ECS 32B - Sorting Algorithms

Aaron Kaloti

UC Davis - Summer Session #2 2020

UC DAVIS
COMPUTER SCIENCE

Overview

Goal

- Much of this slide deck contains explanations about how to use many different algorithms and the worst-case analyses. The analysis is the important part, and sure enough, conceptual HW #4 and/or exam #3 will have you do more analysis about the differences between these algorithms.

Sorting Algorithms

- Comparison-based sorts (provably can't do better than $\Theta(n \lg n)$):
  - Perform adjacent exchanges:
    - Bubble sort.
    - Selection sort.
    - Insertion sort.
  - Shellsort.
  - Heapsort.
  - Mergesort.
  - Quicksort.
- Linear time sorts:
  - Bucket sort.
  - Radix sort.

Note on the spelling of the sorting algorithms: for some reason, certain sorts have a space in the name, and others don't, e.g. "Shellsort" vs. "Bucket sort". I used the conventional spellings for each.

Temporary Assumptions

- All keys are integers.
- All keys are unique.

Bubble Sort[1]

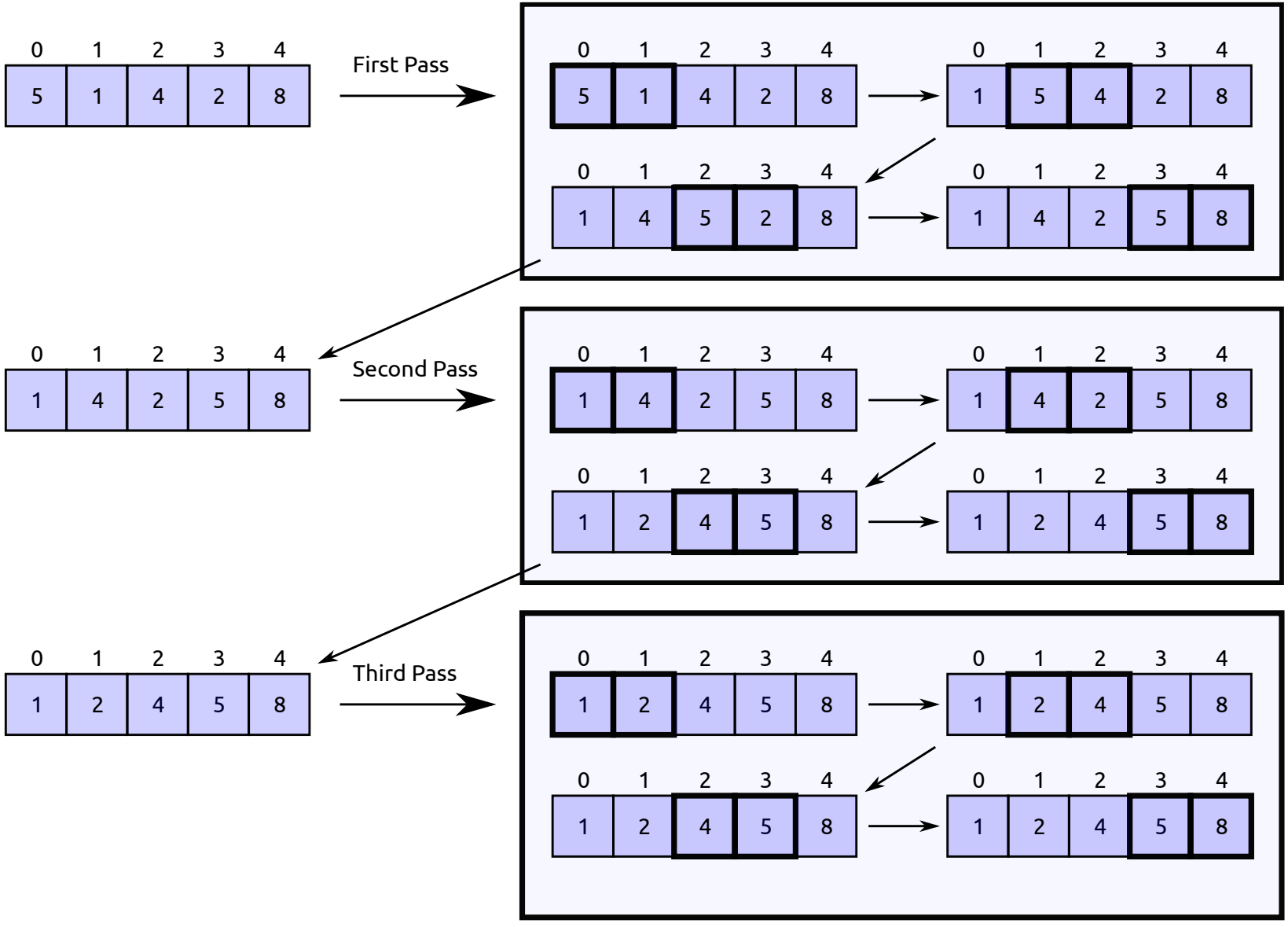
Description

- Keep doing a pass through the input list until sorted.
- While doing a pass:
  - Swap $A[i]$ and $A[i+1]$ if $A[i] > A[i+1]$.

1. Reference / further reading: https://en.wikipedia.org/wiki/Bubble_sort

Bubble Sort

Example



Bubble Sort

Analysis

- Considered the worst reasonable sorting algorithm[1].
- Worst-case time complexity: $\Theta(n^2)$.
  - Nested loops.
    - Outer loop: n iterations.
    - Inner loop: n iterations (per outer loop iteration).

1. The worst sorting algorithm, reasonable or otherwise, would be bogosort.

Selection Sort

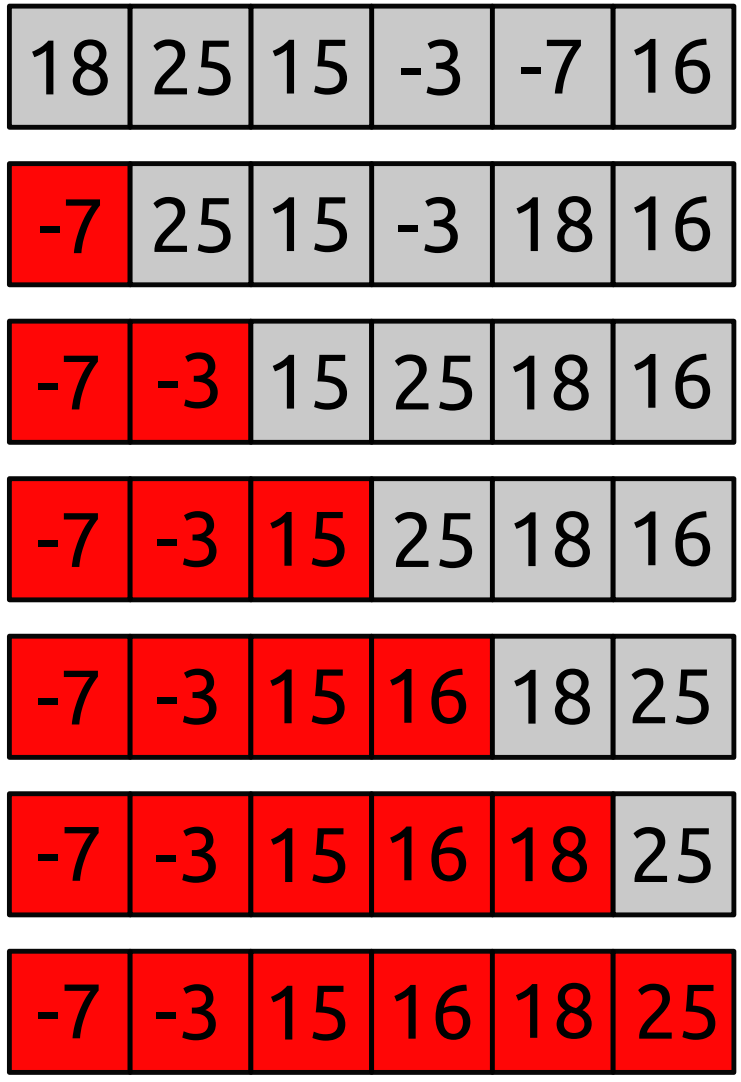
Description

- View list as two partitions:
  1. Sorted part.
     - Initially empty.
  2. (Potentially) unsorted part.
- Until partition #2 is empty, add its smallest element to partition #1.

Analysis

- Usually better than bubble sort.
- Worst-case time complexity: $\Theta(n^2)$.
  - Nested loops.
    - Outer loop: n iterations.
    - Inner loop: $n - i$ iterations (per outer loop iteration), where i is outer loop index.
  - $\sum_{i=0}^{n-1} n - i = \sum_{i=1}^{n} i$
    $= 1 + 2 + 3 + \ldots + (n-2) + (n-1) + n$
    $= (n+1)\frac{n}{2} = \Theta(n^2)$.

Example



Insertion Sort

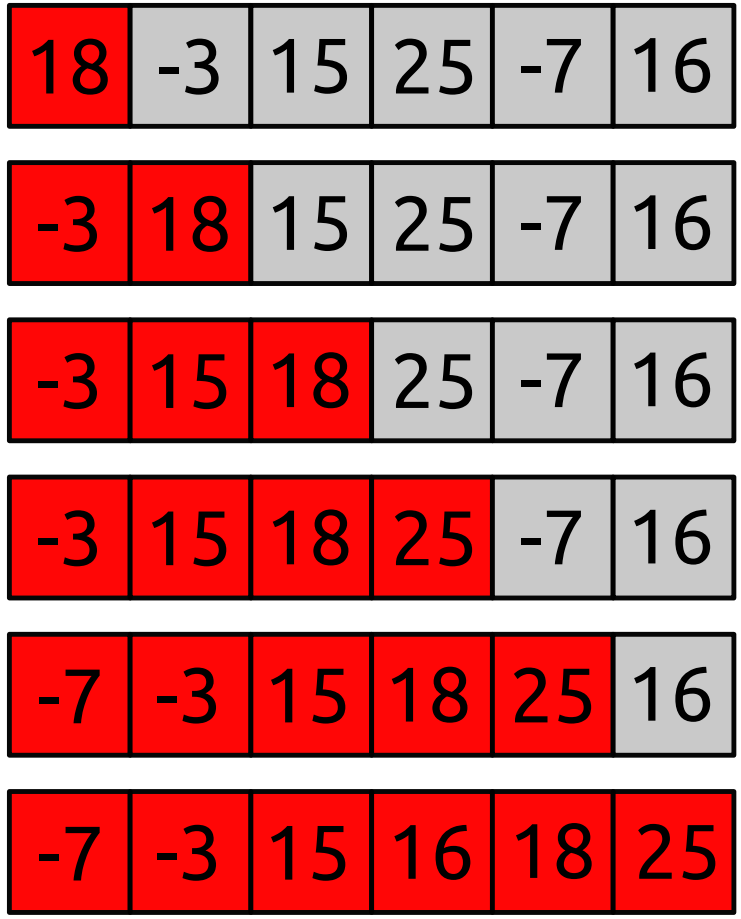
Description

- View list as two partitions:
  1. Sorted part.
     - Initially first element.
  2. (Potentially) unsorted part.
- Until partition #2 is empty, add its ***first*** element to partition #1.

Analysis

- Best quadratic time sort.
- Worst-case time complexity: $\Theta(n^2)$.
  - Nested loops.

Example



Analyzing Adjacent Exchange Sorting Algorithms

- Includes: bubble sort, selection sort, insertion sort.

Lower Bound on Worst Case

- **inversion** (in a list A of numbers): any ordered pair (i, j) such that $i < j$, $A[i] > A[j]$.
- Each swap of adjacent numbers removes *one* inversion.
- Max number of inversions: $\binom{n}{2} = \frac{n!}{2!(n-2)!} = \frac{n(n-1)}{2} = \Theta(n^2) \Rightarrow$ any sorting algorithm that swaps adjacent elements (i.e. eliminates one inversion per swap) takes $\Omega(n^2)$ time in worst case.

Analyzing Adjacent Exchange Sorting Algorithms

Lower Bound on Average Case

- Assume all permutations of n elements are equally likely.
- Average number of inversions: $\frac{n(n-1)}{4} = \Theta(n^2)$.
  - Two ways to prove.
- $\Rightarrow$ any adjacent exchange sorting algorithm takes $\Omega(n^2)$ time on average.

Proof #1: Weiss' Way

- Let A_r be A in reverse order.
- Each $(i, j), 1 \leq i < j \leq n$ (n is list size) is an inversion in *one* of A and A_r.
- Total number of inversions in A and A_r is $\frac{n(n-1)}{2}$, so an average list has half this many, i.e. $\frac{n(n-1)}{4}$.

Analyzing Adjacent Exchange Sorting Algorithms

Lower Bound on Average Case

Proof #2: My Preferred Way

- Number of inversions between index i element and elements after.

i	**Worst Case**	**Average**
0	$n-1$	$\frac{n-1}{2}$
1	$n-2$	$\frac{n-2}{2}$
2	$n-3$	$\frac{n-3}{2}$
...	...	...
$n-3$	2	1
$n-2$	1	$\frac{1}{2}$
$n-1$	0	0

- Average number of inversions: $\sum_{i=0}^{n-1} \frac{n-i-1}{2}$

$= \frac{n-1}{2} + \frac{n-2}{2} + \frac{n-3}{2} + \ldots + 1 + \frac{1}{2} + 0$

$= \frac{n-1}{2}\frac{n}{2} = \frac{n(n-1)}{4}$

Shellsort

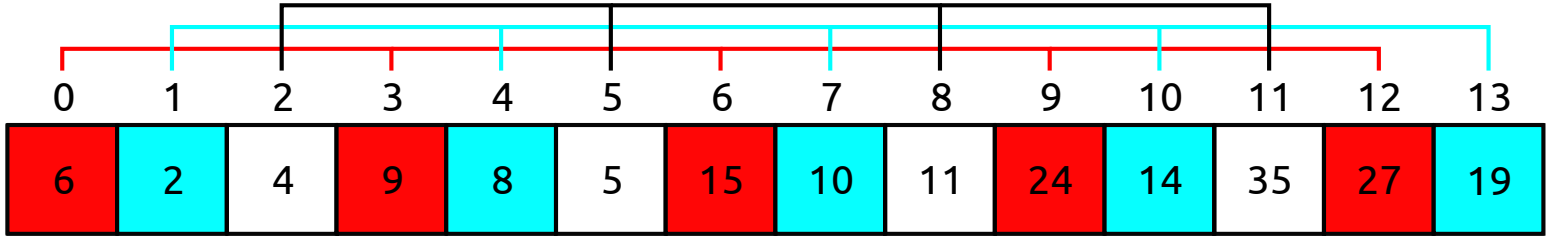
- *Based on previous analysis*: need to compare distant elements ⇒ can eliminate more than one inversion per exchange.
- a.k.a. diminishing increment sort.

Description

- **increment sequence**: $h_1, h_2, ..., h_t$.
  - Any with $h_1 = 1$ works.
- Steps:
  1. Start with greatest $h_i \leq n$, where n is number of elements.
  2. "h_i-sort" the list.
  3. Go to h_{i-1} and repeat #2. Repeat until after h_{i-1}-sort the list.

h_k-Sorted

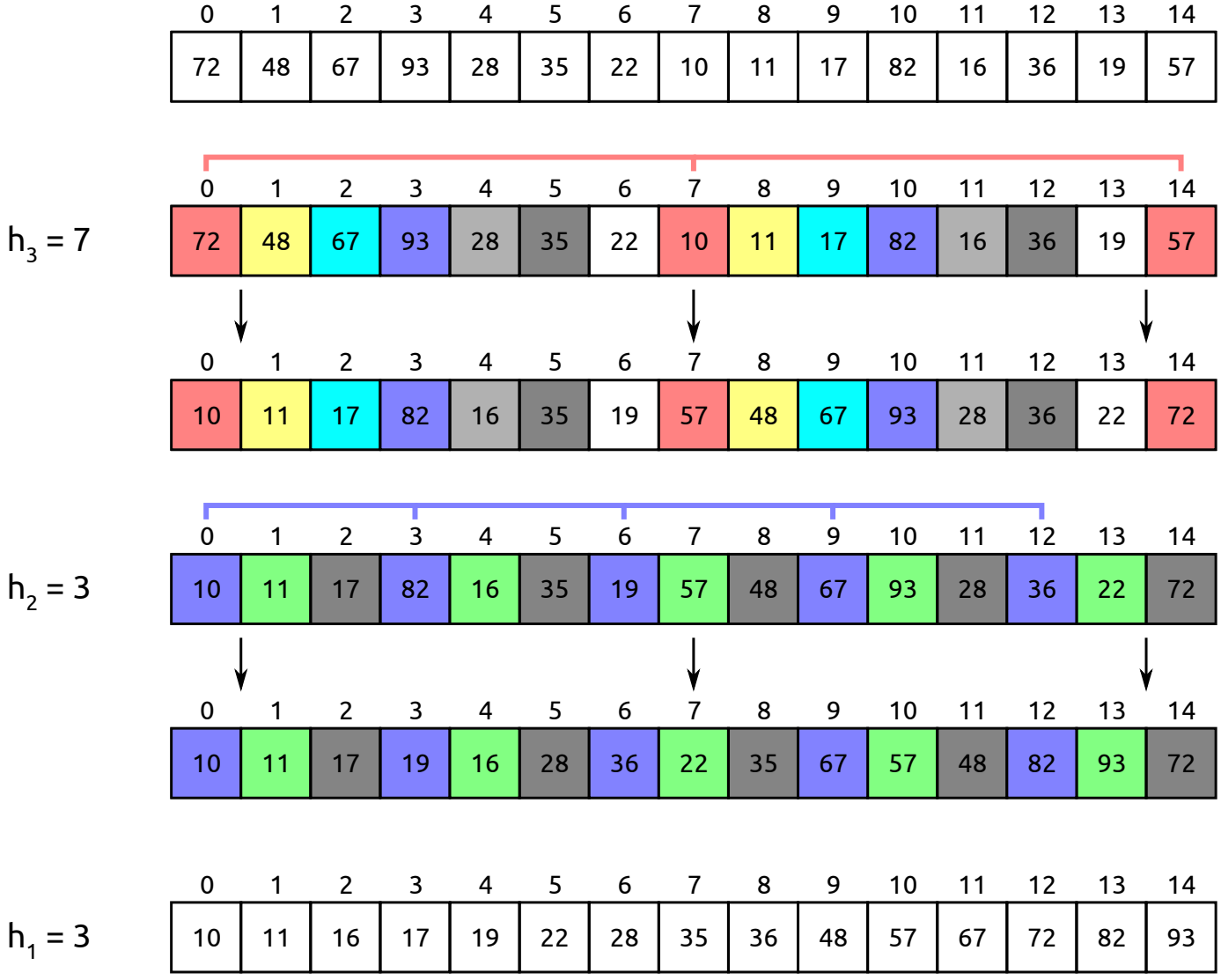
- All elements spaced h_k apart are sorted.
- Example (this list is 3-sorted):



Shellsort

Example

- Use **Shell's increments**[1]: $h_t = \lfloor \frac{n}{2} \rfloor$ and $h_k = \lfloor \frac{h_{k+1}}{2} \rfloor$.



1. Donald Shell invented shellsort. Ironically, his increments are apparently a poor choice.

Shellsort

Worst-Case Time Complexity[1]

- Using Shell's increments: $\Theta(n^2)$.
- Using Hibbard's increments: $\Theta(n^{3/2})$.
  - Hibbard's increments: $1, 3, 7, ..., 2^k - 1$.
- Many increments proposed by Sedgewick give $O(n^{4/3})$.

1. See section 7.4 of Weiss' book for proofs.

Sorting Algorithm Characteristics

- **in place**: uses constant auxiliary space.
- **stable**: preserves relative order of equal values.

Algorithm	In place?	Stable?
Bubble sort	Yes	Yes
Selection sort	Yes	No
Insertion sort	Yes	Yes
Shellsort	Yes	No
Heapsort	Yes	No
Mergesort	No	Yes
Quicksort	No	No

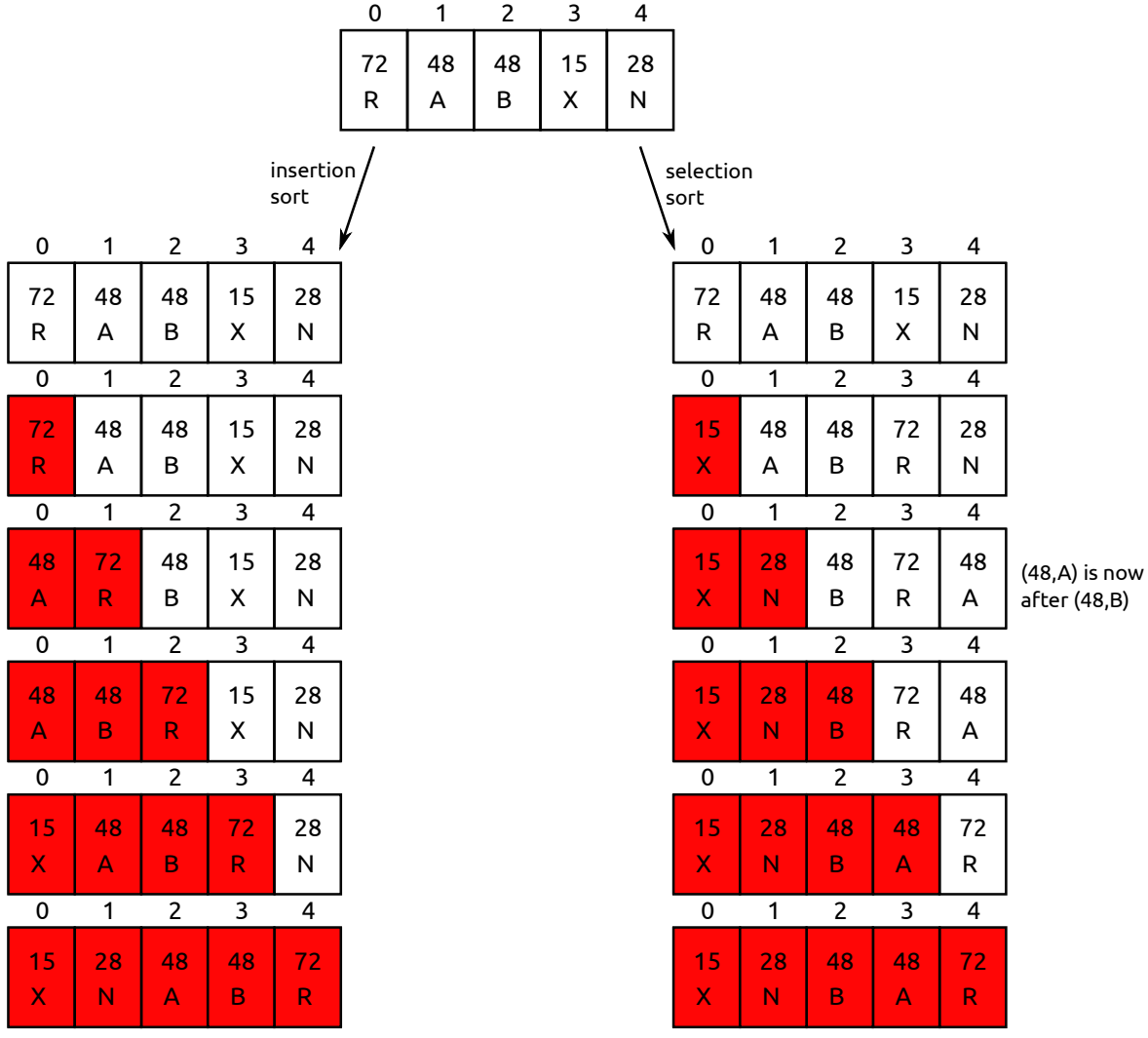
Further reading: https://en.wikipedia.org/wiki/Sorting_algorithm#Comparison_of_algorithms

Sorting Algorithm Characteristics

Stable vs. Unstable

- Stability is only relevant if sorting keys (of key-value pairs) and are duplicate keys.

Example: Insertion Sort vs. Selection Sort



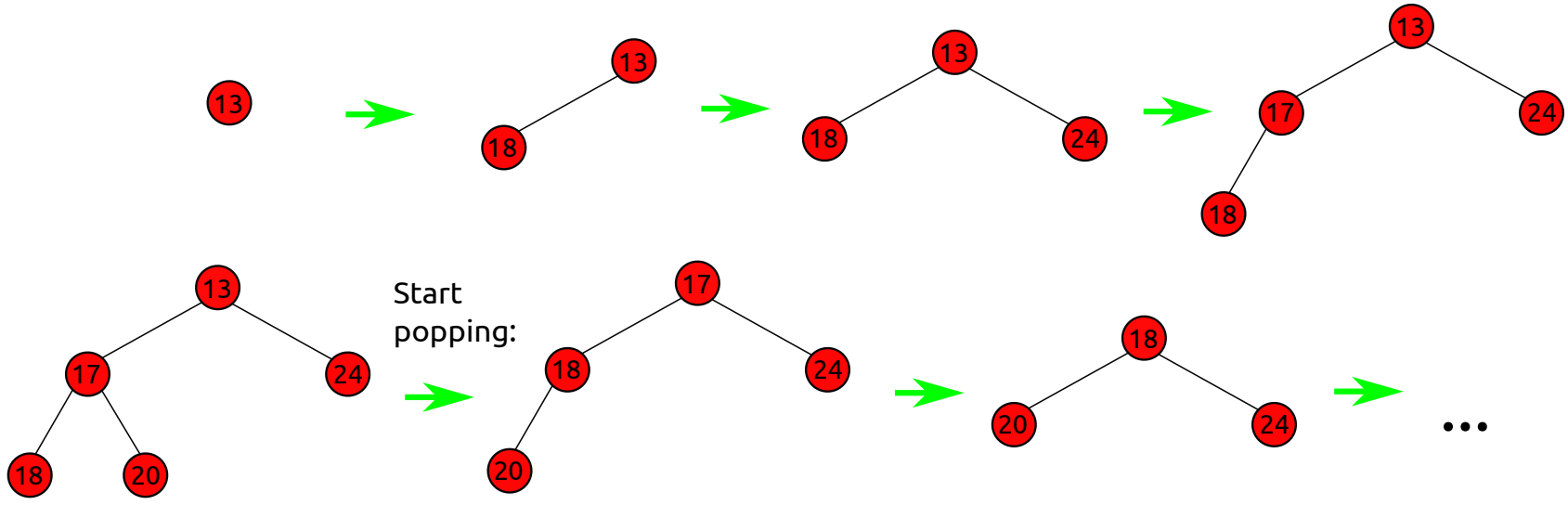
Heapsort

Description

- Steps:
  1. Perform build heap operation on input list to create a min heap. ($\Theta(n)$ time)
  2. Delete the min n times. ($\Theta(n \lg n)$ time)
     - Ends up extracting the sorted list.

Example

- Insertion order: $13, 18, 24, 17, 20$.



- Extraction order: $13, 17, 18, 20, 24$.

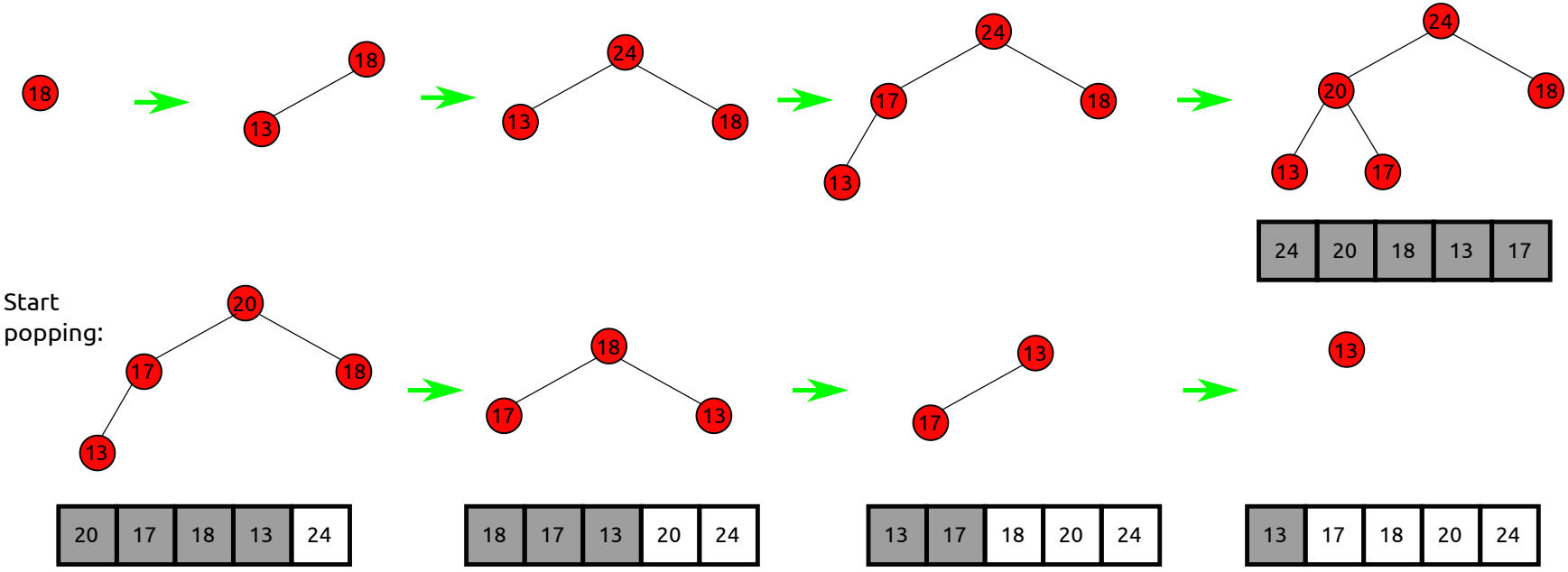
Heapsort

Achieving Constant Auxiliary Space

- Two modifications:
  1. Use *max heap* instead.
  2. When extract element, put at *end of list* (but don't treat it as part of the heap), i.e. swap max with element that would replace it, then percolate replacement down.

Example

- Insertion order: $13, 18, 24, 17, 20$.



Heapsort

Analysis

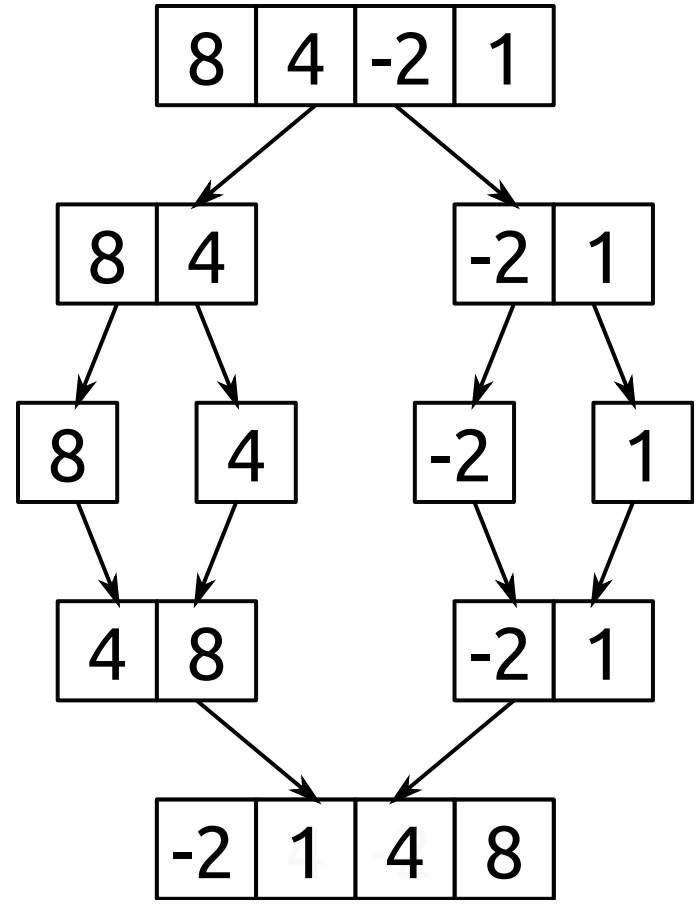
- Worst-case time complexity: $\Theta(n \lg n)$.
- Auxiliary space: $\Theta(1)$.
- Slower than mergesort and quicksort (despite time complexity), but better auxiliary space than them.

Mergesort

Description

- Is a divide-and-conquer algorithm[1].
- Steps:
  1. *Divide* unsorted list into two equal halves.
  2. *Conquer*: recursively mergesort each half.
     - Can view base case as either (doesn't change runtime):
       1. 1 or 2 elements remain.
       2. 1 element remains.
  3. *Combine* two sorted halves into final sorted list.

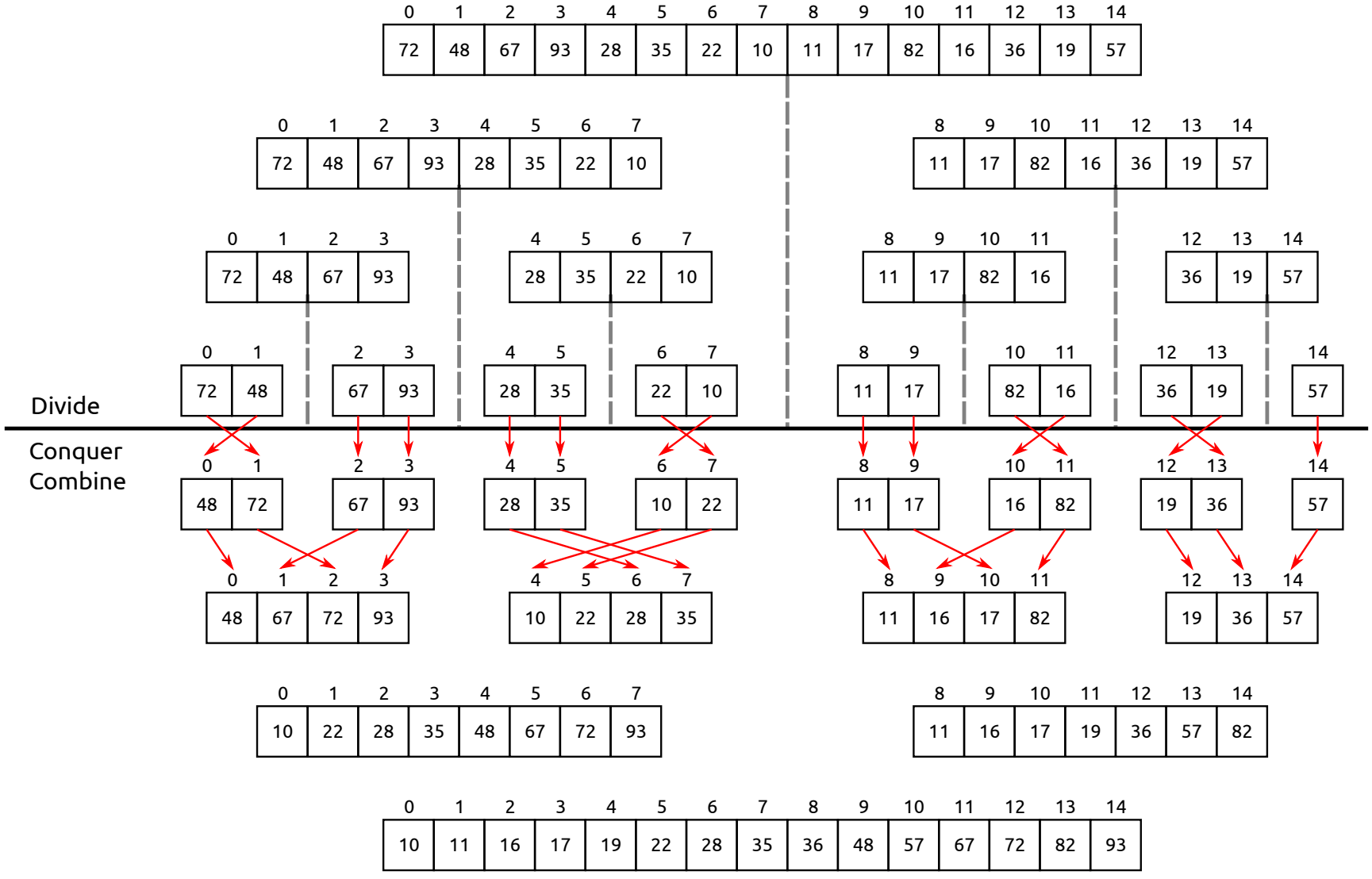
Example #1



1. You'll see more of such algorithms in ECS 122A. Some may argue it should be called divide-conquer-and-combine, but that doesn't stick as well.

Mergesort

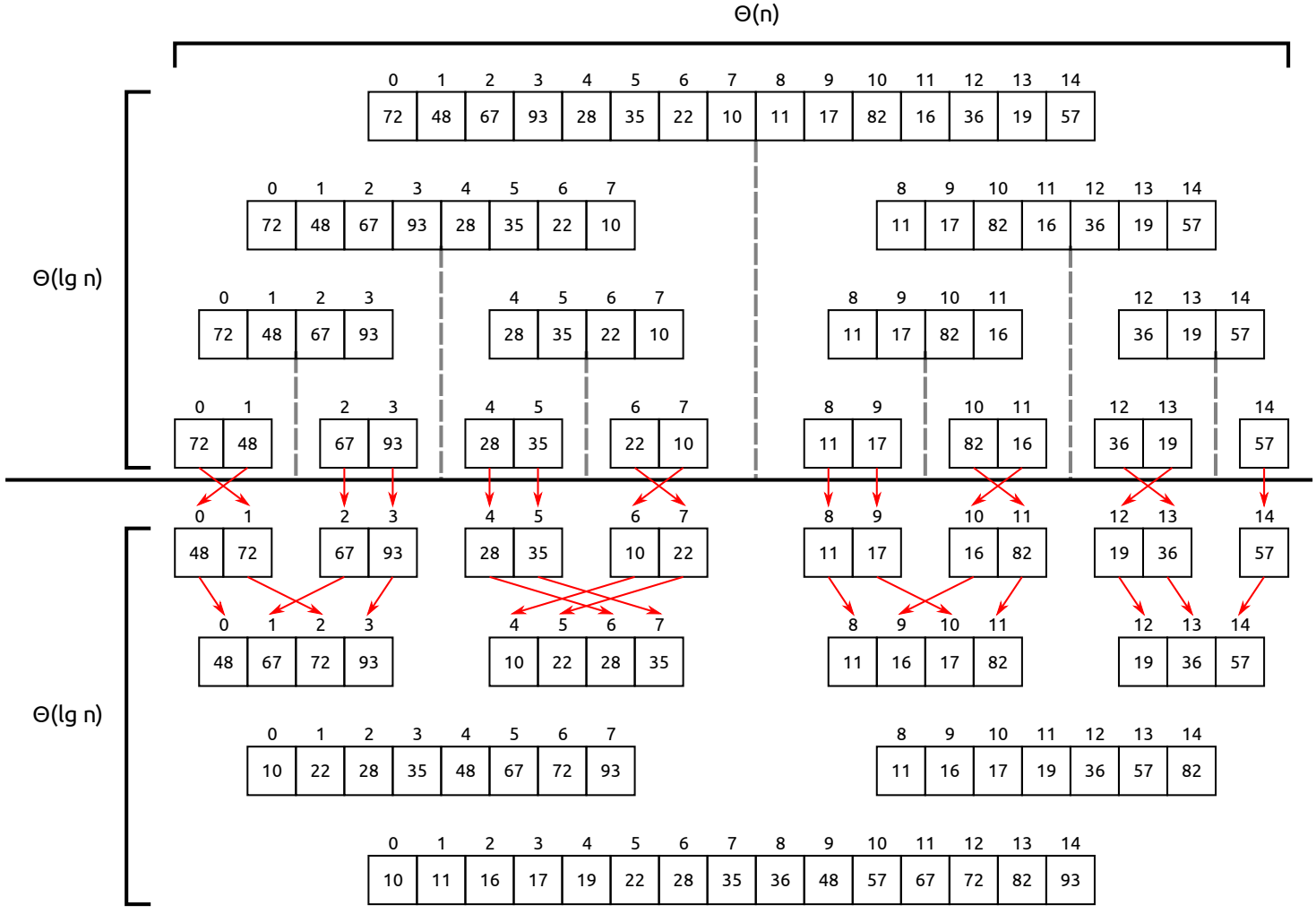
Example #2



Mergesort

Analysis

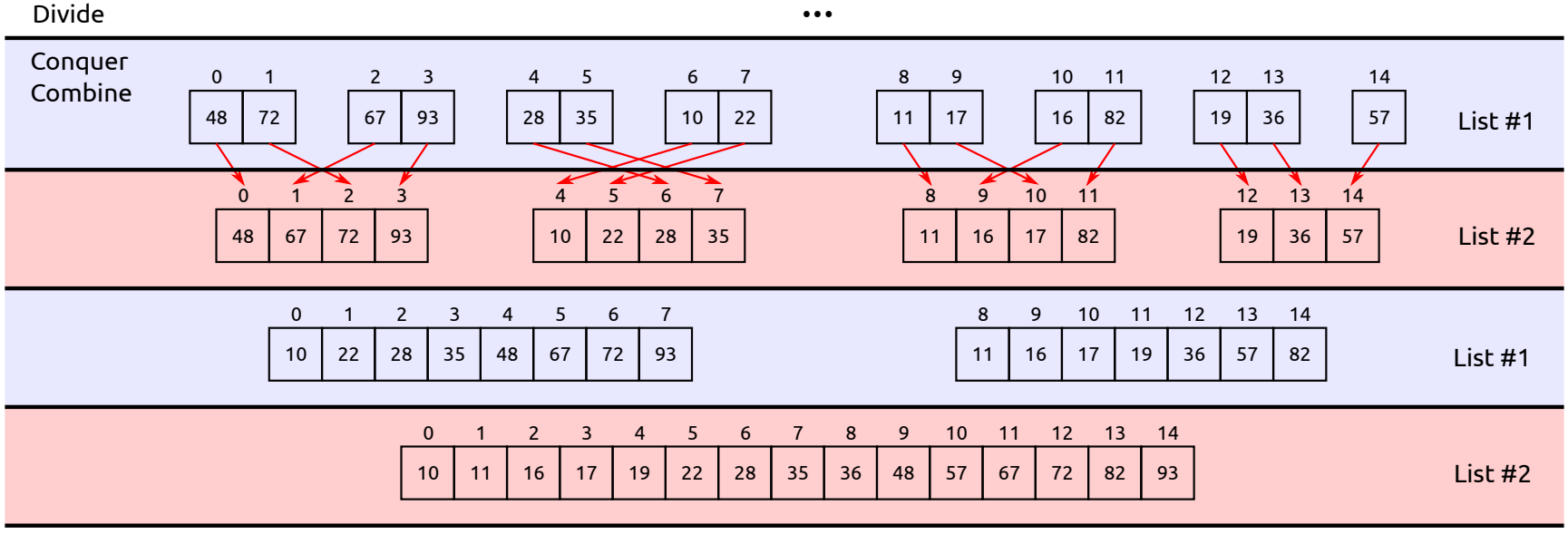
- Worst-case time complexity: $\Theta(n \lg n)$.
  - Can merge two halves in linear time.
  - $\Theta(n)$ work is done $\Theta(\lg n)$ times.



Mergesort

Analysis

- Auxiliary space:
  - Naive approach (keep copying to new list when combine): $\Theta(n \lg n)$.
  - Smarter approach (keep switching between two lists of length n while combining): $\Theta(n)$.



Quicksort

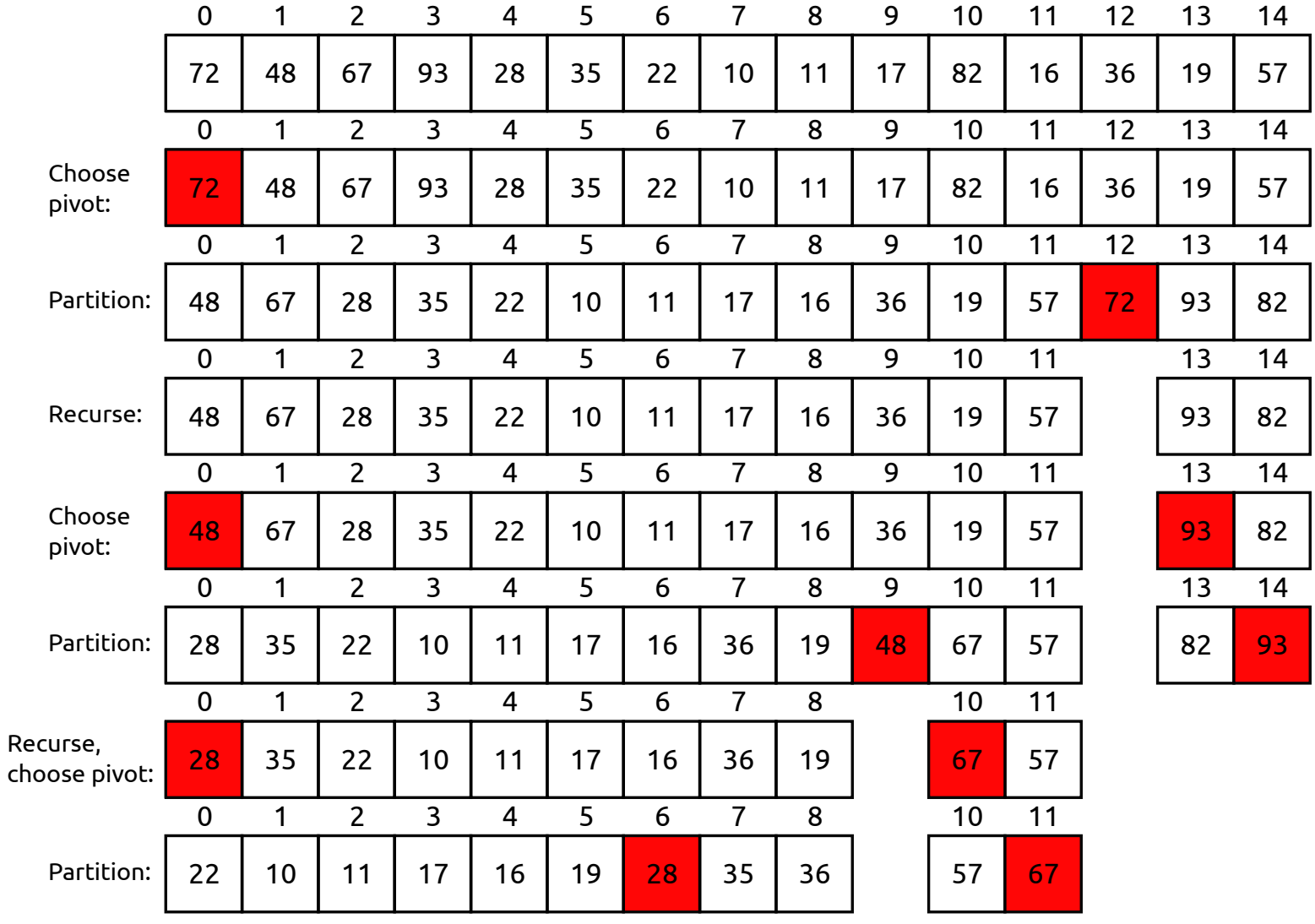
Description

- Worst-case time complexity: $\Theta(n^2)$.
- Average-case time complexity: $\Theta(n \lg n)$.
- In practice, faster than heapsort and mergesort.
- Also a divide-and-conquer algorithm.
- Steps:
  1. According to some rule, choose a pivot element v in list. *Divide* list into two halves, such that first half has elements less than v and second half has elements greater than v.
     - v is excluded from both halves. After this partioning, we know v is in final spot.
  2. *Conquer*: recursively quicksort each half.
     - Base case: 0 or 1 elements remain.
  3. *Combine* two sorted halves into final sorted list.
     - This phase is trivial compared to mergesort's.

Quicksort

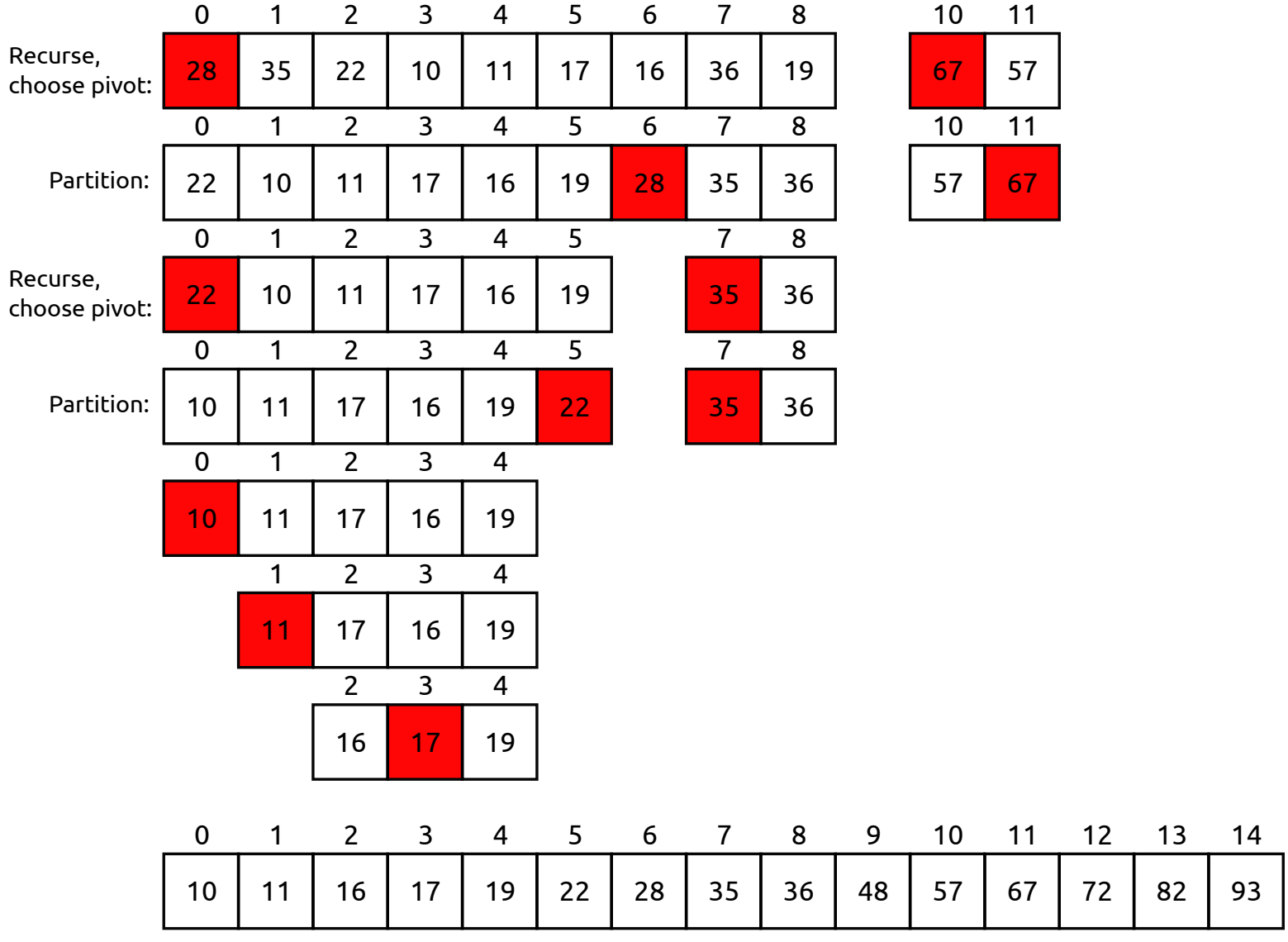
Example #1

- Pivot rule: choose first element in list.



Quicksort

Example #1 (Continued)



Quicksort

Possible Rules for Picking Pivot

- Bad ways:
  - Choose first element.
  - Choose larger of the first two distinct elements.
- Choose random pivot.
- *Best*: median-of-three partioning ⇒ choose median of first, center, and last element.

Quicksort

Partioning

- Careless partioning method can make algorithm wrong or inefficient (in time or space).

Suggested Strategy

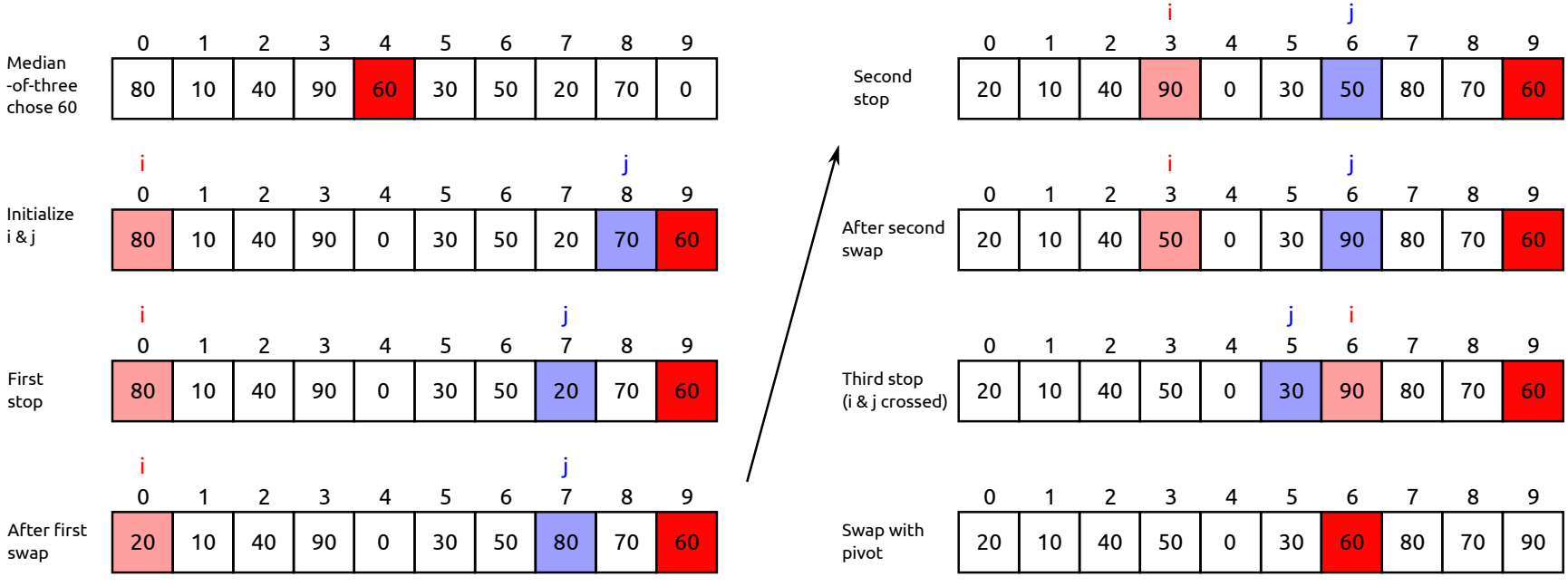
- In place[1].
- Steps[2]:
  1. Swap pivot with last element.
  2. Create two references/indices i and j. i starts at first element; j, at next-to-last element.
  3. While i is to the left of j, move i right until at element *not smaller* than the pivot, and move j left until at element *not larger* than the pivot.
  4. When i and j have stopped, if i is to the left of j, swap their elements and go back to #3. Stop once i and j cross.
  5. Swap element at i with the pivot.

1. Note that the quicksort algorithm as a whole isn't in place.
2. From section 7.7.2 of Weiss' book. Note that the previous example does not use this partioning strategy. *How can you tell?*

Quicksort

Partioning

Suggested Strategy: Example



Hybrid Sorting

- Hybrid sorting uses combination of fast sorting algorithm and slow sorting algorithm.
  - Slow sorting algorithms (e.g. insertion sort) better for smaller inputs.

Introsort[1]

- **introsort**: starts with quicksort, then switches to heapsort after enough recursion, then switches to insertion sort when small number of elements.
- Often used in C++ standard library implementations.
- Worst-case time complexity is provably $\Theta(n \lg n)$.

Timsort[2]

- **timsort**: combines merge sort and insertion sort.
- Used in Python and Java.
- Worst-case time complexity is provably $\Theta(n \lg n)$.

1. https://en.wikipedia.org/wiki/Introsort
2. https://en.wikipedia.org/wiki/Timsort

Analyzing Comparison-Based Sorting Algorithms

- Has been proven[1] that any sorting algorithm that only uses comparisons must perform $\Omega(n \lg n)$ comparisons.
- To do better, need:
  - Non-comparison-based approach.
  - Other information about the input.

1. See section 7.8 of Weiss' book.

Bucket Sort

Description

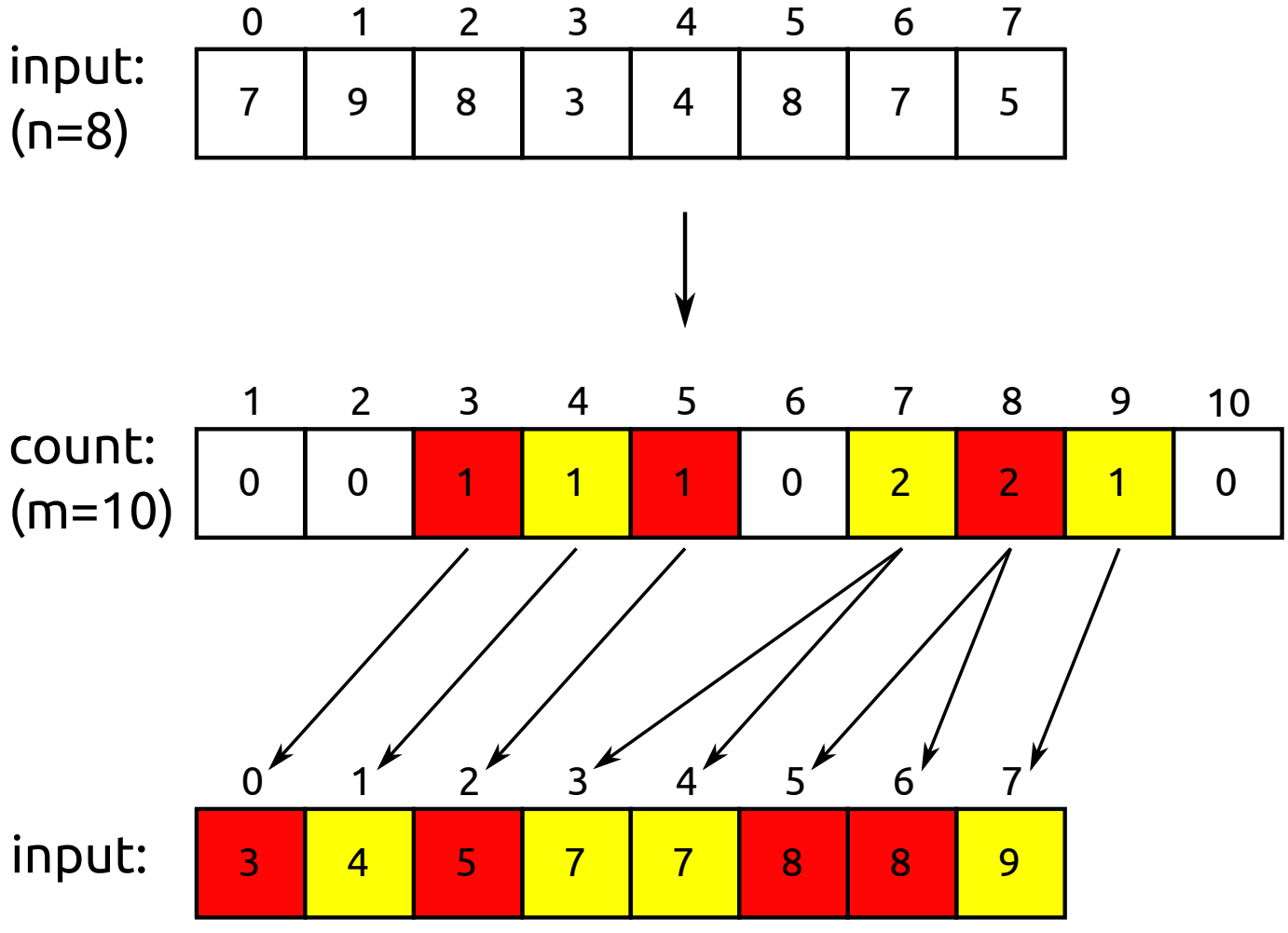
- Input must be only positive integers less than or equal to m, i.e. we're assuming the range of integers is $[1, m]$, where m is something we happen to know.
- Steps:
  1. Create list of integers (initialized to 0) called $count$.
  2. Read each integer. When read x, increment $count[x]$.
  3. Scan $count$, printing out sorted list.

Worst-Case Time Complexity

- $\Theta(m + n)$ time.
- If $m = \Theta(n)$, then total is $\Theta(n)$.
  - This is a tough/strong assumption to make. It implies the range of the integers m is a function of the input size n.

Bucket Sort

Example



1. Fixed image to reflect our assumption that input list contains only positive integers.

Regarding Final Exam

- Student: "Aaron, do I have to know all of these?"
- Aaron: "Sort of."[1]

1. Bad puns aside, consult the updated study guide. Also, the exam is open note (as always).

References / Further Reading

- We will not cover the following sorting algorithms that I previously suggested we might cover:
  - Other linear time sorts (e.g. radix/card sort, counting radix sort).
  - External sorting (on tape).
  - Bogosort. (This one is kind of a joke; look it up.)
- Chapter 7 of *Data Structures and Algorithm Analysis in C++* by Mark Allen Weiss (Fourth Edition).
- Sections 6.6 - 6.12 of *Problem Solving with Algorithms and Data Structures using Python* by Brad Miller and David Ranum.