Bucket Sort; External Sort

EECS 233

Various Sorting Methods Learned

- Selection Sort
- Insertion Sort
- Shell-sort
- Bubble Sort
- Quick-sort
- Merge-sort
- Heap-sort

Review: Various Sorting Methods

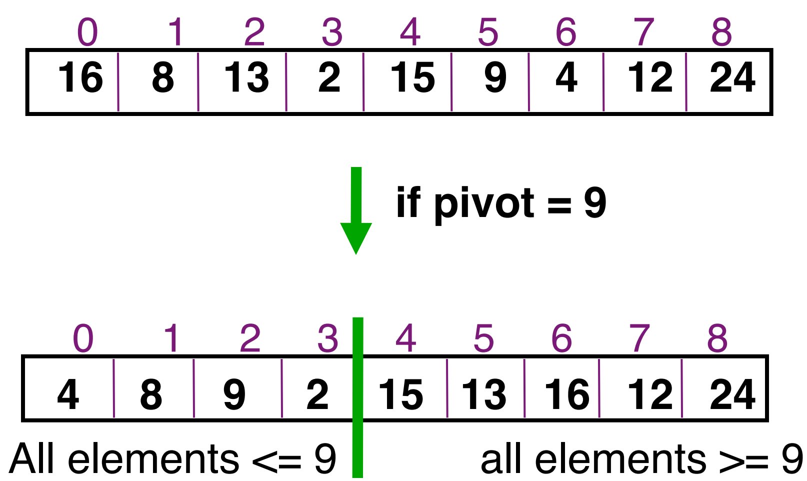
0	1	2	3	4	5	6	7	8
16	8	13	2	15	9	4	12	24

- Selection Sort
  - For each position i, where i runs from 0 to length-1, find the element that should be placed there
  - Small elements are placed in the beginning of the array
- Bubble Sort
  - In each pass, swap out-of-order neighboring elements
  - Large elements are bubbled up towards the end of the array
- Insertion sort
  - For every element (starting from the second in the array), insert it into the sorted partial array to the left.
  - More efficient if the array is sorted or almost sorted
- Shell-sort
  - Generalize the Insertion sort to speed-up movement towards the final destination for an element
  - Divide into k interleaved subarrays, with increment = k
  - For each subarray, run insertion sort
  - Repeated this with a decreased value of k until k=1

0	1	2	3	4	5	6	7	8
16	**8**	**13**	**2**	**15**	**9**	**4**	**12**	**24**

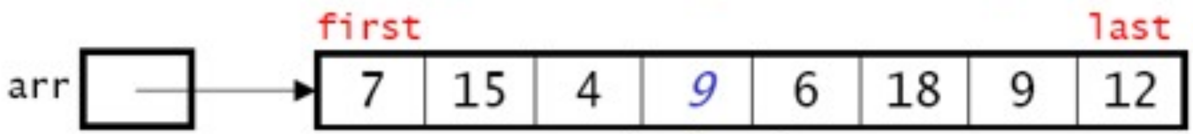
Review: Quick-Sort

- Divide-and-conquer
- Choose pivot value to partition an array into two subarrays such that left subarray <= right subarray
  - By swapping out-of-place elements
- Recursive method to solve the problem

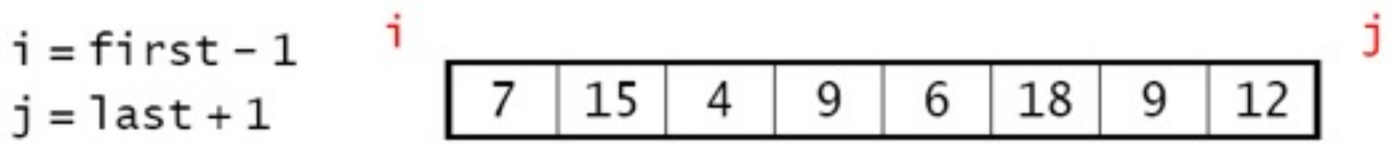


An Example of Partitioning An Array

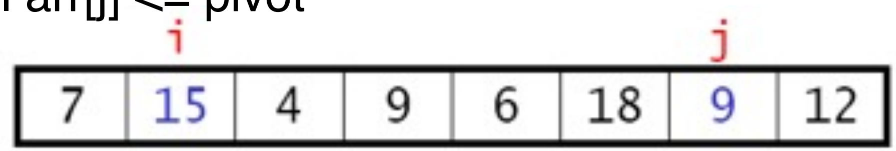
Pivot = middle element



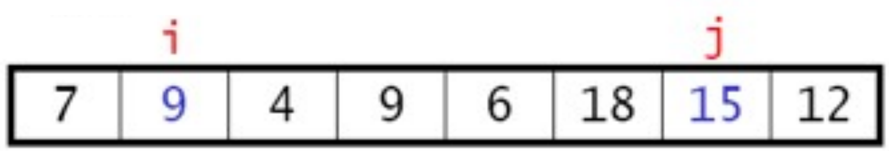
- Maintain indices i and j, starting them “outside” the array:



- *Find* “out-of-place” elements:
  - increment i until arr[i] >= pivot
  - decrement j until arr[j] <= pivot

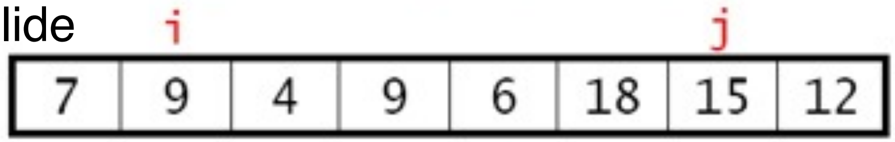


- *Swap* arr[i] and arr[j] if necessary

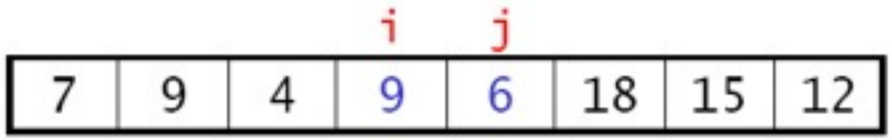


Partitioning An Array

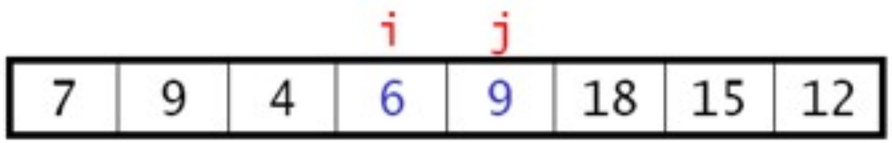
- From previous slide

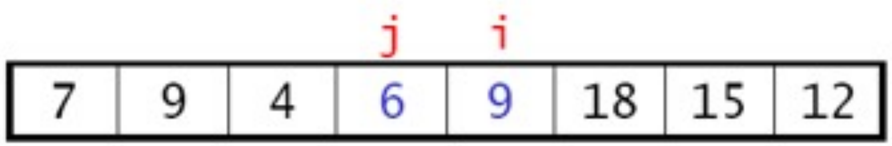


- Advance i and j



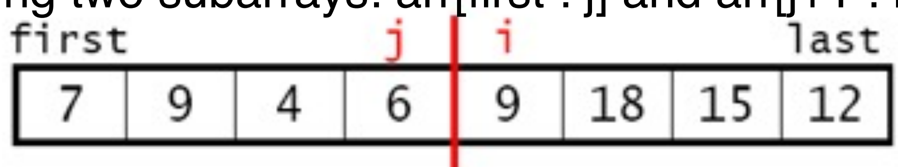
- Need to swap





- i and j cross

- Return j, indicating two subarrays: arr[first : j] and arr[j+1 : last]



An exercise

i j

0	1	2	3	4	5	6	7	8
16	8	13	2	9	15	4	12	24

An exercise

i j

0	1	2	3	4	5	6	7	8
16	8	13	2	9	15	4	12	24

An exercise

i						j		
0	1	2	3	4	5	6	7	8
16	8	13	2	9	15	4	12	24

An exercise

i						j		
0	1	2	3	4	5	6	7	8
4	8	13	2	9	15	16	12	24

An exercise

	i					j		
0	1	2	3	4	5	6	7	8
4	8	13	2	9	15	16	12	24

An exercise

	i		j					
0	1	2	3	4	5	6	7	8
4	8	13	2	9	15	16	12	24

--

An exercise

	i		j					
0	1	2	3	4	5	6	7	8
4	2	13	8	9	15	16	12	24

An exercise

	i		j					
0	1	2	3	4	5	6	7	8
4	2	13	8	9	15	16	12	24

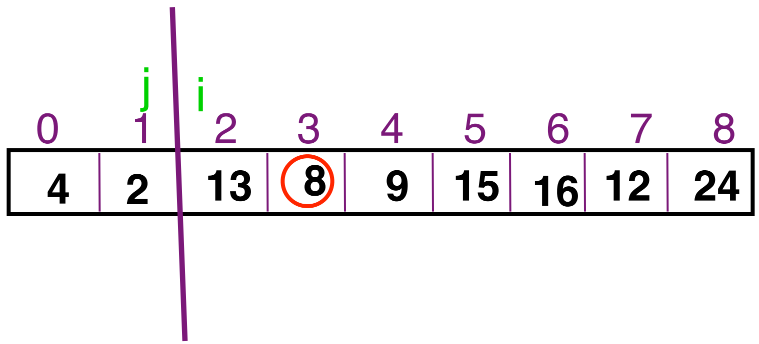
An exercise

		i	j					
0	1	2	3	4	5	6	7	8
4	2	13	8	9	15	16	12	24

An exercise

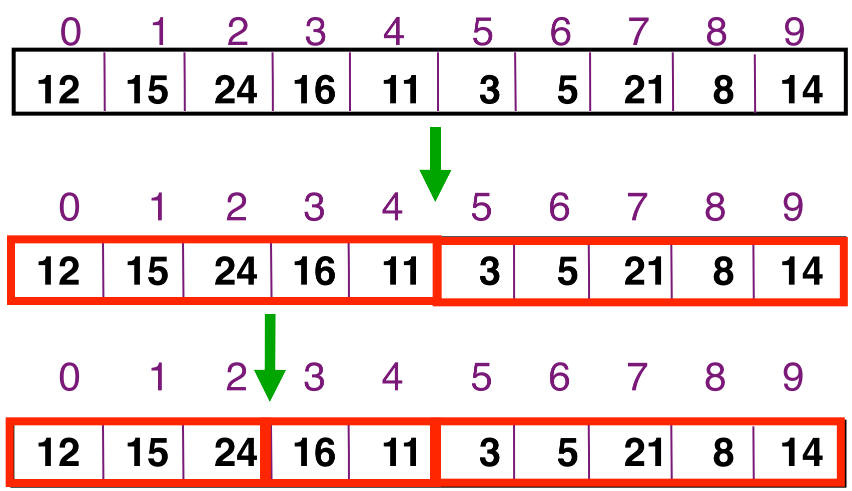
	j	i						
0	1	2	3	4	5	6	7	8
4	2	13	8	9	15	16	12	24

An exercise



Review: Merge-Sort

- Divide-and-conquer
- Split an array into halves, forming two subarrays
- Recursively apply merge-sort to the subarrays
- Merge the sorted subarrays (using a temporary array to contain the sorted array, and copy it back after sorting)



Review: Heap-Sort

- Use the heap data structure, which is an array representation of balanced binary search tree (indeed complete binary tree)

- buildHeap() to heapify an unsorted array in O(N) time

0	1	2	3	4	5	6	7	8
16	**8**	**13**	**2**	**15**	**9**	**4**	**12**	**24**

0	1	2	3	4	5	6	7	8
24	**16**	**13**	**12**	**15**	**9**	**4**	**8**	**2**

- Extract maximum from the max-at-top heap repeatedly and place the maximum to the end

- require O(logN) cost per remove() after buildHeap()

Comparisons

Algorithm	**Best-case**	**Worst-case**	**Average-case**	**Extra space**
Selection sort	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$
Bubble sort	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$
Insertion sort	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$
Shell-sort	$O(n*logn)$	$O(n^{1.5})$	$O(n^{1.5})$, but maybe $O(n^{1.25})$	$O(1)$
Quick-sort	$O(n*logn)$	$O(n^2)$	$O(n*logn)$	$O(logn)$
Merge-sort	$O(n*logn)$	$O(n*logn)$	$O(n*logn)$	$O(n)$
Heap-sort	$O(n*logn)$	$O(n*logn)$	$O(n*logn)$	$O(1)$

Always use heap-sort?

- heap-sort: better worst case than quick-sort: O(nlogn) vs O(n^2)
- heap-sort: also better memory: O(1) vs O(logn)
- Why don't we always use heap-sort?
  - average case is both O(nlogn), but quick-sort has lower constant
  - quick sort makes more efficient use of cache, paralellizes better
- Use heap-sort when good worst-case performance is critical
  - real-time embedded systems
  - high security systems (poor worst-case presents a potential risk)

Applications of Sorting

- Sorting is used extensively in computer systems and various applications

- Recall the selection problem: Given a list of N elements and an integer k (1 <= k <= N), find out the k-th largest element in the list.
  - We have solutions to utilize heap data structures to achieve reasonable running time of O(N + k*logN)

- Can we do better?

- What information are we learning that we don’t need?

Review: The Selection Problem

- Given a list of N items and an integer k (1 <= k <= N), find out the k-th largest item (or k largest items) in the list. E.g., N=1,000,000 and k=100, or k=N/2

- Example:
  - List of items: 4, 9, 0, 3, 5, 7, 10, 12, 2, 8

    12 10 9 8 7 5 4 3 2 0 (sorted order of items)

  - 1st largest item is: 12
  - 10th largest item is: 0
  - 6th largest item is: 5

- What is your method?

The Selection Problem: Naïve Methods

- Don’t think about it method 1
  - Sort N items: $O(N^2)$ (for a simple sort algorithm), O(NlogN) for heap-sort (and a number of other algorithms we will learn)
  - Retrieve the k-th largest item: O(1)

- Some thought method 2
  - Read k items into an array: O(k)
  - Sort the items in the array: $O(k^2)$ or O(k log k)
  - For each of N-k remaining items: compare to the last array item:
    - ✓ If larger, replace last array item with new element and put new element into correct spot in array: O(k)
  - Finally the remaining k items are the results
  - Total Running time: $O(k + k^2 + (N-k) * k) = O(Nk)$.
  - Best case is when k is small.
  - Worst case is k = N/2, which also most useful case (ie the median)

The Selection Problem: Efficient Method 1

- The easy-if-you-know-heaps method
- To find the k-th largest item.
  - Read N items into an array   O(?)
  - Apply buildHeap() to the array   O(?)
  - Perform k remove() operations. O(?)
    - ✓ Last operation will give us the k-th largest one.
- What is the total running time? O(?)

The Selection Problem: Efficient Method 1

- The easy-if-you-know-heaps method
- To find the k-th largest item.
  - Read N items into an array O(N)
  - Apply buildHeap() to the array O(?)
  - Perform k remove() operations. O(?)
    - ✓ Last operation will give us the k-th largest one.
- What is the total running time? O(?)

The Selection Problem: Efficient Method 1

- The easy-if-you-know-heaps method

- To find the k-th largest item.
  - Read N items into an array   O(N)
  - Apply buildHeap() to the array   O(N)
  - Perform k remove() operations. O(?)
    - ✓ Last operation will give us the k-th largest one.

- What is the total running time? O(?)

The Selection Problem: Efficient Method 1

- The easy-if-you-know-heaps method
- To find the k-th largest item.
  - Read N items into an array   O(N)
  - Apply buildHeap() to the array   O(N)
  - Perform k remove() operations. O(k log N)
    - ✓ Last operation will give us the k-th largest one.
- What is the total running time? O(?)

The Selection Problem: Efficient Method 1

- The easy-if-you-know-heaps method
- To find the k-th largest item.
  - Read N items into an array   O(N)
  - Apply buildHeap() to the array   O(N)
  - Perform k remove() operations. O(k log N)
    - ✓ Last operation will give us the k-th largest one.
- What is the total running time? O(N + k log N)
  O(N) for small k
  O(k log N) for large k
  O(N log N) for median (k=N/2)

The Selection Problem: Efficient Method 2

- From the idea of the second naïve method: at any time maintain a set S of k largest items.

- To find the k-th largest item.
  - Read k items into a min-at-top heap S (of size k). O(?)
  - For each remaining item (N-k) of them
    - ✓ Compare it with the smallest item (root) in heap S
    - ✓ If item is larger than root, then put it into S instead of root.
    - ✓ Sift down the root if necessary. O(?)

- Running time: O(k+(N-k)logk) = O(?)
  - Compared to O(N + klogN) of Efficient Method 1
  - Which is better?

The Selection Problem: Efficient Method 2

- From the idea of the second naïve method: at any time maintain a set S of k largest items.

- To find the k-th largest item.
  - Read k items into a min-at-top heap S (of size k). O(k)
  - For each remaining item (N-k) of them
    - ✓ Compare it with the smallest item (root) in heap S
    - ✓ If item is larger than root, then put it into S instead of root.
    - ✓ Sift down the root if necessary. O(?)

- Running time: O(k+(N-k)logk) = O(?)
  - Compared to O(N + klogN) of Efficient Method 1
  - Which is better?

The Selection Problem: Efficient Method 2

- From the idea of the second naïve method: at any time maintain a set S of k largest items.

- To find the k-th largest item.
  - Read k items into a min-at-top heap S (of size k). O(k)
  - For each remaining item (N-k) of them
    - ✓ Compare it with the smallest item (root) in heap S
    - ✓ If item is larger than root, then put it into S instead of root.
    - ✓ Sift down the root if necessary. O(log k)

- Running time: O(k+(N-k)logk) = O(?)
  - Compared to O(N + klogN) of Efficient Method 1
  - Which is better?

The Selection Problem: Efficient Method 2

- From the idea of the second naïve method: at any time maintain a set S of k largest items.

- To find the k-th largest item.
  - Read k items into a min-at-top heap S (of size k). O(k)
  - For each remaining item (N-k) of them
    - Compare it with the smallest item (root) in heap S
    - If item is larger than root, then put it into S instead of root.
    - Sift down the root if necessary. O(log k)

- Running time: O(k+(N-k)logk) = O(k + N log k - k log k)
  - Compared to O(N + klogN) of Efficient Method 1
  - Which is better?

The Selection Problem: Efficient Method 2

- From the idea of the second naïve method: at any time maintain a set S of k largest items.

- To find the k-th largest item.
  - Read k items into a min-at-top heap S (of size k). O(k)
  - For each remaining item (N-k) of them
    - ✓ Compare it with the smallest item (root) in heap S
    - ✓ If item is larger than root, then put it into S instead of root.
    - ✓ Sift down the root if necessary. O(log k)

- Running time: O(k+(N-k)logk) = O(k + N log k - k log k)
  - Compared to O(N + klogN) of Efficient Method 1
    = O(N log k) for k large or small
    = O(N log N) for median
  - Which is better?

Quick-sort for The Selection Problem

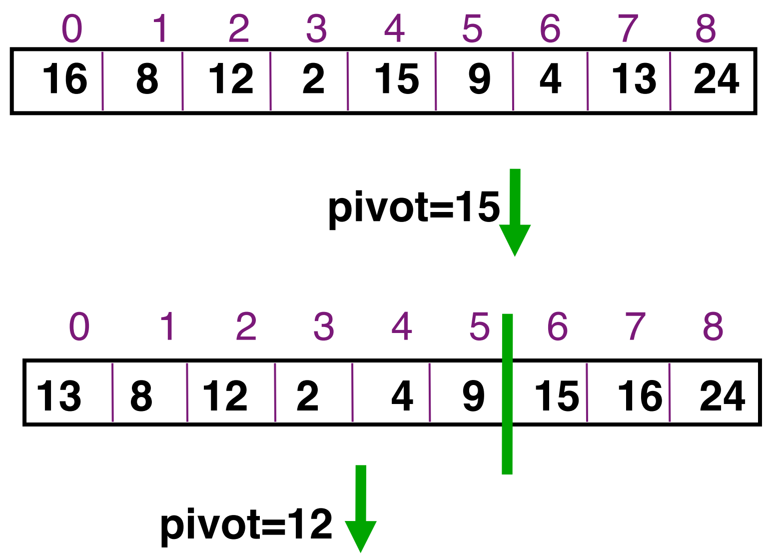
- We can solve the selection problem more efficiently (at least theoretically, needs only linear time O(N))

1. **Pick a pivot value**
2. **Divide: partition the array into left and right subarrays**
3. **Conquer:**
   a) **If (k < right.length), find the kth largest in right recursively**
   b) **If (k > right.length), find the (k - right.length)th largest in left recursively**

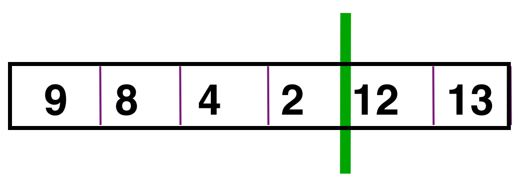
   Either way, we make only one recursive call instead of two, like in quicksort.

Example

- Find the 5th largest value in



- Find the 2nd largest value in left subarray *only*



- Find the 2nd largest value in right subarray *only*

….

Example

- Find the 5th largest value in

0	1	2	3	4	5	6	7	8
16	8	12	2	15	9	4	13	24

pivot=15

0	1	2	3	4	5	6	7	8
13	8	12	2	4	9	15	16	24

pivot=12

- Find the 2nd largest value in left subarray *only*

9	8	4	2	12	13

- Find the 2nd largest value in right subarray *only*

….

Best- and average-case linear time complexity!

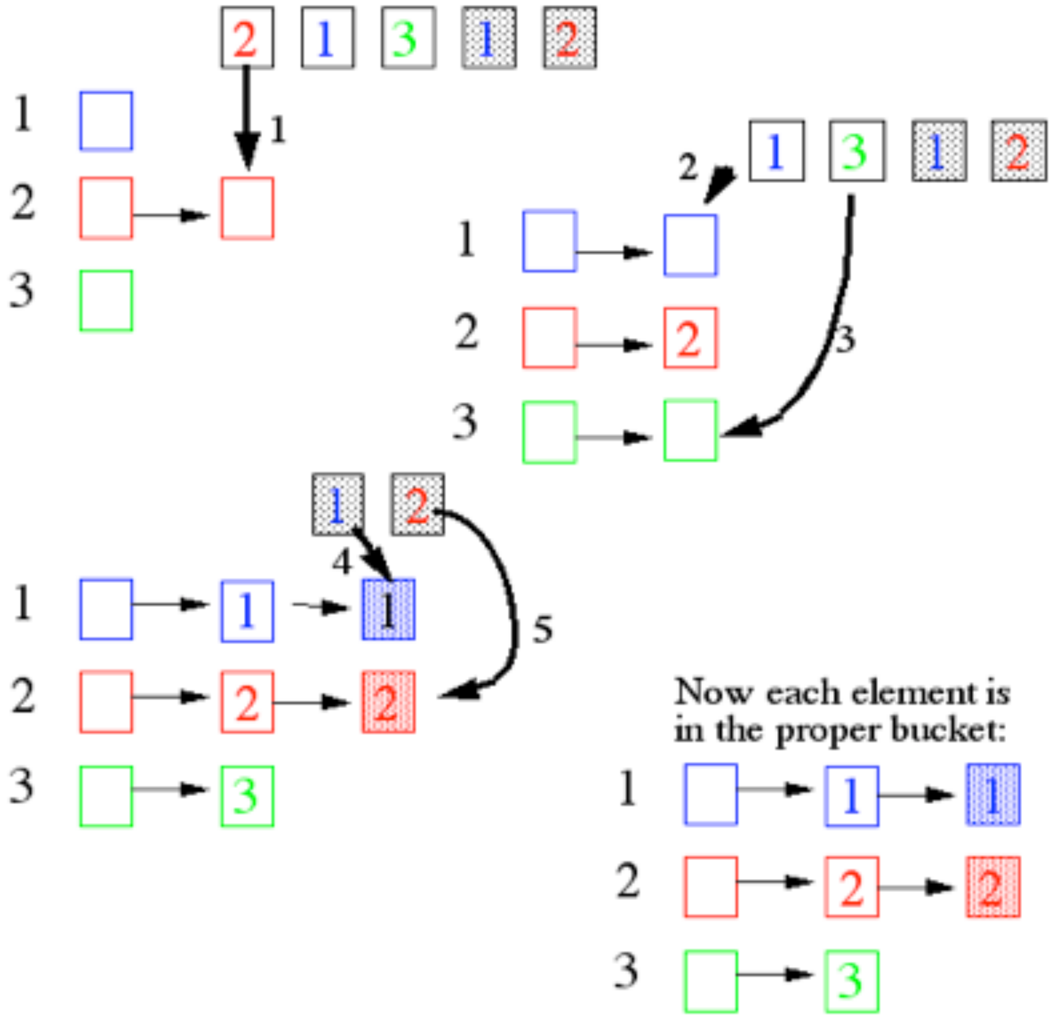
But poor worst-case complexity

Bucket Sort

- Bucket sort
  - Assumption: integer keys in the range [0, M)
  - Basic idea:
    1. Create *M* linked lists (*buckets*), one for each possible key value
    2. Add each input element to appropriate bucket
    3. Concatenate the buckets
  - Expected total time is O(M+N), with N = size of original sequence
  - if M=O(N), then sorting algorithm in O(N)

- Remember hash tables also uses buckets?
  - Bucket sort preserves order (key values) in buckets
  - Hashing mixes up elements with diverse key values

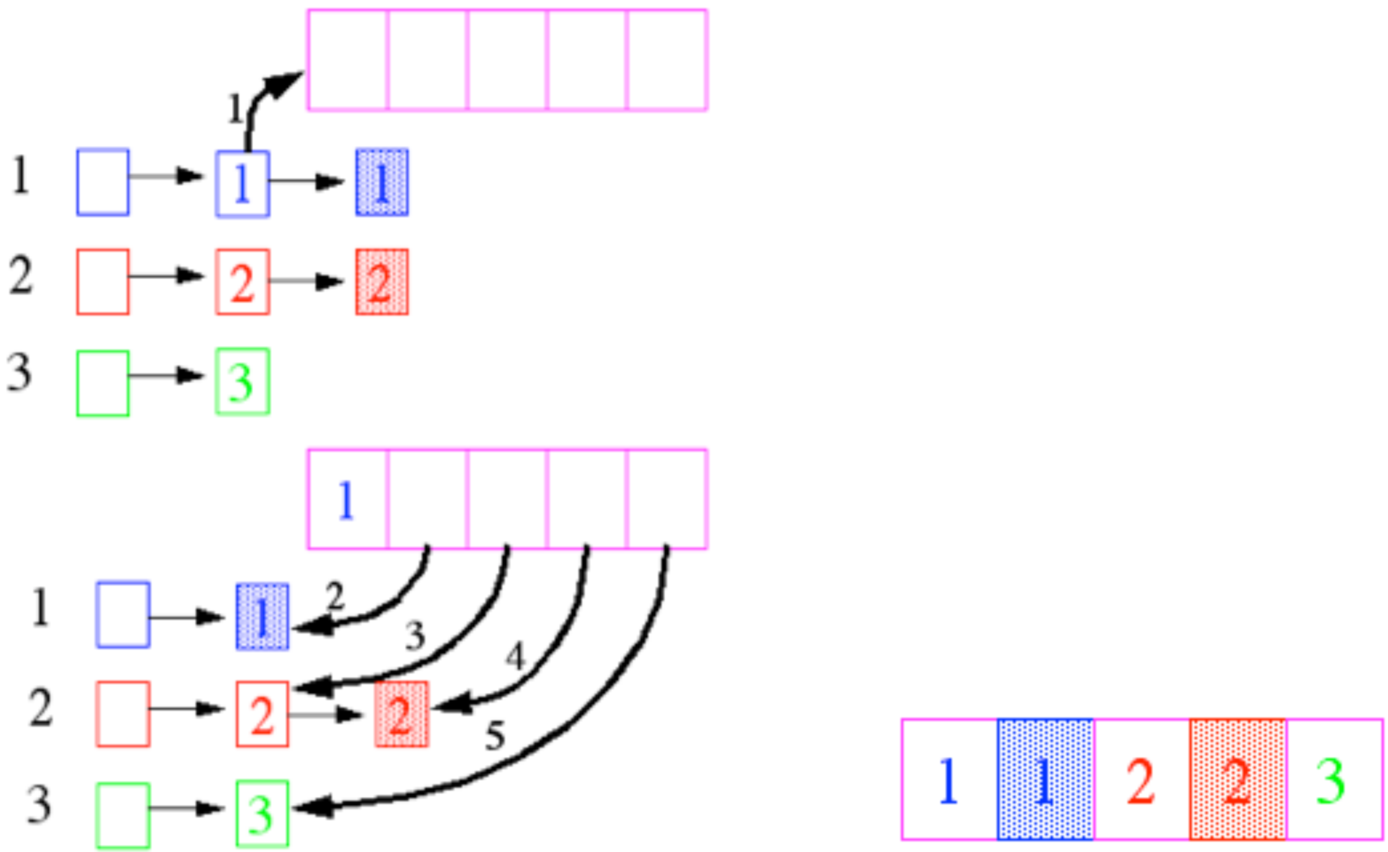
Bucket Sort Example



--

Bucket Sort Example

- Pull the elements from the buckets into the array



Keys of Non-integer Types?

- What if keys are not integers?
  - Assumption: input is *N* floating numbers (scaled) in [0, 1]
  - Basic idea:
    - Create *M* linked lists (*buckets*) to divide interval [0,1] into subintervals of size 1/*M*
    - Add each input element to appropriate bucket and sort the bucket with insertion sort
  - Choose M=O(N)
  - Uniform input distribution → expected bucket size is O(1)
    - Therefore the expected total time is O(N): O(N) to put elements into the buckets and O(N) to move them back into the array

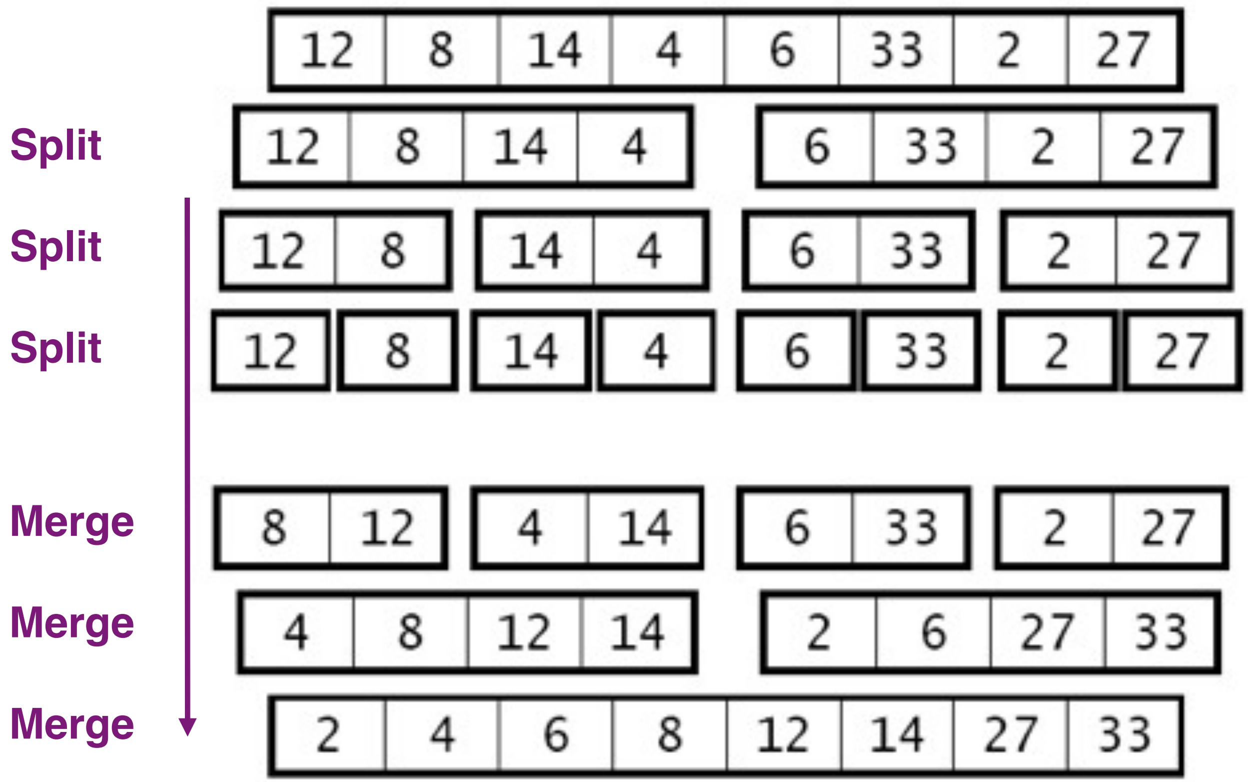
- With uniform key distribution, Bucket Sort has O(N)=o(NlogN) running time
- But sensitive to the key distribution in the range (it may not be uniform)
- Pays with space for time

External Sorting

- Example problem: to sort 1TB of data with 1GB of RAM.
- Elements will be swapped in and out of RAM – expensive!
- Objectives
  - Primary: reduce the cost due to disk I/Os
  - Secondary: CPU processing time

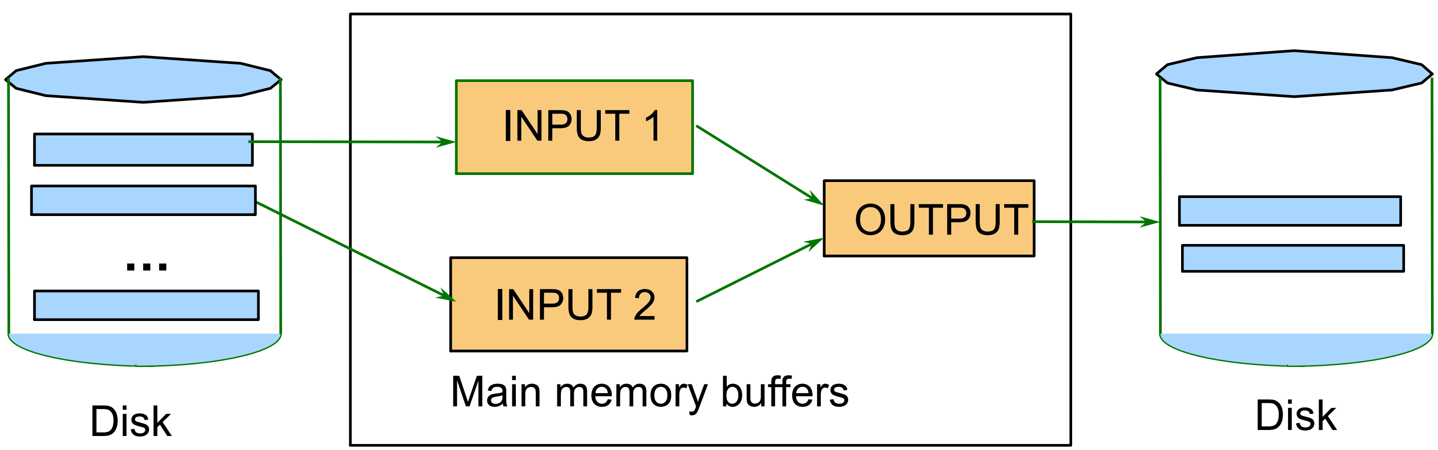
recall Merge-Sort

- merge-sort recursively divides arrays smaller elements
- then progressively merges subarrays into larger arrays



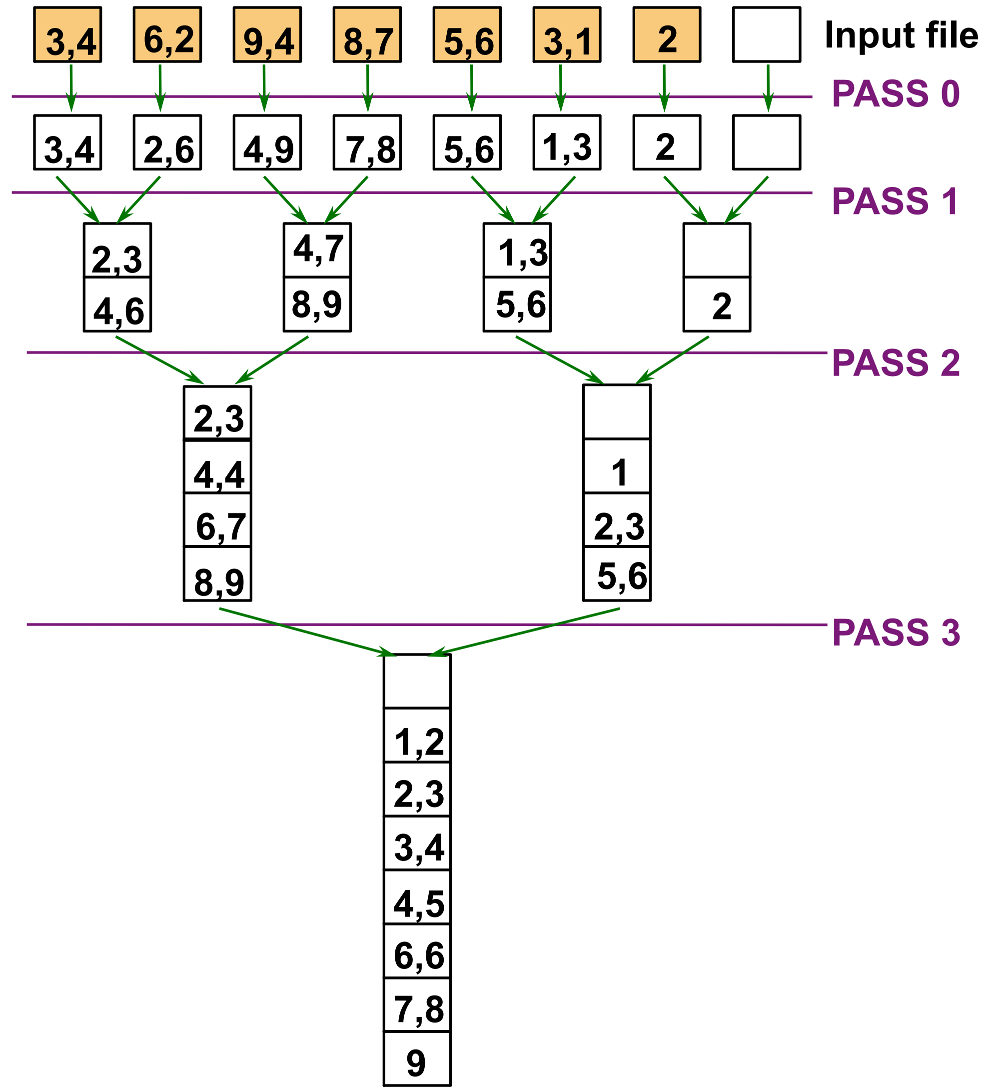
Two-Way External Merge-Sort: The Idea

- Pass 0: Read a page, sort it, and write it back to the disk.
  - only one buffer page is used
  - can use any internal sorting method
- Pass 1, 2, …, etc.:
  - requires 3 buffers
  - merge pairs of runs into runs twice as long



Two-Way External Merge-Sort

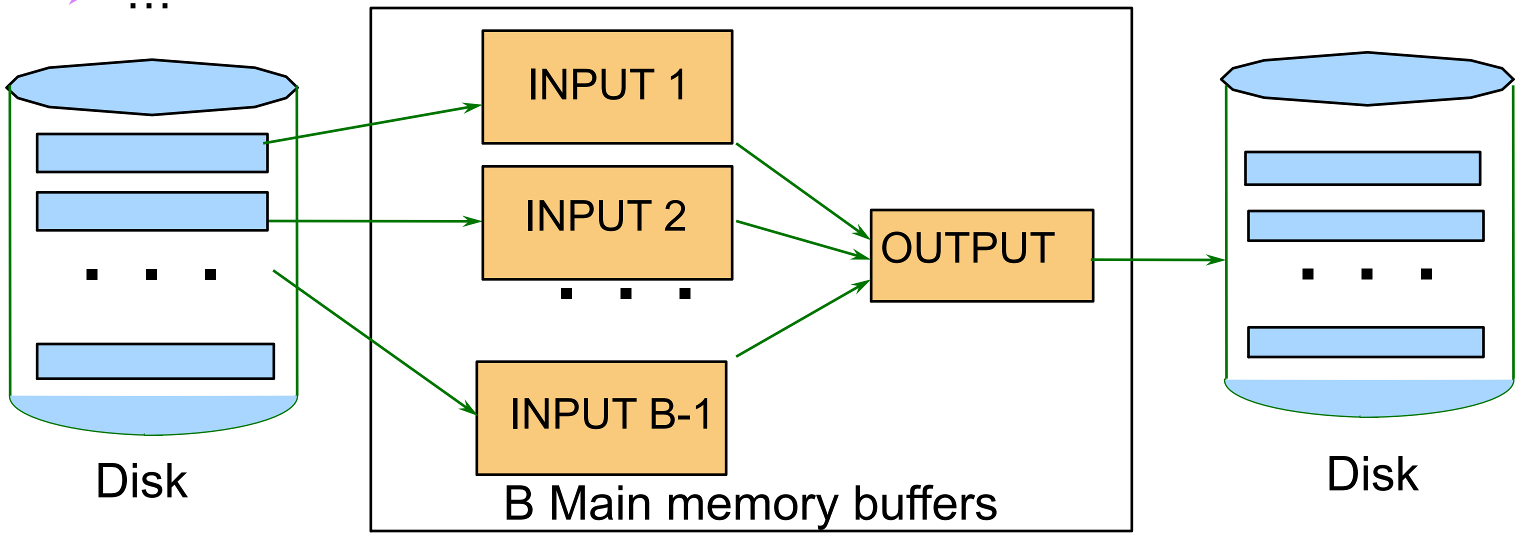
- N pages in the file
- In each pass we read and write each page in file.
  - 2N page I/O operations
- The number of passes $= \lceil \log_2 N \rceil + 1$
- So total cost, in number of page I/Os, is: $2N(\lceil \log_2 N \rceil + 1)$



Multi-Way External Merge-Sort

- We have more than 3 buffer pages, allowing multi-way merge
- To sort a file with *N* pages using *B* buffer pages:
  - Pass 0: Use 1 buffer. Read every page, sort, write back.
  - Pass 1: Use (B-1) input buffers and 1 output buffer.

    Merge every bunch of *B-1* pages into a sorted run of *(B-1)* pages each.
  - Pass 2: merge every bunch of *B-1* runs into a sorted run of $(B-1)^2$ pages each
  - Pass 3: merge every bunch of *B-1* runs into a sorted run of $(B-1)^3$ pages each
  - …



Multi-Way External Merge-Sort: Cost

- Number of passes: $1+\lceil \log_{B-1} N \rceil$
  - N=108, B=5
    $1 + \lceil \log_{B-1} N \rceil = 1 + \lceil \log(108)/\log(4) \rceil = \lceil 4.38 \rceil = 5$

- Cost (in number of page I/Os):
  - 2N*(number of passes)

- Example: with B=5 buffer pages, to sort 108 page file:
  - Pass 0: 216 page I/Os to prepare sorted pages
  - Then do four-way merges:
  - Pass 1: produce $\lceil 108/4 \rceil$ = 27 sorted runs of 4 pages each
  - Pass 2: $\lceil 27/4 \rceil$ = 7 sorted runs of 16 pages each (last run is only 12 pages)
  - Pass 3: $\lceil 7/4 \rceil$ = 2 sorted runs, 64 pages and 44 pages
  - Pass 4: Sorted file of 108 pages

Comparisons

Algorithm	Best-case	Worst-case	Average-case	Extra space
Selection sort	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$
Bubble sort	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$
Insertion sort	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$
Shell-sort	$O(n*logn)$	$O(n^{1.5})$	$O(n^{1.5})$, but maybe $O(n^{1.25})$	$O(1)$
Quick-sort	$O(n*logn)$	$O(n^2)$	$O(n*logn)$	$O(logn)$
Merge-sort	$O(n*logn)$	$O(n*logn)$	$O(n*logn)$	$O(n)$
Heap-sort	$O(n*logn)$	$O(n*logn)$	$O(n*logn)$	$O(1)$
Bucket-sort (for numbers)	$O(n)$	$O(n)$	$O(n)$	$O(n)$
External sort	$2N * \lceil \log_{B-1} N \rceil$ page I/O's			B memory buffers

End of sorting

- Next time new data structure: graphs