Outline of these notes

- Heapsort, priority queues, binary heaps
- Summary of comparison-based sorting methods; stability
- Lower bound on comparison-based sorting
- Address-calculation sorting algorithms:
  - Counting sort
  - Bucket sort
  - Radix sort
- External sorting

Heapsort

Consider the following version of Selection Sort (sometimes called Max sort)

```
def maxSort(A,n):
    for k = n-1 downto 1
        find j such that A[j] == max(A[0],A[1],..., A[k])
        A[j] ↔ A[k]
```

A straightforward implementation requires $O(n^2)$ time, because of the time spent repeatedly finding the maximum of the first k items.

But we can speed this up by using a binary heap.

Priority Queues and Heaps

- Priority Queue
  - Abstract data type
  - Collection of items.
  - Each item has an associated key, which corresponds to a priority.
  - Supports the following operations
    - Insert an item with a given key
    - Delete an item
    - Select the item with the most urgent priority in the priority queue.
  - Most urgent priority may correspond to the lowest key value or to the highest key value, depending on the application.

Binary Heaps

- Specific implementation of priority queue
- Items are stored in an array.
- The array represents a binary tree in level order (breadth-first order).
- Can be max-heap or min-heap
  - In a max-heap, large key values represent more urgent priorities
  - In a min-heap, small key values represent more urgent priorities
- In this introduction, we will be using a max-heap.
- Heap invariant for max-heaps: For any item v other than the root,

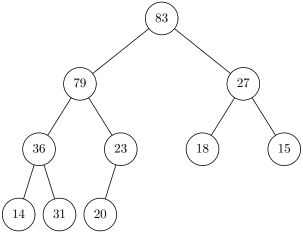
$$\texttt{key}\,(\texttt{parent}(v)) \geq \texttt{key}(v)$$

- In a min-heap, the direction of the inequality is reversed.
- In our examples, items are integers, key is the integer value

Viewing the array as a binary tree

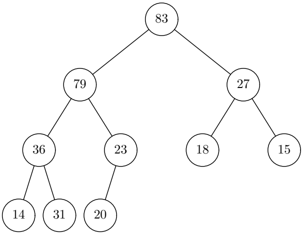
- Root is $H[0]$
- Left child of $H[i]$ is $H[2i+1]$ (provided $2i+1 < n$, where $n = H.\text{size}$)
- Right child of $H[i]$ is $H[2i+2]$ (provided $2i+2 < n$)
- Parent of $H[i]$ is $H[\lfloor (i-1)/2 \rfloor]$ (provided $i > 0$)

83	79	27	36	23	18	15	14	31	20
0	1	2	3	4	5	6	7	8	9



Heap operations in a max-heap:

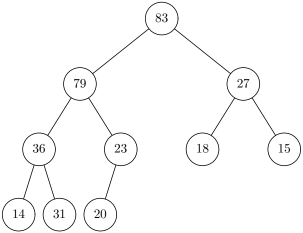
- `FindMax(H)`: Find maximum item in the heap
- `ExtractMax(H)`: Find maximum item and delete it from the heap
- `Insert(H,x)`: Insert the new item x in the heap
- `Delete(H,i)`: Delete the item at location i from the heap



FindMax: Find maximum item in the heap

Findmax is easy: just report the value at the root.

```
def FindMax(H):
    return H[0]
```



Helper functions

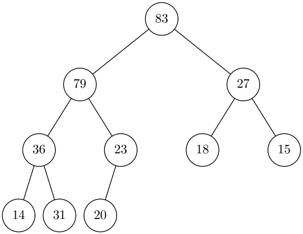
- Except for FindMax, the binary heap operations require some data movement.
- The heap invariant must be preserved after each operation.
- We define two helper functions.
  - `SiftUp(H,i)`: Move the item at location *i* up to its correct position by repeatedly swapping the item with its parent, as necessary.
  - `SiftDown(H,i)`: Move the item at location *i* down to its correct position by repeatedly swapping the item with the child having the larger key, as necessary.

[GT] calls these "up-heap bubbling" and "down-heap bubbling"

SiftUp: Sift an item up to its correct position

```
def SiftUp(H,i):
    parent = (i-1)/2;
    if (i > 0) and (H[parent].key < H[i].key):
        H[i] ↔ H[parent]
        SiftUp(H,parent)
```

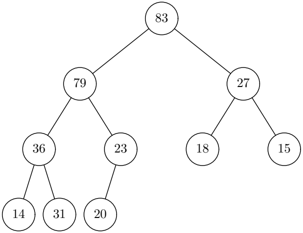
Analysis: at most 1 comparison at each level, so total time is $O(\log n)$



`SiftDown`: Sift an item down to its correct position

```
def SiftDown(H,i):
    n = H.size // number of item in heap
    left = 2i+1; right = 2i+2
    if (right < n) and (H[right].key > H[left].key)
        largerChild = right
    else largerChild = left
    if (largerchild < n) and (H[i].key < H[largerChild].key)
        H[i] ↔ H[largerchild]
        SiftDown(H,largerchild)
```

Analysis: at most 2 comparisons at each level, so total time is $O(\log n)$

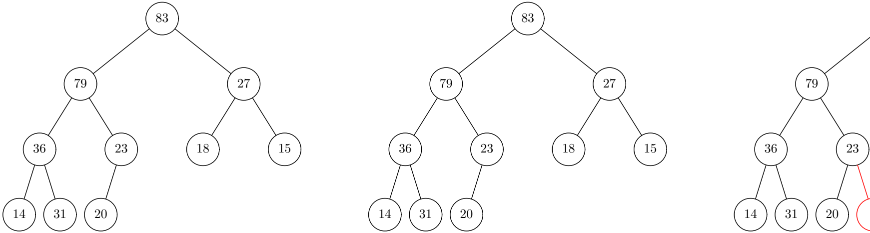


Insert: Insert the new item x

```
def Insert(H,x):
    H.size = H.size+1 // increment number of items
    k = H.size-1 //index of last position
    H[k] = x //insert x in last position
    SiftUp(H,k)
```

Analysis: Siftup time dominates, so total time is $O(\log n)$

`Insert(H,81)`

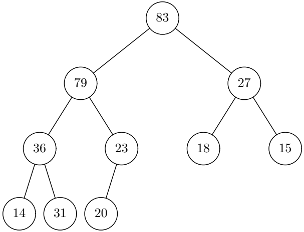


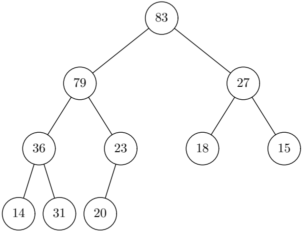
Delete: Delete the item at location *i*

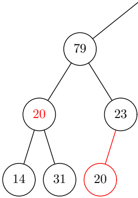
```
def Delete(H,i):
    k = H.size-1 //index of last position
    H[i] = H[k] // overwrite item being deleted with
                   element in last position
    H.size = H.size-1 // decrement number of item
    SiftUp(H,i) // either SiftUp or SiftDown will do nothing
    SiftDown(H,i)
```

Analysis: Siftup/siftdown time dominates, so total time is $O(\log n)$

Delete(H,3)



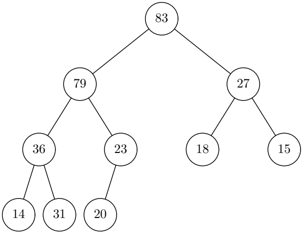




ExtractMax: Find maximum item and delete it

```
def ExtractMax(H):
    x = H[0]
    Delete(H,0)
    return x
```

Analysis: Delete time dominates, so total time is $O(\log n)$



Constructing a heap

How do we efficiently construct a brand-new heap storing n given items?

If we insert the items one at a time, time spent on kth insertion is $O(\log k)$.

So total time is

$$O\left(\sum_{k=1}^{n-1} \log k\right) = O\left(n \log n\right)$$

There is a better way that only requires $O(n)$ time...

Constructing a heap in $O(n)$ time

1. Put the data in H, in arbitrary order. (So H stores the correct data, but does not satisfy the heap invariant.)
2. Run the following `Heapify` function.

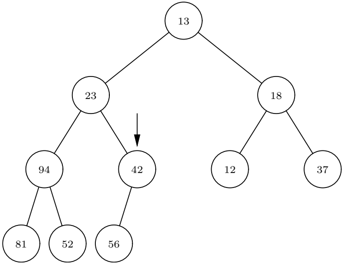
```
def heapify(H,n)
    for i = n-1 down to 0:
        SiftDown(H,i)
```

The code given above can be improved: We can start at $\mathrm{i} = \lfloor(\mathrm{n}\text{-}2)/2\rfloor$ (or equivalently, $\mathrm{i} = \lfloor \mathrm{n}/2 \rfloor - 1$), rather than $i = n - 1$.

Heapify example

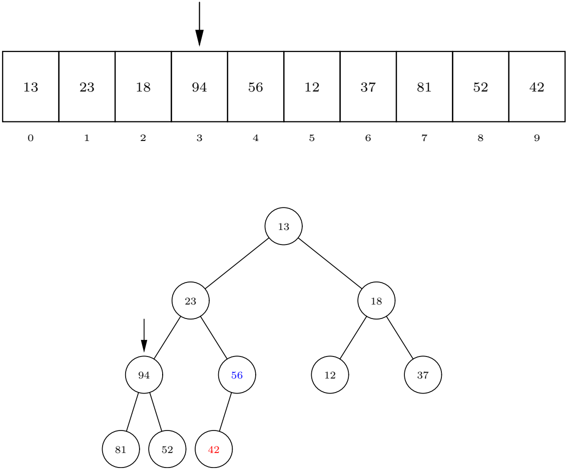
13 23 18 94 42 12 37 81 52 56

13	23	18	94	42	12	37	81	52	56
0	1	2	3	4	5	6	6	8	9



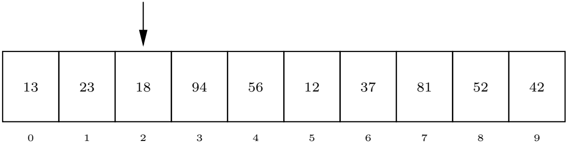
Heapify example, continued

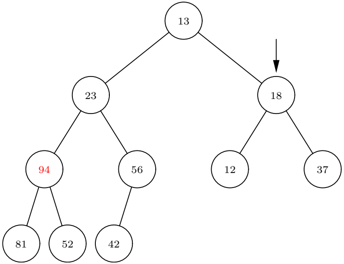
`13 23 18 94 42 12 37 81 52 56`



Heapify example, continued

13 23 18 94 42 12 37 81 52 56

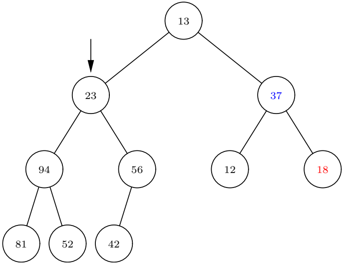




Heapify example, continued

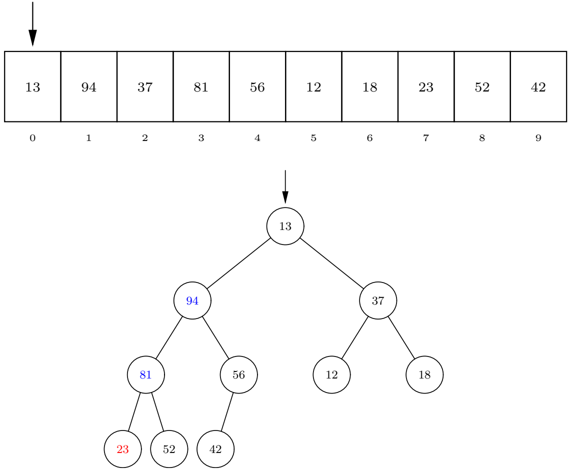
13 23 18 94 42 12 37 81 52 56

13	23	37	94	56	12	18	81	52	42
0	1	2	3	4	5	6	7	8	9



Heapify example, continued

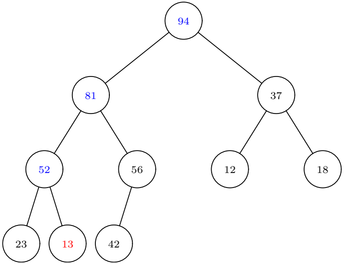
13 23 18 94 42 12 37 81 52 56



Heapify example, continued

13 23 18 94 42 12 37 81 52 56

94	81	37	52	56	12	18	23	13	42
0	1	2	3	4	5	6	7	9	9



Analysis of heap construction algorithm using Heapify

```
Algorithm heapify(H,n);
   for i = n-1 down to 0:
      SiftDown(H,i)
```

- Correctness: After SiftDown(H,i) is executed, subtree rooted at node i satisfies heap invariant. (Can show by induction).
- Running time: `Heapify` runs in $O(n)$ time. We will prove this on the next slide.

Proof that Heapify runs in $O(n)$ time

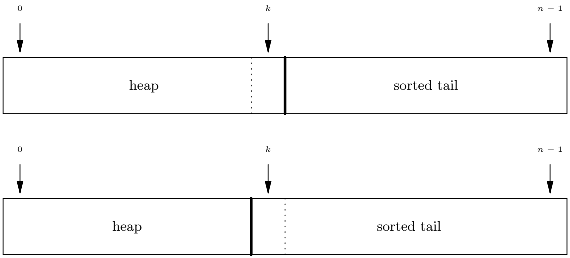
- Suppose the tree has n nodes and d levels (so $2^d \le n < 2^{d+1}$).
- If node i is at level j, `SiftDown(H,i)` needs $\le 2(d-j)$ comparisons.
- There are at most 2^j nodes at level j.
- So total number of comparisons is no more than:

$$
\begin{aligned}
\sum_{j=0}^{d} 2(d-j)2^j &= 2d\sum_{j=0}^{d} 2^j - 2\sum_{j=0}^{d} j2^j \\
&= 2d(2^{d+1}-1) - 2\left[(d-1)2^{d+1}+2\right] \\
&= 2d2^{d+1} - 2d - 2d2^{d+1} + 2\cdot 2^{d+1} - 4 \\
&= 4\cdot 2^d - 2d - 4 \\
&< 4\cdot 2^d \le 4n = O(n)
\end{aligned}
$$

So heap can be constructed using $O(n)$ comparisons.

Heapsort: version based on Max Sort

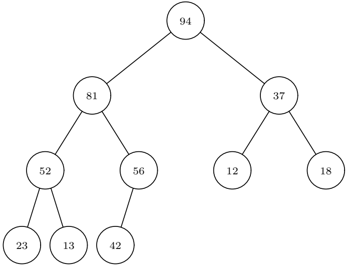
```
def heapsort(A,n):
    heapify(A,n) // form max heap using array A
    for k = n-1 down to 1:
        A[k] = ExtractMax(A)
```



Heapsort example

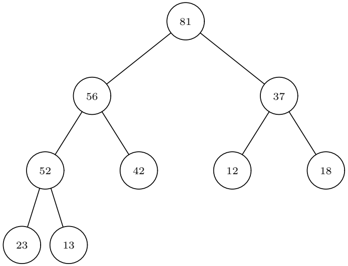
Sort: 13 23 18 94 42 12 37 81 52 56

Heapify:



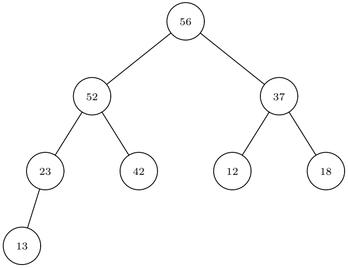
94	81	37	52	56	12	18	23	13	42
0	1	2	3	4	5	6	7	8	9

Heapsort example, continued



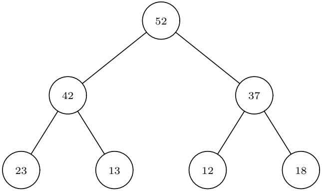
0	1	2	3	4	5	6	7	8	9
81	56	37	52	42	12	18	23	13	94

Heapsort example, continued



56	52	37	23	42	12	18	13	81	94
0	1	2	3	4	5	6	7	8	9

Heapsort example, continued



52	42	37	23	13	12	18	56	81	94
0	1	2	3	4	5	6	7	8	9

Exercise: Finish this example.

Analysis of Heapsort

- Storage: $O(1)$ extra space (in place)
- Time:
  - `Heapify`: $O(n)$
  - All calls to `ExtractMax`:

$$\sum_{k=1}^{n-1} O\left(\log(k+1)\right) = O(n \log n)$$

  - Hence total time is $O(n \log n)$.

Heapsort: Alternate version

- Uses a min-heap (instead of a max-heap)
- Output items in sorted order rather than storing them back in the array

```
def heapsort(A,n):
    heapify(A,n) // Form min heap
    for k = 1 to n:
        x = ExtractMin(A)
        output(x)
```

- Same analysis as previous version: $O(n \log n)$ time, $O(1)$ extra space
- If we stop after computing the first k entries, total work is

$$O(n + k \log n)$$

Comparison-based sorts: Summary/Comparison

Sort	Worst-case Time	Storage Requirement	Remarks
Insertion Sort	$O(n^2)$	In-place	Good if input is almost sorted.
QuickSort	$O(n^2)$	$O(\log n)$ extra for stack	$O(n \log n)$ expected time.
Mergesort	$O(n \log n)$	$O(n)$ extra for merge	
Heapsort	$O(n \log n)$	In-place	Can output k smallest in sorted order in $O(n + k \log n)$ time.

Stable sorting

A sort is stable if keys having the same value appear in the same order in the output array as they do in the input array.

$$[3\ 2\ 1\ 2] \rightarrow [1\ 2\ 2\ 3]$$: Stable

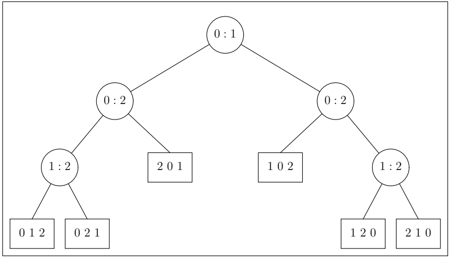
$$[3\ 2\ 1\ 2] \rightarrow [1\ 2\ 2\ 3]$$: Not Stable

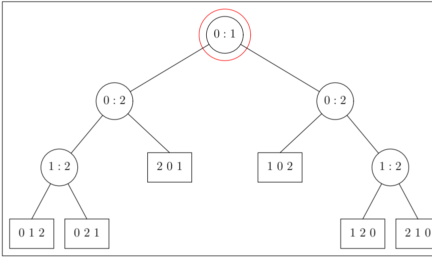
Sort	Stable (without special care)?
Insertion Sort	Yes
Quick-Sort	No
Merge-Sort	Yes (as described here)
Heap-Sort	No

Lower bound on comparison-based sorting

- Based on Decision Tree model.
- Any algorithm that sorts a list or array of size n using comparisons can be modeled as a decision tree:
  - Each internal node is labeled $i : j$, representing a comparison between $L[i]$ and $L[j]$.
  - The left (respectively, right) of a node labeled $i : j$ describes for what happens if $L[i] < L[j]$ (respectively, $L[i] > L[j]$).
  - Each leaf node is a permutation of $0, \ldots n-1$.

Example: Decision tree for sorting 3 items





Lower bound on comparison-based sorting (continued)

1. Any comparison-based algorithm for sorting a list of size n can be modeled by a decision tree with at least $n!$ leaf nodes.
2. Since the decision tree is a binary tree with $n!$ leaves, the depth is at least $\lceil \lg n! \rceil$.
3. The worst-case number of comparisons for the algorithm is the depth of the decision tree.
4. $\lg n! = \Omega(n \log n)$ (proof on next slide)

Fact #2 and Fact #3 imply an exact bound:

Any comparison-based algorithm for sorting a list of size n must perform at least $\lceil \lg n! \rceil$ *comparisons in the worst case.*

The previous statement and Fact #4 imply an asymptotic bound:

Any comparison-based algorithm for sorting a list of size n must perform at least $\Omega(n \log n)$ *comparisons in the worst case.*

Lower bound on comparison-based sorting (continued)

Proof that $\lg n! = \Omega(n \log n)$:

$$n! = n \cdot (n-1) \cdot (n-3) \cdots 2 \cdot 1$$

The first $\lceil n/2 \rceil$ terms in the product are all $\geq \left\lceil \frac{n}{2} \right\rceil$.

This implies:

$$n! \geq \left\lceil \frac{n}{2} \right\rceil^{\left\lceil \frac{n}{2} \right\rceil} \geq \left(\frac{n}{2} \right)^{\frac{n}{2}}$$

Take $\log_2$ of both sides:

$$\lg n! \geq \left(\frac{n}{2} \right) \lg \left(\frac{n}{2} \right) = \left(\frac{n}{2} \right) (\lg n - 1) = \Omega(n \lg n)$$

Asymptotic optimality of MergeSort and HeapSort

We have just shown:

Any comparison-based algorithm for sorting a list of size n must perform at least $\Omega(n \log n)$ comparisons in the worst case.

Earlier we showed:

The worst-case running time of MergeSort and HeapSort on an input of size n is $O(n \log n)$.

Conclusions:

1. MergeSort and HeapSort are asymptotically optimal.
2. The lower bound is asymptotically tight (i.e., cannot be improved asymptotically)

Comparisons by Mergesort vs. the exact lower bound

- Sorting lower bound: $\lceil \lg n! \rceil$.
- Mergesort: Solution of

$$W(n) = \begin{cases} n - 1 + W\left(\left\lceil \frac{n}{2} \right\rceil\right) + W\left(\left\lfloor \frac{n}{2} \right\rfloor\right), & n > 1 \\ 0, & n = 1 \end{cases}$$

- Comparison:

n	Lower Bound	Merge Sort
1	0	0
2	1	1
3	3	3
4	5	5
5	7	8 ←
6	10	11
7	13	14
8	16	17
9	19	21

n	Lower Bound	Merge Sort
10	22	25
11	26	29
12	29	33
13	33	37
14	37	41
15	41	45
16	45	49
17	49	54

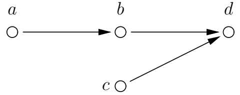
Optimally sorting 5 items

- According to table on previous slide:
  - Mergesort requires 8 comparisons to sort 5 items
  - The lower bound says we need at least 7 comparisons to sort 5 items
- Question: Is it possible to sort 5 items using only 7 comparisons?
- Answer: Yes

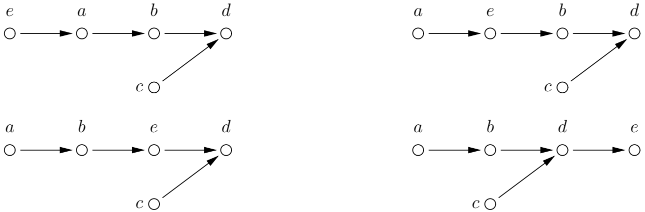
Call the 5 items `a`, `b`, `c`, `d`, `e`...

Sorting 5 items with 7 comparisons

`if` $a > b$`:` $a \leftrightarrow b$
`if` $c > d$`:` $c \leftrightarrow d$
`if` $b > d$`:` $b \leftrightarrow d$, $a \leftrightarrow c$ (3 comparisons)



Find position for e in $[a,b,d]$ (2 more comparisons)



Find position for c (4 cases, 2 more comparisons in each case)

Sorting 5 items with 7 comparisons

As we just showed, we can sort 5 items using exactly 7 comparisons.

So we can conclude:

1. The exact lower bound of 7 comparisons for sorting 5 items is a tight bound.
2. Merge sort is not optimal if we look at exact number of comparisons (because MergeSort requires 8 comparisons to sort 5 items).

Address-Calculation Sorting Algorithms

- Based on data values.
- Performance is not limited by $\Omega(n \log n)$ bound, but does depend on data values.
- Comparisons are not necessarily a reasonable measure of work performed.
- We will discuss 3 algorithms:
  1. Counting sort
  2. Bucket sort
  3. Radix sort

Counting sort

- Let A be the input array, B the output array. Assume there are n items.
- Warning: this algorithm description assumes that the arrays are indexed starting from 1, not from 0.
  - To implement the algorithm in a modern programming language directly from the algorithm description:
    - Allocate each array to be one entry larger than it actually is
    - Ignore location 0.
- Main idea of CountingSort: Suppose A contains exactly j elements $\leq x$
  - If x only appears once in A, then x should go in $B[j]$.
  - If x appears more than once in A and we want a stable sort:
    - Last occurrence of x in A should go in $B[j]$
    - Next-to-last occurrence of x should go in $B[j-1]$
    - etc.

Counting sort

- Assume:
  - We are sorting an array `A[1..n]` of integers
  - Each integer is in the range `1..k`
  - Output array is `B[1..n]`
- Use an auxiliary array `locator[1..k]`
- `locator[x]` contains the index of the position in the output array `B` where a key of `x` should be stored.
  - We make several passes over the data to set the values in the locator array before we do the actual sort.
  - At the start of the final (sorting) pass, `locator[x]` contains the number of elements $\leq$ `x`
- On the final pass:
  - Process the input array `A` from right to left (!). This makes the counting sort a stable sorting algorithm.
  - When a value of `x` is encountered in the input array `A`:
    - Copy the value into location `locator[x]` in the output array. That is, store it in location `B[locator[x]]`
    - Decrement `locator[x]`

Code for Counting sort

```
def CountingSort(A, B, n , k)
    //Initialize:  set each locator[x] to
          the number of entries ≤ x
    for x = 1 to k do locator[x] = 0
    for i = 1 to n do locator[A[i]] = locator[A[i]] + 1
    for x = 2 to k do
        locator[x] = locator[x] + locator[x-1]
    //Fill output array, updating locator values
    for i = n down to 1 do
        B[locator[A[i]]] = A[i]
        locator[A[i]] = locator[A[i]] - 1
```

Analysis: $O(n+k)$ running time.

Counting Sort Example

A:

1	2	3	4	5	6	7	8	9	10
1	3	5	7	5	7	3	8	7	4

1	2
1	3

`locator`:

1	2	3	4	5	6	7	8
1	1	3	4	6	6	9	10

1	2	3	4
1	1	3	4

B:

1	2	3	4	5	6	7	8	9	10

1	2

Bucket Sort

- Divide space of possible keys into contiguous subranges, or buckets.
- Three phases:
  1. Distribute keys into buckets
  2. Sort keys in each bucket
  3. Combine buckets.
- Simplest approach is to divide the space of possible keys into equal sized buckets.
- Typically use insertion sort in phase 2.

Bucket Sort Example

Sort the following keys in the range 0-999, using 10 equal-size buckets:

661 74 835 140 198 923 113 642 467 449

1. Distribute	
0:	74
1:	140 198 113
2:	
3:	
4:	467 449
5:	
6:	661 642
7:	
8:	835
9:	923

2. Sort	
0:	74
1:	113 140 198
2:	
3:	
4:	449 467
5:	
6:	642 661
7:	
8:	835
9:	923

3. Combine
74
113
140
198
449
467
642
661
835
923

Analysis of Bucket Sort

n = number of items to sort
b = number of buckets
s_i = number of items in bucket i ($i = 0, \ldots, b-1$)

Phase	Running time
1. Distribution	$O(n)$
2. Sorting each bucket	$O(b + \sum_{i=0}^{b-1} s_i^2)$
3. Combining buckets	$O(b)$

Total running time is:

$$O\left(n + b + \sum_{i=0}^{b-1} s_i^2\right)$$

- Worst case: $O(n^2)$.
- Best case: $O(n)$.
- Average case: $O(n)$ if certain assumptions are satisfied (next slide)
- Storage: is $O(n + b)$.

Average running time of Bucket Sort

The following result is proved in [CLRS]:

Assume:

1. *The number of buckets is equal to the number of keys (i.e., if $b = n$)*
2. *The keys are distributed independently and uniformly over the buckets*

Then the expected total cost of the intra-bucket sorts is $O(n)$.

Radix Sort

- Useful for sorting multi-field keys in lexicographic order
- Lexicographic order means sorted on the most important field, with ties broken on the next most important field, and so on. It is also called dictionary order
- Examples:
  - Words in dictionaries:
    - clown comes before dog
    - cat comes before clown
    - car comes before cat
  - Dates: `(year, month, day)`
  - Multi-digit numbers: (3-digit numbers in this example)
    - 293 represented as (2,9,3)
    - 71 represented as (0,7,1)

Radix Sort:

Radix sort:

- Sorts on each field in the key, one at a time
- Sorts on least-significant field first
- Uses a stable sort
    - Recall: A sorting algorithm is stable if whenever two keys are equal, the algorithm preserves their order (i.e., does not reverse them.)

```
def radix_sort(A,n);
    for field ranging from rightmost (least significant)
          to leftmost (most significant):
       sort A on field using a stable sort
```

Radix Sort Example:

Sort the following numbers using radix sort (each digit is a field)

661 74 835 140 198 923 113 642 467 449

661	⇒	140	⇒	113	⇒	074
074		661		923		113
835		642		835		140
140		923		140		198
198		113		642		449
923		074		449		467
113		835		661		642
642		467		467		661
467		198		074		835
449		449		198		923

Note the importance of stability.

We break the ties first, and stability makes sure the ties remain broken correctly.

Analysis of Radix Sort

Assume

- n is the number of items
- b is the size of each range
  - Example:
    - Each field of each item is a numbers in the range `0..b-1`.
    - This is true if the numbers we are sorting are integers represented in base b.
- d is the number of fields we are sorting
  - For example, if each item is a base b number with d digits. (i.e., between 0 and $b^d - 1$, inclusive).
- Each field is sorted using Bucket Sort or Counting Sort

Then the running time of radix sort is $O(d(n + b))$.

External Sorting

- Problem: Sorting a large file, bigger than available memory
- Assume
    1. n records (items) in file
    2. m records can fit in memory at once ($m \ll n$)
    3. f input files can be open at once.

Polyphase Merge

- Phase 1:
  - Read in groups of m records
  - Sort each group
  - Write each run (sorted group) to a separate output file
- Subsequent phases: Repeatedly
  - Choose f files (or all that are left if there are fewer than f)
  - Merge the contents of the f input files into a new output file
  - Delete the f input files
- For efficiency, choose the smallest length files or use FIFO ordering

Polyphase Merge example ($n = 54$, $m = 4$, $f = 3$)

```
145 507 354 590 875  29   9 481  47 212 208 929 902 124 250  11
386 281 680 109 100 542  64 508 654 793 538 322 299 686 104 989
465 777 991 931 677 176 230 214 369 106 218 724 779 565 559 873
696 726 326 415 761 915
```

```
Phase 1:  145 507 354 590 ⇒  145 354 507 590 (Run 1)
          875  29   9 481 ⇒    9  29 481 875 (Run 2)
           47 212 208 929 ⇒   47 208 212 929 (Run 3)
          902 124 250  11 ⇒   11 124 250 902 (Run 4)
          386 281 680 109 ⇒  109 281 386 680 (Run 5)
          100 542  64 508 ⇒   64 100 508 542 (Run 6)
          654 793 538 322 ⇒  322 538 654 793 (Run 7)
          299 686 104 989 ⇒  104 299 686 989 (Run 8)
          465 777 991 931 ⇒  465 777 931 991 (Run 9)
          677 176 230 214 ⇒  176 214 230 677 (Run 10)
          369 106 218 724 ⇒  106 218 369 724 (Run 11)
          779 565 559 873 ⇒  559 565 779 873 (Run 12)
          696 726 326 415 ⇒  326 415 696 726 (Run 13)
          761 915         ⇒  761 915         (Run 14)
```

Polyphase Merge example, continued (subsequent phases)

(Run 1 + Run 2 + Run 3) ⇒ Run 15:

9 29 47 145 208 212 354 481 507 590 875 929

(Run 4 + Run 5 + Run 6) ⇒ Run 16:

11 64 100 109 124 250 281 386 508 542 680 902

(Run 7 + Run 8 + Run 9) ⇒ Run 17:

104 299 322 465 538 654 686 777 793 931 989 991

(Run 10 + Run 11 + Run 12) ⇒ Run 18:

106 176 214 218 230 369 559 565 677 724 779 873

(Run 13 + Run 14 + Run 15) ⇒ Run 19:

9 29 47 145 208 212 326 354 415 481 507 590 696 726 761 875
915 929

(Run 16 + Run 17 + Run 18) ⇒ Run 20:

11 64 100 104 106 109 124 176 214 218 230 250 281 299 322 369
386 465 508 538 542 559 565 654 677 680 686 724 777 779 793 873
902 931 989 991

(Run 19 + Run 20) ⇒ Run 21:

9 11 29 47 64 100 104 106 109 124 145 176 208 212 214 218
230 250 281 299 322 326 354 369 386 415 465 481 507 508 538 542
559 565 590 654 677 680 686 696 724 726 761 777 779 793 873 875
902 915 929 931 989 991

Replacement Selection

The initial runs can be made longer by using an improvement called Replacement Selection.

- When a key is written, the next key is read.
  - If the new key is $\geq$ the last key written, it is made part of the current run.
  - If the new key is $<$ the last key written, it is saved for the next run.

Replacement Selection Example

145 507 354 590 875 29 9 481 47 212 208 929 902 124 250 11 386 281 680
109 100 542 64 508 654 793 538 322 299 686 104 989 465 777 991 931 677
176 230 214 369 106 218 724 779 565 559 873 696 726 326 415 761 915

Memory				Run	Run Contents
145_1	507_1	354_1	590_1	1	
875_1	507_1	354_1	590_1	1	145
875_1	507_1	29_2	590_1	1	145 354
875_1	9_2	29_2	590_1	1	145 354 507
875_1	9_2	29_2	481_2	1	145 354 507 590
47_2	9_2	29_2	481_2	1	145 354 507 590 875

Replacement Selection Example, continued

145 507 354 590 875 29 9 481 47 | 212 208 929 902 124 250 11 386 281 680 109 100 542 64 508 654 793 538 322 299 686 104 989 465 777 991 931 677 176 230 214 369 106 218 724 779 565 559 873 696 726 326 415 761 915

Memory				Run	Run Contents
47_2	9_2	29_2	481_2	2	
47_2	212_2	29_2	481_2	2	9
47_2	212_2	208_2	481_2	2	9 29
929_2	212_2	208_2	481_2	2	9 29 47
929_2	212_2	902_2	481_2	2	9 29 47 208
929_2	124_3	902_2	481_2	2	9 29 47 208 212
929_2	124_3	902_2	250_3	2	9 29 47 208 212 481
929_2	124_3	11_3	250_3	2	9 29 47 208 212 481 902
386_3	124_3	11_3	250_3	2	9 29 47 208 212 481 902 929

Replacement Selection Example, continued

145 507 354 590 875 29 9 481 47 212 208 929 902 124 250 11 386 | 281 680 109 100 542 64 508 654 793 538 322 299 686 104 989 465 777 991 931 677 176 230 214 369 106 218 724 779 565 559 873 696 726 326 415 761 915

Memory				Run	Run Contents
386_3	124_3	11_3	250_3	3	
386_3	124_3	281_3	250_3	3	11
386_3	680_3	281_3	250_3	3	11 124
386_3	680_3	281_3	109_4	3	11 124 250
386_3	680_3	100_4	109_4	3	11 124 250 281
542_3	680_3	100_4	109_4	3	11 124 250 281 386
64_4	508_4	100_4	109_4	3	11 124 250 281 386 542 680

Result of Using Replacement Selection in our Example

Run 1:
145 354 507 590 875

Run 2:
9 29 47 208 212 481 902 929

Run 3:
11 124 250 281 386 542 680

Run 4:
64 100 109 508 538 654 686 793 989

Run 5:
104 299 322 465 677 777 931 991

Run 6:
176 214 218 230 369 565 724 779 873

Run 7:
106 326 415 559 696 726 761 915

7 initial runs (vs. 14 without replacement selection)

Result of Using Replacement Selection in our Example (continued)

(Run 1 + Run 2 + Run 3) ⇒ Run 8:

9	11	29	47	124	145	208	212	250	281	354	386
481	507	542	590	680	875	902	929				

(Run 4 + Run 5 + Run 6) ⇒ Run 9:

64	100	104	109	176	214	218	230	299	322	369	465
508	538	565	654	779	793	873	931				
989	991										

(Run 7 + Run 8 + Run 9) ⇒ Run 10:

9	11	29	47	64	100	104	106	109	124	145	176
208	212	214	218	230	250	281	299	322	326	354	369
386	415	465	481	507	508	538	542	559	565	590	654
677	680	686	696	724	726	761	777	779	793	873	875
902	915	929	931	989	991						

10 total runs (vs. 21 without replacement selection)

Polyphase Merge, Replacement Selection Final notes

- On the average, replacement selection doubles the sizes of the runs, assuming uniform distribution of the sort keys.
- Use a min-heap while building initial runs
- Use a min-heap during the merging phase
- Encyclopedic reference: Donald Knuth, *Sorting and Searching*, The Art of Computer Programming, Vol. 3