CSJ61B Lecture 15

Monday, March 30, 2020

In today's lecture, we will work through some difficult algorithm analysis problems.

Simple Nested Loops in Big Theta

Find the order of growth of the worst case runtime of `dup1`:

```
int N = A.length;
for (int i = 0; i < N; i += 1)
   for (int j = i + 1; j < N; j += 1)
        if (A[i] == A[j])
            return true;
return false;
```

Our previous techniques were either finding the family of the order of growth of the `==` operations, or using the geometric argument. In Big Theta notation, this would be:

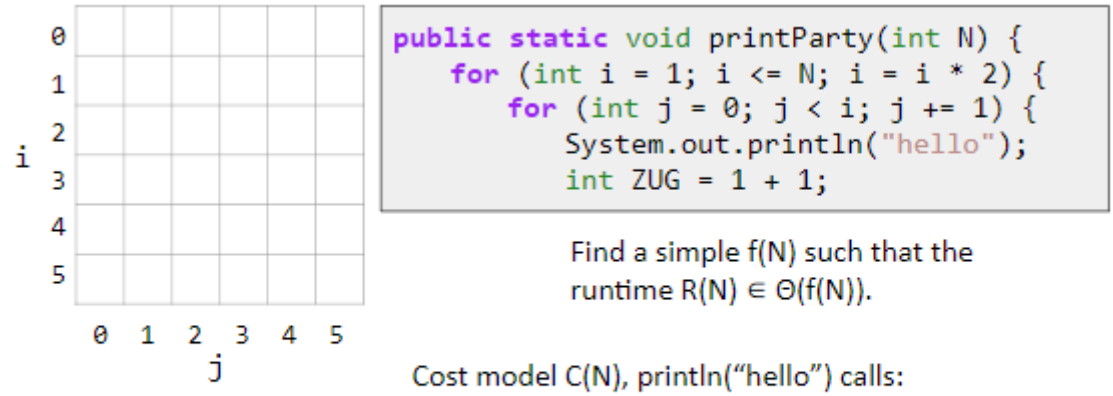
$$\Theta(N^2)$$

Nested For Loops With Geometric Outer Loop

Find a simple f(N) such that the runtime R(N) is in the same Big Theta family. By simple, we mean there should be no unnecessary multiplicative constants or additive terms.

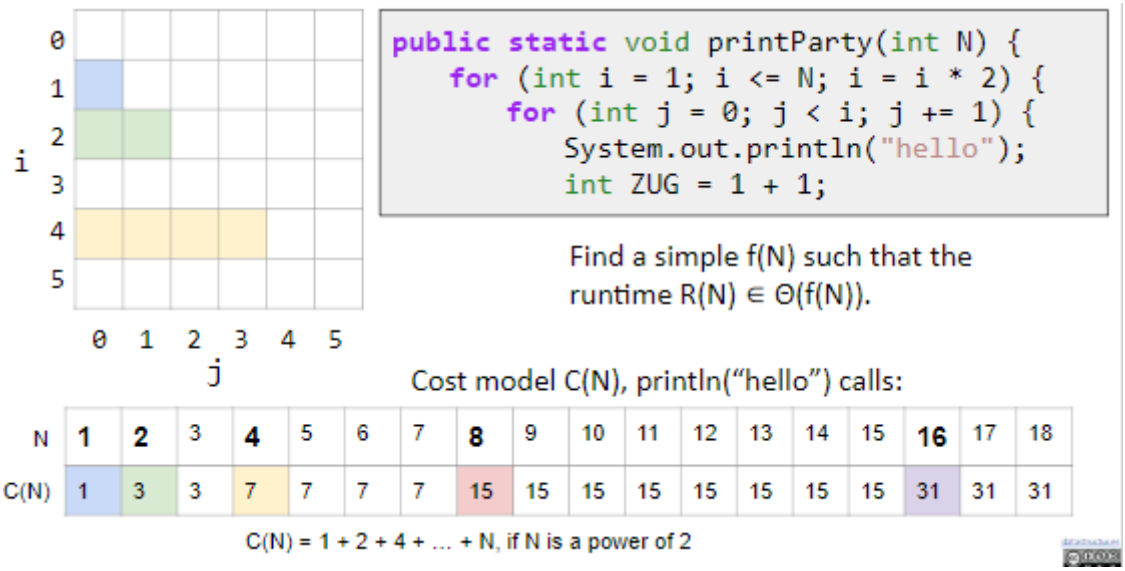
```
public static void printParty(int N) {
    for (int i = 1; i <= N; i = i * 2) {
        for (int j = 0; j < i; j += 1) {
            System.out.println("hello");
            int ZUG = 1 + 1;
        }
    }
}
```

Well, let's go through how to analyze this. We will go through the following table, for every value of N, what is the C(N) (cost of N)?



N	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
C(N)	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?

We find we will end up with the following table:



N	**1**	**2**	3	**4**	5	6	7	**8**	9	10	11	12	13	14	15	**16**	17	18
C(N)	1	3	3	7	7	7	7	15	15	15	15	15	15	15	15	31	31	31

C(N) = 1 + 2 + 4 + ... + N, if N is a power of 2

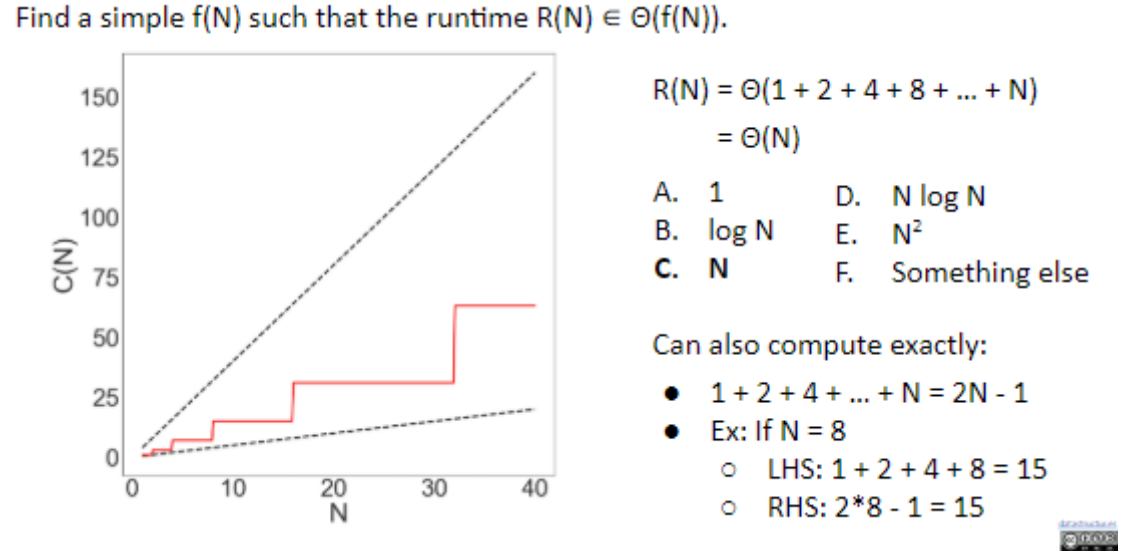
As it turns out,

$$C(N) = 1 + 2 + 4 + \ldots + N$$

leads to the runtime

$$\Theta(N)$$

and here's why:



Again, we can make both a mathematical argument, or the geometric argument.

Philosophy

There is usually no magic shortcut for these problems. Runtime analysis problems often require careful thought, and you will dive deeper into them in both CS70 and especially CS170.

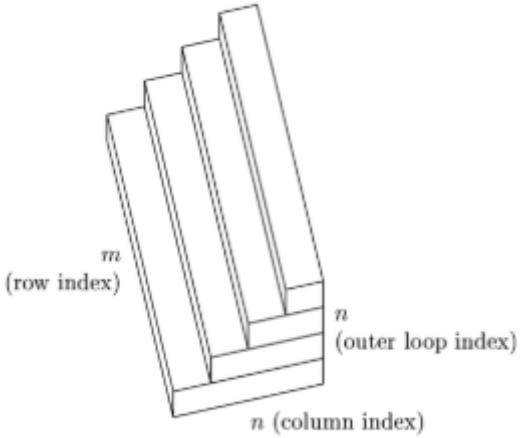
This is not a math class, but you should expect to know these two mathematical sums that we have covered in the past two examples:

$$1 + 2 + 3 + 4 + \ldots + N = N \times \frac{N-1}{2} \in \Theta(N^2)$$

$$1 + 2 + 4 + 8 + \ldots + N = 2N - 1 \in \Theta(N), \text{ where N is a power of two}$$

Some strategies we can use include:

- finding the exact sum
- writing out examples
- drawing pictures



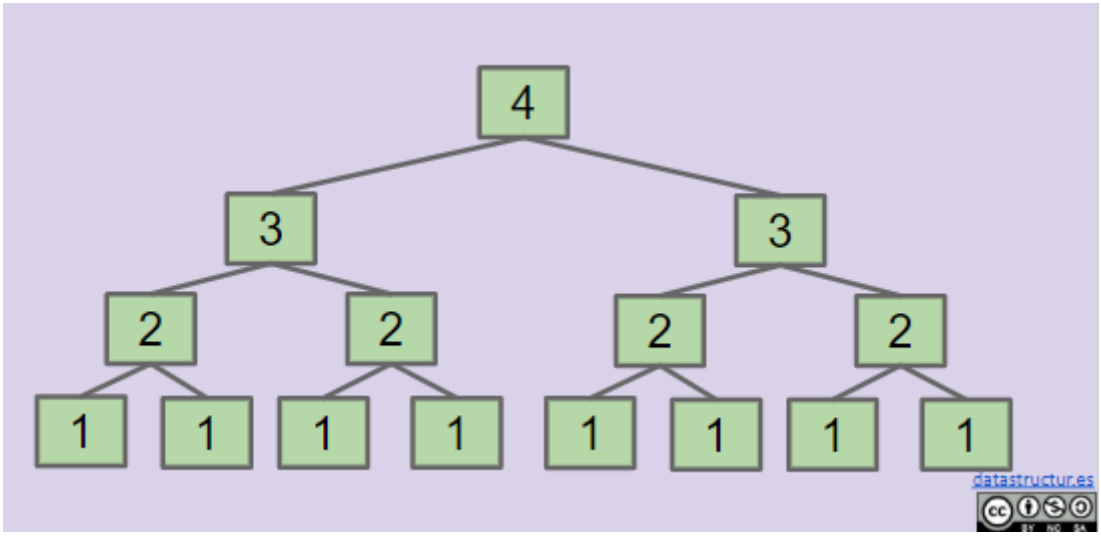
Here's an example of the QR decomposition runtime from "Numerical Linear Algebra" by Trefethen.

Recursion and Tree Recursion

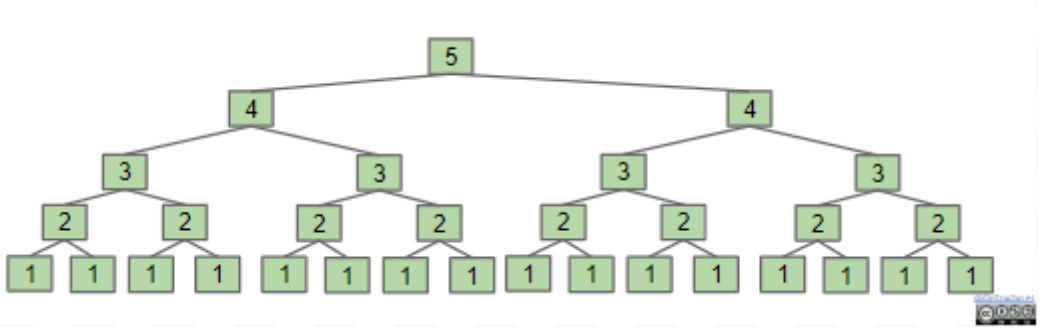
To discuss recursion, let's begin with an example problem. Find a simple f(N) to describe the runtime R(N):

```
public static int f3(int n) {
   if (n <= 1)
        return 1;
   return f3(n-1) + f3(n-1);
}
```

The following diagram may help you:



You will find this is an example of tree recursion from CS61A, where the call to `f3` on 4 will result in this specific tree of calls. We will see that if we then call it on 5, the runtime will increase exponentially:



To describe this, we will say it is in the family:

$$\Theta(2^N)$$

Another approach is to perform exact counting. We find that if we call:

- `C(1)` = 1
- `C(2)` = 1 + 2
- `C(3)` = 1 + 2 + 4

We find that `C(N)` is equal to:

$$1 + 2 + 4 + \ldots + 2^{N-1}$$

Knowing that the sum of the first powers of 2 from before, if the last term is a power of 2, then:

$$1 + 2 + 4 + \ldots + N = 2N - 1$$

Thus, we should calculate it as:

$$C(N) = 2(2^{N-1}) - 1 = 2^N - 1$$

Since the work during each call is constant, we can thus finalize our runtime as

$$\Theta(2^N)$$

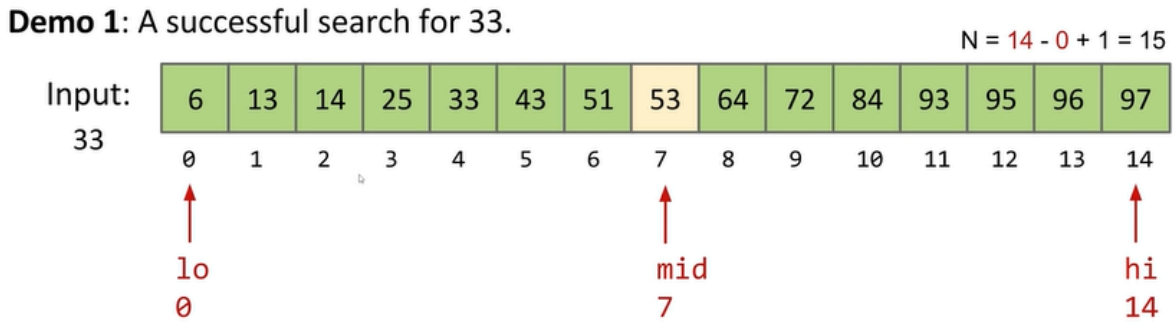
A third approach is what we call "recurrence relation". Given that `C(1)` is 1, what is `C(N)`?

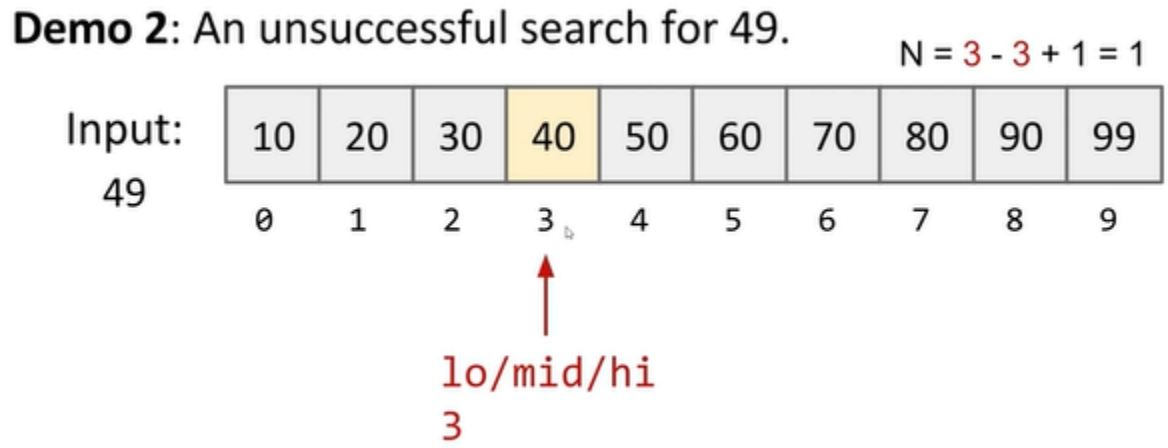
Really digging into the recursion aspect, we know we double the work of one level lower, plus one operation of constant time in the current call. This is a bit more technical, so you don't need to know it, but here's how to do it:

$$
\begin{aligned}
C(1) &= 1 \\
C(N) &= 2C(N-1)+1 \\
&= 2(2C(N-2)+1)+1 \\
&= 2(2(2C(N-2)+1)+1)+1 \\
&= 2(\cdots 2\cdot 1+1)+1)+\cdots 1 \\
&= \underbrace{2(\cdots 2}_{N-1})\cdot 1+1)+\cdots 1 \\
&= 2^{N-1}+2^{N-2}+\cdots+1 = 2^N-1 \in \Theta(2^N)
\end{aligned}
$$

Binary Search

Let's consider the runtime of another recursive function: the binary search. The binary search finds a key in a sorted array, where we compare the key to the middle entry. If our value is too small, go left, and go right if too big. If it is equal, then we have found our value.





Binary search is very interesting, because it's a very important idea, but also a challenging one to implement properly. The idea was first published in 1946, and the first correct implementation was in 1962. There are many incorrect implementations: in 2006, a bug was discovered in Java's binary search algorithm.

```java
public static int binarySearch(int[] a, int key) {
    int low = 0;
    int high = a.length - 1;

    while (low <= high) {
        int mid = (low + high) / 2;
        int midVal = a[mid];
        if (midVal < key)
            low = mid + 1;
        else if (midVal > key)
            high = mid - 1;
        else
            return mid; // key found
    }
    return -(low + 1);  // key not found.
}
```

Above is the incorrect implementation of Java's binary search, which works well enough for our purposes today (but see if you can spot the mistake!).

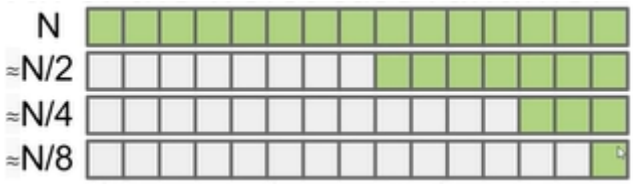
Find the runtime in terms of `N`, which is equal to `hi` - `lo` + 1 (the number of items being considered). Intuitively, what is the order of growth of the worst case runtime?

The answer is that it is:

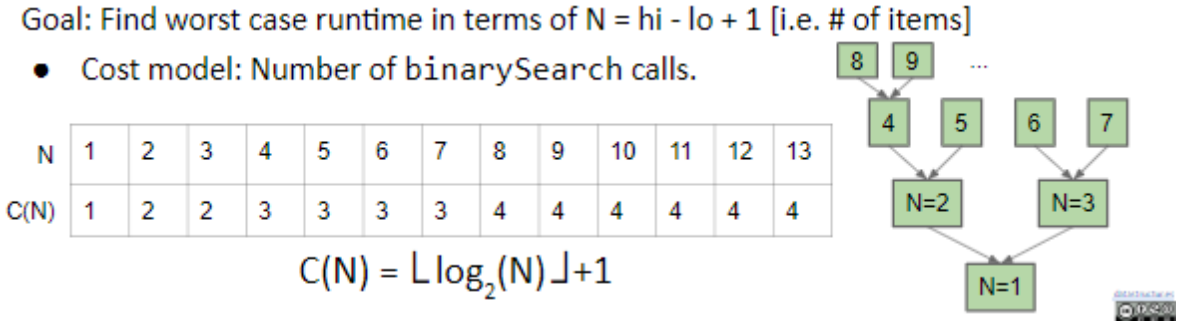
$$\Theta(log_2 N)$$

Why? The problem size halves over and over until it gets down to 1. If C is the number of calls, then solve for:

$$1 = \frac{N}{2^C} \Rightarrow C = log_2(N)$$



If you're looking very carefully, you'll notice it's not exactly halving... it's less than half, and we will delve more deeply into the rigorous proof.



Since each call takes constant time,

$$R(N) = \Theta(\lfloor log_2(N) \rfloor)$$

This f(N) is way too complicated. Let's simplify. We can use three handy properties to help us do this:

- $$\lfloor f(N) \rfloor = \Theta(f(N))$$

- $\lceil f(N) \rceil = \Theta(f(N))$
- $log_p(N) = \Theta(log_Q N)$

Since the base is irrelevant, we can thus simplify to:

$$\Theta(logN)$$

> ### Log Time is Really Terribly Fast
>
> In practice, logarithmic time algorithms have almost constant runtimes. Even for incredibly huge datasets, practically equivalent to constant time.

N	$\log_2$ N	Typical runtime (seconds)
100	6.6	1 nanosecond
100,000	16.6	2.5 nanoseconds
100,000,000	26.5	4 nanoseconds
100,000,000,000	36.5	5.5 nanoseconds
100,000,000,000,000	46.5	7 nanoseconds

Mergesort

Selection Sort: A Prelude to Mergesort

Earlier in class, we discussed a sorting algorithm called selection sort, which works as such:

- Find the smallest unfixed term, move it to the front, and "fix" it.
- Sort the remaining unfixed items using selection sort.

The runtime of selection sort is

$$\Theta(N^2)$$

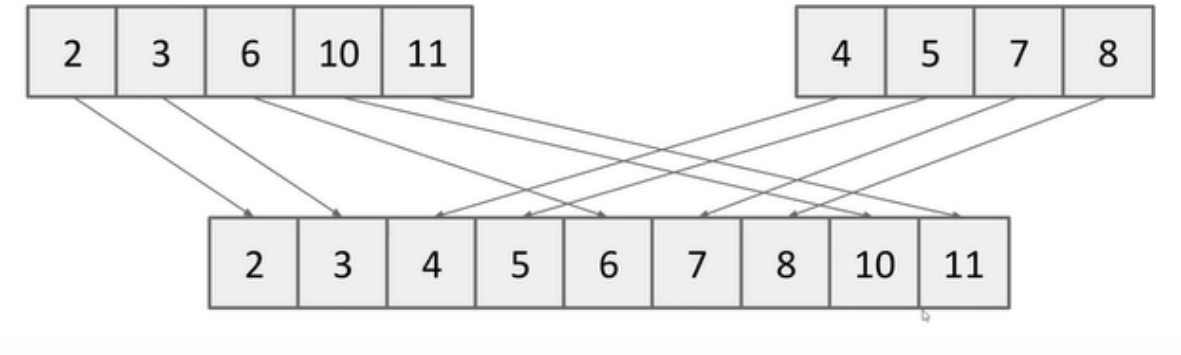
- Look at all N unfixed items to find smallest.
- Then look at N - 1 remaining unfixed.
- So on and so forth, the sum becomes:

$$2 + 3 + 4 + 5 + \ldots + N = \Theta(N^2)$$

Given the runtime is quadratic, for N = 64, we might say the runtime for selection sort is 4096 arbitrary units of time (AU). For N = 6, we find it takes approximately 36 arbitrary units of time.

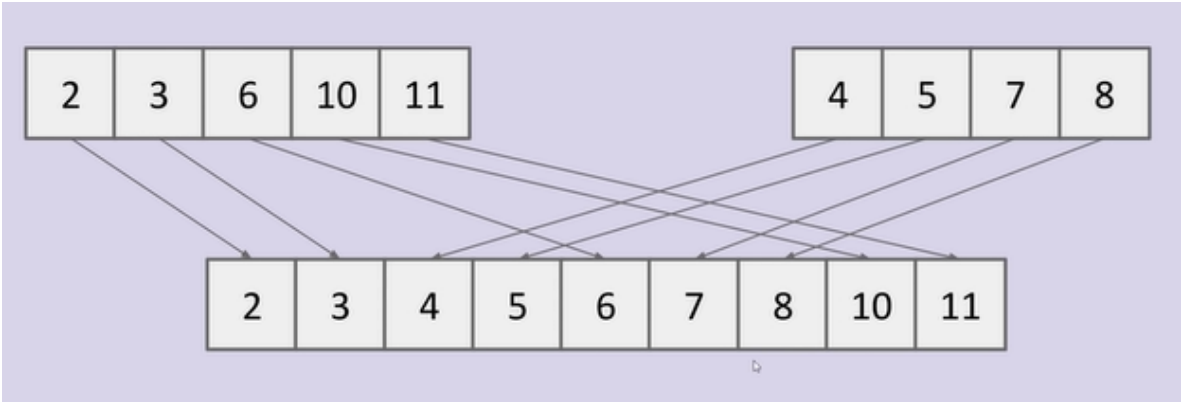
These are rough estimates, and the AU concept is not well-defined in mathematics, but this notion will be helpful to us when we discuss mergesort.

What is merging anyway? It takes two sorted arrays and combines them into a single sorted array by successively copying the smallest item from the two arrays into a target array.



In the next part, we will discuss how to optimize selection sort using this merging algorithm.

Practice



How does the runtime of merge grow with N, the total number of items?

Solution

Using array writes as the cost model, merge does exactly N writes, thus it is:

$$\Theta(N)$$

Using Merge to Speed Up Sort

Merging can give us an improvement over vanilla sort:

- Selection sort the left half, which takes:

$$\Theta(N^2)$$

- Selection sort the right half, which takes:

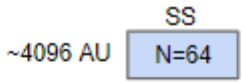
$$\Theta(N^2)$$

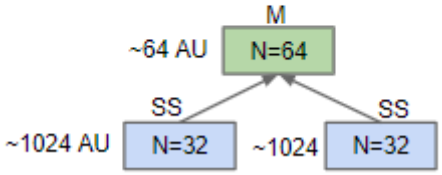
- Then merge the results, which takes:

$$\Theta(N)$$

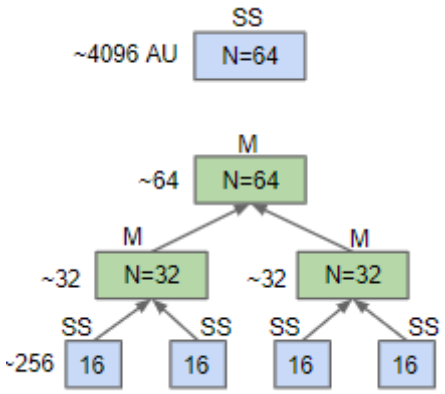
The idea is that since selection sort is so slow, this will yield a tremendous improvement, even though it is still in the N^2 family.

We will use our AU concept to quantify by how much. the merge will take N time: 64 AUs. The two selection sorts, for N = 32, will take 1024 AUs, and multiplied by two this becomes 2048 AUs. Thus, the overall operation takes 2112 AUs, a little more than half of what our original operation took (but still in N^2).





We shouldn't be satisfied with this marginally faster algorithm, and rather than slicing things down into two arrays, we can add a second layer:



The runtime for each sort here takes 256 AUs for each selection sort, multiplied by 4, and the merges take 128 AUs (64 + 2 * 32)., which means it takes 1152 AUs overall.

Here's one final idea: what if we just mergesort **all the way down**?

- If array is of size 1, return.
- Mergesort the left half.
- Mergesort the right half.
- Merge the results (linear time)

The total runtime to merge all the way down here is 384 AUs:

- The top layer: 64 AUs
- Second layer: 2 * 32 = 64 AUs
- Third layer = 4 * 16 = 64 AUs
- Overall runtime in AU is ~64k, where the k is the number of layers.
- $$k = log_2(64) = 6$$

  so ~384 total AUs

Using this information, for an array size of `N`, what is the worst case runtime of mergesort?

As we saw above, and the most important argument in this entire proof: every level takes roughly a constant amount of work: about N AUs. The total runtime is N*k, where k is the number of levels. And we know from above the levels is equal to:

$$k = log_2(N)$$

Thus, the overall runtime is:

$$\Theta(NlogN)$$

The exact count proof is tedious, and not covered in here.

> ### Linear vs Linearithmic
>
> By the same logic where we argued log N is basically constant time in practice, N log N is basically as good as N:
>
> **Linear vs. Linearithmic (N log N) vs. Quadratic**
>
> N log N is basically as good as N, and is vastly better than N^2.
>
> - For N = 1,000,000, the log N is only 20.

	n	$n \log_2 n$	n^2	n^3	1.5^n	2^n	$n!$
$n = 10$	< 1 sec	< 1 sec	< 1 sec	< 1 sec	< 1 sec	< 1 sec	4 sec
$n = 30$	< 1 sec	< 1 sec	< 1 sec	< 1 sec	< 1 sec	18 min	10^{25} years
$n = 50$	< 1 sec	< 1 sec	< 1 sec	< 1 sec	11 min	36 years	very long
$n = 100$	< 1 sec	< 1 sec	< 1 sec	1 sec	12,892 years	10^{17} years	very long
$n = 1{,}000$	< 1 sec	< 1 sec	1 sec	18 min	very long	very long	very long
$n = 10{,}000$	< 1 sec	< 1 sec	2 min	12 days	very long	very long	very long
$n = 100{,}000$	< 1 sec	2 sec	3 hours	32 years	very long	very long	very long
$n = 1{,}000{,}000$	1 sec	20 sec	12 days	31,710 years	very long	very long	very long

Table 2.1 The running times (rounded up) of different algorithms on inputs of increasing size, for a processor performing a million high-level instructions per second. In cases where the running time exceeds 10^{25} years, we simply record the algorithm as taking a very long time.

(from Algorithm Design: Tardos, Kleinberg)

Summary

To summarize today's lesson, theoretical analysis of algorithm performance requires careful thought, and there are no magic shortcuts for analyzing code.

In CS61B, it's OK to do either exact counting or intuitive analysis. It's important to remember the sums of:

$$1 + 2 + 3 + .. + N \in O(N^2)$$

$$1 + 2 + 4 + \ldots + N \in O(N)$$

We will not worry about mathematical proofs in this class.

Many runtime problems you'll do in this class resemble one of the five problems from today. See textbook, study guide, and discussion for more practice. This topic has one of the highest skill ceilings of all topics in the course.

Different solutions to the same problem, e.g. sorting, may have different runtimes.

- N^2 vs. N log N is an enormous difference.
- Going from N log N to N is nice, but not a radical change.