Algorithm and Object-Oriented Programming for Modeling
Part 2: Sort Related Algorithms

MSDM 5051, Yi Wang (王一), HKUST

Why sorting?

- It’s useful
- Ideal arena for understanding algorithm thinking
- Examples of many ideas which can be used elsewhere
  - More practice on divide & conquer
  - Ideas of heap, binary search trees, balance of trees, …

1. Insertion Sort

Idea:

- Get a new card from the right (loop i)
- Compare from right to left, put new card in place (loop j)

```python
def insertion_sort(array):
    n = len(array)
    for i in range(1, n): # The new card: (1, 2, ..., n-1)
        for j in range(i, 0, -1): # From the new card to old cards
            if array[j] < array[j-1]:    # Note: range(3, 0, -1) means 3, 2, 1
                array[j], array[j-1] = array[j-1], array[j]
            else:
                break
    return array
```

Swap (a, b) in C:

```
tmp = a;
a = b;
b = tmp;
```

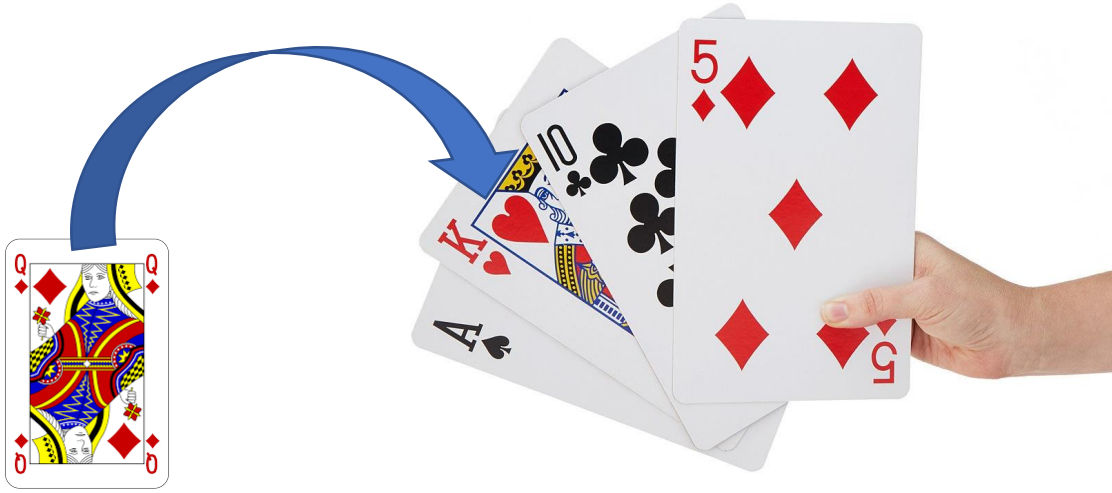
Swap (a, b) in Python:

```
a, b = b, a
```

```python
def insertion_sort(array):
    n = len(array)
    for i in range(1, n): # The new card: (1, 2, ..., n-1)
        for j in range(i, 0, -1): # From the new card to old cards
            if array[j] < array[j-1]:    # Note: range(3, 0, -1) means 3, 2, 1
                array[j], array[j-1] = array[j-1], array[j]
            else:
                break
    return array
```

Time complexity: $O(n^2)$

Can you improve it?



How did I insert a card?

Compare & swap one by one with a sorted list.

Can I try from the middle? (Binary search)

Yes, you can find insertion point in O(logn).

But insertion in the middle takes O(n) …

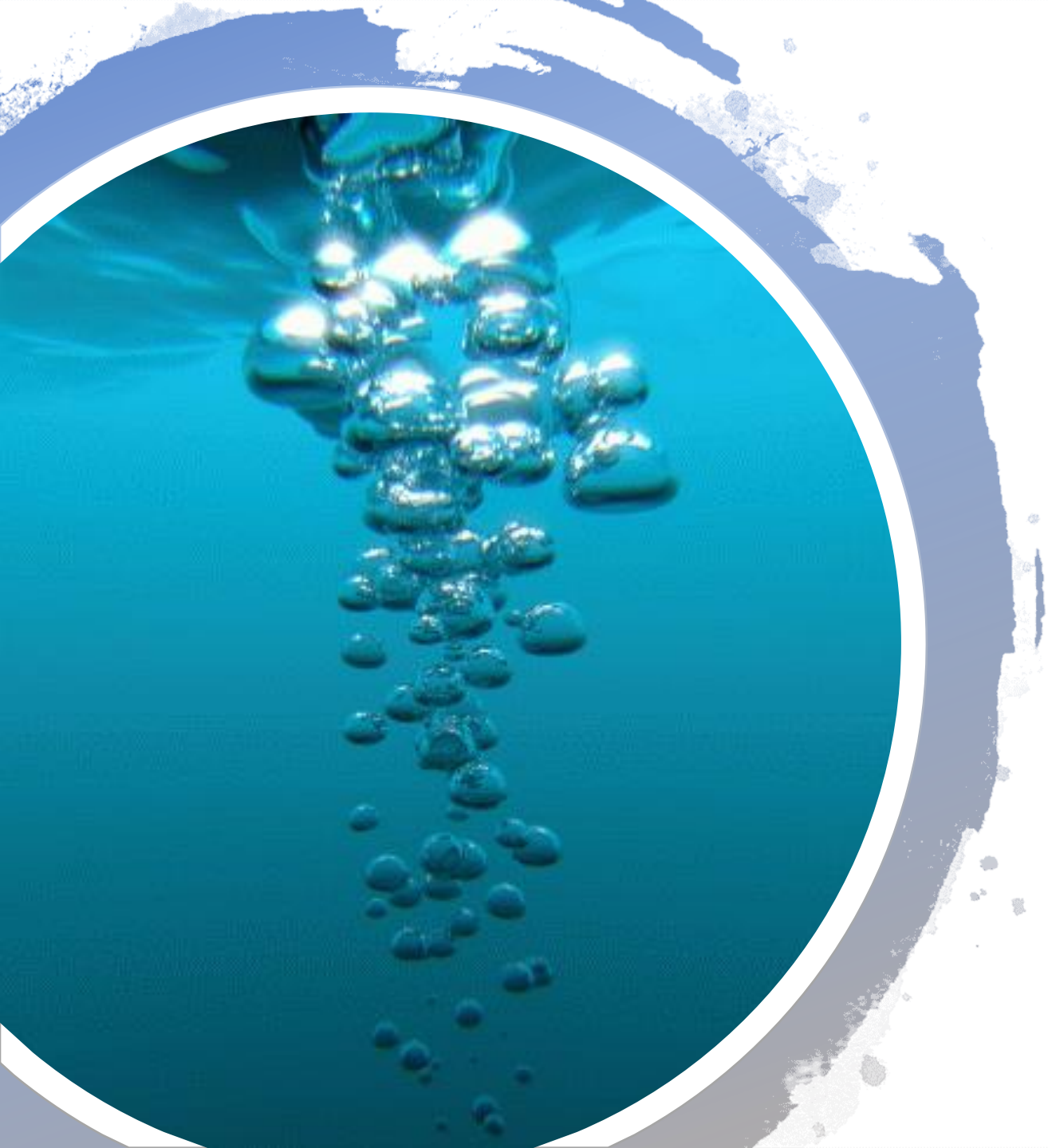
What about using linked list for fast insertion?

Then you cannot do binary search by jumping to the middle …

Any data structure with O(logn) insertion & binary search?

Binary search tree?

Need balance. But itself can sort, no insertion needed. More later.



2. Bubble Sort

Idea: Imagine bubble (largest elem) go up each step by swapping.

```python
def bubble_sort_1(array):          # the actual version with water bubbles
    n = len(array)
    for i in range(n):
        for j in range(n-i-1):  # larger j already sorted (bubble to surface)
            if array[j] > array[j+1]:
                array[j], array[j+1] = array[j+1], array[j]
    return array
```

```python
def bubble_sort_1(array):          # the actual version with water bubbles
    n = len(array)
    for i in range(n):
        for j in range(n-i-1):  # larger j already sorted (bubble to surface)
            if array[j] > array[j+1]:
                array[j], array[j+1] = array[j+1], array[j]
    return array
```

Time complexity?

```python
def bubble_sort_1(array):          # the actual version with water bubbles
    n = len(array)
    for i in range(n):
        for j in range(n-i-1):  # larger j already sorted (bubble to surface)
            if array[j] > array[j+1]:
                array[j], array[j+1] = array[j+1], array[j]
    return array
```

Time complexity: $O(n^2)$ (by default, we mean in the worst case)

```
def bubble_sort_1(array):          # the actual version with water bubbles
    n = len(array)
    for i in range(n):
        for j in range(n-i-1):  # larger j already sorted (bubble to surface)
            if array[j] > array[j+1]:
                array[j], array[j+1] = array[j+1], array[j]
    return array
```

Difference between bubble & insertion sorts?

```python
def bubble_sort_1(array):          # the actual version with water bubbles
    n = len(array)
    for i in range(n):
        for j in range(n-i-1):  # larger j already sorted (bubble to surface)
            if array[j] > array[j+1]:
                array[j], array[j+1] = array[j+1], array[j]
    return array
```

Difference between bubble & insertion sorts?

Insertion: each i-loop: put the current card to the suitable position

Bubble: each i-loop: put the remaining largest element to behind

```python
def bubble_sort_1(array):          # the actual version with water bubbles
    n = len(array)
    for i in range(n):
        for j in range(n-i-1):  # larger j already sorted (bubble to surface)
            if array[j] > array[j+1]:
                array[j], array[j+1] = array[j+1], array[j]
    return array


def bubble_sort_2(array):
    n = len(array)
    while True:
        swapped = False
        for i in range(n-1):
            if array[i] > array[i+1]:
                array[i], array[i+1] = array[i+1], array[i]
                swapped = True
        if swapped == False:
            break
    return array
```

```python
def bubble_sort_1(array):          # the actual version with water bubbles
    n = len(array)
    for i in range(n):
        for j in range(n-i-1):  # larger j already sorted (bubble to surface)
            if array[j] > array[j+1]:
                array[j], array[j+1] = array[j+1], array[j]
    return array


def bubble_sort_2(array):
    n = len(array)
    while True:
        swapped = False
        for i in range(n-1):
            if array[i] > array[i+1]:
                array[i], array[i+1] = array[i+1], array[i]
                swapped = True
        if swapped == False:
            break
    return array
```

Time complexity?

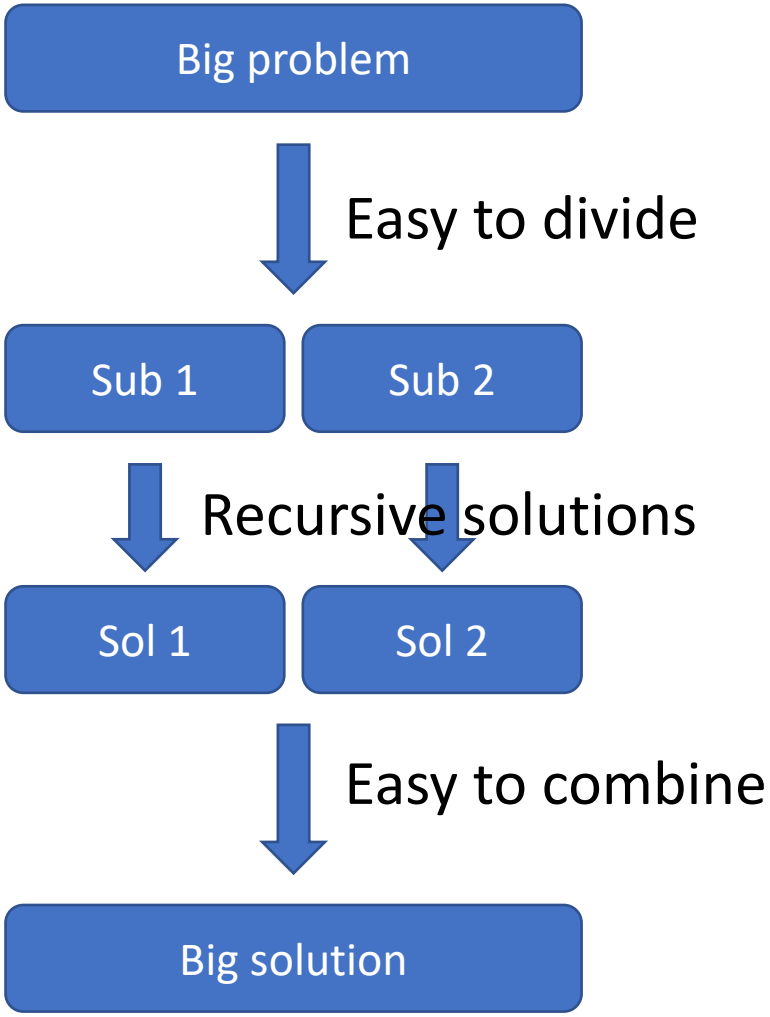
bubble_sort_1 and insertion_sort: always $O(n^2)$

Can we do better?

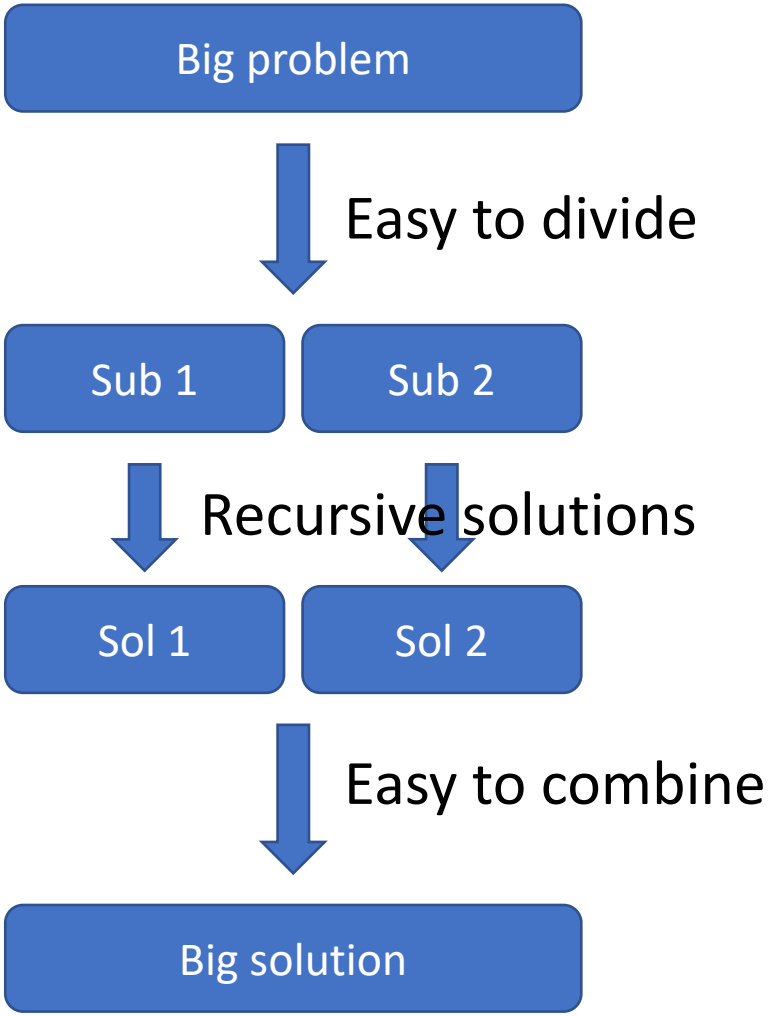
bubble_sort_2: faster if lucky, but on average still $O(n^2)$

Are there faster algorithms, with better average performance?

How to divide and conquer sorting problems?



How to divide and conquer sorting problems?



What's the easiest to divide?
Divide into two directly
→ Merge Sort

What's the easiest to combine?
Left + number + right
→ Quicksort

3. Quicksort

Idea: divide and conquer

3	5	2	1	4

1. Choose pivot, say, 3
2. Partition of list
   - Smaller than pivot: 21
   - Pivot: 3
   - Larger than pivot: 54
3. Quicksort two sublists

```python
def quicksort(array):
    if len(array)<2:
        return array
    pivot = array[0]
    left = [i for i in array[1:] if i <= pivot]
    right = [i for i in array[1:] if i > pivot]
    return quicksort(left) + [pivot] + quicksort(right)
```

Average:

```
def quicksort(array):
    if len(array)<2:
        return array
    pivot = array[0]
    left = [i for i in array[1:] if i <= pivot]
    right = [i for i in array[1:] if i > pivot]
    return quicksort(left) + [pivot] + quicksort(right)
```

Average: $O(n \log n)$ (if not unluckily too bad pivot)

Worst:

```python
def quicksort(array):
    if len(array)<2:
        return array
    pivot = array[0]
    left = [i for i in array[1:] if i <= pivot]
    right = [i for i in array[1:] if i > pivot]
    return quicksort(left) + [pivot] + quicksort(right)
```

Average: $O(n \log n)$ (if not unluckily too bad pivot)

Worst: O(n^2). E.g. [5, 4, 3, 2, 1, 0]

Notes:

1. How to choose pivot?

```python
def quicksort(array):
    if len(array)<2:
        return array
    pivot = array[0]
    left = [i for i in array[1:] if i <= pivot]
    right = [i for i in array[1:] if i > pivot]
    return quicksort(left) + [pivot] + quicksort(right)
```

Average: $O(n \log n)$ (if not unluckily too bad pivot)

Worst: O(n^2). E.g. [5, 4, 3, 2, 1, 0]

Notes:

1. How to choose pivot?

   The first: simplest and fine. But bad if list already ordered

   Median of first, middle, last: a good balance

   Median of list: an additional $O(n)$ search, not used

2. Here not careful about space complexity & constants.

   Otherwise, had better to swap in place instead of new lists

Side remark: How to find median?

- Simplest O(nlogn): sort and take middle
  - Circular – we want to sort now!
- O(n) is possible,
- but non-trivial
- (Blum, Floyd, Pratt, Rivest, Tarjan 1973)

The Python code (from Rcoh):

- just gives you a taste
- not part of this course

```python
def nlogn_median(l):
    l = sorted(l)
    if len(l) % 2 == 1:
        return l[len(l) / 2]
    else:
        return 0.5 * (l[len(l) / 2 - 1] + l[len(l) / 2])
```

```python
def pick_pivot(l):
    """
    Pick a good pivot within l, a list of numbers
    This algorithm runs in O(n) time.
    """
    assert len(l) > 0

    # If there are < 5 items, just return the median
    if len(l) < 5:
        # In this case, we fall back on the first median function we wrote.
        # Since we only run this on a list of 5 or fewer items, it doesn't
        # depend on the length of the input and can be considered constant
        # time.
        return nlogn_median(l)

    # First, we'll split `l` into groups of 5 items. O(n)
    chunks = chunked(l, 5)

    # For simplicity, we can drop any chunks that aren't full. O(n)
    full_chunks = [chunk for chunk in chunks if len(chunk) == 5]


    # Next, we sort each chunk. Each group is a fixed length, so each sort
    # takes constant time. Since we have n/5 chunks, this operation
    # is also O(n)
    sorted_groups = [sorted(chunk) for chunk in full_chunks]

    # The median of each chunk is at index 2
    medians = [chunk[2] for chunk in sorted_groups]

    # It's a bit circular, but I'm about to prove that finding
    # the median of a list can be done in provably O(n).
    # Finding the median of a list of length n/5 is a subproblem of size n/5
    # and this recursive call will be accounted for in our analysis.
    # We pass pick_pivot, our current function, as the pivot builder to
    # quickselect. O(n)
    median_of_medians = quickselect_median(medians, pick_pivot)
    return median_of_medians

def chunked(l, chunk_size):
    """Split list `l` it to chunks of `chunk_size` elements."""
    return [l[i:i + chunk_size] for i in range(0, len(l), chunk_size)]
```

Quicksort in other languages

https://github.com/LingDong-/wenyan-lang

EDITOR

```
吾有一術。名之曰「快排」。欲行是術。必先得一列。曰「甲」。乃行是術曰。
    若「甲」之長弗大於一者。
        乃得「甲」
    也。
    吾有三列。名之曰「首」。曰「頷」。曰「尾」。

    夫「甲」之一。名之曰「甲一」。
    充「頷」以「甲一」。
    夫「甲」之其餘。名之曰「甲餘」。

    凡「甲餘」中之「丁」。
        若「丁」小於「甲一」者。
            充「首」以「丁」。
        若非。
            充「尾」以「丁」
        也。
    云云。
    施「快排」於「首」。昔之「首」者。今其是矣。
    施「快排」於「尾」。昔之「尾」者。今其是矣。

    銜「首」以「頷」以「尾」。名之曰「乙」。
    乃得「乙」。

是謂「快排」之術也。

吾有一列。名之曰「己」。
充「己」以五。以三。以二十。以八。以三十五。以七百。
施「快排」於「己」。書之。
```

OUTPUT

```
3,5,8,20,35,700
```

COMPILED JAVASCRIPT

```
var KUAI4PAI2 = () => 0;
KUAI4PAI2 = function(JIA3) {
    if (JIA3.length <= 1) {
        return JIA3
    };
    var SHOU3 = [];
    var HAN4 = [];
    var WEI3 = [];
    var _ans49 = JIA3[1 - 1];
    var JIA3YI1 = _ans49;
    HAN4.push(JIA3YI1);
    var _ans50 = JIA3.slice(1);
    var JIA3YU2 = _ans50;
    for (var DING1 of JIA3YU2) {
        if (DING1 < JIA3YI1) {
            SHOU3.push(DING1);
        } else {
            WEI3.push(DING1);
        };
    };
    var _ans51 = KUAI4PAI2(SHOU3);
    SHOU3 = _ans51;
    var _ans52 = KUAI4PAI2(WEI3);
    WEI3 = _ans52;
    var _ans53 = SHOU3.concat(HAN4).concat(WEI3);
    var YI3 = _ans53;
    return YI3
};
var JI3 = [];
JI3.push(5, 3, 20, 8, 35, 700);
var _ans54 = KUAI4PAI2(JI3);
console.log(_ans54);
```

Bubble, insertion, quick -- $O(n^2)$

(though quick is usually indeed quick…)

Is there sorting algorithm faster than $O(n^2)$?

Bubble, insertion, quick -- $O(n^2)$

(though quick is usually indeed quick…)

Is there sorting algorithm faster than $O(n^2)$?

One way out:

quicksort with median as pivot (but median is heavy)

Any other ideas?

4. Merge Sort

Idea:

- Cut list into two
- Sort each
- Merge the result

Idea:

- Cut list into two
- Sort each
- Merge the result

Divide & conquer:

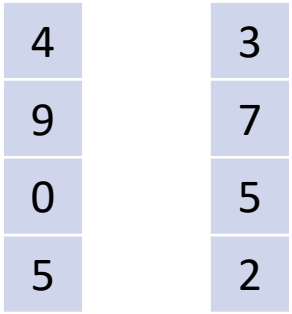
Problem -> subproblem

Idea:

- Cut list into two
- Sort each
- Merge the result

Divide & conquer:

Problem -> subproblem

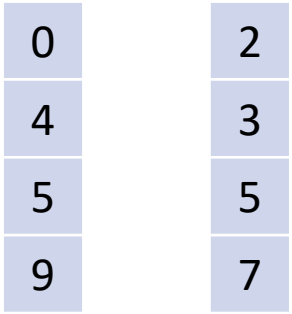


Idea:

- Cut list into two
- Sort each (assume done)
- Merge the result

Divide & conquer:

Problem -> subproblem

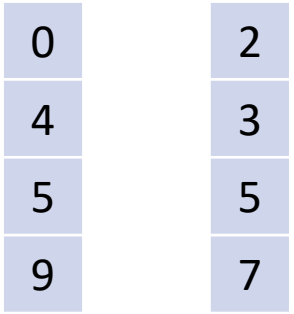


Idea:

- Cut list into two
- Sort each (assume done)
- Merge the result

Divide & conquer:

Problem -> subproblem



Result:

Pick smaller one from top

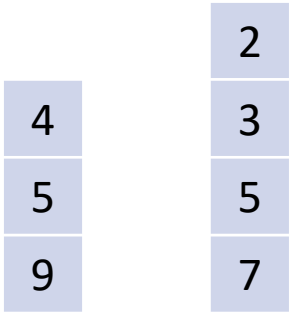
0

Idea:

- Cut list into two
- Sort each (assume done)
- Merge the result

Divide & conquer:

Problem -> subproblem



Result:

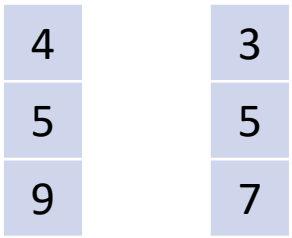
Pick smaller one from top

Idea:

- Cut list into two
- Sort each (assume done)
- Merge the result

Divide & conquer:

Problem -> subproblem



Result:

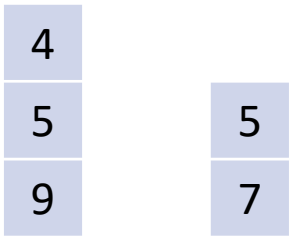
Pick smaller one from top

Idea:

- Cut list into two
- Sort each (assume done)
- Merge the result

Divide & conquer:

Problem -> subproblem



Result:

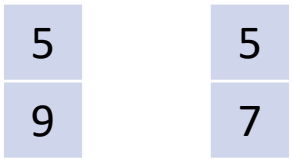
Pick smaller one from top

Idea:

- Cut list into two
- Sort each (assume done)
- Merge the result

Divide & conquer:

Problem -> subproblem



Result:

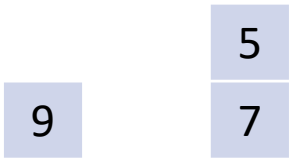
Pick smaller one from top

Idea:

- Cut list into two
- Sort each (assume done)
- Merge the result

Divide & conquer:

Problem -> subproblem



Result:

Pick smaller one from top

Idea:

- Cut list into two
- Sort each (assume done)
- Merge the result

Divide & conquer:

Problem -> subproblem

Result:

Pick smaller one from top

Idea:

- Cut list into two
- Sort each (assume done)
- Merge the result

Divide & conquer:

Problem -> subproblem

9

0
2
3
4
5
5
7

Result:

Pick smaller one from top

Idea:

- Cut list into two
- Sort each (assume done)
- Merge the result

Divide & conquer:

Problem -> subproblem

Result:

Pick smaller one from top

Magic of recursion
For subproblem with length 0 or 1, indeed done :)

Idea:

- Cut list into two
- Sort each (assume done)
- Merge the result

Divide & conquer:

Problem -> subproblem

Result:

Pick smaller one from top

```python
def merge_sort(array):
    if len(array) <= 1:
        return array
    mid = (len(array) + 1) // 2
    sub1 = merge_sort(array[: mid])
    sub2 = merge_sort(array[mid :])
    return ordered_merge(sub1, sub2)

def ordered_merge(a1, a2):
    cnt1, cnt2 = 0, 0
    result = []
    while cnt1 < len(a1) and cnt2 < len(a2):
        if a1[cnt1] < a2[cnt2]:
            result.append(a1[cnt1])
            cnt1 += 1
        else:
            result.append(a2[cnt2])
            cnt2 += 1
    result += a1[cnt1 :]
    result += a2[cnt2 :]
    return result
```

“//” is integer part of division, not comment

Time complexity?

```python
def merge_sort(array):
    if len(array) <= 1:
        return array
    mid = (len(array) + 1) // 2
    sub1 = merge_sort(array[: mid])
    sub2 = merge_sort(array[mid :])
    return ordered_merge(sub1, sub2)

def ordered_merge(a1, a2):
    cnt1, cnt2 = 0, 0
    result = []
    while cnt1 < len(a1) and cnt2 < len(a2):
        if a1[cnt1] < a2[cnt2]:
            result.append(a1[cnt1])
            cnt1 += 1
        else:
            result.append(a2[cnt2])
            cnt2 += 1
    result += a1[cnt1 :]
    result += a2[cnt2 :]
    return result
```

Each cut:

- Cut list to half
- Do $\leq O(n)$ operations

Total: O(n logn)

But on average, not as fast as quick sort. Thus quick sort is very widely used.

5. Heap

and Heap Sort

For the past 33 years, I have looked in the mirror every morning and asked myself: ‘If today were the last day of my life, would I want to do what I am about to do today?’ And whenever the answer has been ‘No’ for too many days in a row, I know I need to change something.

– Steve Jobs

How to do it efficiently?

How to design the next thing to do in your life?

Do you need to sort everything and pick the first?

O(n logn)

How to design the next thing to do in your life?

Do you need to sort everything and pick the first?

O(n logn)

Do you need to find the maximal in O(n)?

How to design the next thing to do in your life?

Do you need to sort everything and pick the first?

O(n logn)

Do you need to find the maximal in O(n)?

Not necessarily. You just need to know what's next.

Priority queue: whatever order in, most important out first.

How to design the next thing to do in your life?

Do you need to sort everything and pick the first?

O(n logn)

Do you need to find the maximal in O(n)?

Not necessarily. You just need to know what's next.

Priority queue: whatever order in, most important out first.

What's an efficient way to organize a priority queue?

Can be done in O(logn) using heap (as will be shown).

What's a heap?

(Here max heap)

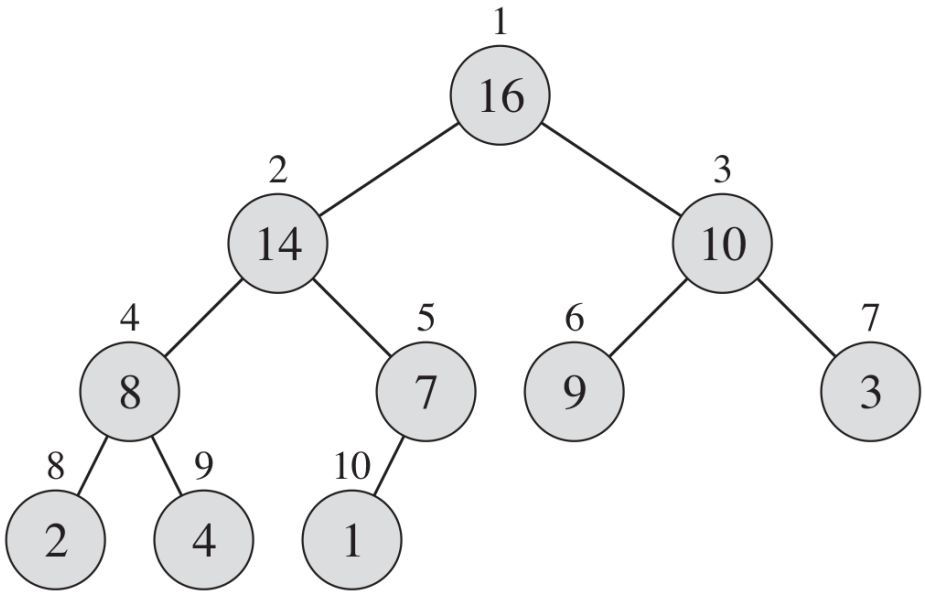
A list, visualized as a tree,

each node is larger than all its children.

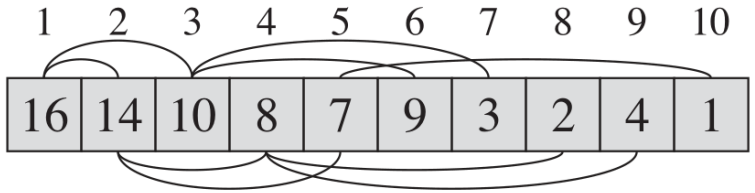
A realization of priority queue

Given set of numbers, heap is not unique (e.g. swap 3 and 1 below).

We just need one.



(a)

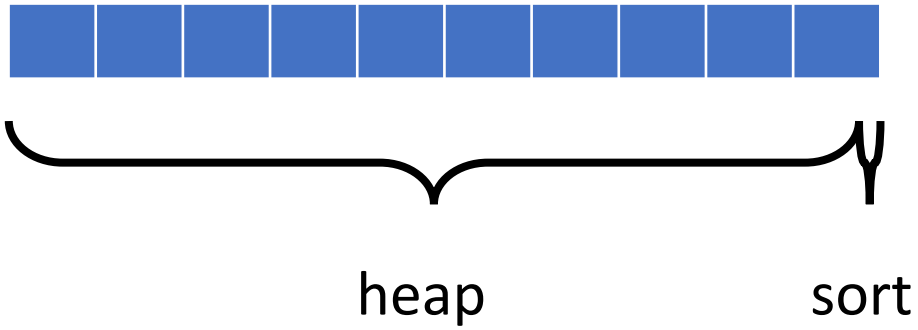


(b)

Heap sort:

1. Construct heap
2. Swap heap[0] and heap[-1]
3. Make the new object: heap[: -1] a heap
4. Go to 2 until len(heap)==1

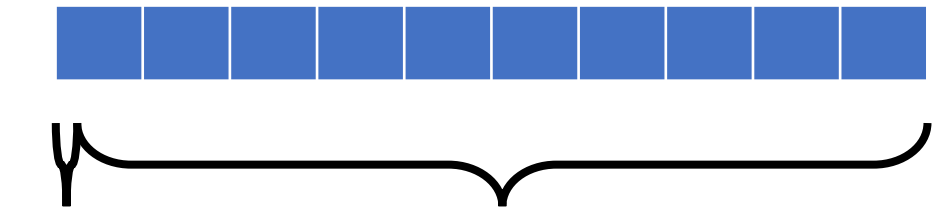
i.e. heap sort = heap & sort in a list



Heap sort:

1. Construct heap
2. Swap heap[0] and heap[-1]
3. Make the new object: heap[: -1] a heap
4. Go to 2 until len(heap)==1

i.e. heap sort = heap & sort in a list



heap sort

Heap sort:

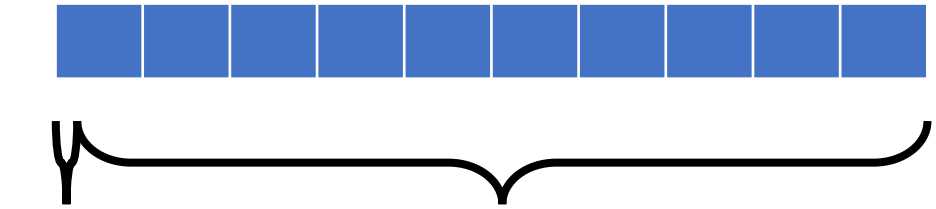
How?

1. Construct heap
2. Swap heap[0] and heap[-1]

How?

3. Make the new object: heap[: -1] a heap
4. Go to 2 until len(heap)==1

i.e. heap sort = heap & sort in a list



heap sort

Fundamental operation: max_heapify(heap)

Assuming

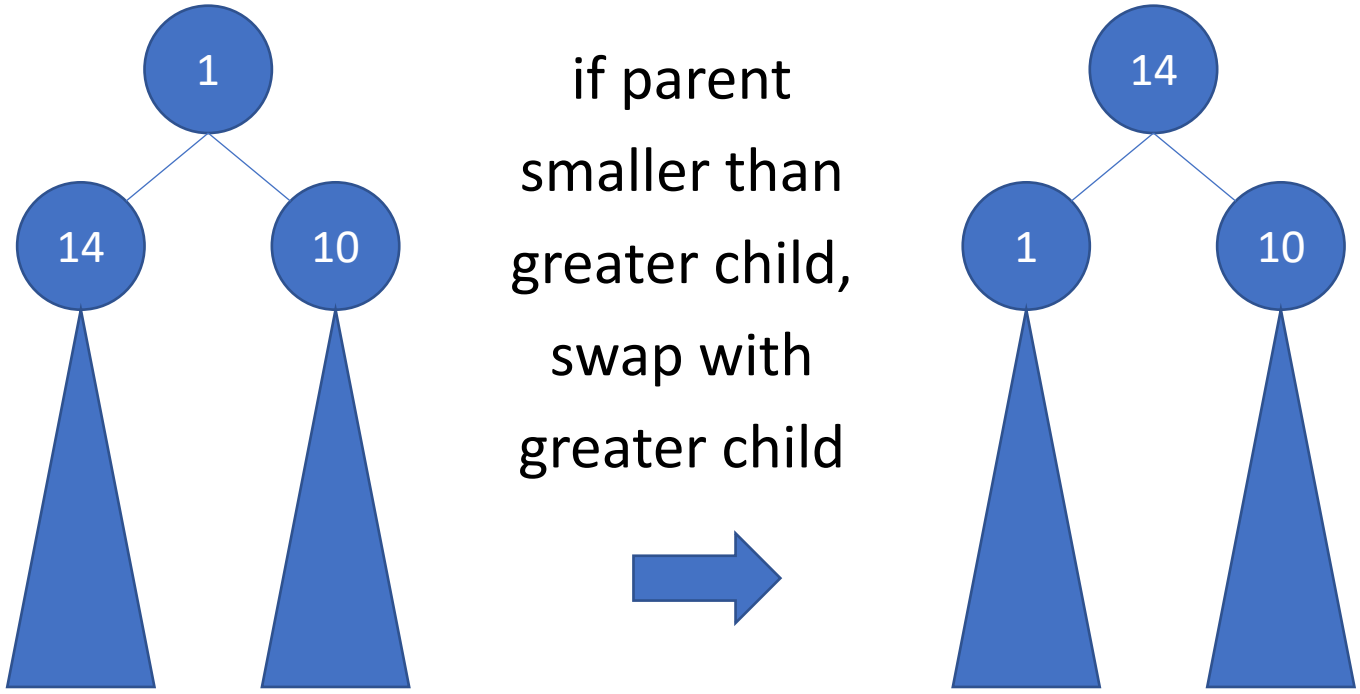
- Only heap[0] is wrong (not the largest).
- All other element satisfy heap conditions.

How to correct this heap[0] mistake?

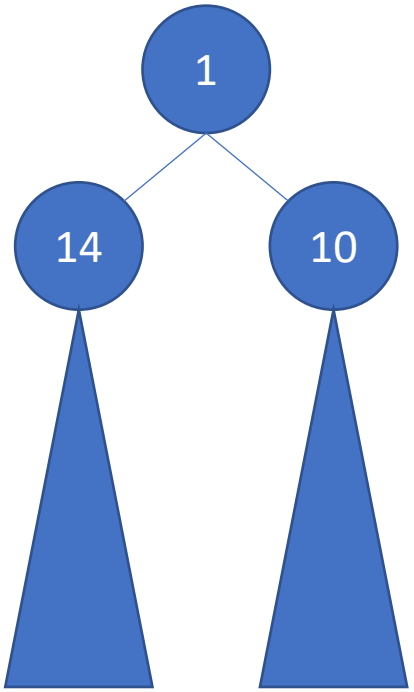
If this is known:

- Can create heap recursively (or in-place, see later).
- Can “make the new object a heap” in last page

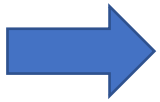
max_heapify

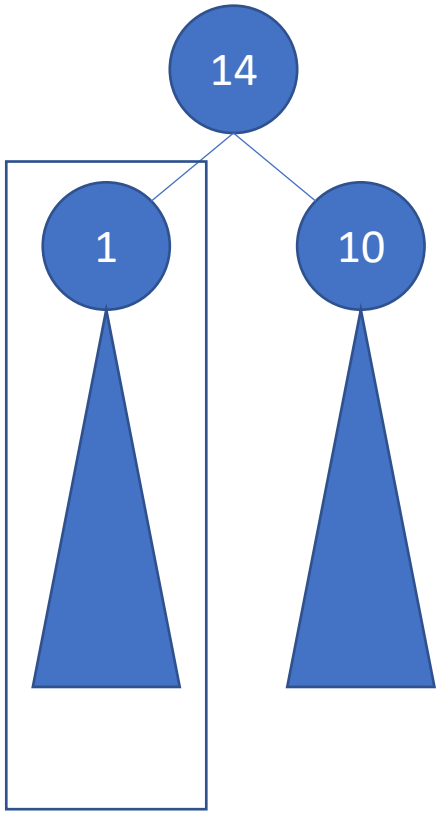

1
14
10
if parent
smaller than
greater child,
swap with
greater child
14
1
10

max_heapify



if parent smaller than greater child, swap with greater child





Solve subproblem, until len(heap) == 1

max_heapify

Time complexity:

Reduce to subproblem: O(1)

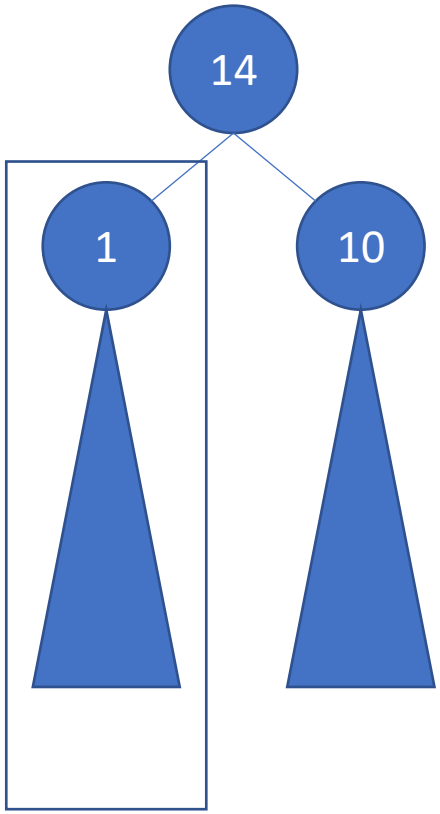
Number of subproblems: O(logn)

Thus, max_heapify takes O(logn)

Build max heap: O(n)

Note: appears O(n logn); Actually better.

Heap sort: O(n logn)



Solve subproblem,

until len(heap) == 1

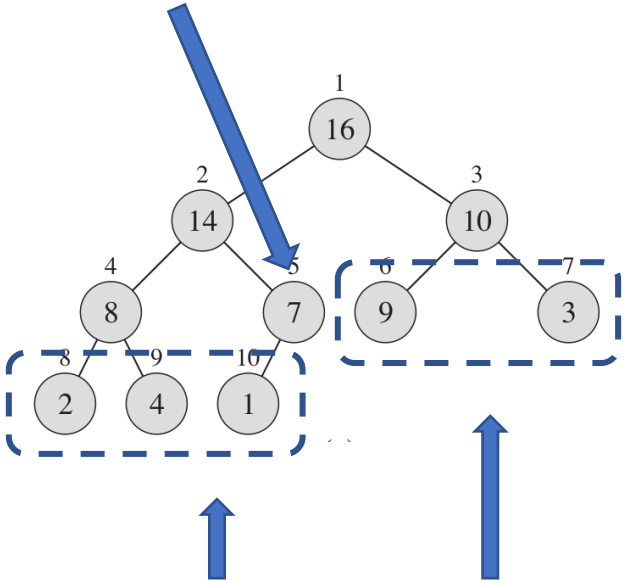
Code: definition & in-place heapify

```
class Heap():
    def __init__(self, array):
        self.heap = array
        self.length = len(array)
        self.build_max_heap()

    def left(self, idx):
        pos = 2 * idx + 1
        return pos if pos < self.length else None

    def right(self, idx):
        pos = 2 * idx + 2
        return pos if pos < self.length else None

    def parent(self, idx):
        return (idx - 1) // 2 if idx > 0 else None

    def build_max_heap(self):
        last_to_heapify = self.parent(self.length - 1)
        # lower limit of loop is 0
        for i in range(last_to_heapify, -1, -1):
            self.max_heapify(i)
```

Only initially. Later we will modify self.length

last_to_heapify:

Parent of last element



no need to heapify

Code: max_heapify and sort

```
def _greater_child(self, i):
    left, right = self.left(i), self.right(i)
    if left is None and right is None:
        return None
    elif left is None:
        return right
    elif right is None:
        return left
    else:
        return left if self.heap[left]>self.heap[right] else right

def max_heapify(self, i):
    greater_child = self._greater_child(i)
    if greater_child is not None and self.heap[greater_child] > self.heap[i]:
        self.heap[i], self.heap[greater_child] = self.heap[greater_child], self.heap[i]
        self.max_heapify(greater_child)

def sort(self):
    while self.length > 1:
        self.heap[0], self.heap[self.length - 1] = self.heap[self.length - 1], self.heap[0]
        self.length -= 1
        self.max_heapify(0)
    return self.heap
```

Insertion & construct heap by insertion

Step 1: insert the new element at the bottom

Step 2: heapify the element from the bottom-up
(swap if greater than parent, till root)

Question: what's *average* insertion for heap?

Appears log(n), since the height of tree is log(n). However, note that

- Heap insertion happens bottom-up
- The last two levels contains more than half elements

Thus, for most elements, insertion takes O(1) only.

Surely, worst insertion is still log(n).

What's the time complexity for building a heap?

O(n) since each element has average O(1). Proof: (leaf has h=0 below)

$$\sum_{h=0}^{\log n} (\#\text{nodes at level h}) \times (\text{heapify time at h}) \sim \sum_{h=0}^{\log n} \frac{n}{2^{h+1}} \times (h) \sim O(n)$$

Merge, heap & AVL are sorting algorithms that sort in O(n logn)

Is it possible to sort in O(n)?

知乎 首页 发现 等你来答 梁朝伟加盟漫威新作《上气》

程序员 算法 编程 计算机 代码 修改

大家都见过哪些让你虎躯一震的代码? 修改

就是瞬间让你怀疑人生的那种

修改

关注问题 写回答 邀请回答 5 条评论 分享 举报 ...

查看全部 587 个回答

柳五
冷知识收集癖

2,510 人赞同了该回答

第一次看到睡眠排序算法，被惊呆了，半天都没缓过来。同样是九年义务教育，为何别人如此优秀。

> 对于数组中的每个数，都对应一个会睡眠这么长时间的线程。例如对于[2,4,5,2,1,7]这样一组数，就创建6个线程，分别睡眠2ms，4ms，5ms，2ms，1ms, 7ms。这些线程睡醒之后，就把自己对应的数报出来即可。

```
import thread
from time import sleep
items = [2, 4, 5, 2, 1, 7]
def sleep_sort(i):
        sleep(i*0.001)
        print i
[thread.start_new_thread(sleep_sort, (i,)) for i in items]
```

Sleep sort.

Each sleep that # seconds
before printing.

Discussion:

Is it really O(n)?

Spaghetti sort

Make length of noodle = number

Make upper end horizontal

Drop, see which noodle first touch the table

Is it really O(1) in your hands/eyes?

Can computer simulate it in O(1)?

* Theorem: Search by comparing at least need $\Theta(\log n)$ time

Sort - - - - - - - - $O(n \log n)$ - - -

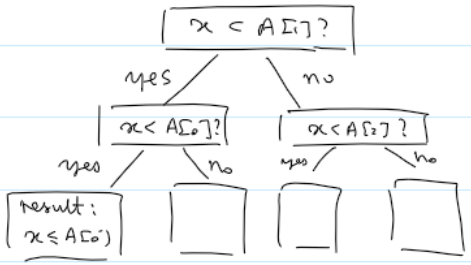
— Input items are abstract data types (black boxes), only allow >><<== operations

— time cost $\geq$ # comparisons

- For comparison based search: E.g. binary search for x in $A[0]\ [1]\ [2]$

Decision tree [DT]

[not tree of data, but tree of procedure of algorithm]



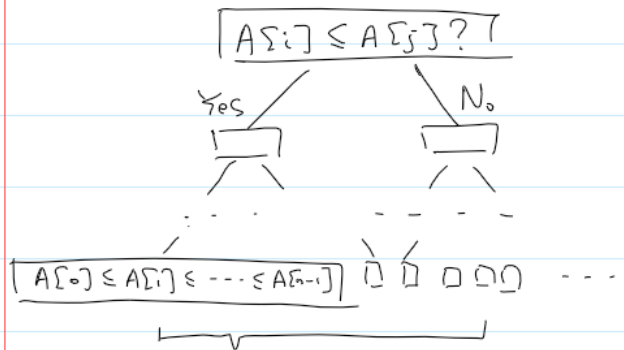
time complexity

DT: Contains all possible answers

$\Rightarrow$ at least n - leaves (return values) Could be more, since multiple leaves can have same return values

$\Rightarrow$ height $\geq \log n$

- For comparison based sorting:



$n!$ leaves (# permutations).

Height $\geq \log(n!) \sim n \log n$ [Stirling's approximation]

8. Counting Sort

Sort n small integers (assuming non-negative).

Or other discrete objects (via dict but need to sort keys)

Idea: vote.

Range of numbers in array == candidate

Array == votes collected

Create an array

0, (candidate 0)

1, (candidate 1)

…,

max_elem

```python
def counting_sort(array):
    max_elem = max(array)
    counts = [0 for i in range(max_elem + 1)]
    for elem in array:
        counts[elem] += 1
    return [i for i in range(len(counts)) for cnt in range(counts[i])]
```

Example: [2,4,3,0,2,4,6,3,2,3,2,1,2,1,0]

candidate id	votes
0	2
1	2
2	5
3	3
4	2
5	0
6	1

candidate[2] += 1

candidate[4] += 1

candidate[3] += 1

candidate[0] += 1

… …

Print (candidate id) for (votes) number of times

[0, 0, 1, 1, 2, 2, 2, 2, 2, 3, 3, 3, 4, 4, 6]

Counting Sort by Given Rank

```
def counting_sort_by(array, max_rank = None, rank = lambda x: x):
    "Counting sort wrt its non-negative integer-valued rank(elem)."
    if max_rank is None:
        max_rank = 0
        for elem in array:
            if rank(elem) > max_rank:
                max_rank = rank(elem)
    # One cannot do counts = [[]] * (max_rank + 1), otherwise all [] have the same reference
    counts = [[] for cnt in range(max_rank+1)]
    for elem in array:
        (counts[rank(elem)]).append(elem)
    return [elem for sublist in counts for elem in sublist]
```

What if a few very large numbers are given?

Say 1073741824 (2^30 =Giga)

Counting sorts will need O(2^30) time & space … bad.

Is there an algorithm which fits for exponentially large numbers?

What about large integers?

(And machine precision numbers)

9. Radix Sort

Idea: sort each digit by counting sort.

Two possible algorithms:

- Start from least significant digit (LSD)
  - Each digit sort once then done.
- Start from most significant digit (MSD)
  - Create an array for numbers with same MSD

```python
def integer_digits(num):
    "Example: integer_digits(3142) == [3,1,4,2]"
    digits = []
    while num > 0:
        digits.append(num % 10)
        num = num // 10
    return digits[::-1]

def from_digits(digits):
    num = 0
    for d in digits:
        num = num * 10
        num += d
    return num

def dig(rd, d):
    "Example: dig([3,1,4,2],0) == 2; dig([3,1,4,2],1) == 4;  dig([3,1,4,2],4) == 0"
    return 0 if d >= len(rd) else rd[-(d+1)]

def radix_LSD_sort(array):
    rd_array = []
    max_n_digits = 0
    for num in array:
        bursted = integer_digits(num)
        rd_array.append(bursted)
        if max_n_digits < len(bursted):
            max_n_digits = len(bursted)
    for d in range(max_n_digits):
        rd_array = counting_sort_by(rd_array, 9, lambda rd: dig(rd, d))
    return [from_digits(digits) for digits in rd_array]
```

Exercise: find the kth (counting from 0) smallest element from a list.

Examples:

[0, 1, 2, 3], k = 0: Return 0.

[4, 2, 1, 3], k = 1: Return 2.

Solution: Sort and take the kth element, O(nlgn).

Any faster possibilities?

Quickselect: like quicksort but not sorting.

```
def quickselect(array, k):
    pivot = array[0]
    left = [i for i in array[1:] if i <= pivot]
    right = [i for i in array[1:] if i > pivot]
    if k > len(left):
        return quickselect(right, k - len(left) - 1)
    elif k < len(left):
        return quickselect(left, k)
    else:
        return pivot
```

Average: O(n). However, worst O(n^2).

Improvement: find a better pivot, then worst O(n) is possible.

Exercise: merge n sorted lists to give one sorted list.

Recap:

Sorting algorithms:

- Insertion
- Bubble
- Quick
- Merge
- Heap
- Counting
- Radix

Array Sorting Algorithms

Algorithm	Time Complexity			Space Complexity
	Best	Average	Worst	Worst
Quicksort	Ω(n log(n))	Θ(n log(n))	O(n^2)	O(log(n))
Mergesort	Ω(n log(n))	Θ(n log(n))	O(n log(n))	O(n)
Timsort	Ω(n)	Θ(n log(n))	O(n log(n))	O(n)
Heapsort	Ω(n log(n))	Θ(n log(n))	O(n log(n))	O(1)
Bubble Sort	Ω(n)	Θ(n^2)	O(n^2)	O(1)
Insertion Sort	Ω(n)	Θ(n^2)	O(n^2)	O(1)
Selection Sort	Ω(n^2)	Θ(n^2)	O(n^2)	O(1)
Tree Sort	Ω(n log(n))	Θ(n log(n))	O(n^2)	O(n)
Shell Sort	Ω(n log(n))	Θ(n(log(n))^2)	O(n(log(n))^2)	O(1)
Bucket Sort	Ω(n+k)	Θ(n+k)	O(n^2)	O(n)
Radix Sort	Ω(nk)	Θ(nk)	O(nk)	O(n+k)
Counting Sort	Ω(n+k)	Θ(n+k)	O(n+k)	O(k)
Cubesort	Ω(n)	Θ(n log(n))	O(n log(n))	O(n)

(More later: BST, AVL, RBT)