DSC 40B

Lecture 3 : Nested Loop (dependent).

Mic!

Careful!

- **Not every nested loop has $\Theta(n^2)$ time complexity!**
- In general, if:
  - outer loop iterates a times;
  - inner loop iterates b times for each outer loop iteration
    - *We are assuming here that the number of inner loop iterations doesn't depend on which outer loop iteration we're in!*
  - then the innermost loop body is executed $a \times b$ times.

```
for x in range(n):
    for y in range(n**2):
        print(x + y)
```

Dependent Nested Loops

Example 3: Tallest Snowman, Again

- Our previous algorithm for the tallest snowman computed height for each ordered pair of people.
  - i = 3 and j = 7 is the same as i = 7 and j = 3
- **Idea**: consider each *unordered* pair only once:

```
for i in range(n):
        for j in range(i + 1, n):
```

- What is the time complexity?

Pictorially

```
for i in range(4):
    for j in range(4):
        print(i, j)
```

(0,0)	(0,1)	(0,2)	(0,3)
(1,0)	(1,1)	(1,2)	(1,3)
(2,0)	(2,1)	(2,2)	(2,3)
(3,0)	(3,1)	(3,2)	(3,3)

Pictorially

```
for i in range(4):
    for j in range(i + 1, 4):
        print(i, j)
```

(0,1)	(0,2)	(0,3)
	(1,2)	(1,3)
		(2,3)

```python
def tallest_snowman_2(heights):
    max_height = -float('inf')
    n = len(heights)
    for i in range(n):
        for j in range(i+1, n):
            height = heights[i] + heights[j]
            if height > max_height:
                max_height = height
    return max_height
```

- **Goal**: How many time does line `height = heights[i] + heights[j]` run in total?
- Now inner nested loop **depends** on nested outer loop

Independent

```
for i in range(n):
    for j in range(n): ...
```

- Inner loop doesn't depend on outer loop iteration #.
- **Just multiply**: inner body executed $n \times n = n^2$ times.

Dependent

```
for i in range(n):
    for j in range(i,n): ...
```

- Inner loop depends on outer loop iteration #.
- Can't just multiply: inner body executed ??? times.

Dependent Nested Loops

```
for i in range(n):
    for j in range(i + 1, n):
        height = heights[i] + heights[j]
```

- **Idea**: find formula $f(\alpha)$ for "number of iterations of inner loop during outer iteration α"
- Then total: $\sum_{\alpha=1}^{n} f(\alpha)$

```
for i in range(n):
    for j in range(i + 1, n):
        height = heights[i] + heights[j]
```

- On outer iter. # 1, inner body runs ________?________ times. (i = 0)
- On outer iter. # 2, inner body runs ________________ times.
- On outer iter. # α, inner body runs ________________ times.
- The outer loop runs ____________ times.

```
for i in range(n):
    for j in range(1, n): #i = 0
        height = heights[i] + heights[j]
```

- On outer iter. # 1, inner body runs ____n - 1____ times. (i = 0)
- On outer iter. # 2, inner body runs ____?____ times. (i = 1)
- On outer iter. # α, inner body runs ________ times.
- The outer loop runs ________ times.

```python
for i in range(n):
    for j in range(2, n): #i = 1
        height = heights[i] + heights[j]
```

- On outer iter. # 1, inner body runs ____n - 1____ times. (i = 0)
- On outer iter. # 2, inner body runs ____n - 2____times. (i = 1)
- On outer iter. # α, inner body runs ____?____times. (i = α)
- The outer loop runs ____________ times.

```
for i in range(n):
    for j in range(α, n): #i = α
        height = heights[i] + heights[j]
```

- On outer iter. # 1, inner body runs ___n - 1___ times. (i = 0)
- On outer iter. # 2, inner body runs ___n - 2___ times. (i = 1)
- On outer iter. # α, inner body runs ___n - α___ times. (i = α)
- The outer loop runs ___?___ times.

```
for i in range(n):
    for j in range(α, n): #i = α
        height = heights[i] + heights[j]
```

- On outer iter. # 1, inner body runs ____n - 1____ times. (i = 0)
- On outer iter. # 2, inner body runs ____n - 2____ times. (i = 1)
- On outer iter. # α, inner body runs ____$n - \alpha$____ times. (i = α)
- The outer loop runs ____n____ times.

Totalling Up

- On outer iteration α, inner body runs $n - \alpha$ times.
    - That is, $f(\alpha) = n - \alpha$
- There are n outer iterations.
- So we need to calculate:

$$\sum_{\alpha=1}^{n} f(\alpha) = \sum_{\alpha=1}^{n} (n - \alpha)$$

$$\sum_{\alpha=1}^{n}(n - \alpha)$$

$$=$$

$$\sum_{\alpha=1}^{n}(n-\alpha)$$

$$=$$

$$\underbrace{(n-1)}_{\text{1st outer iter}} +$$

$$\sum_{\alpha=1}^{n}(n - \alpha)$$

$$=$$

$$\underbrace{(n - 1)}_{\text{1st outer iter}} + \underbrace{(n - 2)}_{\text{2nd outer iter}} +$$

$$\sum_{\alpha=1}^{n}(n - \alpha)$$

$$=$$

$$\underbrace{(n - 1)}_{\text{1st outer iter}} + \underbrace{(n - 2)}_{\text{2nd outer iter}} + \ldots + \underbrace{(n - \alpha)}_{\text{kth outer iter}} +$$

$$\sum_{\alpha=1}^{n}(n - \alpha)$$

$$=$$

$$\underbrace{(n - 1)}_{\text{1st outer iter}} + \underbrace{(n - 2)}_{\text{2nd outer iter}} + \dots + \underbrace{(n - \alpha)}_{\text{kth outer iter}} + \dots + \underbrace{(n - (n - 1)}_{\text{(n-1)th outer iter}} +$$

$$\sum_{\alpha=1}^{n}(n - \alpha)$$

$$=$$

$$\underbrace{(n - 1)}_{\text{1st outer iter}} + \underbrace{(n - 2)}_{\text{2nd outer iter}} + \dots + \underbrace{(n - \alpha)}_{\text{kth outer iter}} + \dots + \underbrace{(n - (n - 1)}_{\text{(n-1)th outer iter}} + \underbrace{(n - n)}_{\text{nth outer iter}}$$

$$\sum_{\alpha=1}^{n} (n - \alpha)$$

$$=$$

$$\underbrace{(n-1)}_{\text{1st outer iter}} + \underbrace{(n-2)}_{\text{2nd outer iter}} + \dots + \underbrace{(n-\alpha)}_{\text{kth outer iter}} + \dots + \underbrace{(n-(n-1)}_{\text{(n-1)th outer iter}} + \underbrace{(n-n)}_{\text{nth outer iter}}$$

$$1 + 2 + 3 + \dots + (n-3) + (n-2) + (n-1)$$

$$=$$

$$\sum_{\alpha=1}^{n}(n-\alpha)$$

$$=$$

$$\underbrace{(n-1)}_{\text{1st outer iter}} + \underbrace{(n-2)}_{\text{2nd outer iter}} + \ldots + \underbrace{(n-\alpha)}_{\text{kth outer iter}} + \ldots + \underbrace{(n-(n-1)}_{\text{(n-1)th outer iter}} + \underbrace{(n-n)}_{\text{nth outer iter}}$$

$$1 + 2 + 3 + \ldots + (n-3) + (n-2) + (n-1)$$

$$=$$

$$n(n-1)/2$$

Aside: Arithmetic Sums

- 1 + 2 + 3 + …+ (n-1) + **n** is an arithmetic sum.
- Formula for total: $n(n + 1)/2$.
- **You should memorize it!**

Time Complexity

- `tallest_snowman_2` has $\Theta(n^2)$ time complexity
- Same as original `tallest_snowman`
- Should we have been able to guess this? Why?

Reason 1: Number of Pairs

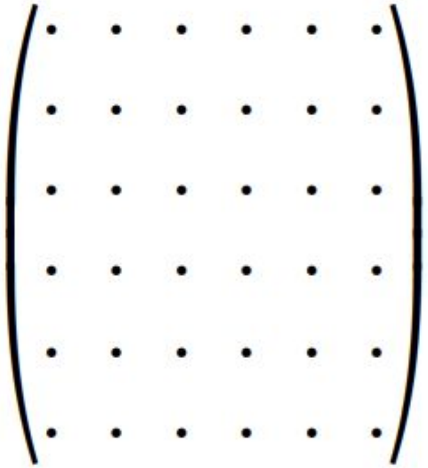
- We're doing constant work for each unordered pair.
- Recall from 40A: number of pairs of n objects is

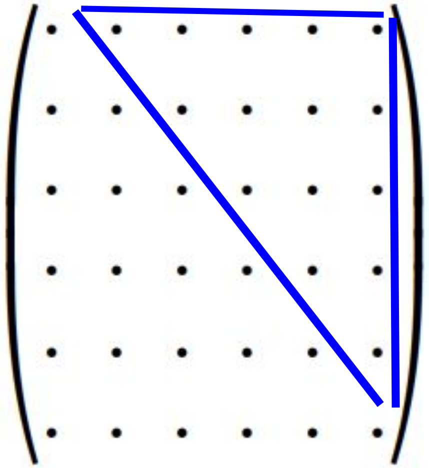
$$\binom{n}{2} = \frac{n!}{2!(n-2)!} = \frac{n(n-1)}{2}$$

- So $\Theta(n^2)$

Reason 2: Half as much work

- Our new solution does roughly half as much work as the old one.
- But Θ doesn't care about constants: 1/2 $\Theta(n^2)$ is still $\Theta(n^2)$.





Main Ideas

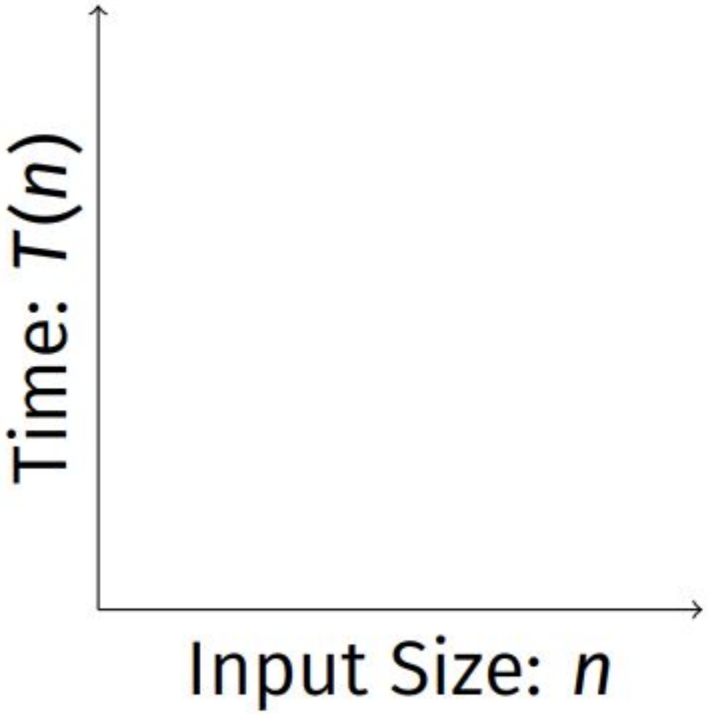
1. If the loops are **dependent**, you'll usually need:
   a. to write down a summation,
   b. evaluate.
2. Halving the work (or thirding, quartering, etc.) doesn't change the time complexity.

Exercise

Design a **linear** time algorithm for this problem.

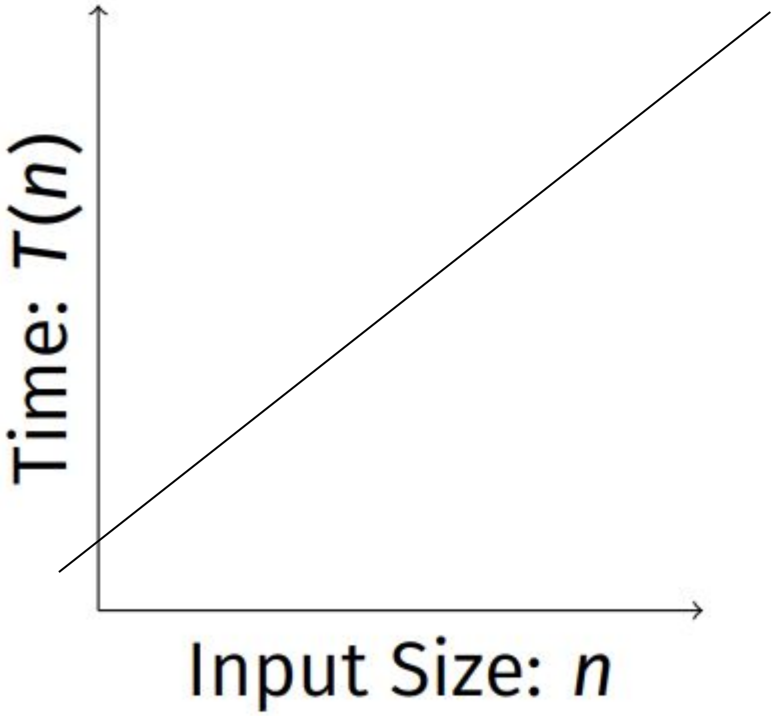
Growth Rates

Linear vs. Quadratic Scaling



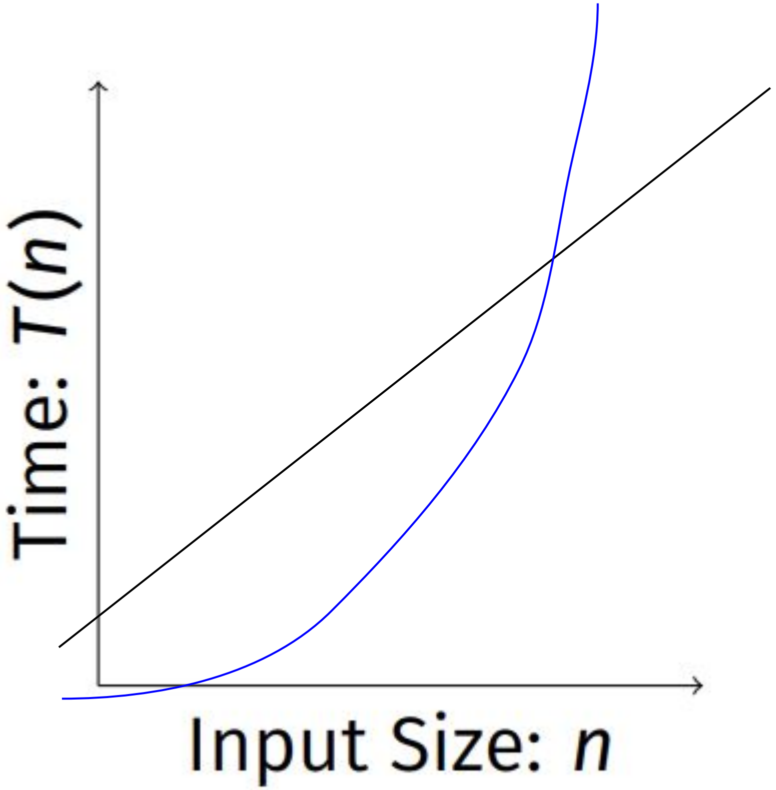
- $T(n) = \Theta(n)$ means "$T(n)$ grows like n"
- $T(n) = \Theta(n^2)$ means "$T(n)$ grows like n^2"

Linear vs. Quadratic Scaling



- $T(n) = \Theta(n)$ means "$T(n)$ grows like n"
- $T(n) = \Theta(n^2)$ means "$T(n)$ grows like n^2"

Linear vs. Quadratic Scaling



- $T(n) = \Theta(n)$ means "$T(n)$ grows like n"
- $T(n) = \Theta(n^2)$ means "$T(n)$ grows like n^2"

Definitions

1. An algorithm is said to run in **linear** time if $T(n) = \Theta(n)$.
2. An algorithm is said to run in **quadratic** time if $T(n) = \Theta(n^2)$.

Linear Growth

- If input size doubles, time roughly doubles.
- If code takes 5 seconds on 1,000 points...
- ...on 100,000 data points it takes ≈ 500 seconds.
- i.e., 8.3 minutes

Quadratic Growth

- If input size doubles, time *roughly* **quadruples**.
- If code takes 5 seconds on 1,000 points…
- …on 100,000 points it takes ≈ 50,000 seconds.
- i.e., ≈ 14 hours

In data science...

- Let’s say we have a training set of 10,000 points.
- If model takes **quadratic** time to train, should expect to wait minutes to hours.
- If model takes **linear** time to train, should expect to wait seconds to minutes.
- These are rules of thumb only.

Exponential Growth

- Increasing input size by **one** *doubles* (triples, etc.) time taken.
- **Grows very quickly!**
- **Example**: brute force search of 2^n subsets.

```
for subset in all_subsets(things):
    print(subset)
```

Logarithmic Growth

- To increase time taken by one unit, must *double* (triple, etc.) the input size.
- **Grows very slowly**
- $\log n$ grows slower than n^α for any $\alpha > 0$
  - I.e., $\log n$ grows *slower* than n, $\sqrt{n}$, $n^{1/1,000}$, etc.

Exercise

What is the asymptotic time complexity of the code below as a function of n?

```
i = 1
while i <= n:
    i = i * 2
```

A: Constant

B: Log

C: Linear

D: Quadratic

Solution

- Same general strategy as before: "how many times does loop body run?"

```
i = 1
while i <= n:
    i = i * 2
```

n	# iters.
1	
2	
3	
4	
5	
6	
7	
8	

Solution

- Same general strategy as before: "how many times does loop body run?"

```
i = 1
while i <= n:
    i = i * 2
```

n	# iters.
1	1
2	
3	
4	
5	
6	
7	
8	

Solution

- Same general strategy as before: "how many times does loop body run?"

```
i = 1
while i <= n:
    i = i * 2
```

n	# iters.
1	1
2	2
3	
4	
5	
6	
7	
8	

Solution

- Same general strategy as before: "how many times does loop body run?"

```
i = 1
while i <= n:
    i = i * 2
```

n	# iters.
1	1
2	2
3	2
4	
5	
6	
7	
8	

Solution

- Same general strategy as before: "how many times does loop body run?"

```
i = 1
while i <= n:
    i = i * 2
```

n	# iters.
1	1
2	2
3	2
4	3
5	
6	
7	
8	

Solution

- Same general strategy as before: "how many times does loop body run?"

```
i = 1
while i <= n:
    i = i * 2
```

n	# iters.
1	1
2	2
3	2
4	3
5	3
6	3
7	3
8	4

Common Growth Rates

- $\Theta(1)$: constant
- $\Theta(\log n)$: logarithmic
- $\Theta(n)$: linear
- $\Theta(n \log n)$: linearithmic
- $\Theta(n^2)$: quadratic
- $\Theta(n^3)$: cubic
- $\Theta(2^n)$: exponential

Question

- Which grows **faster**, $n!$ or 2^n ?

$n! = 1 * 2 * 3 * ..... * n$

$2^n = 2 * 2 * 2 .... * 2$

A: n!

B: 2^n

C: Same

D: Impossible to tell

Thank you!

Do you have any questions?

CampusWire!