Getting started: insertion sort, merge sort, algorithm analysis

Congduan Li

Chinese University of Hong Kong, Shenzhen

congduan.li@gmail.com

Sep 5 & 7, 2017

Algorithms and data structures

- Algorithm: well defined computation procedure taking input and producing output
- Algorithms can be used to solve some specific problems
- Algorithms deal with data
- Data needs be stored, accessed, and modified
- The ways to store and organize data to facilitate access and modifications $\rightarrow$ *data structures*
- They are two key factors to determine the efficiency of solving a problem
- This course: learn some basic and important algorithms and data structures, and analysis on them

Goals of this week:

- Start using frameworks for describing and analyzing algorithms.
- Examine two algorithms for sorting: insertion sort and merge sort.
- See how to describe algorithms in pseudocode.
- Begin using asymptotic notation to express running-time analysis.
- Learn the technique of "divide and conquer" in the context of merge sort.

The sorting problem

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

- The sequences are typically stored in arrays.
- We also refer to the numbers as *keys*.
- Along with each key may be additional information, known as *satellite data*.
- We will see several ways to solve the sorting problem.

Expressing algorithms

- We express algorithms in *pseudocode*
- Pseudocode is similar to C, C++, Python, and Java. If you know any of these languages, you should be able to understand pseudocode.
- Pseudocode is designed for expressing algorithms to humans. Software engineering issues of data abstraction, modularity, and error handling are often ignored.
- We sometimes embed English statements into pseudocode.
- Therefore, unlike for real programming languages, we cannot create a compiler that translates pseudocode to machine code.

Insertion sort

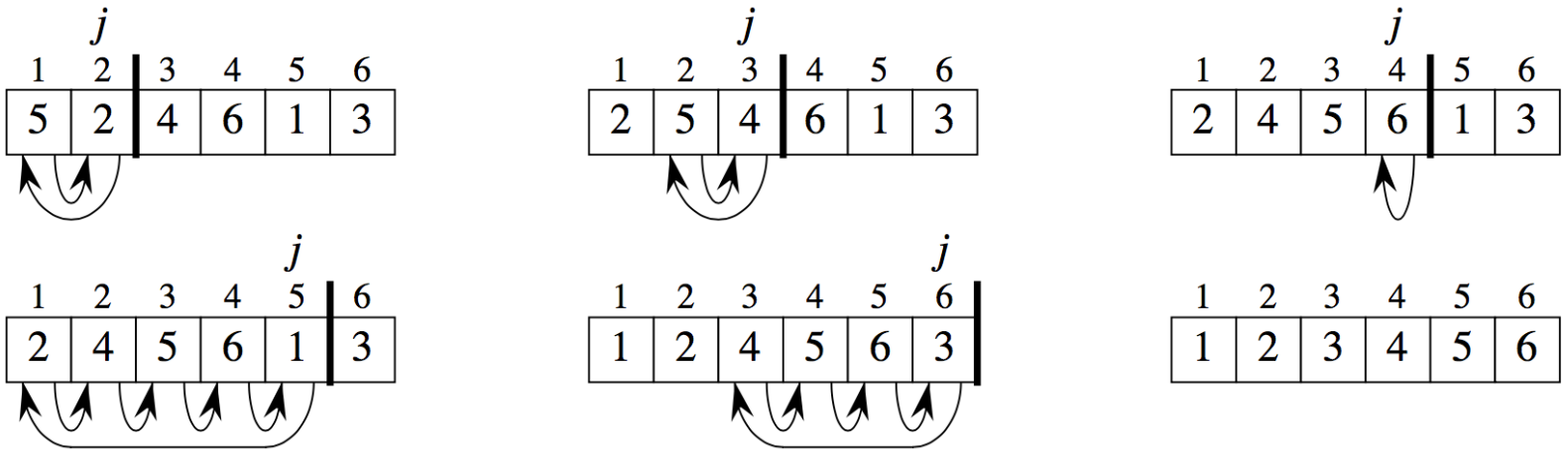
A good algorithm for sorting a small number of elements.
It works the way you might sort a hand of playing cards:

- Start with an empty left hand and the cards face down on the table.
- Then pick up one card at a time from the table, and insert it into the correct position in the left hand.
- To find the correct position for a card, compare it with each of the cards already in the hand, from right to left.
- At all times, the cards held in the left hand are sorted, and these cards were originally the top cards of the pile on the table.

Pseudocode: We use a procedure INSERTION-SORT

```
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
```

Example: INSERTION-SORT



Correctness: loop invariant

For the insertion sort algorithm: each outer loop step j, the sequence $A[1, \ldots, j-1]$ is sorted.
In general, we need three steps to show 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–usually along with the reason that the loop terminated–gives us a useful property that helps show that the algorithm is correct.

Loop invariant: similar as mathematical induction

- To prove that a property holds, you prove a base case and an inductive step.
- Showing that the invariant holds before the first iteration is like the base case.
- Showing that the invariant holds from iteration to iteration is like the inductive step.
- The termination part differs from the usual use of mathematical induction, in which the inductive step is used infinitely.
- We stop the "induction" when the loop terminates.
- We can show the three parts in any order.

Correctness analysis for insertion sort

- **Initialization:** Just before the first iteration, $j = 2$. The subarray $A[1..j-1]$ is the single element $A[1]$, which is the element originally in $A[1]$, and it is trivially sorted.
- **Maintenance:** To be precise, we would need to state and prove a loop invariant for the "inner" while loop. Rather than getting bogged down in another loop invariant, we instead note that the body of the inner while loop works by moving $A[j-1], A[j-2], A[j-3]$, and so on, by one position to the right until the proper position for key (which has the value that started out in $A[j]$) is found. At that point, the value of key is placed into this position.
- **Termination:** The outer for loop ends when $j > n$; this occurs when $j = n + 1$. Therefore, $j - 1 = n$. Plugging n in for $j - 1$ in the loop invariant, the subarray $A[1..n]$ consists of the elements originally in $A[1..n]$ but in sorted order. In other words, the entire array is sorted!

Analyzing algorithms: running time

Random-access machine (RAM) model:

- Instructions are executed one after another. No concurrent operations.
- It is too tedious to define each of the instructions and their associated time costs.
- Instead, we recognize that we'll use instructions commonly found in real computers:
  - Arithmetic: add, subtract, multiply, divide, remainder, floor, ceiling). Also, shift left/shift right (good for multiplying/dividing by 2^k).
  - Data movement: load, store, copy.
  - Control: conditional/unconditional branch, subroutine call and return.
  - Each of these instructions takes a constant amount of time.
- Uses integer and floating-point types.

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'll see that insertion sort takes less time to sort n elements when they are already sorted than when they are in reverse sorted order.

How do we analyze an algorithm's running time?

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.

How do we analyze an algorithm's running time?

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."

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	$n-1$

Analysis of insertion sort

- Assume that the ith line takes time c_i, which is a constant. (Since the third line is a comment, it takes no time.)
- For $j = 2, 3, \ldots, n$, let t_j be the number of times that the **while** loop test is executed for that value of j.
- Note that when a **for** or **while** loop exits in the usual way—due to the test in the loop header—the test is executed one time more than the loop body.

The running time of the algorithm is

$$\sum_{\text{all statements}} (\text{cost of statement}) \cdot (\text{number of times statement is executed}) \ .$$

Let $T(n) =$ running time of INSERTION-SORT.

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

The running time depends on the values of t_j. These vary according to the input.

Analysis of insertion sort

Best case: The array is already sorted.

- Always find that $A[i] \leq key$ upon the first time the **while** loop test is run (when $i = j - 1$).
- All t_j are 1.
- Running time is

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

- Can express $T(n)$ as $an + b$ for constants a and b (that depend on the statement costs c_i) $\Rightarrow$ $T(n)$ is a *linear function* of n.

Analysis of insertion sort

Worst case: The array is in reverse sorted order.

- Always find that $A[i] > key$ in while loop test.
- Have to compare *key* with all elements to the left of the jth position $\Rightarrow$ compare with $j - 1$ elements.
- Since the while loop exits because i reaches 0, there's one additional test after the $j - 1$ tests $\Rightarrow t_j = j$.
- $\displaystyle\sum_{j=2}^{n} t_j = \sum_{j=2}^{n} j$ and $\displaystyle\sum_{j=2}^{n} (t_j - 1) = \sum_{j=2}^{n} (j - 1)$.
- $\displaystyle\sum_{j=1}^{n} j$ is known as an ***arithmetic series***, and equation (A.1) shows that it equals $\dfrac{n(n+1)}{2}$.
- Since $\displaystyle\sum_{j=2}^{n} j = \left(\sum_{j=1}^{n} j\right) - 1$, it equals $\dfrac{n(n+1)}{2} - 1$.

Analysis of insertion sort: worst case

- Letting $k = j - 1$, we see that $\sum_{j=2}^{n}(j-1) = \sum_{k=1}^{n-1} k = \frac{n(n-1)}{2}$.
- Running time is

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

- 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.

Worst-case and average-case analysis

We usually concentrate on finding the worst-case running time: the longest running time for any input of size n.

Reasons:

- The worst-case running time gives a guaranteed upper bound on the running time for any input.
- For some algorithms, the worst case occurs often. For example, when searching, the worst case often occurs when the item being searched for is not present, and searches for absent items may be frequent.
- Why not analyze the average case? Because it is often about as bad as the worst case.

Example

Example: Suppose that we randomly choose n numbers as the input to insertion sort.

On average, the key in $A[j]$ is less than half the elements in $A[1 \,..\, j-1]$ and it's greater than the other half.
$\Rightarrow$ On average, the **while** loop has to look halfway through the sorted subarray $A[1 \,..\, j-1]$ to decide where to drop *key*.
$\Rightarrow t_j = j/2$.

Although the average-case running time is approximately half of the worst-case running time, it's still a quadratic function of n.

Order of growth

- Another abstraction to ease analysis and focus on the important features.
- Look only at the leading term of the formula for running time. Drop lower-order terms.
- Ignore the constant coefficient in the leading term.

Order of growth examples

- For insertion sort, we already abstracted away the actual statement costs to conclude that the worst-case running time is $an^2 + bn + c$.
- Drop lower-order terms $\Rightarrow an^2$.
- Ignore constant coefficient $\Rightarrow n^2$.
- But we cannot say that the worst-case running time $T(n)$ equals n^2. It grows like n^2. But it does not equal n^2.
- We say that the running time is $\Theta(n^2)$ to capture the notion that the order of growth is n^2.
- We usually consider one algorithm to be more efficient than another if its worst-case running time has a smaller order of growth.

Designing algorithms

- There are many ways to design algorithms.
- For example, insertion sort is incremental: having sorted $A[1..j-1]$, place $A[j]$ correctly, so that $A[1..j]$ 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.

Base case: If the subproblems are small enough, just 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 than insertion sort. $O(n^2) \rightarrow O(n \log n)$
- Because we are dealing with subproblems, we state each subproblem as sorting a subarray $A[p..r]$.
- Initially, $p = 1$ and $r = n$, but these values change as we recurse through subproblems.

To sort $A[p..r]$

Divide by splitting into two subarrays $A[p..q]$ and $A[q+1..r]$, where q is the halfway point of $A[p..r]$.

Conquer by recursively sorting the two subarrays $A[p..q]$ and $A[q+1..r]$.

Combine by merging the two sorted subarrays $A[p..q]$ and $A[q+1..r]$ to produce a single sorted subarray $A[p..r]$. To accomplish this step, we'll define a procedure $MERGE(A, p, q, r)$.

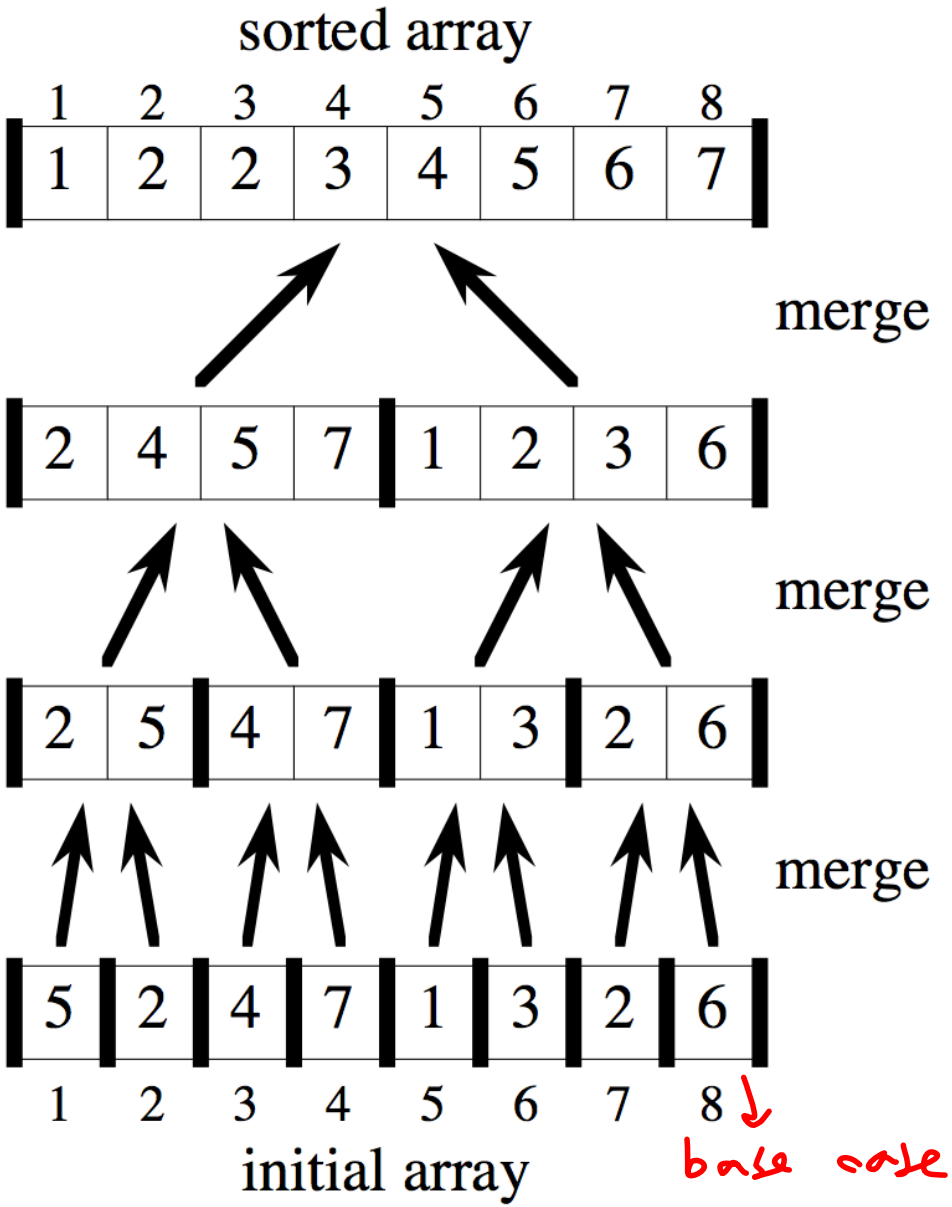
Merge sort code

The recursion bottoms out when the subarray has just 1 element, so that it is trivially sorted.

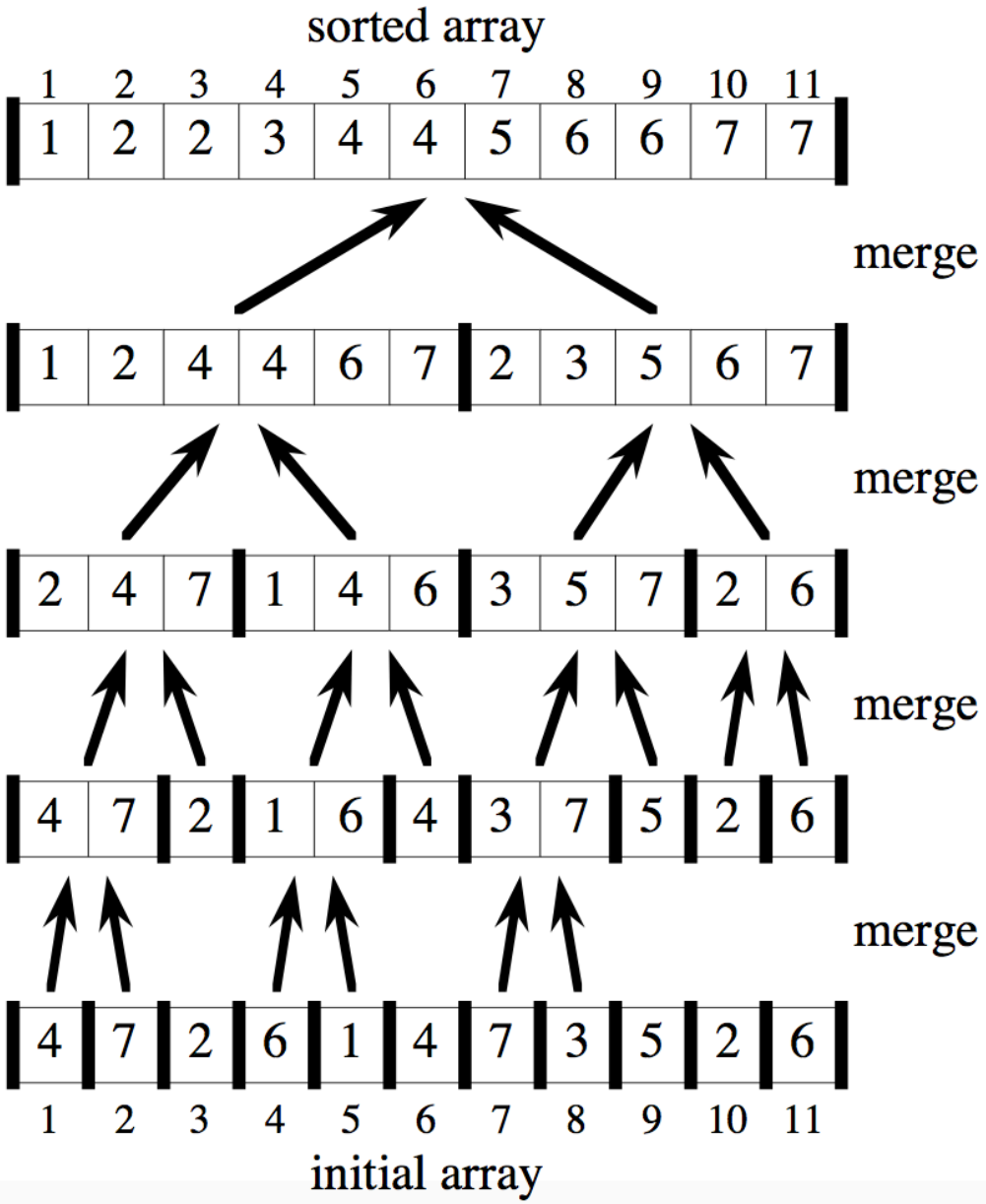
```
MERGE-SORT(A, p, r)
if p < r                                  ▷ Check for base case
  then q ← ⌊(p + r)/2⌋                    ▷ Divide
       MERGE-SORT(A, p, q)                ▷ Conquer
       MERGE-SORT(A, q + 1, r)            ▷ Conquer
       MERGE(A, p, q, r)                  ▷ Combine
```

Initial call: $MERGE - SORT(A, 1, n)$

Example: bottom-up view for $n = 8$



Example: bottom-up view for $n = 11$



Merging

Input: Array A and indices p, q, r such that

- $p \leq q < r$.
- Subarray $A[p..q]$ is sorted and subarray $A[q+1..r]$ is sorted. By the restrictions on p, q, r, neither subarray is empty.

Output: The two subarrays are merged into a single sorted subarray in $A[p..r]$.

We implement it so that it takes $\Theta(n)$ time, where $n = r - p + 1 =$ the number of elements being merged.

What is n? Until now, n has stood for the size of the original problem. But now we're using it as the size of a subproblem. We will use this technique when we analyze recursive algorithms. Although we may denote the original problem size by n, in general n will be the size of a given subproblem.

Idea behind linear-time merging

- 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 these into a single sorted pile, face-down on the table.
- A basic step:
  - Choose the smaller of the two top cards.
  - Remove it from its pile, thereby exposing a new top card.
  - Place the chosen card face-down onto the output pile.
- Repeatedly perform basic steps until one input pile is empty.
- Once one input pile empties, just take the remaining input pile and place it face-down onto the output pile.
- Each basic step should take constant time, since we check just the two top cards. There are $\leq n$ basic steps, since each basic step removes one card from the input piles, and we started with n cards in the input piles.
- Therefore, this procedure should take $\Theta(n)$ time.

Improvement

- We don't actually need to check whether a pile is empty before each basic step.
- Put on the bottom of each input pile a special **sentinel** card.
- It contains a special value that we use to simplify the code.
- We use ∞, since that's guaranteed to "lose" to any other value.

Merging

MERGE(A, p, q, r)

$n_1 \leftarrow q - p + 1$ length

$n_2 \leftarrow r - q$

create arrays $L[1 \,..\, n_1 + 1]$ and $R[1 \,..\, n_2 + 1]$

for $i \leftarrow 1$ **to** n_1

  do $L[i] \leftarrow A[p + i - 1]$

for $j \leftarrow 1$ **to** n_2

  do $R[j] \leftarrow A[q + j]$

$L[n_1 + 1] \leftarrow \infty$

$R[n_2 + 1] \leftarrow \infty$

$i \leftarrow 1$ first Array

$j \leftarrow 1$ 2nd Array

for $k \leftarrow p$ **to** r

  do if $L[i] \leq R[j]$

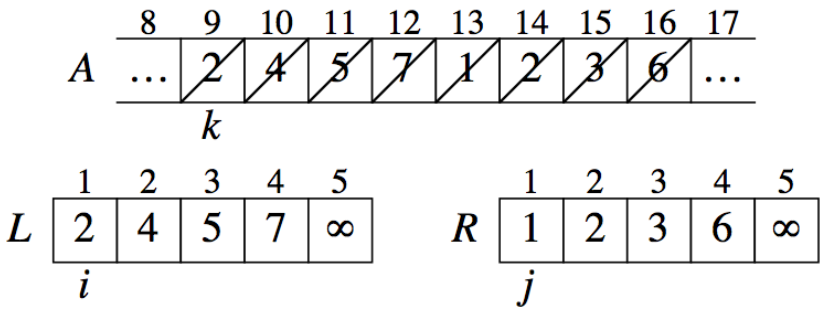
    then $A[k] \leftarrow L[i]$

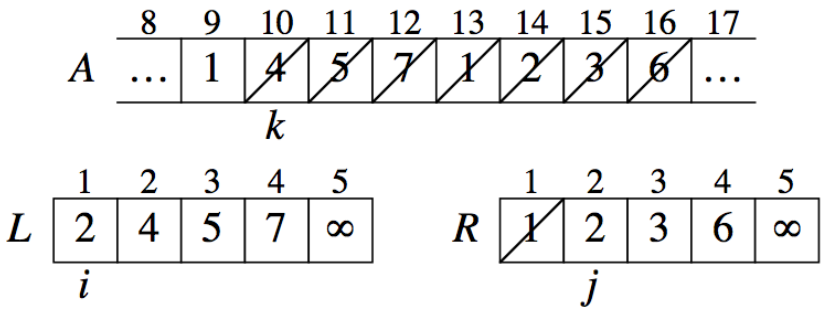
      $i \leftarrow i + 1$

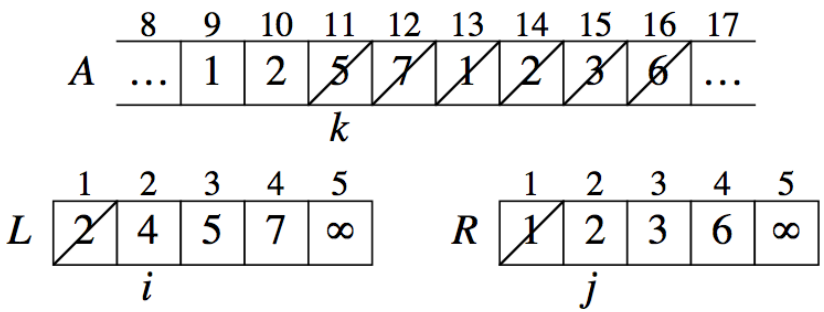
    else $A[k] \leftarrow R[j]$

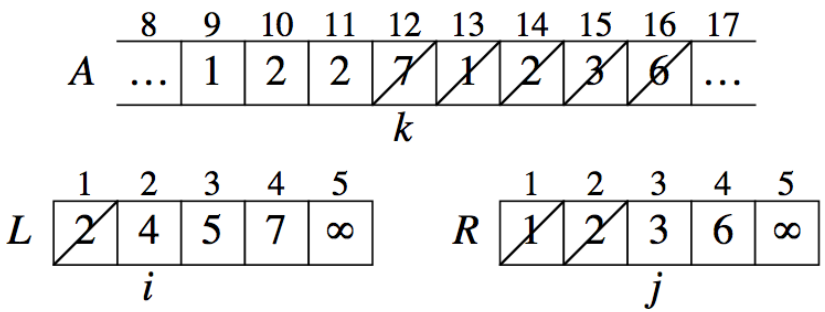
      $j \leftarrow j + 1$

Example: a call of $MERGE(9, 12, 16)$

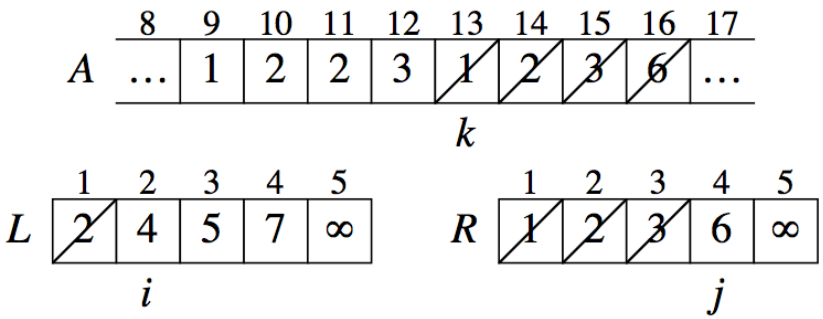


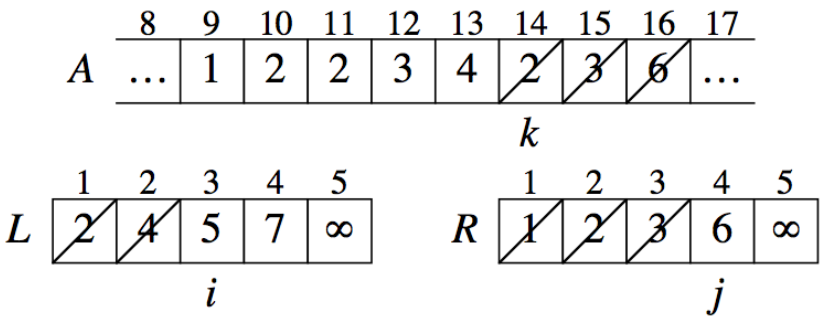


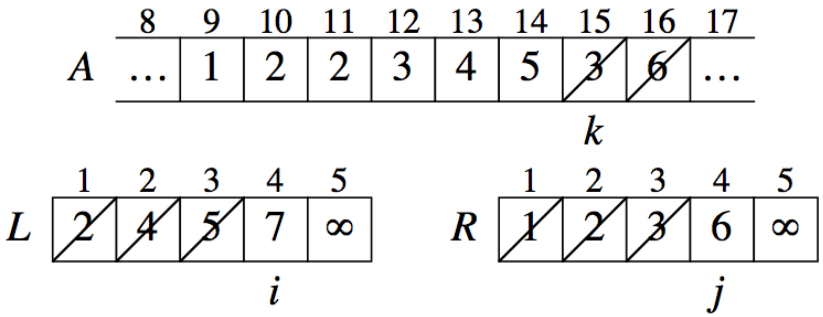


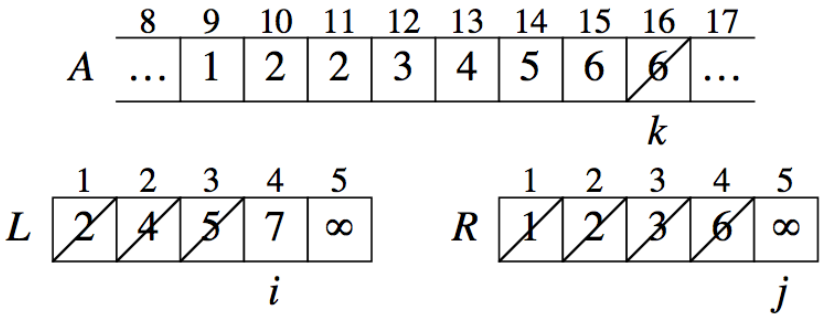


Example: a call of $MERGE(9, 12, 16)$

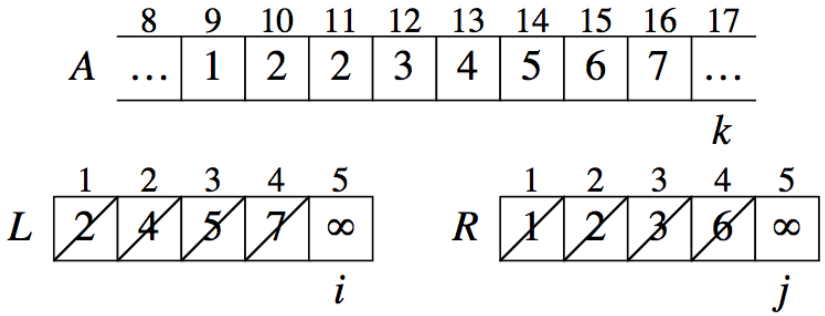








Example: a call of $MERGE(9, 12, 16)$



Running time: The first two **for** loops take $\Theta(n_1 + n_2) = \Theta(n)$ time. The last **for** loop makes n iterations, each taking constant time, for $\Theta(n)$ time.
Total time: $\Theta(n)$.

Analyzing divide-and-conquer algorithms

Use a ***recurrence equation*** (more commonly, a ***recurrence***) to describe the running time of a divide-and-conquer algorithm.

Let $T(n) =$ running time on a problem of size n.

- If the problem size is small enough (say, $n \leq c$ for some constant c), we have a base case. The brute-force solution takes constant time: $\Theta(1)$.
- Otherwise, suppose that we divide into a subproblems, each $1/b$ the size of the original. (In merge sort, $a = b = 2$.)
- Let the time to divide a size-n problem be $D(n)$.
- There are a subproblems to solve, each of size $n/b \Rightarrow$ each subproblem takes $T(n/b)$ time to solve $\Rightarrow$ we spend $aT(n/b)$ time solving subproblems.
- Let the time to combine solutions be $C(n)$.
- We get the recurrence

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

Analyzing merge sort

For simplicity, assume that n is a power of 2 $\Rightarrow$ each divide step yields two subproblems, both of size exactly $n/2$.

The base case occurs when $n = 1$.

When $n \geq 2$, time for merge sort steps:

Divide: Just compute q as the average of p and $r \Rightarrow D(n) = \Theta(1)$.

Conquer: Recursively solve 2 subproblems, each of size $n/2 \Rightarrow 2T(n/2)$.

Combine: MERGE on an n-element subarray takes $\Theta(n)$ time $\Rightarrow C(n) = \Theta(n)$.

Since $D(n) = \Theta(1)$ and $C(n) = \Theta(n)$, summed together they give a function that is linear in n: $\Theta(n) \Rightarrow$ recurrence for merge sort running time is

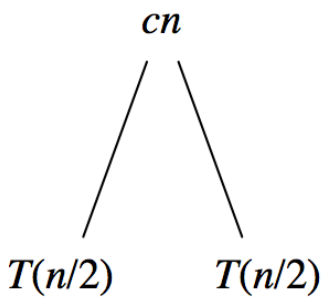
$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1 \,, \\ 2T(n/2) + \Theta(n) & \text{if } n > 1 \,. \end{cases}$$

Solving the merge-sort recurrence

- Let c be a constant that describes the running time for the base case and also is the time per array element for the divide and conquer steps. *[Of course, we cannot necessarily use the same constant for both. It's not worth going into this detail at this point.]*
- We rewrite the recurrence as

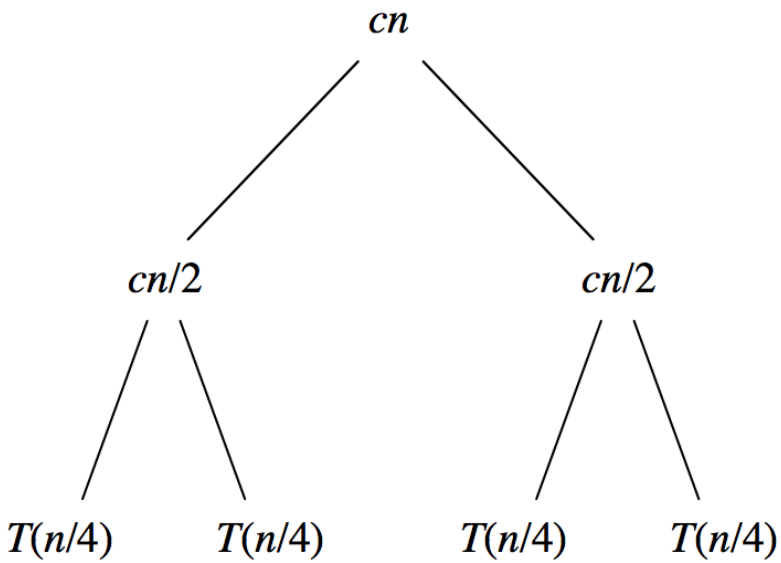
$$T(n) = \begin{cases} c & \text{if } n = 1 \ , \\ 2T(n/2) + cn & \text{if } n > 1 \ . \end{cases}$$

- Draw a ***recursion tree***, which shows successive expansions of the recurrence.
- For the original problem, we have a cost of cn, plus the two subproblems, each costing $T(n/2)$:



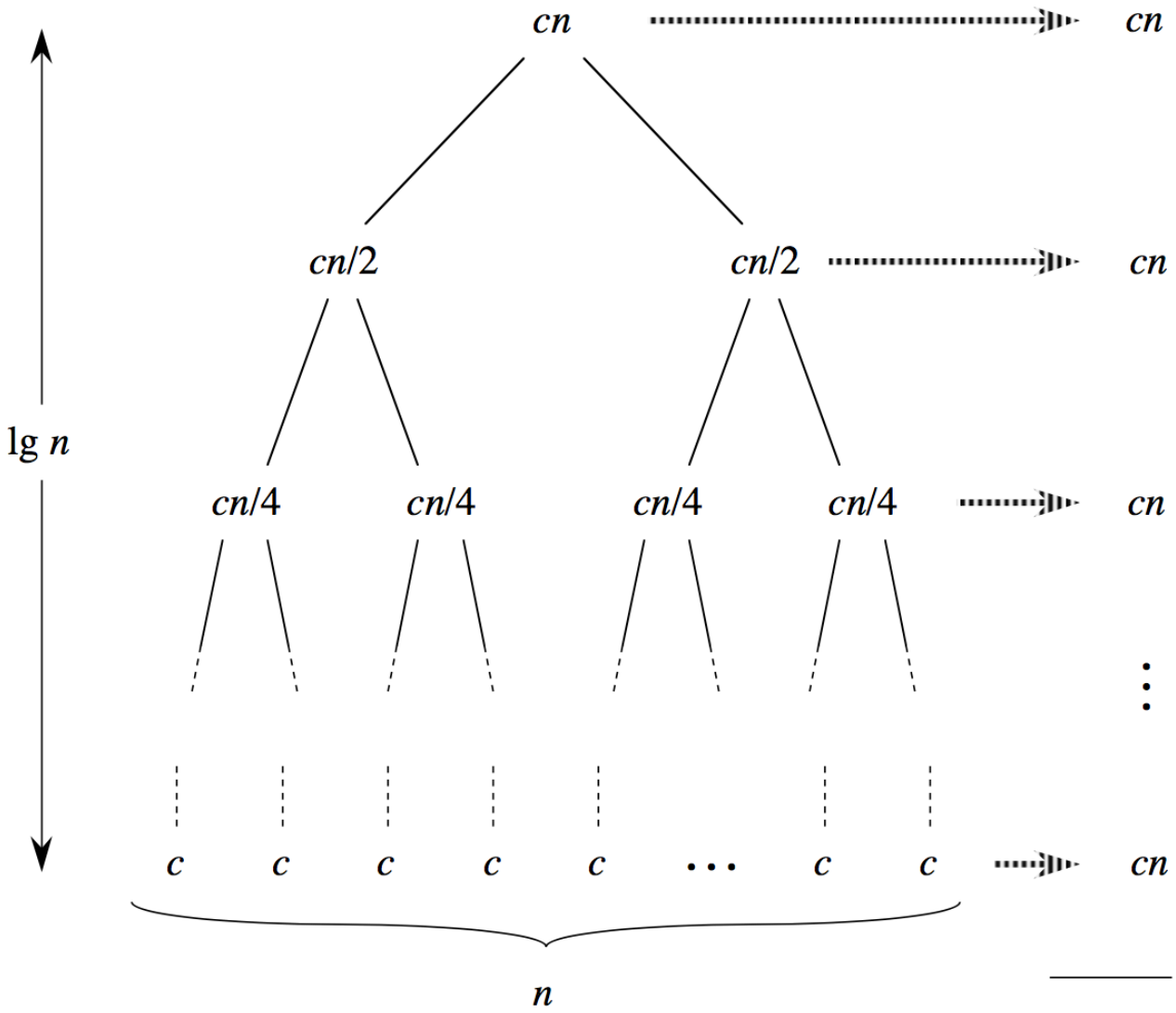
Solving the merge-sort recurrence

- For each of the size-$n/2$ subproblems, we have a cost of $cn/2$, plus two subproblems, each costing $T(n/4)$:



Solving the merge-sort recurrence

- Continue expanding until the problem sizes get down to 1:



Total: $cn \lg n + cn$

Solving the merge-sort recurrence

- Each level has cost cn.
  - The top level has cost cn.
  - The next level down has 2 subproblems, each contributing cost $cn/2$.
  - The next level has 4 subproblems, each contributing cost $cn/4$.
  - Each time we go down one level, the number of subproblems doubles but the cost per subproblem halves $\Rightarrow$ cost per level stays the same.
- There are $\lg n + 1$ levels (height is $\lg n$).
  - Use induction.
  - Base case: $n = 1 \Rightarrow 1$ level, and $\lg 1 + 1 = 0 + 1 = 1$.
  - Inductive hypothesis is that a tree for a problem size of 2^i has $\lg 2^i + 1 = i + 1$ levels.
  - Because we assume that the problem size is a power of 2, the next problem size up after 2^i is 2^{i+1}.
  - A tree for a problem size of 2^{i+1} has one more level than the size-2^i tree $\Rightarrow$ $i + 2$ levels.
  - Since $\lg 2^{i+1} + 1 = i + 2$, we're done with the inductive argument.
- Total cost is sum of costs at each level. Have $\lg n + 1$ levels, each costing $cn \Rightarrow$ total cost is $cn \lg n + cn$.
- Ignore low-order term of cn and constant coefficient $c \Rightarrow \Theta(n \lg n)$.

Compare with insertion sort

$f(n) = \Theta(g(n))$ iff $f = O(g(n))$ and $f = \Omega(g(n))$

(mid term)

- Compared to insertion sort ($\Theta(n^2)$ worst-case time), merge sort is faster. Trading a factor of n for a factor of $\lg n$ is a good deal.
- On small inputs, insertion sort may be faster. But for large enough inputs, merge sort will always be faster, because its running time grows more slowly than insertion sort's.

HW1 assignments

1. Exercise 2.2-2, page 29 of text book.
2. Exercise 2.3-2, page 38 of text book.
3. Exercise 2.3-3, page 39 of text book.
4. Exercise 2.3-4, page 39 of text book.
5. Problem 2-2, page 40 of text book.
6. Problem 2-4, page 41 of text book.