is in ***decreasing order*** if $a[i] > a[j]$ for all $i < j$ where $0 \le i < j \le n$
  - $[5, 4, 3, 2, 1]$

  ←

Sorting Orders, cont.

In addition to sorting in either increasing or decreasing order:

- An array $a[0 \dots n]$ is in ***non-increasing*** where $a[i] \geq a[j]$ for all $i < j$ where $0 \leq i < j \leq n$
  - $[5, 4, 4, 3, 2, 2, 2]$
  - Elements can be repeated
- An array $a[0 \dots n]$ is in ***non-decreasing*** order if $a[i] \leq a[j]$ for all $i < j$ where $0 \leq i < j \leq n$
  - $[1, 2, 2, 2, 3, 4, 4, 5]$
  - Again, elements can be repeated

Divide and Conquer

Three Steps:

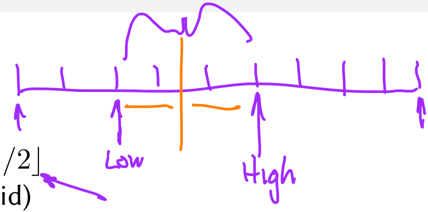
1. ***Divide*** the problem into a number of subproblems.
2. ***Conquer*** the subproblems by solving them recursively.
3. ***Combine*** the solutions to the subproblems into the solution for the original problem.

Divide and Conquer: Applied to Merge Sort

1. ***Divide*** the problem into a number of subproblems: Split the input into two sub-lists.
2. ***Conquer*** the subproblems by solving them recursively: Sort each list half.
3. ***Combine*** the solutions to the subproblems into the solution for the original problem: Merge the sorted halves together.

Merge Sort

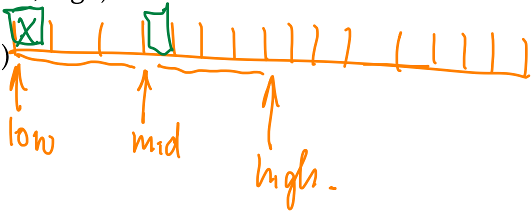
```
MERGE-SORT($\underline{A}, \underline{low}, \underline{high}$)
    if $(low < high)$
        $mid = \lfloor (low + high)/2 \rfloor$
        MERGE-SORT(A, low, mid)
        MERGE-SORT(A, mid+1, high)
        MERGE(A, low, mid, high)
```



Merge Sort

```
MERGE-SORT($A, low, high$)
    if ($low < high$)
        $mid = \lfloor (low + high)/2 \rfloor$
        MERGE-SORT(A, low, mid)
        MERGE-SORT(A, mid+1, high)
        MERGE(A, low, mid, high)
```

```
MERGE(A, low, mid, high)
    L = A[low:mid]
    R = A[mid+1, high]
    if L[0] < R[0]:
        A[low] = L[0]
    else
        A[low] = R[0]
```

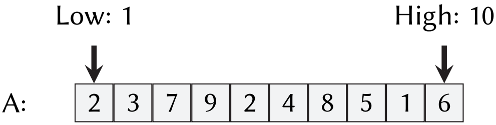


Merge Sort: The Algorithm, Graphically

A:	2	3	7	9	2	4	8	5	1	6

Merge Sort: The Algorithm, Graphically

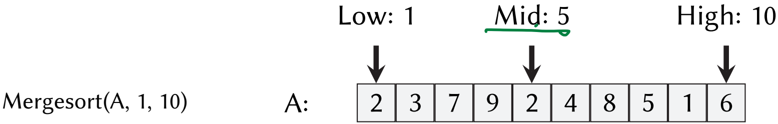
Mergesort(A, 1, 10)



2	3	7	9	2	4	8	5	1	6

2. Start by specifying you want to sort the entire array: Mergesort(A, 1, 10)

Merge Sort: The Algorithm, Graphically



Low: 1

Mid: 5

High: 10

Mergesort(A, 1, 10)

A:

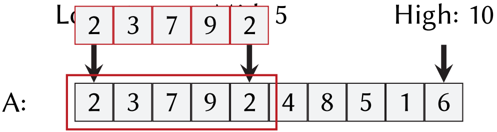
2	3	7	9	2	4	8	5	1	6

3. Find the mid point. mid = 1 + floor(10/2) - 1 = 5

Merge Sort: The Algorithm, Graphically

Mergesort(A, 1, 5)

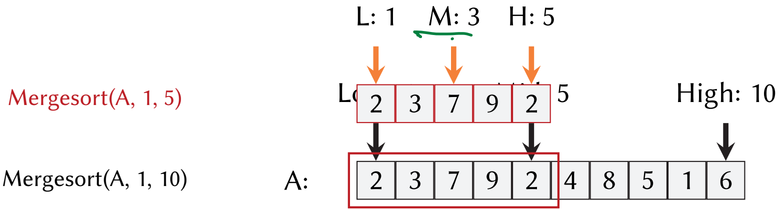
Mergesort(A, 1, 10)



4.Sort the left side (when we're done sorting the left, we'll sort the right):

Mergesort(A, 1, 5)

Merge Sort: The Algorithm, Graphically



L: 1 M: 3 H: 5

Mergesort(A, 1, 5)

Lo 5 High: 10

2	3	7	9	2

Mergesort(A, 1, 10)

A:

2	3	7	9	2	4	8	5	1	6

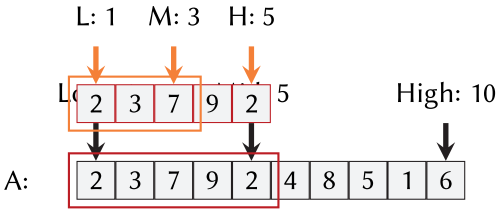
5. Find the midpoint of the left side mid = 3

Merge Sort: The Algorithm, Graphically

Mergesort(A, 1, 3)

Mergesort(A, 1, 5)

Mergesort(A, 1, 10)



L: 1 M: 3 H: 5

L 5 High: 10

2 3 7 9 2

A: 2 3 7 9 2 4 8 5 1 6

6. … and sort the left of that. Mergesort(A, 1, 3)

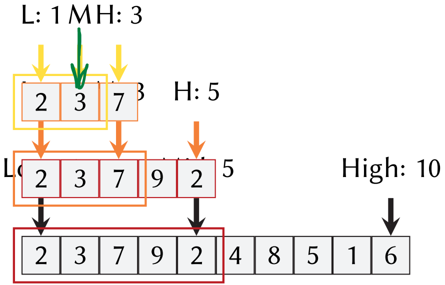
Merge Sort: The Algorithm, Graphically

Mergesort(A, 1, 2)

Mergesort(A, 1, 3)

Mergesort(A, 1, 5)

Mergesort(A, 1, 10)



L: 1 M H: 3

2	3	7

H: 5

2	3	7	9	2

5

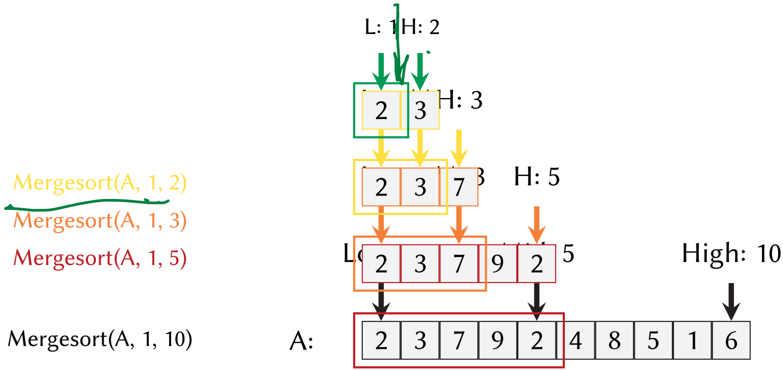
High: 10

A:

2	3	7	9	2	4	8	5	1	6

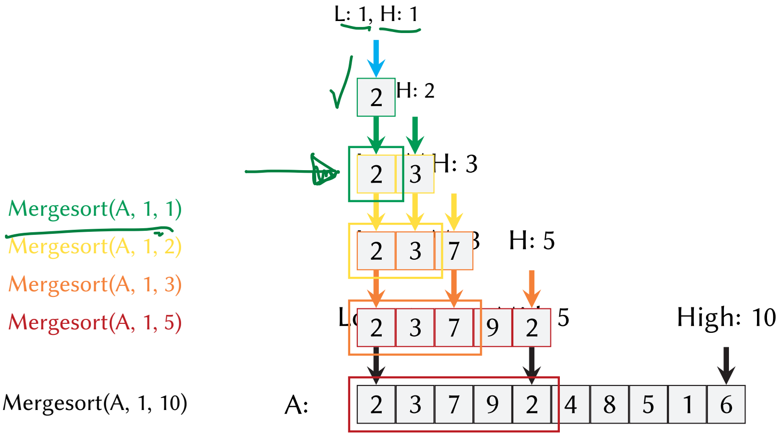
Find the midpoint, sort the left. Mid = 2; Mergesort(A, 1, 2)

Merge Sort: The Algorithm, Graphically



8. Find the midpoint, sort the left.

Merge Sort: The Algorithm, Graphically



9. One more time, sort the left. Since $low = high$, we're done with the left. We move to the next line, which is Mergesort(A, 2, 2)– that is, sorting the right.

Merge Sort: The Algorithm, Graphically

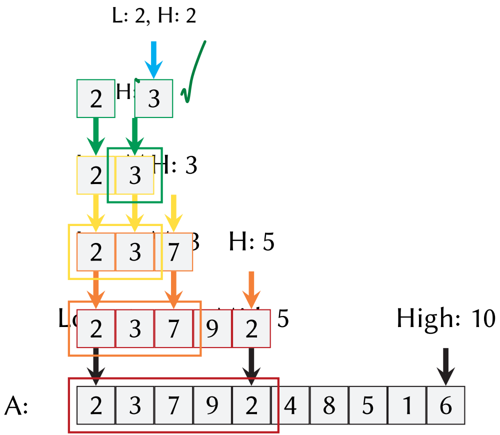
Mergesort(A, 2, 2)

Mergesort(A, 1, 2)

Mergesort(A, 1, 3)

Mergesort(A, 1, 5)

Mergesort(A, 1, 10)



10. Sort the right. Since $low = high$, we're done with the right.

Merge Sort: The Algorithm, Graphically

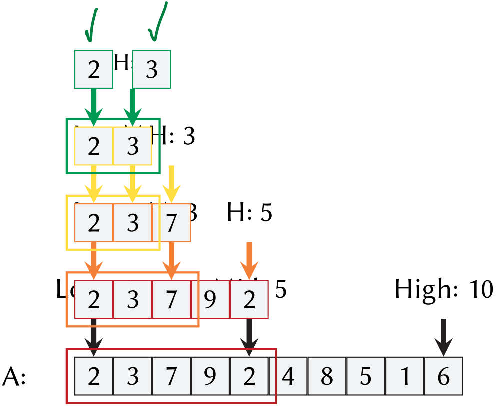
Merge(A, 1, 1, 2)

Mergesort(A, 1, 2)

Mergesort(A, 1, 3)

Mergesort(A, 1, 5)

Mergesort(A, 1, 10)



11. Merge the left and right. Merge(A, 1, 2, 2)

Merge Sort: The Algorithm, Graphically

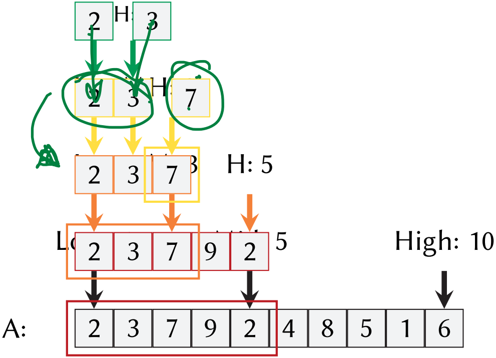
Mergesort(A, 3, 3)

Mergesort(A, 1, 2)

Mergesort(A, 1, 3)

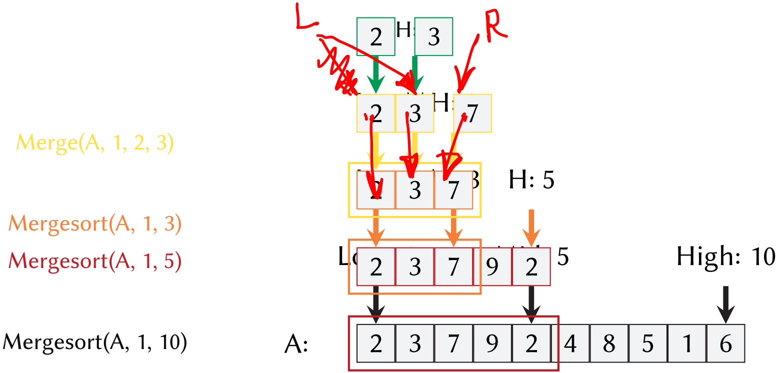
Mergesort(A, 1, 5)

Mergesort(A, 1, 10)



12. Sort the right Mergesort(A, 3, 3)

Merge Sort: The Algorithm, Graphically



L

R

2 H: 3

2 3 H: 7

Merge(A, 1, 2, 3)

2 3 7

H: 5

Mergesort(A, 1, 3)

Mergesort(A, 1, 5)

Lo 2 3 7 9 2 5

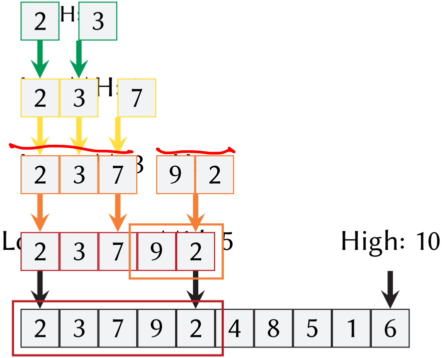
High: 10

Mergesort(A, 1, 10)

A: 2 3 7 9 2 4 8 5 1 6

13. Merge the left and right. Merge(A, 1, 2, 3), which means we're done with Mergesort(A, 1, 3)

Merge Sort: The Algorithm, Graphically



Mergesort(A, 4, 5)

Mergesort(A, 1, 5)

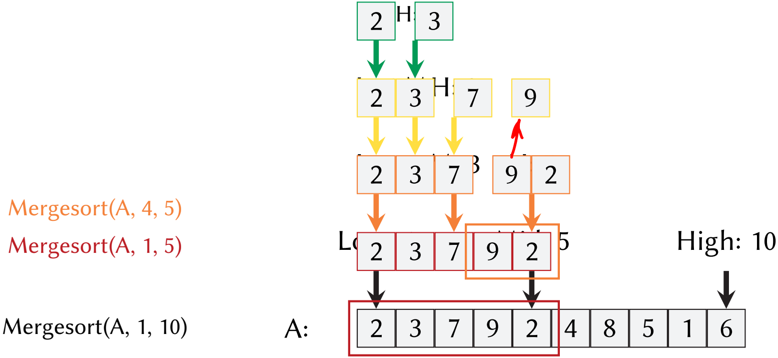
Mergesort(A, 1, 10)

A:

2	3	7	9	2	4	8	5	1	6

14. Sort the right Mergesort(A, 4, 5)

Merge Sort: The Algorithm, Graphically



Mergesort(A, 4, 5)

Mergesort(A, 1, 5)

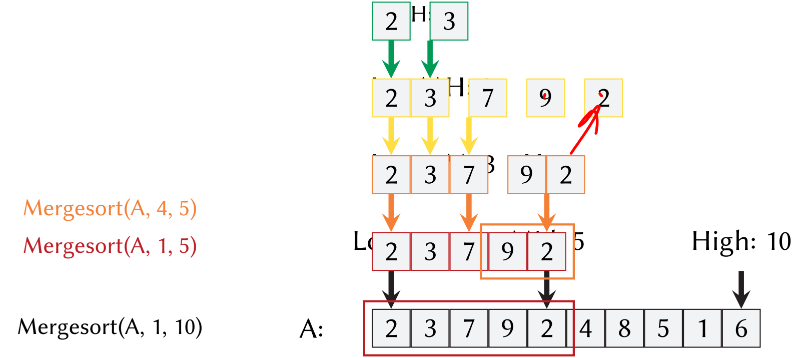
Mergesort(A, 1, 10)

A: 2 3 7 9 2 4 8 5 1 6

High: 10

15. Sort the left. Mergesort(A, 4, 4)

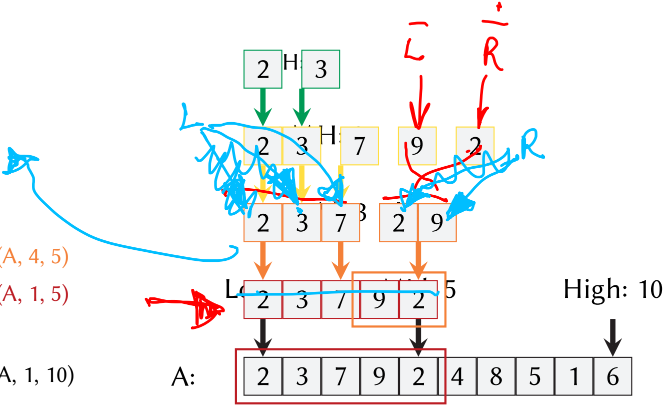
Merge Sort: The Algorithm, Graphically



16. Sort the right. Mergesort(A, 5, 5)

Merge Sort: The Algorithm, Graphically

2 2 3



Mergesort(A, 4, 5)

Mergesort(A, 1, 5)

Mergesort(A, 1, 10)

17. Merge the left and right. Merge(A, 4, 4, 5)

Merge Sort: The Algorithm, Graphically

L: |2|3|7| R: |9|2|

Merge(A, 1, 3, 5)

Merge(A, 2, 3, 5)

Merge(A, 3, 3, 5)

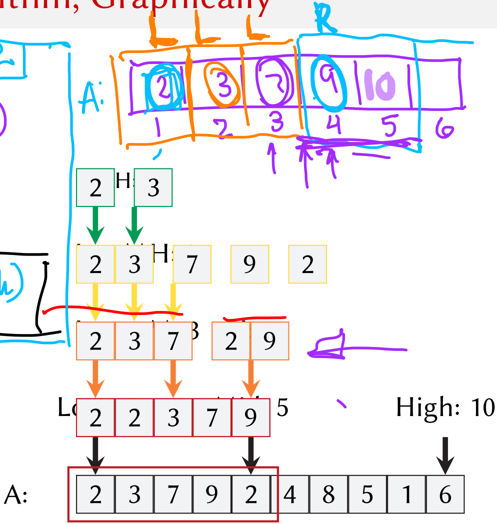
Merge(A, 4, 3, 5)

Merge(A, ~~low+1~~, m, high)

l h m=3

Mergesort(A, 1, 5)

Mergesort(A, 1, 10)



18. Merge the left and right. Merge(A, 1, 3, 5)

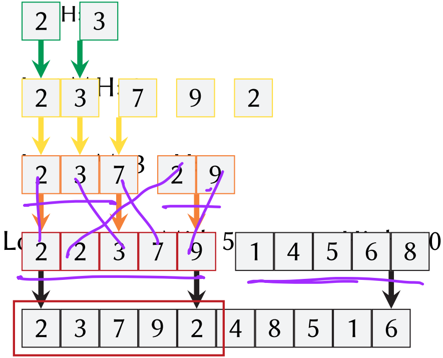
Merge Sort: The Algorithm, Graphically

M-Sort(A, 6, 10)

Mergesort(A, 1, 5)

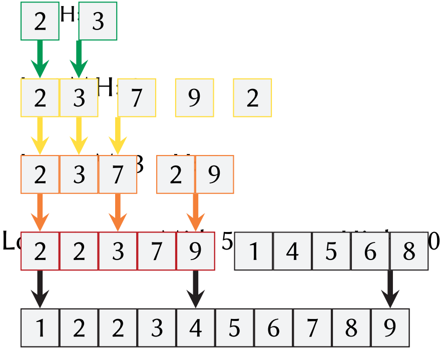
Mergesort(A, 1, 10)

A:



19. The left half is sorted. Do the same for the right.

Merge Sort: The Algorithm, Graphically

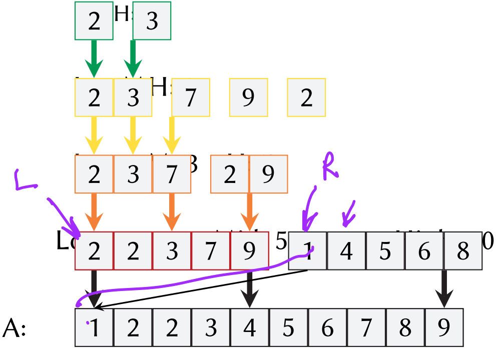


Mergesort(A, 1, 10)

A:

20. Merge the two halves together into a sorted output.

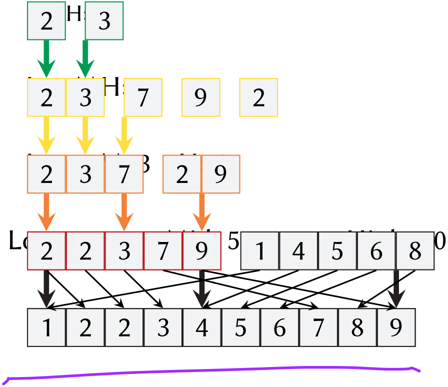
Merge Sort: The Algorithm, Graphically



Mergesort(A, 1, 10)

A:

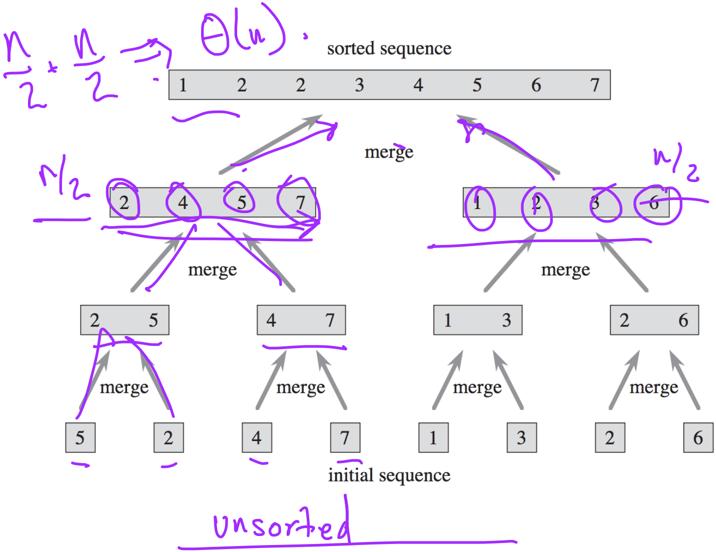
Merge Sort: The Algorithm, Graphically



Mergesort(A, 1, 10)

A:

Merge Sort: The Algorithm, Graphically



Mergesort: Analysis

We do 2 things in our analysis:

- What’s the runtime?
- Is it correct?

Mergesort: What's the runtime?

What's the runtime of Mergesort?

- We'll start by saying that $\underline{T(n)}$ is the runtime of Mergesort on an input of size $\underline{n}$.

$\underline{n = 1} \Longrightarrow T(1) = 1 \ : \ \Theta(1)$

M-S(n): ← $T(n) = T(n/2) + T(n/2) + \boxed{T(\text{Merge})}$

M-S($n/2$)

M-S($n/2$)

Merge.

$T(\text{Merge}) = \Theta(n)$

$$T(n) = T(n/2) + T(n/2) + \Theta(n).$$

Mergesort: What's the runtime?

What's the runtime of Mergesort?

- We'll start by saying that $T(n)$ is the runtime of Mergesort on an input of size n.
- If the size of the input to Mergesort is small (that is, 1), the runtime is easy: $\boxed{\Theta(1)}$

Line 3 + [illegible] $T(n/2)$

4 $T(n/2)$

5 $\Theta(n)$

$$2T(n/2) + \Theta(n)$$

Evening

Mergesort: What's the runtime?

What's the runtime of Mergesort?

- We'll start by saying that $T(n)$ is the runtime of Mergesort on an input of size n.
- If the size of the input to Mergesort is small (that is, 1), the runtime is easy: $\Theta(1)$.
- If the size of the input is bigger, say, n, the runtime is the time it takes to run Mergesort on each half, plus the runtime of Merge:
  - $T(\frac{n}{2}) + T(\frac{n}{2}) + T(Merge)$
- $T(Merge) = \Theta(n)$
- $\Rightarrow T(\frac{n}{2}) + T(\frac{n}{2}) + \Theta(n)$

$$T(n) = \begin{cases} \Theta(1) & \text{if } n \leq 1 \\ 2T(\frac{n}{2}) + \Theta(n) & \text{otherwise} \end{cases}$$

Mergesort: What's the runtime?

What's the runtime of Mergesort?

- We'll start by saying that $T(n)$ is the runtime of Mergesort on an input of size n.
- If the size of the input to Mergesort is small (that is, 1), the runtime is easy: $\Theta(1)$.
- If the size of the input is bigger, say, n, the runtime is the time it takes to run Mergesort on each half, plus the runtime of Merge:
  - $T(\frac{n}{2}) + T(\frac{n}{2}) + T(Merge)$
- $T(Merge) = \Theta(n)$
- $\Rightarrow T(\frac{n}{2}) + T(\frac{n}{2}) + \Theta(n)$

$$T(n) = \begin{cases} \Theta(1) & n = 1 \\ 2T(n/2) + \Theta(n) \end{cases}$$

$$\underline{a_0 = 5.}$$

$$a_n = \underline{a_{n-1}} \underline{+ 3}$$

Evening

Mergesort: What's the runtime?

What's the runtime of Mergesort?

- We'll start by saying that $T(n)$ is the runtime of Mergesort on an input of size n.
- If the size of the input to Mergesort is small (that is, 1), the runtime is easy: $\boxed{\Theta(1)}$

Line 3 $T(n/2)$

〃 4 $T(n/2)$

〃 5 $\Theta(n)$

$$2T(n/2) + \Theta(n)$$

Runtime of a Divide and Conquer Algorithm

A divide and conquer algorithm has 3 steps: divide into subproblems, solve the subproblems, combine to solve the original problem.

$$T(n) = \begin{cases} n \leq 1: & \Theta(1) \leftarrow \\ \text{otherwise:} & 2T(n/2) + \Theta(n) \end{cases}$$

log n

Recurrence Relation.

Runtime of a Divide and Conquer Algorithm

A divide and conquer algorithm has 3 steps: divide into subproblems, solve the subproblems, combine to solve the original problem.

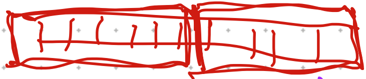
Let's say that $\boxed{D(n)}$ is the time it takes to divide into subproblems.

We break the problem into a problems, by dividing the input into b chunks.

$\boxed{C(n)}$ is the time it takes to combine the subproblems together.

$$T(n) = \begin{cases} \Theta(1) & n = 1 \ (\le c.) \\ aT(n/b) + \boxed{C(n) + D(n)} & \end{cases}$$

Evening



- Divide : $\underline{D(n)}$
- Solve subproblem
- Combine: $\underline{C(n)}$

• $\underline{a}$ problems
$\underline{b}$ chunks.

$$\boxed{T(n) = \begin{cases} \Theta(1) & \text{if } n \leq c \\ \underline{a}D\left(\frac{n}{b}\right) + \underline{C(n)} & \end{cases}}$$

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

Runtime of a Divide and Conquer Algorithm

A divide and conquer algorithm has 3 steps: divide into subproblems, solve the subproblems, combine to solve the original problem.

Let's say that $D(n)$ is the time it takes to divide into subproblems.

We break the problem into a problems, by dividing the input into b chunks.

$C(n)$ is the time it takes to combine the subproblems together.

This means that we can specify that the runtime of a Divide and Conquer algorithm will fit the structure:

$$T(n) = \begin{cases} \Theta(1) & \text{if } n \leq c \\ aT(n/b) + D(n) + C(n) & \text{otherwise} \end{cases}$$

Runtime of a Divide and Conquer Algorithm

A divide and conquer algorithm has 3 steps: divide into subproblems, solve the subproblems, combine to solve the original problem.

Let's say that $D(n)$ is the time it takes to divide into subproblems.

We break the problem into a problems, by dividing the input into b chunks.

$C(n)$ is the time it takes to combine the subproblems together.

This means that we can specify that the runtime of a Divide and Conquer algorithm will fit the structure:

$$T(n) = \begin{cases} \Theta(1) & \text{if } n \leq c \\ aT(n/b) + D(n) + C(n) & \text{otherwise} \end{cases}$$

Mergesort

Our Mergesort analysis fits this pattern!

$$T(n) = \begin{cases} \Theta(1) & \text{if } n \leq c \\ \underline{a}T(n/\underline{b}) + \underline{D(n)} + \underline{C(n)} & \text{otherwise} \end{cases}$$

$$T(n) = \begin{cases} \Theta(1) & \text{if } n \leq 1 \\ \underline{2}T(\underline{\tfrac{n}{2}}) + \underline{\Theta(n)} & \text{otherwise} \end{cases}$$

$D(n) = 0$

Evening

Mergesort

Our Mergesort analysis fits this pattern!

$$T(n) = \begin{cases} \Theta(1) & \text{if } n \leq c \\ aT(n/b) + D(n) + C(n) & \text{otherwise} \end{cases}$$

$$T(n) = \begin{cases} \Theta(1) & \text{if } n \leq 1 \\ 2T(\frac{n}{2}) + \Theta(n) & \text{otherwise} \end{cases}$$

$$\begin{aligned} a &=? \\ b &=? \\ C(n) &=? \\ D(n) &=? \end{aligned}$$

Recurrences

These runtimes are examples of a ***recurrence relation***:
a function defined in terms of itself.

We need to figure out how to give us bounds on the recurrences for the runtime.

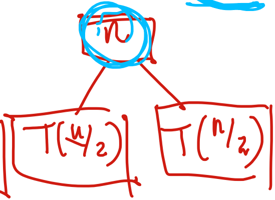
There are 3 approaches:

1. Substitution
2. Master method
3. Recursion trees (sometimes called ***iteration***; referred to as *"unrolling"* the recurrence).

Example: Recursion Tree

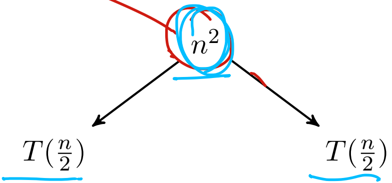
Merge Sort

$T(n) = 2T(\frac{n}{2}) + \underline{\Theta(n)}$



How much time to do work.

$$T(n) = 2T(n/2) + \underline{n^2}$$

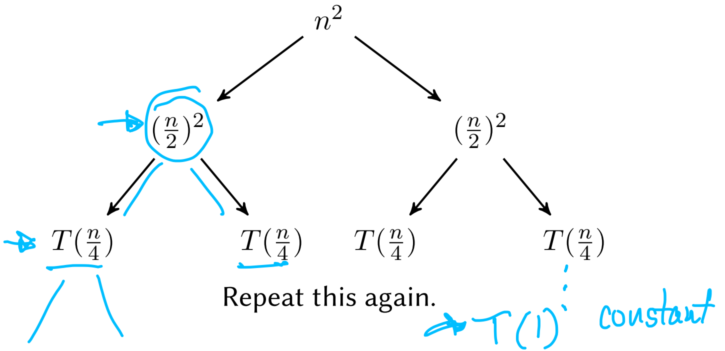


$T(\frac{n}{2}) + T(\frac{n}{2}) + n^2 = 2T(\frac{n}{2}) + n^2$

Example: Recursion Tree

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

$4T(n/4) + \left(\frac{n}{2}\right)^2 + \left(\frac{n}{2}\right)^2 + n^2 \ldots$



n^2

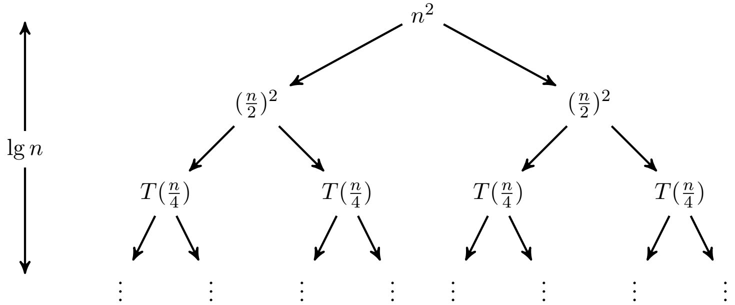
$\left(\frac{n}{2}\right)^2$ $\left(\frac{n}{2}\right)^2$

$T\left(\frac{n}{4}\right)$ $T\left(\frac{n}{4}\right)$ $T\left(\frac{n}{4}\right)$ $T\left(\frac{n}{4}\right)$

Repeat this again.

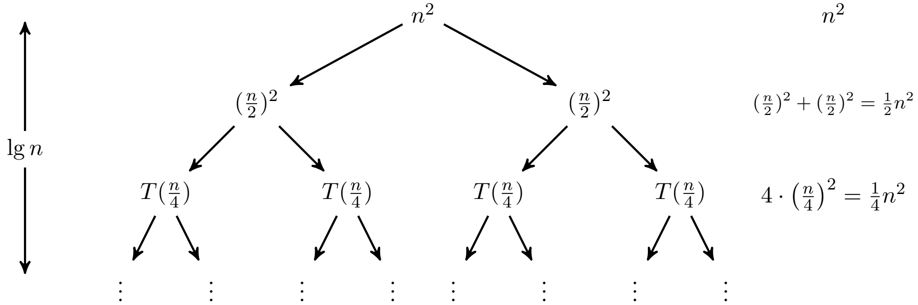
$T(1)$ constant

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



If we keep doing this, we'll have a tree of $\lg n$ levels.

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

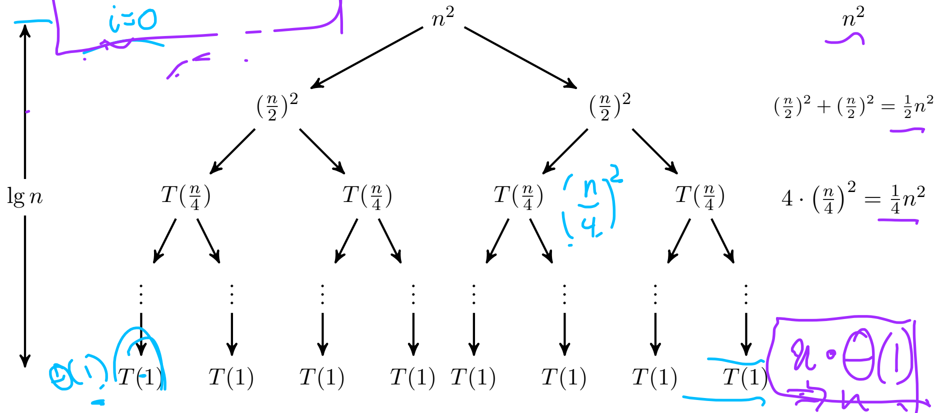


If we keep doing this, we'll have a tree of $\lg n$ levels.

$$\sum_{i=0}^{\lg n} n + \Theta(n^2)$$

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

$$T(n) = \begin{cases} n=1: & \Theta(1) \\ n>1: & \ldots \end{cases}$$



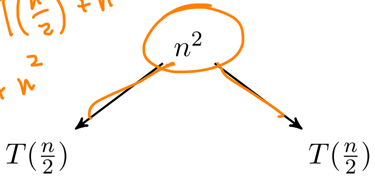
Keep doing this until we get to the base case, which is an input of size 1. Now we have the tree. The next few slides will show how we calculate the bound overall.

Example: Recursion Tree

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

$T(n) = T\left(\frac{n}{2}\right) + T\left(\frac{n}{2}\right) + n^2$

$= 2T\left(\frac{n}{2}\right) + n^2$



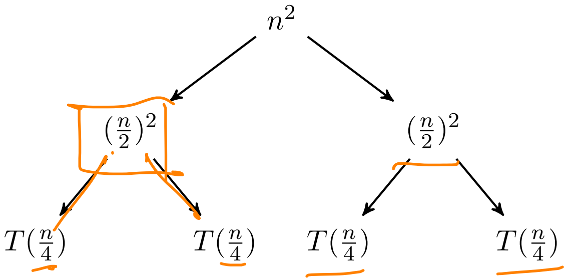
n^2

$T(\frac{n}{2})$ $T(\frac{n}{2})$

Evening

Example: Recursion Tree

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



n^2
$\left(\frac{n}{2}\right)^2$
$\left(\frac{n}{2}\right)^2$
$T\left(\frac{n}{4}\right)$
$T\left(\frac{n}{4}\right)$
$T\left(\frac{n}{4}\right)$
$T\left(\frac{n}{4}\right)$

n^2

$2 \cdot \frac{n^3}{4}$

$4T\left(\frac{n}{4}\right)$

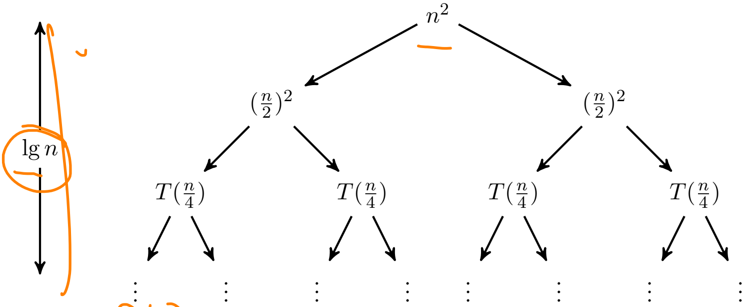
Repeat this again.

Evening

$\log_b a = y$

$b^y = a$

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



$\Theta(1)$

If we keep doing this, we'll have a tree of $\lg n$ levels.

Evening

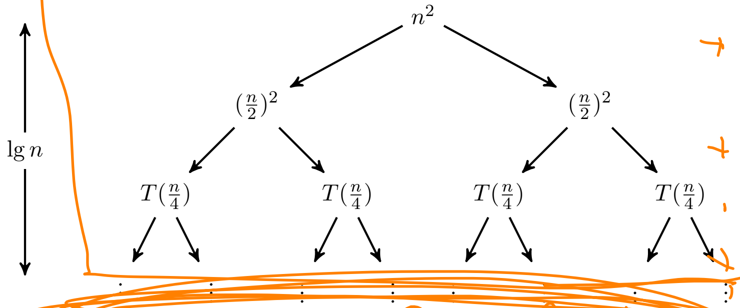
$$\sum_{i=0}^{\lg n-1} \frac{1}{2^i} n^2$$

$i=0$: n^2 ✓

$i=1$: $\frac{1}{2}n^2$ ✓

$i=2$: $\frac{1}{4}n^2$ ✓

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



n^2

+

$(\frac{n}{2})^2 + (\frac{n}{2})^2 = \frac{1}{2}n^2$

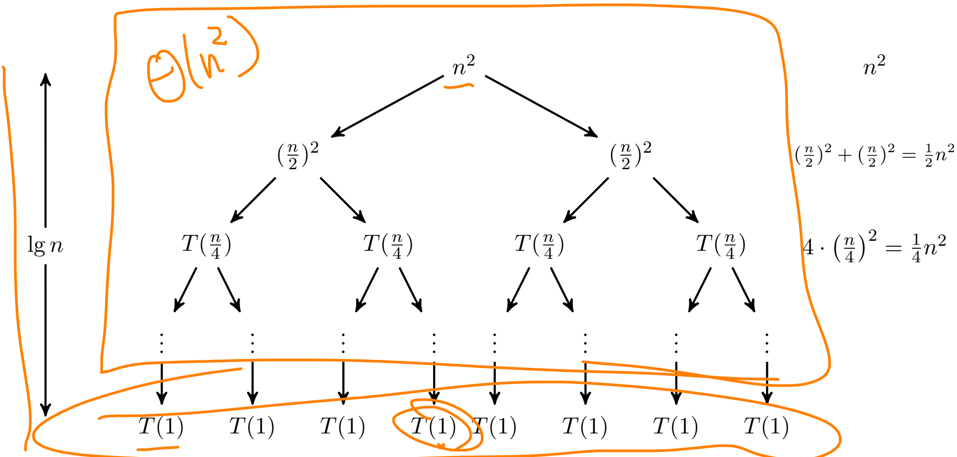
+

$4 \cdot (\frac{n}{4})^2 = \frac{1}{4}n^2$

If we keep doing this, we'll have a tree of $\lg n$ levels.

Evening

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



Keep doing this until we get to the base case, which is an input of size 1. Now we have the tree. The next few slides will show how we calculate the bound overall.

- Add up amount of work done at each node
    - Add up work done at the bottom level
    - Add up work done at all the levels above the bottom
        - Sum over all levels: the amount of work done on each level times the number of nodes on that level.

Computing the work done at the base case (bottom) layer

n: $\Theta(1)$

Bottom layer: $\Theta(n)$

Evening

Computing the work done at the base case (bottom) layer

Computing the work done on all the other layers

$i=0: \ n^2$

$i=1: \ \frac{1}{2}n^2$

$i=2 = \frac{1}{4}n^2$

The height of the tree is:

Amount of work done on each level i:

Amount of work done on all levels other than the bottom [1]:

$$\rightarrow \underline{\sum_{i=0}^{\lg n-1} \frac{1}{2^i} n^2 =}$$

[1] https://en.wikipedia.org/wiki/Summation#Known_summation_expressions

Computing the work done on all the other layers

The height of the tree is:

Amount of work done on each level i:

Amount of work done on all levels other than the bottom [1]:

$$\sum_{i=0}^{\lg n - 1} \frac{1}{2^i} n^2 =$$

[1] https://en.wikipedia.org/wiki/Summation#Known_summation_expressions

Computing the work done on all the other layers

The height of the tree is:

Amount of work done on each level i:

Amount of work done on all levels other than the bottom [1]:

$$\boxed{\sum_{i=0}^{\lg n-1} \frac{1}{2^i} n^2} = n^2 \sum_{i=0}^{\lg n-1} \frac{1}{2^i}$$

$$= n^2 \cdot \left(2 - \frac{1}{2^{\lg n - 1}}\right)$$

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

$$= n^2 \cdot \left(2 - \frac{1}{n}\right) = \boxed{2n^2 - n}$$

$2^{\lg n}$

$b = 2$

$\lg n = y$

$\log_b x = y$

$b^{\log_b x} = x$

$b = 2$

$\Theta(n^2)$

Evening

[1] https://en.wikipedia.org/wiki/Summation#Known_summation_expressions

Computing the work done on all the other layers

The height of the tree is:

Amount of work done on each level i:

Amount of work done on all levels other than the bottom [1]:

$$\sum_{i=0}^{\lg n-1} \frac{1}{2^i} n^2 = \underline{n^2} \sum_{i=0}^{\lg n-1} \frac{1}{2^i}$$

$$\boxed{2^{\lg n} = n}$$

$$= n^2 \cdot \left(2 - \frac{1}{\underline{2^{\lg n-1}}}\right)$$

$$= n^2 \cdot \left(2 - \frac{1}{\underline{2^{\lg n/1}}}\right)$$ Using the log quotient rule

$$= n^2 \cdot \left(2 - \frac{1}{n}\right)$$ Using def of log: $b^{\log_b x} = x$

$$= \underline{2n^2} - \underline{n} \longrightarrow \boxed{\Theta(n^2)}$$

$$\longrightarrow \sum_{i=0}^{\lg n-1} \frac{1}{2^i} n^2 = \underline{\Theta(n^2)}$$

[1] https://en.wikipedia.org/wiki/Summation#Known_summation_expressions

Finishing up

Putting it all together:

We said the amount of work done in the bottom layer is $\Theta(n)$, and the amount of work done in all the other layers is $\Theta(n^2)$.

Therefore:

$$T(n) = 2T(n/2) + n^2 = \Theta(n^2) + \Theta(n) \Rightarrow \Theta(n^2).$$

Helpful notes on tree math

When it comes to recurrence trees, we know the number of leaves, because we (usually) keep dividing the input until the size of the input is 1; that means each of the input is represented as a leaf at some point. That gives us n leaves.

You can assume this for recurrences, unless you have reason to believe otherwise. For example, sometimes we have 4 branches with size $n/2$ at each node. In this case, we'll have more than n nodes at the bottom.

If k is the size of the split of the input:

Number of leaves of the tree (full): k^h, where h is the height of the tree.

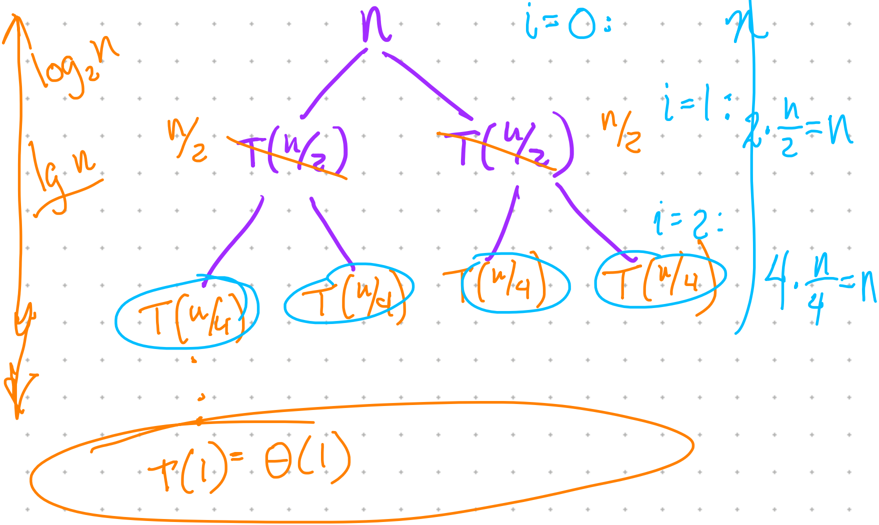
If we know the number of leaves of the tree, we can calculate the height of the tree.

Assume n is the number of leaves:

$$k^h = n \Leftrightarrow \log_k n = h$$

When it comes to recurrences of the form $T(n) = aT(n/b) + c$, the height of the tree is $\log_b n$.

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



Evening

Bottom Row/Leaves: $\Theta(1) \cdot n$.

$$\boxed{= \Theta(n)}$$

Rest: $\displaystyle\sum_{i=0}^{\lg n - 1} \underline{n} = n \sum_{i=0}^{\lg n - 1} 1$

$$\underbrace{1 + 1 + 1 + \dots}_{\lg n}$$

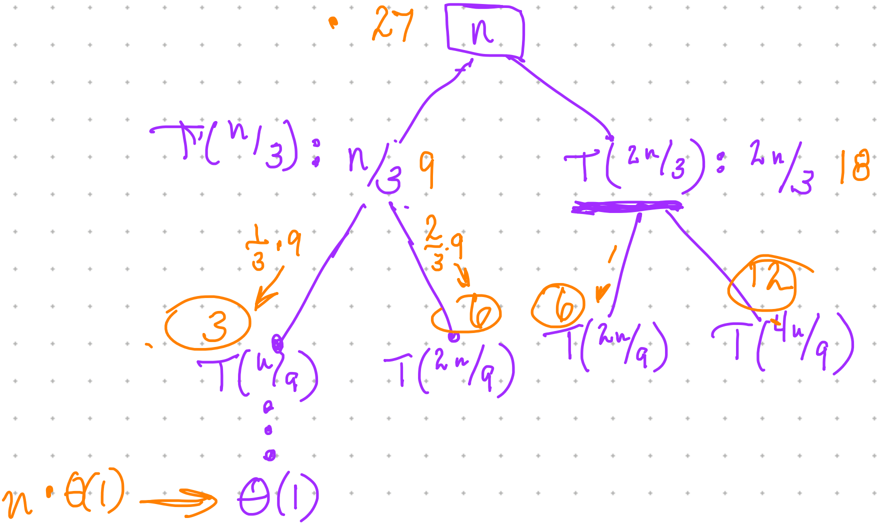
$$\boxed{= n \cdot \lg n}$$

$$T(n) = n \lg n + \Theta(n)$$

$$= \underline{\Theta(n \lg n)}$$

Evening

$$T(n) = \underline{T(n/3)} + \underline{T(2n/3)} + \boxed{n}$$

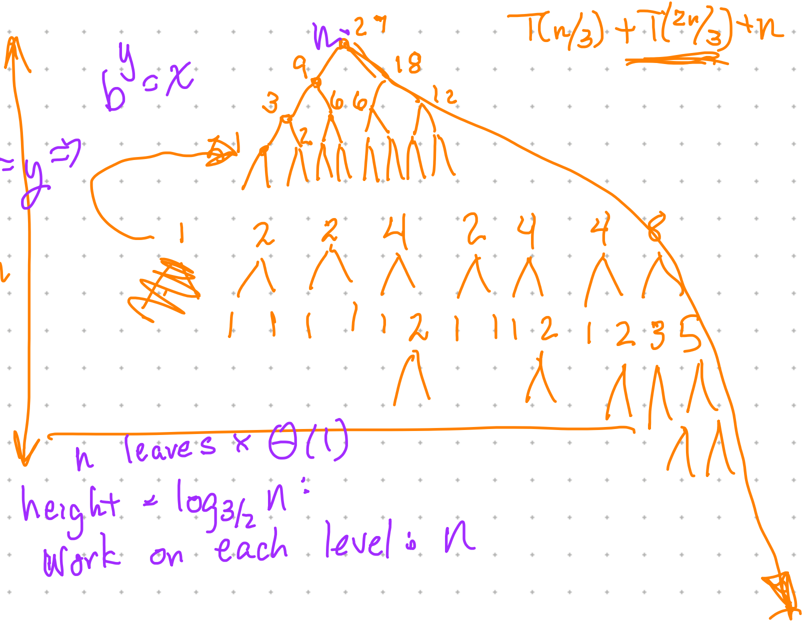


$T(n/3) + T(2n/3) + n$

$b^y = x$

$\log_b x = y \Rightarrow$

$\log_{3/2} n$



n leaves × $\Theta(1)$

height = $\log_{3/2} n$:

Work on each level : n

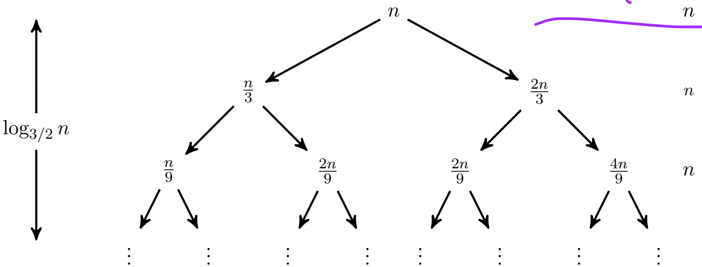
A more complex example

$$\sum_{i=0}^{\log_{3/2} n - 1} n + \Theta(n) =$$

n

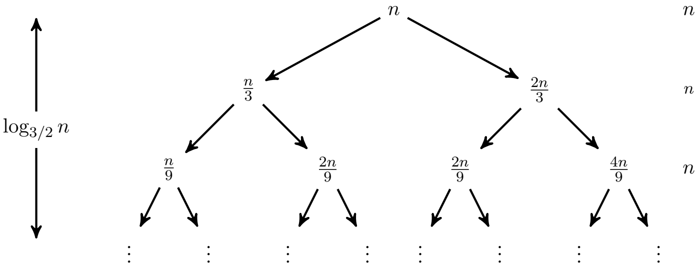
$$T(n) = T(n/3) + T(2n/3) + n$$

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



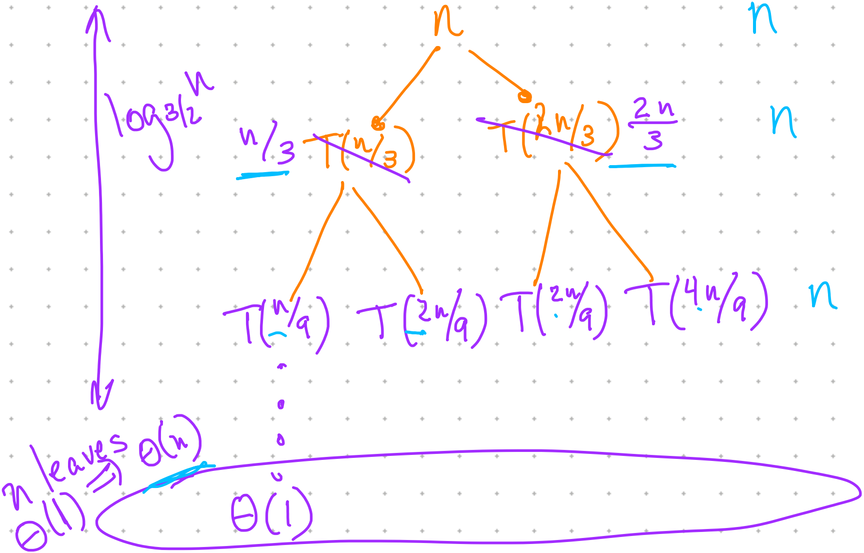
A more complex example

$$T(n) = T(n/3) + T(2n/3) + n$$

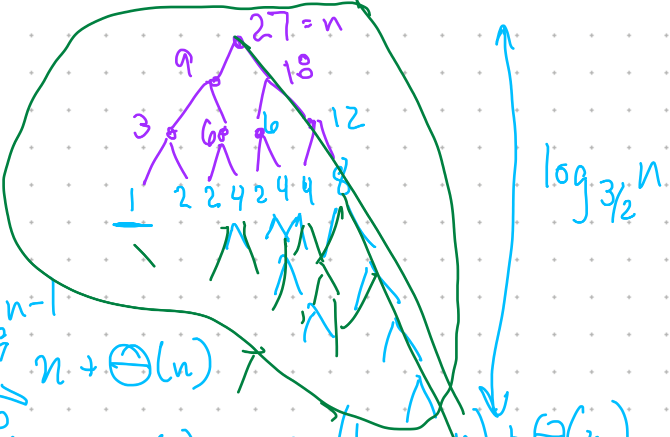


$\Rightarrow$ A bound for the recurrence is $O(n \log n)$

$$T(n) = T(n/3) + T\left(\underline{2n/3}\right) + n$$



Evening



$$\sum_{i=0}^{\log_{3/2} n - 1} n + \Theta(n)$$

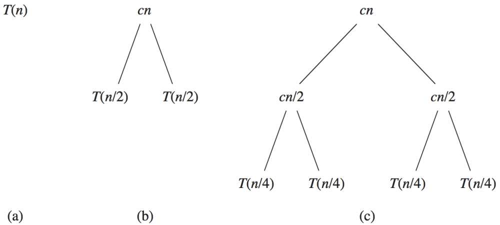
$$n \sum_{i=0} 1 + \Theta(n) = n \cdot (\log_{3/2} n) + \Theta(n)$$

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

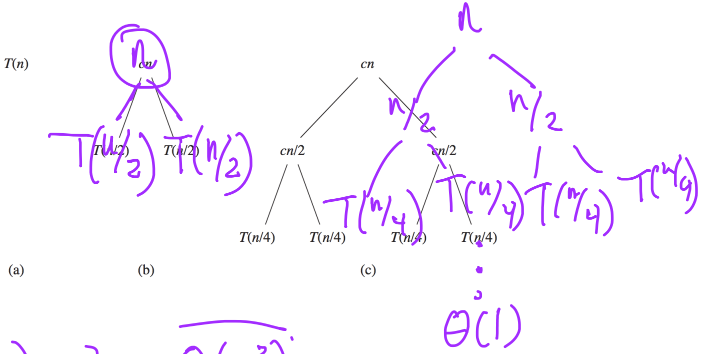
$$T(n) = O(n \log_2 n) = \underline{O(n \lg n)}$$

Evening

Merge Sort :



Evening

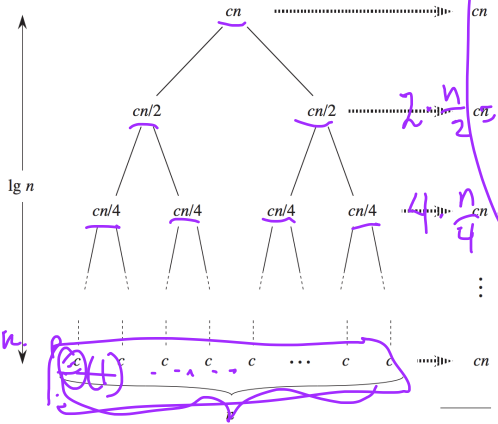


→ $2T(n/2)+n^2 = \overline{O(n^2)}$

→ MS: $\underline{2T(n/2)} + n \Rightarrow \underline{O(n \log n)}$

$\log_2 n$. $\frac{n}{2}$ ...

$\lg n$.



n

$2 \cdot \frac{n}{2} = n$

$4 \cdot \frac{n}{4} = n$

$\Theta(1)$

n.

(d)

$$\sum_{i=0}^{\lg n - 1} n = n \log n$$

$$\lg n \cdot (n) = n \log n.$$

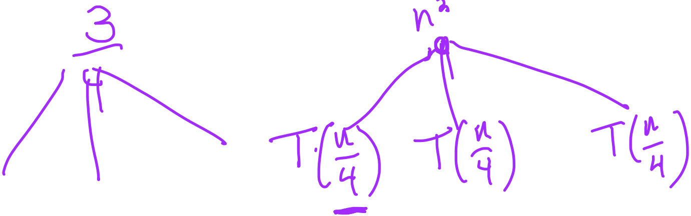
n

Total: $cn \lg n + cn$

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

FIX THIS!!!

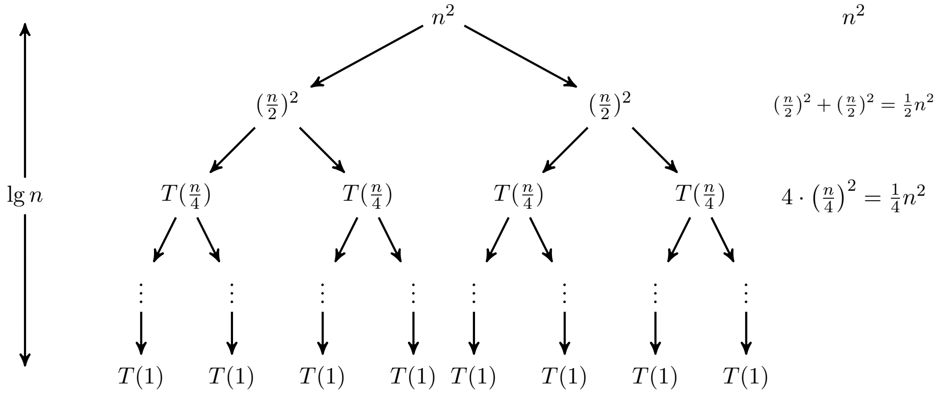
$$T(n) = 3T(\lfloor n/4 \rfloor) + \Theta\left(n^2\right)$$



$\frac{3}{4}$
n^2
$T\left(\frac{n}{4}\right)$
$T\left(\frac{n}{4}\right)$
$T\left(\frac{n}{4}\right)$

$$T(n) = 3T(\lfloor n/4 \rfloor) + \Theta\left(n^2\right)$$

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



Keep doing this until we get to the base case, which is an input of size 1. Now we have the tree. The next few slides will show how we calculate the bound overall.

Section 4

Master Theorem

Plan:

- Introduce the whole Master Theorem
- Break down the Master Theorem and specify definitions
- Restate the Master Theorem to develop intuition
- Use the Master Theorem to solve some recurrences
- Use the Master Theorem to analyze an algorithm

Subsection 1

Defining the Master Theorem

Master Theorem: The Whole Thing.

If a recurrence satisfies the form:

$$T(n) = aT(n/b) + f(n)$$

a is a constant such that $a \geq 1$

b is a constant such that $b > 1$

$f(n)$ is a function

1. If $f(n) = O(n^{\log_b a - \epsilon})$ for some constant $\epsilon > 0$, then $T(n) = \Theta(n^{log_b a})$
2. If $f(n) = \Theta(n^{\log_b a})$, then $T(n) = \Theta(n^{\log_b a} \lg n)$
3. If $f(n) = \Omega(n^{\log_b a + \epsilon})$ for some constant $\epsilon > 0$, and if $af(n/b) \leq cf(n)$ for some $c < 1$, then $T(n) = \Theta(f(n))$.

$$T(n) = \sqrt{2}\, T(n/2) + \log n.$$

$a = \sqrt{2}$

$b = 2$

$f(n) = \underline{\underline{\log n}}$

$$n^{\log_b a} = n^{\log_2 \sqrt{2}} = \underline{\underline{n^{1/2} = \sqrt{n}}}$$

?

$$f(n) = O(\sqrt{n})$$

$f(n) = \log n$

$g(n) = \sqrt{n}$

$f(n) = O(g(n))$

$\Rightarrow$

Case = 1:

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

$$= \underline{\Theta(\sqrt{n}).}$$

$\log_2 \sqrt{2} = y$

$2^y = \sqrt{2}$

$y = 1/2$

$2^{1/2} = \sqrt{2}$

Master Theorem: The recurrence structure

$$T(n) = aT(n/b) + f(n)$$

a is a constant such that $a \geq 1$

b is a constant such that $b > 1$

$f(n)$ is a function

Master Theorem: The recurrence structure

$$T(n) = \overset{\downarrow}{a}T(n/b) + f(n)$$

a is a constant such that $a \geq 1$

b is a constant such that $b > 1$

$f(n)$ is a function

1. a is the number of subproblems you solve
    - You have to have at least one subproblem to solve.

Master Theorem: The recurrence structure

$$T(n) = aT(n/b) + f(n)$$

a is a constant such that $a \geq 1$

b is a constant such that $b > 1$

$f(n)$ is a function

1. a is the number of subproblems you solve
   - You have to have at least one subproblem to solve.
2. b is the number of groups you split the input into
   - You have to have to divide the input into at least 2 groups

Master Theorem: The recurrence structure

$$T(n) = \boxed{aT(n/b)} + \boxed{f(n)}$$

a is a constant such that $a \geq 1$

b is a constant such that $b > 1$

$f(n)$ is a function

1. a is the number of subproblems you solve
   - You have to have at least one subproblem to solve.
2. b is the number of groups you split the input into
   - You have to have to divide the input into at least 2 groups
3. $f(n)$ is the function that describes the breaking/combination of the input/results
   - $f(n)$ must be a polynomial (can't $\boxed{\text{be } 2^n)}$

Reminder: Asymptotic Notation

Big-O: asymptotic upper bound

$f(x) = \underline{O(g(x))}$ means that $f(x)$ is lower than/less than $g(x)$

Big-Θ: asymptotically tight bound

$f(x) = \underline{\Theta(g(x))}$ means that $f(x)$ is similar to $g(x)$

Big-Ω: asymptotic lower bound

$f(x) = \underline{\Omega(g(x))}$ means that $f(x)$ is bigger than $g(x)$

Master Theorem: Focusing on $f(n)$

$$T(n) = aT(n/b) + f(n)$$

Master Theorem: Focusing on $f(n)$

$$T(n) = aT(n/b) + f(n)$$

1. **If $f(n)$ is smaller than $(n^{\log_b a})$, then $T(n) = \Theta(n^{log_b a})$**
   - If the time spent to split or combine the input is less than the time to compute the function on smaller input, then the running time is bounded by the actual computing on the smaller part.
2. **If $f(n)$ is about $(n^{\log_b a})$, then $T(n) = \Theta(n^{\log_b a} \lg n)$**
   - If the time spent to split or combine the input is about the same time to compute the function on smaller input, then the running time is a combination.
3. **If $f(n)$ is bigger than $(n^{\log_b a})$ then $T(n) = \Theta(f(n))$.**
   - If the time spent to split or combine the input is more than the time to compute the function on smaller input, then the running time is approximated by that function.

Extra constraint for Case 3

$af(n/b) \leq cf(n)$for some c < 1

The “regularity” condition; usually not relevant.

However, an example is shown at the end.

Subsection 3

Examples

Example1: Merge Sort

$a = 2$

$b = 2$

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

Example1: Merge Sort

$f(n) = \Theta(n) \,/\, n.$

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

1. $a = 2; b = 2$

Example1: Merge Sort

$T(n) = T(\frac{n}{2}) + T(\frac{n}{3}) + n$

$\log_b x = y$

$b^y = x$

$2^{\boxed{1}} = 2$

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

Case 2:

1. $a = 2; b = 2$
2. $f(n) = \Theta(n)$

$n^{\log_b a} = n^{\log_2 2} = n^1$

$T(n) = \Theta(n^{\log_b a} \lg n) =$

$= \Theta(n^{\underline{\log_2 2}} \lg n) = \Theta(n \log n)$

Example1: Merge Sort

$a = 2$ $b = 2$ $f(n) = \Theta(n)$

$\log_2 2 = \boxed{1}$

$n^{\log_b a} = n^{\log_2 2} = n$

$\log_b x = y$

$b^y = x$

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

$f(n) = \Theta(n)$ $\boxed{\Theta}$? $n^{\log_b a} = n$

Case 2:

→ Θ

→ O

→ Ω

$T(n) = \Theta(n^{\log_b a} \lg n)$

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

Evening

Example1: Merge Sort

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

1. $a = 2; b = 2$
2. $f(n) = \Theta(n)$
3. How does $f(n)$ compare to $n^{\log_b a} \to n^{\log_2 2} \to n^1 \to n$?
4. $\Theta(n) = \Theta(n)$, so Case 2 applies.
5. $T(n) = \Theta(n^{\log_b a} \lg n) \to \Theta(n^{\log_2 2} \lg n) \to \Theta(n \lg n)$ ✓

Example 2

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

Example 3

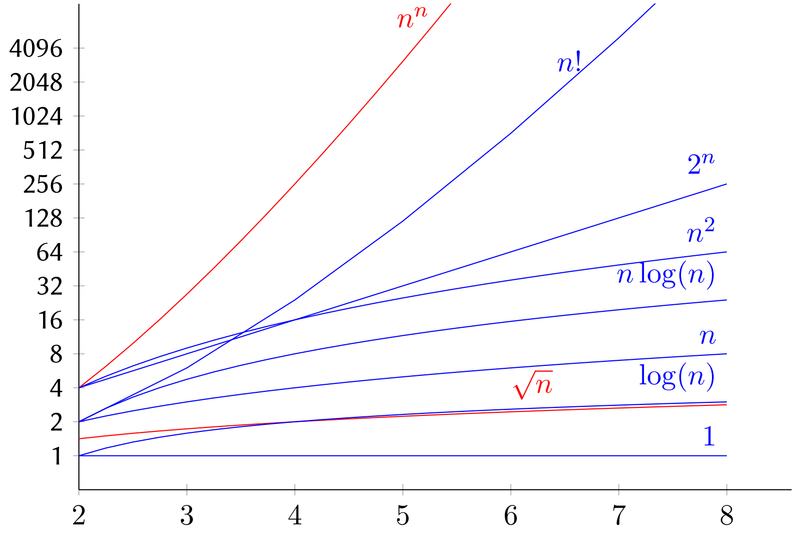
$$T(n) = \sqrt{2}T(n/2) + \log n$$

Example 3

$$T(n) = \sqrt{2}T(n/2) + \log n$$

1. $a = \sqrt{2}; b = 2$
2. $f(n) = \log n$
3. How does $f(n)$ compare to $n^{\log_b a} \rightarrow n^{\log_2 \sqrt{2}} \rightarrow n^{1/2} \rightarrow \sqrt{n}$?
4. $f(n) = O(\sqrt{n})$, so Case 1 applies.
5. $\Rightarrow T(n) = \Theta(n^{\log_b a}) \rightarrow \Theta(\sqrt{n})$ ✓

Growth of Functions



Example 4

$$T(n) = 4T(n/2) + n/\log n$$

Example 4

$$T(n) = 4T(n/2) + n/\log n$$

1. $a = 4; b = 2$
2. $f(n) = \frac{n}{\log n}$
3. How does $f(n)$ compare to $n^{\log_2 4} \rightarrow n^2$?
4. $\frac{n}{\log n} = O(n^2)$, so Case 1 applies.
5. $\Rightarrow T(n) = \Theta(n^{\log_b a}) \rightarrow \Theta(n^2)\checkmark$

Example 5

$$T(n) = 3T(n/4) + n \log n$$

Example 5

$f(n) = n \log n$ $\quad 4^0 = 1$ $\quad 4^1 = 4$ $\quad \log_4 3 = 0.79\ldots$

$n^{\log_b a} = n^{\log_4 3} = n^{0.79}$

$$T(n) = 3T(n/4) + \underline{n \log n}$$

1. $a = 3; b = 4$

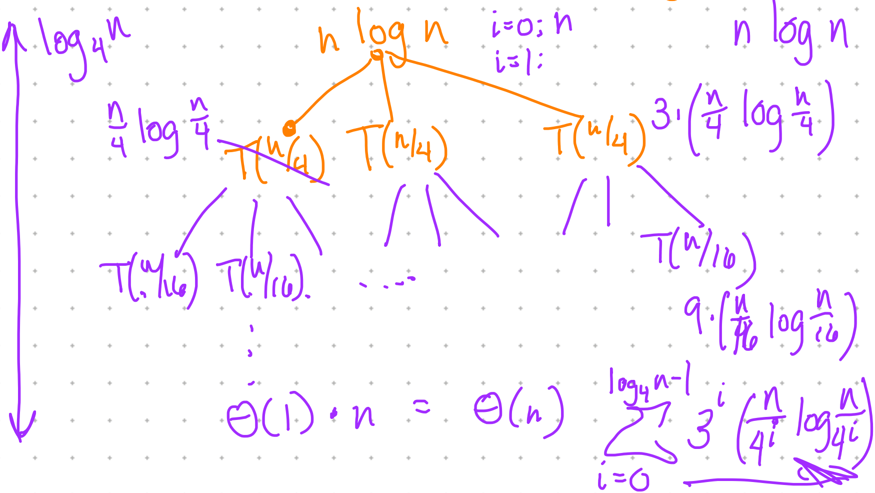
$\underline{n \log n}$ $\boxed{\Omega}$ $n^{0.79}$ $\quad$ Case 3

$f(n) = \Omega(n^{0.79})$

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

Evening

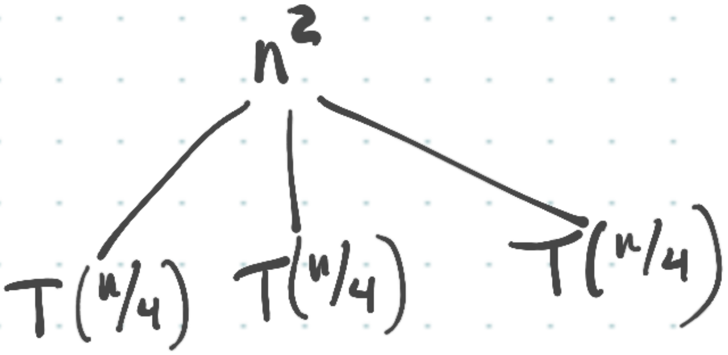
$T(n) = 3T(n/4) + n\log n$

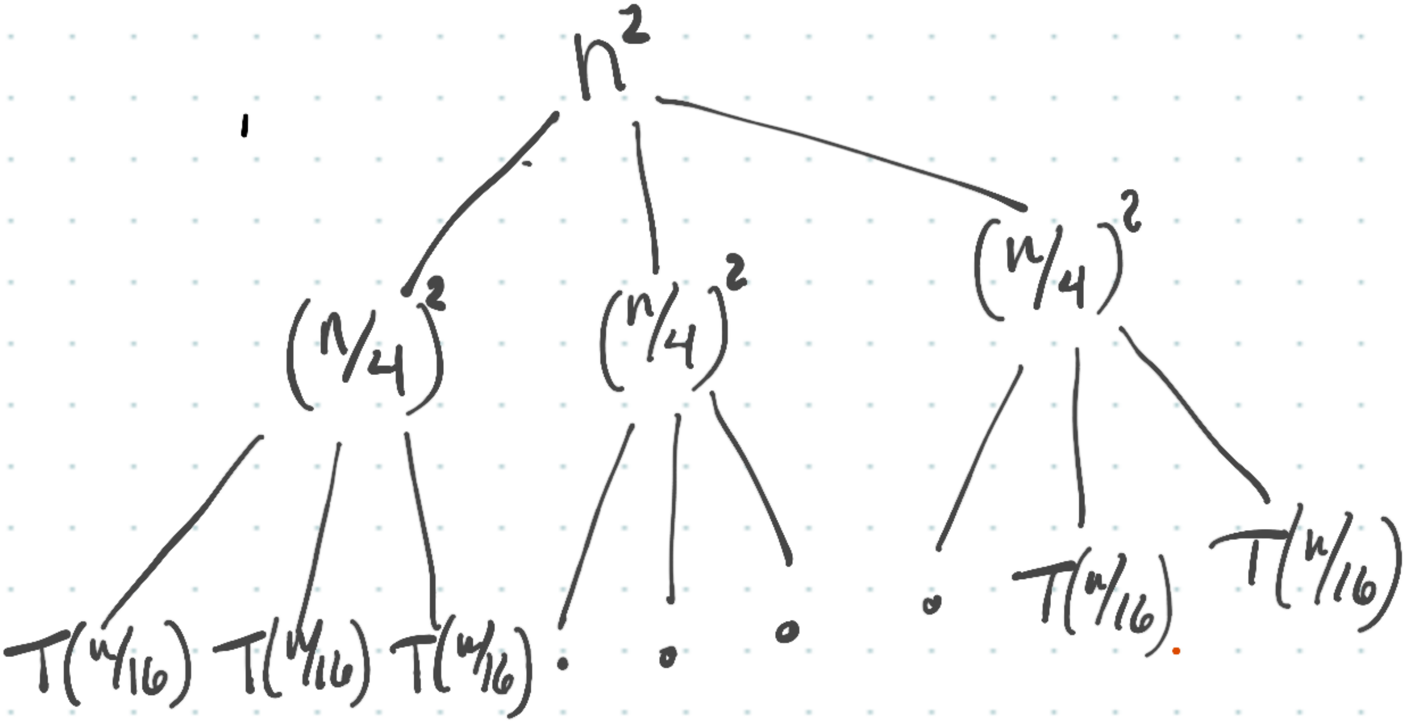

$\log_4 n$
$n\log n$
$i=0: n$
$i=1:$
$n\log n$
$\frac{n}{4}\log\frac{n}{4}$
$T(n/4)$
$T(n/4)$
$T(n/4)$
$3\cdot\left(\frac{n}{4}\log\frac{n}{4}\right)$
$T(n/16)$
$T(n/16)$
$\cdots$
$T(n/16)$
$9\cdot\left(\frac{n}{16}\log\frac{n}{16}\right)$
$\Theta(1)\cdot n = \Theta(n)$
$\sum_{i=0}^{\log_4 n-1} 3^i\left(\frac{n}{4^i}\log\frac{n}{4^i}\right)$

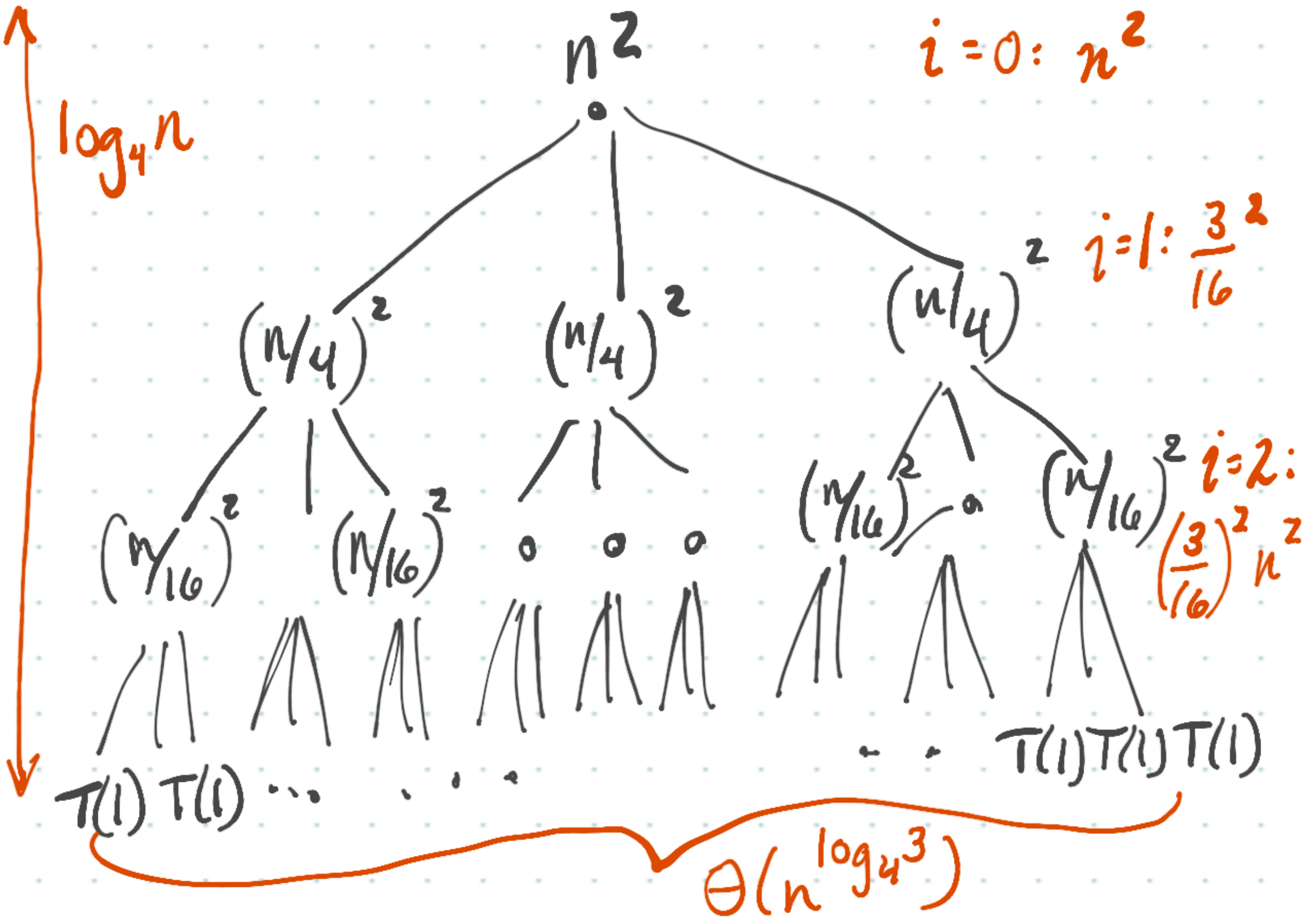
Evening

$$T(n) = 3T(\lfloor n/4 \rfloor) + \Theta(n^2)$$

- drop the floor







Height of tree: smallest input when $\frac{n}{4^i}=1$.

$\Longrightarrow \log_4 n + 1$ levels.

Work done at each level: $\left(\frac{3}{16}\right)^i n^2$

Work done in tree: $\sum_{i=0}^{\log_4 n - 1} \left(\frac{3}{16}\right)^i n^2 = \frac{(3/16)^{\log_4 n} - 1}{(3/16)-1} n^2$

Work done at leaves: $\Theta(n^{\log_4 3})$

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

Example 5

$$T(n) = 3T(n/4) + n \log n$$

1. $a = 3; b = 4$
2. $f(n) = n \log n$
3. How does $f(n)$ compare to $n^{\log_4 3} \rightarrow n^{0.79\ldots}$?
4. $f(n)$ grows faster/is bigger, $\Rightarrow$ Case 3 applies
5. $\Rightarrow T(n) = \Theta(f(n)) = \Theta(n \log n)$✓

Example 6

$$.T(n) = 2T(n/4) + n^{0.51}$$

Example 6

$$.T(n) = 2T(n/4) + n^{0.51}$$

1. $a = 2; b = 4$
2. $f(n) = n^{0.51}$
3. How does $f(n)$ compare to $n^{\log_4 2} \rightarrow n^{1/2} \rightarrow n^{0.5}$?
4. $f(n) = \Omega(\sqrt{n}) \Rightarrow$ Case 3 applies
5. $\Rightarrow T(n) = \Theta(n^{0.51})$

Example 6

$$.T(n) = 2T(n/4) + n^{0.51}$$

1. $a = 2; b = 4$
2. $f(n) = n^{0.51}$
3. How does $f(n)$ compare to $n^{\log_4 2} \rightarrow n^{1/2} \rightarrow n^{0.5}$?
4. $f(n) = \Omega(\sqrt{n}) \Rightarrow$ Case 3 applies
5. $\Rightarrow T(n) = \Theta(n^{0.51})\checkmark$

Example 7

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

Example 7

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

1. $a = 4; b = 2$
2. $f(n) = n^2$
3. How does $f(n)$ compare to $n^{\log_2 4} \rightarrow n^2$?
4. $f(n) = \Theta(n^2) \Rightarrow$ Case 2 applies
5. $T(n) = \Theta(n^{\log_b a} \lg n) \rightarrow \Theta(n^2 \lg n)$

Example 8

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

2

Example 8

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

1. $a = 1; b = 2$
2. $f(n) = n(2 - \cos n)$
3. How does $f(n)$ compare to $n^{\log_2 1} \to n^0 \to 1$?
4. $f(n) = \Omega(1) \Rightarrow$ Case 3 applies
5. BUT! Does this constraint hold?
   $af(n/b) \leq cf(n)$ for some $c < 1$
6. No[2] $\Rightarrow$ Master theorem does not apply.

[2] Consider $n = 2\pi k$, where k is odd and arbitrarily large. For any such choice of n, you can show that $c \geq 3/2$, thereby violating the regularity condition.

Subsection 4

Applying to an algorithm

Binary Search

The runtime of Binary Search can be defined by the recurrence:

$$T(n) = T(n/2) + O(1)$$

Binary Search

The runtime of Binary Search can be defined by the recurrence:

$$T(n) = T(n/2) + O(1)$$

1. $a = 1; b = 2$

Binary Search

The runtime of Binary Search can be defined by the recurrence:

$$T(n) = T(n/2) + O(1)$$

1. $a = 1; b = 2$
2. $f(n) = O(1)$

Binary Search

The runtime of Binary Search can be defined by the recurrence:

$$T(n) = T(n/2) + O(1)$$

1. $a = 1; b = 2$
2. $f(n) = O(1)$
3. How does $f(n)$ compare to $n^{\log_2 1} \rightarrow n^0 \rightarrow 1$?

Binary Search

The runtime of Binary Search can be defined by the recurrence:

$$T(n) = T(n/2) + O(1)$$

1. $a = 1; b = 2$
2. $f(n) = O(1)$
3. How does $f(n)$ compare to $n^{\log_2 1} \rightarrow n^0 \rightarrow 1$?
4. $f(n) = \Theta(1) \Rightarrow$ Case 2 applies

Binary Search

The runtime of Binary Search can be defined by the recurrence:

$$T(n) = T(n/2) + O(1)$$

1. $a = 1; b = 2$
2. $f(n) = O(1)$
3. How does $f(n)$ compare to $n^{\log_2 1} \rightarrow n^0 \rightarrow 1$?
4. $f(n) = \Theta(1) \Rightarrow$ Case 2 applies
5. $T(n) = \Theta(n^{\log_b a} \lg n) \rightarrow \Theta(\lg n)$

- Introduce the whole Master Theorem
- Break down the Master Theorem and specify definitions
- Restate the Master Theorem to develop intuition
- Use the Master Theorem to solve some recurrences
- Use the Master Theorem to analyze an algorithm

Subsection 5

Generating some intuition

When does Case 1 hold?

Case 1 holds when there are “too many” leaves.

- The time taken to calculate the subproblems outweighs the time it takes to split input and combine the results.
- The overall recurrence is more or less bounded by $n^{\log_b a}$

When does Case 2 hold?

Case 2 holds when there is about the same amount of work done at each level.

- $\Theta(n)$ means that $f(n)$ is about the same as $n^{\log_b a}$
- As you move down the tree, each problem gets smaller, but there are more to solve.
- If the sum of the internal evaluation costs (that is, $f(n)$ at each level are equal, the total running time is the cost per level $(n^{\log_b a})$ times the number of levels $(\log_b n)$ for a total running time of $O(n^{\log_b a} \lg n)$

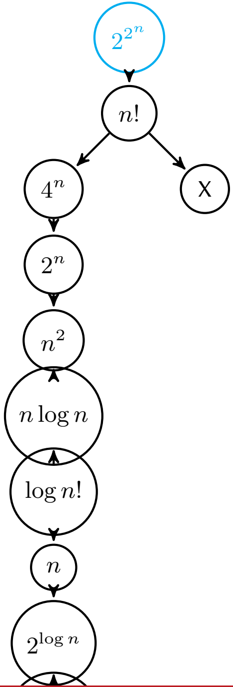
When does Case 3 hold?

Case 3 holds when the “root” is too time consuming.

- $f(n)$ is bigger/grows faster than $n^{\log_b a+c}$
- The size of the overall recurrence is dominated by $f(n)$.
- If the internal evaluation costs grow rapidly enough with n, then the cost of the root evaluation may dominate. Of so, the total running time is $O(f(n))$.

Representative problems

- Case 1 holds for heap construction and matrix multiplication
- Case 2 is for mergesort and binary sort
- Case 3 arises for more awkward algorithms, where the cost of combining the sub-parts dominates everything else


2^{2^n}
$n!$
4^n
X
2^n
n^2
$n \log n$
$\log n!$
n
$2^{\log n}$

Notes on Solving Recurrences

- We usually ignore floors, ceilings, and boundary conditions
- We usually assume that $T(n)$ for a small enough n is $\Theta(1)$ (constant)
    - Changing the value of $T(1)$ doesn't usually change the solution of the recurrence enough to change the order of growth.

Break

Mergesort: Correctness Proof

Slight aside: Induction proofs

Section 6

Induction Proofs

Subsection 1

Situating the Problem

Consider the Equation

$$\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$$

Consider the Equation

$$\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$$

How do we prove this true?

Consider the Equation: It’s true for some numbers…

$$\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$$

Case $n = 1 : \sum_{i=1}^{1} i =$

Consider the Equation: It's true for some numbers...

$$\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$$

Case $n = 1 : \sum_{i=1}^{1} i =$

Case $n = 5 : \sum_{i=1}^{5} i =$

How do we prove this true?

Consider the Equation: It’s true for some numbers...

$$\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$$

$$\text{Case } n = 1 : \sum_{i=1}^{1} i =$$

$$\text{Case } n = 5 : \sum_{i=1}^{5} i =$$

$$\text{Case } n = 30 : \sum_{i=1}^{30} i =$$

How do we prove this true?

Just because we proved this true for a couple of instances doesn’t mean we’ve proved it!

Subsection 2

Mathematical Induction

Mathematical Induction

- Prove the formula for the smallest number that can be used in the given statement.
- Assume it's true for an arbitrary number n.
- Use the previous steps to prove that it's true for the next number $n + 1$.

Step 1: Proving true for smallest number

$$\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$$

$$\text{Case } n = 1 : \sum_{i=1}^{1} i =$$

Step 2: Assume true for arbitrary n

Assumed.

Proof: Summing n integers

$$\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$$

Proof:

- Does it hold true for $n = 1$?
  $1 = \frac{1(1+1)}{2}$ ✓
- Assume it works for n ✓
- Prove that it's true when n is replaced by $n + 1$

Proof Step 3: Summing n integers

Starting with n:

$$\sum_{i=1}^{n} i = \frac{n(n+1)}{2} \tag{1}$$

(6)

Proof Step 3: Summing n integers

Rewriting the left hand side...

$$\sum_{i=1}^{n} i = \frac{n(n+1)}{2} \tag{1}$$

$$1 + 2 + 3 + ...(n-1) + n = \frac{n(n+1)}{2} \tag{2}$$

(6)

Proof Step 3: Summing n integers

Replace n with $n+1$

$$\sum_{i=1}^{n} i = \frac{n(n+1)}{2} \tag{1}$$

$$1+2+3+...(n-1)+n = \frac{n(n+1)}{2} \tag{2}$$

$$1+2+3+...+((n+1)-1)+(n+1) = \frac{(n+1)[(n+1)+1]}{2} \tag{3}$$

(6)

Proof Step 3: Summing n integers

Simplifying

$$\sum_{i=1}^{n} i = \frac{n(n+1)}{2} \tag{1}$$

$$1+2+3+...(n-1)+n = \frac{n(n+1)}{2} \tag{2}$$

$$1+2+3+...+((n+1)-1)+(n+1) = \frac{(n+1)[(n+1)+1]}{2} \tag{3}$$

$$1+2+3+...+n+(n+1) = \frac{(n+1)(n+2)}{2} \tag{4}$$

$$\tag{6}$$

Proof Step 3: Summing n integers

Re-grouping on the left side

$$\sum_{i=1}^{n} i = \frac{n(n+1)}{2} \tag{1}$$

$$1+2+3+...(n-1)+n = \frac{n(n+1)}{2} \tag{2}$$

$$1+2+3+...+((n+1)-1)+(n+1) = \frac{(n+1)[(n+1)+1]}{2} \tag{3}$$

$$1+2+3+...+n+(n+1) = \frac{(n+1)(n+2)}{2} \tag{4}$$

$$(1+2+3+...+n)+(n+1) = \frac{(n+1)(n+2)}{2} \tag{5}$$

$$\tag{6}$$

Proof Step 3: Summing n integers

Replace our known (assumed) formula from #2

$$\sum_{i=1}^{n} i = \frac{n(n+1)}{2} \tag{1}$$

$$1 + 2 + 3 + ...(n-1) + n = \frac{n(n+1)}{2} \tag{2}$$

$$1 + 2 + 3 + ... + ((n+1)-1) + (n+1) = \frac{(n+1)[(n+1)+1]}{2} \tag{3}$$

$$1 + 2 + 3 + ... + n + (n+1) = \frac{(n+1)(n+2)}{2} \tag{4}$$

$$(1 + 2 + 3 + ... + n) + (n+1) = \frac{(n+1)(n+2)}{2} \tag{5}$$

$$\frac{n(n+1)}{2} + (n+1) = \frac{(n+1)(n+2)}{2} \tag{6}$$

Proof Step 3: Summing n integers (pt 2)

Established a common denominator

$$\frac{n(n+1)}{2} + \frac{2(n+1)}{2} = \frac{(n+1)(n+2)}{2} \tag{7}$$

(9)

Proof Step 3: Summing n integers (pt 2)

Simplify

$$
\begin{align}
\frac{n(n+1)}{2} + \frac{2(n+1)}{2} &= \frac{(n+1)(n+2)}{2} \tag{7} \\
\frac{n(n+1)+2(n+1)}{2} &= \frac{(n+1)(n+2)}{2} \tag{8} \\
& \tag{9}
\end{align}
$$

Proof Step 3: Summing n integers (pt 2)

Factor out common factor $n + 1$

$$\begin{align}\frac{n(n+1)}{2} + \frac{2(n+1)}{2} &= \frac{(n+1)(n+2)}{2} \tag{7}\\ \frac{n(n+1) + 2(n+1)}{2} &= \frac{(n+1)(n+2)}{2} \tag{8}\\ \frac{(n+1)(n+2)}{2} &= \frac{(n+1)(n+2)}{2} \tag{9}\end{align}$$

Proof Step 3: Summing n integers (pt 2)

$$\frac{n(n+1)}{2} + \frac{2(n+1)}{2} = \frac{(n+1)(n+2)}{2} \tag{7}$$

$$\frac{n(n+1) + 2(n+1)}{2} = \frac{(n+1)(n+2)}{2} \tag{8}$$

$$\frac{(n+1)(n+2)}{2} = \frac{(n+1)(n+2)}{2} \qquad \checkmark \tag{9}$$

Proof Step 3: Summing n integers (pt 2)

$$\begin{aligned}
\frac{n(n+1)}{2} + \frac{2(n+1)}{2} &= \frac{(n+1)(n+2)}{2} \qquad\qquad \tag{7}\\
\frac{n(n+1)+2(n+1)}{2} &= \frac{(n+1)(n+2)}{2} \tag{8}\\
\frac{(n+1)(n+2)}{2} &= \frac{(n+1)(n+2)}{2} \qquad\qquad \checkmark \tag{9}
\end{aligned}$$

We've proved that the formula holds for $n + 1$.

Proof: Summing n integers

$$\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$$

Proof:

- Does it hold true for $n = 1$?
  $1 = \frac{1(1+1)}{2}$ ✓
- Assume it works for n ✓
- Prove that it's true when n is replaced by $n + 1$ ✓

Mathematical Induction

- Prove the formula for a base case
- Assume it's true for an arbitrary number n
- Use the previous steps to prove that it's true for the next number $n+1$

Subsection 3

Building block: The Well-Ordering Property

The Well-Ordering Property

The Well-Ordering property

The positive integers are *well-ordered*. An ordered set is ***well-ordered*** if each and every nonempty subset has a smallest or least element. Every nonempty subset of the positive integers has a least element.

Note: this property is not true for the set of integers (in which there are arbitrarily small negative numbers) or subsets of, e.g., the positive real numbers (in which there are elements arbitrarily close to zero).

The Well-Ordering Principle

An equivalent statement to the well-ordering principle is as follows:
The set of positive integers does not contain any infinite strictly decreasing sequences.

Proving Well-Ordered Principle with Induction[3]

Let S be a subset of the positive integers with no least element.

Clearly $1 \notin S$ since it would be the least element if it were.

Let T be the complement of S, so $1 \in T$.

Now suppose every positive integer $\leq n$ is in T. Then if $n+1 \in S$ it would be the least element of S since every integer smaller than $n+1$ is in the complement of S.

This is not possible, so $n+1 \in T$ instead.

This implies that every positive integer is in T by strong induction.

Therefore, S is the empty set. ✓

[3] adapted from: https://brilliant.org/wiki/the-well-ordering-principle/

Proving Induction with the Well-Ordered Principle

Suppose P is a property of an integer such that $P(1)$ is true, and $P(n)$ being true implies that $P(n+1)$ is true.

Let S be the set of integers k such that $P(k)$ is false.

Suppose S is nonempty and let k be its least element.

Since $P(1)$ is true$1 \notin S$ so $k \neq 1$ so $k - 1$ is a positive integer, and by minimality $k - 1 \notin S$.

So by definition $P(k-1)$ is true, but then by the property of P given above, $P(k-1)$ being true implies that $P(k)$ is true.

So $k \notin S$; contradiction.

So S is empty; so $P(k)$ is true for all k. ✓

Back to proving Mergesort correct

Subsection 4

Using Induction to Prove Recursive Algorithms Correct

Recursion

A quick review on recursion:

- Test whether input is a *base case.*
- If not, break the input into smaller pieces and re-call the function with the smaller pieces
- Combine the smaller pieces together

Recursion: Example

MERGE SORT

- MERGESORT one half
- MERGESORT the other
- MERGE
  - Put them together “in the right order”

Recursion: Example

Merge sort

- MergeSort one half
- MergeSort the other
- Merge
  - Put them together "in the right order"

What's the base case?

Recursion: Example

MERGE SORT

- MERGESORT one half
- MERGESORT the other
- MERGE
  - Put them together “in the right order”

What’s the base case?

Input is 1 element (that is, low=high

Mergesort: Proof of Correctness

MergeSort

- Prove the formula for the smallest number that can be used in the given statement.
    - In algorithm-speak: Prove the algorithm correct for the base case

Mergesort: Proof of Correctness

MergeSort

- Prove the formula for the smallest number that can be used in the given statement.
    - In algorithm-speak: Prove the algorithm correct for the base case
    - If the input is one element, it's sorted, trivially.

Mergesort: Proof of Correctness

MergeSort

- Prove the formula for the smallest number that can be used in the given statement.
  - In algorithm-speak: Prove the algorithm correct for the base case
  - If the input is one element, it's sorted, trivially.
  - If the input is 2 elements, merge ensures that the two elements get sorted properly.

Mergesort: Proof of Correctness

MERGESORT

- Prove the formula for the smallest number that can be used in the given statement.
    - In algorithm-speak: Prove the algorithm correct for the base case
    - If the input is one element, it's sorted, trivially.
    - If the input is 2 elements, merge ensures that the two elements get sorted properly.
- Assume it's true for an arbitrary number n.

Mergesort: Proof of Correctness

MergeSort

- Prove the formula for the smallest number that can be used in the given statement.
  - In algorithm-speak: Prove the algorithm correct for the base case
  - If the input is one element, it's sorted, trivially.
  - If the input is 2 elements, merge ensures that the two elements get sorted properly.
- Assume it's true for an arbitrary number n.
  - In algorithm-speak: Assume it works for an arbitrarily sized input of size n.

Mergesort: Proof of Correctness

MergeSort

- Prove the formula for the smallest number that can be used in the given statement.
  - In algorithm-speak: Prove the algorithm correct for the base case
  - If the input is one element, it's sorted, trivially.
  - If the input is 2 elements, merge ensures that the two elements get sorted properly.
- Assume it's true for an arbitrary number n.
  - In algorithm-speak: Assume it works for an arbitrarily sized input of size n.
- Use the previous steps to prove that it's true for the next number $n + 1$.

Mergesort: Proof of Correctness

MergeSort

- Prove the formula for the smallest number that can be used in the given statement.
    - In algorithm-speak: Prove the algorithm correct for the base case
    - If the input is one element, it's sorted, trivially.
    - If the input is 2 elements, merge ensures that the two elements get sorted properly.
- Assume it's true for an arbitrary number n.
    - In algorithm-speak: Assume it works for an arbitrarily sized input of size n.
- Use the previous steps to prove that it's true for the next number $n+1$.
    - In algorithm-speak: Use the above to prove that the algorithm works when you add another element to the input.

Mergesort: Proof of Correctness

MERGESORT

- Prove the formula for the smallest number that can be used in the given statement.
  - In algorithm-speak: Prove the algorithm correct for the base case
  - If the input is one element, it's sorted, trivially.
  - If the input is 2 elements, merge ensures that the two elements get sorted properly.
- Assume it's true for an arbitrary number n.
  - In algorithm-speak: Assume it works for an arbitrarily sized input of size n.
- Use the previous steps to prove that it's true for the next number $n+1$.
  - In algorithm-speak: Use the above to prove that the algorithm works when you add another element to the input.
  - When we call MERGESORT on an array of size n (or $n+1$, same thing), it recursively calls MERGESORT on input of size $n/2$

Mergesort: Proof of Correctness

MERGESORT

- Prove the formula for the smallest number that can be used in the given statement.
  - In algorithm-speak: Prove the algorithm correct for the base case
  - If the input is one element, it's sorted, trivially.
  - If the input is 2 elements, merge ensures that the two elements get sorted properly.
- Assume it's true for an arbitrary number n.
  - In algorithm-speak: Assume it works for an arbitrarily sized input of size n.
- Use the previous steps to prove that it's true for the next number $n + 1$.
  - In algorithm-speak: Use the above to prove that the algorithm works when you add another element to the input.
  - When we call MERGESORT on an array of size n (or $n + 1$, same thing), it recursively calls MERGESORT on input of size $n/2$
  - Since we assumed that MERGESORT works, as long as Merge works, MERGESORT works.

Other Divide and Conquer problems

Related problems

- Counting Inversions (info in the notes)
- Closest Points
- Integer Multiplication
- Convolutions/FFT

Summary

What problems did we work on today?

- Sorting

What approaches did we use?

- Divide and Conquer

What tools did we use?

- Recurrences (to characterize run time of recursive algorithms)
- Solving recurrences
    - Finding an upper bound/estimate
    - Substitution
    - Recurrence trees/Iteration/unrolling
- Mathematical Induction
    - Practice the concept in math
    - Apply the concept in proving recursive algorithms correct
    - Apply the concept in using substitution for solving recurrences

1. A first problem: Sorting
   - Sorting Defined
2. Divide and Conquer
   - Using Divide and Conquer to Merge Sort
   - Merge Sort Algorithm
   - Mergesort Graphically
   - Mergesort Analysis
   - Divide and Conquer Runtime (Generally)
3. Solving Recurrences
   - Recursion Trees/“Unrolling”
4. Master Theorem
   - Defining the Master Theorem
   - Restating the Master Theorem casually
   - Examples
   - Applying to an algorithm
   - Generating some intuition
5. Mergesort: Correctness
   - Mathematical Induction
6. Induction Proofs
   - Situating the Problem
   - Mathematical Induction
   - Building block: The Well-Ordering Property
   - Using Induction to Prove Recursive Algorithms Correct
7. Other Divide and Conquer Problems
8. Summary
   - Substitution

Extra Slides

Substitution

Using substitution starts with a simple premise:

- Guess “the form of the solution”
- Use induction to find the constants and show it works

Example: Substitution Method

Recurrence:

Example: Substitution Method

Let’s come up with a bound for this recurrence:

$$\textit{\textbf{Original Recurrence:}}\ T(n) = 2T(\lfloor n/2 \rfloor) + n \tag{1}$$

(9)

Example: Substitution Method

Step 1: Make a guess:

$$\textit{Original Recurrence:}\ T(n) = 2T(\lfloor n/2 \rfloor) + n \tag{1}$$

$$\textit{Guess:}\ T(n) = O(n \lg n) \tag{2}$$

(9)

Example: Substitution Method

Now, we need to show #3, per definition of Big-O.
Apply inductive step, and assume that $T(n) <= cn \lg n$ for all $m < n$, in particular $m = \lfloor n/2 \rfloor$.

$$\textit{\textbf{Original Recurrence:}}\ T(n) = 2T(\lfloor n/2 \rfloor) + n \qquad (1)$$

$$\textit{\textbf{Guess:}}\ T(n) = O(n \lg n) \qquad (2)$$

$$T(n) \leq cn \lg n \qquad (3)$$

$$(9)$$

Example: Substitution Method

Substitute $\lfloor n/2 \rfloor$ into the recurrence, since we know that
$T(n) \leq cn \lg n forn = \lfloor n/2 \rfloor$:

$$\textbf{\textit{Original Recurrence:}}\ T(n) = 2T(\lfloor n/2 \rfloor) + n \tag{1}$$

$$\textbf{\textit{Guess:}}\ T(n) = O(n \lg n) \tag{2}$$

$$T(n) \leq cn \lg n \tag{3}$$

$$T(\lfloor n/2 \rfloor) \leq c\lfloor n/2 \rfloor \lg(\lfloor n/2 \rfloor) \tag{4}$$

(9)

Example: Substitution Method

Multiply both sides by 2 and add n

$$\textit{Original Recurrence:}\ T(n) = 2T(\lfloor n/2 \rfloor) + n \tag{1}$$

$$\textit{Guess:}\ T(n) = O(n \lg n) \tag{2}$$

$$T(n) \leq cn \lg n \tag{3}$$

$$T(\lfloor n/2 \rfloor) \leq c\lfloor n/2 \rfloor \lg(\lfloor n/2 \rfloor) \tag{4}$$

$$T(n) = 2T(n\lfloor n/2 \rfloor) + n \leq 2(c\lfloor(n/2\rfloor \lg(\lfloor n/2 \rfloor)) + n \tag{5}$$

(9)

Example: Substitution Method

Multiply the 2 into the first term. It's $\leq$ because $n/2 \leq floor(n/2)$.

$$\textbf{\textit{Original Recurrence:}}\ T(n) = 2T(\lfloor n/2 \rfloor) + n \tag{1}$$

$$\textbf{\textit{Guess:}}\ T(n) = O(n \lg n) \tag{2}$$

$$T(n) \leq cn \lg n \tag{3}$$

$$T(\lfloor n/2 \rfloor) \leq c\lfloor n/2 \rfloor \lg(\lfloor n/2 \rfloor) \tag{4}$$

$$T(n) = 2T(n\lfloor n/2 \rfloor) + n \leq 2(c\lfloor (n/2 \rfloor \lg(\lfloor n/2 \rfloor)) + n \tag{5}$$

$$\leq cn \lg(n/2) + n \tag{6}$$

$$\tag{9}$$

Example: Substitution Method

Recall, we need to show a solution that looks like line 3. Do something about that $n/2$ term. Let's use the log rule! (TODO: PUT THE REF HERE)

$$\log_b(1/a) = -\log_b a$$

$$
\begin{aligned}
\textbf{\textit{Original Recurrence:}}\ T(n) &= 2T(\lfloor n/2\rfloor) + n && (1)\\
\textbf{\textit{Guess:}}\ T(n) &= O(n \lg n) && (2)\\
T(n) &\le cn \lg n && (3)\\
T(\lfloor n/2\rfloor) &\le c\lfloor n/2\rfloor \lg(\lfloor n/2\rfloor) && (4)\\
T(n) = 2T(n\lfloor n/2\rfloor) + n &\le 2(c\lfloor(n/2\rfloor \lg(\lfloor n/2\rfloor)) + n && (5)\\
&\le cn \lg(n/2) + n && (6)\\
&= cn \lg n - cn \lg 2 + n && (7)\\
& && (9)
\end{aligned}
$$

Example: Substitution Method

$$\lg 2 = \log_2 2 \rightarrow \lg 2 = 1$$

$$\begin{align}
\textbf{\textit{Original Recurrence:}}\ T(n) &= 2T(\lfloor n/2 \rfloor) + n \tag{1}\\
\textbf{\textit{Guess:}}\ T(n) &= O(n \lg n) \tag{2}\\
T(n) &\leq cn \lg n \tag{3}\\
T(\lfloor n/2 \rfloor) &\leq c\lfloor n/2 \rfloor \lg(\lfloor n/2 \rfloor) \tag{4}\\
T(n) = 2T(n\lfloor n/2 \rfloor) + n &\leq 2(c\lfloor (n/2 \rfloor \lg(\lfloor n/2 \rfloor)) + n \tag{5}\\
&\leq cn \lg(n/2) + n \tag{6}\\
&= cn \lg n - cn \lg 2 + n \tag{7}\\
&= cn \lg n - cn + n \tag{8}\\
& \tag{9}
\end{align}$$

Example: Substitution Method

We needed to show that $T(n) \leq cn \lg n$, and we've done it!

$$
\begin{align}
\textbf{\textit{Original Recurrence:}}\ T(n) &= 2T(\lfloor n/2 \rfloor) + n \tag{1}\\
\textbf{\textit{Guess:}}\ T(n) &= O(n \lg n) \tag{2}\\
T(n) &\leq cn \lg n \tag{3}\\
T(\lfloor n/2 \rfloor) &\leq c\lfloor n/2 \rfloor \lg(\lfloor n/2 \rfloor) \tag{4}\\
T(n) = 2T(n\lfloor n/2 \rfloor) + n &\leq 2(c\lfloor (n/2 \rfloor \lg(\lfloor n/2 \rfloor)) + n \tag{5}\\
&\leq cn \lg(n/2) + n \tag{6}\\
&= cn \lg n - cn \lg 2 + n \tag{7}\\
&= cn \lg n - cn + n \tag{8}\\
T(n) &\leq cn \lg n \qquad \square \tag{9}
\end{align}
$$