More on Big O

SLIDES ADAPTED FROM [TODD WITTMAN, CITADEL]

Ex: For Loops

- The run time is usually determined by the loops in the algorithm.
- Ex Given an integer N, print out the #s 1...N twice.

```
for (int i=1; i <= N; i++)
      cout << i << “ “;
for (int j=1; j <= N; j++)
      cout << j << " ";
```

```
N=5
1 2 3 4 5
1 2 3 4 5
```

- Prints total 2N numbers, so the run time is O(N).
- Each loop is O(N), so two loops is still O(N).

Ex: Nested For Loops

- Ex Print out #s in a table.

```
for (int i=1; i <= N; i++) {
        for (int j=1; j <= N; j++)
                cout << i << “ “;
        cout << “\n”;
}
```

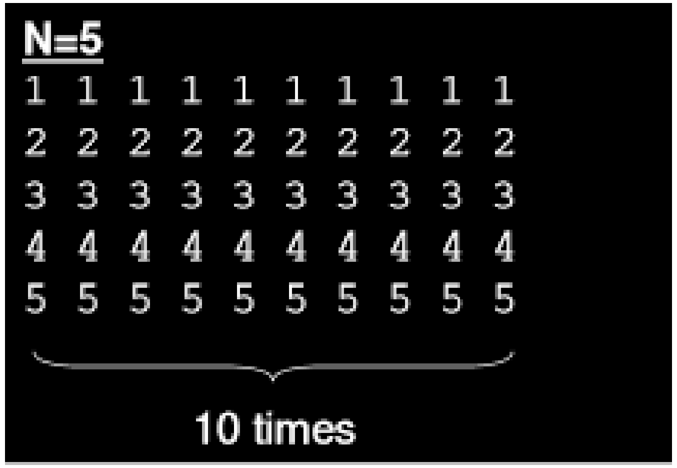
```
N=5
1 1 1 1 1
2 2 2 2 2
3 3 3 3 3
4 4 4 4 4
5 5 5 5 5
```

- In each row, it prints the row # N times.
- For each iteration of the outer loop, the inner loop runs N times. Since the outer loop runs N times, we have a total of (N)(N) = N^2 print statements.
- So T = $O(N^2)$.
- What if we had 3 nested for loops?

Ex: Nested For Loops

- We can often figure out the run time by counting the nested loops.
- But not always...

```
for (int i=1; i <= N; i++)  {
      for (int j=1; j <= 10; j++)
            cout << i << " ";
      cout << "\n";
}
```



- For every iteration of the outer loop, we print 10 numbers. Since the outer loop runs N times, the run time is T = 10 N = O(N).
- What if we changed the j<=10 to j<=1000000?

Notes About Big O

- Ignore all constants

  <u>Ex</u> $T = 45678\, N^3 = O(N^3)$

- The highest-order term dominates

  <u>Ex</u> $T = 2\, N^5 + 347\, N^3 - 20\, N^2 + 5500 = O(N^5)$

- Technically, if $T = N$ then $T=O(N)$ and also $T = O(N^2)$ because $T = N < N^2$. But whenever we report the running time, we always want the minimal order $O(N)$. Don't try to be cute...

Big O Memory

- In addition to the runtime, we can express the memory requirements using Big O.
- Ex Find the sum of all integers in a file.

```
while (fin >> num)
    v.push_back(num);
int sum = 0;
for (int i=0; i<v.size(); i++)
        sum += v[i];
```

- Runtime: O(N)
- Memory: O(N)

```
int sum = 0;
while (fin >> num)
        sum += num;
```

Runtime: O(N)

Memory: O(1)

FASTEST

Run Time	Name	Example Algorithm
$O(1)$	constant	hash table lookup, linked list insertion
$O(\log N)$	logarithmic	binary search in sorted list
$O(N)$	linear	vector insertion
$O(N \log N)$	polylogarithmic	mergesort
$O(N^2)$	quadratic	selection sort
$O(2^N)$	exponential	Subset Sum Problem
$O(N!)$	factorial	recursively list all permutations

SLOWEST

- We usually want the fastest algorithm. But not always...

Issue #1: Data Set Size

- Question: Which algorithm would you choose?

	Algorithm A	Algorithm B
Run time:	$T = 20000N = O(N)$	$T = 2N^2 = O(N^2)$

- Answer: It depends on the size of the data N.
- Normally we would say the O(N) algorithm is the better choice.
- Setting $20000N = 2N^2$ gives $N=100$.
- So for $N<100$, Algorithm B is faster.
- For $N>100$, Algorithm A is faster.
- Note that Big O only tells us which is better for large N.

Issue #2: Memory Limitations

- Question: Which algorithm would you choose?

	Algorithm A	Algorithm B
Run time:	$O(N \log N)$	$O(N^2)$
Memory:	$O(N^3)$	$O(N)$

- Answer: It depends on how much memory your computer can store.
- Choose Algorithm A only if we have lots of memory available.
- Its a general rule in Computer Science that you in order to make your algorithm faster you have to eat up more memory.

Issue #3: Programming Time

- Question: Which algorithm would you choose?

	Algorithm A	Algorithm B
Run time:	$O(N \log N)$	$O(N^2)$
Programming time:	1 week	1 hour

- Answer: It depends how much time you have to write the program.
- Another general rule in Computer Science is that the faster the algorithm is, the more complicated it will be.

Issue #4: Quality of Solution

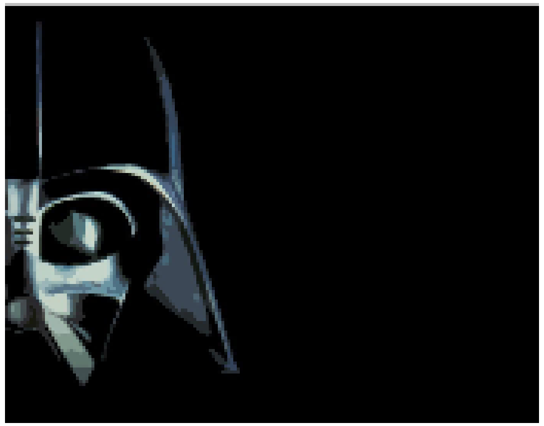
- Question: Which algorithm would you choose?

	Algorithm A	Algorithm B
Run time:	$O(N2^N)$	$O(N^2)$
Solution:	Finds best answer.	Finds good answer.

- Answer: It depends which is more important: time or the quality of the solution.
- An algorithm that finds a good answer, but not necessarily the optimal solution, is called an approximation algorithm.
- Approximation algorithms are generally faster than the algorithm that is guaranteed to find the best answer.

The Subset Sum Problem

- <u>Subset Sum Problem (SSP)</u>: Given a set S of integers and a target integer t, find the subset in S whose sum comes closest to t without going over.



<u>Ex</u> $S = \{ 1, 3, 4, 6, 13, 42 \}$ $t = 12$

Choose the subset $\{1,4,6\}$ because

$$1+4+6 = 11 \leq 12$$

- If size(S)=N, then there are 2^N subsets in N.
- It takes O(N) time to list one subset and find its sum.
- An <u>*enumerative algorithm*</u> that listed all possible subsets and chose the best would take exponential $O(N2^N)$ time.
- Can we do better?

The Subset Sum Problem

- A greedy algorithm would try to repeatedly choose the largest number so that the sum doesn't exceed the target t.

Greedy Algorithm

Input: Set of integers S, target integer t.

Output: List of integers L.

    Set m = 0
    while m <= t and S ≠Ø
        Set m = largest integer in S less than t *(Stop if no such M)*
        Add m to L and remove M from the list S
        Set t = t - M

- What is the running time of this algorithm?

The Subset Sum Problem

- The greedy algorithm finds the max among N integers, then among N-1 integers, then among N-2 integers, and so on until S is empty or the target is reached.
- So the number of integers checked is at most

  $$T \leq N + (N-1) + (N-2) + \ldots + 3 + 2 + 1$$

- Use the mathematical formula:

  $$\sum_{k=1}^{k=N} k = \frac{N(N+1)}{2}$$

- Then we see that

  $$T \leq 1/2\, N(N+1) = .5N^2 + .5\, N = O(N^2)$$

The Subset Sum Problem

- $O(N^2)$ is much faster than exponential time.
- But does it always produce the best answer?

  S = { 4, 4, 6, 6, 7, 10}   t = 20
- The greedy algorithm chooses 10+7 =17
- But the best answer is 4+4+6+6=20

$$\%Error = \frac{|Best - Answer|}{|Best|} = \frac{|20-17|}{|20|} = \frac{3}{20} = 0.15 = 15\%$$

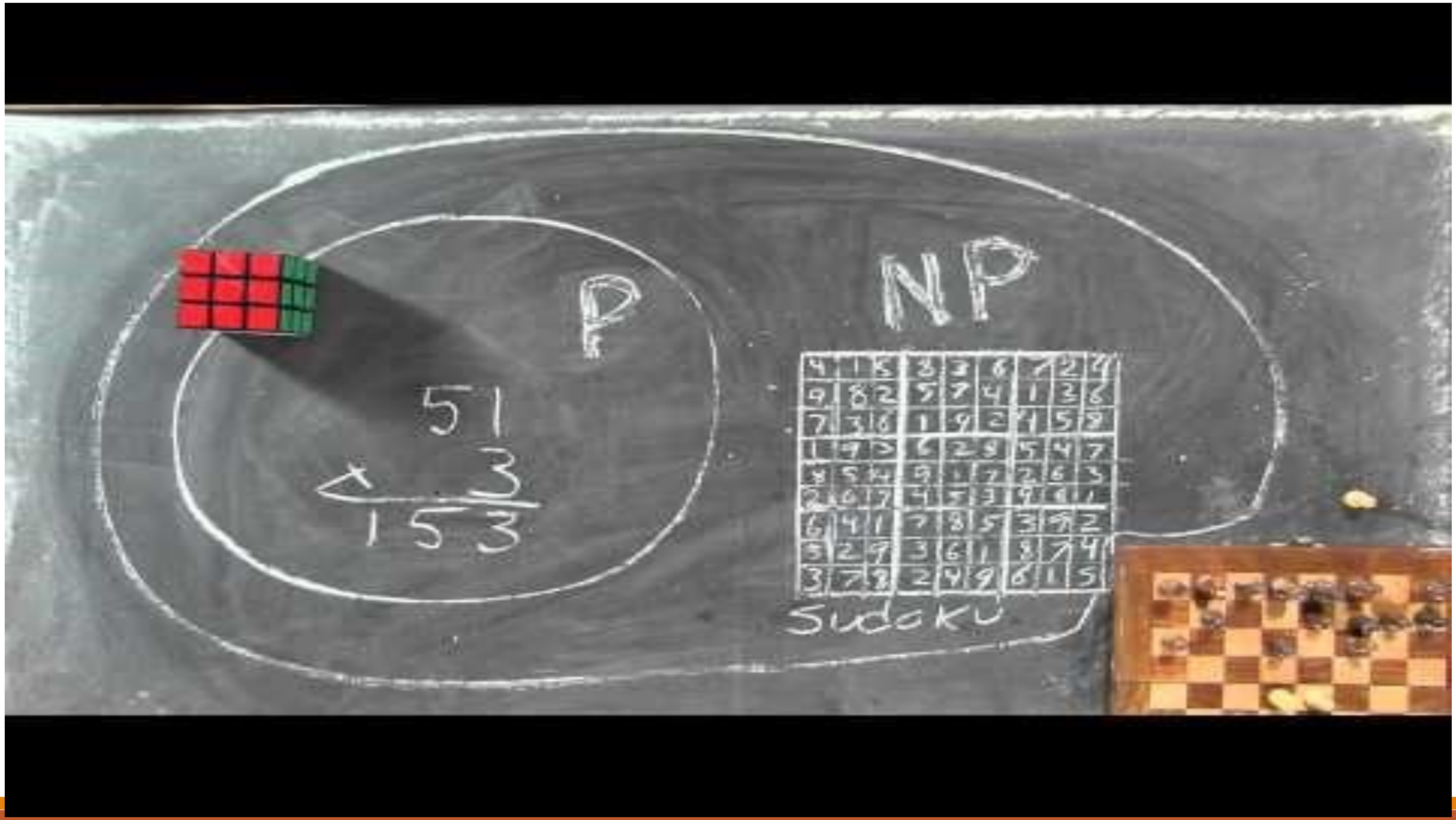
- So our greedy algorithm is fast and usually produces a good answer, but it does not necessarily produce the best answer.
- The greedy algorithm is an *approximation algorithm*.

The NP Class

- The only known algorithm that is guaranteed to find the best solution to the Subset Sum Problem has $O(N2^N)$ exponential running time.
- But this doesn't mean that there is no polynomial time algorithm.
- The class of algorithms for which there is no known polynomial time algorithm is called NP (non-polynomial).
- NP includes the Subset Sum Problem and the Traveling Salesman Problem.

The P≠NP Conjecture: Find a polynomial time algorithm for the Subset Sum Problem or prove there does not exist a polynomial time algorithm.

This is one of the unsolved Clay Millenium Problems.



https://www.youtube.com/watch?v=YX40hbAHx3s