The Heap and the Quick

Serge Kruk

October 12, 2021

Sorts

- Brute-force (sort of) $O(n^2)$
  - Insertion sort
  - Bubble sort
- Divide-and-conquer $O(n \log n)$
  - Merge sort
  - Heap sort
  - Quick sort

Bubble

```python
def bs(a=[]):
    n = len(a)
    for i in range(n-1):
        for j in range(i+1,n):
            if a[i] > a[j]:
                a[i],a[j] = a[j],a[i]
    return a
```

Loop invariant

Entering the outer loop with `i` at value `k` the sub-array `a[0 .. k-1]` contains the k smallest elements of the whole array, in a sorted order.

Insertion sort

```python
def isort(a):
    for i in range(1,len(a)):
      currentvalue = a[i]
      position = i

      while position>0 and a[position-1]>currentvalue:
          a[position] = a[position-1]
          position = position-1

      a[position] = currentvalue
    return a
```

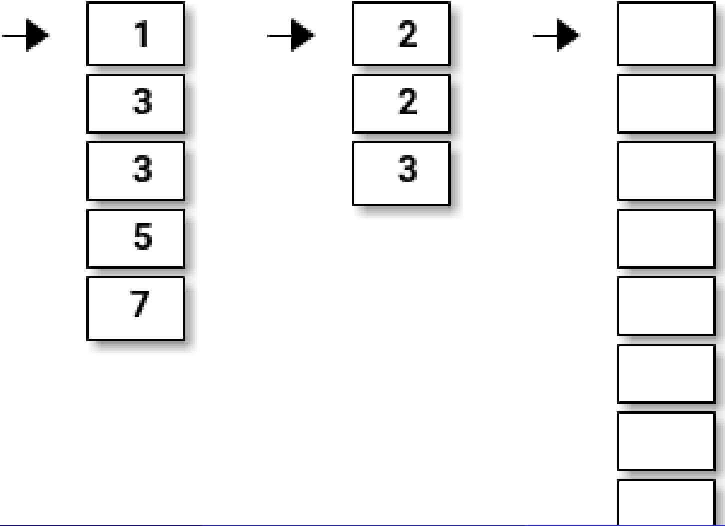
Loop invariant

Entering the outer loop with `i` at value `k` the sub-array `a[0 .. k]` is sorted.

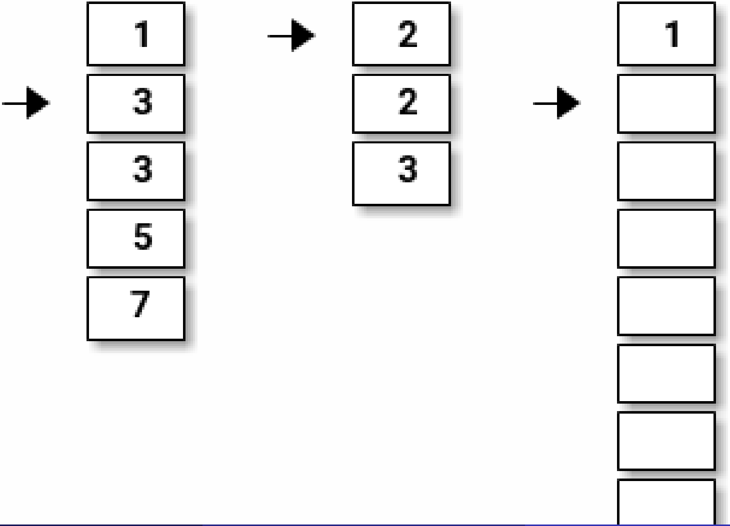
How to merge two sorted arays

- Input is a pair of sorted arrays `a, b`
- Output is a new array, sorted, containing all elements of `a` and `b`.

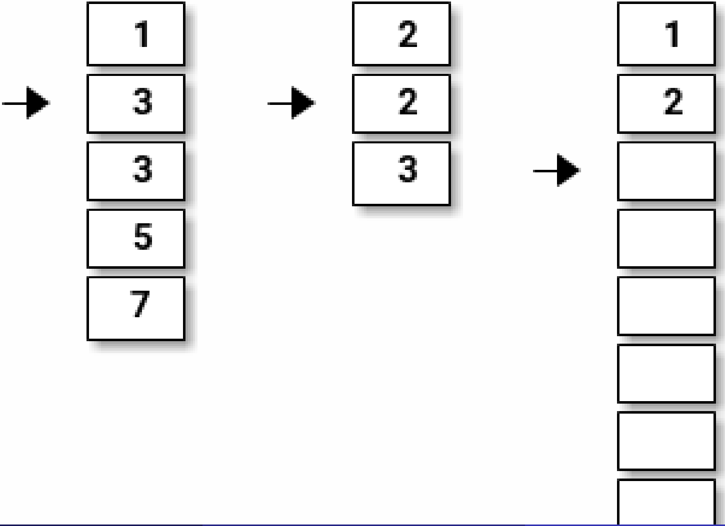
Merging



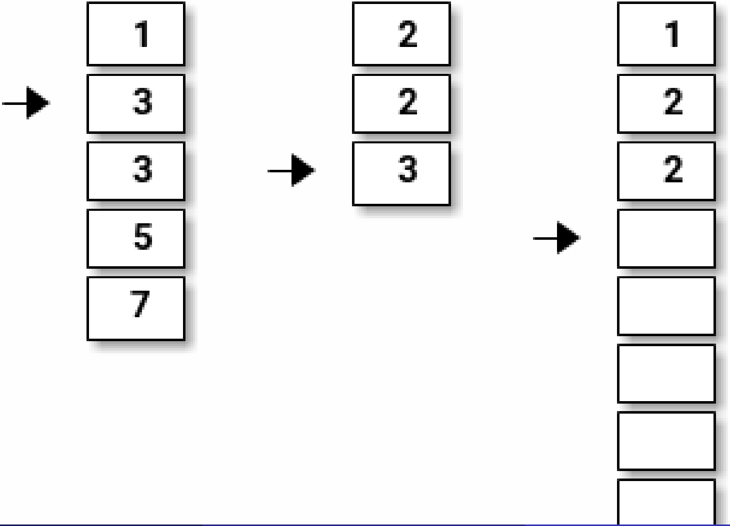
Merging



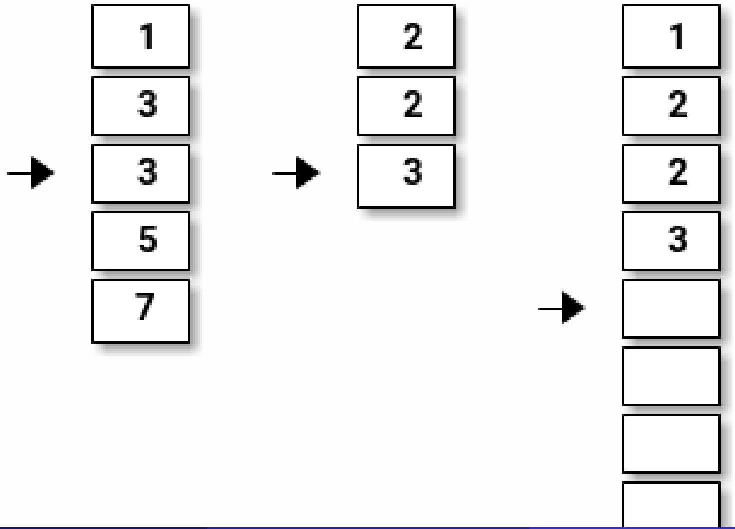
Merging



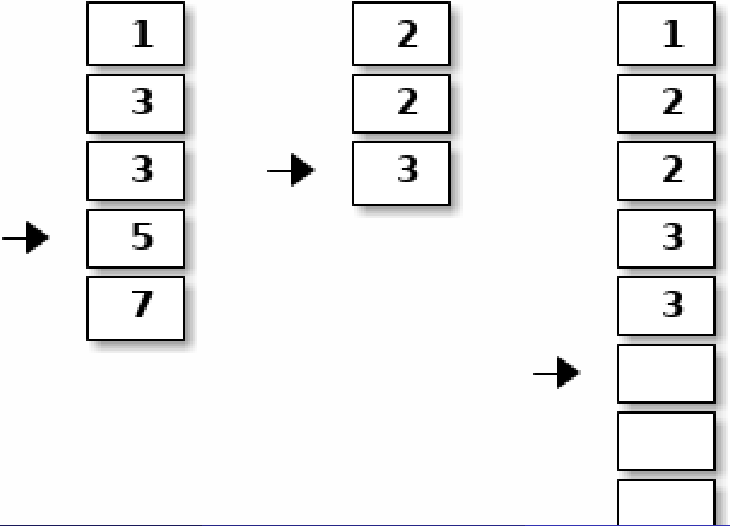
Merging



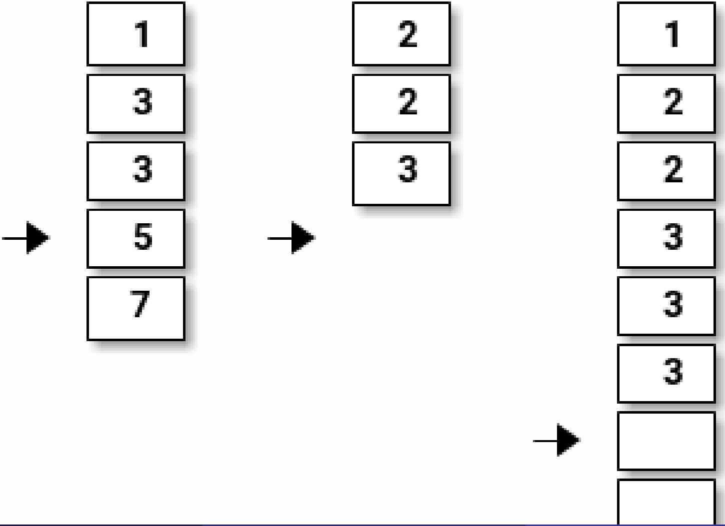
Merging



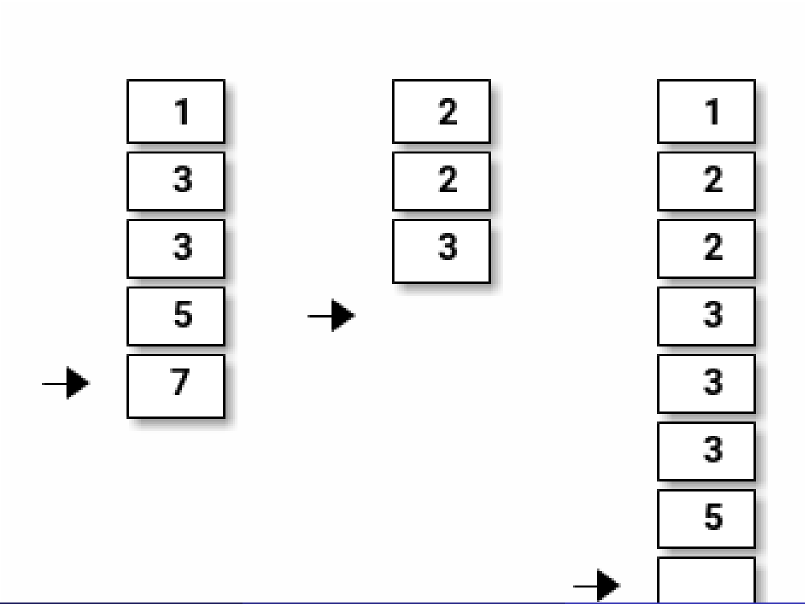
Merging



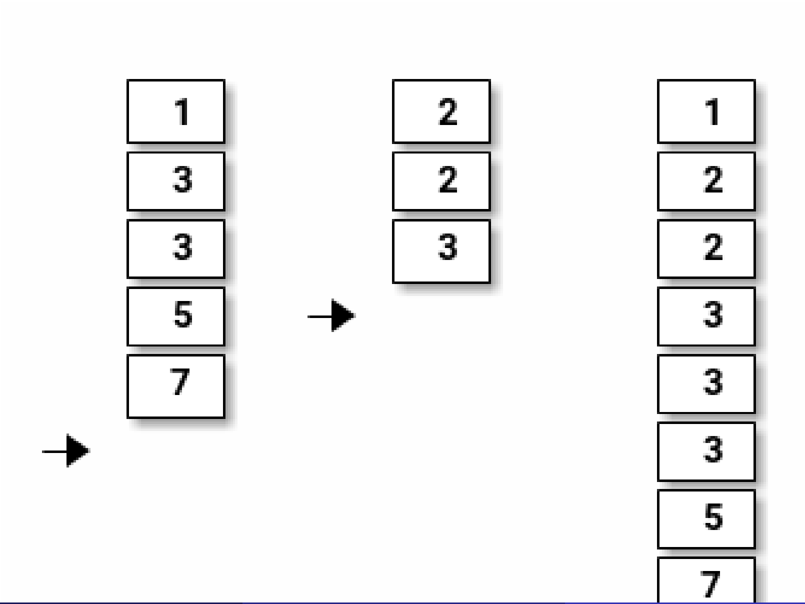
Merging



Merging



Merging



How to merge two sorted arays

```
def merge(a,b):

  return c
```

```
def merge(a,b):
    ia,ib,ic,na,nb = 0,0,0,len(a),len(b)
    nc = na+nb
    c=[0]*nc
    while (ic < nc):
      if (ia < na):
        if (ib < nb):
          if (a[ia] < b[ib]):
            c[ic],ic,ia = a[ia],ic+1,ia+1
          else:
            c[ic],ic,ib = b[ib],ic+1,ib+1
        else:
          c[ic],ic,ia = a[ia],ic+1,ia+1
      else:
        c[ic],ic,ib = b[ib],ic+1,ib+1
    return c
```

Runtime

Given

- n_a as the length of `a`
- n_b as the length of `b`
- Runtime is $\Theta(n_a + n_b)$

Merge sort

General idea:

- Split in half
- Sort the left part
- Sort the right part
- Merge the sorted halfs (which we did above)
- We can (and will) call `merge`. Fill-in the missing part.

```
def msort(a):
    ...
    return ...
```

Merge sort

- ```
  def msort(a):
      n = len(a)
      m = n//2
      if n <= 1:
          return a
      else:
          return merge(msort(a[0:m]), msort(a[m:]))
  ```
- ```
  print([msort([])==[],
         msort([4])==[4],
         msort([1,3,3,2,3,1])==[1,1,2,3,3,3],
         msort([10,9,8,7,6,5,4,3,2,1]) == [1,2,3,4,5,6,7,8,9,10]]
  ```

  ```
  [True, True, True, True]
  ```

Merge sort

- As a one-liner. Just to mess with you.

```
def msort(a):
  return a if len(a)//2 <= 1 else
         merge(msort(a[0:len(a)//2]), msort(a[len(a)//2:]))
```

Merge sort

Correctness proof

By induction

Proof of correctness

- Base case: if the array is of length 0 or 1, we return this array.
- Induction hypothesis: Assume `msort` works for arrays of length 0 to k.
  - Consider an array of length any length up to $k + l$ (as long as $k + l < 2k$)
  - We split it in two parts, one of length $\lfloor (k+l)/2 \rfloor$, the other $\lceil (k+l)/2 \rceil$
  - Since both parts are at most length k, the hypothesis holds and they return sorted.
  - Assuming that `merge` works correctly, we return a sorted array of length $k + l$.

Conclusion

`msort` is correct if `merge` is correct.

Merge sort runtime

```
def msort(a):
  m = len(a)//2
  if m <= 1:
    return a
  else:
    return merge(msort(a[0:m]), msort(a[m:]))
```

Fill-in the recurrence

$$T(n) = \begin{cases} 1 & n \leq 1 \\ ??? & \text{Otherwise} \end{cases}$$

Merge sort runtime

$$T(n) = \begin{cases} 1 & n \leq 1 \\ 2T(n/2) + n & \text{Otherwise} \end{cases}$$

Solves to $\Theta(n \log n)$.

Trick(?) question

```python
def msort(a):
  m = len(a)//2
  if m <= 1:
    return a
  else:
    return merge(msort(a[0:m]), msort(a[m:]))
```

If I call `msort([a1, a2, a3, a4, a5, a6, a7, a8])` what are the next three calls to msort to start executing?

- `msort([a1, a2, a3, a4, a5, a6, a7, a8])`
    - `msort([a1, a2, a3, a4])`
    - `msort([a1, a2])`
    - `msort([a1])`

A sort not covered in textbooks anymore

Shell Sort

Shell sort

For the next few minutes, keep this sequence of integers in mind.

```
gaps = [701, 301, 132, 57, 23, 10, 4, 1]
```

(It is called a gap sequence.)

Shell sort

```
def shellSort(lst,gaps):
    for gap in gaps:
        for i in range(gap):
            gapISort(lst,i,gap)

def gapISort(lst,start,gap):
    for i in range(start+gap,len(lst),gap):
        v = lst[i]
        p = i
        while p >= gap and lst[p-gap] > v:
            lst[p] = lst[p-gap]
            p = p-gap
        lst[p]=v
```

Proof of correctness

The last gap is an insertion sort.

The gap sequence determines the runtime

- $\lfloor n/2^k \rfloor$ will yield $\Theta(n^2)$
- $2\lfloor n/2^{k+1} \rfloor + 1$ will yield $\Theta(n^{\frac{3}{2}})$
- $2^p 3^q$ will yield $\Theta(n \log^2 n)$
- Better? Best? Is still open!

Read

- Knuth The Art of Computer Programming for beautiful mathematics.
- On Nostalgia Night, I will tell you why I am fond of ShellSort.

Sorts up to now

Runtime

- From $O(n^2)$ down to $O(n \log n)$
- Depends on clever procedures

Now for something completely different

```
def abstractSort(a):
    n,na = len(a),[]
    for i in range(n):
        smallest = extractSmallestAndDelete(a)
        append smallest to na
    return na
```

- We could easily do this in $O(n^2)$.
- We will attempt to do it more efficiently.

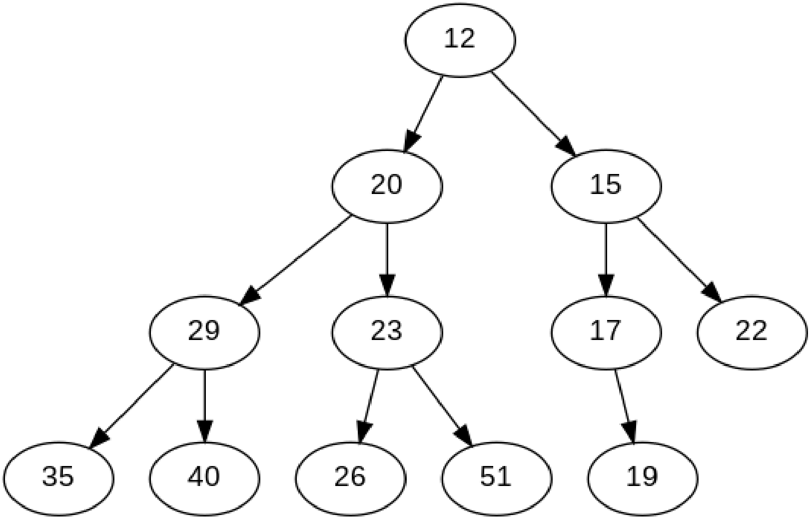
Consider the following data structure properties

- Binary Tree-like, but always balanced (unlike BST)
- Invariant: the value of a node is always less than the value of its children.

It exists.

It is called a `min-heap`

Graphical representation



Implementation

How would you do it?

Here is the "standard" implementation

- Store in an array of size $1 + n$

Here is the “standard” implementation

- Store in an array of size $1 + n$
- The number of elements is in position 0

Here is the “standard” implementation

- Store in an array of size $1 + n$
- The number of elements is in position 0
- The Root is at position 1

Here is the “standard” implementation

- Store in an array of size $1 + n$
- The number of elements is in position 0
- The Root is at position 1
- The Left child of node at position i is at 2i

Here is the “standard” implementation

- Store in an array of size $1 + n$
- The number of elements is in position 0
- The Root is at position 1
- The Left child of node at position i is at 2i
- The Right child of node at position i is at 2i+1

Here is the “standard” implementation

- Store in an array of size $1 + n$
- The number of elements is in position 0
- The Root is at position 1
- The Left child of node at position i is at 2i
- The Right child of node at position i is at 2i+1
- Therefore, the parent of i is at $\lfloor i/2 \rfloor$

Defining property

Min-heap property

An array $a[1..n]$ has the `min-heap` property if

$$a\left[\lfloor i/2 \rfloor\right] \leq a[i] \quad 2 \leq i \leq n$$

- Note that this does not imply the array is sorted.

Building the heap

- We need to build this heap, one element at a time.

Building the heap

- We need to build this heap, one element at a time.
- We will assume that we know ahead of time how large it can grow.

Building the heap

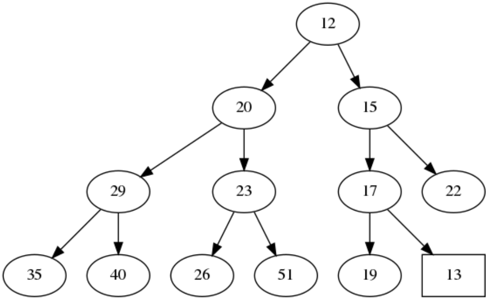
- We need to build this heap, one element at a time.
- We will assume that we know ahead of time how large it can grow.
- Start, we allocate an array of size one more and put 0 at position 0.

Building the heap

- We need to build this heap, one element at a time.
- We will assume that we know ahead of time how large it can grow.
- Start, we allocate an array of size one more and put 0 at position 0.
- Adding the first element is simple. You see this?

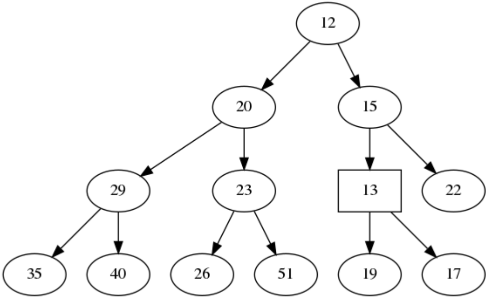
How do we add a node? Say value 13

Step 0 : Add it at the end



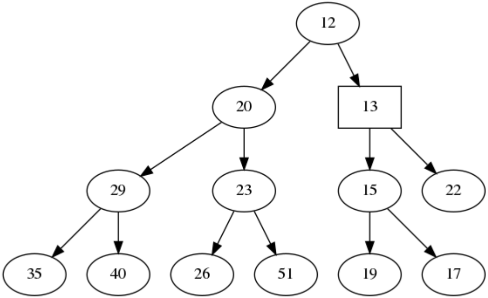
How do we add a node? Say value 13

Step 1 : Swap with its parent if necessary



How do we add a node? Say value 13

Step 2 : Repeat until `min-heap` property is restored



Let us decompose this into

```
def newheap(n):
    return [0]*(n+1)
def insert(a,e):
    # Inserting element e into min-heap a
    a[0] = a[0] + 1
    a[a[0]] = e
    heapfixup(a,a[0])
```

Implement this “fixing up”

```python
def heapfixup(a,i):
    # Fix up from position i to restore
    # min-heap property of heap a
```

Implementation

```python
def heapfixup(a,i):
    while i > 1:
        p = i // 2
        if a[p] > a[i]:
            a[p],a[i] = a[i],a[p]
            i = p
        else:
            return
```

Small tests

```
v = [1,4,2,5,3,6,4,7]
h = newheap(len(v))
print(h)
for e in v:
  insert(h,e)
  print(h)
```

```
[0, 0, 0, 0, 0, 0, 0, 0, 0]
[1, 1, 0, 0, 0, 0, 0, 0, 0]
[2, 1, 4, 0, 0, 0, 0, 0, 0]
[3, 1, 4, 2, 0, 0, 0, 0, 0]
[4, 1, 4, 2, 5, 0, 0, 0, 0]
[5, 1, 3, 2, 5, 4, 0, 0, 0]
[6, 1, 3, 2, 5, 4, 6, 0, 0]
[7, 1, 3, 2, 5, 4, 6, 4, 0]
[8, 1, 3, 2, 5, 4, 6, 4, 7]
```

Notice that the heap condition holds at every step

Correctness and runtime

- LI: Descendants of position i are larger than a[i].
- $\Theta(\log n)$ where n is the size of the heap since we divide by two at every step.

Recall our abstract sort

```
def abstractSort(a):
    n,na = len(a),[]
    for i in range(n):
        smallest= extractSmallestAndDelete(a)
        na.append(smallest)
    return na
```

- Where is the smallest? At position 1.
- Now what?

Extract smallest and delete

How would you do it?

Extract smallest and delete from heap

- Save the element at position 1
- Move element n to position 1
- Fix heap property downward

Implementation

```
def extractsmallest(a):
    e,a[1],a[0] = a[1],a[a[0]],a[0]-1
    heapfixdown(a,1)
    a[a[0]+1]=0
    return e
```

Implementation

```
def heapfixdown(a,i):
```

Implementation

```
def heapfixdown(a,i):
    while 2*i <= a[0]:
        c = 2*i
        if c+1 <= a[0]:
            if a[c+1] < a[c]:
                c = c+1
        if a[i] > a[c]:
            a[i],a[c] = a[c],a[i]
            i = c
        else:
            return
```

Small tests

```
v = [1,4,2,5,3,6,4,7]
h = newheap(len(v))
for e in v:
  insert(h,e)
print(0, "--", h)
for _ in range(h[0]):
  e = extractsmallest(h)
  h[h[0]+1] = e
  print(e, "--", h)
```

```
0 -- [8, 1, 3, 2, 5, 4, 6, 4, 7]
1 -- [7, 2, 3, 4, 5, 4, 6, 7, 1]
2 -- [6, 3, 4, 4, 5, 7, 6, 2, 1]
3 -- [5, 4, 5, 4, 6, 7, 3, 2, 1]
4 -- [4, 4, 5, 7, 6, 4, 3, 2, 1]
4 -- [3, 5, 6, 7, 4, 4, 3, 2, 1]
5 -- [2, 6, 7, 5, 4, 4, 3, 2, 1]
6 -- [1, 7, 6, 5, 4, 4, 3, 2, 1]
7 -- [0, 7, 6, 5, 4, 4, 3, 2, 1]
```

Correctness and runtime

- LI: Every ancestor of position i is smaller than a[i].
- $\Theta(\log n)$

We have all we need to create our sort

```
def abstractSort(a):
    n,na = len(a),[]
    for i in range(n):
        smallest= extractSmallestAndDelete(a)
        na.append(smallest)
    return na
```

Heapsort

```
def heapsort(x):
    n = len(x)
    a = newheap(n)
    for i in range(n):
        insert(a,x[i])
    for i in range(n):
        x[i] = extractsmallest(a)
    return x
```

Runtime

- Runtime : $\Theta(n \log n)$
- Space required: $2n$

Morals

The distinction between algorithms and data structure is fuzzy!

Extracting the data structure of heap sort

The operations:

- Insert
- Extract minimum and delete

Form the basis of an abstract data structure called a priority queue
Applications include scheduling jobs by an operating system.

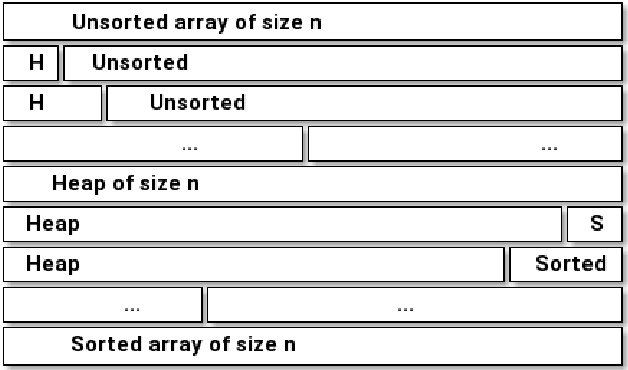
Various implementation of priority queues

	Insert	Extract
Sorted array	n	1
Min heap	$\log n$	$\log n$
Unsorted array	1	n

Project: Heapsort can be improved

Hint: Use the space of the array for the heap. Discuss

Improved heapsort (graphical view)



Improved heapsort

- Runtime is still $\Theta(n \log n)$
- Space requirement is n

Homework/Test questions

- Insert the following items in a min-heap, 18,5,19,3,27,11. Draw the heap at each step.
- Is an array, sorted in non-decreasing order, a min-heap?
- Is a min-heap an array in non-decreasing order?

Project

Problem

Given an (unsorted) array of n elements, return the largest k elements.

- Simplistic, obvious approach? $(n \log n)$
- You are asked to do it in $(n \log k)$

```python
def largest_k(a,k=1):
    """a is an array of n elements. We return the k largest."""
    return b
```

Sorts up to now

- Brute-force (sort of) $O(n^2)$
  - Insertion sort
  - Bubble sort
- Clever modification of insertion sort
  - Shell $O(n^{3/2})$
  - Shell $O(n \log^2 n)$
- Divide-and-conquer $O(n \log n)$
  - Merge sort
- New data structure $O(n \log n)$
  - Heap sort

Now for the last "Classical" sort

Can you guess?

Sir C.A.R. Hoare

Algol 60 is a language so far ahead of its time that it was not only an improvement on its predecessors but also on nearly all of its successors.
Due credit must be paid to the designers of Algol 60 who included recursion in their language and allowed me to describe my invention (quicksort) so elegantly to the world.

Bubble sort before Algol

```
SUBROUTINE sort (array_x, array_y, datasize)
 REAL array_x(*)
 REAL array_y(*)
 INTEGER datasize
REAL x_temp
REAL y_temp
LOGICAL inorder
inorder = .false.
```

```
do 90 while (inorder.eq..false.)
 inorder = .true.
 do 91 i=1, (datasize-1)
 if (array_x(i).eq.array_x(i+1) ) then
  if (array_y(i).lt.array_y(i+1) ) then
   x_temp = array_x(i)
   y_temp = array_y(i)
   array_x(i) = array_x(i+1)
   array_y(i) = array_y(i+1)
   array_x(i+1) = x_temp
   array_y(i+1) = y_temp
   inorder = .false.
  endif
 endif
```

```
      if (array_x(i).lt.array_x(i+1) )then
       x_temp = array_x(i)
       y_temp = array_y(i)
       array_x(i) = array_x(i+1)
       array_y(i) = array_y(i+1)
       array_x(i+1) = x_temp
       array_y(i+1) = y_temp
       inorder = .false.
      endif
91    continue
90    continue
     END SUBROUTINE sort
```

Bubble sort in Algol

```
PROC sort = (REF[]DATA array)VOID:
(
  BOOL sorted;
  FOR size FROM UPB array - 1 BY -1 WHILE
    sorted := TRUE;
    FOR i FROM LWB array TO size DO
      IF array[i+1] < array[i] THEN
        swap(array[i:i+1]);
        sorted := FALSE
      FI
    OD;
    NOT sorted
  DO SKIP OD
);
```

Historical tidbits

- Before algol
    - Fortran (imperative)
    - Lisp (functional)
- Algol introduced
    - Coercion
    - Structures (unions)
    - Local variables
    - Recursion into imperative programs
- First compiler by Dijkstra and Zonneveld
    - Fits in 4K bytes
    - Written in a few months by two mathematicians

Quicksort (the most used sorting algo)

Illustration of brilliant idea. Say we have this array

55	41	59	26	53	58	97	93

Pick one element, say 55 then partition by moving smaller than 55 to the left and the larger (or equal) to the right.

41	26	53	55	59	58	97	93

Note that 55 is in its final place! (This is key)
Now recurse on both the left and right subarrays.

Partition

- Pick the first element of the array as the pivot.
- Move all smaller elements to the left of the pivot.
- Move all larger elements to the right of the pivot.

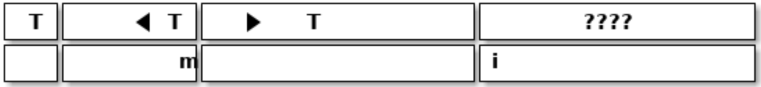
```
def partition(a,l,u)

  return m
```

Partition

```
def partition(a,l,u):
    t = a[l]
    m = l
    for i in range(l+1,u+1):
        if a[i] < t:
            m = m+1
            a[i],a[m] = a[m],a[i]
    a[m],a[l] = a[l],a[m]
    return m
```

Loop invariant



Implement Quicksort (recursively!)

- Hint: Divide and conquer
- Assume you have `partition`

```
def qsort(a):


    return a
```

Implementation

```
def qsort0(a,l=0,u=None):
    if u is None:
        u = len(a)-1
    if l < u:
        m = partition(a,l,u)
        qsort0(a,l,m-1)
        qsort0(a,m+1,u)
    return a
```

Runtime

- Assume that we will `partition` in $O(n)$.
- Assume that we partition `equally`.

$$T(n) = \begin{cases} C & n \leq n_0 \\ 2T(n/2) + n & \text{otherwise} \end{cases}$$

Runtime

$$
\begin{aligned}
T(n) &= 2T(n/2) + n \\
&= 2[2T(n/4) + n/2] + n & &= 4T(n/4) + 2n \\
&= 4[2T(n/8) + n/4] + 2n & &= 8T(n/8) + 3n \\
&= \vdots \\
&= 2^k T(n/2^k) + kn
\end{aligned}
$$

Since we do $\log n$ steps, we obtain $T(n) = \Theta(n \log n)$

A dare

For those of you who are still reluctant to consider recursion, I dare you to write quicksort without recursion (and get it right). It's doable, of course, but it's a nightmare to get right.

Inefficiency

- What if we partition with only one element on one side?

Inefficiency

$$
\begin{aligned}
T(n) &= T(n-1) + T(1) + n \\
&= T(n-2) + T(1) + n - 1 + T(1) + n \\
&= \ldots \\
&= T(1) + T(1) + \ldots + T(1) + 1 + 2 \\
&\quad + \ldots + n - 1 + n \\
&\in \Theta(n^2)
\end{aligned}
$$

Pivot choice

1. What happens if the partition is bad, say 10%, 90%?
2. Randomly is a possibility
3. Some form of median is best

Read the paper by McIllroy

Better implementation?

```
def qsort1(a,l=0,u=None):
    if u is None:
        u = len(a)-1
    if l < u-128:
        m = partition(a,l,u)
        qsort1(a,l,m-1)
        qsort1(a,m+1,u)
    else l < u:
        insertion_sort(a,l,u)
    return a
```

Why?

Practicalities

C

```c
void qsort(void *base,
           size_t nmemb,
           size_t size,
           int (*compar)(const void *, const void *));
```

Practicalities

Java

```java
public <A extends Comparable<? super A>>
      void sort(List<A> list) { }
```

Practicalities

Go

```go
func Sort(data Interface)
type Interface interface {
        Len() int
        Less(i, j int) bool
        Swap(i, j int)
}
```

Bentley and McIllroy

- Contrast
  - Textbook (Knuth): overhead $\approx$ comparisons $<$ swaps
  - Findings: overhead $<$ swaps $<$ comparisons
- Why? Because cmp is a function call (interpreted).
- Textbook presentation was flawed (from 1970 to 2000).
- Which explains why we are counting comparisons.

Homework/Test questions

- Why does Quicksort switch to insertion sort?
- What happens if the partition is 90 - 10?
- Which pivot choice is used by Bentley and McIllroy?
- Why would you not used a random pivot?
- What if there are multiple repeated elements? (Better partition?)
- Heapsort and Quicksort are both $n \log n$. Tradeoffs?
- Why would one try an iterative implementation of either?
- Which sort would be easier to implement in parallel?

Class project

You have two datasets indexed by student ID, one large and one small.
Your task is to extract the elements that appear in both sets?
You can call a sorting routine (no need to code it).
To simplify let us assume that the input to your function is a pair of arrays.
In each array, the elements are a tuple

Student ID (9digits)	**Student record (thousands of bytes)**

```python
def extract_common(A, B):
  """ A is HUGE;  B is relatively small """
  return C
```

Try to do this as efficiently as possible.