CS 590: Algorithm

Lecture 3: Sorting Algorithm and Computation Complexity 1

Reading Assignment: Chapter 2, 3

Next Week: Chapter 4, 5

Outline of Lecture:

- 3.1. Analyzing algorithm
- 3.2. Growth of Function
- 3.3. Insertion Sort
- 3.4. Merge Sort
- 3.5. Homework – Report Sample

- 3.1. Analyzing algorithm
- 3.2. Growth of Function
- 3.3. Insertion Sort
- 3.4. Merge Sort
- 3.5. Homework – Report Sample

3.1. Analyzing Algorithms

Analyzing Algorithm – *predicting* the resources that the algorithm requires (usually running time).

To predict resource requirements, we need a computational model.

- **Random Access Machine (RAM) – summing up** the number of steps needed to execute the algorithm on a set of data.

3.1. Analyzing Algorithms

RAM:

- Instructions are executed one after another. **No concurrent operations.**
- It's too tedious to define each of the instructions and their associated time costs.
- Instead, we recognize that we will use instructions commonly found in real computers:
  - **Arithmetic**: add, subtract, multiply, divide, remainder, floor, ceiling. Also, shift left/shift right (good for multiplying or dividing by 2^k).
  - **Data movement**: load, store, copy.
  - **Control**: conditional/unconditional branch, subroutine call and return.
- It uses integer and floating-point types.
  - We do not worry about precision.
  - There is **a limit on the word size** – when working with inputs of size n, assume that the integers are presented by $c \lg n$ bits for some constant $c \geq 1$. ($\lg n = \log_2 n$)
    - $c \geq 1 \Rightarrow$ we can hold the value of n $\Rightarrow$ can index the individual elements.
    - c is constant $\Rightarrow$ the word size cannot grow arbitrarily.

3.1. Analyzing Algorithms

How do we analyze an algorithm's running time?

The time taken by an algorithm depends on the input.

- Sorting 1000 numbers takes longer than sorting 3 numbers.
- A given sorting algorithm may even take differing amounts of time on two inputs of the same size.
- For example, we will see that insertion sort takes less time to sort n elements when they are already sorted than when they are in reverse sorted order.

Input size: Depends on the problem being studied.

- Usually, the number of items in the input. Like the size n of the array being sorted.
- But could be something else. If multiplying two integers, could be the total number of bits in the two integers.
- Could be described by more than one number. For example, graph algorithm running times are usually expressed in terms of the number of vertices and the number of edges in the input graph.

3.1. Analyzing Algorithms

Running time: On a particular input, it is the number of primitive operations (steps) executed.

- Want to define steps to be **machine-independent**.
- Figure that each line of pseudocode requires a constant amount of time.
  - One line may take a different amount of time than another, but each execution of line *i* takes the same amount of time c_i .
- This is assuming that the line consists **only of primitive operations**.
  - If the line is a *subroutine call*, then the actual call takes constant time, but the execution of the subroutine being called might not.
  - If the line specifies operations other than primitive ones, then it might take more than constant time. Example: "sort the points by *x*-coordinate".

- 3.1. Analyzing algorithm
- 3.2. Growth of Function
- 3.3. Insertion Sort
- 3.4. Merge Sort
- 3.5. Homework – Report Sample

3.2. Growth of Functions

Asymptotic Notation: O −notation

$$O(g(n)) = \{f(n): \text{there exist positive constant } c \text{ and } n_0, \text{ such that (s.t.) } 0 \le f(n) \le cg(n) \text{ for all } n \ge n_0\}$$

- $g(n)$ is an asymptotic **upper bound** for $f(n)$ when $n \ge n_0$.
- If $f(n) \in O(g(n))$, we write $f(n) = O(g(n))$
- $2n^2 = O(n^3)$ with $c = 1$ and $n_0 = 2$
- $n^2, n^2 + n, n^2 + 1000n$, etc… are all $\in O(n^2)$.

3.2. Growth of Functions

Asymptotic Notation: Ω –notation

$$\Omega(g(n)) = \{f(n): \text{there exist positive constant } c \text{ and } n_0 \text{ s.t. } 0 \le cg(n) \le f(n) \text{ for all } n \ge n_0\}$$

- $g(n)$ is an asymptotic **lower bound** for $f(n)$.
- If $f(n) \in \Omega(g(n))$, we write $f(n) = \Omega(g(n))$
- $\sqrt{n} = \Omega(\lg n)$ with $c = 1$ & $n_0 = 16$.
- $n^2, n^2 + n, n^2 - n, n^2 + 1000n, n^2 - 1000n$, etc… are all $\in \Omega(n^2)$.

3.2. Growth of Functions

Asymptotic Notation: Θ −notation

$$\Theta\big(g(n)\big) = \{f(n)\text{: there exist positive constant } c_1, c_2, \text{and } n_0 \text{ s.t. } 0 \le c_1 g(n) \le f(n) \le c_2 g(n) \text{ for all } n \ge n_0\}$$

- $g(n)$ is an asymptotic **tight bound** for $f(n)$.
- $f(n) = \Theta\big(g(n)\big)$ i.f.f. $f = O(g(n))$ & $f = \Omega(g(n))$.
- $\frac{n^2}{2} - 2n$ with $c_1 = \frac{1}{4}, c_2 = \frac{1}{2}, \& \, n_0 = 8$.
- $n^2, n^2 + n, n^2 - n, n^2 + 1000n, n^2 - 1000n$, etc… are all $\in O(n^2), \Omega(n^2), \therefore \Theta(n^2)$.

3.2. Growth of Functions

Asymptotic notations in equations:

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

We can ***chain*** together:

<u>Interpretation:</u>

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

$$o(g(n)) = \{f(n): \textbf{for all constants } c > 0, \textbf{ there exist constant } n_0 > 0 \textbf{ s.t. } 0 \le f(n) < cg(n) \textbf{ for all } n \ge n_0\}$$

This makes the difference between big-O and o.

- $g(n)$ is an **upper bound that is not asymptotically tight** for $f(n)$.
- $2n = o(n^2)$ but $2n^2 \neq o(n^2)$. Hence, $2n^2 = O(n)$.
- $f(n)$ becomes insignificant relative to $g(n)$ as n approaches infinity:
  - $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$

3.2. Growth of Functions

Asymptotic Notation: ω −notation

$$\boldsymbol{\omega(g(n)) = \{f(n)\text{: for all constants c} > 0,}$$
$$\boldsymbol{\text{there exist constant } n_0 > 0 \text{ s.t. } 0 \le cg(n) < f(n) \text{ for all } n \ge n_0\}}$$

- Another way to define: $g(n)$ is an **lower bound that is not asymptotically tight** for $f(n)$.
- $f(n) \in \omega(g(n))$ if and only if $g(n) \in o(f(n))$.
- $n^2/2 = \omega(n)$ but $n^2/2 \neq \omega(n^2)$.
- $f(n)$ becomes arbitrarily large relative to $g(n)$ as n approaches infinity:
  - $\lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty$

3.2. Growth of Functions

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

3.2. Growth of Functions

Transitivity	$f(n) = \Theta(\mathrm{g(n)})$ and $g(n) = \Theta(\mathrm{h(n)})$ imply $f(n) = \Theta(\mathrm{h(n)})$ (same for $O, \Omega, \mathrm{o}, \ \& \ \omega$)
Reflexivity	$f(n) = \Theta(f(n)), f(n) = O(f(n)), f(n) = \Omega(f(n))$. (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.

- 3.1. Analyzing algorithm
- 3.2. Growth of Function
- 3.3. Insertion Sort
- 3.4. Merge Sort
- 3.5. Homework – Report Sample

3.3. Insertion Sort

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

<u>Definition:</u>

- **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.

5	2	1	6

Figure 2.1 Sorting a hand of cards using insertion sort.

3.3. Insertion Sort

Algorithm (Insertion sort)

INPUT: $A[1 \ldots n]$
OUTPUT: $A[1 \ldots n]$ *sorted*

```
for (2 ≤ j ≤ n) do
    key = A[j]
    /* Insert A[j] into the sorted sequence A[1..j − 1]. */
    i = j − 1
    while (i > 0 and A[i] > key) do
        A[i + 1] = A[i]
        i = i − 1
    A[i + 1] = key
od
```

3.3. Insertion Sort

5	2	1	6

Line 1: j = 2

Line 2: j =2, key = A[2]=2

Line 4: i = 1

Line 5: j = 2, key = 2, i = 1, and A[1] > 2

Line 6: j = 2, key = 2, A[2] = A[1] ,

A[1]=? & A[2]=?

Line 7: j =2, key = 2, i = 0

Line 8: j =2, key = 2, A[2] = 5, i = 0, A[1] = 2 (why?)

2	5	1	6

Algorithm (Insertion sort)

INPUT: $A[1 \ldots n]$
OUTPUT: $A[1 \ldots n]$ *sorted*

```
(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)      A[i + 1] = key
```

3.3. Insertion Sort

2	5	1	6

Algorithm (Insertion sort)

INPUT: $A[1 \ldots n]$
OUTPUT: $A[1 \ldots n]$ *sorted*

```
(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)      A[i + 1] = key
(9)  od
```

Line 1: j =3, key = 2, i = 0, A[1] = 2

Line 2: j =3, key = A[3]=1

Line 4: i = 2

Line 5: j = 3, key = 1, i = 2, A[2]=5 > 1

Line 6: j = 3, key = 1, i = 2, A[3] = A[2]=5

Line 7: j =3, key = 1, A[2] = ?, A[3]=5, i = 1, A[1]=2

Line 5: j = 3, key = 1, i = 1, A[1]=2 > 1

Line 6: j = 3, key = 1, i = 1, A[2] = A[1]=2

Line 7: j =3, key = 1, A[2] = 2, A[1]=?, i = 0

Line 5: j = 3, key = 1, i = 0, Fail the while-loop condition

Line 8: j =3, i = 1, key = 1, A[1]=1

2		5	6
	2	5	6
1	2	5	6

3.3. Insertion Sort

1 2 5 6

Line 1: j =4, key = 1, i = 1, A[1] = 1

Line 2 & 4: j =4, key = 6, i = 3

Line 5: j =4, key = 6, i = 3, A[3] < 6

Step 6 &7: SKIP

Line 8: j =4, i = 3, A[4] = 6, key = 6

1 2 5 6

Algorithm (Insertion sort)

INPUT: $A[1 \ldots n]$
OUTPUT: $A[1 \ldots n]$ *sorted*

```
(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)     A[i + 1] = key
(9) od
```

3.3. Insertion Sort

We often use a **loop invariant** to help us understand why an algorithm gives the correct answer.

Loop Invariant:

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

To prove <u>correctness</u>, we must show three things about loop invariant:

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

3.3. Insertion Sort

Initialization:

- Show the loop invariant holds before the 1st loop: $j = 2$
- The subarray $A[1, \cdots, j-1]$ consists 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.

Algorithm (Insertion sort)

INPUT: $A[1 \ldots n]$
OUTPUT: $A[1 \ldots n]$ *sorted*

```
for (2 ≤ j ≤ n) do
    key = A[j]
    /* Insert A[j] into the sorted sequence A[1..j − 1]. */
    i = j − 1
    while (i > 0 and A[i] > key) do
        A[i + 1] = A[i]
        i = i − 1
    A[i + 1] = key
od
```

3.3. Insertion Sort

Maintenance:

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

Algorithm (Insertion sort)

INPUT: $A[1 \ldots n]$
OUTPUT: $A[1 \ldots n]$ *sorted*

```
for (2 ≤ j ≤ n) do
    key = A[j]
    /* Insert A[j] into the sorted sequence A[1..j − 1]. */
    i = j − 1
    while (i > 0 and A[i] > key) do
        A[i + 1] = A[i]
        i = i − 1
    A[i + 1] = key
od
```

3.3. Insertion Sort

Termination:

- The for loop terminates when $j > A.length = n$ and we have $j = n + 1$ at that time.
- We have subarray $A[1, \cdots, n]$ in sorted order.

Algorithm (Insertion sort)

INPUT: $A[1 \ldots n]$
OUTPUT: $A[1 \ldots n]$ *sorted*

```
for (2 ≤ j ≤ n) do
    key = A[j]
    /* Insert A[j] into the sorted sequence A[1..j − 1]. */
    i = j − 1
    while (i > 0 and A[i] > key) do
        A[i + 1] = A[i]
        i = i − 1
    A[i + 1] = key
od
```

3.3. Analysis of insertion sort

Why n?

Algorithm (Insertion sort)

INPUT: $A[1 \ldots n]$
OUTPUT: $A[1 \ldots n]$ *sorted*

```
(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)      A[i + 1] = key
(9)  od
```

Line	Cost	Times
(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	$n-1$

- Assume the jth line takes time **constant** c_j.
- Let t_j be the number of times that the while loop test 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. Analysis of insertion sort

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

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

- Running time depends on **size of the input,** n**,** and t_j.
- Value of t_j depends on the size and the input itself.

3.3. Analysis of insertion sort

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

$$T(n)$$

$$= c_1 n + c_2(n-1) + c_4(n-1) + c_5 \overset{(n-1)}{\overset{\parallel}{\sum_{j=2}^{n} t_j}} + c_6 \overset{0}{\overset{\parallel}{\sum_{j=2}^{n} (t_j - 1)}} + c_7 \overset{0}{\overset{\parallel}{\sum_{j=2}^{n} (t_j - 1)}} + c_8(n-1)$$

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

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

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

3.3. Analysis of insertion sort

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

- Always find that $\boldsymbol{A[i] > key}$ in while loop test.
- Has to compare key with all elements to the left of the jth position.
- Since the while loop exists because i reaches 0, there is one additional test after $j-1$ tests $\Rightarrow$ $t_j = j$.
- $\sum_{j=2}^{n} t_i = \sum_{j=2}^{n} j$ and $\sum_{j=2}^{n}(t_i - 1) = \sum_{j=2}^{n}(j-1)$.
- 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. Analysis of insertion sort

The running time $T(n)$ is is a quadratic function of n:

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

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

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

- $T(n)$ is a **quadratic function** of n.
- $T(n) = \Theta(n^2)$

3.3. Analysis of insertion sort

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

- Worst-case gives us a guaranteed 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.
- Average case is often as bad as the worst case:
  - On average half of the elements in $A[1, \cdots, j-1]$ are less than $A[j]$ and half are greater.
    - $\Rightarrow t_j$ is on average $\frac{j}{2}$ and the while loop has to look halfway through the already sorted subarray $A[1, \cdots, j-1]$.
    - $\Rightarrow$ The average-case running time is a quadratic function of n.

3.3. Analysis of insertion sort

We used simplified abstractions:

- Ignored the actual cost of each statement.
- But still observed that constants gave more than needed.
- We still can make this analysis even simpler – order of growth of the running time
  - Consider only the *leading* term
  - Ignore the coefficient of leading term
  - Example: $T(n) = an^2 + bn + c \Rightarrow T(n) \approx an^2 \Rightarrow T(n) \rightarrow \Theta(n^2)$

3.3. Designing algorithms

For insertion sort, we used an incremental approach.
We can also use “divide-and-conquer” method to design a sorting algorithm.

- The worst-case running time is much less than the incremental approach.
- They break the problem into several subproblems that are similar to the original problem but
  - Divide smaller in size
  - Conquer the subproblems recursively
  - Combine there solutions to create a solution to the original problem.

- 3.1. Analyzing algorithm
- 3.2. Growth of Function
- 3.3. Insertion Sort
- 3.4. Merge Sort
- 3.5. Homework – Report Sample

3.4. Merge Sort

Merge sort is based in divide-and-conquer – worst-case running time is better than insertion sort.

- Divide by splitting into two subarrays $A[p, \cdots, q]$ and $A[q+1, \cdots, r]$ where q is the middle point of $A[p, \cdots, r]$.
- Conquer by recursively sorting these two subarrays.
- Combine by merging two sorted subarrays $A[p, \cdots, q]$ and $A[q+1, \cdots, r]$ to generate a single sorted subarray. We define procedure $\text{Merge}(A, p, q, r)$ to accomplish that step.

The recursion stops when the subarray consists of only one element. One element arrays are trivially sorted.

3.4. Merge Sort – Merge(A,p,q,r)

(1)	$n_1 = q - p + 1, n_2 = r - q$	Computes lengths
(2)	Let $L[1 \dots n_1 + 1]$ and $R[1 \dots n_2 + 1]$ be new arrays	Create arrays L and R
(3)	for $(1 \le i \le n_1)$ do	Copies the subarray into L
(4)	$\quad L[i] = A[p + i - 1]$	
(5)	for $(1 \le j \le n_2)$ do	Copies the subarray into R
(6)	$\quad R[j] = A[q + j]$	
(7)	$L[n_1 + 1] = \infty, R[n_2 + 1] = \infty$	Put sentinels at the ends of the arrays L & R
(8)	$i = 1, j = 1$	
(9)	for $(p \le k \le r)$ do	The subarray $A[p \dots k - 1]$ contains the $k - p$ smallest
(10)	$\quad$ if $(L[i] \le R[j])$ then	Elements of $L[i \dots n_1 + 1]$ & $R[1 \dots n_2 + 1]$, in sorted order.
(11)	$\quad\quad A[k] = L[i]$	$L[i]$ and $R[j]$ are the smallest elements of their
(12)	$\quad\quad i = i + 1$	Arrays that have not been copied back into A.
(13)	$\quad$ else	
(14)	$\quad\quad A[k] = R[j]$	
(15)	$\quad\quad j = j + 1$	
(16)	Done	

*MERGE takes time $\Theta(n)$ where $n = r - p + 1$ is the total number of elements being merged.

- Line 1-2 & 7-8 takes constant time.
- Line 3-6 takes $\Theta(n_1 + n_2) = \Theta(n)$
- Line 12-17 takes constant time in for loop n iterations

3.4. Merge Sort

Line 8-16:

2	4	5	7	1	2	3	6	A
2	4	5	7	1	2	3	6	L & R
1	2	2	3	4	5	6	7	A (sorted)

2	4	5	7	∞

1	2	3	6	∞

$L[1] > R[1], k = 1, j = 2, i = 1$
$2 > 1$

1

$L[1] \leq R[2], k = 2, i = 2, j = 2$
$2 \leq 2$

1	2

for $(1 \leq k \leq 8)$ do
  if $(L[i] \leq R[j])$ then
    $A[k] = L[i]$
    $i = i + 1$
  else
    $A[k] = R[j]$
    $j = j + 1$
Done

3.4. Merge Sort

Line 8-16:

2	4	5	7	1	2	3	6	A
2	4	5	7	1	2	3	6	L & R
1	2	2	3	4	5	6	7	A (sorted)

2	4	5	7	∞

1	2	3	6	∞

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

1	2	2	3	4	5	6	7

```
(8)   i = 1, j = 1
(9)   for (p ≤ k ≤ r) do
(10)      if (L[i] ≤ R[j]) then
(11)          A[k] = L[i]
(12)          i = i + 1
(13)      else
(14)          A[k] = R[j]
(15)          j = j + 1
```

3.4. Merge Sort

Each iteration of the loop maintains the invariant that provides a useful property to show correctness when the loop terminates.

Initialization:

- Prior to the first iteration of the loop we have k=p and $A[p \ldots k-1]$ is empty.
- This contains smallest elements of L and R.
- Since $i = j = 1$, both L[i] and R[j] are the smallest elements of their arrays that have not been copied back into A.

3.4. Merge Sort

Maintenance:

- Suppose $L[i] \leq R[j]$, then $L[i]$ is the smallest element not yet copied back into A because $A[p \dots k-1]$ contains the k-p smallest elements.
- After line 11 copies L[i] into A[k], the subarray $A[p \dots k]$ 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 $L[i] > R[j]$.

Termination:

- Terminates when $k = r + 1$.
- By loop invariant, the subarray contains the smallest element of $L[1 \dots n_1 + 1]$ and $R[1 \dots n_2 + 1]$ in sorted order.
- Together contains $n_1 + n_2 + 2 = r - p + 3$ elements.
- All but the two largest (the sentinels) have been copied back into A.

3.4. Merge Sort

Using the Merge procedure, we can make MERGE-SORT(A, p, r):

- If $p \geq r$, the subarray has at most one element and is therefore already sorted.
- Otherwise, divide step simply computes an index q that partitions $A[p \ldots r]$ into two subarrays.
- $A[p \ldots q]$ contains n/2 elements and $A[q + 1 \ldots r]$ contains n/2 elements.

```
MERGE-SORT(A,p,r)
(1)   if (p < r) then
(2)       q = ⌊(p + r)/2⌋
(3)       MERGE-SORT(A,p,q)
(4)       MERGE-SORT(A,q+1,r)
(5)       MERGE(A,p,q,r)
```

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

3.4. Merge Sort

```
if (p < r) then
    q = ⌊(p + r)/2⌋
    MERGE-SORT(A,p,q)
    MERGE-SORT(A,q+1,r)
    MERGE(A,p,q,r)
```

$$MS(A,1,8): p = 1, r = 8, q = 4$$
$$MS(A, p = 1, q = 4) \mid MS(A, q + 1 = 5, r = 8):$$
$$MS(A, p = 1, q = 2) \mid MS(A, q + 1 = 3, r = 4) \mid MS(A, p = 5, q = 6) \mid MS(A, q + 1 = 7, r = 8)$$
$$MS(A, p = 1, q = 1) \mid MS(A, q + 1 = 2, r = 2) \mid MS(A, p = 3, q = 3) \mid MS(A, q + 1 = 4, r = 4)$$
$$M(A, p = 1, q = 1, r = 2) \mid M(A, p = 3, q = 4, r = 4)$$
$$M(A, p = 1, q = 2, r = 4)$$
$$MS(A, p = 5, q = 5) \mid MS(A, q + 1 = 6, r = 6) \mid MS(A, p = 7, q = 7) \mid MS(A, q + 1 = 8, r = 8)$$
$$M(A, p = 5, q = 6, r = 6) \mid M(A, p = 7, q = 7, r = 8)$$
$$M(A, p = 5, q = 6, r = 8)$$
$$M(A, p = 1, q = 4, r = 8)$$

3.4. Merge Sort

```
if (p < r) then
    q = ⌊(p + r)/2⌋
    MERGE-SORT(A,p,q)
    MERGE-SORT(A,q+1,r)
    MERGE(A,p,q,r)
```

5	2	4	7	1	3	2	6	(original array)
5	2	4	7	1	3	2	6	(recursion)
5	2	4	7	1	3	2	6	(recursion)
5	2	4	7	1	3	2	6	(sub problems sorted, merge)
2	5	4	7	1	3	2	6	(merge)
2	4	5	7	1	2	3	6	(merge)
1	2	2	3	4	5	6	7	(elements sorted)

3.4. Analyzing Divide-and-Conquer Algorithm

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

- The recurrence equation is

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

- Base case: if the problem size is small enough, $n \leq c$, the base takes constant time $\Theta(1)$.
- Otherwise, divide into a number of a subproblems of size $1/b$
  - There are a subproblems to solve, each of size n/b.
  - Each subproblem takes $T(n/b)$ times to solve and we spend $aT(n/b)$ time solving subproblems.
- The time to divide a size-n problem is $D(n)$.
- The time to combine solutions is $C(n)$.

3.4. Analyzing Merge Sort

For simplicity, we assume that $n = 2^m$.

- Two subproblems of size $\frac{n}{2}$ in each step.
- Base case for $n = 1, T(n) = \Theta(1)$.
- For $n \geq 2$:
  - Divide: $D(n) = \Theta(1)$ (Compute q as the average of p and r is constant time).
  - Conquer: Solve 2 subproblems of size $n/2$ recursively: $a = b = 2 \Rightarrow 2T\left(\frac{n}{2}\right)$.
  - Combine: $C(n) = \Theta(n)$. MERGE on an n-element subarray takes linear time.
  - The total running Time is

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

3.4. Analyzing Merge Sort

- Running time for merge-sort is $T(n) = \Theta(n \lg n)$ where $\lg n = \log_2 n$.
- Rewrite recurrence as

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

- Draw the recursion tree to solve.

3.4. Analyzing Merge Sort

Each level has cost cn.

- Each time we go down one level, the number of subproblems doubles but the cost per subproblem halves ⇒ cost/level stays the same.

There are $\lg n + 1$ levels (height is $\lg n$).
Total cost is sum of costs at each level.

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

3.4. Divide and Conquer

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.

- 3.1. Analyzing algorithm
- 3.2. Growth of Function
- 3.3. Insertion Sort
- 3.4. Merge Sort
- 3.5. Homework – Report Sample

3.5. Homework

The first homework will be open on this Friday 12 AM.
Information

- Due: 10/7th Friday 11:59 PM
- Headers and other necessary files will be provided in zip file together with the description in docx.
- Need to work on source file called “sort.cpp”. See the next file.
- Requires a report
  - Make tables and plots
  - Must explain your findings.
  - Evaluate your work – agree or disagree with theorical running time.
- When you submit, make sure compress only code files.
  - **Do not just submit sort.cpp**
  - Submit the compressed file and the report in docx file (not 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 at your own.
  - There will be penalty (-10%) of not using C++ from HW#2.

3.5. Homework

You are given an integer vector which is represented by *int** an array of integers and its dimension *n* as a separate parameter. We are interested in sorting arrays of integer vectors according to a pre-defined notion of vector length. You therefore are given the function *ivector_length(v, n)* that computes and returns the length of vector *v* with dimension *n* as $\sum_{i=1}^{n} |v_i|$.

You are given a naive (and very inefficient) implementation of insertion sort for arrays of integer vectors.

3.5. Homework

Questions (100 points)

1. Develop an improved implementation of **insertion sort** for integer vector (*insertion_sort_im*) that precomputes the length of each vector before the sorting. Keep in mind that the vectors are sorted according to their length (see *ivector_length* function). You can test the correctness of your sorting algorithm using the provided *check_sorted* function.

2. Implement a **merge sort** for an array of integer vectors. For this implementation of the merge sort, as is the case for the improved insertion sort algorithm, you should precompute the length of the vectors before the sorting, and the sorting is done according to the vector lengths. Test the correctness of your merge sort implementation using the provided *check_sorted* function.

3.5. Homework

Questions (100 points)

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

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
  - $\big[[4,2,1],[9,2,1],[1,2,1]\big] \rightarrow [7,12,4] \rightarrow [[1,2,1],[4,2,1],[9,2,1]]$

3.5. Homework

- "Report and comment" means that you have to analyze and interpret your findings properly. [20 pts]
  - The report must have visual presentations (plots & tables) for each sort at different directions.
    - Table: For each n, tabulate the input order, the length, all measured running time in an array format, and the average running time.

n	$Order$	m	$t\ (ms)$	$\bar{t}\ (ms)$
10	Random	1000	[xxx,xxx,…,xxx,xxx]	xxx
10	Ransom	5000	[xxx,xxx,…,xxx,xxx]	xxx

    - Figure: The figures can be done in any computational languages or in Excel.
    - The figures should present visualizations the growth of running time in a function of m for every input order (random, sorted, and inverse sorted) for each n.
    - The author must express how the results agree with expectations discussed in the lecture.
    - The report must be submitted separately in docx file.

3.5. Homework

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

1. Choose an arbitrary constant C.
2. Adjust C until fits.

3.5. Homework

Abstract – An objective statement

Result:

- Test sorting algorithms – insertion sort, modified insertion sort, and merge sort
- Testing vectors:
  - 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 and have a summary of results.
- Do not have analyze any work in this section.

Discussion:

- Evaluate your test results by explaining
  - Agreement of running time?
  - Limitations? Threshold of m or n?
  - Which factor between m and n was the most attributed factor in running time
  - Ways to improve

Conclusion:

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

3.5. Homework

1. After you program, remember to give the inputs size of vector array, dimension of integer vector array, type of sort, and direction of array.

```
int main(int argc, char* argv[])
{
  random_generator rg;
  timer t;
  int n, m, d, a;

  if (argc > 1)
  {
    m = atoi(argv[1]);
    m = (m < 1) ? 1 : m;
  }
  else
  {
    cout << "hw1: [m] [n] [algo] [direction]" << endl;
    cout << "[m]              size of the random integer vector array" << endl;
    cout << "[n]              dimension of integer vector array" << endl;
    cout << "[algo]           0: run insertion sort" << endl;
    cout << "                 1: run merge sort" << endl;
    cout << "                 <0: run improved insertion sort" << endl;
    cout << "[direction]      0: random" << endl;
    cout << "                 1: sorted" << endl;
    cout << "                 -1: reverse sorted" << endl;
    cout << endl;
    return 0;
  }
```

2. An example of inputs in Xcode.

```
Timer (generate): 1ms real  0ms user    0ms sys
Running insertion sort algorithm:
Timer (sort): 72ms real 70ms user   0ms sys
The output is sorted!
Program ended with exit code: 0
```