Heaps, Heapsort, Priority Queues

Heap:

A data structure and associated algorithms, NOT GARBAGE COLLECTION

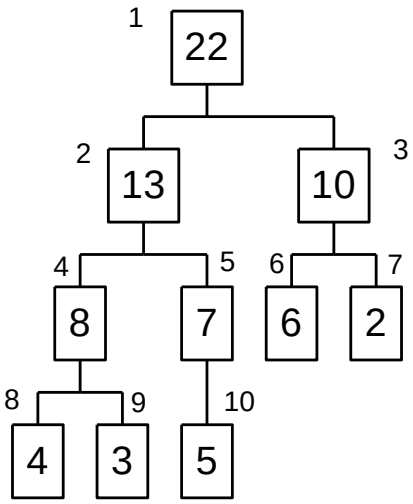
A heap data structure is an array of objects than can be viewed as a complete binary tree such that:

Each tree node corresponds to elements of the array
The tree is complete except possibly the lowest level, filled from left to right

The heap property for all nodes I in the tree must be maintained except for the root:

Parent node(I) $\geq$ I

Example: Given array [22 13 10 87 6 2 4 3 5]



Note that the elements are not sorted, only max element at root of tree.

The **height** of a node in the tree is the number of edges on the longest simple downward path from the node to a leaf; e.g. height of node 6 is 0, height of node 4 is 1, height of node 1 is 3.

The height of the tree is the height from the root. As in any complete binary tree of size n, this is lg n.

Caveats: 2^h nodes at level h. $2^{h+1}-1$ total nodes in a complete binary tree.

A heap represented as an array A represented has two attributes:

1. Length(A) – Size of the array
2. HeapSize(A) - Size of the heap

The property Length(A) ≥ HeapSize(A) must be maintained. (why ?)
The heap property is stated as A[parent(I)] ≥ A[I]

The root of the tree is A[1].
Formula to compute parents, children in an array:

Parent(I) = A[$\lfloor I / 2 \rfloor$]
Left Child(I) = A[2I]
Right Child(I) = A[2I+1]

(Show how to represent the above tree as an array in the example)

Where might we want to use heaps? Consider the Priority Queue problem: Given a sequence of objects with varying degrees of priority, and we want to deal with the highest-priority item first.

Managing air traffic control - want to do most important tasks first.
Jobs placed in queue with priority, controllers take off queue from top
Scheduling jobs on a processor - critical applications need high priority
Event-driven simulator with time of occurrence as key. Use min-heap, which keeps smallest element on top, get next occurring event.

To support these operations we need to extract the maximum element from the heap:

```
HEAP-EXTRACT-MAX(A)
    remove A[1]
    A[1] ← A[n]          ; n is HeapSize(A), the length of the heap, not array
    n ← n-1              ; decrease size of heap
    Heapify(A,1,n)       ; Remake heap to conform to heap properties
```

Runtime: Θ(1) + Heapify time

Differences from book :

no error handling
n instead of HeapSize(A)
slightly higher abstraction
Passing "n" to Heapify routine

Note: Successive removals will result in items in reverse sorted order!

We will look at:

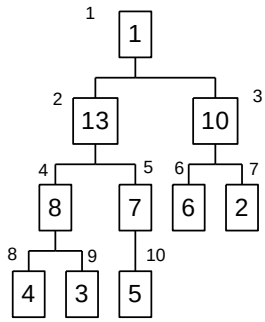
Heapify : Maintain the heap property
Build Heap : How to initially build a heap
Heapsort : Sorting using a heap

Heapify: Maintain heap property by "floating" a value down the heap that starts at I until it is in the right position.

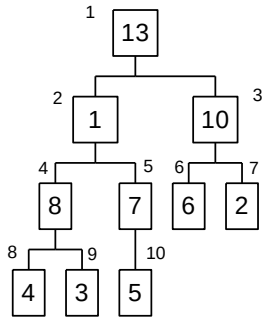
```
Heapify(A,I,n)          ; Array A, heapify node I, heapsize is n
        ; Note that the left and right subtrees of I are also heaps
        ; Make I's subtree be a heap.
        If 2I ≤ n and A[2I]>A[I]
                ; see which is largest of current node and its children
                then largest ← 2I
                else largest ← I
        If 2I+1 ≤ n and A[2I+1]>A[largest]
                then largest ← 2I+1
        If largest ≠ I
                then swap A[I] ↔ A[largest]
                     Heapify(A,largest,n)
```

Differences from book : 2I and 2I+1 instead of left and right, n instead of heapsize

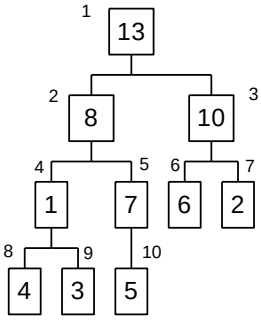
Example: Heapify(A,1,10). A=[1 13 10 8 7 6 2 4 3 5]



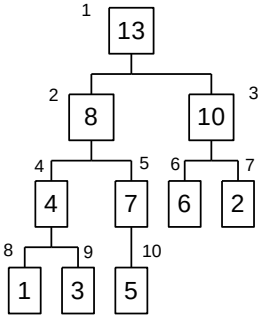
Find largest of children and swap. All subtrees are valid heaps so we know the children are the maximums.



Next is Heapify(A,2,10). A=[13 1 10 8 7 6 2 4 3 5]



Next is Heapify(A,4,10). A=[13 8 10 1 7 6 2 4 3 5]



Next is Heapify(A,8,10). A=[13 8 10 4 7 6 2 1 3 5]
On this iteration we have reached a leaf and are finished. (Consider if started at node 3, n=7)

Runtime: We make one pass down the height of the tree which is logarithmic to the number of nodes in height. So the easy analysis is this will take Θ(lg n) runtime.

More formally, we can describe the runtime with the recurrence: $T(n) \le T(\frac{2n}{3}) + \Theta(1)$.

We can always split the problem into at least 2/3 the size. Consider the number of nodes on the left side vs. the right in the most unbalanced state:

In the worst case a heap of height n has all of the bottom leaves of the left child filled and the right child has height n-1. This is the most unbalanced a tree will ever become due to the heap property.

For any complete binary tree of n nodes and l leaves, where the lowest level is full, l=n+1. That is, half of the tree is leaves.

For a tree of height h, the number of leaves in a complete binary tree is 2^h and the number of nodes (not counting leaves) is 2^h-1.

So in the worst case, the left subtree has about $\frac{1}{2}2^h + \frac{1}{2}2^h$ leaves + nodes.

The right subtree has $\frac{1}{2}2^h$ nodes.

If we take the ratio of the left subtree over the total number of nodes:

$$\frac{\frac{1}{2}2^h + \frac{1}{2}2^h}{\frac{1}{2}2^h + \frac{1}{2}2^h + \frac{1}{2}2^h} = \frac{2}{3}$$

So we are able to split the problem by at least 1/3 each iteration of the loop in the worst case (and this would only happen once).

Given : $T(n) = T(\frac{2n}{3}) + \Theta(1)$ Can solve by the master theorem.

Case 2:

a=1,b=3/2

Is $\Theta(1) = \Theta(n^{\log_{1.5}1})$?

$\Theta(1) = \Theta(n^0)$?

YES, so T(n)= $\Theta(f(n)\lg n) = \Theta(\lg n)$

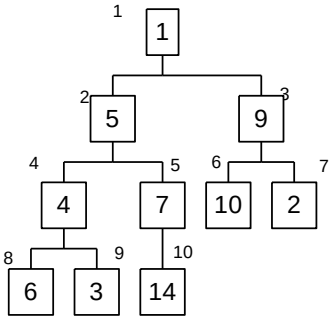
Building The Heap:

Given an array A, we want to build this array into a heap.
Note: Leaves are already a heap! Start from the leaves and build up from there.

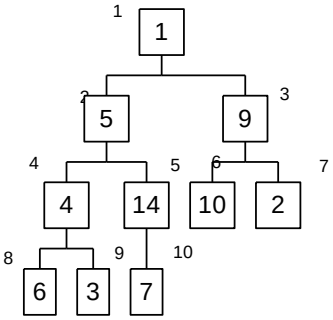
```
Build-Heap(A,n)
        for I ← n downto 1                    ; could start at n/2
                do Heapify(A,I,n)
```

Start with the leaves (last ½ of A) and consider each leaf as a 1 element heap. Call heapify on the parents of the leaves, and continue recursively to call Heapify, moving up the tree to the root.

Example: Build-Heap(A,10). A=[1 5 9 4 7 10 2 6 3 14]

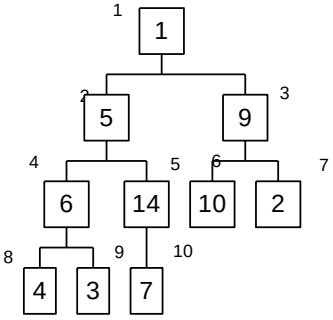


Heapify(A,10,10) exits since this is a leaf.
Heapify(A,9,10) exits since this is a leaf.
Heapify(A,8,10) exits since this is a leaf.
Heapify(A,7,10) exits since this is a leaf.
Heapify(A,6,10) exits since this is a leaf.
Heapify(A,5,10) puts the largest of A[5] and its children, A[10] into A[5]:



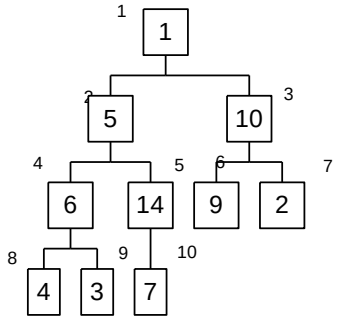
A=[1 5 9 4 14 10 2 6 3 7]

Heapify(A,4,10):



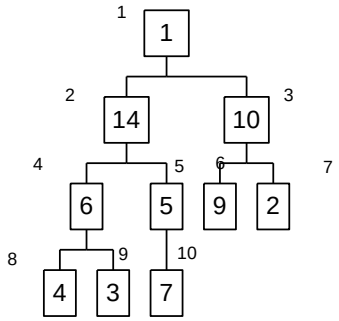
A=[1 5 9 6 14 10 2 4 3 7]

Heapify(A,3,10):

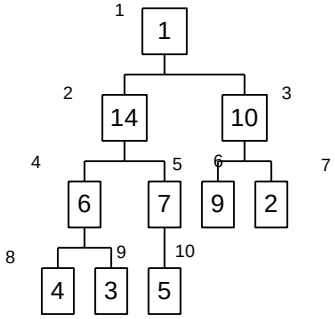


A=[1 5 10 6 14 9 2 4 3 7]

Heapify(A,2,10): First iteration:

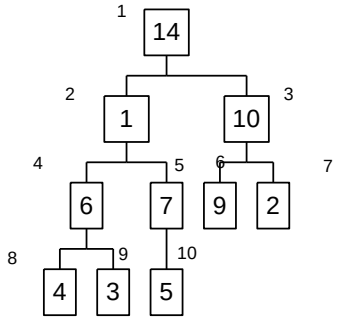


this calls Heapify(A,5,10):

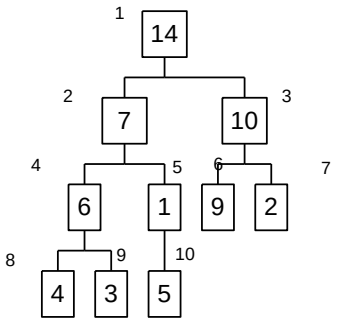


A=[1 14 10 6 7 9 2 4 3 5]

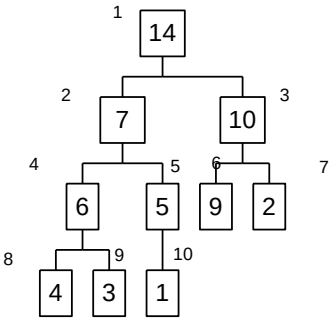
Heapify(A,1,10):



Calls Heapify(A,2,10):



Calls Heapify(A,5,10):



Finished heap: A=[14 7 10 6 5 9 2 4 3 1]

Running Time: We have a loop of n times, and each time call heapify which runs in Θ(lgn). This implies a bound of O(nlgn). This is correct, but is a loose bound! We can do better. Note: This is a good approach in general. Start with whatever bound you can determine, then try to tighten it.

Key observation: Each time heapify is run within the loop, it is not run on the entire tree. We run it on subtrees, which have a lower height, so these subtrees do not take lgn time to run. Since the tree has more nodes on the leaf, most of the time the heaps are small compared to the size of n.

Better Bound for Build-Heap:

Property: In an n-element heap there are at most $\frac{n}{2^h}$ nodes of height h (The leaves are height 1 and root at lgn, this is backwards from normal). The time required by Heapify when called in Build-Heap on a node at height h is O(h); h=lgn for the entire tree.

Cost of Build-Heap is:

$$T(n) = \sum_{h=1}^{heap\_height} (\#nodes\_at\_h)(Heapify - Time)$$

$$T(n) = \sum_{h=1}^{\lg n} \frac{n}{2^h} O(h)$$

$$T(n) = O\left(\sum_{h=1}^{\lg n} \frac{n}{2^h} h \right)$$

We know that $\sum_{n=0}^{\infty} nx^n = \frac{x}{(1-x)^2}$. If x=1/2 then $(1/2)^n=1/2^n$ so:

$$\sum_{n=0}^{\infty} h\left(\frac{1}{2}\right)^h = \frac{1/2}{(1-1/2)^2} = 2$$

Substitute this back in, which is safe because the sum from 0 to infinity is LARGER than the sum from 1 to lgn. This means we are working with a somewhat looser upper bound on the right hand side::

$$T(n) \le O\left(n \sum_{h=0}^{\infty} h \frac{1}{2^h} \right)$$

$$T(n) \le O(n2)$$

$$T(n) = O(n)$$

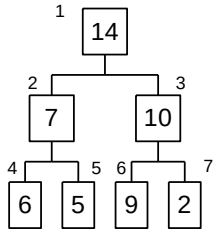
DONE!

HeapSort: Once we can build a heap and heapify a heap, sorting is easy. Idea is to:

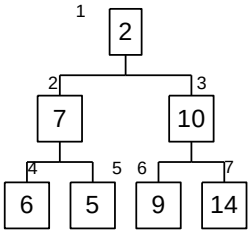
```
HeapSort(A,n)
        Build-Heap(A,n)
        for I ←n downto 2
                do      Swap(A[1]↔A[I]
                        Heapify(A,1,I-1)
```

Slightly different from book. The book removes root, puts into sorted list. In this example we are sticking it at the end of the array and the loop decreases the size, so the element is not touched again.

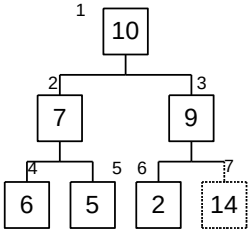
Example: HeapSort(A,7) A=[14 7 10 6 5 9 2] (already a heap)



Swap root with 7:

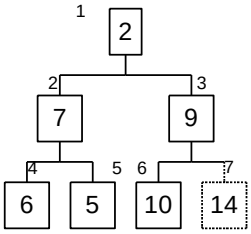


Heapify(A,1,6)

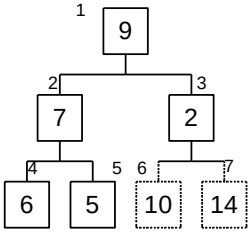


A=[10 7 9 6 5 2 14]

Swap root with 6:

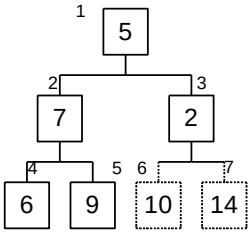


Heapify(A,1,5)

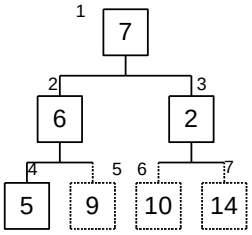


A=[9 7 2 6 5 10 14]

Swap root with 5:

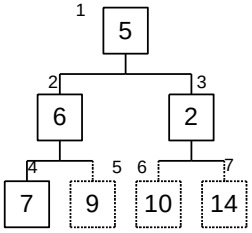


Heapify(A,1,4)

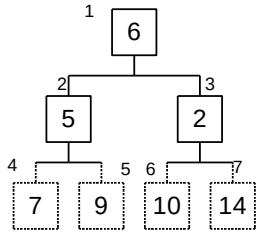


A=[7 6 2 5 9 10 14]

Swap root with 4:

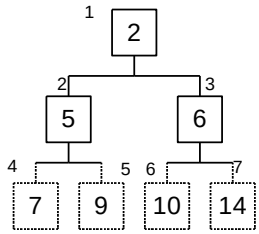


Heapify(A,1,3)

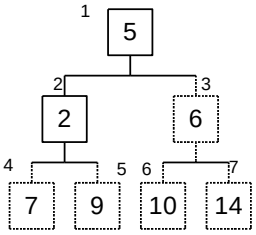


A=[6 5 2 7 9 10 14]

Swap root with 3:

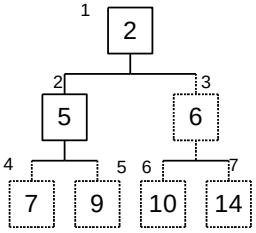


Heapify(A,1,2)



A=[5 2 6 7 9 10 14]

Swap root with 2:



A=[2 5 6 7 9 10 14]

We are done!

Runtime is O(nlgn) since we do Heapify on n-1 elements, and we do Heapify on the whole tree. Note: In-place sort, required no extra storage variables unlike Merge Sort, which used extra space in the recursion. Nice alternative to QuickSort with guaranteed O(nlgn) runtime.

Variation on heaps:

Heap could have min on top instead of max
Heap could be k-ary tree instead of binary

Priority Queues: A priority queue is a data structure for maintaining a set of S elements each with an associated key value. Operations:

Insert(S,x) puts element x into set S
Max(S,x) returns the largest element in set S
Extract-Max(S) removes the largest element in set S

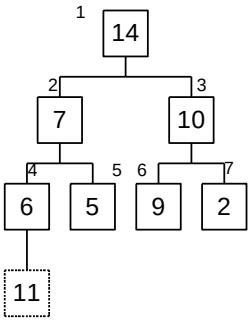
Max(S,x) : Just return root element. Takes O(1) time.

Heap-Insert(A,key)

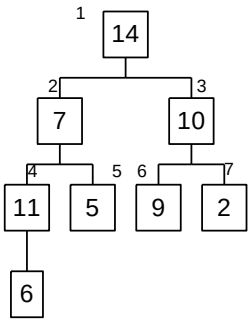
n ← n+1
I ← n
while I > 1 and A[$\lfloor i/2 \rfloor$] < key
do A[I] ← A[$\lfloor i/2 \rfloor$]
I ← $\lfloor i/2 \rfloor$
A[I] ← key

Idea: same as heapify. Start from a new node, and propagate its value up to right level.

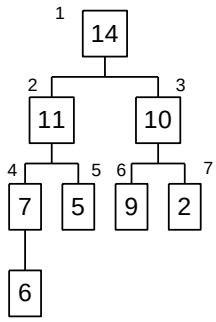
Example: Insert new element “11” starting at new node on bottom, I=8



Bubble up:



I=4, bubble up again



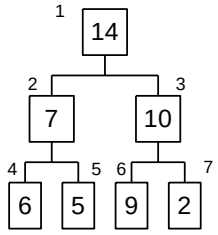
At this point, the parent of 2 is larger so the algorithm stops.
Runtime = O(lgn) since we only move once up the tree levels.

Heap-Extract-Max(A,n)

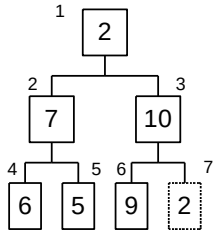
max ← A[1]
A[1] ← A[n]
n ← n-1
Heapify(A,1,n)
return max

Idea: Make the nth element the root, then call Heapify to fix.
Uses a constant amount of time plus the time to call Heapify, which is O(lgn).
Total time is then O(lgn).

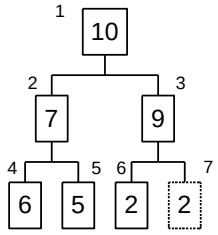
Example: Extract(A,7):



Remove 14, so max=14. Stick 7 into 1:



Heapify (A,1,6):



We have a new heap that is valid, with the max of 14 being returned. The 2 is sitting in the array twice, but since n is updated to equal 6, it will be overwritten if a new element is added, and otherwise ignored.

Non-Comparison Based Sorting

How fast can we sort?

Insertion-Sort $O(n^2)$

Merge-Sort, Quicksort (expected), Heapsort : $\Theta(n \lg n)$

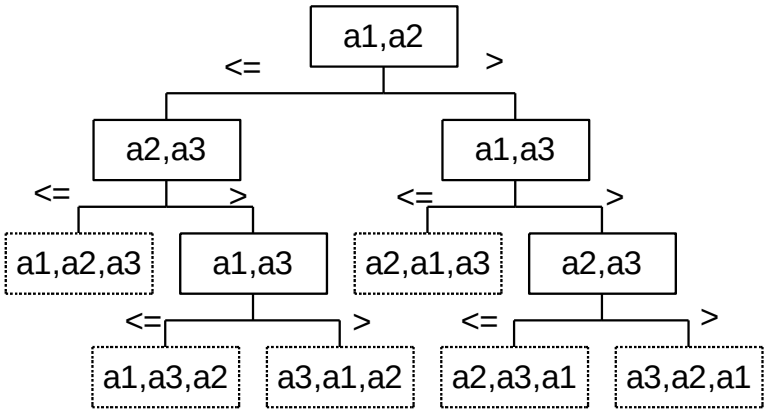
Can we do faster? What is the theoretical best we can do?

So far we have done comparison sorts: A sort based only on comparisons between input elements. E1<E2, E1=E2, E1>E2. We will show that any comparison-based sort MUST make $\Omega(n \lg n)$ comparisons. This means that merge sort and heap sort are optimal.

This is important because it is not always possible that you can prove that your algorithm is the best one possible for a problem!

A decision tree is used to represent the comparisons of a sorting algorithm. Assume that all inputs are distinct. A decision tree compares all possible inputs to each other to determine the sequence of outputs.

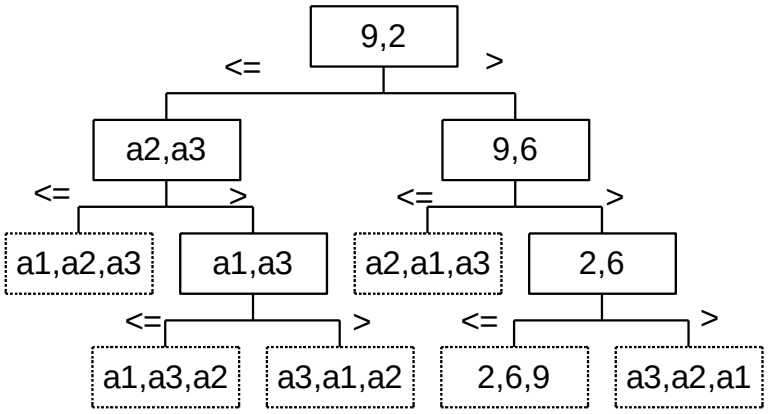
Decision Tree for three elements a1,a2,a3 : If at the root, a1 ≤ a2 go left and compare a2 to a3, otherwise go right and compare a1 to a3. Each path represents a different ordering on a1,a2,a3.



This type of decision tree will have n! leaves – one for each permutation of the input.

Any comparison-based sorting algorithm will have to go through the steps in the decision tree as a minimum (can do more comparisons if we want to, of course!)

Example of 9,2,6 :



The sorted elements are 2,6,9, in order of a2,a3,a1.

Decision trees can model comparison sorts. For any sorting algorithm:

1. One tree for each input length n
2. An algorithm "splits" at each decision/comparison unwinding the actual execution into a tree path
3. The tree is all possible execution traces

What is the height of the decision tree? This gives us the minimum number of comparisons necessary to sort the input.

For n inputs, the tree must have n! leaves. A binary tree of height h has no more than 2^h leaves:

$$n! \leq 2^h$$

Take log:

$$\lg(n!) \leq h$$

Stirling's approximation says that $n! > \sqrt{2n\pi}(n/e)^n > (n/e)^n$

So:
$$\lg(n/e)^n \leq h$$
$$n\lg(n/e) \leq h$$
$$n(\lg n - \lg e) \leq h$$
$$n\lg n - n\lg e \leq h$$

This means h= $\Omega(n\lg n)$ and we are DONE! We need to do at least nlgn comparisons to reach the bottom of the tree.

Does this mean that we can't do any better?? NO! (well, in some cases)
We can actually do some types of sorting in LINEAR TIME.

Counting Sort

This may work in O(n) time. How? Because it uses no comparisons! But we have to make assumptions about the size and nature of the input.

Input: A[1..n] where A[I]={1…k}
Output: B[1..n], sorted
Uses: C[1..k] auxiliary storage

Idea: Using random access array, count up number of times each input element appears and then collect them together.

Algorithm:

```
Count-Sort(A,n)
    for I←1 to k do C[I] ←0          ; Initialize to 0
    for j←1 to n do C[A[j]] ++       ; Count
    j←1
    for I←1 to k do
        if (C[I]>0) then
            for z←1 to C[I] do
                B[j]=I
                j++
```

Ex: A=[1 5 3 2 2 4 9]
C=[0 0 0 0 0 0 0 0 0]
C=[1 2 1 1 1 0 0 0 1]
B=[1 2 2 3 4 5 9]

This works! How long does it take? O(n+k). If k=n, then this runs in O(n) time. However, a bad example would be a input list like A[1,2,99999999].

One disadvantage of the current algorithm: it is not **stable**
An algorithm is stable if the occurrences of a value I appear in the same order in the output as they do in the input. That is, ties between two numbers are broken by the rule that whichever number appears first in the input array appears first in the output array.

Why do we want a stable algorithm? If the thing we are sorting is just a key of a record (perhaps a zip code, or a job indicating priority where we want the first one in to have precedence) then stability may be important.

Ex: A[3 5a 9 2 4 5b 6]
Sorts to A[2 3 4 5a 5b 6 9]
and not to A[2 3 4 5b 5a 6 9]

Can modify algorithm to make it stable:

```
Stable-Count-Sort(A,n)
        for I←1 to k do C[I] ←0          ; Initialize to 0
        for j←1 to n do C[A[j]] ++       ; Count
        for I←2 to k do
                C[I] ←C[I]+C[I-1]        ; Sum elements so far
                                         ; C[I] contains num elements <= I
        for j←n downto 1 do
                B[C[A[j]]] ←A[j]
                C[A[j]] ←C[A[j]]-1
```

Example: A=[1 5 3 2 2 4 9]
C=[0 0 0 0 0 0 0 0 0]
C=[1 2 1 1 1 0 0 0 1]
C=[1 3 4 5 6 6 6 6 7]
B=[. . . . . 9]
C=[1 3 4 5 6 6 6 6 6]
B=[. . . . 4 . 9]
…
B=[1 2 2 3 4 5 9]

This version is stable, since we fetch from the original array.

Radix Sort

Works like the punch-card readers of the early 1900's. Only works on input items with digits!
Idea somewhat counterintuitive: Sort on the least significant digit first.

```
Radix-Sort(A,d,n)          ; A is an n element array, each element d digits long
        for i←1 to d
                do  Use a stable sort to sort array A on digit i
```

Example:

A						
492		031		102		031
299		492		204		102
102		102		031		204
031	→	204	→	835	→	299
996		835		492		492
204		996		996		835
835		299		299		996

Sort must be stable so numbers chosen in the correct order! Assumes that lower order digits are already sorted to work.

If each digit is not large, counting sort is a good choice to use for the sort method.
If k is the maximum value of the digit, then counting sort takes $\Theta(k+n)$ time for one pass. We have to make d passes, so the total runtime is $\Theta(dk+dn)$.

If d is a constant and k is smaller than O(n), Radix-Sort runs in O(n) linear time!

Radix or Counting sorts are simple to code and the method of choice if the input is of the right form.

Bucket Sort

Similar to count sort, but uses a "bucket" to hold a range of inputs. Works for real numbers!
Like the other sorts, bucket sort is fast because it assumes something about the input:

1. Input is randomly generated
2. Input elements randomly distributed over the interval [0..1]. In many cases we can divide by some "max" value to force the input key for comparison to be between 0 and 1. This assumption means that elements are generated with uniform probability over [0..1] or that each element has the same likelihood of being generated.

Idea:
1. Divide [0..1] into n equal sized parts or "buckets"
2. Put each of the n inputs into one of the buckets. Some buckets may be empty and some may have more than 1 element.
3. Sort each bucket.
4. To produce output, go through the buckets in order, listing the elements in each.

Linked Lists is a good mechanism for storing the buckets.

```
Bucket-Sort(A,n)
for i ← 1 to n do
        Insert A[I] into list B[nA[I]]
for i ← 0 to n-1 do
        sort list B[I] with insertion sort
concatenate the lists B[0], B[1], … B[n-1] together in order
```

Buckets are automatically numbered in this case from 0..n-1

All the lines but line 5 take O(n) time in the worst case.
Line 5 is insertion sort which takes $O(n^2)$ time but since the input is generated uniformly we dont expect any bucket to have many elements in it so Insertion-Sort should be called on very small lists.

Example:

A=[0.44 0.12 0.73 0.29 0.67 0.49]
Bucket I will get the values between I/n and (I+1)/n since buckets are numbered from 0 to n-1.

B
0..0.16 $\rightarrow$ 0.12
0.16..0.33 $\rightarrow$ 0.29
0.33..0.50 $\rightarrow$ 0.44 $\rightarrow$ 0.49
0.50..0.66 $\rightarrow$
0.66..0.83 $\rightarrow$0.73 $\rightarrow$0.67
0.83..1 $\rightarrow$

Sort the buckets with insertion sort and then combine buckets to get:

0.12 0.29 0.44 0.49 0.67 0.73

Informal Argument on the average time:

Since any element in A comes from [0..1] with an equal probability then the probability that an element e is in bucket B[I] is 1/n (each bucket covers 1/n of [0..1].

This means that the average number of elements that end up in bucket B[I] is 1. There is a little more to the analysis than this, but the basic idea is that the distribution of the input will cause the calls to Insertion-Sort to be on very short lists and so the other steps in the algorithm will use more time. The average running time of Bucket-Sort is then T(n)=O(n).

Postman Sort

There are many other sorting algorithms that have been proposed. Robert Ramey proposed the Postman Sort in the August 1992 issue of the C Programming journal. We will briefly discuss it here and may use it as an exercise. The full article is available at http://www.rrsd.com/software_development/postmans_sort/index.htm.

To quote Ramey's article regarding a generalized distributed sorting algorithm:

> When a postal clerk receives a huge bag of letters he distributes them into other bags by state. Each bag gets sent to the indicated state. Upon arrival, another clerk distributes the letters in his bag into other bags by city. So the process continues until the bags are the size one man can carry and deliver. This is the basis for my sorting method which I call the postman's sort.
>
> Suppose we are given a large list of records to be ordered alphabetically on a particular field. Make one pass through the file. Each record read is added to one of 26 lists depending on the first letter in the field. The first list contains all the records with fields starting with the letter "A" while the last contains all the records with fields starting with the letter "Z". Now we have divided the problem down to 26 smaller subproblems. Now we address subproblem of sorting all the records in the sublist corresponding to key fields starting with the letter "A". If there are no records in the "A" sublist we can proceed to deal with the "B" sublist. If the "A" sublist contains only one record it can be written to output and we are done with that sublist. If the "A" sublist contains more that one record, it must be sorted then output. Only when the "A" list has been disposed of we can move on to each of the other sublists in sequence. The records will be written to the output in alphabetical order. When the "A" list contains more than one record it has to be sorted before it is output. What sorting algorithm should be used? Just like a real postman, we use the postman's sort. Of course we just apply the method to the second letter of the field. This is done to greater and greater depths until eventually all the words starting with "A" are written to the output. We can then proceed to deal with sublists "B" through "Z" in the same manner.

Example: Consider sorting "BOB", "BILL", BOY", "COW", "DOG"

How fast is it? (Exercise for the reader)