香港中文大學(深圳)
The Chinese University of Hong Kong

CSC3100 Data Structures
Lecture 4: Insertion sort, merge sort

Yixiang Fang
School of Data Science (SDS)
The Chinese University of Hong Kong, Shenzhen

Outline

- Use array to solve the sorting problem

- Insertion sort
  - Recursion
  - Algorithm analysis

- Merge sort
  - Divide and conquer
  - Algorithm analysis

Paradigms of algorithm design

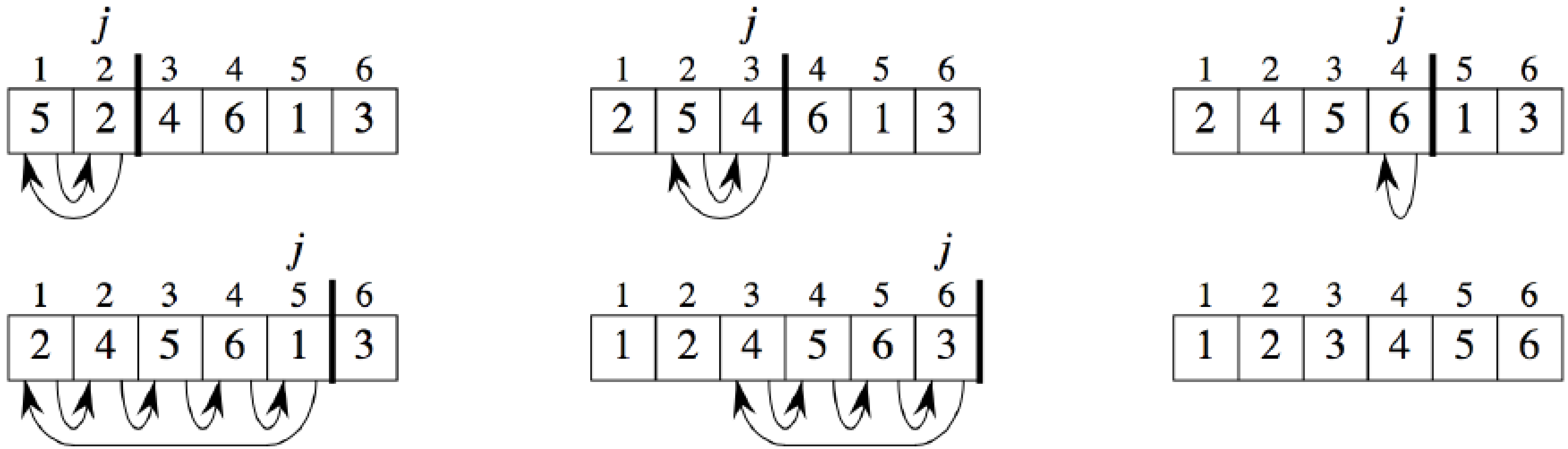
The sorting problem

- 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 input such that $a'_1 <= a'_2 <= \ldots <= a'_n$
  - Stored in arrays
  - The numbers are referred as keys
- Many sorting algorithms

Insertion sort

- A simple algorithm for <u>a small number of elements</u>
- Similar to sort a hand of playing card
  - Start with an empty left hand
  - Pick up one card and insert it into the correct position
  - To find the correct position, compare it with each of the cards in the hand, from right to left
  - The cards held in the left hand are sorted

Example of insertion sort



Insertion sort pseudocode

```
INSERTION-SORT(A)
for j ← 2 to 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
```

Correctness: loop invariant

- A property of the loop: loop invariant
  - Insertion sort: in each iteration, the array $A[1,\ldots,j-1]$ is sorted
- Help us prove the correctness of the algorithm
  - Initialization: true before the begin of loop
  - Maintenance: if true before an iteration, then also true after it
  - Termination: when the loop stops, use the invariant to show the algorithm is correct
- Similar to the mathematical induction

Correctness: loop invariant

INSERTION-SORT(A)

for $j \leftarrow 2$ **to** n Initialization

do $key \leftarrow A[j]$

M A I N T E N A N C E

$\triangleright$ Insert $A[j]$ into the sorted sequence $A[1\,..\,j-1]$.

$i \leftarrow j - 1$

while $i > 0$ and $A[i] > key$

do $A[i+1] \leftarrow A[i]$

$i \leftarrow i - 1$

$A[i+1] \leftarrow key$

Endfor

Termination

Loop invariant: insertion sort

- Proof:
  - Initialization: true before the begin of loop
    Only one element A[1]
  - Maintenance: true before an iteration and after it
    A[j] is in the correct position j' ⇔ A[j'-1]<= A[j']<= A[j'+1]
  - Termination: when the loop stops, use the loop invariant to show the algorithm is correct
    j = n when loop stops, A[1,...,j-1] is sorted

How to analyze running time?

- Random-access machine (RAM) model
  - Sequential and no concurrent operations
  - Operations taking a constant amount of time:
    - E.g., arithmetic, data movement, conditions, function all, etc.

- For a given input, the time cost can be measured by the number of primitive operations (steps) executed

- Each line of pseudocode is composed of some numbers of operations and therefore requires a constant amount of time
  - One line may take a different amount of time than another

Analysis of insertion sort

INSERTION-SORT(A)	*cost*	*times*
for $j \leftarrow 2$ **to** n	c_1	n
do $key \leftarrow A[j]$	c_2	$n-1$
$\triangleright$ Insert $A[j]$ into the sorted sequence $A[1 \,..\, j-1]$.	0	$n-1$
$i \leftarrow j-1$	c_4	$n-1$
while $i > 0$ and $A[i] > key$	c_5	$\sum_{j=2}^{n} t_j$
do $A[i+1] \leftarrow A[i]$	c_6	$\sum_{j=2}^{n}(t_j-1)$
$i \leftarrow i-1$	c_7	$\sum_{j=2}^{n}(t_j-1)$
$A[i+1] \leftarrow key$	c_8	$n-1$

$$
\begin{aligned}
T(n) \;=\; & c_1 n + c_2(n-1) + c_4(n-1) + c_5 \sum_{j=2}^{n} t_j + c_6 \sum_{j=2}^{n} (t_j - 1) \\
& + c_7 \sum_{j=2}^{n} (t_j - 1) + c_8(n-1)\,.
\end{aligned}
$$

$T(n)$ depends on n and t_j

Analysis of insertion sort

- Best case: the array is sorted

  => t_j=*1*

$$
\begin{aligned}
T(n) &= c_1 n + c_2(n-1) + c_4(n-1) + c_5(n-1) + c_8(n-1) \\
&= (c_1 + c_2 + c_4 + c_5 + c_8)n - (c_2 + c_4 + c_5 + c_8)\,.
\end{aligned}
$$

- Worst case: the array is in reverse order

  => t_j= *j*

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

When talking about best/worst case, the algorithm itself should be able to handle all the cases

Analysis of insertion sort

- Worse case (con't)

$$
\begin{aligned}
T(n) &= c_1 n + c_2(n-1) + c_4(n-1) + c_5\left(\frac{n(n+1)}{2} - 1\right) \\
&\quad + c_6\left(\frac{n(n-1)}{2}\right) + c_7\left(\frac{n(n-1)}{2}\right) + c_8(n-1) \\
&= \left(\frac{c_5}{2} + \frac{c_6}{2} + \frac{c_7}{2}\right) n^2 + \left(c_1 + c_2 + c_4 + \frac{c_5}{2} - \frac{c_6}{2} - \frac{c_7}{2} + c_8\right) n \\
&\quad - (c_2 + c_4 + c_5 + c_8) .
\end{aligned}
$$

Can express $T(n)$ as $an^2 + bn + c$ for constants a, b, c (that again depend on statement costs) $\Rightarrow T(n)$ is a *quadratic function* of n.

Analysis of insertion sort

- Concentrate on the worst-case running time
  - Give a guaranteed upper bound for any input
  - For some algorithms, the worst case occurs often
    - For example, search for absent items
  - Why not analyze the average case?
    - Because it is often as bad as the worst case
- On average, A[j] is less than half of A[1,…,j-1] => $t_j = j/2$
  - The average case is about half of the worse case but still a quadratic of *n*
  - Note: when comparing the complexity, we only keep the higher-order term, e.g., n^2 *vs* $1000n+10000$

Recursion

- What is recursion?
  - self-reference
  - recursive function: based upon itself
  - Solution of the whole problem is composed of solutions of sub-problems

$$f(x) = 2f(x-1) + x^2$$

```
public int f( int x ){
        if ( x == 0 )
                return 0;
        else
                return 2 * f( x - 1 ) + x*x;
}
```

Recursion

- Characteristics of a recursive definition
  - It has a stopping point (base case)
  - It recursively evaluates an expression involving a variable *n* from a higher value to a lower value of *n*
  - Base case must be reached

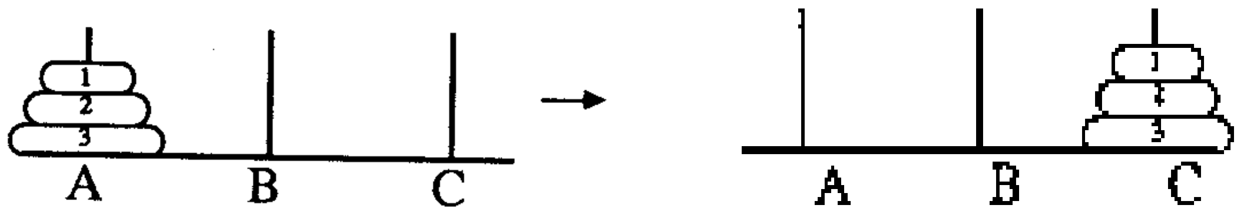
```
public static int bad (int N)
{
   if (N == 0)
         return 0;
   else
         return bad (N / 3 + 1) + N - 1;
}
```

Recursion: insertion sort

- Base Case: If array size is 1 or smaller, return
- Recursively sort first n-1 elements
- Insert last element at its correct position in sorted array

Recursion: Tower of Hanoi

- Problem:
  - It consists of three rods and a number of disks of different diameters, which can slide onto any rod



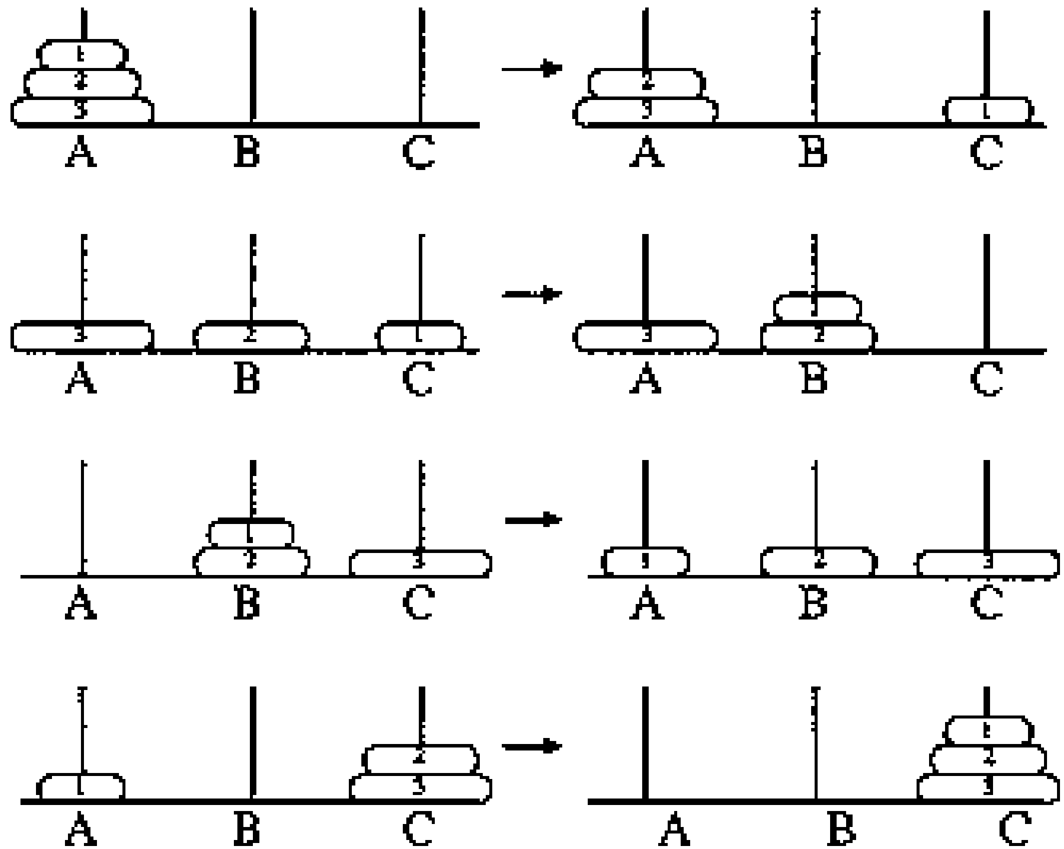
- Constraints:

(1) only one disk can be moved at a time, and

(2) at no time may a disk be placed on top of a smaller disk

Recursion: Tower of Hanoi

- N=3



Recursion: Tower of Hanoi

- Solution
  - If n = 1, move the single disk from A to C and stop;
  - Otherwise, move the top n-1 disks from A to B, using C as auxiliary,
  - Move the remaining disk from A to C,
  - Move the n-1 disks from B to C, using A as auxiliary

Alternative sorting algorithm

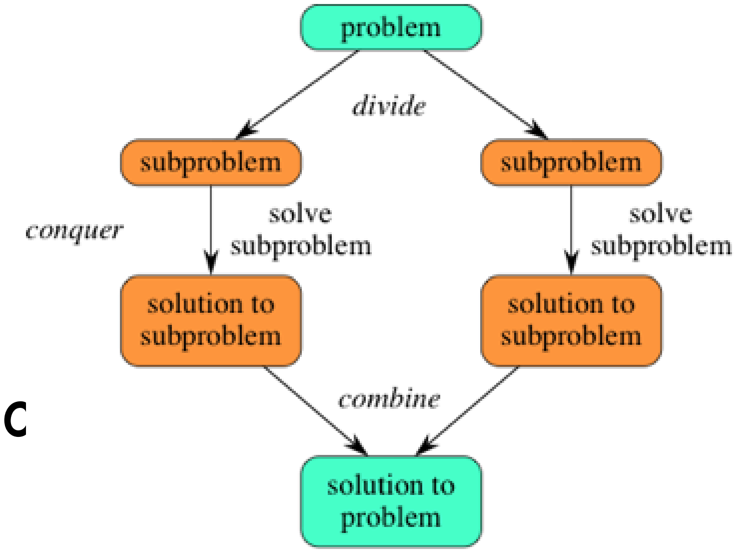
- Many ways to sort
- Insertion sort is incremental: having sorted $A[1,\ldots,j-1]$, place $A[j]$ correctly, so that $A[1,\ldots,j]$ is sorted.
- Another common approach: **divide and conquer**

Divide and conquer

- **Divide** the problem into a number of subproblems

- **Conquer** the subproblems by solving them recursively (further divide if nc small enough)
  - **Base case**: If the subproblems are small enough, may solve them by brute force

- **Combine** the subproblem solutions to give a solution to the original problem



Merge sort

- A sorting algorithm based on divide and conquer

- Its worst-case running time has a lower order of growth rate than insertion sort

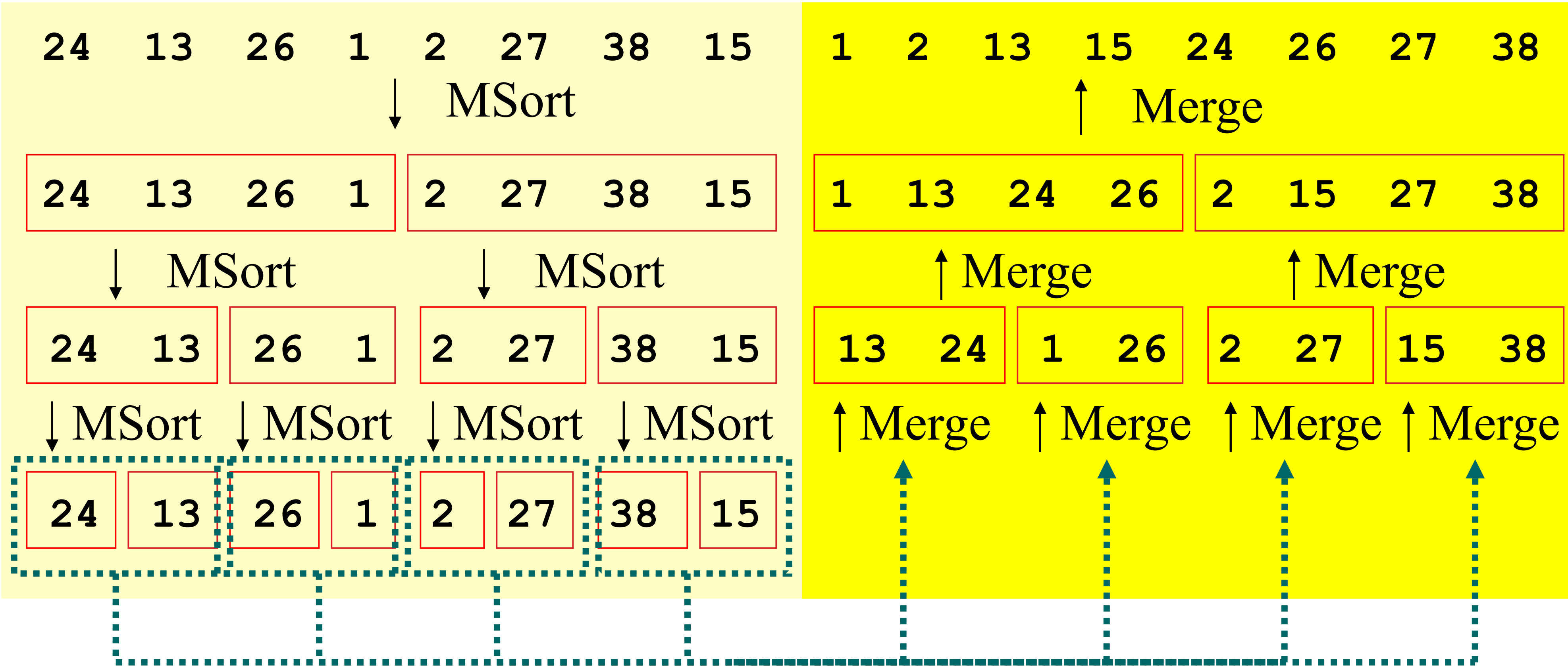
- Each subproblem is to sort a subarray A[p,...,r].
  - p=1, r=n at the start and changes during splitting

To sort A[p,…,r]

- Algorithm steps
  - Divide it into two subarrays A[p,…,q] and A[q+1,…,r], where q is the middle point
  - Conquer by recursively sorting the two subarrays A[p,…,q] and A[q+1,…,r]
  - Merge the two sorted subarrays A[p,…,q] and A[q+1,…,r]

```
MERGE-SORT(A, p, r)
if p < r                              ▷ Check for base case
  then q ← ⌊(p + r)/2⌋                ▷ Divide
       MERGE-SORT(A, p, q)            ▷ Conquer
       MERGE-SORT(A, q + 1, r)        ▷ Conquer
       MERGE(A, p, q, r)              ▷ Combine
```

Example: $n = 8$



24 13 26 1 2 27 38 15
MSort
24 13 26 1
2 27 38 15
MSort
MSort
24 13
26 1
2 27
38 15
MSort
MSort
MSort
MSort
24
13
26
1
2
27
38
15
1 2 13 15 24 26 27 38
Merge
1 13 24 26
2 15 27 38
Merge
Merge
13 24
1 26
2 27
15 38
Merge
Merge
Merge
Merge

Merge

- Merge ordered subarray A[p,…,q] and ordered subarray A[q+1,…,r]

- How to efficiently implement it?
  - Think of two piles of cards.
  - Each pile is sorted and placed face-up on a table with the smallest cards on top.
  - We will merge them into a single sorted pile.
  - Basic idea
    - Choose the smaller of the two top cards
    - Remove it from its pile
    - Repeat

Merge: example

1 13 24 26 2 15 27 38

↑ ↑ ↑

Aptr *Bptr* *Cptr*

1 13 24 26 2 15 27 38 1

↑ ↑ ↑

Aptr *Bptr* *Cptr*

1 13 24 26 2 15 27 38 1 2

↑ ↑ ↑

Aptr *Bptr* *Cptr*

Merge: example

1 13 24 26 2 15 27 38 1 2 13

↑ ↑ ↑

Aptr *Bptr* *Cptr*

1 13 24 26 2 15 27 38 1 2 13 15

↑ ↑ ↑

Aptr *Bptr* *Cptr*

1 13 24 26 2 15 27 38 1 2 13 15 24

↑ ↑ ↑

Aptr *Bptr* *Cptr*

Implementation of merge sort

```
public static void mergeSort(int[] a) {
    int[] tmpArray = new int[a.length];
    mergeSort(a, tmpArray, 0, a.length - 1);
}

private static void mergeSort(int[] a, int[] tmpArray, int left, int right) {
    if (left < right) {
            int center = (left + right) / 2;
            mergeSort(a, tmpArray, left, center);
            mergeSort(a, tmpArray, center + 1, right);
            merge(a, tmpArray, left, center + 1, right);
    }
}
```

Implementation of merge sort

```
private static void merge(int[] a, int[] tmpArray, int leftPos, int rightPos, int rightEnd){
    int leftEnd = rightPos – 1, tmpPos = leftPos;
    int numElements = rightEnd - leftPos + 1;

    while (leftPos <= leftEnd && rightPos <= rightEnd)
        if (a[leftPos] <= a[rightPos])
                tmpArray[tmpPos++] = a[leftPos++];
        else
                tmpArray[tmpPos++] = a[rightPos++];

    while (leftPos <= leftEnd)
        tmpArray[tmpPos++] = a[leftPos++];

    while (rightPos <= rightEnd)
        tmpArray[tmpPos++] = a[rightPos++];

    for (int i = 0; i < numElements; i++, rightEnd--)
        a[rightEnd] = tmpArray[rightEnd];
}
```

Analyzing merge sort

- Suppose N is a power of 2

$$T(n) = \begin{cases} c & \text{if } n = 1\,, \\ 2T(n/2) + cn & \text{if } n > 1\,. \end{cases}$$

$$T(1) = C$$

$$T(N) = 2\,T(N/2) + CN$$

$$\frac{T(N)}{N} = \frac{T(N/2)}{N/2} + C = \ldots = \frac{T(1)}{1} + C\log N$$

$$T(N) = CN\log N + CN = O(N\log N)$$

Compare with insertion sort

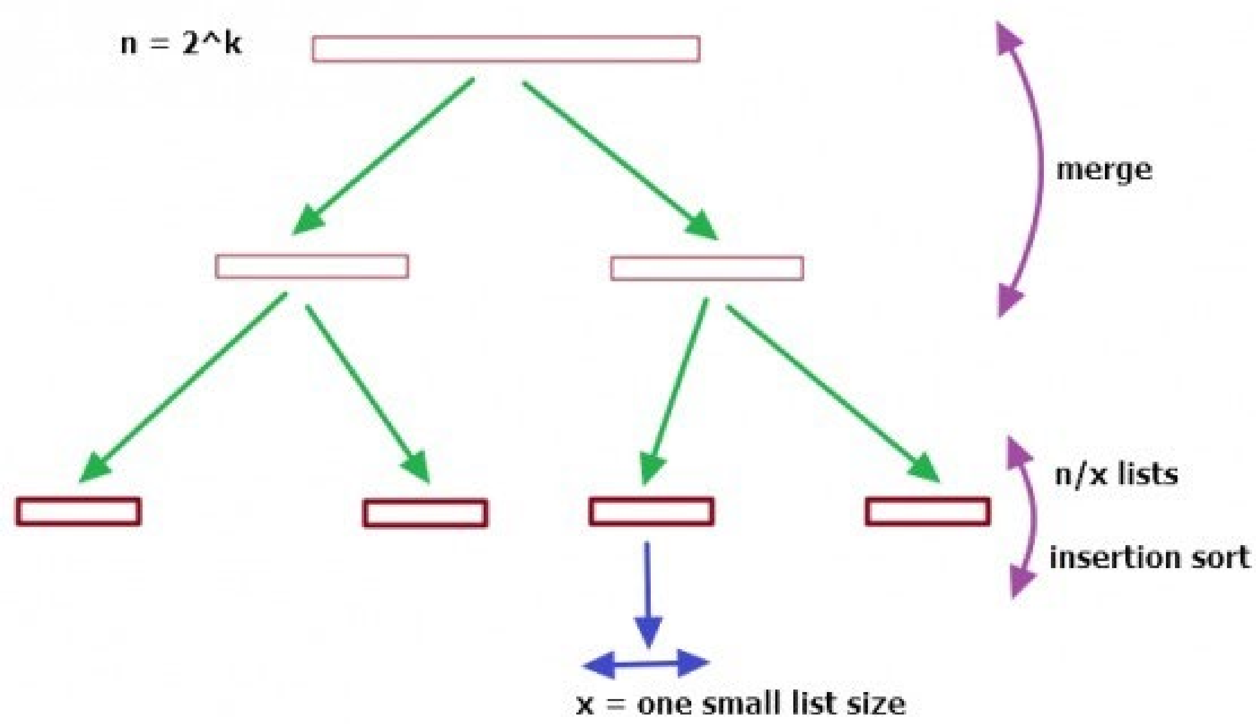
- Compared to insertion sort (worst-case time is a quadratic of n), merge sort is faster

- On small inputs, insertion sort may be faster, but for large enough inputs, merge sort will always be faster

- What is your thinking now?

Exercise

- Implement a hybrid sorting algorithm combining merge sort and selection sort



Recommended reading

- Reading this week
  - Chapter 2, textbook
- Next lecture
  - Complexity analysis: chapter 3, textbook