CS61B Lectures #27

Today:

- Merge sorts
- Quicksort

Readings: Today: *DS(IJ)*, Chapter 8; Next topic: Chapter 9.

Merge Sorting

Idea: Divide data into subsequences; recursively sort the subsequences; merge results.

- We've already seen the analysis (Lecture #16): $\Theta(N \lg N)$.
- Good for *external sorting:*
  - First break the data into small enough chunks to fit in memory and sort each.
  - Then repeatedly merge into bigger and bigger sequences.
- Can merge K sorted sequences of *arbitrary size* on secondary storage using $\Theta(K)$ storage:

```
Data[] V = new Data[K];
For all i, set V[i] to the first data item of sequence i;
while there is data left to sort:
    Find k so that V[k] has data and is smallest;
    Write V[k] to the output sequence;
    If there is more data in sequence k, read it into V[k],
        otherwise, clear V[k];
```

Merge Sorting Illustrated

Input:	55	20	31	80	58	25	-4	34	16	8	61	39	35	42	60

Merge Sorting Illustrated

Input:

55	20	31	80	58	25	-4	34	16	8	61	39	35	42	60

Input subsequences:

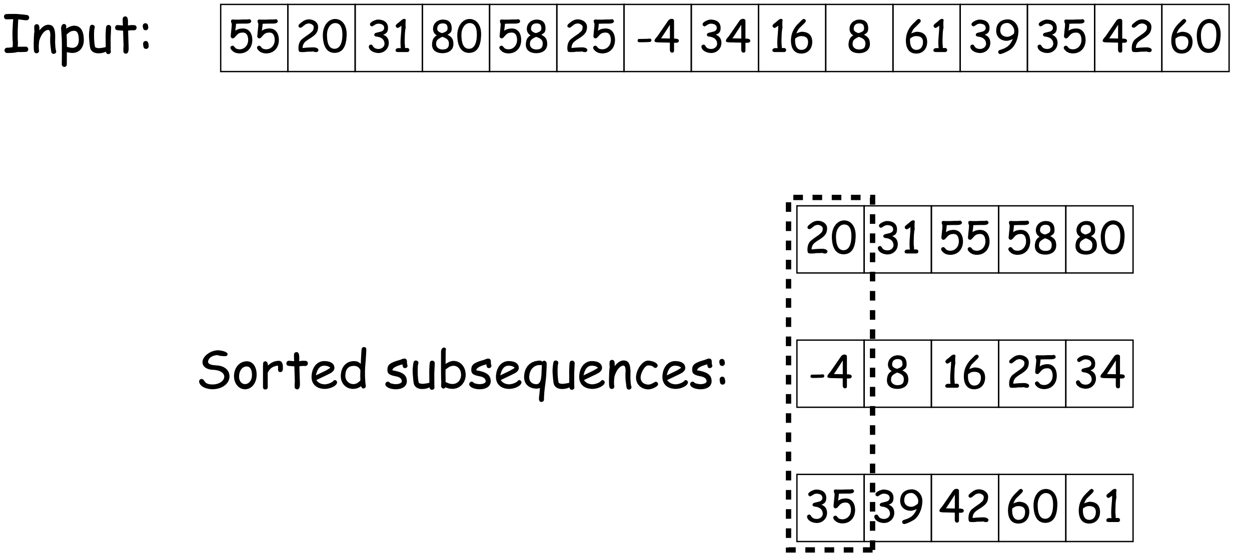
55	20	31	80	58

25	-4	34	16	8

61	39	35	42	60

- First, the input data sequence is divided into subsequences.

Merge Sorting Illustrated



Input: 55 20 31 80 58 25 -4 34 16 8 61 39 35 42 60

Sorted subsequences:

20 31 55 58 80

-4 8 16 25 34

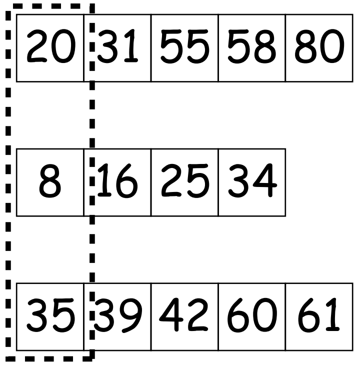
35 39 42 60 61

- First, the input data sequence is divided into subsequences.
- Next, the subsequences are themselved sorted (possibly by a recursive application of mergesort.)

Merge Sorting Illustrated

Input: 55 20 31 80 58 25 -4 34 16 8 61 39 35 42 60

Remaining subsequences:



20 31 55 58 80

8 16 25 34

35 39 42 60 61

Result: -4

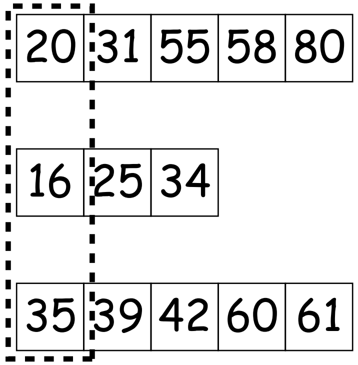
- The dashed window shows the input data that must be in memory at any given time. It is of constant size, no matter what the size of the input.
- One by one, the smallest item in the dashed window is removed from its sequence and added to the result.

Merge Sorting Illustrated

Input:

55	20	31	80	58	25	-4	34	16	8	61	39	35	42	60

Remaining subsequences:



20	31	55	58	80
16	25	34		
35	39	42	60	61

Result:

-4	8

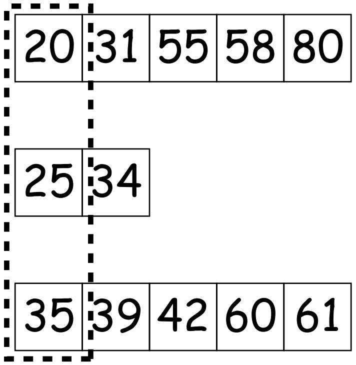
- The dashed window shows the input data that must be in memory at any given time. It is of constant size, no matter what the size of the input.
- One by one, the smallest item in the dashed window is removed from its sequence and added to the result.

Merge Sorting Illustrated

Input:

55	20	31	80	58	25	-4	34	16	8	61	39	35	42	60

Remaining subsequences:



20 31 55 58 80

25 34

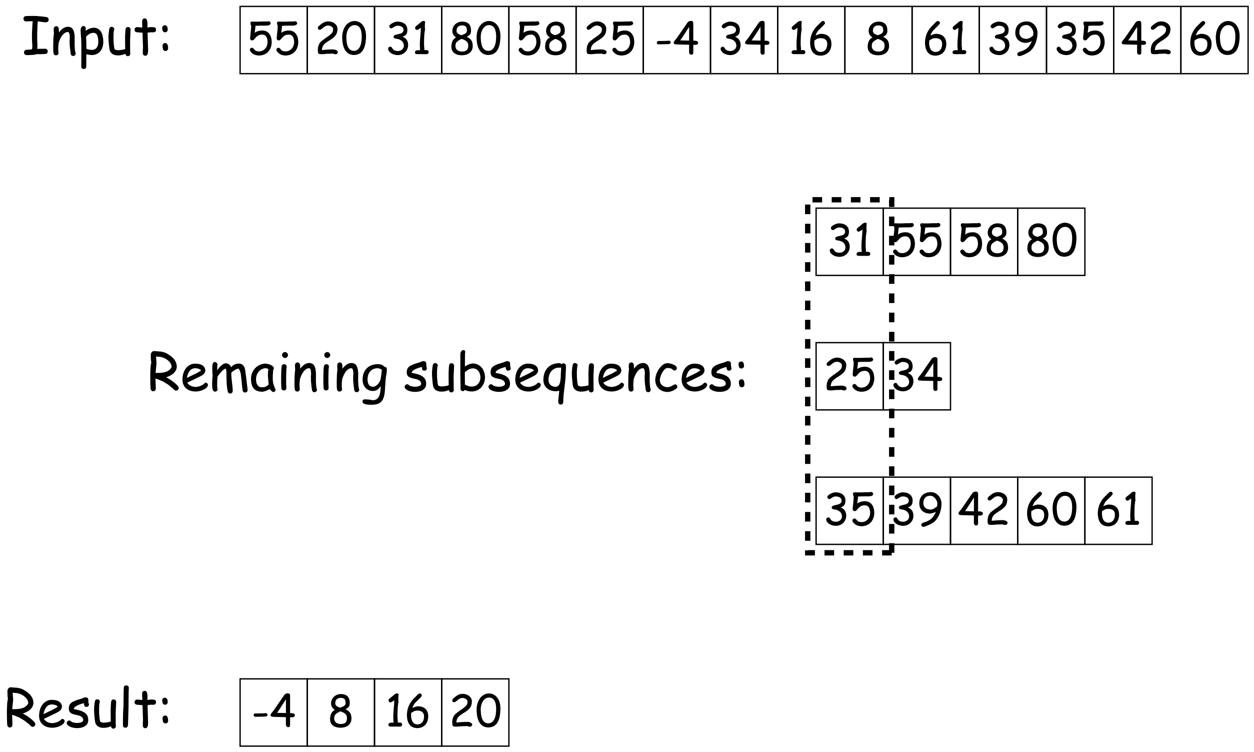
35 39 42 60 61

Result:

-4	8	16

- The dashed window shows the input data that must be in memory at any given time. It is of constant size, no matter what the size of the input.
- One by one, the smallest item in the dashed window is removed from its sequence and added to the result.

Merge Sorting Illustrated



Input: 55 20 31 80 58 25 -4 34 16 8 61 39 35 42 60

Remaining subsequences:

31 55 58 80

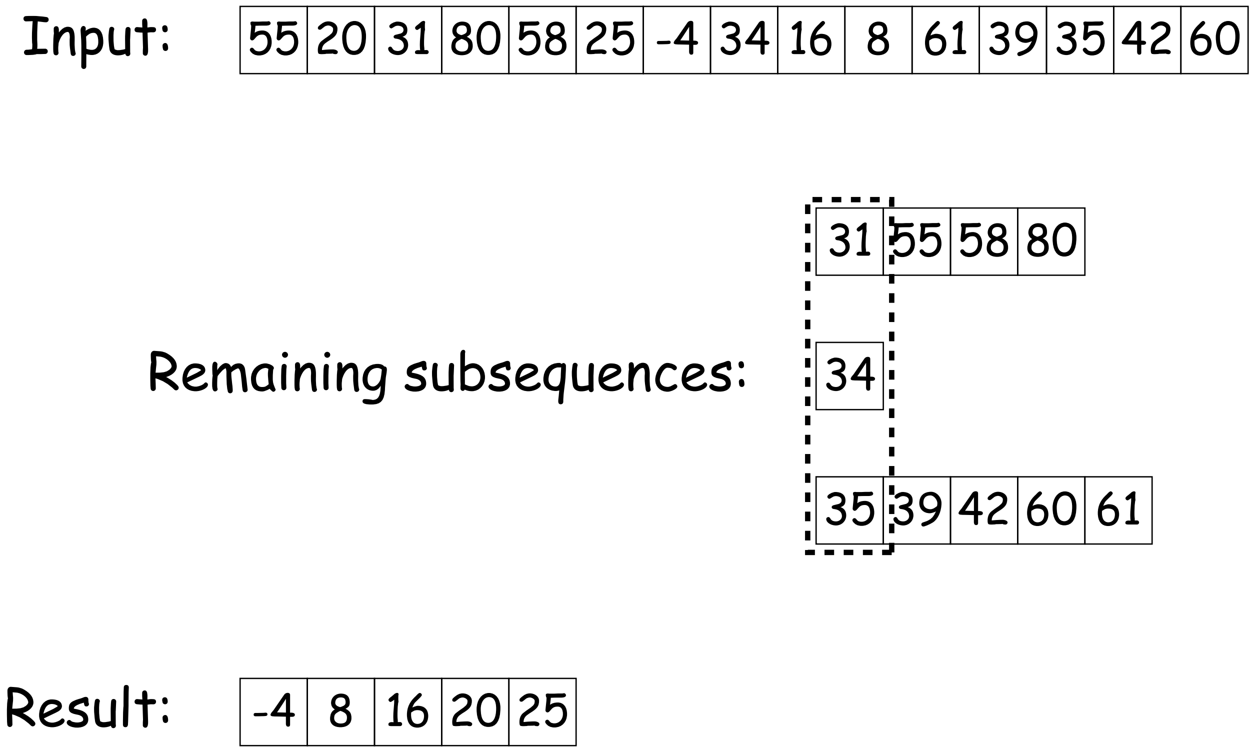
25 34

35 39 42 60 61

Result: -4 8 16 20

- The dashed window shows the input data that must be in memory at any given time. It is of constant size, no matter what the size of the input.
- One by one, the smallest item in the dashed window is removed from its sequence and added to the result.

Merge Sorting Illustrated



Input: 55 20 31 80 58 25 -4 34 16 8 61 39 35 42 60

Remaining subsequences:

31 55 58 80

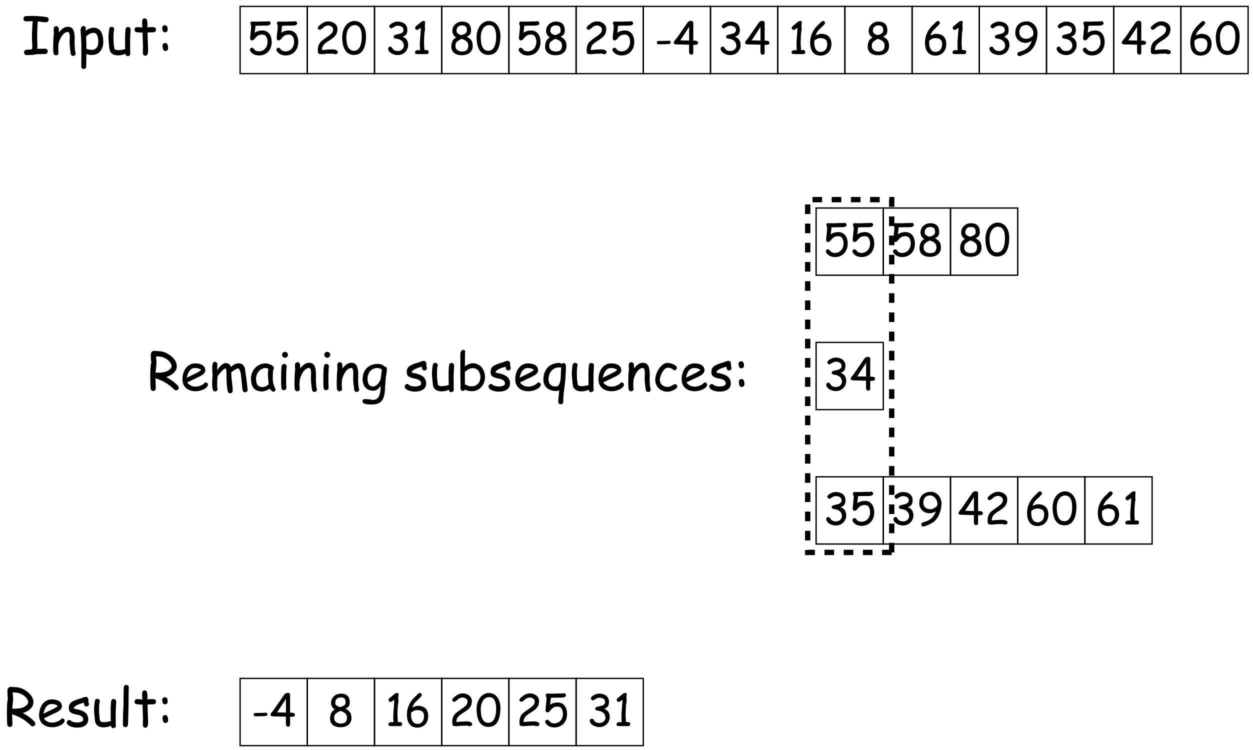
34

35 39 42 60 61

Result: -4 8 16 20 25

- The dashed window shows the input data that must be in memory at any given time. It is of constant size, no matter what the size of the input.
- One by one, the smallest item in the dashed window is removed from its sequence and added to the result.

Merge Sorting Illustrated



Input: 55 20 31 80 58 25 -4 34 16 8 61 39 35 42 60

Remaining subsequences:

55 58 80

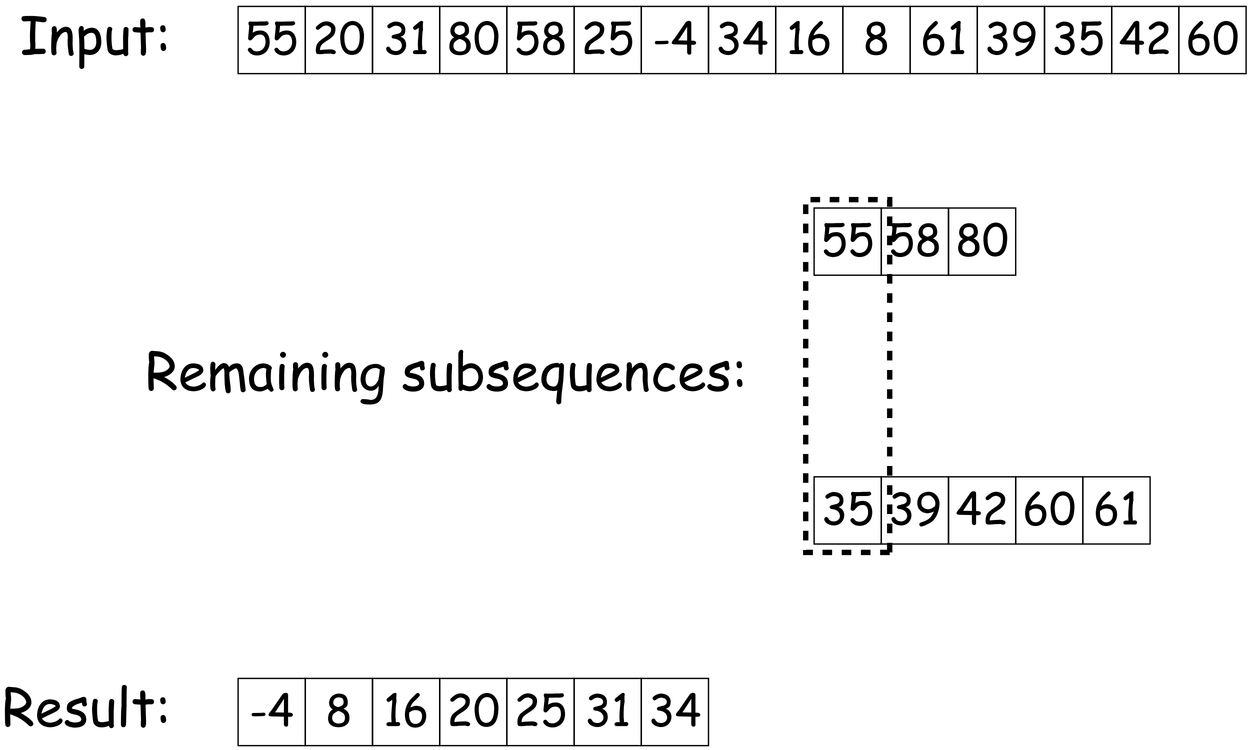
34

35 39 42 60 61

Result: -4 8 16 20 25 31

- The dashed window shows the input data that must be in memory at any given time. It is of constant size, no matter what the size of the input.
- One by one, the smallest item in the dashed window is removed from its sequence and added to the result.

Merge Sorting Illustrated



Input: 55 20 31 80 58 25 -4 34 16 8 61 39 35 42 60

Remaining subsequences:

55 58 80

35 39 42 60 61

Result: -4 8 16 20 25 31 34

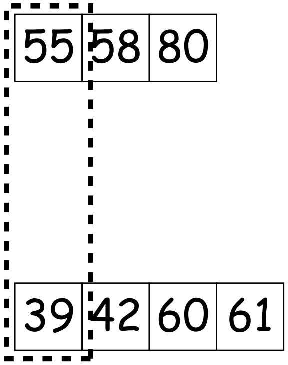
- The dashed window shows the input data that must be in memory at any given time. It is of constant size, no matter what the size of the input.
- One by one, the smallest item in the dashed window is removed from its sequence and added to the result.

Merge Sorting Illustrated

Input:

55	20	31	80	58	25	-4	34	16	8	61	39	35	42	60

Remaining subsequences:



Result:

-4	8	16	20	25	31	34	35

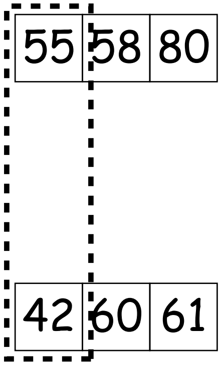
- The dashed window shows the input data that must be in memory at any given time. It is of constant size, no matter what the size of the input.
- One by one, the smallest item in the dashed window is removed from its sequence and added to the result.

Merge Sorting Illustrated

Input:

55	20	31	80	58	25	-4	34	16	8	61	39	35	42	60

Remaining subsequences:



Result:

-4	8	16	20	25	31	34	35	39

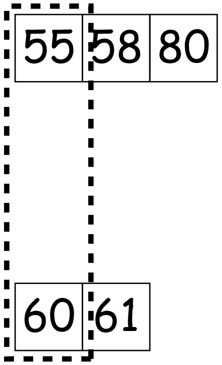
- The dashed window shows the input data that must be in memory at any given time. It is of constant size, no matter what the size of the input.
- One by one, the smallest item in the dashed window is removed from its sequence and added to the result.

Merge Sorting Illustrated

Input:

55	20	31	80	58	25	-4	34	16	8	61	39	35	42	60

Remaining subsequences:



Result:

-4	8	16	20	25	31	34	35	39	42

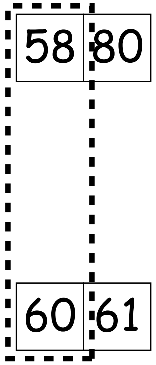
- The dashed window shows the input data that must be in memory at any given time. It is of constant size, no matter what the size of the input.
- One by one, the smallest item in the dashed window is removed from its sequence and added to the result.

Merge Sorting Illustrated

Input:

55	20	31	80	58	25	-4	34	16	8	61	39	35	42	60

Remaining subsequences:



58	80
60	61

Result:

-4	8	16	20	25	31	34	35	39	42	55

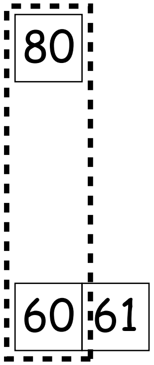
- The dashed window shows the input data that must be in memory at any given time. It is of constant size, no matter what the size of the input.
- One by one, the smallest item in the dashed window is removed from its sequence and added to the result.

Merge Sorting Illustrated

Input:

55	20	31	80	58	25	-4	34	16	8	61	39	35	42	60

Remaining subsequences:



Result:

-4	8	16	20	25	31	34	35	39	42	55	58

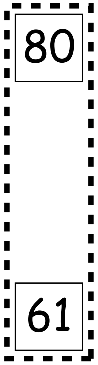
- The dashed window shows the input data that must be in memory at any given time. It is of constant size, no matter what the size of the input.
- One by one, the smallest item in the dashed window is removed from its sequence and added to the result.

Merge Sorting Illustrated

Input:

55	20	31	80	58	25	-4	34	16	8	61	39	35	42	60

Remaining subsequences:

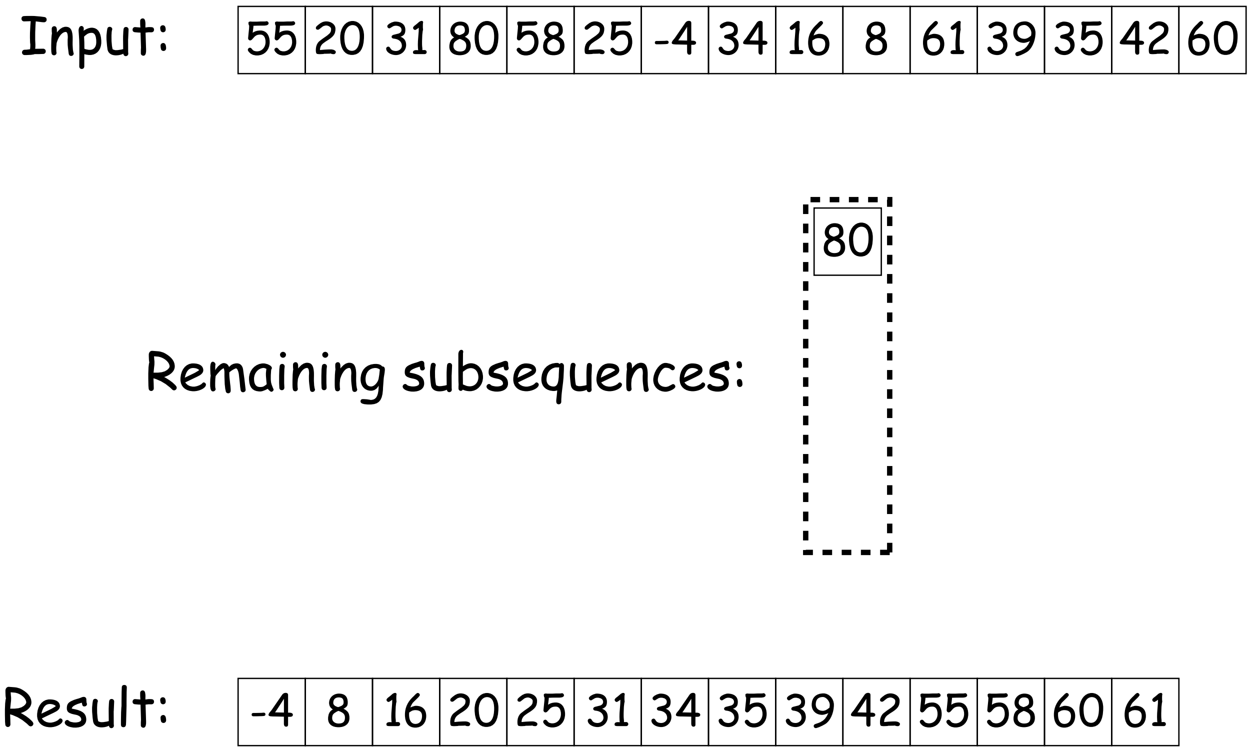


Result:

-4	8	16	20	25	31	34	35	39	42	55	58	60

- The dashed window shows the input data that must be in memory at any given time. It is of constant size, no matter what the size of the input.
- One by one, the smallest item in the dashed window is removed from its sequence and added to the result.

Merge Sorting Illustrated



Input:

55	20	31	80	58	25	-4	34	16	8	61	39	35	42	60

Remaining subsequences:

80

Result:

-4	8	16	20	25	31	34	35	39	42	55	58	60	61

- The dashed window shows the input data that must be in memory at any given time. It is of constant size, no matter what the size of the input.
- One by one, the smallest item in the dashed window is removed from its sequence and added to the result.

Merge Sorting Illustrated

Input:

55	20	31	80	58	25	-4	34	16	8	61	39	35	42	60

Remaining subsequences:

Result:

-4	8	16	20	25	31	34	35	39	42	55	58	60	61	80

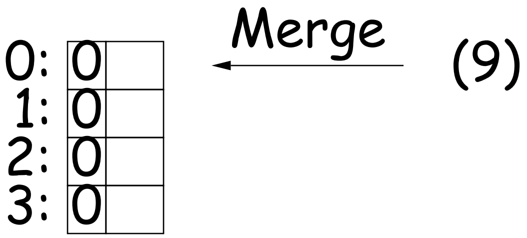
- The dashed window shows the input data that must be in memory at any given time. It is of constant size, no matter what the size of the input.
- One by one, the smallest item in the dashed window is removed from its sequence and added to the result.

Illustration of Internal Merge Sort

For internal sorting, we can use a *binomial comb* to orchestrate an iterative merge sort.

- Start with $\lg N + 1$ buckets that can contain sublists, initially empty.
- Bucket #k is either empty or contains 2^k sorted items at any time.
- For each item in the input list, turn it into a 1-element list, and merge it into bucket 0 (or simply put it in bucket 0 if that is empty).
- You will only merge lists of length 2^k into bucket k. Whenever that gives a list of size 2^{k+1}, merge it into bucket $k+1$ and clear bucket k (and so on as needed with buckets $k+2$, etc.)
- When all inputs are processed, merge all the buckets into the final list.

L: (9, 15, 5, 3, 0, 6, 10, -1, 2, 20, 8)



0: 0

1: 0

2: 0

3: 0

Merge

(9)

Illustration of Internal Merge Sort

For internal sorting, we can use a *binomial comb* to orchestrate an iterative merge sort.

- Start with $\lg N + 1$ buckets that can contain sublists, initially empty.
- Bucket #k is either empty or contains 2^k sorted items at any time.
- For each item in the input list, turn it into a 1-element list, and merge it into bucket 0 (or simply put it in bucket 0 if that is empty).
- You will only merge lists of length 2^k into bucket k. Whenever that gives a list of size 2^{k+1}, merge it into bucket $k+1$ and clear bucket k (and so on as needed with buckets $k+2$, etc.)
- When all inputs are processed, merge all the buckets into the final list.

L: (9, 15, 5, 3, 0, 6, 10, -1, 2, 20, 8)

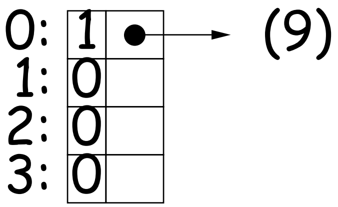


Illustration of Internal Merge Sort

For internal sorting, we can use a *binomial comb* to orchestrate an iterative merge sort.

- Start with $\lg N + 1$ buckets that can contain sublists, initially empty.
- Bucket #k is either empty or contains 2^k sorted items at any time.
- For each item in the input list, turn it into a 1-element list, and merge it into bucket 0 (or simply put it in bucket 0 if that is empty).
- You will only merge lists of length 2^k into bucket k. Whenever that gives a list of size 2^{k+1}, merge it into bucket $k+1$ and clear bucket k (and so on as needed with buckets $k+2$, etc.)
- When all inputs are processed, merge all the buckets into the final list.

L: (9, 15, 5, 3, 0, 6, 10, -1, 2, 20, 8)

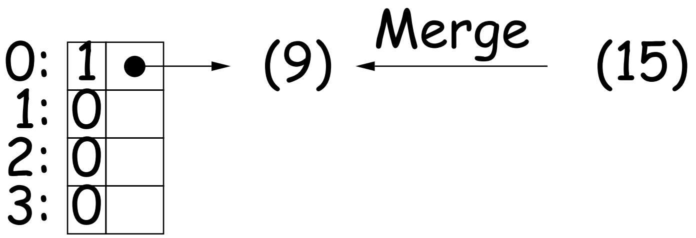


Illustration of Internal Merge Sort

For internal sorting, we can use a *binomial comb* to orchestrate an iterative merge sort.

- Start with $\lg N + 1$ buckets that can contain sublists, initially empty.
- Bucket #k is either empty or contains 2^k sorted items at any time.
- For each item in the input list, turn it into a 1-element list, and merge it into bucket 0 (or simply put it in bucket 0 if that is empty).
- You will only merge lists of length 2^k into bucket k. Whenever that gives a list of size 2^{k+1}, merge it into bucket $k+1$ and clear bucket k (and so on as needed with buckets $k+2$, etc.)
- When all inputs are processed, merge all the buckets into the final list.

L: (9, 15, 5, 3, 0, 6, 10, -1, 2, 20, 8)

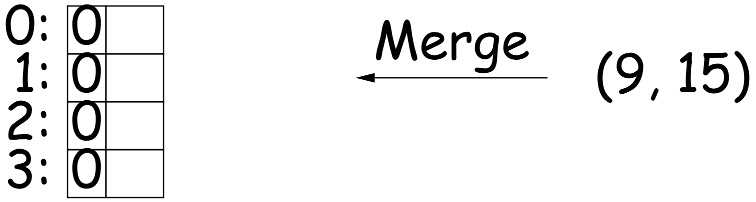
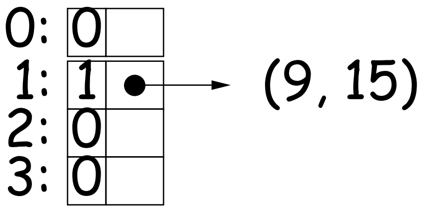


Illustration of Internal Merge Sort

For internal sorting, we can use a *binomial comb* to orchestrate an iterative merge sort.

- Start with $\lg N + 1$ buckets that can contain sublists, initially empty.
- Bucket #k is either empty or contains 2^k sorted items at any time.
- For each item in the input list, turn it into a 1-element list, and merge it into bucket 0 (or simply put it in bucket 0 if that is empty).
- You will only merge lists of length 2^k into bucket k. Whenever that gives a list of size 2^{k+1}, merge it into bucket $k+1$ and clear bucket k (and so on as needed with buckets $k+2$, etc.)
- When all inputs are processed, merge all the buckets into the final list.

L: (9, 15, 5, 3, 0, 6, 10, -1, 2, 20, 8)



0: 0
1: 1 → (9, 15)
2: 0
3: 0

Illustration of Internal Merge Sort

For internal sorting, we can use a *binomial comb* to orchestrate an iterative merge sort.

- Start with $\lg N + 1$ buckets that can contain sublists, initially empty.
- Bucket #k is either empty or contains 2^k sorted items at any time.
- For each item in the input list, turn it into a 1-element list, and merge it into bucket 0 (or simply put it in bucket 0 if that is empty).
- You will only merge lists of length 2^k into bucket k. Whenever that gives a list of size 2^{k+1}, merge it into bucket $k+1$ and clear bucket k (and so on as needed with buckets $k+2$, etc.)
- When all inputs are processed, merge all the buckets into the final list.

L: (9, 15, 5, 3, 0, 6, 10, -1, 2, 20, 8)

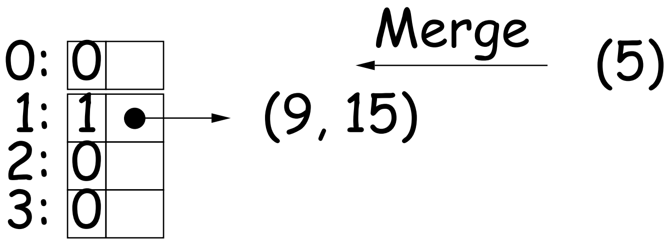


Illustration of Internal Merge Sort

For internal sorting, we can use a *binomial comb* to orchestrate an iterative merge sort.

- Start with $\lg N + 1$ buckets that can contain sublists, initially empty.
- Bucket #k is either empty or contains 2^k sorted items at any time.
- For each item in the input list, turn it into a 1-element list, and merge it into bucket 0 (or simply put it in bucket 0 if that is empty).
- You will only merge lists of length 2^k into bucket k. Whenever that gives a list of size 2^{k+1}, merge it into bucket $k+1$ and clear bucket k (and so on as needed with buckets $k+2$, etc.)
- When all inputs are processed, merge all the buckets into the final list.

L: (9, 15, 5, 3, 0, 6, 10, -1, 2, 20, 8)

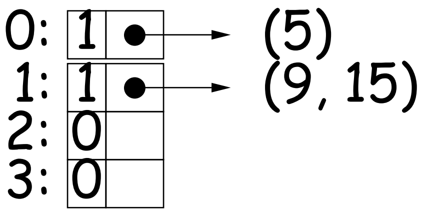

0: 1 (5)
1: 1 (9, 15)
2: 0
3: 0

Illustration of Internal Merge Sort

For internal sorting, we can use a *binomial comb* to orchestrate an iterative merge sort.

- Start with $\lg N + 1$ buckets that can contain sublists, initially empty.
- Bucket #k is either empty or contains 2^k sorted items at any time.
- For each item in the input list, turn it into a 1-element list, and merge it into bucket 0 (or simply put it in bucket 0 if that is empty).
- You will only merge lists of length 2^k into bucket k. Whenever that gives a list of size 2^{k+1}, merge it into bucket $k+1$ and clear bucket k (and so on as needed with buckets $k+2$, etc.)
- When all inputs are processed, merge all the buckets into the final list.

L: (9, 15, 5, 3, 0, 6, 10, -1, 2, 20, 8)

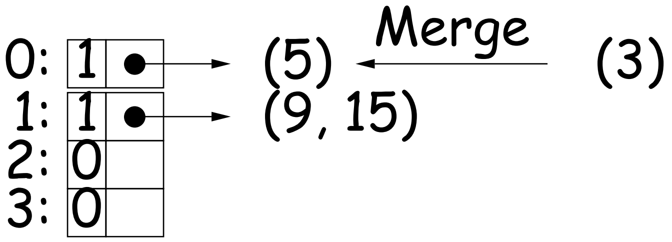


Illustration of Internal Merge Sort

For internal sorting, we can use a *binomial comb* to orchestrate an iterative merge sort.

- Start with $\lg N + 1$ buckets that can contain sublists, initially empty.
- Bucket #k is either empty or contains 2^k sorted items at any time.
- For each item in the input list, turn it into a 1-element list, and merge it into bucket 0 (or simply put it in bucket 0 if that is empty).
- You will only merge lists of length 2^k into bucket k. Whenever that gives a list of size 2^{k+1}, merge it into bucket $k+1$ and clear bucket k (and so on as needed with buckets $k+2$, etc.)
- When all inputs are processed, merge all the buckets into the final list.

L: (9, 15, 5, 3, 0, 6, 10, -1, 2, 20, 8)

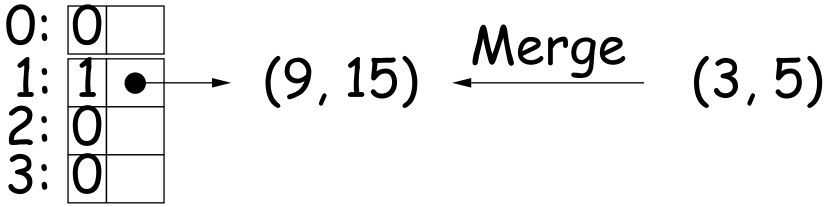
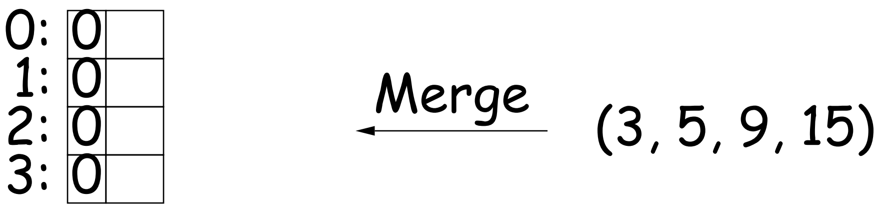


Illustration of Internal Merge Sort

For internal sorting, we can use a *binomial comb* to orchestrate an iterative merge sort.

- Start with $\lg N + 1$ buckets that can contain sublists, initially empty.
- Bucket #k is either empty or contains 2^k sorted items at any time.
- For each item in the input list, turn it into a 1-element list, and merge it into bucket 0 (or simply put it in bucket 0 if that is empty).
- You will only merge lists of length 2^k into bucket k. Whenever that gives a list of size 2^{k+1}, merge it into bucket $k+1$ and clear bucket k (and so on as needed with buckets $k+2$, etc.)
- When all inputs are processed, merge all the buckets into the final list.

L: (9, 15, 5, 3, 0, 6, 10, -1, 2, 20, 8)


0: 0
1: 0
2: 0
3: 0

Merge

(3, 5, 9, 15)

Illustration of Internal Merge Sort

For internal sorting, we can use a *binomial comb* to orchestrate an iterative merge sort.

- Start with $\lg N + 1$ buckets that can contain sublists, initially empty.
- Bucket #k is either empty or contains 2^k sorted items at any time.
- For each item in the input list, turn it into a 1-element list, and merge it into bucket 0 (or simply put it in bucket 0 if that is empty).
- You will only merge lists of length 2^k into bucket k. Whenever that gives a list of size 2^{k+1}, merge it into bucket $k+1$ and clear bucket k (and so on as needed with buckets $k+2$, etc.)
- When all inputs are processed, merge all the buckets into the final list.

L: (9, 15, 5, 3, 0, 6, 10, -1, 2, 20, 8)

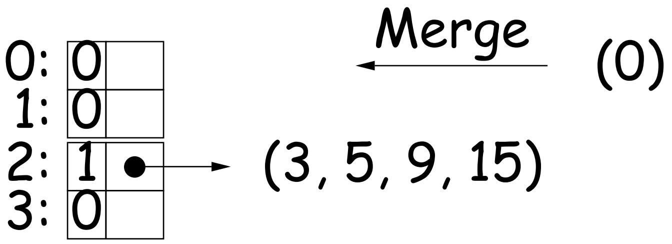
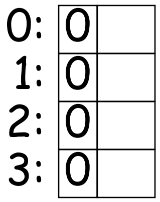
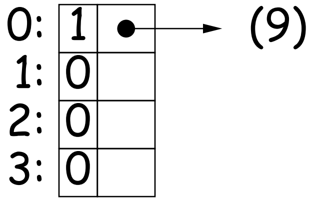


Illustration of Internal Merge Sort (II)

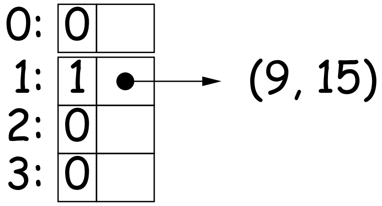
L: (9, 15, 5, 3, 0, 6, 10, -1, 2, 20, 8)



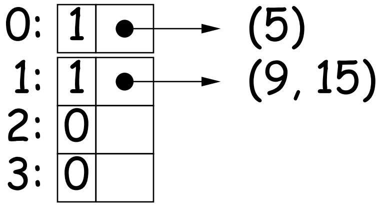
0 elements processed



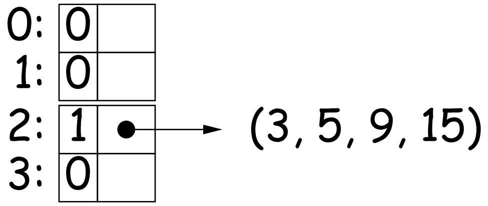
1 element processed



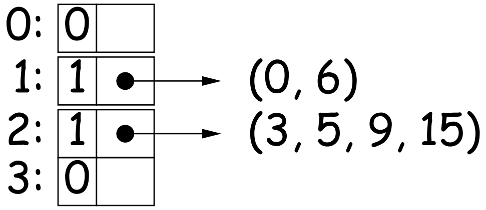
2 elements processed



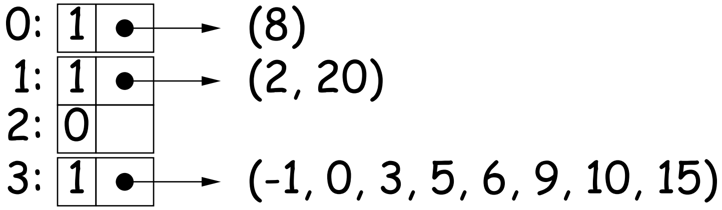
3 elements processed



4 elements processed



6 elements processed



11 elements processed

Final Step: Merge all the lists into (-1, 0, 2, 3, 5, 6, 8, 9, 10, 15, 20

Quicksort: Speed through Probability

Idea:

- *Partition* data into pieces: everything $>$ a *pivot* value at the high end of the sequence to be sorted, and everything $<$ on the low end, and everything $=$ between.
- Repeat recursively on the high and low pieces.
- For speed, stop when pieces are "small enough" and do insertion sort on the whole thing.
- Reason: insertion sort has low constant factors. By design, no item will move out of its piece [why?], so when pieces are small, #inversions is, too.
- Have to choose pivot well. E.g.: *median* of first, last and middle items of sequence.

Example of Quicksort

- In this example, we continue until pieces are size ≤ 4.
- Pivots for next step are starred. We arrange to move the pivot to dividing line each time.
- Last step is insertion sort.

16 10 13 18 -4 -7 12 -5 19 15 0 22 29 34 -1*

-4 -5 -7 ‖ -1 ‖ 18 13 12 10 19 15 0 22 29 34 16*

-4 -5 -7 ‖ -1 ‖ 15 13 12* 10 0 ‖ 16 ‖ 19* 22 29 34 18

-4 -5 -7 ‖ -1 ‖ 10 0 ‖ 12 ‖ 15 13 ‖ 16 ‖ 18 ‖ 19 ‖ 29 34 22

- Now everything is "close to" right (just 7 inversions), so just do insertion sort:

-7	-5	-4	-1	0	10	12	13	15	16	18	19	22	29	34

Performance of Quicksort

- Probabalistic time:
  - With a good choice of pivots, we divide data by two each time, giving $\Theta(N \lg N)$ and a good constant factor relative to merge or heap sort.
  - With a bad choice of pivots, most items will be on one side each time: leading to a $\Theta(N^2)$ time.
  - Time is $\Omega(N \lg N)$ even in the best case, so insertion sort is better for nearly ordered input sets.
- Interesting point: randomly shuffling the data before sorting makes $\Omega(N^2)$ time *very* unlikely!

Quick Selection

The Selection Problem: for given k, find k^{th} smallest element in data.

- Obvious method: sort, select element #k, time $\Theta(N \lg N)$.
- If $k \leq$ some constant, we can easily do in $\Theta(N)$ time:
  - Go through array, keeping the smallest k items.
- Get *probably* $\Theta(N)$ *time* for all k by adapting quicksort:
  - Partition around some pivot, p, as in quicksort, arrange for that pivot to end up at the dividing line.
  - Suppose that in the result, the pivot is at index m, all elements $\leq$ pivot have indicies $\leq m$.
  - If $m = k$, you're done: p is answer.
  - If $m > k$, recursively select the k^{th} largest from the left half of the sequence.
  - If $m < k$, recursively select the $(k - m - 1)^{\text{th}}$ largest from the right half of sequence.

Selection Example

Problem: Find just item #10 in the sorted version of array:

Initial contents:

51	60	21	-4	37	4	49	10	40*	59	0	13	2	39	11	46	31
0																

Looking for #10 to left of pivot 40:

13	31	21	-4	37	4*	11	10	39	2	0	‖	40	‖	59	51	49	46	60
0																		

Looking for #6 to right of pivot 4:

-4	0	2	‖	4	‖	37	13	11	10	39	21	31*	‖	40	‖	59	51	49	46	60
						4														

Looking for #1 to right of pivot 31:

-4	0	2	‖	4	‖	21	13	11	10	‖	31	‖	39	37	‖	40	‖	59	51	49	46	60
													9									

Just two elements; just sort and return #1:

-4	0	2	‖	4	‖	21	13	11	10	‖	31	‖	37	39	‖	40	‖	59	51	49	46	60
													9									

Result: 39

Selection Performance

- For this algorithm, if m is roughly in the middle each time, the cost is

$$\begin{aligned} C(N) &= \begin{cases} 1, & \text{if } N = 1, \\ N + C(N/2), & \text{otherwise.} \end{cases} \\ &= N + N/2 + \ldots + 1 \\ &= 2N - 1 \in \Theta(N) \end{aligned}$$

- But in worst case, we get $\Theta(N^2)$, as for quicksort.
- By another, non-obvious algorithm, we can get $\Theta(N)$ worst-case time for all k (take CS170).