How to Measure Program Efficiency? Surveying and Timing

- **Survey**
  - Hire several test subjects to use your program
  - Survey: How fast is it? Rate from 1 to 10
  - Collect and analyze survey results
  - Obviously not very reliable
- **Write program and time it**
  - Record running times
  - Do this several experiments and get average

Timing Is Not Enough

- Need to write code first before knowing if it will work
  - Maybe waste of time
- No insight on why program is slow, how to fix
  - Bad way: repeatedly make some random tweaks and time the program
- No insight how to design fast programs in the first place
- What we need: a *scientific* way of measuring efficiency

Why Not Just Buy a Faster Computer?

- Well… How to make faster computers in the first place?
- Just buy faster parts!
- Well… How to make faster parts in the first place?
- Just buy faster materials!
- Well… How to make faster materials in the first place?
- Just buy… a faster universe!

How the Science of Computing Works

- Efficiency is not *absolute* but *relative*
- Assume at some level the operations have a fixed, unchangeable cost
  - **Model of computation**: what operations we assume are available, and at what fixed costs
- Focus on making the higher-level operations require as few of these lower-level operations as possible to minimize the cost
- Efficiency of higher-level operations depends on **algorithm**
  - How the lower-level operations are structured to achieve the goal

How to Measure Program Efficiency? Counting Operations

- Choose a set of operations we consider fixed
  - Example: primitive operations available in programming language
- Measure the average cost for each of those operations
- Count number of times we perform each type of operation
- Total cost = sum of number of times $\times$ cost per type (over all types)

How to Measure Program Efficiency? Counting Operations

Operation	Cost (in microseconds)
Add two integers	0.5
Multiply two integers	2.5
Change value of a variable	1.0
Print an integer	3.0

How to Measure Program Efficiency? Counting Operations

- Even better:
  - Don’t measure in seconds/bytes/etc.
    - Actual number depends on factors (hardware) we have no control over
    - Temporarily treat real number as an irrelevant detail
  - Choose set of operations such that each has roughly the same average cost
- Total cost = total number of operations
- Later, to get real total cost, just multiply by real average cost
- But because we only need an *estimate* at first, we can ignore this

How to Measure Program Efficiency? Counting Operations

Operation	Cost (in number of operations)
Add two integers	1
Multiply two integers	1
Change value of a variable	1
Print an integer	1

- Average time per operation = 2 microseconds

Let's Try It

```
int S = 0;
for(int i = 1; i <= n; i++)
    S += i;
```

- **Model of computation**: the following operations have cost 1
  - Declaring a variable
  - Assigning a number to variable
  - Reading the value of a variable
  - Performing one addition
  - Performing one comparison
  - Doing a condition check

How Many Times Did We Do Each Operation?

```
int S = 0;
for(int i = 1; i <= n; i++)
    S += i;
```

- Declaration: 2
- Assignment:
  - 2 for initialization
  - 2 per iteration ⇒ $2n$
- Variable read:
  - 4 per iteration ⇒ $4n$
  - 2 for last loop condition check
- Addition: 2 per iteration ⇒ $2n$
- Comparison
  - 1 per iteration ⇒ n
  - 1 for last loop condition check
- Condition check: same as above
- Total: $10n + 8$

Notice

- Number of operations can grow with input size: write cost as $T(n)$
  - T for "time" (but we are really counting number of operations)
- Still requires writing code before knowing if it will work
- Counting operations is too difficult for larger programs
  - Even for the small program above, you probably have different answers
- Minor implementation details can change answer
  - Minor: doesn't change the "essence" of the algorithm

“Minor Implementation Details” Example: Significantly Different

```
int S = 0;
for(int i = 1; i <= n; i++)
    S += i;
```

```
int S = n * (n + 1) / 2;
```

“Minor Implementation Details” Example: Not Significantly Different

```
int S = 0;
for(int i = 1; i <= n; i++)
    S += i;
```

```
int S = 0;
for(int i = 0; i < n; i++)
    S += i + 1;
```

Big Idea in Computer Science: Analyze Large-Scale Behavior of Cost Function

- Efficiency only matters when the problem is large
- As problem size grows, lower-order terms of cost function grow insignificant compared to highest-order term
- Generally also true: the constant factor (coefficient) in front of the highest-order term is irrelevant to the "essence" of the algorithm
  - In practice, this can matter, and it matters more than lower order terms, but matters less than the "degree" of the highest-order term

Interactive Demonstration

- https://www.desmos.com/calculator/k4p7sewkq2
- When zoomed in, the four graphs all look very different
- When zoomed out, $T(n)$ looks basically the same as $f(n)$, but still quite different from $g(n)$ and $h(n)$

How to Measure Program Efficiency? Large-Scale Analysis

- It is enough to look at the large-scale, big picture behavior of an algorithm to determine if it is efficient
- We say an algorithm or program runs in $\Theta(f(n))$ if the number of operations is *kinda like* $f(n)$
  - More precise phrase: *on the order of* $f(n)$
  - Even more precisely: it's $f(n)$ if you ignore all the lower-order terms and the constant factor (coefficient) in front of the leading term

How to Measure Program Efficiency? Large-Scale Analysis

- By the way, we can use this to measure not just number of operations/speed but any resource we are interested in conserving in general
  - "The algorithm uses $\Theta(f(n))$ additions"
    - Example: in a problem asking you to limit the number of additions performed to x
  - $\Theta(f(n))$ multiplications
  - $\Theta(f(n))$ space
  - $\Theta(f(n))$ messages sent over a network

How to Measure Program Efficiency? Large-Scale Analysis

- Specifically for measuring speed, we also say, “the **running time** of an algorithm is $\Theta(f(n))$”
  - Remember, we really are counting the number of operations
  - But since the number of operations is directly proportional to time, informally, we just use these interchangeably
  - Since we ignored constant factors, using the word “time” is technically correct again: time is operation count $\times$ constant

How to Measure Program Efficiency?
Large-Scale Analysis

- Measure efficiency, not by *fine-grain* counting all the individual operations, but by *course-grain* **large-scale analysis** of the algorithm
- Figure out the Θ rather than the exact function
- Not always easy, but easier than counting individual operations

Example

```
int S = n * (n + 1) / 2;
```

- Actual number of operations

$$T(n) = 6$$

- Or simply: $\Theta(1)$

```
int S = 0;
for(int i = 1; i <= n; i++)
    S += i;
```

- Actual number of operations

$$T(n) = 10n + 8$$

- Or simply: $\Theta(n)$

Example

```
int S = n * (n + 1) / 2;
```

- Without counting actual number of operations first: by "rough feel," since we are doing only a fixed number of basic operations, it is $\Theta(1)$

```
int S = 0;
for(int i = 1; i <= n; i++)
    S += i;
```

- Without counting actual number of operations first: by "rough feel," due the loop, we are doing $\Theta(n)$ operations

More Examples: Two Loops Done in Succession

```python
minnie = a[0]
for i in range(n):
    minnie = min(minnie, a[i])

maxxie = a[0]
for i in range(n):
    maxxie = max(maxxie, a[i])

print(maxxie - minnie)
```

More Examples: Two Loops Done in Succession

- Each loop is $\Theta(n)$
- So the two loops together is $\Theta(2n)$
- Or simply $\Theta(n)$, since we treat the constant factor as an unimportant detail

More Examples: Two Loops, One Nested

```
ans = 0
for a in range(1, n + 1):
    for b in range(1, n + 1):
        if a % 2 == b % 2:
            ans += a * b
```

More Examples: Two Loops, One Nested

- Outer loop runs the inner loop $\Theta(n)$ times
- The inner loop is $\Theta(n)$
- Therefore, the total time is $\Theta(n \times n) = \Theta(n^2)$
- One done **a**fter another → **a**dd
- One done **w**ithin another → **m**ultiply

M
"Here's your problem spongebob you have it set to 'M' for Mini...
W
When it should be set to 'W' or Wumbo!"
VIA 9GAG

aspc ateneo summer programming camp

More Examples: Three Levels of Nesting

```
ans = 0
for i in range(n):
    for j in range(n):
        for k in range(n):
            ans += a[i] * a[j] * a[k]
```

More Examples: Three Levels of Nesting

- i loop runs the j loop $\Theta(n)$ times
- j loop runs the k loop $\Theta(n)$ times
- k loop is $\Theta(n)$
- Therefore, j loop is $\Theta(n \times n) = \Theta(n^2)$
- And i loop is $\Theta(n \times n^2) = \Theta(n^3)$

Can You Guess the Running Time?

```
ans = 0
for i in range(n):
    for j in range(n):
        for k in range(n):
            for x in range(n):
                for y in range(n):
                    for z in range(n):
                        ans += 1
```

Can You Guess the Running Time?

- 6 nested $\Theta(n)$ loops $\Rightarrow \Theta(n^6)$
- With r loops nested like this, what is the running time?
  - $\Theta(n^r)$

More Examples: Two Variables

```
ans = 0
for i in range(n):
    for j in range(m):
        ans += a[i] * b[j]
```

More Examples: Two Variables

- Outer loop runs the inner loop $\Theta(n)$ times
- The inner loop is $\Theta(m)$
- Therefore, the total time is $\Theta(nm)$
- Running time may depend on more than one size variable, if the problem has many different measures of size

More Examples: Magic Formula

```
n = int(input())
print(n * (n + 1) * (2 * n + 1) // 6)
```

More Examples: Magic Formula

- The running time does not depend on the input size
- $\Theta(1)$
- We're assuming n is small enough
  - Python and C++ can do arithmetic operations on numbers o up to 18 digits instantly
  - Beyond that, we need to start caring, because even a computer should get slower with bigger numbers
  - For the most part, you can just not care for now, but keep it at the back of your mind

More Examples: Something More Magical

```
n = int(input())
ans = 0
ans += n
ans -= n // 2
ans -= n // 3
ans -= n // 5
ans -= n // 7
ans += n // 6
ans += n // 10
ans += n // 14
ans += n // 15
ans += n // 21
ans += n // 35
ans -= n // 30
ans -= n // 42
ans -= n // 70
ans -= n // 105
ans += n // 210
print(ans)
```

More Examples: Something More Magical

- Looks like a lot of code, but remember: *large-scale analysis*
- In the grand scheme of things, for example, when n is roughly 10^{18}, the number of operations here is insignificant
- It's still a small, fixed constant which does not depend on n
- $\Theta(1)$

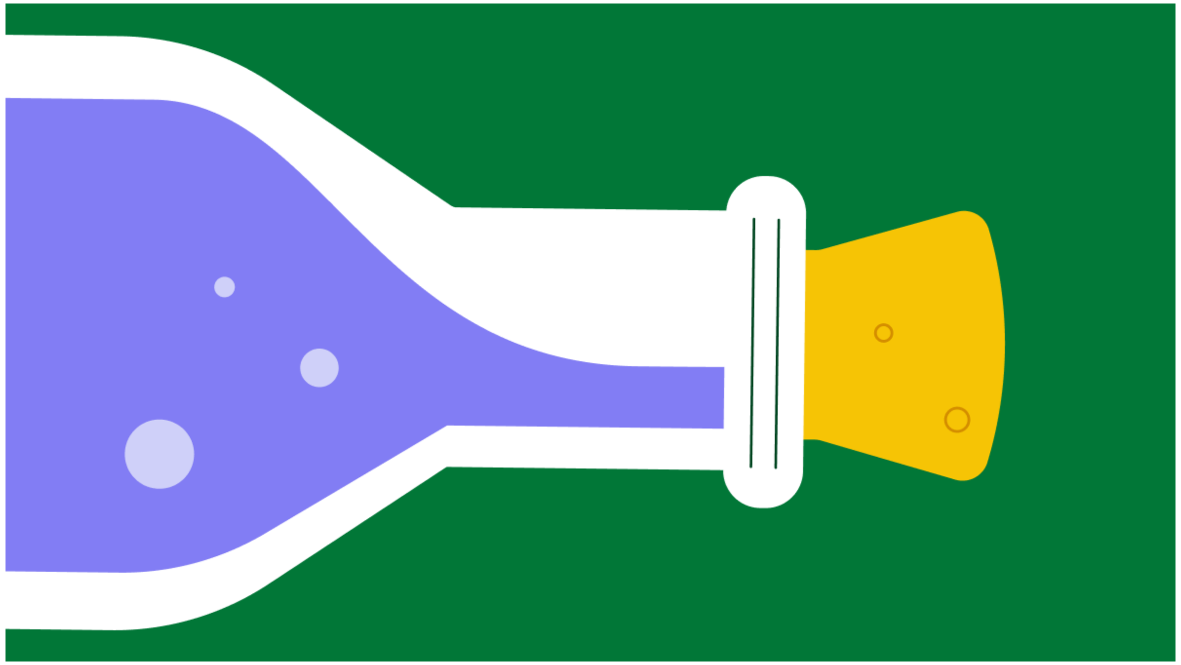
More Examples: Only the Bottleneck Matters

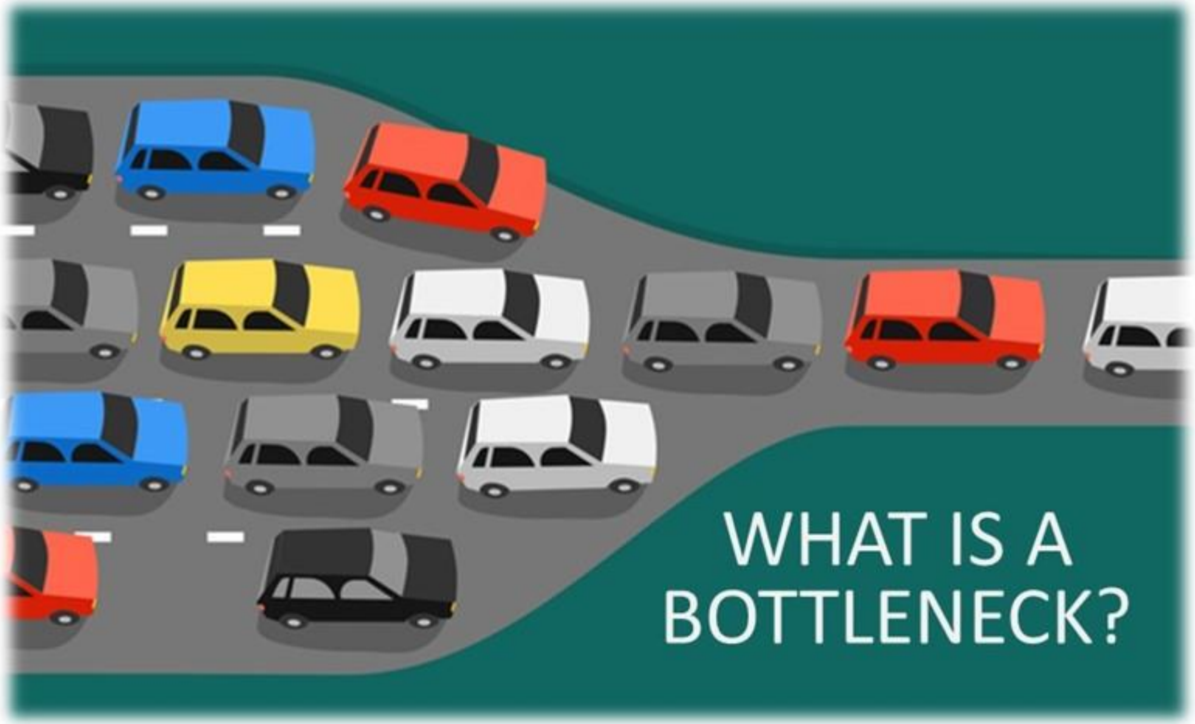
```
ans = 0
for i in range(n):
    for j in range(n):
        for k in range(n):
            ans += max(a[i], a[j], a[k])

for i in range(n):
    for j in range(n):
        ans -= min(a[i], a[j])
```

More Examples: Only the Bottleneck Matters

- 3 nested loops $\Rightarrow \Theta(n^3)$
- 2 nested loops $\Rightarrow \Theta(n^2)$
- Run one after the other $\Rightarrow \Theta(n^3 + n^2)$
- Or simply $\Theta(n^3)$, since we treat lower-order terms as an unimportant detail
- In other words, only the **bottleneck** (least efficient) part of the code matters
  - No matter how you good you optimize the $\Theta(n^2)$, the $\Theta(n^3)$ will *dominate* the running time

Bottleneck





More Examples: Multiple Nested Blocks

```
for i in range(n):
    less = 0
    for j in range(n):
        if a[j] < a[i]:
            less += 1

    greater = 0
    for j in range(n):
        if a[j] > a[i]:
            greater += 1
```

More Examples: Multiple Nested Blocks

- The thing inside the i loop runs $\Theta(n)$ times
- The thing inside the i loop has two parts, the first part being $\Theta(n)$ and the second part being $\Theta(n)$
- The thing inside the i loop is $\Theta(n+n) = \Theta(n)$
- The whole thing is $\Theta(n \times n) = \Theta(n)$

More Examples: Variable Amount of Nested Work

```
ans = 0
for i in range(n + 1):
    for j in range(i):
        ans += a[i] * a[j]
```

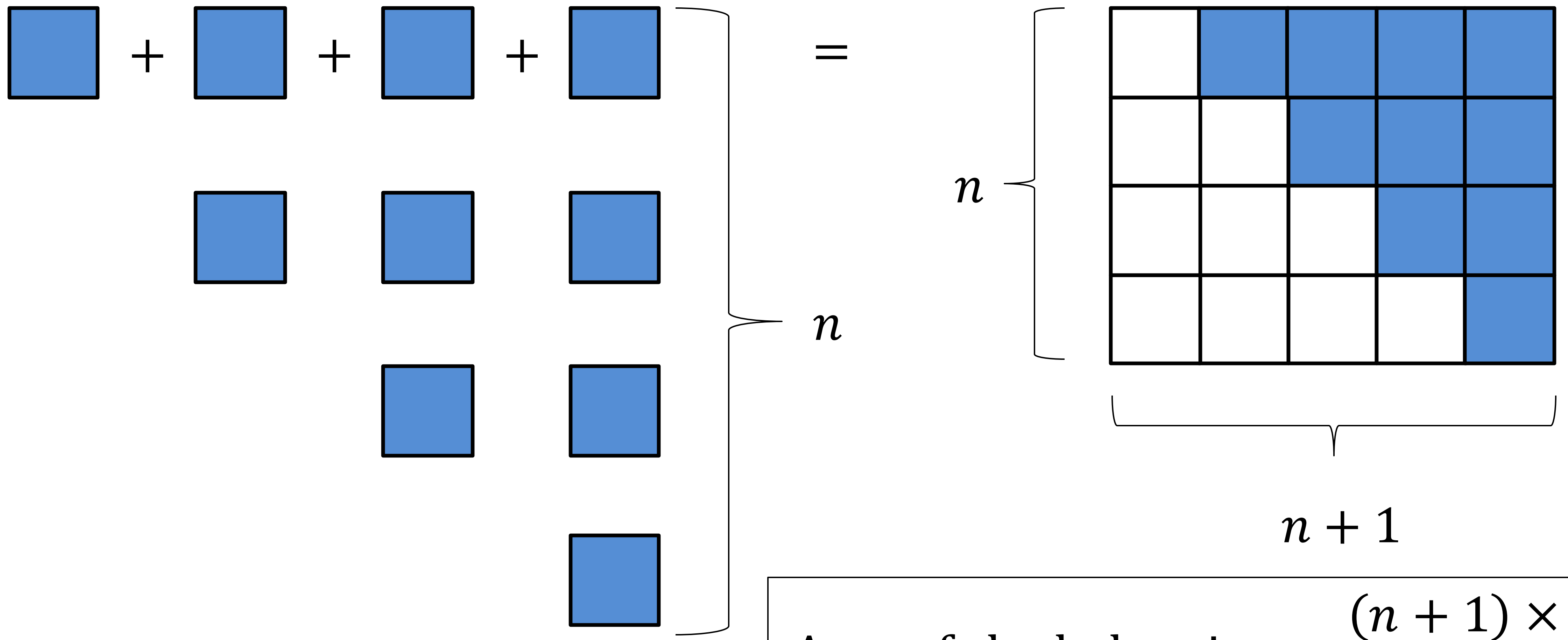
More Examples: Variable Amount of Nested Work

- i loop goes from 1 to n, but it doesn't do the same amount of work at each step, so simple multiplication doesn't work here
  - When $i = 1, j$ loop goes up to 1
  - When $i = 2, j$ loop goes up to 2
  - When $i = 3, j$ loop goes up to 3
  - …
  - When $i = n, j$ loop goes up to n

More Examples: Variable Amount of Nested Work

- i loop goes from 1 to n, but it doesn't do the same amount of work at each step, so simple multiplication doesn't work here
- The total amount of work for all the times the j loop is run is $1 + 2 + 3 + \cdots + (n - 1) + n$
- This is $\Theta(n^2)$

Why $1 + 2 + 3 + \cdots + (n-1) + n = \Theta(n^2)$



$$\boxed{\text{Area of shaded region} = \frac{(n+1) \times n}{2}}$$

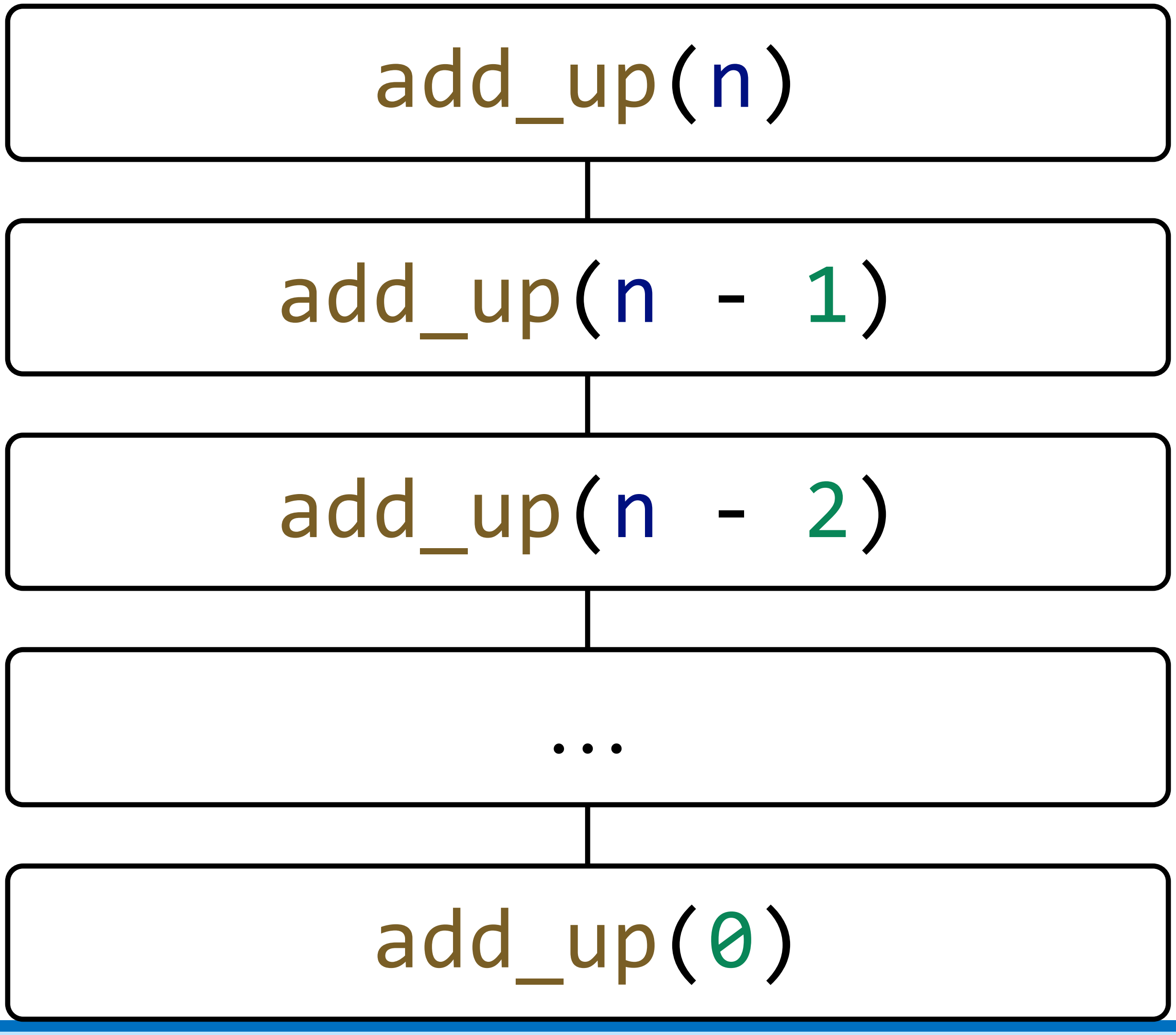
More Examples: Recursive Programs

```
def add_up(n):
    if n == 0:
        return 0
    else:
        return 1 + add_up(n - 1)
```

More Examples: Recursive Programs

- With recursive functions, one way to figure out the running time is by looking at the **recursion tree**, figuring out how much non-recursive work each *node* in the tree is doing, and adding them all up

Recursion Tree for add_up



Analysis for `add_up`

- How many nodes in this tree? $\Rightarrow \Theta(n)$
- How much non-recursive work is each done at each node?
- Non-recursive work typically consists of the following:
  - Work to transform original problem to smaller subproblem(s) $\Rightarrow$ `1 + add_up(n - 1)`
  - Work to transform answer from smaller subproblem(s) into answer to original problem $\Rightarrow$ `1 + add_up(n - 1)`
  - Work done at base case $\Rightarrow$ `return 0`
  - All $\Theta(1)$

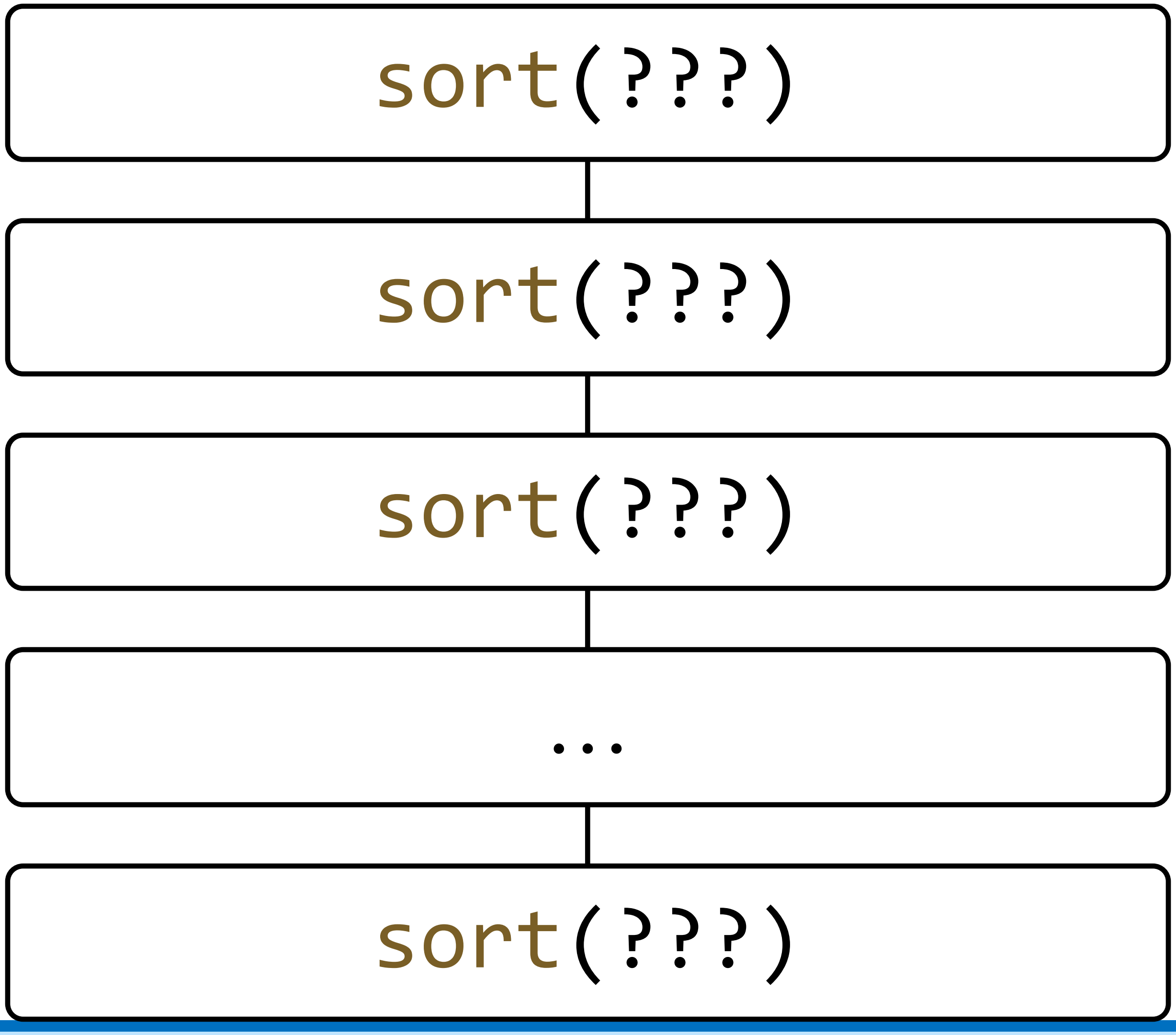
Analysis for add_up

- $\Theta(n)$ nodes $\times$ $\Theta(1)$ per node = $\Theta(n)$

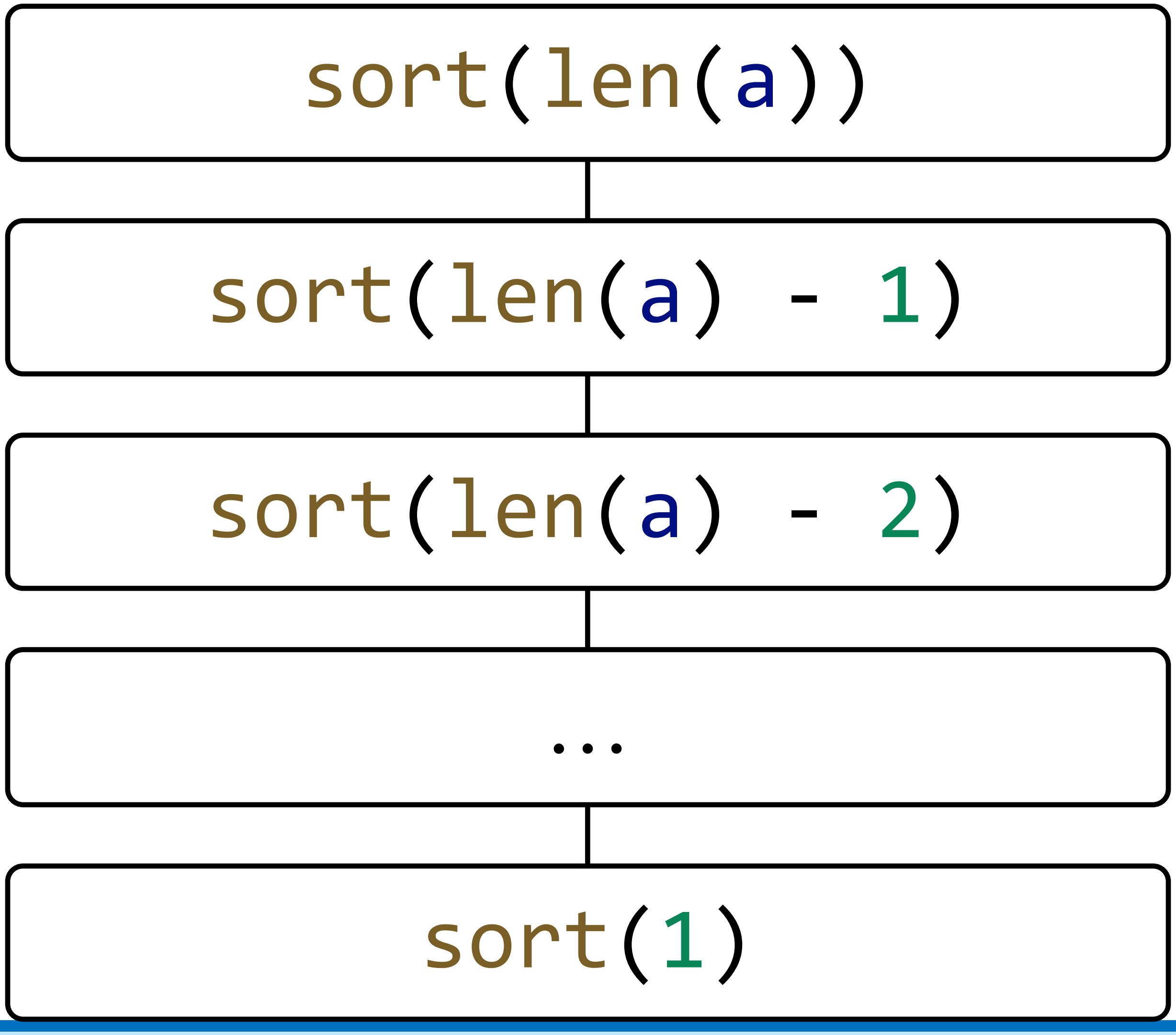
More Examples: Recursive sort

```
def sort(a):
    if len(a) <= 1:
        return a
    else:
        i = index_of_min(a)
        return [a[i]] + sort(a[:i] + a[i+1:])
```

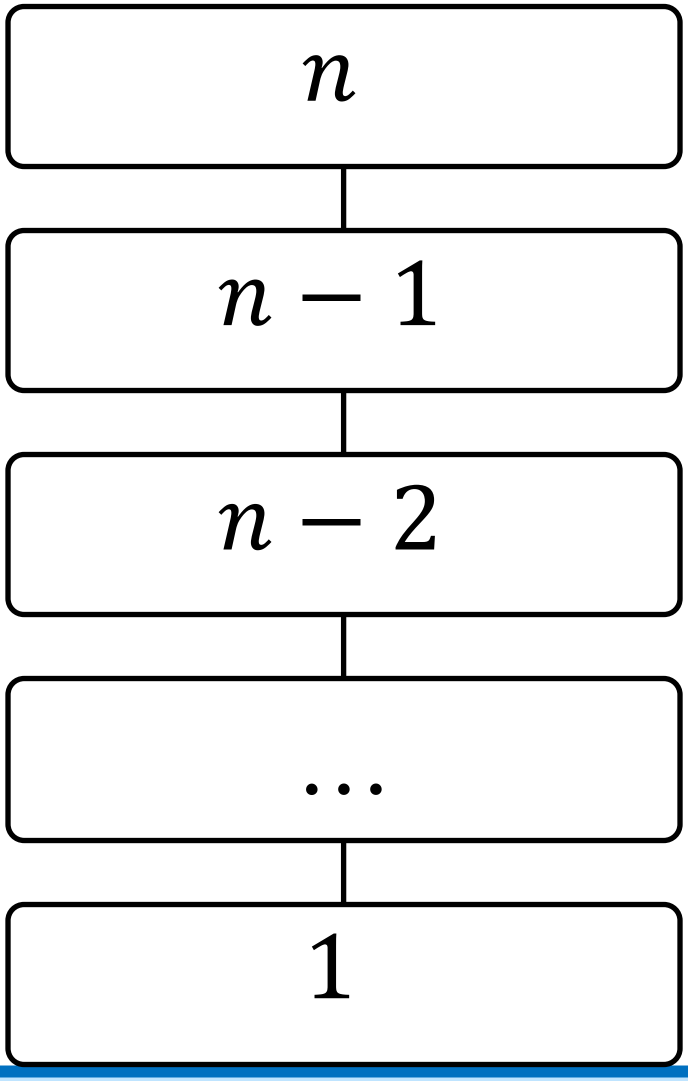
Recursion Tree for `sort`



Instead of Drawing the Tree with Actual Arguments, Draw the Tree with the Problem Sizes



Letting $n =$ `len(a)`, We Can Remove Some Irrelevant Details from the Drawing



Analysis for `sort`

- How many nodes?
  - $\Theta(n)$
- This time, amount of work done per node varies depending on the size of the subproblem, let’s say a subproblem has size s
  - So we can’t just do a simple multiplication

Analysis for `sort`

- What are all the parts doing non-recursive work? How much work for each part?
  - `index_of_min` is $\Theta(s)$
    - You can imagine how this might be done with a loop
  - `a[:i] + a[i+1:]` is $\Theta(s)$
    - Creating a length l slice of a list is $\Theta(l)$
    - Adding two lists, where the first has length x and the second has length y is $\Theta(y)$
  - `[a[i]] +` is $\Theta(s)$
    - For the same reason as above

Analysis for `sort`

- What are all the parts doing non-recursive work? How much work for each part?
  - Recursive case: $\Theta(s)$
  - Base case: $\Theta(1)$, but $s = 1$ in this case so we can also say $\Theta(s)$

Analysis for `sort`

- Putting everything together, we have n nodes…
  - Work done at node n is $\Theta(n)$
  - Work done at node $n-1$ is $\Theta(n-1)$
  - Work done at node $n-2$ is $\Theta(n-2)$
  - …
  - Work done at node 1 is $\Theta(1)$
- $1+2+3+\cdots+(n-1)+n=\Theta(n^2)$

Note

- The built-in sort functions in C++ and Python work in $\Theta(n \log n)$
  - How? Future lesson
- Do not use our own sort function!
  - It is only a way to start learning about recursion
  - Similarly, do not use our own recursive min function
    - By same analysis as our own sort function, it is $\Theta(n^2)$
    - Whereas the built-in one is $\Theta(n)$

More Examples: Recursive `num_digits`

```
def num_digits(n):
    if n < 10:
        return 1
    else:
        return 1 + num_digits(n // 10)
```

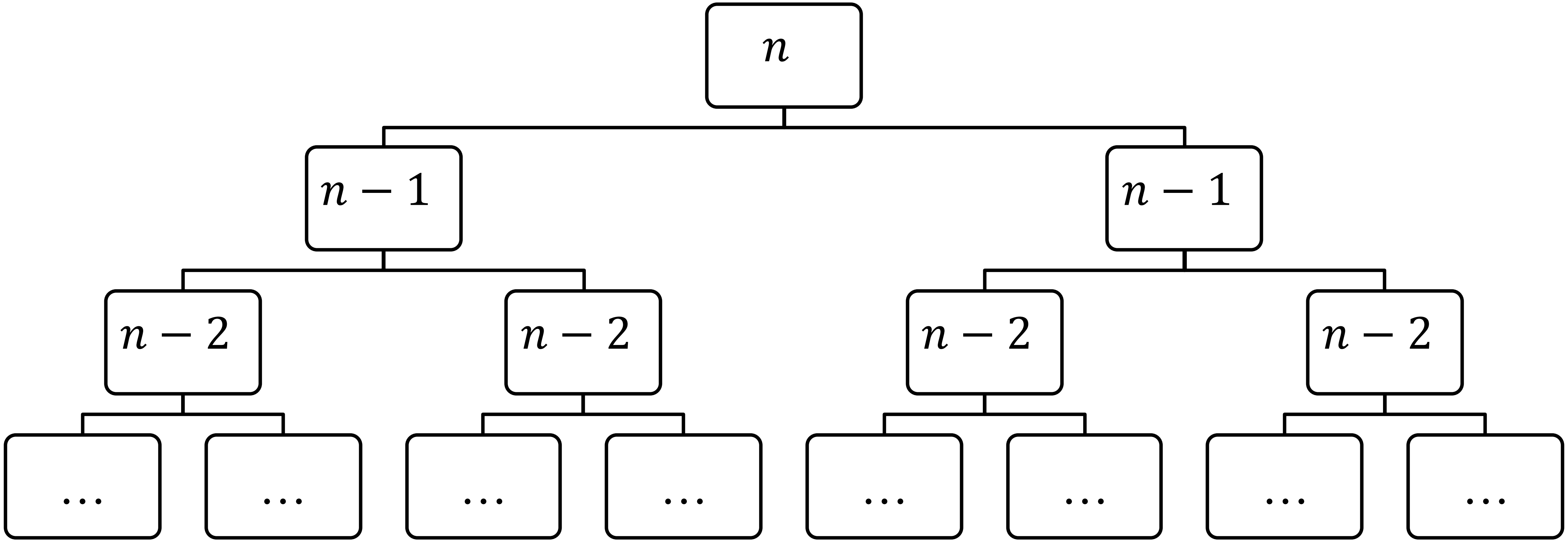
Analysis for `num_digits`

- Pretty easy to see $\Theta(1)$ non-recursive work per node
- But how many nodes?
  - How many steps must you divide n by 10 until it becomes a single-digit number (base case)?
  - AKA the number of digits of n
  - AKA $\approx \log_{10} n$
  - $\Theta(\log n)$

More Examples: A Non-Boring Recursion Tree

```python
def move_disks(n, start_rod, temp_rod, goal_rod):
    if n > 0:
        move_disks(n - 1, start_rod, goal_rod, temp_rod)
        print(f'Move from {start_rod} to {goal_rod}')
        move_disks(n - 1, temp_rod, start_rod, goal_rod)
```

Recursion Tree for `move_disks`



Analysis for `move_disks`

- It's clear that $\Theta(1)$ per node, hard part is number of nodes
- When the recursion tree branches, it's helpful to think about the *branching factor* and the *number of levels*
  - Branching factor: how many subproblems does each node have?
  - Number of levels: how far is the top node from the base case?
- Branching factor for `move_disks`: 2
- Number of levels for `move_disks`: n

Analysis for `move_disks`

- Every level has twice the number of nodes of the previous
- Level 0 has 1 node (size n)
- Level 1 has 2 nodes (size $n - 1$)
- Level 2 has 4 nodes (size $n - 2$)
- Level 3 has 8 nodes (size $n - 3$)
- Level 4 has 16 nodes (size $n - 4$)
- …
- Do you see the pattern?

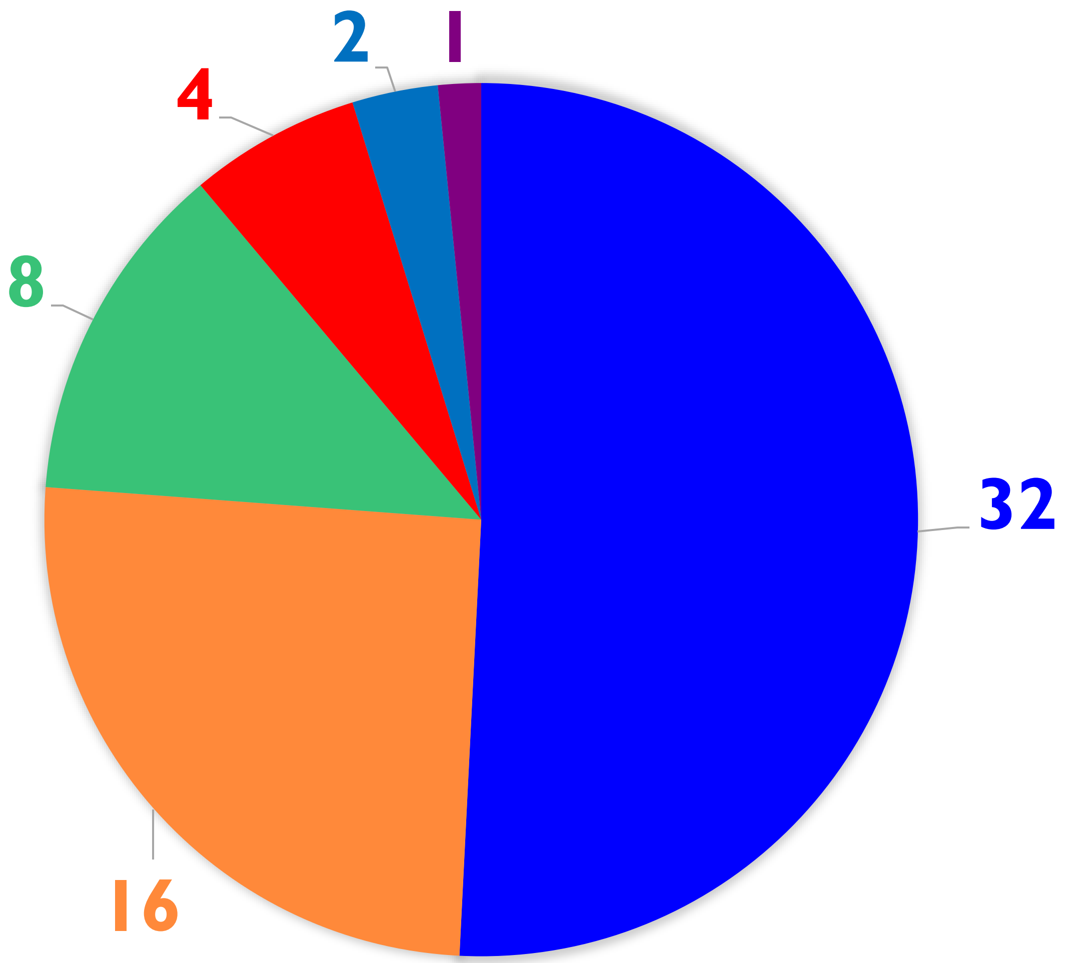
Analysis for `move_disks`

- Every level has twice the number of nodes of the previous
- Level 0 has $1 = 2^0$ node (size n)
- Level 1 has $2 = 2^1$ nodes (size $n - 1$)
- Level 2 has $4 = 2^2$ nodes (size $n - 2$)
- Level 3 has $8 = 2^3$ nodes (size $n - 3$)
- Level 4 has $16 = 2^4$ nodes (size $n - 4$)
- $\ldots$
- Level n has 2^n nodes (size 0)

Analysis for `move_disks`

- So, the total work is $2^0 + 2^1 + 2^2 + \cdots + 2^n$
- Can we simplify this?
  - $1 + 2 = 3$
  - $1 + 2 + 4 = 7$
  - $1 + 2 + 4 + 8 = 15$
  - $1 + 2 + 4 + 8 + 16 = 31$
  - $1 + 2 + 4 + 8 + 16 + 32 = 63$
- Do you see the pattern?
  - $2^0 + 2^1 + 2^2 + \cdots + 2^n = 2^{n+1} - 1 = \Theta(2^{n+1})$

Why $2^0 + 2^1 + 2^2 + \cdots + 2^n = \Theta(2^{n+1})$



- The whole circle is roughly twice the size of the largest piece
- This picture works no matter which power of 2 you choose to end at

Note

- In all our examples, the code was already written before we analyzed
- But you can do this even without the code (with practice)
  - All the details were not necessary to do the analysis
  - Once you have an algorithm in mind, you can imagine how many loops will be nested, or how they will come one after the other
  - Or you can write some fake code that doesn't have all the details yet, and the analysis can work on that fake code

For Convenience, We Sometimes Use the Following English Words

$\Theta(1)$	constant
$\Theta(\log n)$	logarithmic
$\Theta(n)$	linear
$\Theta(n \log n)$	linearithmic (less frequently used)
$\Theta(n^2)$	quadratic
$\Theta(n^3)$	cubic
$\Theta(n^c)$ where c is some constant > 0	polynomial
$\Theta(b^n)$ where b is some constant > 1	exponential

aspc ateneo summer programming camp

Upper Bound Analysis

- Sometimes, it's hard to determine even the highest-order term $f(n)$ such that the cost is $\Theta(f(n))$
- Determine the **upper bound** $O(g(n))$ instead
  - The highest order term could be $g(n)$ *or smaller*
  - Θ is like $=$ while O is like $\leq$
- If we can show that our algorithm is "good enough," then we can go ahead and implement it
  - Its actual running time might be better, but all we care about is that it doesn't exceed a certain limit

Upper Bound Analysis: Example

```
for(int i = 0, j = n - 1; i < n; i++)
    for(; j >= 0 and a[i] + a[j] >= k; j--)
        if(a[i] + a[j] == k)
            cout << i << " " << j << endl;
```

- Easy to see by the "shape" of this program that it is $O(n^2)$
- However, it is actually $\Theta(n)$
- Seeing it is $\Theta(n)$ is harder, but if $O(n^2)$ is acceptable, then we don't need to analyze further

Worst Case Analysis

- Efficiency depends not only on input size, but also what kind of input
- On certain cases, algorithm may perform significantly better/worse than on others
- Since we want algorithm to work for all possible valid cases, when we analyze, we should always assume the **worst case** and try to ensure worst case behavior is good enough

Worst Case Analysis: Example

```
bool found = false;
for(int i = 0; i < n and not found; i++)
    if(a[i] == target)
        found = true;
```

- Loop stops as soon as target is found
- Best case: target is at the beginning of array ⇒ $\Theta(1)$ time
  - Not useful for writing fast or fixing slow algorithms
- Worst case: target is at the end of array ⇒ $\Theta(n)$ time

Average Case Analysis

```
bool found = false;
for(int i = 0; i < n and not found; i++)
    if(a[i] == target)
        found = true;
```

- Target is somewhere in the middle $\Rightarrow \Theta\left(\frac{n}{2}\right) = \Theta(n)$

Average Case Analysis

- Sometimes, worst case analysis is too pessimistic
- However, it's much simpler to do than average case analysis
  - Average case analysis can require thinking about probabilities
- So we will mostly stick to worst case
  - Ok for the same reason that upper bound analysis is ok

Large-Scale Analysis Gives Insight into Why Programs Are Slow and How to Fix

- Why so slow?
  - Should not work according to analysis ⇒ algorithm bad
  - Should work according to analysis ⇒ implementation bad
  - Analyzing each part of the program gives insight into where the slowness is coming from
- How to fix
  - If algorithm bad, need to completely rethink our approach
    - <u>If you don't do this, then no matter how much you micro-optimize, your program will still be too slow!</u>
  - Fix the most inefficient part: this is the *bottleneck*

Applying Large-Scale Analysis in Practice to Estimate Running Time

1. Figure out the $O(f(n))$ running time
2. Take expected input size N and compute $f(N)$
3. Multiply by actual number of seconds per operation
   - Modern computers can do $\sim 10^9$ operations per second
     - Giga = 10^9, hertz = per second
   - In practice, because each primitive operation in modern programming languages translates to several CPU operations, we should take this number to be $\sim 10^8$ (C++) or $\sim 10^7$ (Python) operations per second instead
   - One operation takes 10^{-8} (C++) or 10^{-7} (Python) seconds

Applying Large-Scale Analysis in Practice to Estimate Running Time

- Estimate will be off by some constant factor, but this is ok
  - Using 10^{-8} also provides some leeway on the constant factor
  - Most likely, it is ≤ 10, which sits well with the fact that the real number is 10^{-9} per operation

Applying Large-Scale Analysis in Practice to Check if Algorithm Works in Time Limit

1. Estimate running time
2. Check if it fits within time limit

This can be reversed (easier):

1. Find number of operations doable in given amount of time
   - 1 second = 10^8 (C++) or 10^7 (Python) operations
2. Check if number of operations we are doing fits this limit

Exercise

- $O(n)$ for $n = 10^5$
- $O(1)$ for $n = 10^{10}$
- $O(n^2)$ for $n = 10^5$
- $O(n \log n)$ for $n = 10^5$
- $O(n \log n)$ for $n = 100$
- $O(n^2 \log n)$ for $n = 10^4$
- $O(n^2)$ for $n = 10^4$
- $O(n^2 2^n)$ for $n = 18$

- Will a program with the following running times finish within 3 seconds when run on the following input sizes?

Answers

▪ $O(n)$ for $n = 10^5$	$\Rightarrow$ Yes
▪ $O(1)$ for $n = 10^{10}$	$\Rightarrow$ Yes
▪ $O(n^2)$ for $n = 10^5$	$\Rightarrow$ No
▪ $O(n \log n)$ for $n = 10^5$	$\Rightarrow$ Yes
▪ $O(n \log n)$ for $n = 100$	$\Rightarrow$ Yes
▪ $O(n^2 \log n)$ for $n = 10^4$	$\Rightarrow$ No
▪ $O(n^2 2^n)$ for $n = 18$	$\Rightarrow$ Yes
▪ $O(n^2)$ for $n = 10^4$	$\Rightarrow$ Risky, likely not in Python

Large-Scale Analysis Gives Insight into How to Design Efficient Programs

- Given constraints of the problem, we can *reverse engineer* to find out what target efficiency an acceptable solution should have
- Eliminate solutions that are "clearly too inefficient"
- Target efficiency sometimes gives a hint on what the structure of the solution should be
- Helps us narrow down to the correct solution

Applying Large-Scale Analysis in Practice to Guide Program Design

1. Given the expected input size and time limit, figure out worst Θ (or O) an acceptable solution must have
2. Given the target Θ (or O), come up with an algorithm

Exercise

- $n = 10^5$, 3 seconds
- $n = 100$, 3 seconds
- $n = 1000$, 3 seconds
- $n = 10^4$, 6 seconds
- $n = 10^{18}$, 60 seconds
- $n = 10^9$, 3 seconds
- $n = 10^7$, 3 seconds
- $n = 20$, 3 seconds

- Given the following maximum input sizes and time limit, what target efficiency should your algorithm have?

Possible Answers

$n = 10^5$, 3 seconds	$\Rightarrow O(n \log n)$
$n = 100$, 3 seconds	$\Rightarrow O(n^4)$ risky, $O(n^3)$ safe
$n = 1000$, 3 seconds	$\Rightarrow O(n^2)$
$n = 10^4$, 6 seconds	$\Rightarrow O(n^2)$ risky
$n = 10^{18}$, 60 seconds	$\Rightarrow O(\log n)$
$n = 10^9$, 3 seconds	$\Rightarrow O(\sqrt{n})$
$n = 10^7$, 3 seconds	$\Rightarrow O(n)$
$n = 20$, 3 seconds	$\Rightarrow O(2^n)$

Exercise

- $O(n)$
- $O(n^2)$
- $O(n^3)$
- $O(n^4)$
- $O(2^n)$

- How much bigger of an input can a program with the following running times process in the same time, given a computer twice as fast?
- In other words, find m such that $\frac{O\left(T(m)\right)}{2} = O(T(n))$ in terms of n

Answers

- $O(n)$ $\Rightarrow m = 2n$
- $O(n^2)$ $\Rightarrow m = \sqrt{2}n \approx 1.414$ times
- $O(n^3)$ $\Rightarrow m = \sqrt[3]{2}n \approx 1.2599$ times
- $O(n^4)$ $\Rightarrow m = \sqrt[4]{2}n \approx 1.1892$ times
- $O(2^n)$ $\Rightarrow m = n + 1$

Try Running Your Tower of Hanoi Code for $n = 64$

- Are you willing to wait for it to end?
- $2^{65} \approx 3.68 \times 10^{19}$operations
- $= 3.68 \times 10^{11}$ seconds
  - Assuming 10^8 computer operations per second
- $\approx 11{,}000$ years

From the Towers of Hanoi Wikipedia Page

- Numerous myths regarding the ancient and mystical nature of the puzzle popped up almost immediately. There is a story about an Indian temple in Kashi Vishwanath which contains a large room with three time-worn posts in it, surrounded by 64 golden disks. Brahmin priests, acting out the command of an ancient prophecy, have been moving these disks in accordance with the immutable rules of Brahma since that time. The puzzle is therefore also known as the Tower of Brahma puzzle. According to the legend, when the last move of the puzzle is completed, the world will end.

Try Running Your Tower of Hanoi Code for $n = 64$

- Are you willing to wait for it to end?
- $2^{65} \approx 3.68 \times 10^{19}$operations
- $= 3.68 \times 10^{19}$seconds
  - Assuming 1 Brahmin priest operation per second
- $\approx 1 \times 10^{12}$ years
- For context, the age of the universe is 13.7×10^{9} years
- If the legend were true, we have nothing to worry about!

The Lesson

- Buying a faster computer is not enough
- All the effort put into making computers faster is wasted with bad algorithms
  - Exponential-time algorithms are exceptionally bad
- Algorithm analysis is important!
- Having good algorithms is important!

The Model of Computation We Will Be Using

- All of the following operations take $\Theta(1)$ time
  - Variable declaration/read/write
  - Add/subtract/multiply/divide/compare two "small" numbers
    - E.g. `int`, `double`, `long long` in C++
    - Floating-point numbers or integers with up to ≈ 20 digits in Python
  - Call/return from a function
    - As in assigning parameters, jumping to the function definition and back
    - Function may have loops and do something not $\Theta(1)$, but that's not what we're referring to here
  - Read/write to a single cell of an array or list

The Model of Computation We Will Be Using

- All of the following operations take $\Theta(n)$ time
  - Declare an array of size n
  - Read a string of length n
  - Print a string of length n

The Model of Computation We Will Be Using

- We pick these because they're close to what actually happens when you're programming in a modern language
- From earlier slide: choose set of operations such that each has roughly the same average cost
  - Not exactly true anymore, because some primitive operations we want to model have variable cost
  - It's ok, we just determine which operations are $\Theta(1)$, which ones are $\Theta(n)$, etc., and because we're ignoring constant factors anyway, we can ignore the slight differences within each group

The Model of Computation We Will Be Using

- Note: Python has primitive operations which would be $\Theta(n)$, not $\Theta(1)$, despite the loops being invisible
  - `sum`/`min`/`max` on a list of n items
  - Creating a length n slice of a list

Scarier But Official Words

Not-Scary Words	**Official Words**
Large-Scale Analysis	Asymptotic Analysis
Running Time	Time Complexity

Practice Problems

- https://progvar.fun/problemsets/asymptotic-analysis

Thanks!

accenture

PLDT

stratpoint
FAST FORWARD TO THE FUTURE

MERALCO

Thinking Machines
Data Science