CS 5002: Discrete Structures **Fall 2018**

Lecture 10: November 15, 2018 [1]

Instructors: Adrienne Slaughter, Tamara Bonaci

Disclaimer: *These notes have not been subjected to the usual scrutiny reserved for formal publications. They may be distributed outside this class only with the permission of the Instructor.*

Divide and Conquer

Readings for this week:
Cormen: Algorithms Unlocked, pp 40-59 (mergesort, quicksort)
Rosen: Chapter 8.3: Divide-and-Conquer Algorithms and Recurrence Relations

Objective: Analyzing Divide and Conquer Algorithms

1. Introducing sorting
2. Divide and Conquer
3. Divide and Conquer example: Merge Sort
4. Analyzing Divide and Conquer: Runtime
   - Recurrence Trees
   - Master Theorem
5. Analyzing Divide and Conquer: Correctness
   - Induction

The key to algorithms is understanding different problems that have been solved and how they've been solved. When you come across a new problem, try to think of how it's similar or different than problems you've seen before, and see if a similar approach can solve your problem.

The study of algorithms includes understanding:

- Problems to be solved
- Approaches to designing algorithms
- Approaches to analyzing algorithms

1. 10.1 A first problem: Sorting

Sorting tends to be a first problem for algorithm students, partly because it's very easy to understand, it's very important in the world of computer science overall, and there are many algorithm design and analysis techniques that can be demonstrated by different kinds of sorting algorithms.

In that vein, we'll start with studying a specific sort today, Merge Sort. Merge sort is an example of a ***divide and conquer*** algorithm. To analyze a divide and conquer algorithm, we'll learn two techniques to help: recurrence trees and induction proofs.

Why sorting?

Learning sorts is like learning scales for musicians.

- Lots of other algorithms are built around various sorting algorithms.
- Different ideas around algorithms manifest themselves in different sorting algorithms
  - Divide and conquer
  - Randomization
  - Data structures
  - Recursion
- Computers (historically) have spent more time sorting than anything else.
  - Knuth (in 1998!) claimed that more than 25% of cycles were spent sorting data.
  - Sorting is still one of the most ubiquitous computing problems
- It's the most thoroughly-studied problems in CS.
  - So many algorithms; each has the particular case where it performs better than other algorithms.

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.

10.1.1 Sorting Defined

Definition 10.1 (Sorting) *An array $a[0 \dots n]$ is* ***sorted*** *if $a[i] \leq a[j]$ for all $i < j$ where $0 \leq i < j \leq n$*

Example: $[1, 2, 3, 4, 5]$ *is sorted;* $[2, 5, 3, 1, 4]$ *is not sorted.*

Collections can be sorted in different orders:

Definition 10.2 (Increasing order) *An array $a[0 \dots n]$ is in* ***increasing order*** *if $a[i] < a[j]$ for all $i < j$ where $0 \leq i < j \leq n$*

Example: $[1, 2, 3, 4, 5]$

Definition 10.3 (Decreasing order) *An array $a[0 \dots n]$ is in* ***decreasing order*** *if $a[i] > a[j]$ for all $i < j$ where $0 \leq i < j \leq n$*

Example: $[5, 4, 3, 2, 1]$

Definition 10.4 (Non-Increasing Order) *An array $a[0 \dots n]$ is in* ***non-increasing*** *where $a[i] \geq a[j]$ for all $i < j$ where $0 \leq i < j \leq n$*

Example: $[5, 4, 4, 3, 2, 2, 2]$

Note: Elements can be repeated

Definition 10.5 (Non-Decreasing Order) *An array $a[0 \dots n]$ is in* ***non-decreasing*** *order if $a[i] \leq a[j]$ for all $i < j$ where $0 \leq i < j \leq n$*

Example: $[1, 2, 2, 2, 3, 4, 4, 5]$ *Again, elements can be repeated*

10.2 Divide and Conquer

Divide and Conquer is a common algorithm design approach.

There are three steps to applying divide and conquer to a problem:

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.

10.2.1 Using Divide and Conquer 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.

10.2.2 Merge Sort Algorithm

```
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)
```

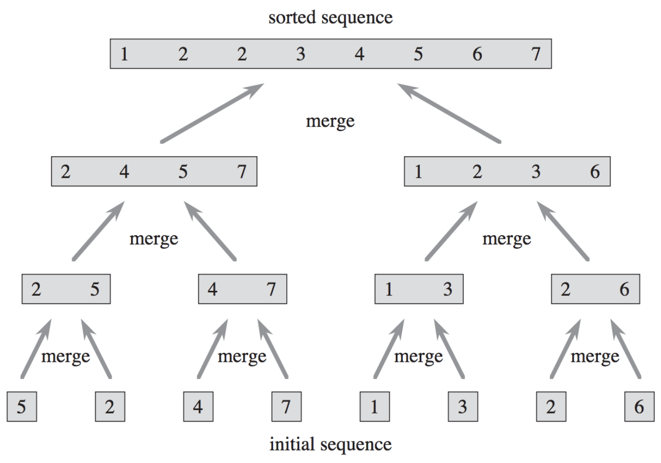
```
MERGE(A, low, mid, high)
L = A[low:mid] (L is a new array copied from A[low:mid])
R = A[mid+1, high]
$i = 1$
$j = 1$
for $k$ =low to high
if L[i] < R[j]:
    A[k] = L[i]
    $i = i + 1$
else
    A[k] = R[j]
    $j = j + 1$
```

10.2.3 Mergesort Example

TODO: put the diagram in

1. Start by specifying you want to sort the entire array: Mergesort(A, 1, 10)
2. Find the mid point. mid = 1 + floor(10/2) - 1 = 5
3. Sort the left side (when we're done sorting the left, we'll sort the right): Mergesort(A, 1, 5)
4. Find the midpoint of the left side mid = 3
5. ... and sort the left of that. Mergesort(A, 1, 3)
6. Find the midpoint, sort the left. Mid = 2; Mergesort(A, 1, 2)
7. Find the midpoint, sort the left.
8. 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.

9. Sort the right. Since $low = high$, we're done with the right.
10. Merge the left and right. Merge(A, 1, 2, 2)
11. Sort the right Mergesort(A, 3, 3)
12. Merge the left and right. Merge(A, 1, 2, 3), which means we're done with Mergesort(A, 1, 3)
13. Sort the right Mergesort(A, 4, 5)
14. Sort the left. Mergesort(A, 4, 4)
15. Sort the right. Mergesort(A, 5, 5)
16. Merge the left and right. Merge(A, 4, 4, 5)
17. Merge the left and right. Merge(A, 1, 3, 5)
18. The left half is sorted. Do the same for the right.
19. Merge the two halves together into a sorted output.



10.2.4 Mergesort Analysis

We do 2 things in our analysis:

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

10.2.4.1 Mergesort: 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:

$$T(\frac{n}{2}) + T(\frac{n}{2}) + T(Merge) \tag{10.1}$$

$$T(Merge) = \Theta(n) \tag{10.2}$$

$$\Rightarrow T(\frac{n}{2}) + T(\frac{n}{2}) + \Theta(n) \tag{10.3}$$

$$T(n) = \begin{cases} \Theta(1) & \text{if } n \leq 1 \\ 2T(\frac{n}{2}) + \Theta(n) & \text{otherwise} \end{cases} \tag{10.4}$$

10.2.5 Divide and Conquer Runtime (Generally)

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:

$$T(n) = \begin{cases} \Theta(1) & \text{if } n \leq c \\ aT(n/b) + D(n) + C(n) & \text{otherwise} \end{cases} \tag{10.5}$$

Mergesort analysis fits this pattern.

$$T(n) = \begin{cases} \Theta(1) & \text{if } n \leq 1 \\ 2T(\frac{n}{2}) + \Theta(n) & \text{otherwise} \end{cases} \tag{10.6}$$

$$a = ? \tag{10.7}$$
$$b = ? \tag{10.8}$$
$$C(n) = ? \tag{10.9}$$
$$D(n) = ? \tag{10.10}$$

10.3 Solving Recurrences

We've seen some ways to solve a recurrence, by coming up with a closed form given an initial condition. Now we're going to see some approaches to use asymptotic analysis to estimate runtime of a recurrence.

There are three approaches to generate a boundary for a recurrence:

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

Today we'll Recursion Trees; Next week we'll look at the Master Method and substituion.

10.3.1 Recursion Trees/"Unrolling"

- A convenient way to visualize what's happening with a recurrence.
- Start with the time it takes to combine at that level, with the next level down being the split input.
- Repeat this process

- Add up the time taken at each level
- Use the depth and time taken at each level to observe/calculate an upper bound on the runtime.

Example: Recursion Tree

$$T(n) = 2T(n/2) + n^2 \tag{10.11}$$

$$\begin{tikzcd}
 & n^2 \arrow[dl] \arrow[dr] & \\
T(\frac{n}{2}) & & T(\frac{n}{2})
\end{tikzcd}$$

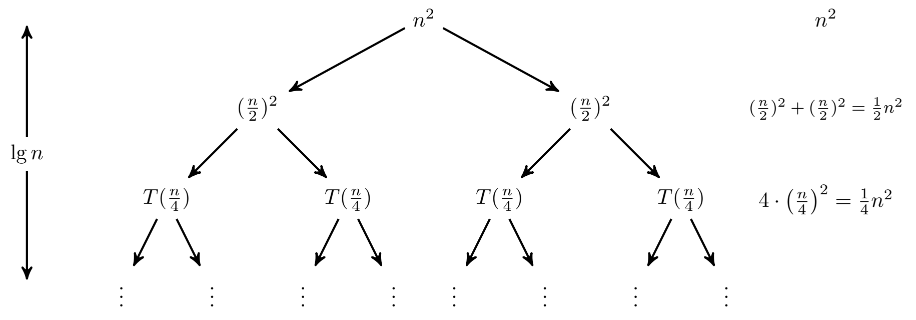
Starting with our recurrence, we draw a tree where the root node represents the time it takes to combine the subproblems.

Since we're solving 2 subproblems (the $2T$ part), there are two children.

Each child gets an input of size $n/2$.

$$\begin{tikzcd}
 & & & n^2 \arrow[dll] \arrow[drr] & & & \\
 & (\frac{n}{2})^2 \arrow[dl] \arrow[dr] & & & & (\frac{n}{2})^2 \arrow[dl] \arrow[dr] & \\
T(\frac{n}{4}) & & T(\frac{n}{4}) & & T(\frac{n}{4}) & & T(\frac{n}{4})
\end{tikzcd}$$

Repeat this again:



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

At each level, we add up the time it takes at each node in the tree. We'll have $\Theta(n^2)$ time.

Since the values decrease geometrically, the total is at most a constant factor more than the largest (first) term. $\Rightarrow$ A bound for the recurrence is $O(n^2)$

Now that we have the tree, we can use it to calculate the amount of work done overall, by adding up the amount of work done at each node.

Big picture:

- 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.

We do that by considering the bottom layer (the base case layer) independently of the other layers. The earlier layers of the recurrence include time spent dividing and combining the input, so take different time than the base case nodes.

With divide and conquer, we break the input down to a base case that can be solved in constant time; that is, $\Theta(1)$. That means each leaf has work done of $\Theta(1)$. Since there are n leaves, the amount of work done at the bottom layer of the tree is $\Theta(1) \cdot n \Rightarrow \Theta(n)$.

Thus, we need to figure out the amount of work done on all the layers above the bottom layers...

The height of the tree is $\lg n$, because we split the input by 2 at each step, until we get to the base case of input size 1.

Amount of work done on each level i: $\frac{1}{2^i}n^2$

Amount of work done on all levels other than the bottom [2]:

$$\sum_{i=0}^{\lg n-1} \frac{1}{2^i}n^2 = n^2 \sum_{i=0}^{\lg n-1} \frac{1}{2^i} \tag{10.12}$$

$$= n^2 \cdot \left(2 - \frac{1}{2^{\lg n-1}}\right) \tag{10.13}$$

$$= n^2 \cdot \left(2 - \frac{1}{2^{\lg n/1}}\right) \text{ Using the log quotient rule} \tag{10.14}$$

$$= n^2 \cdot \left(2 - \frac{1}{n}\right) \text{ Using def of log: } b^{\log_b x} = x \tag{10.15}$$

$$= 2n^2 - n \tag{10.16}$$

$$\sum_{i=0}^{\lg n-1} \frac{1}{2^i}n^2 = \Theta(n^2) \tag{10.17}$$

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:

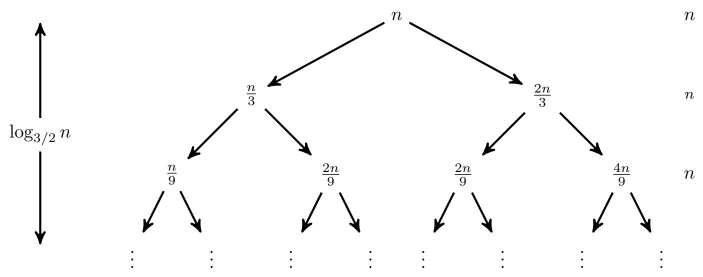
$$T(n) = 2T(n/2) + n^2 = \Theta(n^2) + \Theta(n) \Rightarrow \Theta(n^2). \tag{10.18}$$

A more complex example

We usually don't see something like this, but I'm including it as an example to help show the mechanics and develop intuition about the process.

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

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



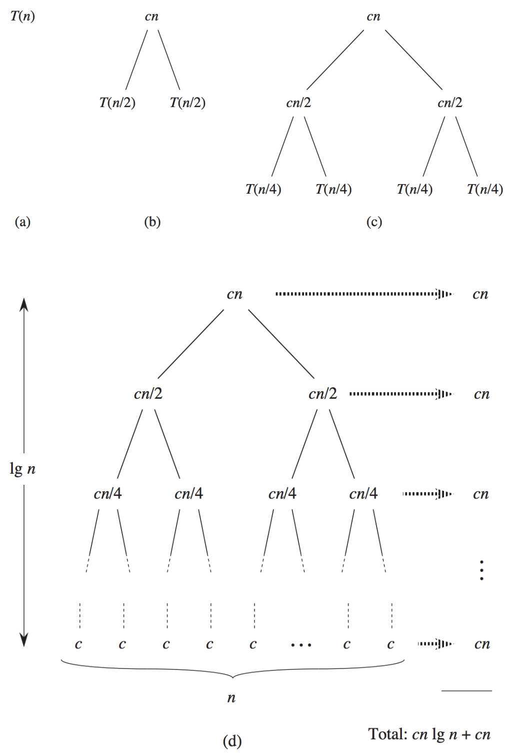
$\Rightarrow$ A bound for the recurrence is $O(n \log n)$
Number of leaves: n
Amount of work done on each leaf: Assume $\Theta(1)$
Amount of work done on the base case layer: $n \cdot \Theta(1) = \Theta(n)$
Height of tree: $\log_{3/2} n$ (We go with the term that takes longer to get to the bottom)
Work on each level of the tree: n
Work overall:

$$\sum_{i=0}^{\log_{3/2} n - 1} n + \Theta(n) = n \cdot (\log_{3/2} n - 1) + \Theta(n) \quad (10.19)$$

$$= n \log_{3/2} n - n + \Theta(n) \quad (10.20)$$

$$\Rightarrow O(n \log n) \quad (10.21)$$

Recurrence tree for Merge Sort

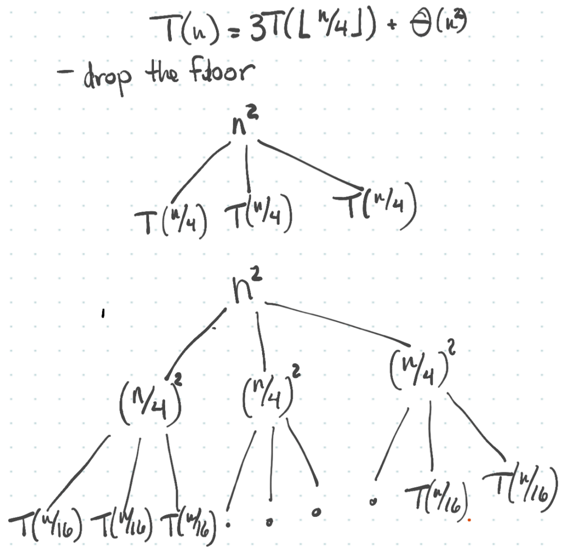


Another Example

Use a recurrence tree to find a bound for the following recurrence:

$$T(n) = 3T(\lfloor n/4 \rfloor) + cn^2 \tag{10.22}$$

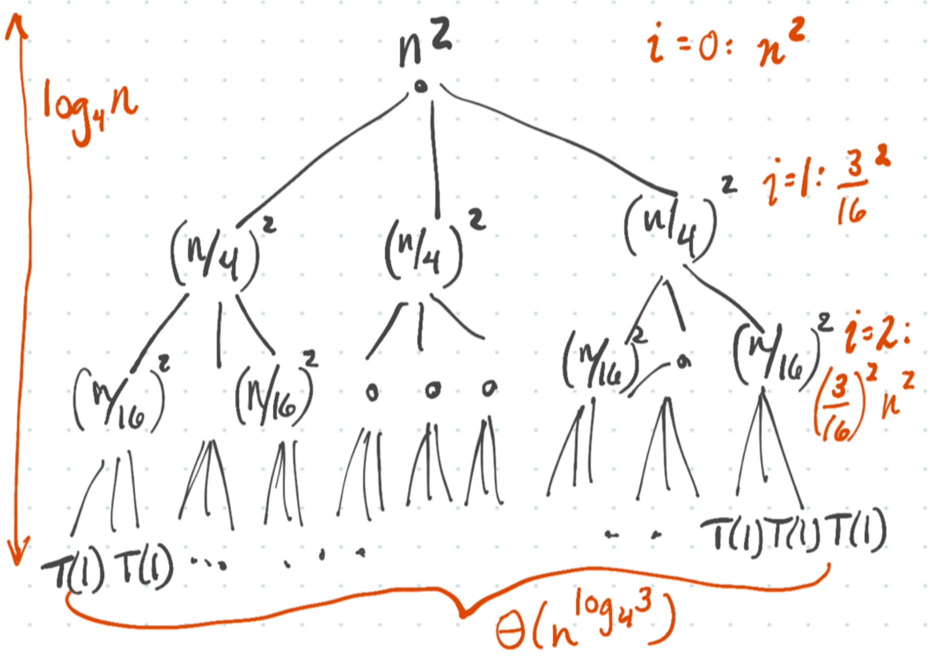
First, we draw the initial recurrence tree:



$T(n) = 3T(\lfloor n/4 \rfloor) + \Theta(n^2)$
- drop the floor
n^2
$T(n/4)$ $T(n/4)$ $T(n/4)$
n^2
$(n/4)^2$ $(n/4)^2$ $(n/4)^2$
$T(n/16)$ $T(n/16)$ $T(n/16)$ $T(n/16)$ $T(n/16)$

Then:

- Draw another layer or two
- Determine the height of the tree
- Determine the amount of work done in each layer
- Define the sum to calculate the amount of work in the tree except at the bottom layer
- Determine the number of nodes at the bottom layer (the number of leaves).
- Use the work done at the bottom layer plus the work done in the rest of the tree to determine a bound.



Height of tree: smallest input when $\frac{n}{4^i}=1$.
$\implies \log_4 n + 1$ levels.

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

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

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

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

10.3.1.1 Helpful notes on the 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:

$$k^h = n \Leftrightarrow \log_k n = h \tag{10.23}$$

$$\tag{10.24}$$

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

- 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.

Solving Recurrences

Three ways to come up with a bound on a recurrence:

- Substitution Method
  - Guess the form of the solution
  - Use mathematical induction to find the constants and show that the solution works.
- Recursion Tree Method
  - Draw a tree to represent the recurrence
  - Each node represents the cost of a subproblem
  - Sum the costs within each level of the tree to obtain a set of per-level costs
  - Sum all the per-level costs to determine the total cost of all levels of the recursion
  - Sometimes the recursion tree can be used to help develop a good guess for the substitution method; you can be sloppier
  - If you're using the recursion tree as a proof, be more careful.
- Master Method
  - Figure out which case of the Master Theorem applies to your recurrence
  - Plug in the numbers to find the bound

10.4 Mergesort: Correctness

10.4.1 Mathematical Induction

Slight aside: Induction proofs

A common approach to proving correctness for recursive algorithms is ***inductive proofs***.

This approach is based on ***mathematical induction***.

We'll go through an intro to mathematical induction, then show how it applies to recursive algorithms.

10.5 Induction Proofs

10.5.1 Situating the Problem

Consider the Equation

$$\sum_{i=1}^{n} i = \frac{n(n+1)}{2} \tag{10.25}$$

How do we prove this true?

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

$$\sum_{i=1}^{n} i = \frac{n(n+1)}{2} \tag{10.26}$$

$$\text{Case } n = 1 : \sum_{i=1}^{1} i = \frac{1(1+1)}{2} \Rightarrow 1 = 1 \tag{10.27}$$

$$\text{Case } n = 5 : \sum_{i=1}^{5} i = \frac{5(5+1)}{2} = 1 + 2 + 3 + 4 + 5 = 15 \Rightarrow 15 \leq 15 \tag{10.28}$$

$$\text{Case } n = 30 : \sum_{i=1}^{30} i = \frac{30(30+1)}{2} \Rightarrow 465 = 465 \text{ Check my math on your own!} \tag{10.29}$$

$$\tag{10.30}$$

How do we prove this true?

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

We see intuitively because $n \cdot n$ is only so big, but you can start to see how $n!$ will continue to dominate as the numbers get bigger.

10.5.2 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

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

Case $n = 1 : \sum_{i=1}^{1} i = \frac{1(1+1)}{2} \Rightarrow 1 = 1 \checkmark$

Step 2: Assume true for arbitrary n

Assumed.✓

Proof Step 3: Prove it's true when n is replaced by $n + 1$

Starting with n:

Rewriting the left hand side...

Replace n with $n + 1$

Simplifying

Re-grouping on the left side

Replace our known (assumed) formula from #2

$$\sum_{i=1}^{n} i = \frac{n(n+1)}{2} \tag{10.31}$$

$$1+2+3+...(n-1)+n = \frac{n(n+1)}{2} \tag{10.32}$$

$$1+2+3+...+((n+1)-1)+(n+1) = \frac{(n+1)[(n+1)+1]}{2} \tag{10.33}$$

$$1+2+3+...+n+(n+1) = \frac{(n+1)(n+2)}{2} \tag{10.34}$$

$$(1+2+3+...+n)+(n+1) = \frac{(n+1)(n+2)}{2} \tag{10.35}$$

$$\frac{n(n+1)}{2}+(n+1) = \frac{(n+1)(n+2)}{2} \tag{10.36}$$

Proof Step 3: Summing n integers (pt 2)

Established a common denominator Simplify Factor out common factor $n+1$

$$\frac{n(n+1)}{2}+\frac{2(n+1)}{2} = \frac{(n+1)(n+2)}{2} \tag{10.37}$$

$$\frac{n(n+1)+2(n+1)}{2} = \frac{(n+1)(n+2)}{2} \tag{10.38}$$

$$\frac{(n+1)(n+2)}{2} = \frac{(n+1)(n+2)}{2} \qquad \checkmark \tag{10.39}$$

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 $n+1$

10.5.3 Building block: The Well-Ordering Principle

The Well-Ordering Principle

The Well-Ordering principle

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.

TODO: definition!

Note that this property is not true for subsets of the integers (in which there are arbitrarily small negative numbers) or 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. ✓

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

10.5.4 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"

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

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
  - 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.

10.6 Recursion vs Divide and Conquer

What is recursion?

- Define a function in terms of itself
- The call needs to ensure that parameters change/make progress toward the base case.
- Need to ensure the function returns on the base case

Recursive procedure: Example

Factorial: $n! = n \cdot (n-1) \cdot (n-2) \cdot \ldots \cdot 1$

```

int factorial(n){
  if (n=1):
    return 1;
  return n * factorial(n);
}


```

Recursive vs Iterative

Factorial: $n! = n \cdot (n-1) \cdot (n-2) \cdot \ldots \cdot 1$

```

int factorial(n){
  int out = 1;
  for (int i=1; i<= n; i++){
    out *= i;
  }
  return out;
}



```

What is Divide and Conquer?

- Divide a given into multiple subparts

- Solve each of the subparts
- Combine the solutions for each subpart to solve the problem.

Similarities between Recursion and Divide and Conquer

10.7 Other Divide and Conquer Problems

Other Divide and Conquer problems

Problems we'll look at more next week:

- Counting Inversions
- Closest Points
- Integer Multiplication
- Convolutions/FFT

10.8 Summary

Summary

What problems did we work on today?

- Sorting

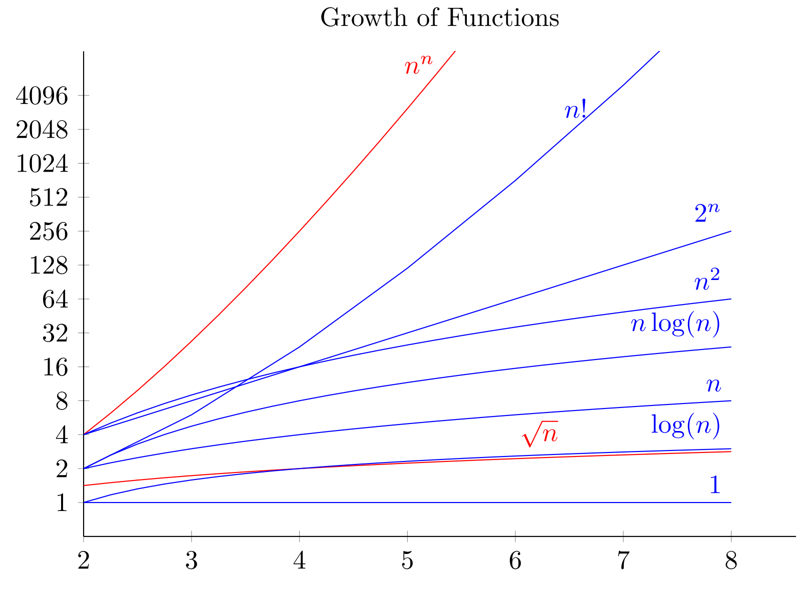
What approaches did we use?

- 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

Readings for next week:
Cormen: Algorithms Unlocked, pp 40-59 (mergesort, quicksort)
Rosen: Chapter 5: Induction and Recursion


Growth of Functions
n^n
$n!$
2^n
n^2
$n\log(n)$
n
$\log(n)$
$\sqrt{n}$
1
4096
2048
1024
512
256
128
64
32
16
8
4
2
1
2
3
4
5
6
7
8

10.9 Appendix: Counting inversions

Inversions: Step 1– What's the problem?

We want to compare two sets of rankings.

This matters when it comes to things like ***collaborative filtering***, which is used for recommendations.

Consider the problem:

These are my favorite movies:

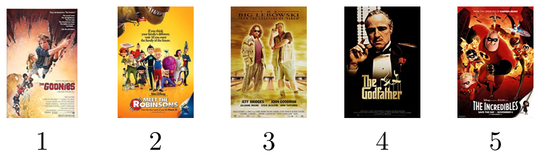
How does Netflix recommend me a new movie I might like?

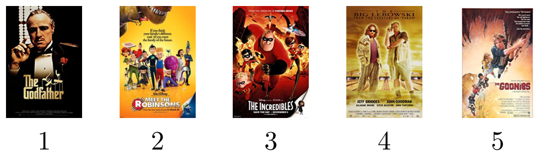
Very briefly, one approach is:

1. Find people who feel similarly as I do about the same movies
2. Recommend me movies those people also like that I haven't seen

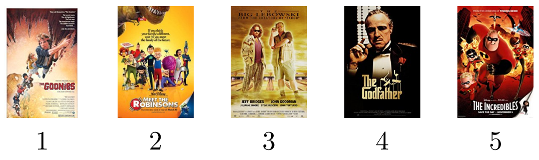
How do they find people similar to me?

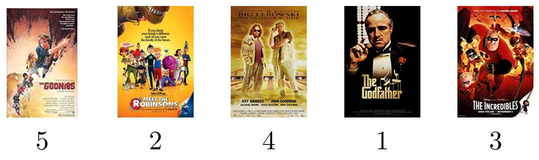
How to find people similar to me?





How similar are these rankings?





First, order the movies according to my ranking.

Then, look at someone else's rankings of these movies.

How many are out of order?

A more formal distillation of the problem:

We have a sequence of n numbers $\{a_1, a_2, a_3, \ldots, a_n\}$.

(Assume all numbers are distinct.)

Can we come up with a measure of how out-of-order this list is?

This measure should be 0 if the sequence is perfectly ascending, and go higher as the sequence is more scrambled.

One suggestion: Count ***inversions***.

An inversion definition: two indices $i < j$ form an inversion if $a_i > a_j$.

Counting Inversions: The Brute-Force Approach

- Look at all pairs of numbers; count which ones are inverted.
- $O(n^2)$

Can we do better?

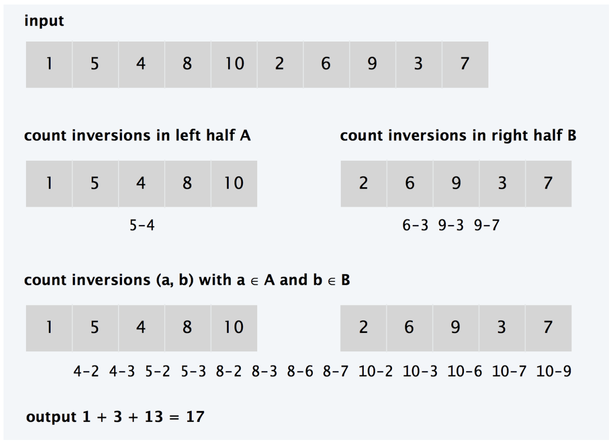
Improving on Brute Force

How to improve?

- Well, let's try divide and conquer.
- The number of inversions in a list is the number of inversions in one half of the list plus the number in the other half of the list, plus the number of inversions between those on the first half and those on the second half.
- To get the number of inversions between the two halves is an easier problem to solve if the two halves are sorted.
- Well, now it's starting to sound like MergeSort can be modified to solve this problem.
- Instead of Merging, can we count inversions between the left and right half?

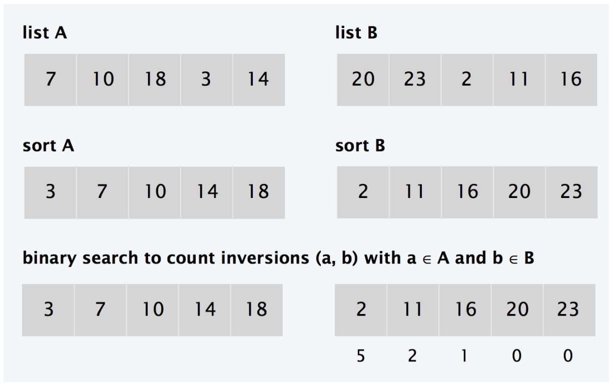
Applying Divide and Conquer

- **Divide:** separate list into two halves A and B.
- **Conquer:** recursively count inversions in each list.
- **Combine:** count inversions (a, b) with $a \in A$ and $b \in B$.
- Return sum of three counts.



Slight aside:

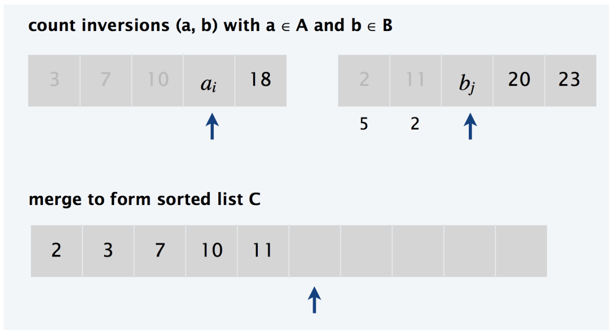
1. Sort A and B.
2. For each element b in B, find how many elements in A are greater than b.



Combining the solved subproblems

Count inversions (a, b) with $a \in A$ and $b \in B$, assuming A and B are sorted.

- Scan A and B from left to right.
- Compare a_i and b_j.
- If $a_i < b_j$, then a_i is not inverted with any element left in B.
- If $a_i > b_j$, then b_j is inverted with every element left in A.
- Append smaller element to sorted list C.


count inversions (a, b) with a ∈ A and b ∈ B
3 7 10 a_i 18
2 11 b_j 20 23
5 2
merge to form sorted list C
2 3 7 10 11

Counting Inversions: The Algorithm

```
MERGE-AND-COUNT(A, low, mid, high)
int numInversions = 0;
L = A[low:mid]
R = A[mid+1:high]
lInd, rInd = 0;
while low < high:
    if L[lInd] < R[rInd]:
        A[low] = L[lInd++]
    else
        A[low] = R[rInd++]
        numInversions += (mid - lInd);
    low++
```

Getting the num inversions, using Merge-And-Count

```
SORT-AND-COUNT(L)
if list L has one element:
    // there are no inversions
    return (0, L)
else
    // Divide the list into two halves:
    list A gets first $\lceil n/2 \rceil$ elements
    list B gets the remaining $\lfloor n/2 \rfloor$ elements
    (rA, A) = SORT-AND-COUNT(A)
    (rB, B) = SORT-AND-COUNT(B)
    (r,L) = MERGE-AND-COUNT(A,B)
return (rA +rB +r, L)
```

Counting Inversions: Runtime

$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1 \\ T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + \Theta(n) & \text{if } n > 1 \end{cases}$$

Summary of Counting Inversions

-

10.10 Appendix: Solving Recurrences with Substitution and Master Method

10.10.1 Substitution

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:

Step 1: Make a guess:

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 $m = \lfloor n/2 \rfloor$.

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

Multiply both sides by 2 and add n

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

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

$$\lg 2 = \log_2 2 \to \lg 2 = 1 \tag{10.40}$$

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

10.10.2 Master Theorem

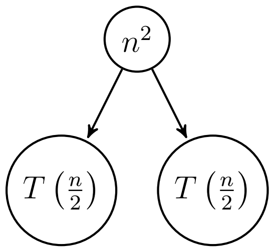
Recurrence

Definition 10.6 *Definition: Recurrence A* ***recurrence*** *is an equation or inequality that describes a function in terms of its value on smaller inputs.*

Example (from merge sort):

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

Recursion Tree



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

- Break the input into 2 chunks
- Solve the problem on each chunk
- Spend n^2 time dividing and re-combining the input/results.

A recurrence can be graphically represented by a recursion tree. Here, the recurrence shows that the input is broken into to chunks, solved, and then recombined in n^2 time.

10.11 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

10.11.1 Defining the Master Theorem

Master Theorem: The Whole Thing.

If a recurrence satisfies the form:

$$T(n) = aT(n/b) + f(n)$$
$$a \text{ is a constant such that } a \geq 1$$
$$b \text{ is a constant such that } b > 1$$
$$f(n) \text{ is a function}$$

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

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

10.11.2 Restating the Master Theorem casually

Master Theorem: The recurrence structure

$$T(n) = aT(n/b) + f(n)$$
$$a \text{ is a constant such that } a \geq 1$$
$$b \text{ is a constant such that } b > 1$$
$$f(n) \text{ is a function}$$

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 be 2^n)

Reminder: Asymptotic Notation

Big-O: asymptotic upper bound

$f(x) = O(g(x))$ means that $f(x)$ is lower than/less than $g(x)$

Big-Θ: asymptotically tight bound

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

Big-Ω: asymptotic lower bound

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

Big-O: asymptotic upper bound

$f(x) = O(g(x))$ means that $f(x)$ is lower than/less than $g(x)$

Big-Θ: asymptotically tight bound

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

Big-Ω: asymptotic lower bound

$f(x) = \Omega(g(x))$ means that $f(x)$ is bigger than $g(x)$.

Master Theorem: Focusing on $f(n)$

$$T(n) = aT(n/b) + \bigcirc\!\!\!\!\!\!\!\!\;\; f(n)$$

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.

10.11.3 Examples

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 $n^{\log_b a} \rightarrow n^{\log_2 2} \rightarrow n^1 \rightarrow n$?
4. $\Theta(n) = \Theta(n)$, so Case 2 applies.
5. $T(n) = \Theta(n^{\log_b a} \lg n) \rightarrow \Theta(n^{\log_2 2} \lg n) \rightarrow \Theta(n \lg n)$ ✓

This is an example of Case 2

Example 2

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

1. $a = 2^n; b = 2$
2. Trick question: a is not a constant! $\Rightarrow$ Can't use the Master Theorem.

Example 3

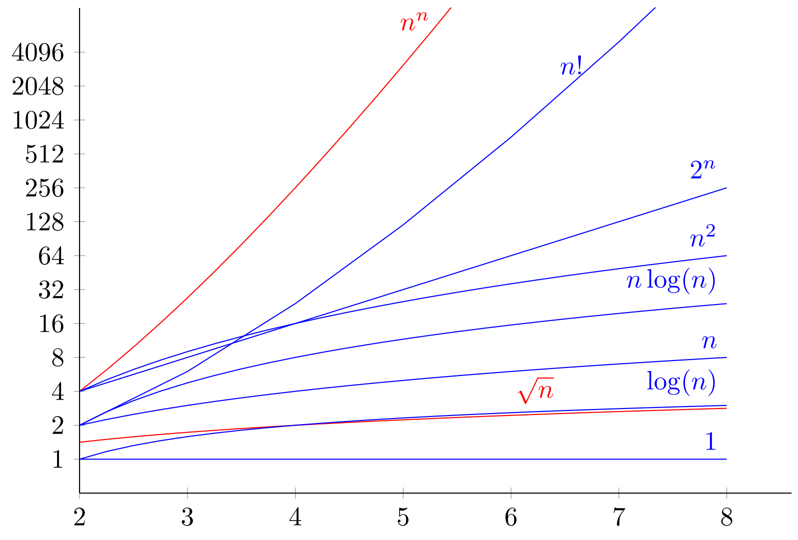
$$T(n) = \sqrt{2}T(n/2) + \log n$$

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. $\Rightarrow T(n) = \Theta(n^{\log_b a}) \rightarrow \Theta(\sqrt{n})$ ✓

This is an example of Case 1

Growth of Functions



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. $\Rightarrow T(n) = \Theta(n^{\log_b a}) \rightarrow \Theta(n^2)$✓

This is an example of Case 1

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. $\Rightarrow T(n) = \Theta(f(n)) = \Theta(n\log n)$✓

This is an example of Case 3

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 $n^{\log_4 2} \to n^{1/2} \to n^{0.5}$?
4. $f(n) = \Omega(\sqrt{n}) \Rightarrow$ Case 3 applies
5. $\Rightarrow T(n) = \Theta(n^{0.51})$✓

This is an example of Case 3

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 $n^{\log_2 4} \to n^2$?
4. $f(n) = \Theta(n^2) \Rightarrow$ Case 2 applies
5. $T(n) = \Theta(n^{\log_b a} \lg n) \to \Theta(n^2 \lg n)$

This is an example of Case 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[4] $\Rightarrow$ Master theorem does not apply.

This is an example of Case 3, where the regularity condition is violated.

10.11.4 Applying to an algorithm

Binary Search

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

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

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

- Introduce the whole Master Theorem
- Break down the Master Theorem and specify definitions
- Restate the Master Theorem to develop intuition

[4] 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.

- Use the Master Theorem to solve some recurrences
- Use the Master Theorem to analyze an algorithm

10.11.5 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

Summary

- Defined the Master Theorem
- Deconstructed the Master Theorem
- Solved some problems
- Describing the run time of an algorithm using Master Theorem
- Generated intuition of Master Theorem