Algorithms

授課老師：張景堯

Priority Queue/Heapsort

Chapter.4-Sorting and Searching

Think about Selection Sort

- Selection sort scans through the entire array, repeatedly finding the smallest remaining element.
- For $i = 1$ to n
  - A: Find the smallest of the first $n - i + 1$ items.
  - B: Pull it out of the array and put it first.
- Selection sort takes $O(n(T(A) + T(B))$ time.

The Data Structure Matters

- Using arrays or unsorted linked lists as the data structure, operation A takes $O(n)$ time and operation B takes $O(1)$, for an $O(n^2)$ selection sort.
(找最小的花$O(n)$；把那最小的放到前面則$O(1)$)
- Using balanced search trees or heaps, both of these operations can be done within $O(\log n)$ time, for an $O(n \log n)$ selection sort called heapsort.
(平衡樹找最小的跟把那最小拿走再調整樹都是$O(\log n)$)
- Balancing the work between the operations achieves a better tradeoff.
- Key question: “Can we use a different data structure?”

Priority Queues(優先佇列)

- Priority queues are data structures which provide extra flexibility over sorting.
- This is important because jobs often enter a system at arbitrary intervals. It is more cost-effective to insert a new job into a priority queue than to re-sort everything on each new arrival.
(作業系統常用優先佇列，因為工作會不時進到系統等待執行，有些工作優先序高有些低，這時就直接放入優先佇列中就好而不必每次都要將佇列所有工作重新排序)

Priority Queue Operations

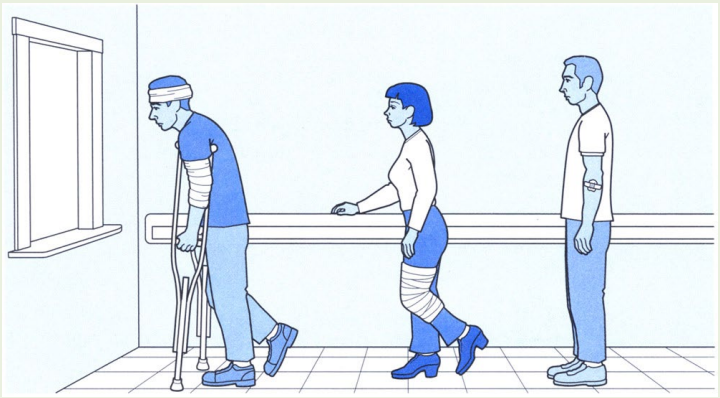
- The basic priority queue supports three primary operations:
  - $Insert(Q, x)$: Given an item x with key k, insert it into the priority queue Q.
  - $Find\_Minimum(Q)$ or $Find\_Maximum(Q)$: Return a pointer to the item whose key is smaller (larger) than any other key in the priority queue Q.
  - $Delete\_Minimum(Q)$ or $Delete\_Maximum(Q)$: Remove the item from the priority queue Q whose key is minimum (maximum).
- Each of these operations can be easily supported using heaps or balanced binary trees in $O(\log n)$.

Applications of Priority Queues: Dating

- What data structure should be used to suggest who to ask out next for a date? (怎樣維護一個約會佇列)
- It needs to support retrieval by *desirability*, not name.
- Desirability changes (up or down), so you can re-insert the max with the new score after each date.
- New people you meet get inserted with your observed desirability level.
- There is no reason to delete anyone until they rise to the top.

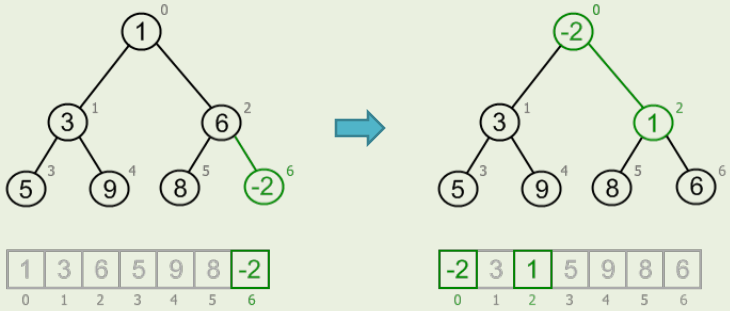
Applications of Priority Queues: Discrete Event Simulations

- In simulations of airports, parking lots, and triage in a hospital emergency room – priority queues can be used to maintain who goes next.
- The stack and queue orders are just special cases of orderings. In real life, certain people cut in line, and this can be modeled with a priority queue.



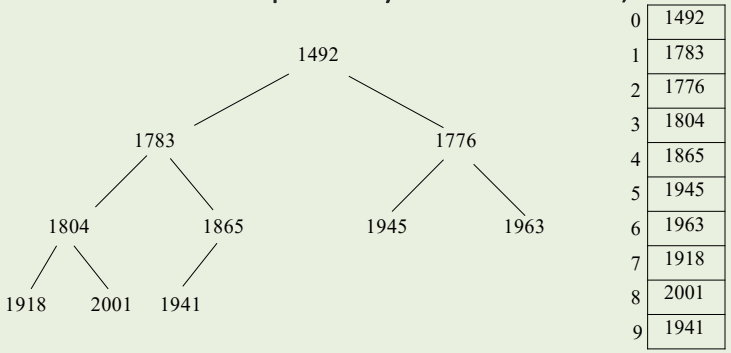
Heap Definition(堆)

- A *binary heap* is defined to be a binary tree with a key in each node such that:
  1. All leaves are on, at most, two adjacent levels.
  2. All leaves on the lowest level occur to the left, and all levels except the lowest one are completely filled. (總是一棵完全樹)
  3. The key in root is ≤ all its children, and the left and right subtrees are again binary heaps.(任意節點小於它的所有後裔)
- Conditions 1 and 2 specify shape of the tree, and condition 3 the labeling of the tree.



Binary Heaps(二元堆積)

- Heaps maintain a partial order on the set of elements which is weaker than the sorted order (so it can be efficient to maintain) yet stronger than random order (so the minimum element can be quickly identified).



- A heap-labeled tree of important years (l), with the corresponding implicit heap representation (r)

Heapsort Animation

- Watch as
  - We build the heap by repeated insertion.
  - Embed it in an array.
  - Then repeatedly remove the maximum to sort:
- https://upload.wikimedia.org/wikipedia/commons/4/4d/Heapsort-example.gif
- https://www.youtube.com/watch?v=Xw2D9aJRBY4
- The partial order defined by the heap structure is weaker than sorting, which explains why it is easier to build: linear time if you do it right.

Array-Based Heaps

- The most natural representation of this binary tree would involve storing each key in a node with pointers to its two children.
- However, we can store a tree as an array of keys, using the position of the keys to *implicitly* satisfy the role of the pointers.
- The *left* child of k sits in position $2k + 1$ and the *right* child in $2k + 2$. (0->1,2; 1->3,4; 2->5,6)
- The parent of k is in position $\lfloor (k-1)/2 \rfloor$. ($\left\lfloor \frac{4-1}{2} \right\rfloor$->1)

(Heap這類完全二元樹非常適合用Array的連續空間來存放)

Can we Implicitly Represent Any Binary Tree?

- Suppose our height h tree was *sparse*, meaning that the number of nodes $n < 2^h$.
- All missing internal nodes still take up space in our structure. This is why we insist on heaps as being as balanced/full at each level as possible.
- The array-based representation is also not as flexible to arbitrary modifications as a pointer-based tree.

(但以陣列表示二元樹看似簡單省空間，其實並不盡然且彈性不足，當要移動一個子樹時，得執行很多步搬移，反而沒指標或參照改一下方便。)

Constructing Heaps

- Heaps can be constructed incrementally, by inserting new elements into the $(n+1)$st position in the n-element array.
- If the new element is smaller than its parent, swap their positions and recur. (最小堆積)
- Since all but the last level is always filled, the height h of an n element heap is bounded because:

$$\sum_{i=1}^{h} 2^i = 2^{h+1} - 1 \geq n$$

- so $h = \lfloor \log n \rfloor$.
- Doing n such insertions really takes $\Theta(n \log n)$, because the last $n/2$ insertions require $O(\log n)$ time each.

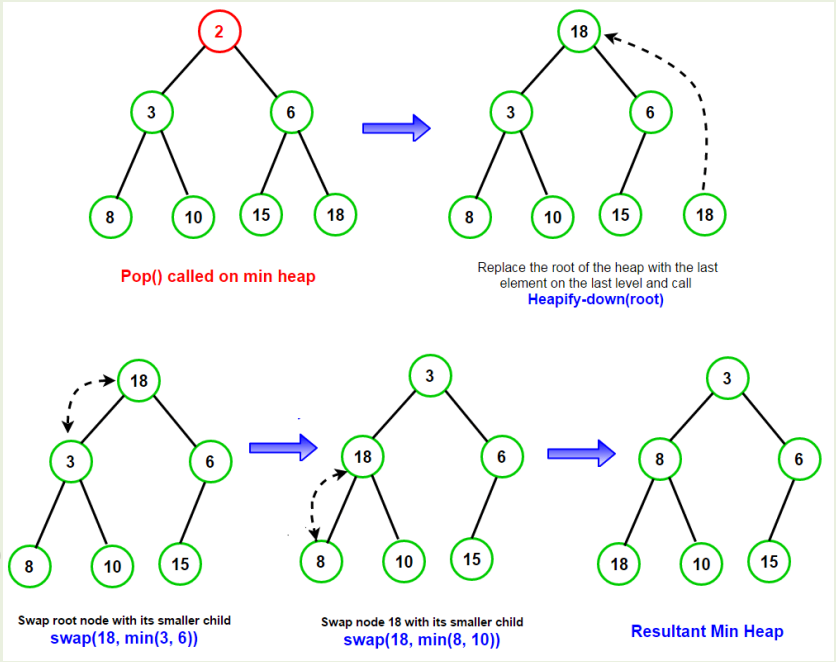
Heap Insertion & Bubble Up

```
def pq_insert(q:list, x:int):
    q.append(x)
    bubble_up(q, len(q)-1)

def bubble_up(q:list, p:int):
    if p <= 0:
        return
    else:
        parent = (p-1)//2
        if (q[parent] > q[p]):
            q[parent], q[p] = q[p], q[parent]
            bubble_up(q,parent)
```

An Even Faster Way to Build a Heap

- Given two heaps and a fresh element, they can be merged into one by making the new one the root and bubbling down (*heapify*).
- Build-heap(A)
  - $n = |A| - 1$
  - For $i = \lfloor n/2 \rfloor$ to 0 do Heapify(A, i)

(建立Heap另一方法，不是一個個加在最後面再調整，而是先把資料都放入，從中間開始往前逐個執行bubble-down)



Bubble Down Implementation

```
def bubble_down(q:list, p:int):
    c = p * 2 + 1 #child index
    min_index = p #index of lightest child
    for i in range(c,c+2):
        if i < len(q):
            if q[min_index] > q[i]:
                min_index = i
    if min_index != p:
        q[min_index], q[p] = q[p], q[min_index]
        bubble_down(q, min_index)
```

Heapsort

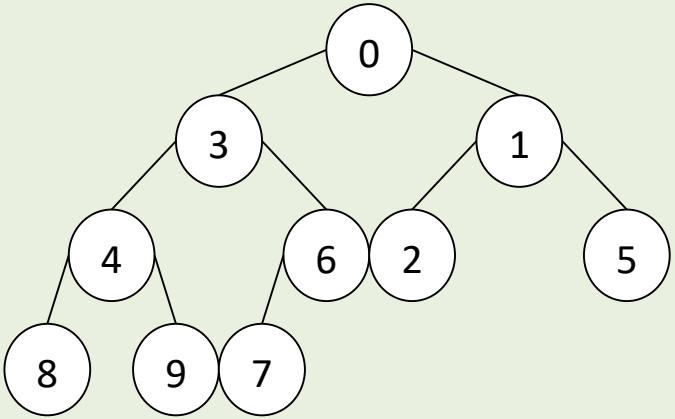
- Heapify can be used to construct a heap, using the observation that an isolated element forms a heap of size 1.
- Heapsort(A)
  - Build-heap(A)
  - for $i = n$ to 0 do
    swap(A[0],A[i])
    $n = n - 1$
    Heapify(A,0)
- Exchanging the maximum element with the last element and calling heapify repeatedly gives an $O(n \log n)$ sorting algorithm.

```
def heapsort(nums:list):
    heap = []
    for i in nums:
        pq_insert(heap,i)
    while len(heap) > 1:
        print(heap[0], end=' ')
        heap[0]=heap.pop()
        bubble_down(heap, 0)
    else:
        print(heap[0])
```

[補充] Example of Heapsort

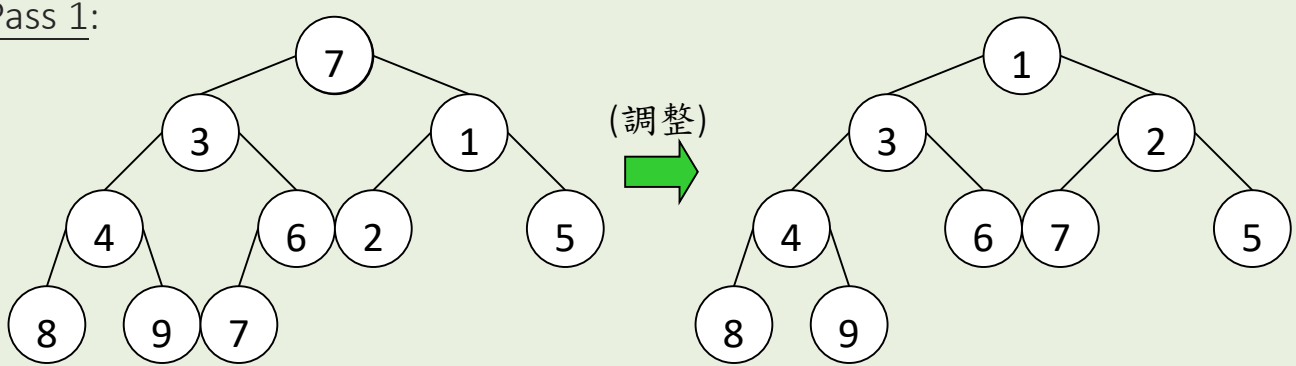
- Step:
  - ① 將資料先以"pq_insert/bubble_up" 的方式建立Min-Heap
  - ② 執行 n-1 回合的"delete min/bubble_down" 動作
- 給予 3, 0, 1, 8, 7, 2, 5, 4, 9, 6，寫出Heap Sort的過程。

Sol: ①先以"bubble_up"的方式建立Min-Heap

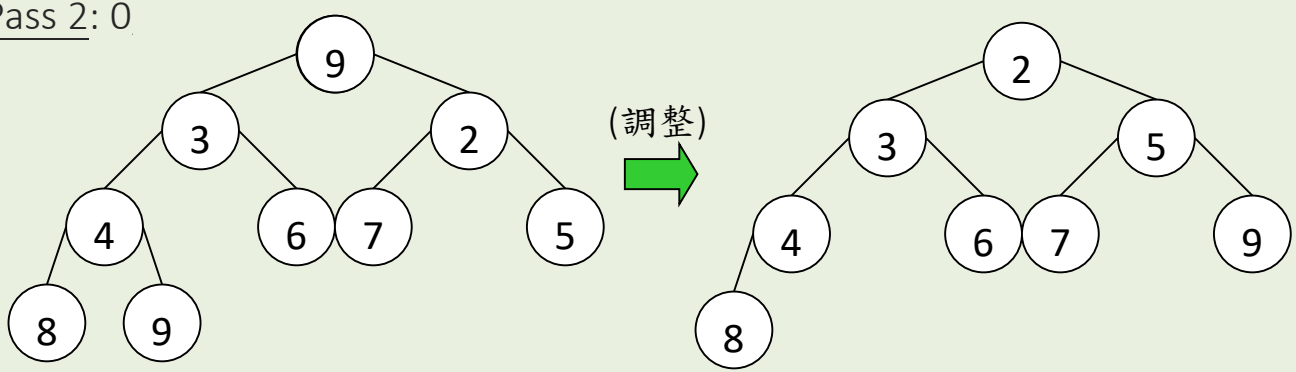


② 執行 n-1 回合的"delete min/bubble_down" 動作

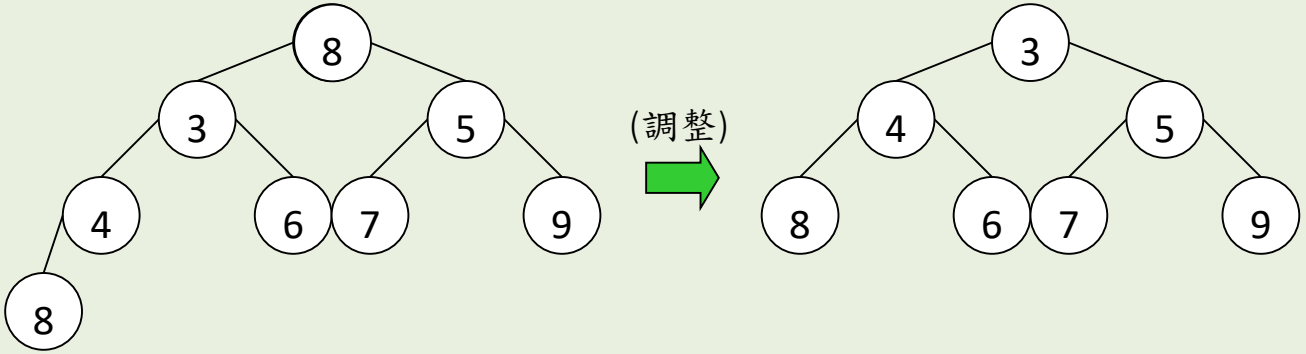
Pass 1:



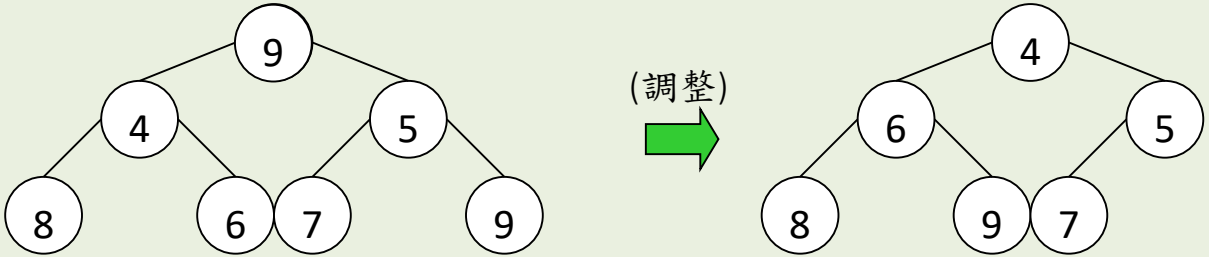
Pass 2: 0



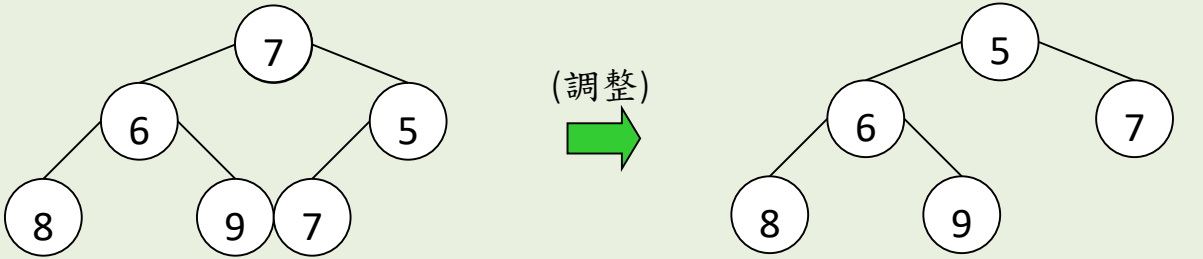
Pass 3: 0, 1



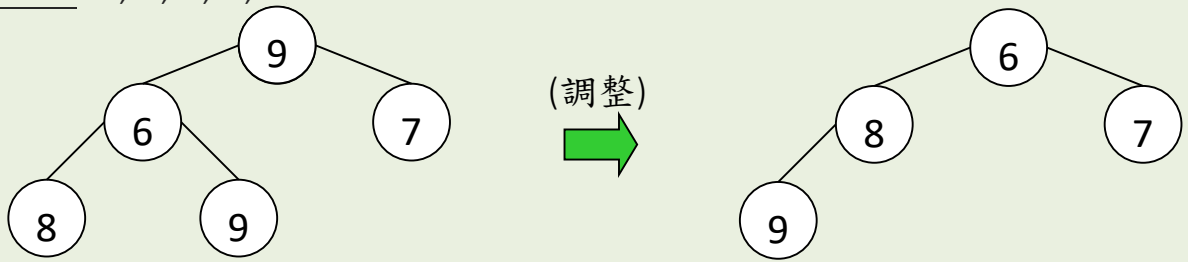
Pass 4: 0, 1, 2



Pass 5: 0, 1, 2, 3



Pass 6: 0, 1, 2, 3, 4



Pass 7: 0, 1, 2, 3, 4, 5

Pass 8: 0, 1, 2, 3, 4, 5, 6

Pass 9: 0, 1, 2, 3, 4, 5, 6, 7

Pass 10: 0, 1, 2, 3, 4, 5, 6, 7, 8

[補充] Heap-based Priority Queues in Python with **heapq**

- The **heapq** module in Python provides an implementation of the heap queue algorithm.
- It allows you to efficiently manage a priority queue, where elements are retrieved based on their priority (smallest element first for a min-heap).
- Key functions:
  - **heappush**(heap, item) - Push elements onto the heap.
  - **heappop**(heap) - Remove & output the smallest element
  - **heapify**(list) - Heapify an existing list
  - **nlargest**(n, iterable) - Get n largest element
  - **nsmallest**(n, iterable) - Get n smallest element
- heapq uses a list as the underlying data structure.

Core `heapq` Operations

```python
import heapq

heap = [] # Create an empty heap

# Push elements onto the heap
heapq.heappush(heap, 3)
heapq.heappush(heap, 1)
heapq.heappush(heap, 4)
heapq.heappush(heap, 2)

# Pop the smallest element
smallest = heapq.heappop(heap)
print(f"Smallest element: {smallest}")
# Output: Smallest element: 1
```

```python
# Heapify an existing list
data = [5, 2, 9, 1, 5, 6]
heapq.heapify(data)
print(f"Heapified list: {data}")
# Out: Heapified list: [1, 2, 6, 5, 5, 9]

# Get n largest element
largest = heapq.nlargest(2, data)
print(f"Two largest number: {largest}")
# Output: Two largest number: [9, 6]

# Get n smallest element
smallest = heapq.nsmallest(2, data)
print(f"Two smallest number: {smallest}")
# Output: Two smallest number: [1, 2]
```

Merging Sorted Iterables with **heapq.merge**

- **heapq.merge**(*iterables, key=None, reverse=False)- efficiently merges multiple sorted iterables into a single sorted iterator

```
import heapq


list1 = [2, 4, 6, 8]
list2 = [1, 3, 5, 7]
list3 = [0, 9, 10]


merged_list = list(heapq.merge(list1, list2, list3))
print(f"Merged list: {merged_list}")
# Output: Merged list: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

About Sorting

Chapter.4-Sorting and Searching

Importance of Sorting

- Why don't CS profs ever stop talking about sorting?
  - Computers spend a lot of time sorting, historically 25% on mainframes.
  - Sorting is the best studied problem in computer science, with many different algorithms known.
  - Most of the interesting ideas we will encounter in the course can be taught in the context of sorting, such as divide-and-conquer, randomized algorithms, and lower bounds.
- You should have seen most of the algorithms, so we will concentrate on the analysis.

Efficiency of Sorting

- Sorting is important because that once a set of items is sorted, many other problems become easy.
- Further, using $O(n \log n)$ sorting algorithms leads naturally to sub-quadratic algorithms for all these problems.

n	$n^2/4$	$n \log n$
10	*25*	33
100	2,500	*664*
1,000	250,000	*9,965*
10,000	25,000,000	*132,877*
100,000	2,500,000,000	*1,660,960*
1,000,000	250,000,000,000	*13,815,551*

- Large-scale data processing is impossible with $\Omega(n^2)$ sorting.
(由上表得知若用n^2演算法排序大數據，效能是無法接受的)

Application of Sorting: Searching

- Binary search lets you test whether an item is in a dictionary in $O(\log n)$ time.
- Search preprocessing is perhaps the single most important application of sorting.

Application of Sorting: Closest pair

- Given n numbers, find the pair which are closest to each other.
- Once the numbers are sorted, the closest pair will be next to each other in sorted order, so an $O(n)$ linear scan completes the job.

Application of Sorting: Element Uniqueness

- Given a set of n items, are they all unique or are there any duplicates?
- Sort them and do a linear scan to check all adjacent pairs. This is a special case of closest pair above.

Application of Sorting: Mode

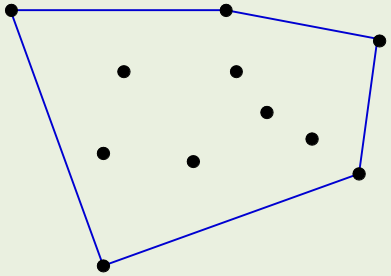
- Given a set of n items, which element occurs the largest number of times? More generally, compute the frequency distribution.
- Sort them and do a linear scan to measure the length of all adjacent runs.
- The number of instances of k in a sorted array can be found in $O(\log n)$ time by using binary search to look for the positions of both $k - \varepsilon$ and $k + \varepsilon$(where ε is arbitrarily small) and then taking the difference of these positions.

Application of Sorting: Median and Selection

- What is the kth largest item in the set?
- Once the keys are placed in sorted order in an array, the kth largest can be found in constant time by simply looking in the kth position of the array.
- There is a linear time algorithm for this problem, but the idea comes from partial sorting.

Application of Sorting: Convex hulls(凸包)

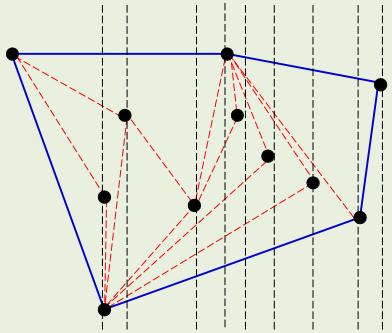
- Given n points in two dimensions, find the smallest area polygon which contains them all.



- The convex hull is like a rubber band stretched over the points.
- Convex hulls are the most important building block for more sophisticated geometric algorithms.

Finding Convex Hulls

- Once you have the points sorted by x-coordinate, they can be inserted from left to right into the hull, since the rightmost point is always on the boundary.



- Sorting eliminates the need check whether points are inside the current hull.
- Adding a new point might cause others to be deleted.

Pragmatics of Sorting: Comparison Functions

- Alphabetizing is the sorting of text strings.
- Libraries have very complete and complicated rules concerning the relative collating sequence of characters and punctuation.
  - Is *Skiena* the same key as *skiena*?
  - Is *Brown-Williams* before or after *Brown America* before or after *Brown, John*?
- Explicitly controlling the order of keys is the job of the ***comparison function*** we apply to each pair of elements, including the question of increasing or decreasing order.

(實務上進行排序前要決定好比較函數：怎麼判定先後順序)

Pragmatics of Sorting: Equal Elements

- Elements with equal keys will all bunch together in any total order, but sometimes the relative order among these keys matters.
- Often there are secondary keys (like first names) to test after the primary keys. This is a job for the comparison function.
- Certain algorithms (like quicksort) require special care to run efficiently with large numbers of equal elements.

(比較函數可能也需要能決定如何處理主鍵相同時的順序)

Pragmatics of Sorting: Library Functions

- Any reasonable programming language has a built-in sort routine as a library function. You are almost always better off using the system sort than writing your own routine.

(實務上請直接使用程式語言提供的程式庫內建排序來用)

```
my_list = [3,2,5,7,10,1,4,9,6,8]
my_list.sort()
print(my_list)
```

Mergesort

Chapter.4-Sorting and Searching

Mergesort(合併排序)

- Recursive algorithms are based on reducing large problems into small ones.
- A nice recursive approach to sorting involves partitioning the elements into two groups, sorting each of the smaller problems recursively, and then interleaving the two sorted lists to totally order the elements.
- https://upload.wikimedia.org/wikipedia/commons/c/cc/Merge-sort-example-300px.gif
- https://www.youtube.com/watch?v=XaqR3G_NVoo

Mergesort Implementation

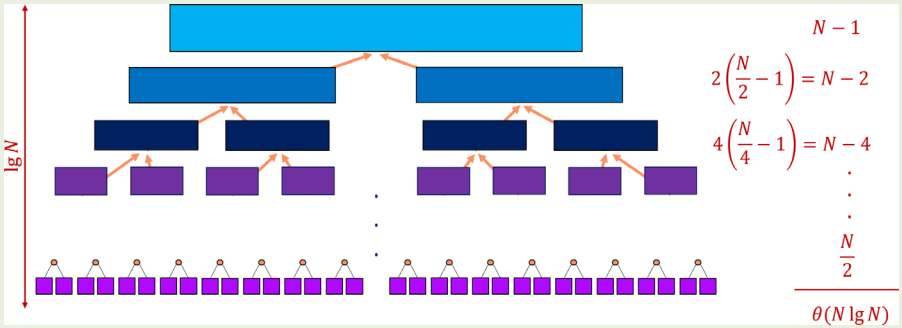
```
def merge_sort(s:list, low:int, high:int):
    if low < high:
        middle = (low+high)//2
        merge_sort(s, low, middle)
        merge_sort(s, middle+1, high)

        merge(s, low, middle, high)
```

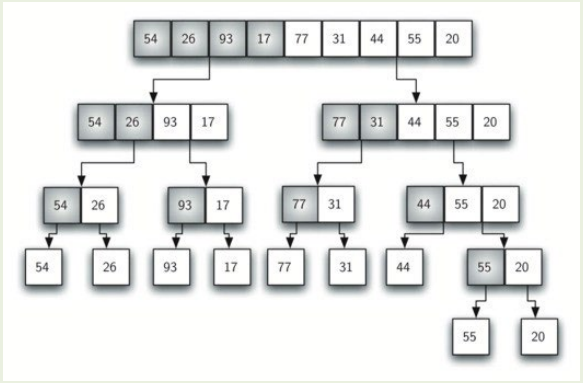
Merging Sorted Lists(合併兩個排序好的串列)

- The efficiency of mergesort depends upon how efficiently we combine the two sorted halves into a single sorted list.
- This smallest element can be removed, leaving two sorted lists behind, one slightly shorter than before.
- Repeating this operation until both lists are empty merges two sorted lists (with a total of n elements between them) into one, using at most $n - 1$ comparisons or $O(n)$ total work Example: A = $\{5, 7, 12, 19\}$ and B = $\{4, 6, 13, 15\}$.

Mergesort Analysis

- A linear amount of work is done merging along all levels of the mergesort tree.
- The height of this tree is $O(\log n)$.
- Thus the worst case time is $O(n \log n)$.





External Sorting(外部排序)

- Which $O(n \log n)$ algorithm you use for sorting doesn't matter much until n is so big the data does not fit in memory. Mergesort proves to be the basis for the most efficient *external sorting* programs.
- Disks are much slower than main memory, and benefit from algorithms that read and write data in long streams – not random access.

Divide and Conquer(分而治之)

- Divide and conquer is an important algorithm design technique using in mergesort, binary search, the fast Fourier transform (FFT), and Strassen's matrix multiplication algorithm.
- We divide the problem into two smaller subproblems, solve each recursively, and then meld the two partial solutions into one solution for the full problem.
(將問題切成兩個子問題，一直切下去到很好解決時分頭去解決得到部分答案，再一步步回頭合成完整答案)
- When merging takes less time than solving the two subproblems, we get an efficient algorithm.

$$T(n) = 2 \cdot T(\frac{n}{2}) + \Theta(n) \rightarrow T(n) = \Theta(n \log n)$$

[補充]Divide-and-Conquer 技巧

■Divide-and-conquer 的設計策略包含下列的步驟:

- 切割 (Divide) 一個較大的問題以成為一個或多個較小的問題。
- 征服 (Conquer；或稱解決solve) 每一個較小的問題。
  - *除非問題已經足夠的小，否則使用**遞迴**來解決。*
- **如果需要**, 將所有小問題的解答加以合併(combine) ，以獲得原始問題的解答。
  - *需要合併的問題: Merge sort*
  - *不需要合併的問題: Binary search*

Exercise

Problems of the Day

- What is the final min-heap array after the following sequence of operations?
  - Initial empty heap, sequentially insert: 10, 5, 15, 7, 12, 20, 6
- What is the time complexity of determining whether the kth smallest element in a min-heap of n elements is greater than or equal to a given value x?
- How many comparisons are required to merge two sorted arrays of sizes n/2 each?
- Which of the following best describes the divide-and-conquer approach used in merge sort?

Program Exercises (Moodle CodeRunner)

- Exercise 04 (close at 3/25 23:59)
  - Merge Two Sorted Arrays: Suppose you are given two sorted arrays of integers. If one array has enough empty entries at its end, it can be used to store the combined entries of the two arrays in sorted order.
    - *For example, consider [5,13,17,None,None,None,None] and [3,7,11,19], where "None" denotes an empty entry. Then the combined sorted entries can be stored in the first array as [3,5,7,11,13,17,19].*
  - Find k closest numbers: Given a sorted integer array arr, and two integers k and x, return the k closest integers to x in the array. The result should also be sorted in ascending order. An integer a is closer to x than an integer b if: |a - x| < |b - x|, or |a - x| == |b - x| and a < b.
    - *For example, arr = [1,2,3,4,5], k = 4, x = 3: Returns [1,2,3,4]*