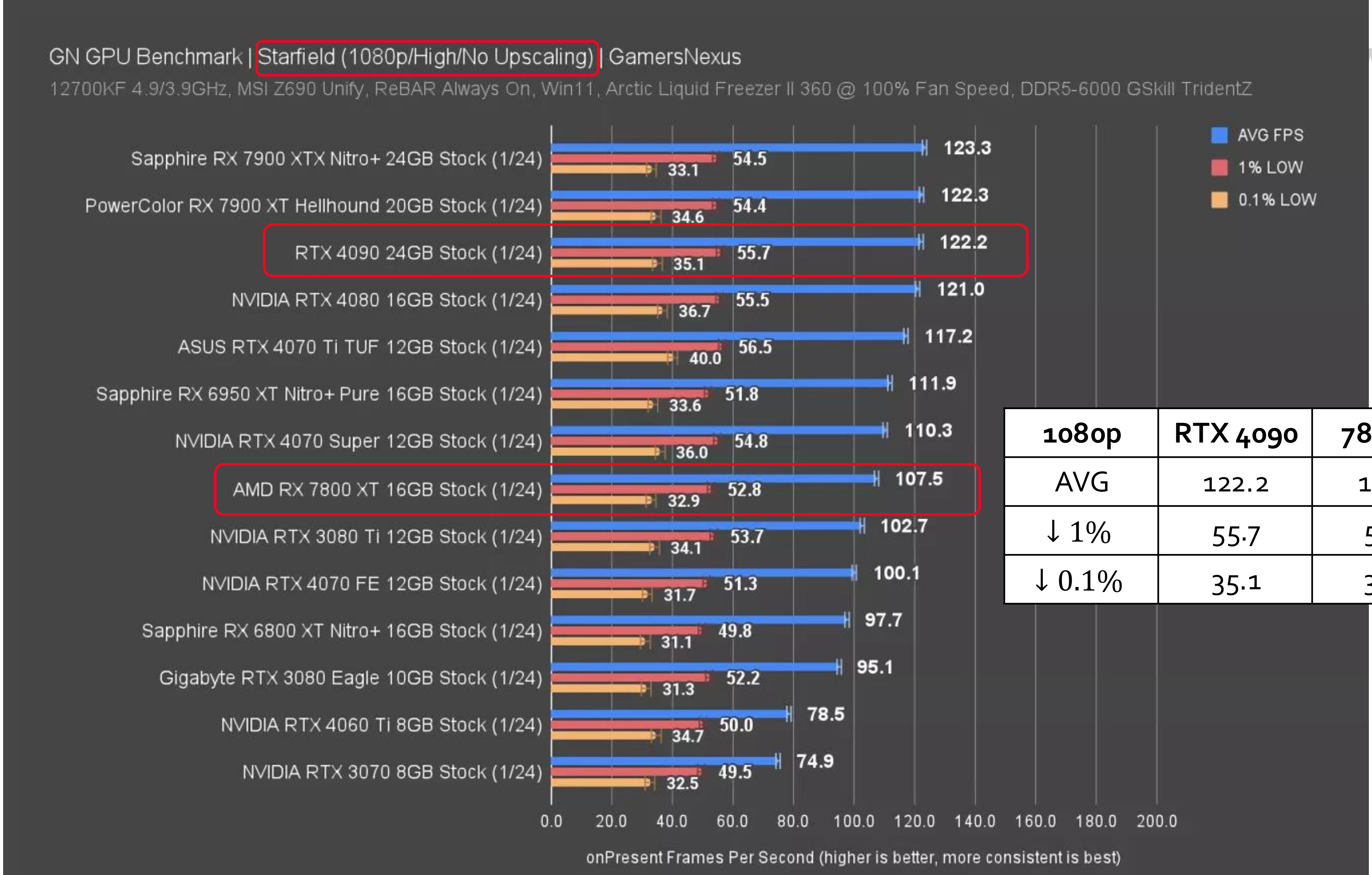
WASHINGTON STATE UNIVERSITY

CPTS 223 Advanced Data Structure C/C++

Algorithm Analysis

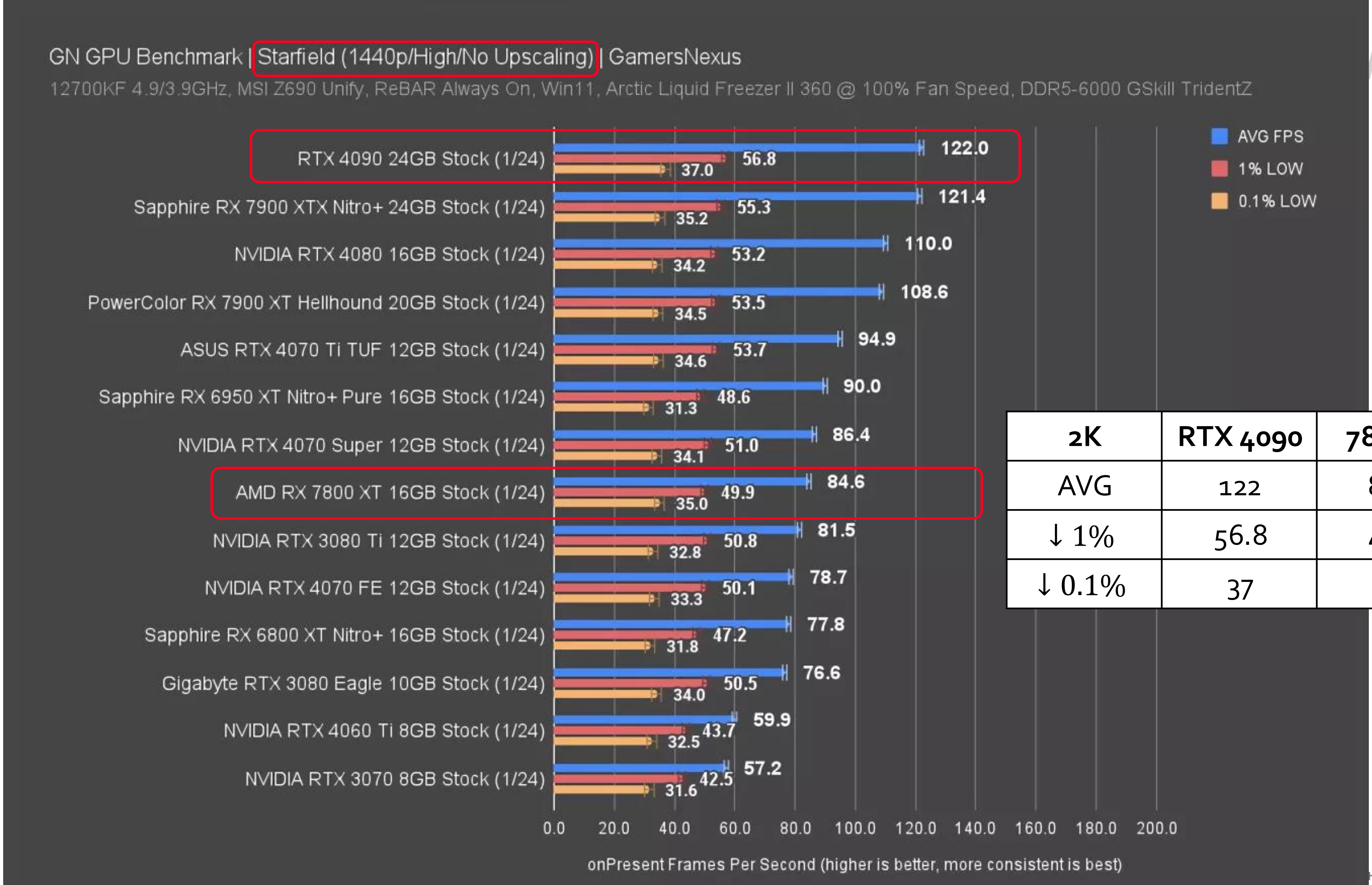
How to compare algorithms?

- Compare time requirements
  - How much time will it take to execute the algorithm?
- Compare space requirements
  - How much extra space is required for this algorithm to execute?
- Compare algorithms on some benchmarks?
  - An algorithm might run faster on one machine versus another
    - e.g., AMD vs Intel; AMD vs NVIDIA; Xbox vs PS vs Switch; Windows vs Mac; etc.)
  - The selected inputs for a given run of the algorithm might not be a representative sample
  - It takes a lot of time to maintain a set of benchmarks



1080p	RTX 4090	7800XT
AVG	122.2	107.5
↓ 1%	55.7	52.8
↓ 0.1%	35.1	32.9

Image credit: https://gamersnexus.net/gpus/nvidia-geforce-rtx-4070-super-review-benchmarks-vs-rtx-4070-rx-7800-xt-more



2K	RTX 4090	7800XT
AVG	122	84.6
↓ 1%	56.8	49.9
↓ 0.1%	37	35

Image credit: https://gamersnexus.net/gpus/nvidia-geforce-rtx-4070-super-review-benchmarks-vs-rtx-4070-rx-7800-xt-more

WSU

How to compare algorithms?

An easier algorithm?

Larger fps*?

A harder algorithm?

Smaller fps?

1080p	RTX 4090	7800XT
AVG	122.2	107.5
↓ 1%	55.7	52.8
↓ 0.1%	35.1	32.9

2K	RTX 4090	7800XT
AVG	122	84.6
↓ 1%	56.8	49.9
↓ 0.1%	37	35

*Fps: frames per second

How to compare algorithms?

An easier algorithm?

Larger fps*?

1080p	RTX 4090	7800XT
AVG	122.2	107.5
↓ 1%	55.7	52.8
↓ 0.1%	35.1	32.9

A harder algorithm?

Smaller fps?

2K	RTX 4090	7800XT
AVG	122	84.6
↓ 1%	56.8	49.9
↓ 0.1%	37	35

*Fps: frames per second

WSU

How to compare algorithms?

An easier algorithm?

Larger fps*?

1080p	RTX 4090	7800XT
AVG	122.2	107.5
↓ 1%	55.7	52.8
↓ 0.1%	35.1	32.9

A harder algorithm?

Smaller fps?

2K	RTX 4090	7800XT
AVG	122	84.6
↓ 1%	56.8	49.9
↓ 0.1%	37	35

*Fps: frames per second

How to compare algorithms?

An easier algorithm?

Larger fps*?

1080p	RTX 4090	7800XT
AVG	122.2	107.5
↓ 1%	55.7	52.8
↓ 0.1%	35.1	32.9

A harder algorithm?

Smaller fps?

2K	RTX 4090	7800XT
AVG	122	84.6
↓ 1%	56.8	49.9
↓ 0.1%	37	35

Which benchmark we should use?

*Fps: frames per second

Benchmarks can be sensitive

- Given the list {3, 9, 1, 2, 3, 5} as input
- Find(X): return the index of X
- We expect Find(3) to execute faster than Find(5)
  - Which is more representative of this input list, Find(3) or Find(5)?
  - How is our benchmark affected if we decide to use different lists?
    - e.g. {1, 2, 3, 3, 3, 4, 4, 4, 5, 9}
- Benchmarks are great for telling us how an algorithm will execute under a given set of circumstances
- but we ideally want something more generalizable

What is algorithm analysis?

- A mathematical technique for estimating the rate at which execution time grows relative to the size of its input parameters
- A formal name: asymptotic analysis

Is there non-asymptotic analysis?

- Asymptotic analysis is a method of estimation that groups algorithms based on their growth rate
- Asymptotic analysis is unable to tell us for sure how one algorithm will perform exactly in absolute timed execution relative to another
- but it does give us some good hints

WSU

What is algorithm analysis?

- Asymptotic analysis result: an example in AI/ML

Asymptotic analysis:
Problem scale approaches to infinitely large:
n→\infty

Proposition 1. *Denote* $z = \frac{j}{n}$ *and* $\gamma(z) := \sum_{l=0}^{q-1} z^l (1-z)^{s-l-1} \frac{s!}{l!(s-l-1)!}$. *Then, it holds that*

$$\boxed{\lim_{j,n\to\infty, j/n=z}} \gamma_j = \frac{1}{n}\gamma(z).$$

Moreover, it holds that $1 - \frac{1}{s}\gamma(z)$ *is the cumulative distribution function of* $Beta(z; q, s-q)$.

Kawaguchi, Kenji, and Haihao Lu. "Ordered sgd: A new stochastic optimization framework for empirical risk minimization." In *International Conference on Artificial Intelligence and Statistics*, pp. 669-679. PMLR, 2020. https://arxiv.org/abs/1907.04371

What is algorithm analysis?

- Asymptotic analysis result: an example in AI/ML

Asymptotic analysis: ignore dependencies other than ϵ, n

Table 1: Summary of complexity results of this work and previous works for findi... ϵ-duality-gap solution for SCSC or an ϵ-stationary solution for WCSC min-m... problems. We focus on comparison of existing results without assuming smooth... of the objective function. Restriction means whether an additional condition ...out the objective function's structure is imposed.

Setting	Works	Restriction	Convergence	Complexity
SCSC	Nemirovski et al. (2009)	No	Duality Gap	$O\left(1/\epsilon^2\right)$
	Yan et al. (2019)	Yes	Primal Gap	$O\left(1/\epsilon + n\log(1/\epsilon)\right)$
	This paper	**No**	**Duality Gap**	$\boldsymbol{O\left(1/\epsilon\right)}$
WCSC	Rafique et al. (2018)	No	Nearly Stationary	$\widetilde{O}\left(1/\epsilon^6\right)$
	Rafique et al. (2018)	Yes	Nearly Stationary	$\widetilde{O}\left(1/\epsilon^4 + n/\epsilon^2\right)$
	This paper	**No**	**Nearly Stationary**	$\boldsymbol{\widetilde{O}\left(1/\epsilon^4\right)}$

Yan, Yan, Yi Xu, Qihang Lin, Wei Liu, and Tianbao Yang. "Optimal epoch stochastic gradient descent ascent methods for min-max optimization." *Advances in Neural Information Processing Systems* 33 (2020): 5789-5800. https://arxiv.org/abs/2002.05309

WSU

What is algorithm analysis?

(1) Not necessarily infinitely large scale
(2) Not necessarily hide constants

- Non-asymptotic analysis result: an example in AI/ML

Theorem 1 *Suppose Assumption 1 and Assumption 2 hold and let* $\delta \in (0,1)$ *be a failing probability and* $\epsilon \in (0,1)$ *be the target accuracy level for the duality gap. Let* $K = \lceil \log(\frac{\epsilon_0}{\epsilon}) \rceil$ *and* $\tilde{\delta} = \delta / K$*, and the initial parameters are set by* $R_1 \geq 2\sqrt{\frac{2\epsilon_0}{\min\{\mu,\lambda\}}}$, $\eta_x^1 = \frac{\min\{\mu,\lambda\} R_1^2}{40(5+3\log(1/\tilde{\delta}))B_1^2}$, $\eta_y^1 = \frac{\min\{\mu,\lambda\} R_1^2}{40(5+3\log(1/\tilde{\delta}))B_2^2}$ *and*

$$T_1 \geq \frac{\max\left\{320^2(B_1+B_2)^2 3\log(1/\tilde{\delta}), 3200(5+3\log(1/\tilde{\delta}))\max\{B_1^2, B_2^2\}\right\}}{\min\{\mu,\lambda\}^2 R_1^2}$$

Then the total number of iterations of Algorithm 1 to achieve an ϵ*-duality gap, i.e.,* $Gap(\bar{x}_K, \bar{y}_K) \leq \epsilon$*, with probability* $1-\delta$ *is*

$$T_{tot} = \frac{\max\left\{320^2(B_1+B_2)^2 3\log(\frac{1}{\tilde{\delta}}), 3200(5+3\log(1/\tilde{\delta}))\max\{B_1^2, B_2^2\}\right\}}{4\min\{\mu,\lambda\}\epsilon}.$$

Yan, Yan, Yi Xu, Qihang Lin, Wei Liu, and Tianbao Yang. "Optimal epoch stochastic gradient descent ascent methods for min-max optimization." *Advances in Neural Information Processing Systems* 33 (2020): 5789-5800. https://arxiv.org/abs/2002.05309

Worst-case growth rate

- It is great to have a positive disposition, but as scientists and engineers, we need to know worst-case behavior so that we can plan accordingly (Why?)
- Worst-case analysis is called "Big O" (pronounced "Big Oh") analysis
- Big-O analysis categorizes algorithms based on their growth rate

WSU

Algorithm complexity

- T(n) is time to run given an input size of n elements
- T(n) = O(f(n)): exist [c, n_o] such that T(n) <= cf(n) when n >= n_o
  - e.g., T(n) <= 2.45 n^2, where f(n)=n^2
- T(n) = Ω(g(n)) when +[c, n_o] such that T(n) >= cg(n) when n >= n_o
  - e.g., T(n) >= 1.03 n^2, where g(n)=n^2
- T(n) = Θ(h(n)) if and only if T(n) = O(h(n)) and T(n) = Ω(h(n))
  - e.g., 1.03 n^2 <= T(n) <= 2.45 n^2, where h(n)=n^2

Bounds

- O(f(n)) is an UPPER bound of T(n) -- "Worst case can be no more than"
  ←T(n) <= cf(n)
- Ω(g(n)) is a LOWER bound of T(n) -- "Best case can be no faster than"
  ←T(n) >= cg(n)
- Only the order of the algorithm
- No details, e.g., constants (asymptotic analysis)
- T(n) = Θ(g(n)) is when O(g(n)) = Ω(g(n)) -- "It must be exactly"
- You will find Theta (Θ) also used as an average case

What is T(n)?

- T(n) is the time for a function to run
- It is more specific than O(n), since O(n) is only of the order:
- T(n) = n^2 + n+1
- O(n) = n^2

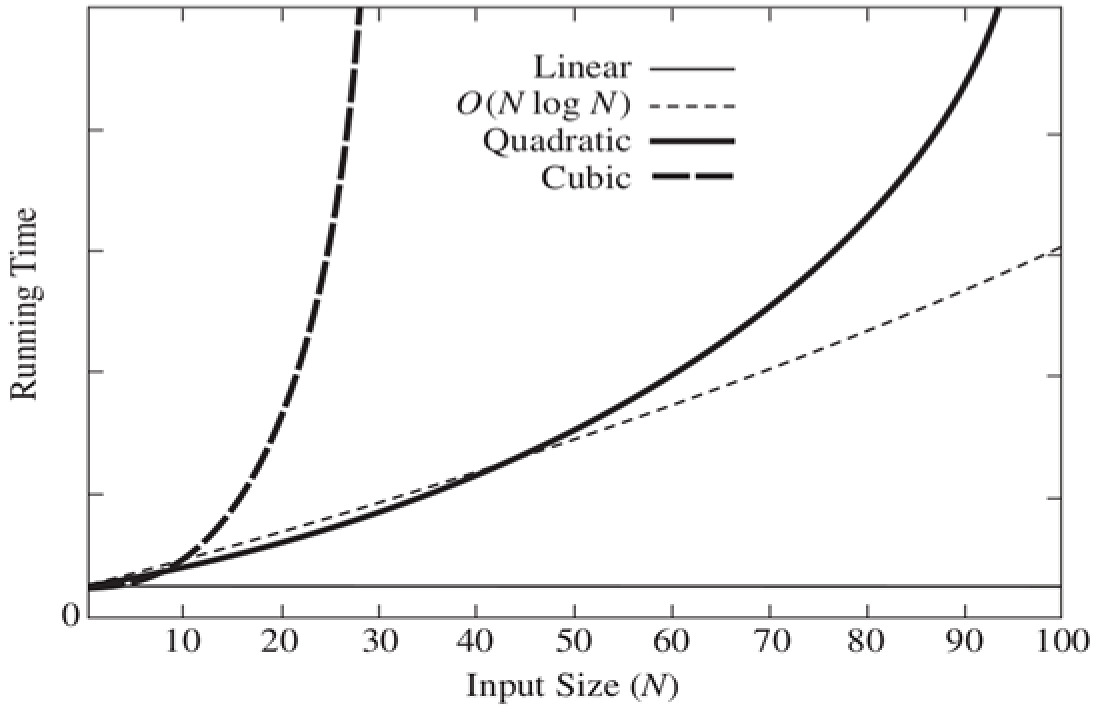
WSU

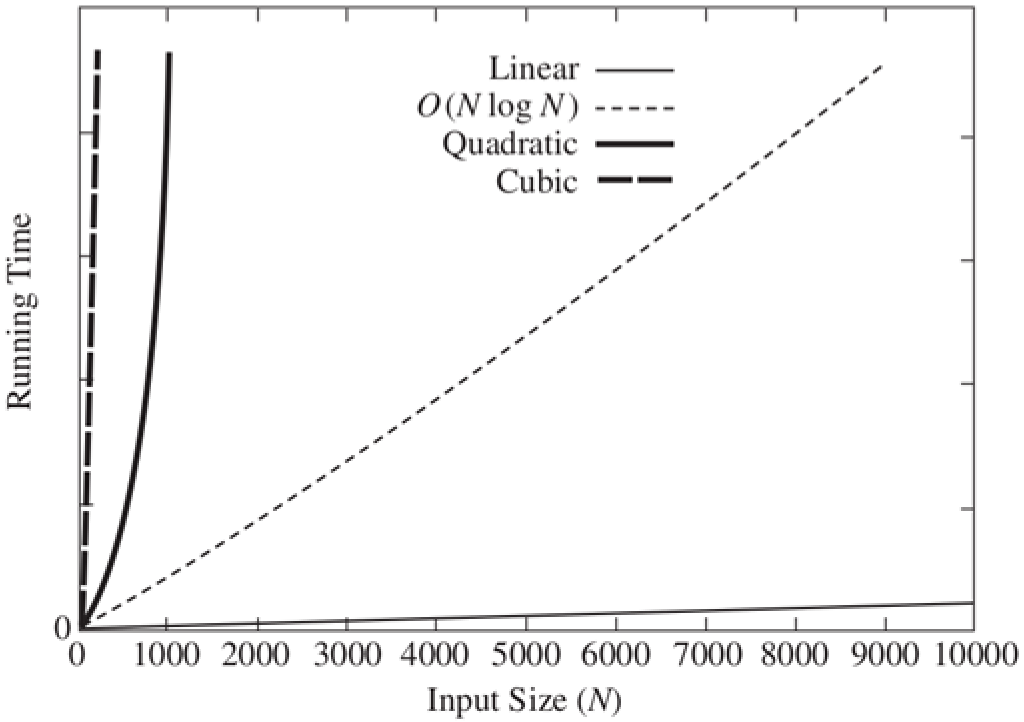
Big-O V.S. wall-clock time

	Algorithm Time			
Input Size	1 $O(N^3)$	2 $O(N^2)$	3 $O(N \log N)$	4 $O(N)$
$N = 100$	0.000159	0.000006	0.000005	0.000002
$N = 1{,}000$	0.095857	0.000371	0.000060	0.000022
$N = 10{,}000$	86.67	0.033322	0.000619	0.000222
$N = 100{,}000$	NA	3.33	0.006700	0.002205
$N = 1{,}000{,}000$	NA	NA	0.074870	0.022711

Running times of several algorithms for maximum subsequence sum (in seconds)

How to compare algorithms?





O(1): constant complexity

- Also called constant time operations
- Execute in a certain amount of time
- Examples include:
  - my_array[50]
  - int my_int = 3;
  - sum = my_int + 5;
  - product = my_int * 50;
  - int foo = new int;
  - if(my_int == 3)

WSU

Big-O: worst-case analysis

- In Big-O analysis, we are interested the maximum number of operations required to complete an algorithm.
- How many operations are required to execute the following code?

O(3)

O(1)

O(2)

```
if(answer == "n")
{
        cout << "Thanks for playing!" << endl;
}
```

What about O(100), O(1,000), O(10,000)?

For any O(X) where:

X = number of constant time operation.

If there exists a constant k such that k*1 >= X, we reduce to O(1)

Therefore, O(3) = O(1)

Big-O analysis: inaccurate

- What are their Big-Os?

```
Segment #1:
    int i = 0;

Segment #2:
    cout << "Hello";
    cout << "Hello";
    cout << "Hello";
```

Big-O analysis: growth rate

- Going back to our list: {3, 9, 1, 2, 3, 5}
- Big-O analysis: always find the last (worst-case)
- Performing a Find(5) is directly affected by the size of the list
- Which Find(5) is faster?
  - {3, 9, 1, 2, 3, 5}
  - {3, 9, 1, 2, 3, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 5}
  - {3, 9, 1, 2, 3, $\underbrace{1, 1, 1, 1, 1, \ldots, 1, 1, 1, 1, 1}_{\text{10000 items}}$, 5}

WSU

How many operations in Find()?

- A: It depends
- Q: Depends on what?
- A: It depends on the number of items in our list
- Q: How do we represent a list whose size can vary.
- A: With a variable!

Time complexity of Find()

- n: the number of elements in the list.
- n determines the number of operations to be executed.
- This relationship (n v.s. # of operations) is linear.
- The growth rate (i.e. Big-O) is also linear.
- We denote a linear relationship with the variable n.
  - → O(n)

Complexity for loops

- FOR loops:
  - for(int i = 0; i < num_items; i++);
- WHILE loops:
  - while(keep_going == 'y');

O(n) typically

Nested loops

- ```
  for(int i = 0; i < num_items; i++) {
      for(int j = 0; j < num_items; j++) {
          swap(items[i], items[j])
      }
  }
  ```
- In this case, we multiply the effect that num_items has on the growth rate, yielding O(n^2)
  ```

WSU

Unrelated loops

```
for(int i = 0; i < num_items; i++) {
        cout << "hello";
}
for(int j = 0; j < num_items; j++) {
        cout << "goodbye";
}
```

- O(n + n) or O(2n) → simplify to O(n)

Reduction of non-constant time

- In Big-O analysis, we always drop coefficients:
  - O(2n) → O(n)
  - O(40n) → O(n)
  - O(1000000n) → O(n)
- This is because Big-O cares about placing algorithms into performance groups, not absolute T(n) calculations

Why dropping constants?

- Adopt the convention that there are no particular units of time
- Only the dominating factor matters for large n values:
  - 1000 n v.s. n^2
  - 1000 n + 1,000,000 v.s. n^2
  - n^3 v.s. n^2 + 30,000
  - n^3 v.s. n^3 + n^2

Asymptotic analysis

- Lower-order terms can generally be ignored for Big-O analysis, and constants thrown away if there is a higher order factor
  - We only care about the growth rate over large n values for Big-O analysis
  - There is plenty of work in the small n space too – example is for n <= 10 in sorting

How to compare algorithms?

Function	Name
c	Constant
log(n)	Logarithmic
log^2(n)	Log-squared
n	Linear
n log(n)	(Will see this in sorting *a lot*)
n^2	Quadratic
n^3	Cubic
2^n	Exponential

WSU

Time complexity: example #1

```
public static int sum( int n ) {
        int partialSum;
        partialSum = 0;
        for( int i = 1; i <= n; i++ )
                partialSum += i * i * i;
        return partialSum;
}
```

- $\sum_{i=1}^{n} i^3$
- Time complexity?

WSU

Time complexity: example #1

```
public static int sum( int n ) {
        int partialSum;                              O(1)
        partialSum = 0;                              O(1)
        for( int i = 1; i <= n; i++ )                O(n)
                partialSum += i * i * i;             O(1)
        return partialSum;                           O(1)
}
```

- $\sum_{i=1}^{n} i^3$
- Time complexity?
  - O(1+1+n*1+1) = O(n)

WSU

Time complexity: example #2

- Search Problem:
  - Given an integer k and an array of integers:
    $A_0 , A_1 , A_2 , A_3 , A_4 \ldots A_{n-1}$
    which are pre-sorted, find i such that $A_i = k$. (Return –1 if k is not in the list.)
- For example, {-32, 2, 3, 9, 45, 1002}:
  Given that k = 9 $\rightarrow$ the program will return ?

WSU

Time complexity: example #2

- Search Problem:
  - Given an integer k and an array of integers:
    $A_0, A_1, A_2, A_3, A_4 \ldots A_{n-1}$
    which are pre-sorted, find i such that $A_i = k$. (Return –1 if k is not in the list.)
- For example, {-32, 2, 3, 9, 45, 1002}:
  - Given that k = 9 → the program will return 3
  - → the number 9 in the 3rd position.
  - Note: always start counting positions from 0, unless otherwise specified

WSU

Sequential search

```
public int bruteForceSearch(int k, int[] array){
        for(int i=0; i<array.length; i++){
                if(a[i] = = k){
                        return i;          /*found it!*/
                }
        }
        return -1;                         /*didn't find, not in array*/
}
// Takes O(N)
```

A typical for-loop

Binary search: an alternative

1. Start in the middle of array.
2. If that is the correct number return.
3. If not, then check if the correct number is larger or smaller than the number in the current position.
4. Take correct half of the array and go to the middle of that one.
5. Repeat.

Binary search: example

- Let's look for k = 54.
- Start in middle of array

  11, 13, 21, 26, 29, 36, 40, 41, 45, 51, 54, 56, 65, 72, 77, 83
- Is 54 bigger than 41? Yes, so look in upper half of array.

  11, 13, 21, 26, 29, 36, 40, 41, 45, 51, 54, 56, 65, 72, 77, 83
- Is 54 bigger than 56? No, so take lower half of remaining array.

  11, 13, 21, 26, 29, 36, 40, 41, 45, 51, 54, 56, 65, 72, 77, 83
- 5) Is 54 bigger than 51? Yes, so take upper half of remaining array.

  11, 13, 21, 26, 29, 36, 40, 41, 45, 51, 54, 56, 65, 72, 77, 83

  6) And 51 is in the 9th position (starting from 0)

Binary search: decrease the size of search by roughly ½

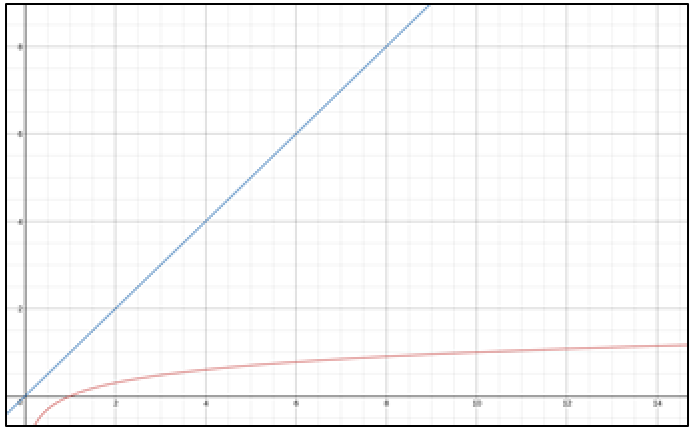
Binary search: example

```
public int binarySearch(int k, int[] array){
    int left = -1;
    int right = array.length;                   //left and right are the array bounds
    while(left+1 ! = right) {                   //stop when left and right meet
        int middle = (left+right)/2;            // find the middle point
        if(k < array[middle]).                  // in left half
            right = middle;                     // new right is the old middle
            if(k == array[middle]).             // found it!
                return middle;                   // new right is the old middle
            if(k > array[middle])               // in right half
                left = middle;                  // new left is the old middle
      }
      return -1;                               // didn't find it. Not in array
}
```

WSU

Binary search: example

- Big-O analysis: the worst-case scenario
- The worst case is that the array size has to be halved until we are down to an array size of 1 (just like the example).
- Example: Once through for size 32, then size 16, 8, 4, 2, 1(stop)
  - How many times through the loop? 5
- Generalization: if the array size is $n = 2^i$
  - The time complexity is O(log(n))
  - Compare with sequential search O(n)
  - Binary search is more efficient!



Log(n) example

```
for(int i = 1; i<n; i *= 37){
        total++;
}
```

- i increases by a factor of 37 each time, so takes log(n) time
- If a loop is halved over and over, it is usually some form of O(log(n))
- Equivalently, if a loop's work jumps by a constant factor each iteration, it is O(log(n))

WSU

Linear complexity

```
for(int i = 0; i<n; i += 2){
     total++;
}
```

- Increases by 2 each time, but not by a multiplicative factor of 2, so not log(n).
- What is the run time?

  i = 0, 2, 4, 6, 8, …
  - This will run for n/2 iterations and the runtime is O(n)
- Conclusion:
  - When a loop increases or decreases by a constant amount each iteration, then its growth rate is O(n).

Simple iterative loop

```
for(int i = 1; i < n; i++){      ---------- O(n)
    for(int j = 1; j < n; j++){ -------     O(n)
        total++; ---------------------------------- O(1)
    }
}
```

- Nested loop:
  - Outer loop goes n times
  - Inner loop goes n times.
- n*n means:

  O(n^2)

WSU

Simple iterative loop

```
for(int i = 1; i < n; i++){----------------- O(n)
    for(int j = 1; j < n; j *= 2){----- O(log(n))
        total++;  ---------------------------------- O(1)
    }
}
```

- Nested loop:
  - Outer loop goes n times.
  - Inner loop goes log(n) times
- So: 1 * log(n) * n

  → O(n log(n))

WSU

Simple iterative loop

```
for(int i = 1; i < n; i++){
    for(int j = 1; j < n; j*=2){
        total++;  }
    for(int k = 1; k < n; k++){
            total++;  }
}
for(int x = 1; x < n; x++) { total++; }
```

WSU

Simple iterative loop

```
for(int i = 1; i < n; i++){ -------------------------- O(n)
    for(int j = 1; j < n; j*=2){ ----------------- O(log(n))
        total++;  } ------------------------------------------ O(1)
    for(int k = 1; k < n; k++){ ------------------- O(n)
            total++;  }-------------------------------- O(1)
}
for(int x = 1; x < n; x++) { total++; }----- O(n)
```

- O(n * (log(n) + n) + n)
- Simplified → O(n log(n) + n^2 + n) → O(n^2)

Maximum subsequence sum

Maximum Subsequence Sum Problem

Given (possibly negative) integers $A_1, A_2, \ldots, A_N$, find the maximum value of $\sum_{k=i}^{j} A_k$.
(For convenience, the maximum subsequence sum is 0 if all the integers are negative.)

Example:

For input -2, 11, -4, 13, -5, -2, the answer is 20 (A_2 through A_4).

```
/**
 * Cubic maximum contiguous subsequence sum algorithm.
 */
int maxSubSum1( const vector<int> & a )
{
    int maxSum = 0;

    for( int i = 0; i < a.size( ); ++i )
        for( int j = i; j < a.size( ); ++j )
        {
            int thisSum = 0;

            for( int k = i; k <= j; ++k )
                thisSum += a[ k ];

            if( thisSum > maxSum )
                maxSum = thisSum;
        }

    return maxSum;
}
```

Figure 2.5 Algorithm 1

```
/**
 * Quadratic maximum contiguous subsequence sum algorithm.
 */
int maxSubSum2( const vector<int> & a )
{
    int maxSum = 0;

    for( int i = 0; i < a.size( ); ++i )
    {
        int thisSum = 0;
        for( int j = i; j < a.size( ); ++j )
        {
            thisSum += a[ j ];

            if( thisSum > maxSum )
                maxSum = thisSum;
        }
    }

    return maxSum;
}
```

Figure 2.6 Algorithm 2

Sec 2.4.3, Page 55

Bubblesort

First Pass:

(**5 1** 4 2 8) –> (**1 5** 4 2 8), Here, algorithm compares the first two elements, and swaps since 5 > 1.
(1 **5 4** 2 8) –> (1 **4 5** 2 8), Swap since 5 > 4
(1 4 **5 2** 8) –> (1 4 **2 5** 8), Swap since 5 > 2
(1 4 2 **5 8**) –> (1 4 2 **5 8**), Now, since these elements are already in order (8 > 5), algorithm does not swap them.

Second Pass:

(**1 4** 2 5 8) –> (**1 4** 2 5 8)
(1 **4 2** 5 8) –> (1 **2 4** 5 8), Swap since 4 > 2
(1 2 **4 5** 8) –> (1 2 **4 5** 8)
(1 2 4 **5 8**) –> (1 2 4 **5 8**)

Now, the array is already sorted, but our algorithm does not know if it is completed. The algorithm needs one **whole** pass without **any** swap to know it is sorted.

Third Pass:

(**1 2** 4 5 8) –> (**1 2** 4 5 8)
(1 **2 4** 5 8) –> (1 **2 4** 5 8)
(1 2 **4 5** 8) –> (1 2 **4 5** 8)
(1 2 4 **5 8**) –> (1 2 4 **5 8**)

Algorithm Analysis

Bubblesort

First Pass:

(**5 1** 4 2 8) –> (**1 5** 4 2 8), Here, algorithm compares the first two elements, and swaps since 5 > 1.
(1 **5 4** 2 8) –> (1 **4 5** 2 8), Swap since 5 > 4
(1 4 **5 2** 8) –> (1 4 **2 5** 8), Swap since 5 > 2
(1 4 2 **5 8**) –> (1 4 2 **5 8**), Now, since these elements are already in order (8 > 5), algorithm does not swap them.

Second Pass:

(**1 4** 2 5 8) –> (**1 4** 2 5 8)
(1 **4 2** 5 8) –> (1 **2 4** 5 8), Swap since 4 > 2
(1 2 **4 5** 8) –> (1 2 **4 5** 8)
(1 2 4 **5 8**) –> (1 2 4 **5 8**)

Now, the array is already sorted, but our algorithm does not know if it is completed. The algorithm needs one **whole** pass without **any** swap to know it is sorted.

Third Pass:

(**1 2** 4 5 8) –> (**1 2** 4 5 8)
(1 **2 4** 5 8) –> (1 **2 4** 5 8)
(1 2 **4 5** 8) –> (1 2 **4 5** 8)
(1 2 4 **5 8**) –> (1 2 4 **5 8**)

WSU

Bubblesort

First Pass:

(**5 1** 4 2 8) –> (**1 5** 4 2 8), Here, algorithm compares the first two elements, and swaps since 5 > 1.

(1 **5 4** 2 8) –> (1 **4 5** 2 8), Swap since 5 > 4

(1 4 **5 2** 8) –> (1 4 **2 5** 8), Swap since 5 > 2

(1 4 2 **5 8**) –> (1 4 2 **5 8**), Now, since these elements are already in order (8 > 5), algorithm does not swap them.

Second Pass:

(**1 4** 2 5 8) –> (**1 4** 2 5 8)

(1 **4 2** 5 8) –> (1 **2 4** 5 8), Swap since 4 > 2

(1 2 **4 5** 8) –> (1 2 **4 5** 8)

(1 2 4 **5 8**) –> (1 2 4 **5 8**)

Now, the array is already sorted, but our algorithm does not know if it is completed. The algorithm needs one **whole** pass without **any** swap to know it is sorted.

Third Pass:

(**1 2** 4 5 8) –> (**1 2** 4 5 8)

(1 **2 4** 5 8) –> (1 **2 4** 5 8)

(1 2 **4 5** 8) –> (1 2 **4 5** 8)

(1 2 4 **5 8**) –> (1 2 4 **5 8**)

Algorithm Analysis

Bubblesort

First Pass:

(**5 1** 4 2 8) –> (**1 5** 4 2 8), Here, algorithm compares the first two elements, and swaps since 5 > 1.

(1 **5 4** 2 8) –> (1 **4 5** 2 8), Swap since 5 > 4

(1 4 **5 2** 8) –> (1 4 **2 5** 8), Swap since 5 > 2

(1 4 2 **5 8**) –> (1 4 2 **5 8**), Now, since these elements are already in order (8 > 5), algorithm does not swap them.

Second Pass:

(**1 4** 2 5 8) –> (**1 4** 2 5 8)

(1 **4 2** 5 8) –> (1 **2 4** 5 8), Swap since 4 > 2

(1 2 **4 5** 8) –> (1 2 **4 5** 8)

(1 2 4 **5 8**) –> (1 2 4 **5 8**)

Now, the array is already sorted, but our algorithm does not know if it is completed. The algorithm needs one **whole** pass without **any** swap to know it is sorted.

Third Pass:

(**1 2** 4 5 8) –> (**1 2** 4 5 8)

(1 **2 4** 5 8) –> (1 **2 4** 5 8)

(1 2 **4 5** 8) –> (1 2 **4 5** 8)

(1 2 4 **5 8**) –> (1 2 4 **5 8**)

Bubblesort

First Pass:

(**5 1** 4 2 8) –> (**1 5** 4 2 8), Here, algorithm compares the first two elements, and swaps since 5 > 1.

(1 **5 4** 2 8) –> (1 **4 5** 2 8), Swap since 5 > 4

(1 4 **5 2** 8) –> (1 4 **2 5** 8), Swap since 5 > 2

(1 4 2 **5 8**) –> (1 4 2 **5 8**), Now, since these elements are already in order (8 > 5), algorithm does not swap them.

Second Pass:

(**1 4** 2 5 8) –> (**1 4** 2 5 8)

(1 **4 2** 5 8) –> (1 **2 4** 5 8), Swap since 4 > 2

(1 2 **4 5** 8) –> (1 2 **4 5** 8)

(1 2 4 **5 8**) –> (1 2 4 **5 8**)

Now, the array is already sorted, but our algorithm does not know if it is completed. The algorithm needs one **whole** pass without **any** swap to know it is sorted.

Third Pass:

(**1 2** 4 5 8) –> (**1 2** 4 5 8)

(1 **2 4** 5 8) –> (1 **2 4** 5 8)

(1 2 **4 5** 8) –> (1 2 **4 5** 8)

(1 2 4 **5 8**) –> (1 2 4 **5 8**)

WSU

Bubblesort

First Pass:

(**5 1** 4 2 8) –> (**1 5** 4 2 8), Here, algorithm compares the first two elements, and swaps since 5 > 1.
(1 **5 4** 2 8) –> (1 **4 5** 2 8), Swap since 5 > 4
(1 4 **5 2** 8) –> (1 4 **2 5** 8), Swap since 5 > 2
(1 4 2 **5 8**) –> (1 4 2 **5 8**), Now, since these elements are already in order (8 > 5), algorithm does not swap them.

Second Pass:

(**1 4** 2 5 8) –> (**1 4** 2 5 8)
(1 **4 2** 5 8) –> (1 **2 4** 5 8), Swap since 4 > 2
(1 2 **4 5** 8) –> (1 2 **4 5** 8)
(1 2 4 **5 8**) –> (1 2 4 **5 8**)
Now, the array is already sorted, but our algorithm does not know if it is completed. The algorithm needs one **whole** pass without **any** swap to know it is sorted.

Third Pass:

(**1 2** 4 5 8) –> (**1 2** 4 5 8)
(1 **2 4** 5 8) –> (1 **2 4** 5 8)
(1 2 **4 5** 8) –> (1 2 **4 5** 8)
(1 2 4 **5 8**) –> (1 2 4 **5 8**)

Bubblesort

First Pass:

(**5 1** 4 2 8) –> (**1 5** 4 2 8), Here, algorithm compares the first two elements, and swaps since 5 > 1.

(1 **5 4** 2 8) –> (1 **4 5** 2 8), Swap since 5 > 4

(1 4 **5 2** 8) –> (1 4 **2 5** 8), Swap since 5 > 2

(1 4 2 **5 8**) –> (1 4 2 **5 8**), Now, since these elements are already in order (8 > 5), algorithm does not swap them.

Second Pass:

(**1 4** 2 5 8) –> (**1 4** 2 5 8)

(1 **4 2** 5 8) –> (1 **2 4** 5 8), Swap since 4 > 2

(1 2 **4 5** 8) –> (1 2 **4 5** 8)

(1 2 4 **5 8**) –> (1 2 4 **5 8**)

Now, the array is already sorted, but our algorithm does not know if it is completed. The algorithm needs one **whole** pass without **any** swap to know it is sorted.

Third Pass:

(**1 2** 4 5 8) –> (**1 2** 4 5 8)

(1 **2 4** 5 8) –> (1 **2 4** 5 8)

(1 2 **4 5** 8) –> (1 2 **4 5** 8)

(1 2 4 **5 8**) –> (1 2 4 **5 8**)

Bubblesort

First Pass:

(**5 1** 4 2 8) –> (**1 5** 4 2 8), Here, algorithm compares the first two elements, and swaps since 5 > 1.

(1 **5 4** 2 8) –> (1 **4 5** 2 8), Swap since 5 > 4

(1 4 **5 2** 8) –> (1 4 **2 5** 8), Swap since 5 > 2

(1 4 2 **5 8**) –> (1 4 2 **5 8**), Now, since these elements are already in order (8 > 5), algorithm does not swap them.

Second Pass:

(**1 4** 2 5 8) –> (**1 4** 2 5 8)

(1 **4 2** 5 8) –> (1 **2 4** 5 8), Swap since 4 > 2

(1 2 **4 5** 8) –> (1 2 **4 5** 8)

(1 2 4 **5 8**) –> (1 2 4 **5 8**)

Now, the array is already sorted, but our algorithm does not know if it is completed. The algorithm needs one **whole** pass without **any** swap to know it is sorted.

Third Pass:

(**1 2** 4 5 8) –> (**1 2** 4 5 8)

(1 **2 4** 5 8) –> (1 **2 4** 5 8)

(1 2 **4 5** 8) –> (1 2 **4 5** 8)

(1 2 4 **5 8**) –> (1 2 4 **5 8**)

WSU

Bubblesort

First Pass:
(**5 1** 4 2 8) –> (**1 5** 4 2 8), Here, algorithm compares the first two elements, and swaps since 5 > 1.
(1 **5 4** 2 8) –> (1 **4 5** 2 8), Swap since 5 > 4
(1 4 **5 2** 8) –> (1 4 **2 5** 8), Swap since 5 > 2
(1 4 2 **5 8**) –> (1 4 2 **5 8**), Now, since these elements are already in order (8 > 5), algorithm does not swap them.

Second Pass:
(**1 4** 2 5 8) –> (**1 4** 2 5 8)
(1 **4 2** 5 8) –> (1 **2 4** 5 8), Swap since 4 > 2
(1 2 **4 5** 8) –> (1 2 **4 5** 8)
(1 2 4 **5 8**) –> (1 2 4 **5 8**)
Now, the array is already sorted, but our algorithm does not know if it is completed. The algorithm needs one **whole** pass without **any** swap to know it is sorted.

Third Pass:
(**1 2** 4 5 8) –> (**1 2** 4 5 8)
(1 **2 4** 5 8) –> (1 **2 4** 5 8)
(1 2 **4 5** 8) –> (1 2 **4 5** 8)
(1 2 4 **5 8**) –> (1 2 4 **5 8**)

Algorithm Analysis

Bubblesort

First Pass:

(**5 1** 4 2 8) –> (**1 5** 4 2 8), Here, algorithm compares the first two elements, and swaps since 5 > 1.
(1 **5 4** 2 8) –> (1 **4 5** 2 8), Swap since 5 > 4
(1 4 **5 2** 8) –> (1 4 **2 5** 8), Swap since 5 > 2
(1 4 2 **5 8**) –> (1 4 2 **5 8**), Now, since these elements are already in order (8 > 5), algorithm does not swap them.

Second Pass:

(**1 4** 2 5 8) –> (**1 4** 2 5 8)
(1 **4 2** 5 8) –> (1 **2 4** 5 8), Swap since 4 > 2
(1 2 **4 5** 8) –> (1 2 **4 5** 8)
(1 2 4 **5 8**) –> (1 2 4 **5 8**)

Now, the array is already sorted, but our algorithm does not know if it is completed. The algorithm needs one **whole** pass without **any** swap to know it is sorted.

Third Pass:

(**1 2** 4 5 8) –> (**1 2** 4 5 8)
(1 **2 4** 5 8) –> (1 **2 4** 5 8)
(1 2 **4 5** 8) –> (1 2 **4 5** 8)
(1 2 4 **5 8**) –> (1 2 4 **5 8**)

Bubblesort

First Pass:
(**5 1** 4 2 8) –> (**1 5** 4 2 8), Here, algorithm compares the first two elements, and swaps since 5 > 1.
(1 **5 4** 2 8) –> (1 **4 5** 2 8), Swap since 5 > 4
(1 4 **5 2** 8) –> (1 4 **2 5** 8), Swap since 5 > 2
(1 4 2 **5 8**) –> (1 4 2 **5 8**), Now, since these elements are already in order (8 > 5), algorithm does not swap them.
Second Pass:
(**1 4** 2 5 8) –> (**1 4** 2 5 8)
(1 **4 2** 5 8) –> (1 **2 4** 5 8), Swap since 4 > 2
(1 2 **4 5** 8) –> (1 2 **4 5** 8)
(1 2 4 **5 8**) –> (1 2 4 **5 8**)
Now, the array is already sorted, but our algorithm does not know if it is completed. The algorithm needs one **whole** pass without **any** swap to know it is sorted.
Third Pass:
(**1 2** 4 5 8) –> (**1 2** 4 5 8)
(1 **2 4** 5 8) –> (1 **2 4** 5 8)
(1 2 **4 5** 8) –> (1 2 **4 5** 8)
(1 2 4 **5 8**) –> (1 2 4 **5 8**)

WSU

Bubblesort

First Pass:

(**5 1** 4 2 8) –> (**1 5** 4 2 8), Here, algorithm compares the first two elements, and swaps since 5 > 1.
(1 **5 4** 2 8) –> (1 **4 5** 2 8), Swap since 5 > 4
(1 4 **5 2** 8) –> (1 4 **2 5** 8), Swap since 5 > 2
(1 4 2 **5 8**) –> (1 4 2 **5 8**), Now, since these elements are already in order (8 > 5), algorithm does not swap them.

Second Pass:

(**1 4** 2 5 8) –> (**1 4** 2 5 8)
(1 **4 2** 5 8) –> (1 **2 4** 5 8), Swap since 4 > 2
(1 2 **4 5** 8) –> (1 2 **4 5** 8)
(1 2 4 **5 8**) –> (1 2 4 **5 8**)

Now, the array is already sorted, but our algorithm does not know if it is completed. The algorithm needs one **whole** pass without **any** swap to know it is sorted.

Third Pass:

(**1 2** 4 5 8) –> (**1 2** 4 5 8)
(1 **2 4** 5 8) –> (1 **2 4** 5 8)
(1 2 **4 5** 8) –> (1 2 **4 5** 8)
(1 2 4 **5 8**) –> (1 2 4 **5 8**)

Bubblesort

First Pass:

(**5 1** 4 2 8) –> (**1 5** 4 2 8), Here, algorithm compares the first two elements, and swaps since 5 > 1.
(1 **5 4** 2 8) –> (1 **4 5** 2 8), Swap since 5 > 4
(1 4 **5 2** 8) –> (1 4 **2 5** 8), Swap since 5 > 2
(1 4 2 **5 8**) –> (1 4 2 **5 8**), Now, since these elements are already in order (8 > 5), algorithm does not swap them.

Second Pass:

(**1 4** 2 5 8) –> (**1 4** 2 5 8)
(1 **4 2** 5 8) –> (1 **2 4** 5 8), Swap since 4 > 2
(1 2 **4 5** 8) –> (1 2 **4 5** 8)
(1 2 4 **5 8**) –> (1 2 4 **5 8**)

Now, the array is already sorted, but our algorithm does not know if it is completed. The algorithm needs one **whole** pass without **any** swap to know it is sorted.

Third Pass:

(**1 2** 4 5 8) –> (**1 2** 4 5 8)
(1 **2 4** 5 8) –> (1 **2 4** 5 8)
(1 2 **4 5** 8) –> (1 2 **4 5** 8)
(1 2 4 **5 8**) –> (1 2 4 **5 8**)

WSU

Bubblesort

```
bubblesort(vector<int> & list) {
  i, j, temp;
  for(i=1; i < list.size(); ++i) {
    for(j=0; j < (list.size()-i); ++j) {
      if(list[j] > list[j+1]) {
        temp = list[j];
        list[j] = list[j+1];
        list[j+1] = temp;
      }
    }
  }
}
```

- Time complexity?
  - $\sum_{i=1}^{n} n-i = \sum_{i=0}^{n-1} i = \frac{(n-1)n}{2} = \frac{n^2-n}{2}$ → Θ(n^2)

If statement rule

- if(condition)
  S1
  Else
  S2
- Use the larger complexity of S1 or S2
- If you know the ratio of the two you could do a deeper analysis for a tighter bound, but the default is to just take the larger branch cost

Complexity of recursive calls

- ```
  sample(k){
    if k < 2
        return 0
    return 1 + sample(k/2)
  }
  ```
- How much does each call change?
- What is the time complexity of this algorithm?
- What if it was sample(k/3)?
  ```

WSU

Measure the execution time

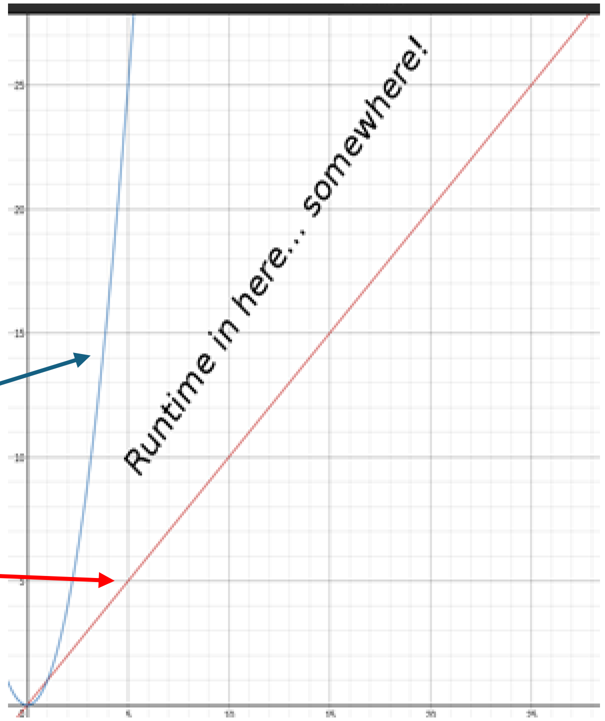
- Linux (Unix) has a “time” command to time how long it takes to run a process: **`time [program with options]`**
- Can be used in assignments to clock program execution
  - Real: Wallclock time from start to end of program
  - User: Actual processing time of program
  - Sys: Kernel processing time for the program

```
(base) yanyan@Yans-Air-2 quicksort_vs_mergesort % time ./build/Merge
sort_vs_Quicksort
Enter the size of the array: 100000
Mergesort time difference = 92836750[ns]
Quicksort time difference = 20720194500[ns]
./build/Mergesort_vs_Quicksort  20.79s user 0.03s system 89% cpu 23.
188 total
(base) yanyan@Yans-Air-2 quicksort_vs_mergesort %
```

WSU

Why time varies?

- Definitely varies because other programs running
- User time varies because of input variability
  - Input can make algorithms vary radically!
    - Bubblesort goes from Ω(n) to O(n^2)
- Sys varies if kernel needs to do extra bookkeeping while your program runs
  - Memory management, I/O operations, definitely if networking overhead



How to compare algorithms?

- Compare time requirements
  - How much time will it take to execute the algorithm?
- Compare space requirements
  - How much extra space is required for this algorithm to execute?

What to analyze in an algorithm?

- Options include:
  - T_ave(n)
  - T_worst(n)
  - T_optimal(n)
- T_optimal(n) <= T_ave(n) <= T_worst(n)
- Do implementation details matter for algorithms analysis?
  - No, implementation isn’t about algorithm analysis
  - Actual running time: copying big arrays v.s. pass by reference example
  - So: it matters in the real world when you code

Summary

- Big-O is the asymptotic run time for an algorithm
  - once n gets “big enough”, which is defined as n > n_o
- All lower order runtime elements in the analysis are dropped for a large n
- Halving work each time gets O(log(n))
- Increasing in a linear fashion gets you O(n)