CS 590: Algorithms

Lecture 02 - Foundations I:
Growth of Functions / Insertion sort/Merge sort
Fall 2023

Reading Assignment: Chapters 2, 3
Next Week: Chapters 4

Outline of Lecture:

- 3.1. Analyzing Algorithms
- 3.2. Growth of Function
- 3.3. Insertion Sort
- 3.4. Designing algorithms – Merge Sort
- 3.5. C++ and Homework Discussion

3.1. Analyzing Algorithms

3.1. Analyzing Algorithms

Analyzing Algorithms is

- to ***predict*** the resources that the algorithm requires.
- is characterized by common resources such as memory, communication bandwidth, computer hardware, etc.
- to measure the **running time**.

We need a ***computational model*** to predict the running time.

- There are technology-dependent models.
- **Random Access Machine (RAM)**
  - Arithmetically **sums** the number of steps needed to execute the algorithm on a data set.
  - **No concurrent operations** – assumes all instructions are executed one after another.

3.1. Analyzing Algorithms

Issues on RAM:

- **Precision of characterization is a main concern:**
  - It is difficult and tedious to define every instruction and its associated time costs.
  - For example, how can we define the cost if the instruction says to sort?

3.1. Analyzing Algorithms

RAM: Solutions to Issues

1. We stay with instructions commonly found in real computers:
   - **Arithmetic**: add, subtract, multiply, divide, remainder, floor, ceiling, shift left, or shift right.
   - **Data movement**: load, store, copy.
   - **Control**: conditional/unconditional branch, subroutine call, and return.
2. We use integer and floating-point types.
3. We do not model the memory hierarchy (no caches or virtual memory).
   - They are significant in real programs on real machines.
   - They are relatively more complex than RAM.

3.1. Analyzing Algorithms

RAM:

- The goal is to identify the most efficient algorithm (shortest running time) among algorithms that solve the same problem.
  - Algorithms may differ by methodology.
  - They may have different orders of operations.
  - They may require some specific conditions (or assumptions) on input and output.

3.1. Analyzing Algorithms

Why Input size?

- It is the most commonly realistic independent variable in analysis than any other resource in algorithms.
  - Operations depend on the size of the input.
  - But the size of the input is not dependent on the operations.

What is running time?

- It is the number of machine-independent primitive operations (steps) executed.
- A constant operational cost c_i will be assigned to individual operations.
- The running time for each operation is the product of cost and the number of executions.

3.2. Growth of Functions

3.2. Growth of Functions

Limitation on algorithm's efficient running time expression:

- It is not easy to measure the numerical running time without running an actual program.
- It is not easy to describe or forecast the actual real-time behavior of algorithms.
- At the same time, we do not want to analyze the algorithms too roughly.
- We want to compare algorithms to other relative performances of alternative algorithms.

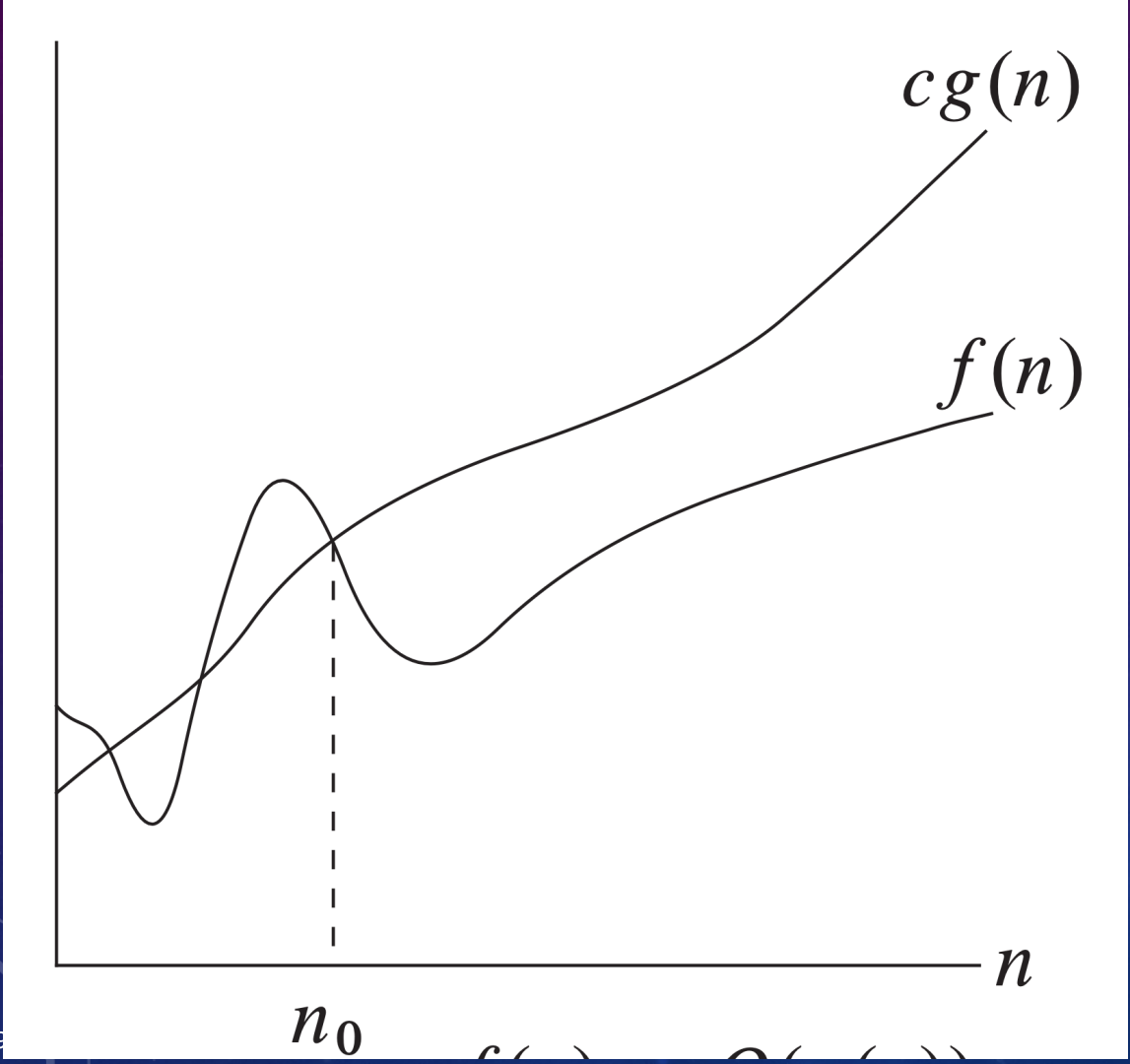
3.2. Growth of Functions

Overcome:

- We need mathematical tools to express the running time behavior with some boundary conditions asymptotically.
- So, the expression is
  - not subjective
  - not complicate
  - not broad

3.2. Growth of Functions

Asymptotic Notation: O –notation



- If $f(n) \in cg(n)$ for a some constant $c > 0$,
  - $0 \leq f(n) \leq cg(n)\ \forall\ n \geq n_0$
  - $f(n) = O(g(n))$
  - $g(n)$ is an asymptotic **upper bound** for $f(n)$ when $n \geq n_0$.

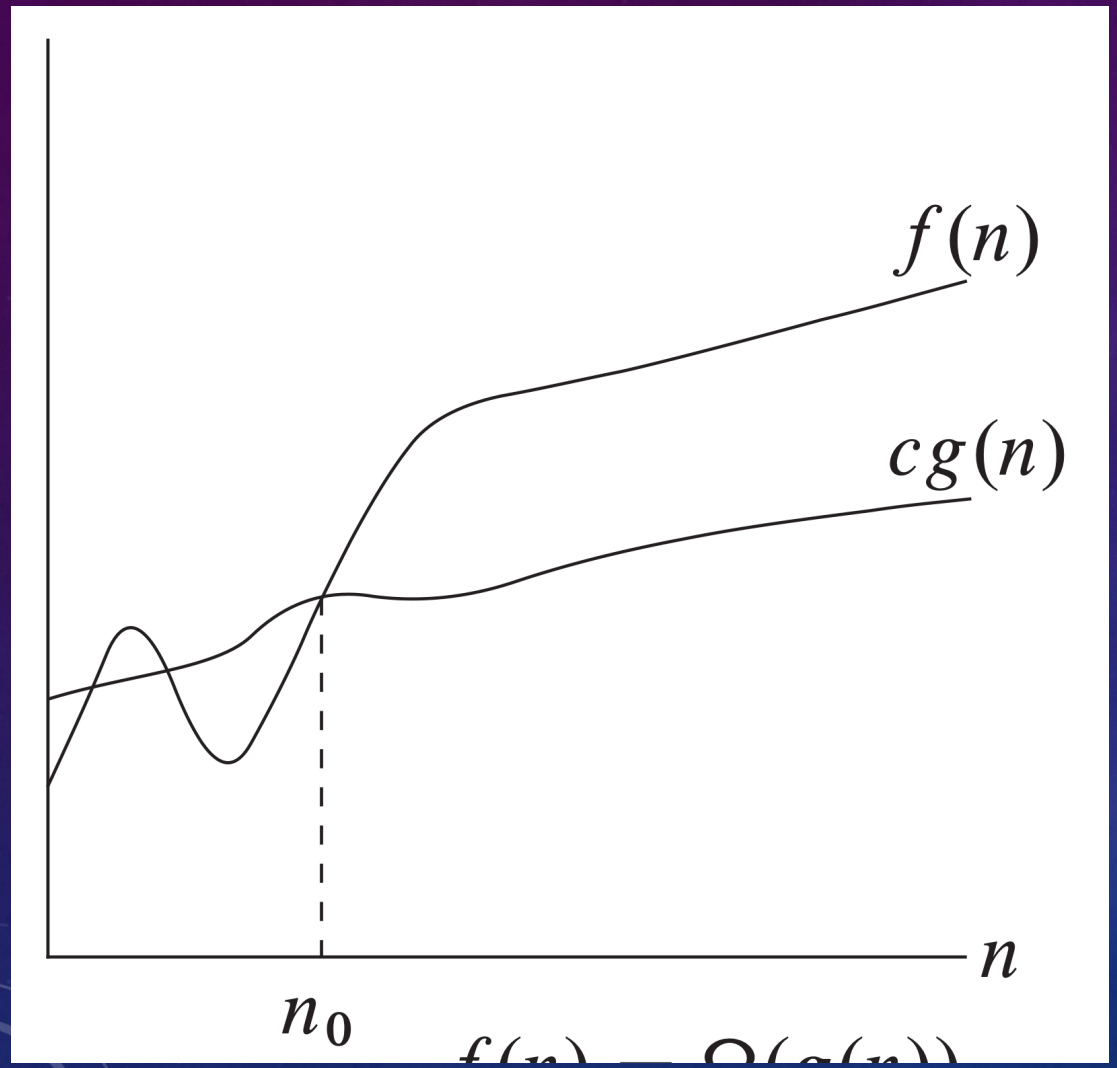
3.2. Growth of Functions

Asymptotic Notation: O −notation

- $2n^2 = O(n^2)$?
  - Let $f(n) = 2n^2$ and $g(n) = cn^2$.
  - $g(n) \geq f(n)$ if $n \geq 1$ and $c \geq 2$. Otherwise, $f(n) > g(n)$.
  - The c condition is easy to identify that satisfies the boundary condition.
  - If $c < 2$ (e.g., 0.5), $f(n) > g(n) \forall\, n > 0$.
    - e.g., $2(3^2) > 0.5(3^2)$
- Any polynomial function, $f(n)$, can be described as $O\big(g(n)\big)$ where $g(n)$ is a function of n with the highest power.
  - $n^2, n^2 + n, n^2 + 1000n$, etc… are all $\in O(n^2)$.

3.2. Growth of Functions

Asymptotic Notation: Ω −notation



- If $f(n) \in cg(n)$ with a constant $c > 0$ and $n \geq n_0$, then $f(n) = \Omega(g(n))$.
- $0 \leq cg(n) \leq f(n)\ \forall n \geq n_0$
- $g(n)$ is an asymptotic **lower bound** for $f(n)$.

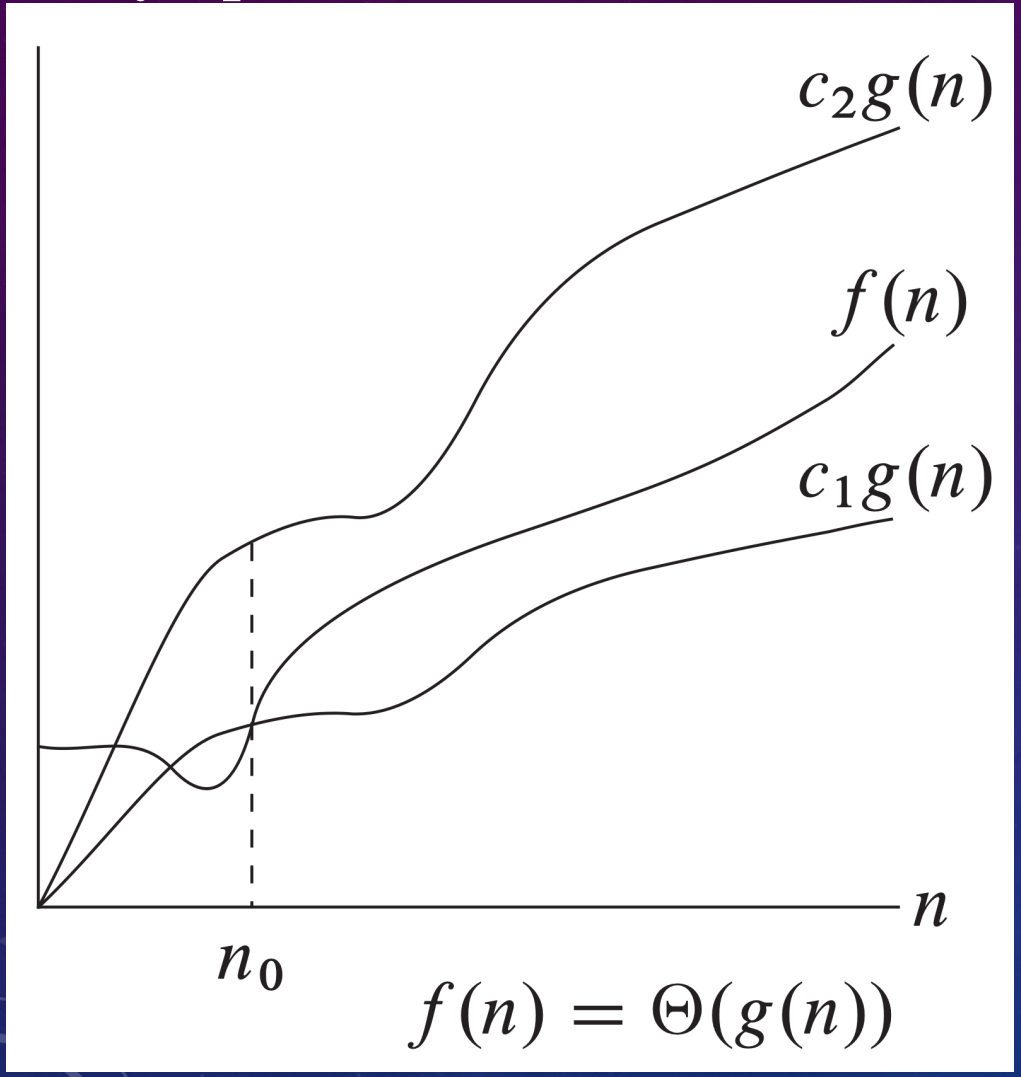
3.2. Growth of Functions

Asymptotic Notation: Ω −notation

- $\sqrt{n} = \Omega(\lg n)$ with $c = 1$ & $n_0 = 16$?
  - Let $f(n) = \sqrt{n}$ and $g(n) = \lg n$.
  - $g(n) \leq f(n)$ for $n \geq 16$. Otherwise, $g(n) > f(n)$.
  - For example,
    - If $n = 64$, $\sqrt{64} > \lg(64)$.
    - If $n = 16$, $\sqrt{16} = \lg(16)$.
    - If $n = 4$, $\sqrt{4} = \lg(4)$.
    - If $n = 2$, $\sqrt{2} > \lg(2)$.
- For any polynomials, $f(n) \geq cg(n)\ \forall n \geq n_0$ and $c > 0$.
  - $n^2, n^2 + n, n^2 - n, n^2 + 1000n, n^2 - 1000n$, etc… are all $\in \Omega(n^2)$.

3.2. Growth of Functions

Asymptotic Notation: Θ −notation



- $g(n)$ is an **asymptotic tight bound** for $f(n)$ if and only if $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$.
- Then, $f(n) = \Theta\big(g(n)\big)$
- For some constants, $c_1, c_2 > 0$,
- $0 < c_1 g(n) \leq f(n) \leq c_2 g(n)\ \forall\ n \geq n_0$.

3.2. Growth of Functions

Asymptotic Notation: Θ −notation

- $\frac{n^2}{2} - 2n = \Theta(n^2)$ with $c_1 = \frac{1}{4}, c_2 = \frac{1}{2}, \& \, n_0 = 8.$
  - $f(n) = \frac{n^2}{2} - 2n$ and $g(n) = n^2.$
  - For $n = 10, \frac{1}{4}(10^2) < \frac{10^2}{2} - 2(10) < \frac{1}{2}(10^2).$
  - For $n = 8, \frac{1}{4}(8^2) = \frac{8^2}{2} - 2(8) < \frac{8^2}{2}.$
  - For $n = 4, \frac{4^2}{2} - 2(4) < \frac{4^2}{4} < \frac{4^2}{2}.$

- For all polynomials $f(n) = O\big(g(n)\big)\ and = \Omega\big(g(n)\big), f(n) = \Theta\big(g(n)\big).$
- $n^2, n^2 + n, n^2 - n, n^2 + 1000n, n^2 - 1000n,$ etc… are all $\in O(n^2), \Omega(n^2)$
  - $\therefore f(n) = \Theta(n^2).$

3.2. Growth of Functions

Chain rule: Asymptotic notations in equations:

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

Interpretation:

1. There exists $f(n) \in \Theta(n)$ s.t. $2n^2 + 3n + 1 = 2n^2 + f(n)$.
2. For all $g(n) \in \Theta(n)$, there exists $h(n) \in \Theta(n^2)$ s.t. $2n^2 + g(n) = h(n)$.

3.2. Growth of Functions

Asymptotic Notation: o −notation

- If $f(n) < cg(n)$ for $c, n_0 > 0$ and $\forall n \geq n_0$, then $f(n) = o(g(n))$.
- $g(n)$ is an **upper bound that is not asymptotically tight** for $f(n)$.
- $f(n)$ becomes insignificant relative to $g(n)$ as n approaches infinity:
  - $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$
- Consider $f(n) = 2n$.
  - $f(n)$ is tightly and asymptotically bounded to $O(n)$ but not to $o(n^2)$.
  - But we still can say $2n = o(n^2) = O(n)$ if the ratio limits
  - $\lim_{n\to\infty} \frac{2n}{n^2} = 0$
  - Note: $2n^2 \neq o(n^2)$ but $= o(n^3) = O(n^2)$.

3.2. Growth of Functions

Asymptotic Notation: ω −notation

- If $f(n) > cg(n) \geq 0$ for $c, n_0 > 0$ and $\forall n \geq n_0$, then $f(n) = \omega\big(g(n)\big)$.
- $g(n)$ is a **lower bound that is not asymptotically tight** for $f(n)$.
- $f(n) \in \omega\big(g(n)\big)$ if and only if $g(n) \in o\big(f(n)\big)$.
- $f(n)$ becomes arbitrarily large relative to $g(n)$ as n approaches infinity:
  - $\lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty$
- Consider $f(n) = \frac{n^2}{2}$,
  - $\frac{n^2}{2} > cn \Rightarrow f(n) = \omega(n) = \Omega(n^2)$.
  - But $f(n) \neq \omega(n^2)$.

3.2. Growth of Functions

The analogy between asymptotic comparison of two functions f and g and the comparison of two real numbers a and b:

$f(n) = O(g(n))$	$a \leq b$
$f(n) = \Omega(g(n))$	$a \geq b$
$f(n) = \Theta(g(n))$	$a = b$
$f(n) = o(g(n))$	$a < b$
$f(n) = \omega(g(n))$	$a > b$

- $f(n)$ is asymptotically **smaller** than $g(n)$ if $f(n) = o(g(n))$.
- $f(n)$ is asymptotically **larger** than $g(n)$ if $f(n) = \omega(g(n))$.

3.2. Growth of Functions

Transitivity	$f(n) = \Theta(g(n))$ and $g(n) = \Theta(h(n))$ imply $f(n) = \Theta(h(n))$ (same for $O, \Omega, o,$ & ω)
Reflexivity	$f(n) = \Theta(g(n)), f(n) = O(g(n)), f(n) = \Omega(g(n))$. (not for o & ω)
Symmetry	$f(n) = \Theta(g(n))$ if and only if $g(n) = \Theta(f(n))$.
Transpose symmetry	$f(n) = O(g(n))$ if and only if $g(n) = \Omega(f(n))$. $f(n) = o(g(n))$ if and only if $g(n) = \omega(f(n))$.

3.2. Growth of Functions

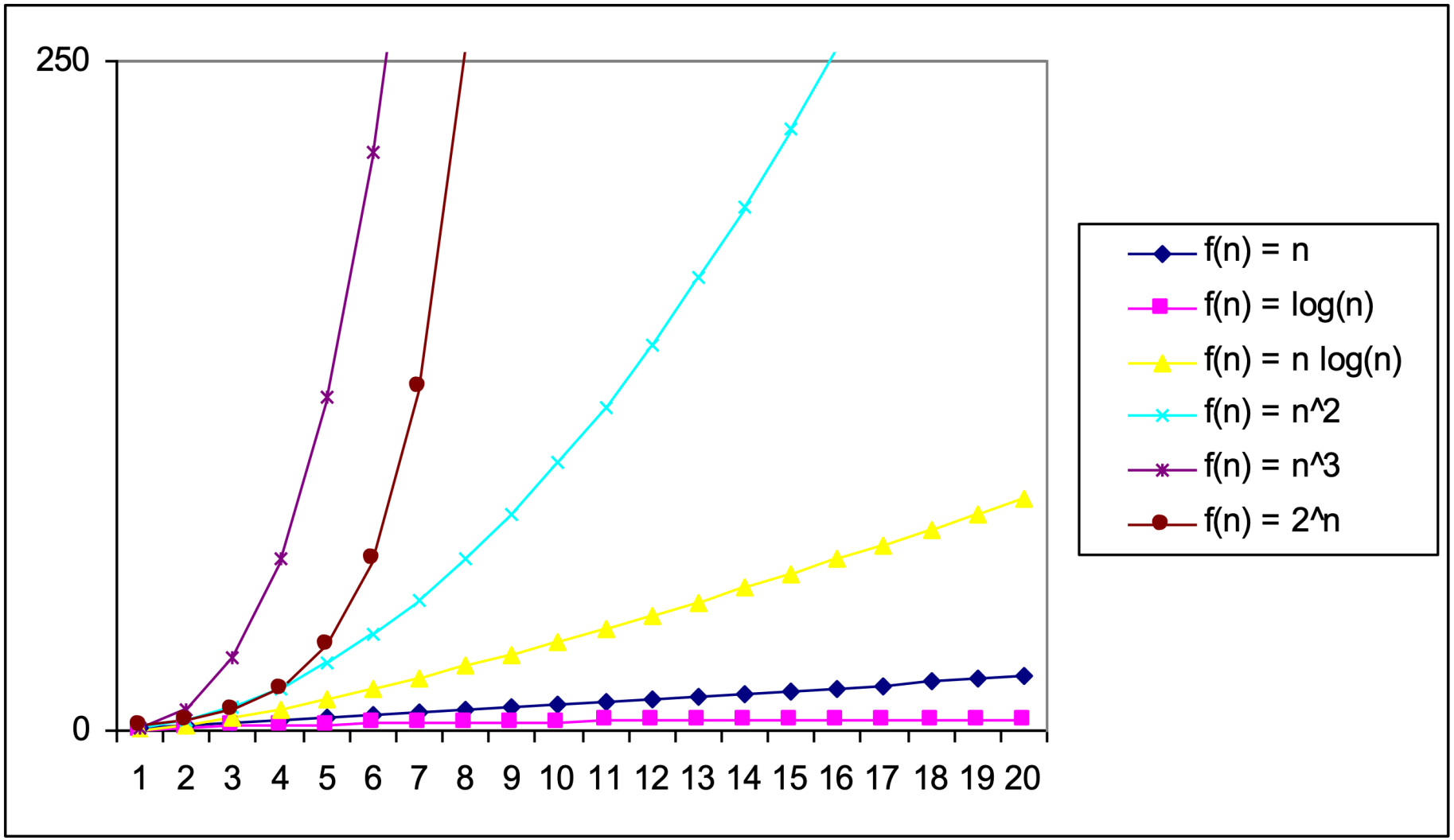
Monotonicity:

- $f(n)$ is **monotonically** increasing if $m \leq n$ implies that $f(m) \leq f(n)$.
- $f(n)$ is **monotonically** decreasing if $m \leq n$ implies that $f(m) \geq f(n)$.
- $f(n)$ is **strictly** increasing if $m < n$ implies that $f(m) < f(n)$.
- $f(n)$ is **strictly** decreasing if $m < n$ implies that $f(m) > f(n)$.

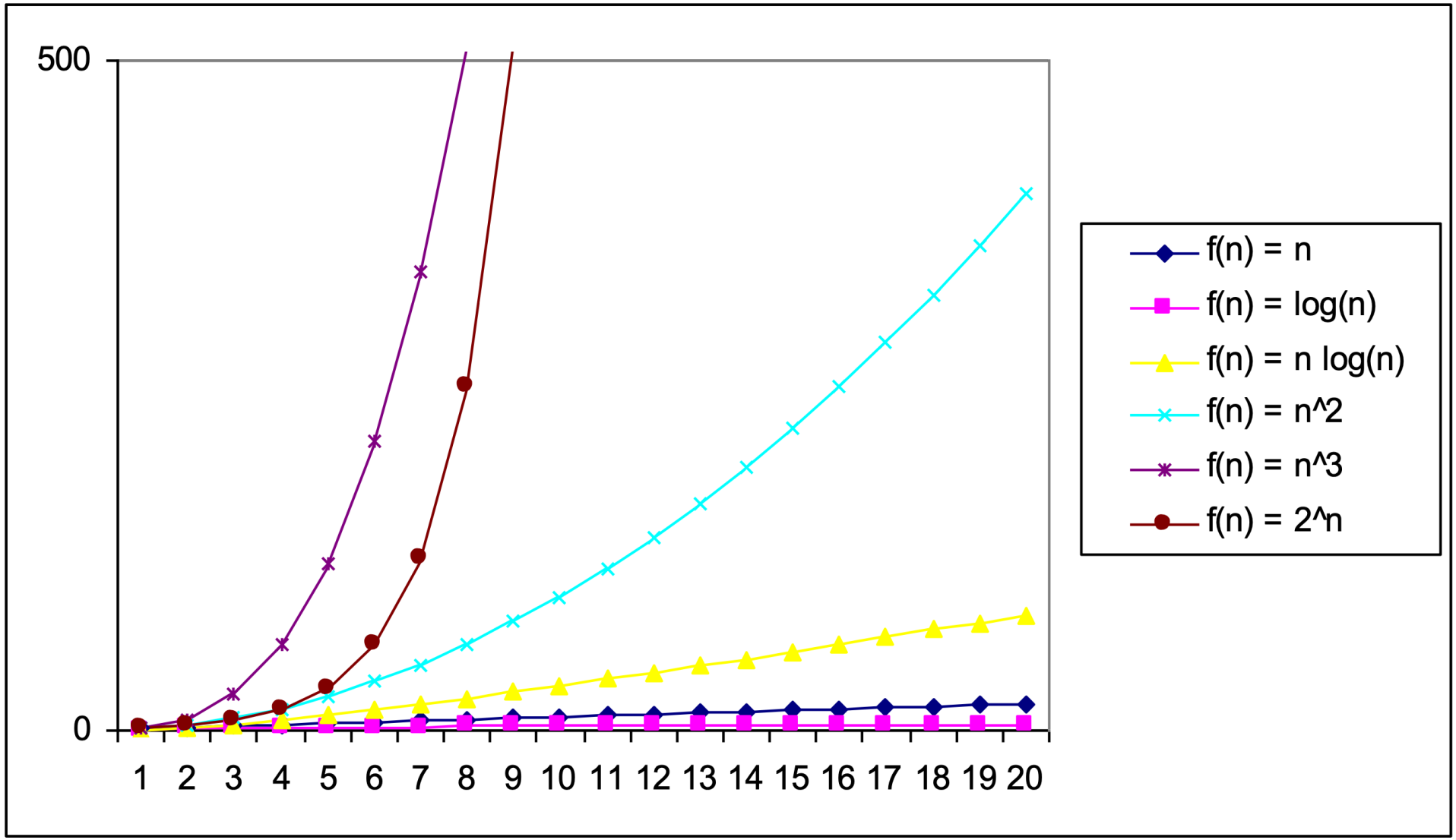
Floors and Ceilings:

- For any real number x, we denote **the greatest integer less than or equal** to by $\lfloor x \rfloor$.
  - We read "**the floor of x**". E.g., $\lfloor 2.3 \rfloor = 2$.
- For any real number x, we denote **the least integer greater than or equal** to by $\lceil x \rceil$.
  - We read "**the ceiling of x**". E.g., $\lceil 2.3 \rceil = 3$.
- $x - 1 < \lfloor x \rfloor \leq x \leq \lceil x \rceil < x + 1$
- The floor function $f(x) = \lfloor x \rfloor$ is monotonically increasing, as is the ceiling function $f(x) = \lceil x \rceil$.

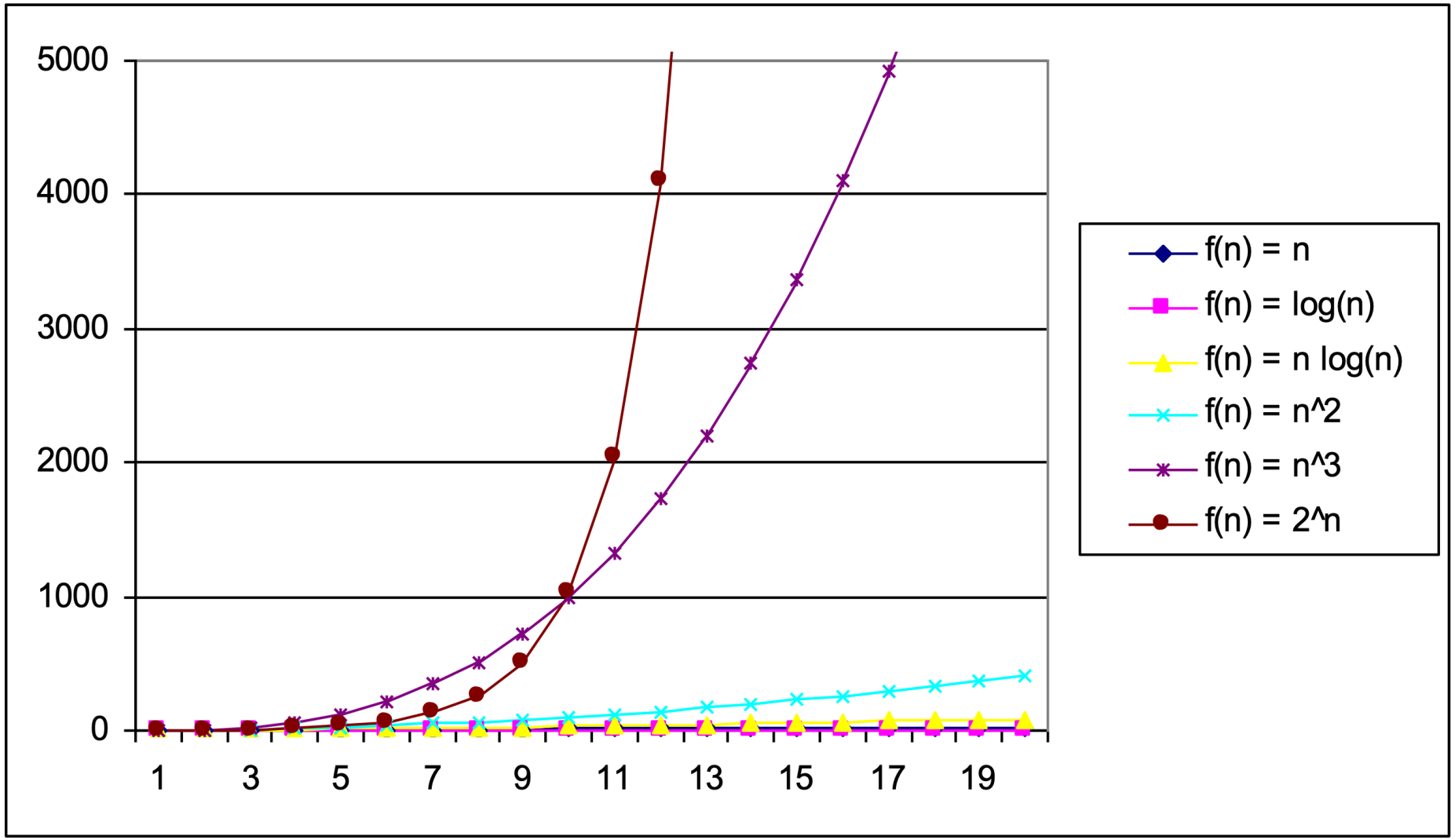
3.2. Growth of Functions



3.2. Growth of Functions



3.2. Growth of Functions



3.2. Growth of Functions

- Order of growth gives a simple characterization of the algorithm's efficiency.
- Allows us to compare the relative performance of alternative algorithms.
- For large enough n, the running time $\Theta(n \lg n)$ beats $\Theta(n^2)$: merge sort beats insertion sort for large enough n. (We will see this later)
- Asymptotic efficiency of algorithms ⇒ input size large enough s.t. order of growth is running time relevant.
- Asymptotically more efficient algorithm usually the best choice for all n (except for small n).
- Used in analysis of algorithms.

3.3. Insertion Sort

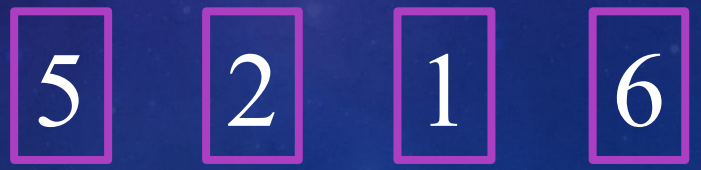
3.3. Insertion Sort

- Input: A sequence of n numbers $\langle a_1, a_2, \ldots, a_n \rangle$
- Output: A permutation (reordering) $\langle a'_1, a'_2, \ldots, a'_n \rangle$ of the input sequence such that
  - $a'_1 \leq a'_2 \leq \cdots \leq a'_n$.

Definition:

- **Keys** – the numbers that we wish to sort

In this problem, even though we are sorting a sequence, the input comes to us in the form of an array with n elements.



3.3. Insertion Sort

- Suppose the length of input array A is known.
- A is not a zero-index array.

InsertionSort(A)

```
 1  for (2 ≤ j ≤ n) do
 2      key = A[j]
 3      //Insert A[j] into the sorted sequence A[1,...,j-1].
 4      i = j- 1
 5      while (i > 0 and A[i] > key) do
 6          A[i+1] = A[i]
 7          i = i - 1
 8          od
 9      A[i+1] = key
10      od
```

3.3. Insertion Sort

5 2 1 6

for loop	while loop	Description
1		1: j = 2 2: j = 2, key = A[2] = 2 4: i = 1 5: j = 2, key = 2, i = 1, and A[1] = 5, 5 > 2
	1	6: j = 2, key = 2, A[2] = A[1] => A[2]=5 7: j = 2, key = 2, i = 0 5: i < 0, while loop terminates
		8: j =2, i = 0, key = 2, A[1]=2

2 5 1 6

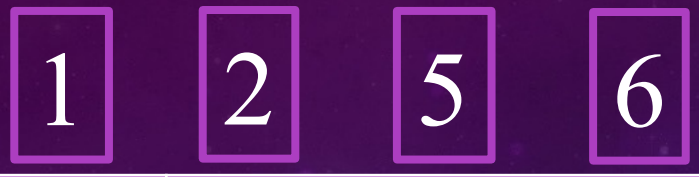
```
1  for (2 ≤ j ≤ n) do
2      key = A[j]
3      //Insert A[j] into …
4      i = j- 1
5      while (i > 0 and A[i] > key) do
6          A[i+1] = A[i]
7          i = i - 1
8          od
9      A[i+1] = key
10     od
```

3.3. Insertion Sort


```
1  for (2 ≤ j ≤ n) do
2      key = A[j]
3      //Insert A[j] into …
4      i = j- 1
5      while (i > 0 and A[i] > key) do
6          A[i+1] = A[i]
7          i = i - 1
8          od
9      A[i+1] = key
10     od
```

for loop	while loop	Description
2		1: j = 3, key = 2, i = 0, A[1] = 2 2: j = 3, key = A[3] = 1 4: i = 2 5: j = 3, key = 1, i = 2, A[2] = 5 > 1
	1	6: j = 3, key = 1, i = 2, A[3] = A[2]=5 7: j =3, key = 1, A[2] = ?, A[3]=5, i = 1, A[1]=2 5: j = 3, key = 1, A[1] = 2 > 1
	2	6: j = 3, key = 1, i = 1, A[2] = A[1]=2 7: j =3, key = 1, A[2] = 2, A[1]=?, i = 0 5: j = 3, key = 1, i = 0, Fail the while-loop condition – while loop terminates
		8: j =3, i = 1, key = 1, A[1]=1

3.3. Insertion Sort



for loop	while loop	Description
3		1: j =4, key = 1, i = 1, A[1] = 1 2: j =4, key = 6 4: i = 3 5: j =4, key = 6, i = 3, A[3] < 6 while loop terminates
		8: j =4, i = 3, A[4] = 6, key = 6


```
1  for (2 ≤ j ≤ n) do
2       key = A[j]
3       //Insert A[j] into …
4       i = j- 1
5       while (i > 0 and A[i] > key) do
6          A[i+1] = A[i]
7          i = i - 1
8          od
9       A[i+1] = key
10      od
```

3.3. Insertion Sort

We often use a **loop invariant** to help us understand why/how an algorithm gives the correct answer when the algorithm has one or more loops.

To prove correctness, we must show three things about loop invariant:

- **Initialization** – It must be true before the first iteration of the loop starts.
- **Maintenance** – If it were true before an iteration of the loop, it remains true until the next iteration starts.
- **Termination** – When the loop terminates, the invariant gives us a useful property that helps show that the algorithm is correct.

Insertion Sort Loop Invariant:

- At the start of each iteration of the "outer" for loop – the loop indexed by j.
- The subarray $A[1, \ldots, j-1]$ consists of the elements originally in $A[1, \ldots, j-1]$ but in sorted order.

3.3. Insertion Sort

Initialization:

- Show the loop invariant holds before the 1st loop: $j = 2$
- The subarray $A[1, \cdots, j-1]$ consists of the single element $A[1]$, which is the original element in $A[1]$.
- This subarray is sorted, showing the loop invariant holds prior to the first iteration of the loop.

```
 1  for (2 ≤ j ≤ n) do
 2      key = A[j]
 3      //Insert A[j] into …
 4      i = j- 1
 5      while (i > 0 and A[i] > key) do
 6          A[i+1] = A[i]
 7          i = i - 1
 8          od
 9      A[i+1] = key
10      od
```

3.3. Insertion Sort

Maintenance:

- Each iteration maintains the loop invariant – for loop works by moving $A[j-1], A[j-2], A[j-3]$, ... by one position to the right until it finds the proper position for $A[j]$ at the point it inserts the value of $A[j]$
- The subarray consists of in sorted order.
- Increment j for the next iteration of the for loop preserves the invariant.

```
 1  for (2 ≤ j ≤ n) do
 2      key = A[j]
 3      //Insert A[j] into …
 4      i = j- 1
 5      while (i > 0 and A[i] > key) do
 6          A[i+1] = A[i]
 7          i = i - 1
 8          od
 9      A[i+1] = key
10      od
```

3.3. Insertion Sort

Termination:

- The for loop terminates when $j = n + 1$.
- We have subarray $A[1, \cdots, n]$ in sorted order.

```
1  for (2 ≤ j ≤ n) do
2      key = A[j]
3      //Insert A[j] into …
4      i = j- 1
5      while (i > 0 and A[i] > key) do
6          A[i+1] = A[i]
7          i = i - 1
8          od
9      A[i+1] = key
10     od
```

3.3. Insertion Sort

```
1  for (2 ≤ j ≤ n) do
2      key = A[j]
3      //Insert A[j] into …
4      i = j- 1
5      while (i > 0 and A[i] > key) do
6          A[i+1] = A[i]
7          i = i - 1
8          od
9      A[i+1] = key
10     od
```

Line	Cost (C_k)	Time T_j
1	c_1	n
2	c_2	$n-1$
4	c_4	$n-1$
5	c_5	$\sum_{j=2}^{n} t_j$
6	c_6	$\sum_{j=2}^{n}(t_j - 1)$
7	c_7	$\sum_{j=2}^{n}(t_j - 1)$
8	c_8	$n-1$

- Suppose the k^{th} line pays **constant** cost c_k for its instruction.
- If the instruction is inside the loop, consider the repeating times T_k.
- Let t_k be the number of times the while loop test is executed for that value of j.
- Note that when a *for* or *while* loop exists in the usual way, **the test is executed one time more than the loop body.**

3.3. Insertion Sort

The running time $T(n)$ of the algorithm is $\sum(\textbf{cost}) \cdot (\#\ \textbf{of times is executed})$:

$$T(n) = \sum_{k=1}^{K} C_k \cdot T_k(n)$$

$$T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5 \sum_{j=2}^{n} t_j + c_6 \sum_{j=2}^{n} (t_j - 1) + c_7 \sum_{j=2}^{n} (t_j - 1) + c_8(n-1)$$

- Running time depends on the **size of the input,** n**,** and t_j.
- The value of t_j depends on the size and the input itself.
- For the completion, we are going to look into three cases:
  - Best case
  - Worst case
  - Average case

3.3. Insertion Sort

If the array is already sorted (**the best case**), n is the same, and $t_j = 1$,

$$T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5 \sum_{j=2}^{n} t_j + c_6 \sum_{j=2}^{n} (t_j - 1) + c_7 \sum_{j=2}^{n} (t_j - 1) + c_8(n-1)$$

$$\sum_{j=2}^{n} t_j = (n-1) \quad \sum_{j=2}^{n} (t_j - 1) = 0$$

$$= c_1 n + c_2(n-1) + c_4(n-1) + c_5(n-1) + c_8(n-1)$$

$$= (c_1 + c_2 + c_4 + c_5 + c_8)n + (c_2 + c_4 + c_5 + c_8)$$

- $T(n)$ is a **linear function** of n.
- $T(n) = \Theta(n)$

3.3. Insertion Sort

If an array is **reverse sorted order (worst-case)**:

- Always find that $\boldsymbol{A[i] > key}$ in the while loop test (L5).
- It compares the key with all elements to the left of the j^{th} position:
  - $A[j] > A[k] \; \forall \; j > k$
- Since the while loop exists because i reaches 0, there is one additional test after the $(j-1)^{th}$ tests. Therefore, $t_j = j$.
  - Then, the iteration times at L6 and L7 are $\sum_{j=2}^{n} t_j = \sum_{j=2}^{n} j$ and $\sum_{j=2}^{n}(t_j - 1) = \sum_{j=2}^{n}(j-1)$, respectively.
- Using $\sum_{j=1}^{n} j = \frac{n(n+1)}{2}$,

$$\sum_{j=2}^{n} j = \left(\sum_{j=1}^{n} j\right) - 1 = \frac{n(n+1)}{2} - 1.$$

$$\sum_{j=2}^{n} (j-1) = \left(\sum_{j=1}^{n} (j-1)\right) - 1 = \frac{n(n-1)}{2}.$$

3.3. Insertion Sort

If an array is **reverse sorted order (worst-case)**:

- Use the arithmetic sum result:

$$\sum_{j=1}^{n} x = \frac{n(n+1)}{2}$$

- The L5 and L6 running times can be expressed in terms of n as follows:

$$\sum_{j=2}^{n} j = \left(\sum_{j=1}^{n} j\right) - 1 = \frac{n(n+1)}{2} - 1$$

$$\sum_{j=2}^{n} (j-1) = \left(\sum_{j=1}^{n} (j-1)\right) - 1 = \frac{n(n-1)}{2}$$

3.3. Insertion Sort

The worst case continues. The running time $T(n)$ is a quadratic function of n, $T(n) = O(n^2)$:

$$T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5 \sum_{j=2}^{n} t_j + c_6 \sum_{j=2}^{n} (t_j - 1) + c_7 \sum_{j=2}^{n} (t_j - 1) + c_8(n-1)$$

$$\sum_{j=2}^{n} t_j = \frac{n(n+1)}{2} - 1 \qquad \sum_{j=2}^{n} (t_j - 1) = \frac{n(n-1)}{2}$$

$$= c_1 n + c_2(n-1) + c_4(n-1) + c_5\left(\frac{n(n+1)}{2} - 1\right) + (c_6 + c_7)\left(\frac{n(n-1)}{2}\right) + c_8(n-1)$$

$$= \frac{n^2}{2}(c_5 + c_6 + c_7) + n\left(c_1 + c_2 + c_4 + c_8 + \frac{1}{2}(c_5 - c_6 - c_7)\right) - (c_2 + c_4 + c_5 + c_8)$$

$$= C_1\left(\frac{n^2}{2}\right) + C_2 n - C_3 = \mathrm{O}(\mathrm{n}^2)$$

3.3. Insertion Sort

We usually concentrate on finding the worst-case running time:

- The worst-case guarantees us the upper bound on the running time for any input.
- For some algorithms, the worst case occurs fairly often, e.g., searching for absent information might happen quite often.
- The average case is often as bad as the worst case:
  - Assume half of the elements in $A[1, \cdots, j-1]$ are less than $A[j]$ and half are greater.
    - Then $t_j = j/2$ and the while loop looks halfway through the sorted subarray $A[1, \cdots, j-1]$.
    - $\Rightarrow$ The average-case running time is a quadratic function of $n, \mathrm{T(n)} = \mathrm{O(n^2)}$
    - $T(n) = \frac{C_1}{2}\left(\frac{n^2}{2}\right) + \frac{C_2}{2}n - C_3$

3.3. Insertion Sort

We can use simplified abstractions:

- Ignore the actual cost (e.g., c_j) of each statement.
- But still, observe that constants give more than needed.
- The order of growth of the running time makes the analysis even simpler.
  - Consider only the *leading* term and ignore the coefficient of the leading term.
  - Example: $T(n) = an^2 + bn + c \Rightarrow T(n) \approx an^2 \Rightarrow T(n) \rightarrow \Theta(n^2)$

3.4. Designing Algorithms - Merge Sort

3.4. Designing Algorithms – Merge Sort

- For insertion sort, we used an ***incremental*** approach.
- We can also use the "**divide-and-conquer**" method to design a sorting algorithm called the **merge sort**.
- We will find that the worst-case running time is much less than the incremental approach (insertion sort).
- We are going to break the problem into several subproblems similar to the original problem by
  - **Dividing** into smaller sizes.
  - **Conquering** the subproblems **recursively.**
  - **Combining** the solutions to create a solution to the original problem.

3.4. Designing Algorithms – Merge Sort

Merge sort

- **Divides** by splitting the input array into two subarrays `A[p,…,q]` and `A[q+1,…,r]`, where `q` is the middle point of `A[p,…,r]`.
- **Conquers** by recursively sorting these two subarrays.
- **Combines** by merging two sorted subarrays $A[p, \cdots, q]$ and $A[q+1, \cdots, r]$ to generate a single sorted subarray.

We will design the merge sort algorithm into two parts:

- `MergeSort(A,p,r)` will be recursively called to split the subarrays.
- `Merge(A,p,q,r)` to accomplish the combination step (last step).
  - We will add a sentinel ∞ at the end of each subarray to ensure the subsolutions are always correct.
- We let the recursion stop when the subarray consists of one element.

3.4. Designing Algorithms – Merge Sort

2	4	5	7	1	2	3	6
1	2	2	3	4	5	6	7

Left subarray: A[p, … , q, ∞]

2	4	5	7	∞

Right subarray: A[q+1, … , r, ∞]

1	2	3	6	∞

$L[i = 1] > R[j = 1]$, A[1]=R[1], j++

1

$L[1] \leq R[2]$, A[2]=2, i++

1	2

```
n1=q-p+1
L[n1+1]
for (1<=i<=n1):
    L[i]=A[p+i-1]
L[n1+1]=∞
```

```
n2=r-q
R[n2+1]
for (1<=j<=n2):
    R[j]=A[q+j]
R[n2+1]=∞
```

3.4. Designing Algorithms – Merge Sort

2	4	5	7	1	2	3	6
1	2	2	3	4	5	6	7

2	4	5	7	∞

1	2	3	6	∞

$$A[3] = 2, L[2] > R[2] \rightarrow 4 > 2, i = 2, j = 3$$
$$A[4] = 3, L[2] > R[3] \rightarrow 4 > 3, i = 2, j = 4$$
$$A[5] = 4, L[2] \leq R[4] \rightarrow 4 \leq 6, i = 3, j = 4$$
$$A[6] = 5, L[3] \leq R[4] \rightarrow 5 \leq 6, i = 4, j = 4$$
$$A[7] = 6, L[4] > R[4] \rightarrow 7 > 6, i = 4, j = 5$$
$$A[8] = 7, L[4] \leq R[5] \rightarrow 7 \leq \infty, i = 4, j = 6$$

1	2	2	3	4	5	6	7

```
i=1, j=1
for (p<=k<=r):
    if(L[i]<=R[j]):
      A[k]=L[i]
      i=i+1
    else:
      A[k]=R[j]
      j=j+1
```

3.4. Merge Sort – Merge(A,p,q,r)

Merge(A,p,q,r)

```
 1 n1=q-p+1, n2=r-q
 2 L[n1+1], R[n2+1]
 3 for (1<=i<=n1):
 4     L[i]=A[p+i-1]
 5 for (1<=j<=n2):
 6     R[j]=A[q+j]
 7 L[n1+1]=∞, R[n2+1]=∞
 8 i=1, j=1
 9 for (p<=k<=r):
10     if(L[i]<=R[j]):
11       A[k]=L[i]
12       i=i+1
13     else:
14       A[k]=R[j]
15       j=j+1
```

Analysis:

- Let each instruction have its own constant cost.
- L1, 2, 7, and 8 do not repeat.
  - They sum up to constant C_1.
- The combination of L4 and 6 is $C_2 n$
  - L4 repeats n1 many times.
  - L6 repeats n2 many times.
- The last for-loop (L9-15) iterates r-q+1 (=n) many times: $C_3 n$
- The total running time of **Merge()** is $\Theta(n)$.

3.4. Designing Algorithms – Merge Sort

Loop Invariants on the 3rd for-loop (L8-15):

Initialization:

- Before the first iteration of the loop, we have `k=p`, and `A[p,…,k-1]` is empty.
  - It contains the `k-p` `(=0th)` smallest elements of `L` and `R`.
- Since `i=j=1` (L8), both `L[i]` and `R[j]` are the smallest elements of their arrays that have not been copied back into A.
- This is true even if p does not start from 0.

3.4. Designing Algorithms – Merge Sort

Loop Invariants on the 3rd for-loop (L8-15):

Maintenance:

- Suppose `L[i]<=R[j]`, then `L[i]` is the smallest element not yet copied back into A because `A[p,…,k-1]` contains the `k-p` smallest elements.
- After L11, the algorithm copies `L[i]` into `A[k]`, the subarray `A[p,…,k]` will contain the `k-p+1` smallest elements. The for-loop updates and reestablishes the loop invariant for the next iteration.
- Same idea for a case of `L[i]>R[j]`.

3.4. Designing Algorithms – Merge Sort

Loop Invariants on the 3rd for-loop (L8-15):

Termination:

- Terminates when `k=r+1`.
- By loop invariant, the subarray contains the smallest element of `L[1,…,n1+1]` and `R[1,…,n2+1]` in sorted order.
- `L` and `R` combine a total of `n1+n2+2=r-p+3` elements.
- All except the two largest (the sentinels) have been copied back into `A`.
- `A` has a total of n many elements.

3.4. Designing Algorithms – Merge Sort

Using the Merge procedure, we can make **`MergeSort(A,p,r)`**:

- The recursion stops when the subarray has at most one element – the subarray is already sorted.
- Calculate an index q that partitions an array A into two subarrays.
- Let **`MergeSort()`** splits and sorts subarrays recursively.
- Use **`Merge()`** to conquer and combine the subarrays.

`MergeSort(A,p,q)`

```
1 if (p<r):
2    q = floor((p+r)/2)
3    MergeSort(A,p,q)
4    MergeSort(A,q+1,r)
5    Merge(A,p,q,r)
```

The initial call is MERGE-SORT(A,1,n)

3.4. Designing Algorithms – Merge Sort

```
MergeSort(A,1,8)
1 if (p<r):
2    q = floor((p+r)/2)
3    MergeSort(A,p,q)
4    MergeSort(A,q+1,r)
5    Merge(A,p,q,r)
```

MS(A,1,8): p=1,r=8,q=4

MS(A,1,4)||MS(A,5,8)

MS(A,1,2)|MS(A,3,4)||MS(A,5,6)|MS(A,7,8)

MS(A,1,1)|MS(A,2,2)|MS(A,3,3)|MS(A,4,4)||MS(A,5,5)|MS(A,6,6)|MS(A,7,7)|MS(A,8,8)

M(A,1,2,4)||M(A,5,6,8)

M(A,1,4,8)

3.4. Designing Algorithms – Merge Sort


```
MergeSort(A,1,8)
1 if (p<r):
2    q = floor((p+r)/2)
3    MergeSort(A,p,q)
4    MergeSort(A,q+1,r)
5    Merge(A,p,q,r)
```

5	2	4	7	1	3	2	6	(original array)
5	2	4	7	1	3	2	6	(1st recursion)
5	2	4	7	1	3	2	6	(2nd recursion)
5	2	4	7	1	3	2	6	(3rd recursion, merge)
2	5	4	7	1	3	2	6	(2nd merge)
2	4	5	7	1	2	3	6	(1st merge)
1	2	2	3	4	5	6	7	(elements sorted)

3.4. Designing Algorithms – Merge Sort

When an algorithm contains a **recursive** call to itself, we use a **recurrence equation** to describe the running time of a divide-and-conquer algorithm.

- The recurrence equation is

$$T(n) = \begin{cases} \Theta(1) & if\ n \le c, \\ aT\left(\frac{n}{b}\right) + D(n) + C(n) & otherwise. \end{cases}$$

- Base case: if the problem size is small enough, n<=c, the base case takes constant time $\Theta(1)$.
- In other cases: an array divides into a-many subproblems with the rate of 1/b.
  - There are a-many subproblems to solve, each of size n/b.
  - Each subproblem takes $T(n/b)$ times to solve, and the total time is $aT\left(\frac{n}{b}\right)$.
  - The time to divide a n-size problem takes $D(n)$.
  - The time to combine solutions takes $C(n)$.

3.4. Designing Algorithms – Merge Sort

For simplicity, we assume that the array has the length of n($= 2^m$).

- Suppose the problem divides into two subproblems $\left(\frac{n}{2}\right)$ in each step.
- Base case for $n = 1$, $T(n) = \Theta(1)$.
- When $n \geq 2$:
  - $D(n) = \Theta(1)$: computes q as an average of p and r.
  - aT(n/b)=2T(n/2): **`MergeSort(A,p,r)`** solves two (a=2) n/2 long subproblems (b=2) recursively.
  - $C(n) = \Theta(n)$. **`MERGE(A,p,q,r)`** takes linear time.
  - The total running time recursion equation can be written as

$$T(n) = \begin{cases} \Theta(1) & if\ n = 1 \\ 2T\left(\frac{n}{2}\right) + \Theta(n) + \Theta(1) & if\ n = 1 \end{cases}.$$

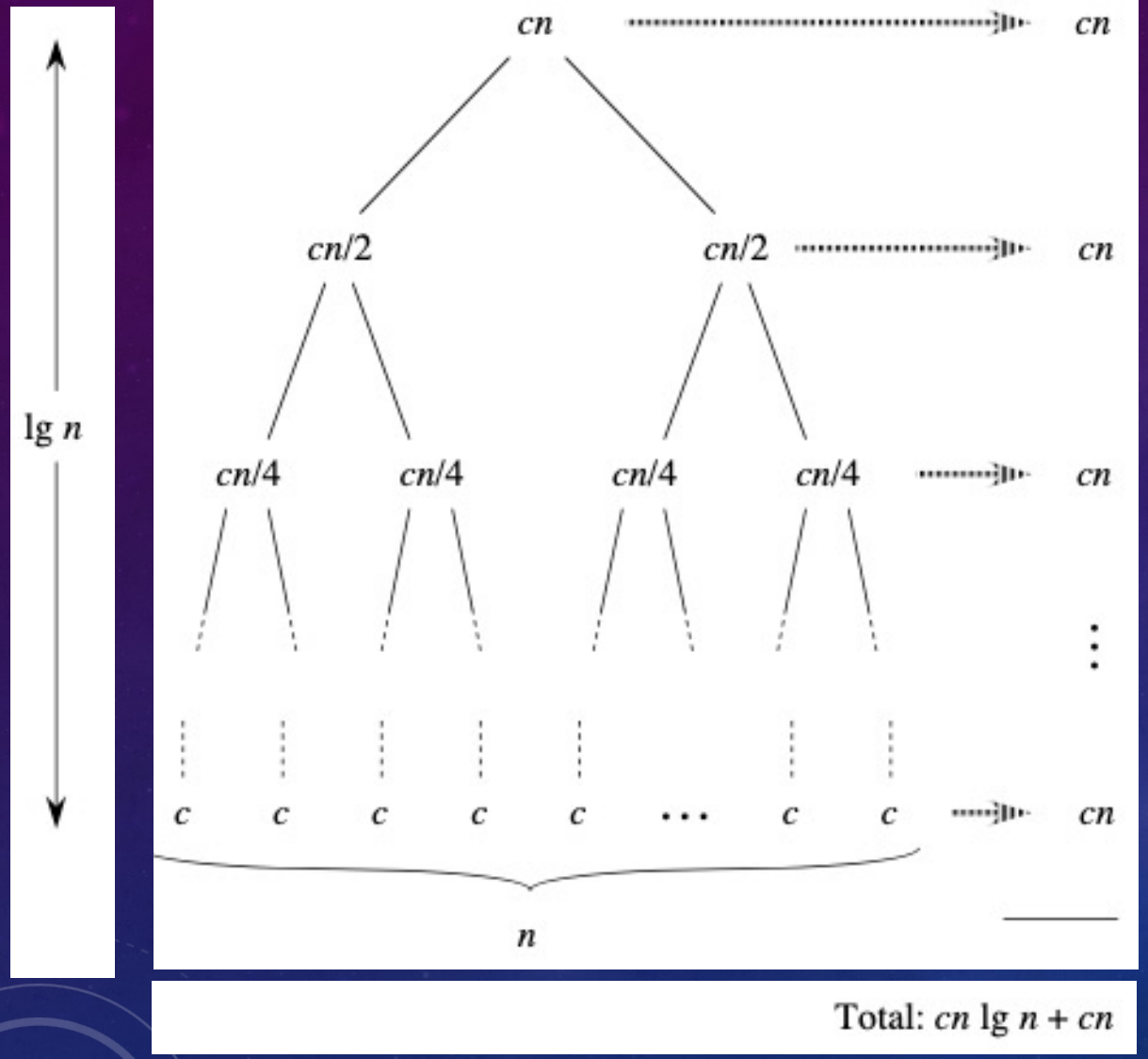
3.4. Designing Algorithms – Merge Sort

- Rewrite the recurrence equation as

$$T(n) = \begin{cases} c & if\ (n = 1), \\ 2T\left(\frac{n}{2}\right) + c_1 n & if\ (n > 1). \end{cases}$$

- We will draw a **recursion tree** to solve the equation.
- The running time for merge-sort is $T(n) = \Theta(n \lg n)$ where $\lg n = \log_2 n$.

3.4. Designing Algorithms – Merge Sort



Each depth has cost of cn.

- Each time we go down one depth, the number of subproblems **doubles**.
- The cost per subproblem **halves**.
- The makes cost/level stays the same format
- The total cost at each depth is then cn.

The height of the tree is lg(n)+1.

- The tree splits until each leaf has one element.
- n elements makes a total of 2^{i+1} nodes.

Total cost is sum of total cost at each depth.

- Total cost is $cn \lg n + cn \Rightarrow \Theta(n \lg n)$.

3.4. Designing Algorithms – Merge Sort

Recall the divide-and-conquer technique:

- Divide the problem into a number of subproblems that are small instances of same problem.
- Conquer the subproblem by use of recursion. Small enough or trivial subproblems (base case) are solved in a straightforward manner.
- Combine the subproblems solutions into the solution for the original problem.

We use recurrence to characterize the running time of a divide-and-conquer algorithm and its solution gives us the asymptotic running time.

3.5 C++ and Homework Discussion

3.5. Homework Discussion

The first homework will be open this Friday (9/15) @ 12 AM.

Information

- Due: 9/29th Friday 11:59 PM
- Headers and other necessary files are compressed in a zip file with the description in docx file.
- Students need to work on the source file, “sort.cpp”, only.
- This assignment requires a report (see slide 74).
- When you submit, make sure to compress only code files.
  - **Do not just submit the sort.cpp file.**
  - Submit the compressed file and the report in a docx file (not in pdf).
- Keep in mind that
  - The description is written for C++ users. If you are going to use other than C++, you have to measure the running time on your own.

3.5. Homework Discussion

3. Measure the runtime performance of insertion sort (naive and improved) and merge sort for random, sorted, and inverse sorted inputs of size m = 10000; 25000; 50000; 100000; 250000; 500000; 1000000; 2500000 and vector dimension n = 10; 25; 50. You can use the provided functions *create_random_ivector*, *create_sorted_ivector, create reverse_sorted_ivector.*

Repeat each test a number of times (usually at least 10 times) and compute the average running time for each combination of algorithm, input, size m, and vector dimension n. Report and comment on your results.

3.5. Homework Discussion

Sample test case:
Consider input [[4,2,1], [9,2,1], [1,2,1]],
Output should be [[1,2,1], [4,2,1], [9,2,1]]

Explanation:

- 1+2+1 < 4+2+1 < 9+2+1
- Sorting occurs according to sum of numbers at that index as shown above.
- Vector just means an array. Eg. [4,2,1] is a vector
- Length means sum of numbers in that vector. Eg. Length of vector [4,2,1] is 7
  - $[[4,2,1], [9,2,1], [1,2,1]] \rightarrow [7,12,4] \rightarrow [[1,2,1], [4,2,1], [9,2,1]]$

3.5. Homework Discussion

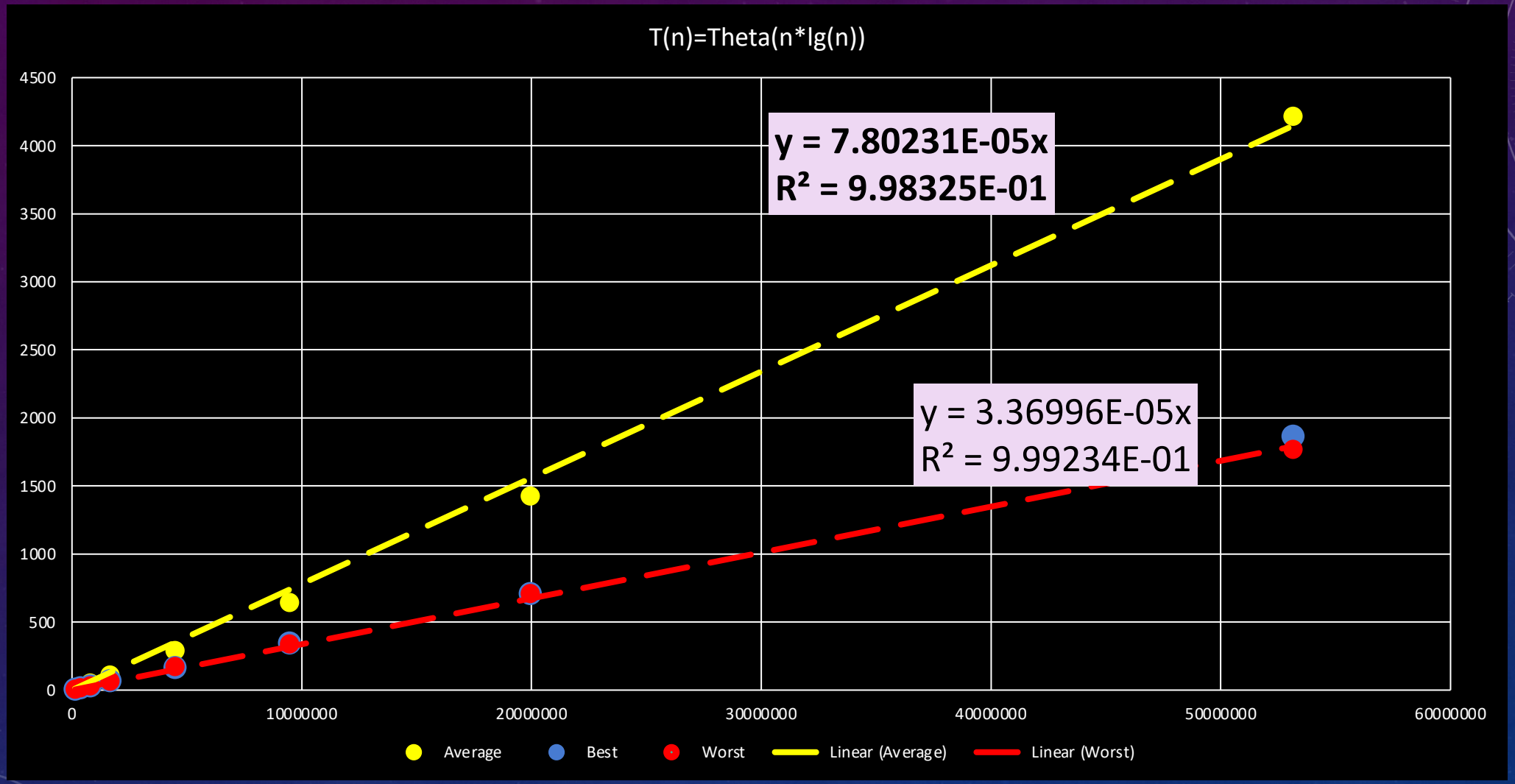
Table 1: Merge Sort Runtime in ms (Average Summary)

	n = 10			n = 25			n = 50		
m	Random Vector	Sorted Vector	Inverse Sorted Vector	Random Vector	Sorted Vector	Inverse Sorted Vector	Random Vector	Sorted Vector	Inverse Sorted Vector
10000	9.7	6.5	6.1	18.5	11.5	11.7	33.1	20.3	18.4
25000	25.7	17.3	17.6	48.1	30.2	28.6	89.2	51.6	48.6
50000	51.7	32.4	31.2	102	58	56.1	187.3	106.2	102.3
100000	109.7	68.4	65.6	212.2	120.5	118.1	402.1	217.7	208.7
250000	292.8	167.3	172.2	595.7	305.1	310.4	1110	550.2	536.5
500000	646.1	348.3	338.4	1251	615.5	618.5	2288.4	1170	1133.8
1000000	1425.6	709.6	710.7	2760.5	1271	1279.7	4936.3	2507	2319.3
2500000	4216.3	1868	1770.3	7730.1	3679	3532.8	14040.5	6820	6628.8

Table 2: Insertion Sort Runtime in ms (Average Summary)

	n = 10			n = 25			n = 50		
m	Random Vector	Sorted Vector	Inverse Sorted Vector	Random Vector	Sorted Vector	Inverse Sorted Vector	Random Vector	Sorted Vector	Inverse Sorted Vector
10000	1135.6	0.5	2172.9	2582.8	1.2	5288.7	5575.9	2.5	10652.4
25000	7760.7	1.2	14010.2	17816.3	2.8	33459.6	38181.2	5.5	69056.1
50000	31252.2	2.1	53042.3	67483.8	5	122766.5	165510	11.4	262562.3
100000	122095.5	4.6	217219.2	352616	12.7	-	-	27.5	-
250000	1202565	11.8	-	-	33	-	-	58.1	-
500000	-	26.7	-	-	57.5	-	-	113.7	-
1000000	-	53.4	-	-	109.6	-	-	225.8	-
2500000	-	120.9	-	-	261.6	-	-	563.1	-

3.5. Homework Discussion



3.5. Homework Discussion

Abstract – An objective statement

Result:

- Test sorting algorithms – insertion sort, modified insertion sort, and merge sort
- Testing vectors are
  - Size $m = 1000, 5000, 10000, 50000, 100000, \ and \ 500000$
  - Dimension $n = 10, 25, 50, and \ 75$
  - Order descending, ascending, and random
  - Test the combination of size, dimension, and order
- Present results (tables and plots) and summarize the findings.
- Discussion:
- Evaluate your test results by explaining
  - Agreement of running time with theories
  - Any limitation on m or n?
  - Which factor between m and n was the most attributed to running time?

Conclusion:

- Explain the success of your test based on the result and discussion.