Lecture Note 1

CSCI 6470 Algorithms
(Fall 2024)

Liming Cai

School of Computing UGA

August 27, 2024

Chapter 1 Fundamentals

1. Worst-case time complexity
2. The big-O notation
3. Series and recurrence relations
4. Time complexity of recursive algorithms

1. Worst-case time complexity

- **Basic operations**: arithmetic ops, logic ops, assignment, branching in high-level programming language
  - Corresponding operations in assembly (machine) languages and corresponding micro-instructions

    Example: $C = A + B$ is compiled into assembly code and executed with micro-instructions

1. Worst-case time complexity

- **Basic operations**: arithmetic ops, logic ops, assignment, branching in high-level programming language
  - Corresponding operations in assembly (machine) languages and corresponding micro-instructions

    Example: $C = A + B$ is compiled into assembly code and executed with micro-instructions
  - Concept of machine cycle

    real time = the number of machine cycles needed to execute the basic operations in algorithm A

    but the number of machine cycles differ across different computers and system platforms, not suitable for measuring time complexity of algorithms

1. Worst-case time complexity

- **Time** (**complexity**) of an algorithm A on input x is the number of **basic operations** carried out by A on x, denotes as function $t(n, x)$, where n is the **size** of x.
  - instead of the number of machine cycles required for running the basic operations.
  - however, $t(x, n)$ is input x (content)-dependent
- **The worst case time** complexity of algorithm A is function $T(n)$, such that for every $n \geq 0$,

$$\forall x, \text{ of size } n, t(n, x) \leq T(n)$$

independent of the content of input

1. Worst-case time complexity

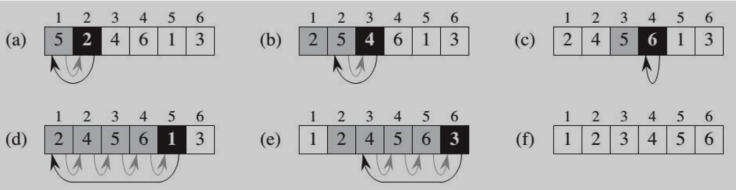
Example 1. Find $T(n)$ for iterative INSERTION SORT algorithm

But first, what is the idea of the insertion sort?

1. Worst-case time complexity

Example 1. Find $T(n)$ for iterative INSERTION SORT algorithm

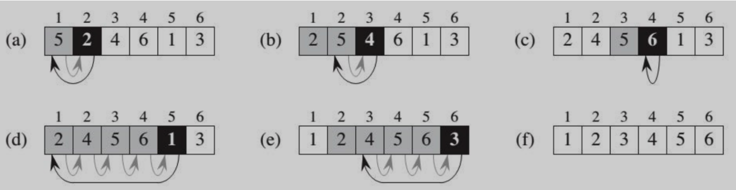
But first, what is the idea of the insertion sort?



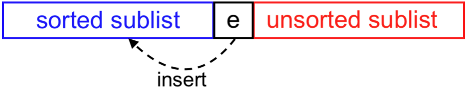
1. Worst-case time complexity

Example 1. Find $T(n)$ for iterative INSERTION SORT algorithm

But first, what is the idea of the insertion sort?



Schematic representation of the dynamic of insertion sort:



1. Worst-case time complexity

Algorithm INSERTION SORT

1. Worst-case time complexity

Algorithm Insertion Sort

```
Function Insertion Sort(L, n);
for i = 2 to n
    e = L[i];
    j = i-1;
    while (L[j] > e) AND (j>0)
        L[j+1] = L[j];
        j = j - 1;
    L[j+1] = e;
return
```

1. Worst-case time complexity

Algorithm Insertion Sort

```
Function Insertion Sort(L, n);
1. for i = 2 to n
2.     e = L[i];
3.     j = i-1;
4.     while (L[j] > e) AND (j>0)
5.         L[j+1] = L[j];
6.         j = j - 1;
7.     L[j+1] = e;
8. return
```

- Count the number of basic operations:

  Line 1: $2 \times n + (n-1)$

  2,3,7: $2 \times (n-1)$

  4: $2 \times t_j$ $\longleftarrow$ t_j dependent on j, overall on input x

  5: $(t_j - 1)$

  6: $2 \times (t_j - 1)$

  Line 8: 1

1. Worst-case time complexity

- Count the number of basic operations:

$$t(n, x) = an + b + \sum_{j=1}^{n-1} c \times t_j$$

for some constants $c > 0$, a, and b

- Because t_j can only be as worst (big) as j

$$t(n, x) \leq an + b + \sum_{j=1}^{n-1} c \times j = T(n)$$

$$T(n) = c\frac{n-1}{2}n + an + b = \frac{c}{2}n^2 + (a - \frac{c}{2})n + b$$

$$= c_1 n^2 + c_2 n + c_3$$

1. Worst-case time complexity

Additional issues

- About time used for arithmetic operations

  e.g., $A + B$, where A and B are of scale $2^{1000000}$;
  time needed is $c \times \frac{1000000}{64} = c_1 \times 1000000$,
  the time complexity is related to the binary length of data

1. Worst-case time complexity

Additional issues

- About n, the **size** of input x, what does **size** refer to?

  (1) size n refers to the number of data items in the input as in INSERTION SORT

  Consider to sort 4 very large elements, e.g., of scale $2^{1000000}$
  $T(n) = c_1 n^2 + c_2 n + c_3$, then $T(4)$ is a small constant time,
  However, this is not an accurate measure because even just comparison of two large elements takes $c \times 1000000$ steps

1. Worst-case time complexity

(2) size n is the number of binary bits that encode the input x, denoted as $n = |x|$

then for INSERTION SORT on m elements, time is bounded by

$$= c_1 m^2 |x| + c_2 m |x| + c_3 |x|$$

$$\leq c_1 n^3 + c_2 n^2 + c_3 n = T(n)$$

Why?

Exercise: given algorithm

```
Function Fibonacci (x);
1. F[1] = 1;
2. F[2] = 1;
3. for i = 3 to x
4.     L[i] = L[i-1] + L[i-2];
5. return L[x];
```

What is $T(n)$ for `Fibonacci`? Is it really a linear function?

1. Worst-case time complexity

Additional issues

(3) How to find a simple upper bound for time expressions?

e.g., worst case time for INSERTION SORT

$$\begin{aligned} T(n) &= c_1 n^2 + c_2 n + c_3 \\ &\leq (c_1 + c_2 + c_3) n^2 \\ &= cn^2 \end{aligned}$$

Exercise: find a simple upper bound for

$$T(n) = 5n^2 + 4n \log_2 n - 20n + 89$$

2. The big-O notation

Needs for a succinct notation for worst time complexity

- articulate the growth of time function
  ignore constant coefficients
- asymptotic of the exact time function

  e.g., intuitively $T(n) = 50n^2 + 4n + 8$ is of $O(n^2)$ because $T(n)$ does not grow faster than some function cn^2.

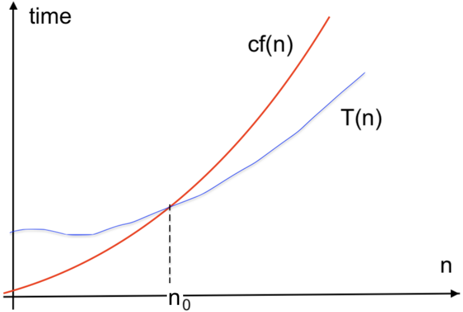
2. The big-O notation

Definition Let $T(n)$ and $f(n)$ be two functions in n. If there exist two constants $c > 0$ and $n_0 \geq 0$, such that

$$T(n) \leq cf(n)$$

for all $n \geq n_0$, then $T(n)$ is said to be of the order of n^2, denoted as

$$T(n) = O(f(n))$$



2. The big-O notation

What is the big-O for function $T(n) = 3n^2 - 20n + 100$?

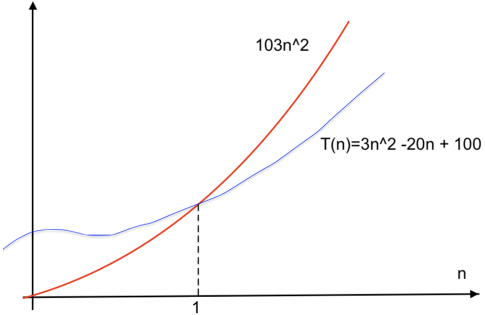
2. The big-O notation

What is the big-O for function $T(n) = 3n^2 - 20n + 100$?

$$\begin{aligned} 3n^2 - 20n + 100 &\leq 3n^2 + 100 \\ &\leq 3n^2 + 100n^2 \text{ when } n \geq 1 \\ &= 103n^2 \end{aligned}$$

We have found $c = 103$ and $n_0 = 1$ such that $T(n) \leq cn^2$.

By definition, $T(n) = O(n^2)$.



2. The big-O notation

- Is there another way to show

$$T(n) = 3n^2 - 20n + 100 = O(n^2) \ ?$$

- Proving big-O (by finding constants c and n_0)

(1) $3n^3 + 2n - 6 = O(?)$

(2) $3n \log_2 n + 5n + 7 \log_2 n = O(?)$

(3) $2^{2n} + 3 \cdot 2^n = O(?)$

(4) $5 \ln n + 7 \log_{10} n + 2 \log_2 n = O(?)$

2. The big-O notation

Summary: deriving big-O for iterative algorithms:

- count the worst-case total number of basic operations, and formulate $T(n) =$ as an expression in n
- rule of thumb: choose $f(n)$ to be the highest-order term in $T(n)$ expression
- identify constants c and n_0 to allow

$$T(n) \leq cf(n), \quad \text{for all } \; n \geq n_0$$

leading to $T(n) = O(f(n))$, by the big-O definition.

Take-home exercises I(A)

1. Write/find pseudo codes for the following algorithms:
   (1) Iterative algorithm that searches a list for a key and returns its index in the list;
   (2) iterative **selection Sort** algorithm;
   (3) iterative algorithm that returns the n^{the} Fibonacci number, given n;
   (4) iterative binary search algorithm;
2. Derive a worst-case time function $T(n)$ from each of the pseudo codes designed in 1.
3. Show the big-O for each of the derived time functions $T(n)$ in 2.
4. Using the big-O definition to prove: If $T_1(n) = O(f(n))$ and $T_2(n) = O(g(n))$, then

   (1) $T_1(n) + T_2(n) = O(f(n) + g(n))$;
   (2) $T_1(n) \times T_2(n) = O(f(n)g(n))$

3. Series and recurrence relations

Series used to compute time complexity

- arithmetic series:

$$\sum_{k=1}^{n} k = 1 + 2 + \cdots + n = \frac{n}{2}(n+1)$$ where did you see it?

- geometric series:

$$\sum_{k=0}^{n} c^k = 1 + c + c^2 + \cdots + c^n = \frac{c^{n+1} - 1}{c - 1}$$ where did you see it?

- harmonic series:

$$\sum_{k=1}^{n} \frac{1}{k} = 1 + \frac{1}{2} + \cdots + \frac{1}{k} = \ln(2n+1)$$

3. Series and recurrence relations

If there is a known formula for a series, use it directly

If no known formula, how to solve them?

- via combinatorics/math methods
- use **recurrence relations**

  e.g., $A_n = 1 + 2 + \cdots + (n-1) + n$, then

$$A_n = \begin{cases} 1 & n = 1 \\ A_{n-1} + n & n \geq 2 \end{cases}$$

We can prove $A_n = \sum_{k=1}^{n} k = 1 + 2 + \cdots + n = \frac{n}{2}(n+1)$ by proving with **math induction**

(1) it works for A_1; and
(2) if it works for A_{n-1}, it also works for A_n.

3. Series and recurrence relations

- It is easy to get recurrence from series

  e.g., $G_n = \sum_{k=0}^{n} c^k = 1 + c + c^2 + \cdots + c^k$

$$G_n = \begin{cases} 1 & n = 0 \\ G_{n-1} + c^n & n \geq 1 \end{cases}$$

  Recurrence helps deriving formula for G_n without even using induction

3. Series and recurrence relations

Time complexities derived from from recursive algorithms often come with recurrences.

```
Function Fib(n);
if ( n = 1) OR (n = 2)
    return 1;
else
    return Fib(n-1) + Fib(n-2)
```

3. Series and recurrence relations

Time complexities derived from from recursive algorithms often come with recurrences.

```
Function Fib(n);
    if ( n = 1) OR (n = 2)
        return 1;
    else
        return Fib(n-1) + Fib(n-2)
```

- assume $T(n)$ to be worst case time for `Fib(n)`, then

$$T(n) = \begin{cases} c_1 & n = 1, 2 \\ T(n-1) + T(n-2) + c_2 & n \geq 3 \end{cases}$$

3. Series and recurrence relations

Time complexities derived from from recursive algorithms often come with recurrences.

```
Function Fib(n);
1.   if ( n = 1) OR (n = 2)
2.       return 1;
3.   else
4.       return Fib(n-1) + Fib(n-2)
```

- assume $T(n)$ to be worst case time for `Fib(n)`, then

$$T(n) = \begin{cases} c_1 & n = 1, 2 \\ T(n-1) + T(n-2) + c_2 & n \geq 3 \end{cases}$$

But does '$+$' take just a constant time?

4. Time complexity of recursive algorithms

Recursive algorithms

- What is recursion?

  (recursive algorithm)
  A process (to solve a problem) consists of steps that are of the same process (recursive algorithm)

  (recursive definition)
  A description (to define a problem) with terms that are of the same description

- Elements in any (meaningful) recursion

  basic element;
  recursive elements;
  changes in "size" of some involved object(s)

4. Time complexity of recursive algorithms

(1) How to creating recursive algorithms
(without predefined recursive formulas to use)

- An algorithm deals with input data and produces output data
- **recursive definition of data** (input or output data) may lead to **recursive algorithms**

4. Time complexity of recursive algorithms

(1) How to creating recursive algorithms
(without predefined recursive formulas to use)

- An algorithm deals with input data and produces output data
- **recursive definition of data** (input or output data) may lead to **recursive algorithms**

  e.g., *a list of n elements consists of a list of $n - 1$ elements concatenated with an additional element*

- this leads to the following recursive **Insertion Sort** algorithm:

  Step 0: if list has single element, return it as it is;
  Step 1: sort the sublist of $(n - 1)$ elements;
  Step 2: insert the last element into the sorted sublist;

4. Time complexity of recursive algorithms

(1) How to creating recursive algorithms

e.g., *a* <u>*sorted list of* n *elements*</u> *consists of a* <u>*sorted list of* $n-1$ *elements*</u> *concatenated with the* <u>*largest element*</u>

4. Time complexity of recursive algorithms

(1) How to creating recursive algorithms

e.g., *a sorted list of n elements consists of a sorted list of $n-1$ elements concatenated with the largest element*

this leads to the following recursive **Selection Sort** algorithm:

Step 0: if list has single element, return it as it is;
Step 1: swap the maximum element with the last element;
Step 2: sort the remaining sublist of $n-1$ elements

4. Time complexity of recursive algorithms

(2) deriving time complexity for recursive algorithms

```
function Selection-Sort(L, n);
1. if n=1 return;
2. k = FindMaxInd(L, n); \\ find index of the max
3. Swap(L, k, n);        \\ swap L[k] with L[n]
4. Selection-Sort(L, n-1);
```

$$T(n) = \begin{cases} c_1 & n = 1 \\ O(n) + c_2 + T(n-1) & n \geq 2 \end{cases}$$

where $O(n)$ is the time for function $FindMaxInd(L, n)$, i.e.,

4. Time complexity of recursive algorithms

(2) deriving time complexity for recursive algorithms

```
function Selection-Sort(L, n);
1. if n=1 return;
2. k = FindMaxInd(L, n); \\ find index of the max
3. Swap(L, k, n);        \\ swap L[k] with L[n]
4. Selection-Sort(L, n-1);
```

$$T(n) = \begin{cases} c_1 & n = 1 \\ O(n) + c_2 + T(n-1) & n \geq 2 \end{cases}$$

where $O(n)$ is the time for function $FindMaxInd(L, n)$, i.e.,

$$T(n) = \begin{cases} c_1 & n = 1 \\ c_3 n + c_2 + T(n-1) & n \geq 2 \end{cases}$$

Take-home exercises I(B)

1. Design a recursive algorithm `FindMaxInd(L, n)` that returns the index of the maximum element in the input list L of n elements.

   Hint: consider a recursive definition of the input list L.

2. Design a recursive algorithm `InsertList(L, k, x)` that inserts a given element of value x into sorted list $L[1..k]$.

   Hint: consider a recursive definition of the input sorted list $L[1..k]$.

3. Design a recursive **Insertion Sort** algorithm that calls the subroutine `InsertList(L, k, x)` designed in 2.

4. Derive a time complexity recurrence for each algorithm in 1, 2, and 3, respectively.

4. Time complexity of recursive algorithms

(3) Solving recurrences to get analytic form

- deriving lower and upper bounds;
- precise solution via unfolding;
- precise solution via math induction (guess and substitute)
- the Master theorem and its proof

4. Time complexity of recursive algorithms

- Driving time lower and upper bounds, e.g., for `Fib(n)`

$$T(n) = \begin{cases} c_1 & n = 1, 2 \\ T(n-1) + T(n-2) + c_2 n & n \geq 3 \end{cases}$$

Lower bound:

$$2T(n-2) \leq T(n), \text{unfolding } \; c_1 2^{\frac{n-1}{2}} \leq T(n)$$

upper bound:

$$T(n) \leq 2T(n-1) + c2n, \text{unfolding } \; T(n) \leq c_1 2^{n-2} + \sum_{k=3}^{n} k$$

So $T(n) = O(2^n)$ and $T(n) = \Omega(2^{n/2})$ (will discuss Ω later)

4. Time complexity of recursive algorithms

- Precise solutions via unfolding

  Method 1: summation over (in)equalities

  example-1 (Insertion Sort):

$$T(n) = \begin{cases} a & n = 1 \\ T(n-1) + bn & n \geq 2 \end{cases}$$

  example-2 (Binary Search)

$$T(n) = \begin{cases} a & n = 0 \\ T(\lfloor n/2 \rfloor) + b & n \geq 1 \end{cases}$$

4. Time complexity of recursive algorithms

Method 2: recursive trees

example-3 (Unbalanced Merge Sort):

$$T(n) = \begin{cases} a & n \geq 1 \\ T(\lfloor 2n/3 \rfloor) + T(\lceil n/3 \rceil) + bn & n \geq 2 \end{cases}$$

will be discussed in Note 2.

4. Time complexity of recursive algorithms

- Precise solutions via math induction

The math induction

To prove some property $\mathcal{P}(n)$ holds for all integers $n \geq b$, where $b \geq 0$ is some fixed integer, it suffices to prove

(1) $\mathcal{P}(b)$ is true, i.e., property $\mathcal{P}$ holds for the base case b;

(2) $\mathcal{P}(k) \longrightarrow \mathcal{P}(k+1)$ is true, for every $k \geq b$, i.e., if $\mathcal{P}$ holds for k, it also holds for $k+1$.

4. Time complexity of recursive algorithms

- Precise solutions via math induction

The math induction

To prove some property $\mathcal{P}(n)$ holds for all integers $n \geq b$, where $b \geq 0$ is some fixed integer, it suffices to prove

(1) $\mathcal{P}(b)$ is true, i.e., property $\mathcal{P}$ holds for the base case b;

(2) $\mathcal{P}(k) \longrightarrow \mathcal{P}(k+1)$ is true, for every $k \geq b$, i.e., if $\mathcal{P}$ holds for k, it also holds for $k+1$.

The **strong** math induction

(2) is replaced with $\mathcal{P}(b), \mathcal{P}(b+1), \ldots, \mathcal{P}(k) \longrightarrow \mathcal{P}(k+1)$ is true

4. Time complexity of recursive algorithms

Example: *Let sum $S_n = 1 + 2 + \cdots + n$. Using math induction to prove property $\mathcal{P}(n)$ holds for all $n \geq 1$:*

$$S_n = \frac{n}{2}(n+1)$$

Proof:

base case: for $n = 1$, $S_1 = 1$, while $\frac{n}{2}(n+1) = \frac{1}{2}(1+1) = 1$, the desired equation holds for $n = 1$;

assumption: $S_k = \frac{k}{2}(k+1)$, the equation holds $n = k$,
induction: for $n = k + 1$,

$$\begin{aligned} S_{k+1} &= 1 + 2 + \cdots + k + (k+1) = S_k + (k+1) \\ &= \frac{k}{2}(k+1) + (k+1) \text{ by the assumption} \\ &= (k+1)\Big(\frac{k}{2} + 1\Big) \\ &= \frac{(k+1)}{2}\Big((k+1)+1\Big), \text{ the equation holds for } n = k+1 \end{aligned}$$

4. Time complexity of recursive algorithms

Example: *Let sum* $S_n = 1 + 2 + \cdots + n$*. Using math induction to prove property* $\mathcal{P}(n)$ *holds for all* $n \geq 1$*:*

$$S_n = \frac{n}{2}(n+1)$$

Proof:

<u>base case</u>: for $n = 1$, $S_1 = 1$, while $\frac{n}{2}(n+1) = \frac{1}{2}(1+1) = 1$, the desired equation holds for $n = 1$;

<u>assumption</u>: $S_k = \frac{k}{2}(k+1)$, the equation holds $n = k$,
<u>induction</u>: for $n = k + 1$,

$$\begin{aligned}
S_{k+1} &= 1 + 2 + \cdots + k + (k+1) = S_k + (k+1) \\
&= \frac{k}{2}(k+1) + (k+1) \text{ by the assumption} \\
&= (k+1)\Big(\frac{k}{2} + 1\Big) \\
&= \frac{(k+1)}{2}\Big((k+1) + 1\Big), \text{ the equation holds for } n = k+1
\end{aligned}$$

In the induction part, which steps are critical?

4. Time complexity of recursive algorithms

Solving recurrence relations with math induction

E.g., given

$$T(n) = \begin{cases} a & n \geq 1 \\ T(n-1) + bn & n \geq 2 \end{cases}$$

we can use math induction to prove **that there is a constant** $c > 0$**, such that**

$$T(n) \leq cn^2, \quad \textbf{for all } n \geq 1$$

4. Time complexity of recursive algorithms

Solving recurrence relations with math induction

E.g., given

$$T(n) = \begin{cases} a & n \geq 1 \\ T(n-1) + bn & n \geq 2 \end{cases}$$

we can use math induction to prove **that there is a constant** $c > 0$**, such that**

$$T(n) \leq cn^2, \quad \textbf{for all } n \geq 1$$

(1) What is the property $\mathcal{P}$ to be proved here?
(2) What do we need to prove with the math induction method?

4. Time complexity of recursive algorithms

Proof:

<u>basis</u>: $n = 1$, $T(n) = T(1) = a$, to allow $a \leq c \times 1^2$, it suffice to choose $c \geq a$;

<u>assumption</u>: for $k \geq 1$, $T(k) \leq c \times k^2$ for some $c \geq a$;

<u>induction</u>: for $k + 1 \geq 2$,

$$\begin{aligned} T(k+1) &= T(k) + b(k+1) \\ &\leq ck^2 + b(k+1) \\ &= c(k^2 + 2k + 1) - 2ck - c + bk + b \\ &= c(k+1)^2 - (2c-b)k - (c-b) \\ &\leq c(k+1)^2 \text{ when } c \geq b \end{aligned}$$

We have proved that there exists constant $c = \max\{a, b\}$ such that

$$T(n) \leq cn^2 \text{ for all } n \geq 1$$

4. Time complexity of recursive algorithms

Master theorem:

Consider a general recurrence

$$T(n) = \begin{cases} c & n \le n_0 \\ aT(\frac{n}{b}) + f(n) & n > n_0 \end{cases}$$

The so-called Master theorem gives the big-O (actually big-Θ) for $T(n)$ under various a, b, and $f(n)$.

It is actually easy to derive those results by ourselves, using the introduced unfolding technique!

4. Time complexity of recursive algorithms

Master theorem:

Consider a general recurrence

$$T(n) = \begin{cases} c & n \le n_0 \\ aT(\frac{n}{b}) + f(n) & n > n_0 \end{cases}$$

The so-called Master theorem gives the big-O (actually big-Θ) for $T(n)$ under various a, b, and $f(n)$.

It is actually easy to derive those results by ourselves, using the introduced unfolding technique!

E.g., $a = 8$, $b = 2$, $f(n) = 100n^2$.
The Master theorem says $T(n) = O(n^3)$

We can use unfolding to show it too!

4. Time complexity of recursive algorithms

Merge Sort:

```
function MergeSort(L, low, high);
if low < high
    mid = floor((low + high)/2);
    MergeSort(L, low, mid);
    MergeSort(L, mid+1, high);
    MergeTwo(L, low, mid, high);
    return;
```

4. Time complexity of recursive algorithms

Merge Sort:

```
function MergeSort(L, low, high);
if low < high
    mid = floor((low + high)/2);
    MergeSort(L, low, mid);
    MergeSort(L, mid+1, high);
    MergeTwo(L, low, mid, high);
    return;
```

Let $n = \texttt{high} - \texttt{low} + 1$, a power of 2, and
$T(n)$ be the worst case time for `MergeSort(L, low, high)`

Then

$$T(n) = \begin{cases} a & n \leq 1 \\ 2T(\frac{n}{2}) + +O(n) & n \geq 2 \end{cases}$$

where $O(n)$ is the time used in `MergeTwo(L, low, mid, high)`.