Part IA Algorithms Notes

1 Sorting

1.1 Insertion Sort

```
def insertionSort(a):
    for i from 1 included to len(a) excluded:

        j = i - 1
        while j >= 0 and a[i] > a[j + 1]:
            swap(a[j], a[j + 1])
            j = j - 1
```

- Run the insertion sort algorithm on the integer array a, sorting it in place
- Precondition - Array a contains len(a) integer values
- Postcondition - Array a contains the same integer values as before, but now they are sorted in ascending order
- Has a complexity of $O(n^2)$

1.2 Selection Sort

```
def selectionSort(a)
    for k from 0 included to len(a) excluded:
            iMin = k
            for j from iMin + 1 included to len(a) excluded:
                if a[j] < a[iMin]:
                    iMin = j
            swap(a[k], a[iMin])
```

- Repeatedly choose the smallest element in the unsorted partition and append it onto the sorted partition
- Precondition - Array a contains len(a) integer values
- Postcondition - Array a contains the same integer values as before, but now they are sorted in ascending order
- The number of comparisons is independent of the state of the input array
- Has a complexity of $\Theta(n^2)$ and therefore $O(n^2)$

1.3 Binary Insertion Sort

```
def binaryInsertionSort(a):
    for k from 1 included to len(a) excluded:
        if i != k:
            tmp = a[k]
            for j from k - 1 included down to i - 1 excluded:
                a[j + 1] = a[j]
            a[i] = tmp
```

- Repeatedly inserts the next element into the sorted portion of the list, using a binary search to find the new element's correct position
- Precondition - Array a contains len(a) integer values
- Postcondition - Array a contains the same integer values as before, but now they are sorted in ascending order
- Requires only $O(n\,lg\,n)$ comparisons, a binary search per item and $O(n^2)$ swaps, giving a complexity of $O(n^2)$

1.4 Bubble Sort

```
def bubbleSort(a):
    repeat:
        didSomeSwapsInThisPass = False
        for k from 0 included to len(a) − 1 excluded:
            if a[k] > a[k + 1]:
                swap(a[k], a[k + 1])
                didSomeSwapsInThisPass = True
    until didSomeSwapsInThisPass == False
```

- Repeated passes through the array during which adjacent values are compared and if out of order swapped. The algorithm terminates as soon as a full pass requires no swaps
- Has a complexity of $O(n^2)$, but on an already sorted list it will require only linear time

1.5 Mergesort

```
def sort(a):
    if len(unsorted_list) < 2:
        return a
    else:
        midpoint = len(a)/2
        a1 = sort(a[0:midpoint])
        a2 = sort(a[midpoint:END])
        a3 = list the same length as the list a
        i1 =0
        i2 = 0
        for i3 from 0 included to len(a) excluded:
            if i1 >= len(a1):
                a3[i3] = a2[i2]
                i2 += 1
            elif i2 >= len(a2):
                a3[i3] = a1[i1]
                i1 += 1
            elif a1[i1] < a2[i2]:
                a3[i3] = a2[i2]
                i2 += 1
            else:
                a[i3] = a1[a1])
                i1 +=1
        return a3
```

- Split the array in two and call mergesort on both arrays, merge the result, to give the value to be returned
- The algorithm will not sort in place, it will instead return a sorted list
- This has a time complexity of $O(n\,lg\,n)$ and space complexity of $O(n)$
- The above is top-down merge sort, using recursion. Can instead do bottom-up merge sort, which repeatedly merges increasingly large sorted lists

1.6 Quicksort

```
def quickSort(a, p, r):
    if p < r:
        x = a[r]
        i = p − 1
        for j from p included to r excluded:
            if a[j] <= x:
                i = i + 1
                swap(a[i], a[j])
        swap(a[i + 1], a[r])
        q = i + 1
```

```

        quickSort(A, p, q - 1)
        quickSort(A, q + 1, r)
```

- Choose a pivot, and partition list into those less than or equal to the pivot and those greater than the pivot
- Repeat this procedure on each partition
- Has a worst case complexity of $O(n^2)$, when the pivot is either then largest or smallest in the partition
- Has an average case complexity of $\Theta(n\,lg\,n)$, can be found using the following recurrence relation

$$f(n) = kn + \frac{1}{n}\sum_{i=1}^{n}\Big(f(i-1) + f(n-1)\Big)$$

- Because in real life most input lists will be at least partially sorted, the pivot is often chosen at random
- Due to quicksort's overheads, if is often faster to use another sort on small lists, e.g. insertion sort
- **Median and Order Statistics**
  - Looking to find an item at a position k in an ordered list
  - Partition the list, if the required index is less than the position of the pivot then recurse on the lower partition, else on the larger
  - This gives the following recurrence relation for the average case, $f(n) = f(n/2) + kn$, and a complexity of $O(n)$

1.7 Heapsort

```
def heapSort(a):
    for k from floor(END/2) excluded down to 0 included:
        heapify(a, END, k)

    for k from END excluded down to 1 excluded:
        swap(a[0], a[k - 1])
        heapify(a, k - 1, 0)

def heapify(a, iEnd, iRoot):
    if a[iRoot] satisfies the max-heap property:
        return
    else:
        let j point to the largest among the existing children of a[iRoot]
        swap(a[iRoot], a[j])
        heapify(a, iEND, j)
```

- **The max-heap property -** The value at index k is greater than or equal to the values at $2k+1$ and $2k+2$, zero indexed array
- Array is isomorphic to a binary tree
- A binary heap is an almost full binary tree, every level in the tree except the last must be full, the last must either be full or have its empty spaces on the right
- Has two phases
  1. Turns an unsorted array into a max-heap ($O(n)$)
  2. Removes the root of the heap, swaps it to the end of the array and reforms the heap with one fewer elements ($O(n\,lg\,n)$)
- `heapify` takes a root with two heaps as children, it tests if the root violates the heap property, if it does it swaps the root with the larger child and recurses down that subtree

- Building a heap can be done in $O(n)$, proof:
  First remember:

level	num nodes in level	height of subtree	max cost of heapify
0	1	h	kh
1	2	$h-1$	$k(h-1)$
2	4	$h-2$	$k(h-2)$
$\vdots$	$\vdots$	$\vdots$	$\vdots$
j	2^j	$h-j$	$k(h-j)$
$\vdots$	$\vdots$	$\vdots$	$\vdots$
h	2^h	0	0

  Heapify must be applied to all elements, on all levels giving

$$
\begin{aligned}
C(h) &= \sum_{j=0}^{h} 2^j \cdot k(h-j) \\
&= k\frac{2^h}{2^h}\sum_{j=0}^{h} 2^j \cdot (h-j) \\
&= k2^h \sum_{j=0}^{h} 2^{j-h} \cdot (h-j) \\
&let\ l = h-j \\
&= k2^h \sum_{l=0}^{h} l2^{-l} \\
&= k2^h \sum_{l=0}^{h} l\left(\frac{1}{2}\right)^l \\
&Use\ the\ following\ convergence: \\
|x| < 1 &\implies \sum_{i=0}^{\infty} ix^i = \frac{x}{(1-x)^2} \\
C(h) &\in O(2^h) \\
&\in O(n)
\end{aligned}
$$

1.8 Faster Sorts

- $O(n\,lg\,n)$ is the minimum complexity of a comparison sort, different types of sort can have lower complexities
- **Counting Sort** - Sorting integer keys from a fixed range
  - The procedure
    1. Create a second array indexed by every integer in your range, appearing in the desired order
    2. Iterate through the input list and for every value k encountered increment the value held in the second array at index k
    3. Use the second array to calculate the starting position of each block of values
    4. Iterate through the input array and for each value k encountered query the position that is should be placed in the output and increment the position counter
  - This has a complexity of $O(n)$
- **Bucket Sort** - Sorting real keys uniformly distributed over a fixed range

- The procedure, for range scaled to be between 0 and 1
  1. Create an array of n linked lists
  2. Insert each item into the array at position $\lfloor k \cdot n \rfloor$
  3. Sort each list with insert sort and then output it
- Although insert sort has a complexity of $O(n^2)$ it can be shown that on average it bucket sort has a complexity of $O(n)$

- **Radix Sort** - Short keys of a fixed length
  - The procedure
    1. First sort the elements by their least significant digit, using some stable sort
    2. Repeat the above on all digits, moving from least to most significant digit

2 Complexity and Correctness

2.1 Correctness

- A useful technique for proving correctness is to split the algorithm into smaller sub-problems and use mathematical induction on each of these
- Place assertions throughout the algorithm, invariants that will hold for all cases

2.2 Computational Complexity

- It is necessary to make a number of simplifying assumptions before performing cost estimation
  1. Only worry about the worst possible amount of time
  2. Only look at *rate of growth*, ignore constant multiples, $1000f(n)$ considered $f(n)$
  3. Any finite number of exceptions are unimportant, as long as it holds for all $n > N$ for some large N
  4. Purely mathematical, no reality checks applied
- **Big-O Notation**
$$f(n) \in O(g(n)) \iff (\exists\ k > 0, N > 0.\ (0 \leq f(n) \leq k \cdot g(n)))$$
  - $f(n)$ will never exceed $g(n)$ for sufficiently large n, except from a constant proportion of $g(n)$
  - Means that $f(n)$ grows at most like $g(n)$, no faster
- **Big-Θ Notation**
$$f(n) \in \Theta(g(n)) \iff (\exists\ k_1 > 0, k_2 > 0, N > 0.\ (0 \leq k_1 \cdot g(n) \leq f(n) \leq k_2 \cdot g(n)))$$
  - $f(n)$ and $g(n)$ agree with each other within a constant factor
  - Means that $f(n)$ grows exactly at the same rate as $g(n)$
- **Big-Ω Notation**
$$f(n) \in \Omega(g(n)) \iff (\exists\ k > 0, N > 0.\ (k \cdot g(n) \leq f(n) \leq \infty))$$
  - $g(n)$ will never exceed $f(n)$ for sufficiently large n, except from a constant proportion of $g(n)$
  - Means that $f(n)$ grows at least like $g(n)$
- **Small-o Notation**
$$f(n) \in o(g(n)) \iff (\exists\ k > 0, N > 0.\ (0 < f(n) < k \cdot g(n)))$$
  - Bounds not tight

- **Small-ω Notation**

$$f(n) \in \omega(g(n)) \iff (\exists\, k > 0, N > 0.\ (k \cdot g(n) < f(n) < \infty))$$

  - Bounds not tight

- Informally

	If...	then $f(n)$ grows ... $g(n)$	$f(n)$ ... $g(n)$
Small-o	$f(n) \in o(g(n))$	strictly more slowly than	$<$
Big-O	$f(n) \in O(g(n))$	at most as quickly as	$\leq$
Big-Θ	$f(n) \in \Theta(g(n))$	exactly like	$=$
Big-Ω	$f(n) \in \Omega(g(n))$	at least as quickly	$\geq$
small-ω	$f(n) \in \omega(g(n))$	strictly more quickly than	$>$

- Types of analysis
  - Worst Case
  - Average Case
  - Amortized Analysis
- Minimum Cost of Sorting
  - Lower bound on exchanges
    * In an n item array then $\Theta(n)$ exchanges always suffices to put items in order. In the worst case, $\Theta(n)$ are actually needed
  - Lower bound on comparisons
    * Sorting by pairwise comparison, necessarily costs at least $\Omega(n\, lg\, n)$
    * $n!$ permutations of an element array, identifying one of these using pairwise comparisons requires $lg(n!)$ comparisons
    * By Stirling's Approximation, $ln(n!) = \Theta(n\, lg\, n)$

2.3 Recurrence Relations

- Take a substitution of $n = 2^m$
- Repeatedly expand the recurrence expression, until you reach f(1) or f(0), which have constant know costs
- Rearrange the expression to give $f(n)$ in terms of only constants and m
- Sub n back into the formula to give the complexity of the recurrence

2.4 Amortized Analysis

- Some data structures may have procedures that ordinary are fast, but occasionally are slow
- Using normal worst-case analysis these slow procedures would dominate; however, this is a needlessly pessimistic bound
- Instead we average the time required to perform a sequence of data structure operations over all operations performed
- Amortized analysis guarantees the *average performance of each operation in the worst case*
- There are two ways of performing amortized analysis

2.4.1 Aggregate Analysis

- We show that for all n, a sequence of n operations takes a worst-case time of $T(n)$, therefore the average cost, or amortized cost, per operation is $T(n)/n$
- This cost then applies to each operation

2.5 The Potential Method

- Some data structures have a natural way to measure *stored up mess that has to be cleaned up eventually*
- Let Φ be the **Potential function**, that maps possible states of the data structure to non-negative real numbers
- Let c be the true cost of an operation, let $\mathcal{S}$ be the state of the data structure just before, let $\mathcal{S}'$ be the state after
- Define the **amortized cost** of the operation to be

$$c + \Phi(\mathcal{S}) - \Phi(\mathcal{S}')$$

- Consider a sequence of operations on the data structure

$$\mathcal{S}_0 \to_{c_1} \mathcal{S}_1 \to_{c_2} \mathcal{S}_2 \to_{c_3} \cdots \to_{c_k} \mathcal{S}_k$$

  with the true costs $c_1, \dots, c_k$, the total amortized cost

$$\begin{aligned}
&= [-\Phi(\mathcal{S}_0) + c_1 + \Phi(\mathcal{S}_1)] + [-\Phi(\mathcal{S}_1) + c_2 + \Phi(\mathcal{S}_2)] + \cdots + [-\Phi(\mathcal{S}_{k-1}) + c_k + \Phi(\mathcal{S}_k)] \\
&= c_1 + c_2 + \cdots + c_k - \Phi(\mathcal{S}_0) + \Phi(\mathcal{S}_k) \\
&= \textit{total true cost} - \Phi(\mathcal{S}_0) + \Phi(\mathcal{S}_k)
\end{aligned}$$

- It is convenient to set $\Phi = 0$ when the data structure is created, and require that $\Phi \geq 0$ for all possible states $\mathcal{S}$, this guarantees that the total true cost of any sequence of operations is $\leq$ the total cost

3 Algorithm Design

3.1 Dynamic Programming

- Solving a problem by splitting it into smaller subproblems and then solving these once
- Two different way of solving these problems only once
  - Bottom-up, solve the smaller problems first and keep track of them
  - Memoization - This defines a function that before computing a function tests if it has already been computed, if so it will return the previous answer, if not it will compute and store it
- Good for the following types of problem
  - There exists many choices, each with a score, which must be optimised
  - The number of choices is exponential in the size of the problem, so brute-force not applicable
  - The structure of the optimal solution is such that it is composed of optimal solutions to smaller problems
  - There is overlap: in general, the optimal solution to a sub-problem is required to solve several higher-level problems, not just one

3.2 Greedy Algorithms

- Solve a problem by choosing the action that will move you closest to the goal
- Must be careful because it can get you stuck in a local maximum instead of the global
- Proving the correctness of a greedy algorithm
  1. Cast the problem as one where we make a greedy choice and are left with just one smaller problem to solve
  2. Prove that the greedy choice is always part of an optimal solution
  3. Prove that there's optimal substructure, i.e. that the greedy choice plus an optimal solution of the subproblem yields an optimal solution for the overall problem

3.3 Recognise a Variant of a Known Problem

- Can adapt an algorithm for another problem to solve others

3.4 Reduce to a Simpler Problem

- Recursive solutions often lend themselves to this
- Then use a inductive proof to prove the algorithm's correctness

3.5 Divide and Conquer

- Problem can sometimes be solved in three steps
  1. **Divide**
     - If problem small use brute force
     - Otherwise split problem into two (or more, although usually two) roughly equal sub-problems
  2. **Conquer**
     - Recurse on these smaller problems
  3. **Combine**
     - Create a final solution by using information from the solution to the smaller problems

3.6 Backtracking

- If the algorithm requires a search
- Split the search into two parts
  - The first ploughs through investigating what it thinks is the optimal path
  - The second back tracks when needed
- The second part is invoked when the first reaches a dead end
- Often useful in graph algorithms

3.7 The MM Method

- Give a group of students the problem and some time and wait, they will come up with something no one person could
- Or try every sequence of instructions until you find the sequence that achieves the desired output

3.8 Others

- Write a simple method and remove wasteful complexity
- Find a mathematical lower bound, use this to aid in the algorithm's design

4 Data Structures

4.1 Implementing Data structures

- A vector is an abstract data type, an array is a low-level type provided by programming languages
- Diagrams showing points as called **record-pointer diagrams**

4.2 Abstract Data Structures

- Like a Java interface, provides a definition of the things that a data structure must achieve
- If the precondition is violated the result of the function is undefined

4.3 Stacks

```
ADT Stack {
    boolean isEmpty();
    // Behaviour: return true iff the structure is empty

    void push(item x);
    //Behaviour: add element x to the top of the stack
    //Postcondition: isEmpty() == False
    //Postcondition: top() == x

    item pop();
    //Precondition: isEmpty() == False
    //Behaviour: return and remove element from the top of the stack

    item top();
    //Precondition: isEmpty() == False
    //Behaviour: Return the element on top of the stack (without removing it)
}
```

- First-in last-out
- Can be simply implemented in one of two ways
  - An array and a pointer pointing to the top of the stack
  - A linked list, with the head of the list as the top of the stack

4.4 List

```
ADT List {
    boolean isEmpty();
    // Behaviour: Return true iff the structure is empty

    item head();
    // Precondition: isEmpty() == false
    // Behaviour returns the first element of the list (without removing it)

    void prepend(item x);
    // Behaviour: add element x to the beginning of the list
    // Postcondition: isEmpty() == false
    // Postcondition: head() == x
```

```
13
14      List tail();
15      // Precondition: isEmpty() == false
16      // Behaviour: Return the list of all the elements except the first (without removing them)
17
18      void setTail(List newTail);
19      // Precondition: isEmpty() == false
20      // Behaviour: Replace the tail of this list with newTail
21  }
```

- Can be implemented with waggons and pointers, arrays, and so forth

4.5 Queue

```
1   ADT Queue {
2       boolean isEmpty();
3       // Behaviour: return true iff the structure is empty
4
5       void put(item x);
6       // Behaviour: insert the element x at the end of the queue
7       // Postcondition: ieEmpty() == false
8
9       item get();
10      // Precondition: isEmpty() == false
11      // Behaviour: return the first element in the queue, removing it from the queue
12
13      item first();
14      // Precondition: isEmpty() == false
15      // Behaviour: return the first element of the queue, without removing it
16  }
```

- This is a first-in first-out (fifo) structure
- Can also have double queue or deque

4.6 Deque

```
1   ADT Deque {
2       boolean isEmpty();
3
4       void putFront(item x);
5       void putRear(item x);
6       // Postcondition: isEmpty() == false
7
8       item getFront();
9       item getRear();
10      // Precondition: isEmpty() == false
11  }
```

- Stacks and queues can be thought of as subclasses of this

4.7 Priority Queue

```
1   ADT PriorityQueue {
2       void insert(Item x);
3       // Behaviour: add item x to the queue
4
5       Item first();
6       // Behaviour: return the item with the smallest key (without removing it from the queue)
7
8       Item extractMin();
9       // Behaviour: return the item with the smallest ket and remove it from the queue
10
11      void decreaseKey(Item x, Key new);
```

```
12      // Precondition: new < x.key
13      // Precondition: Item x is already in the queue
14      // Postcondition: x.key == new
15      // Behaviour: Change the key of the key of the designated item to the designated value,
        increasing the item's priority
16
17      void delete(Item x);
18      // Precondition: item x is already in the queue
19      // Behaviour: remove item x from the queue
20      // Implementation: make x the new minimum by calling decrease key with a value (minus
        infinity) smaller than any in the queue; then extract the minimum and discard it
21  }
22
23  ADT Item {
24      // A total order is defined on the keys
25      Key k;
26      Value v;
27      }
```

- Could just use a sorted array, but would have to keep it sorted with every operation, giving linear costs for all operations
- Instead use a form of heap (see below)

4.8 Dictionary

```
1  ADT Dictionary {
2      void set(Key k, Value v);
3      // Behaviour: store the given (k, v) pair in the dictionary, if k is already assigned then
       the previous value is overwritten
4      // Postcondition: get(k) == v
5
6      Value get(Key k);
7      // Precondition: a pair with the key k is in the dictionary
8      // Behaviour: return the value associated with the supplied k, without removing it from the
        dictionary
9
10     void delete(Key k);
11     // Precondition: a pair with the given key k has already been inserted
12     // Behaviour: remove from dictionary the key-value pair indexed by k
13 }
```

- Can be thought of as a function, every object in the domain set has zero or one mappings to something in the codomain
- If the key are within an integer range, or can be mapped to that range (e.g. by subtraction) then a vector can be used
  - This is called **direct-addressing**
  - If the range is large, this can be too inefficient in space even though `get()` and `set()` have $O(1)$ complexity
- Can instead store it in a list and preppend key-value pair each time
- Can store in a sorted array and use a binary search, giving the recurrence relation $f(n) = f(n/2) + k$ and therefore a complexity of $\Theta(ln\, n)$
- Can be stored in a binary search tree

4.9 Sets

```
ADT Set {
    void set(Key k, Value v);
    // Behaviour: store the given (k, v) pair in the set, if k is already assigned then the
    previous value is overwritten
    // Postcondition: get(k) == v

    Value get(Key k);
    // Precondition: a pair with the key k is in the set
    // Behaviour: return the value associated with the supplied k, without removing it from the
     set

    void delete(Key k);
    // Precondition: a pair with the given key k has already been inserted
    // Behaviour: remove from set the key-value pair indexed by k

    boolean isEmpty();
    // Behaviour: return true iff the structure is empty

    boolean hasKey(Key x);
    // Behaviour: return true iff the set contains a pair keyed by x

    Key chooseAny();
    // Precondition: isEmpty() == false
    // Behaviour: return the key of an arbitrary element from the set

    Key min();
    // Precondition: isEmpty() == false
    // Behaviour: return the smallest key in the set

    Key max();
    // Precondition: isEmpty() == false
    // Behaviour: return the largest key in the set

    Key predecessor(Key k);
    // Precondition: hasKey(k) == true
    // Precondition: min() != k
    // Behaviour: return the largest key in the set that is smaller than k

    Key successor(Key k);
    // Precondition: hasKey(k) == true
    // Precondition: max() != k
    // Behaviour: return the smallest key in the set that is larger than k

    Set unionWith(Set s);
    // Behaviour: change this set to become the set obtained by forming
    //              the union of this set and s
}
```

4.10 Disjoint Sets

```
ADT DisjointSet {
    Handle get_set_with(Item x);
    // Postcondition: The handle must be stable, as long as the DisjointSet is not modified
    // Behaviour: Returns the handle to the set containing an item

    add_singleton(Item x);
    // Behaviour: Ass a new set consisting of a single item

    merge(Handle x, Handle y);
    // Behaviour: Merge two sets into one
}
```

- Used to keep track of of a dynamic collection of disjoint sets
- Different implementations
    1. Flat Forest

- One member of the set is the root
- All other members are children of the root, all the trees have a depth of 1 or 2
- Items in a tree could be stored as a linked list
- `get_set_with` is just a single look up
- `merge` must iterate through all nodes in a tree and change their pointer, $O(n)$
- **weighted union heuristic** - Keep the size of each set and update the pointers in the smaller set
  * Aggregate analysis, m operations on n elements, $O(m + n\,lg\,n)$

2. Deep forest
   - To merge two sets, just set the root of one to be the child of the other's root
   - `get_set_with` must walk up to the root, $O(h)$, where h is the height of the tree
   - `merge` attach one root to the other, which only involves updating a single pointer
   - **union by rank heuristic** - Keep track of the rank of each root (the tree's height) and always attach the lower-rank
     * For two trees of ranks r_1 and r_2, the resulting rank is given by

$$\text{Resulting rank} = \begin{cases} max(r_1,\ r_2), & r_1 \neq r_2; \\ r_1 + 1, & r_1 = r_2; \end{cases}$$

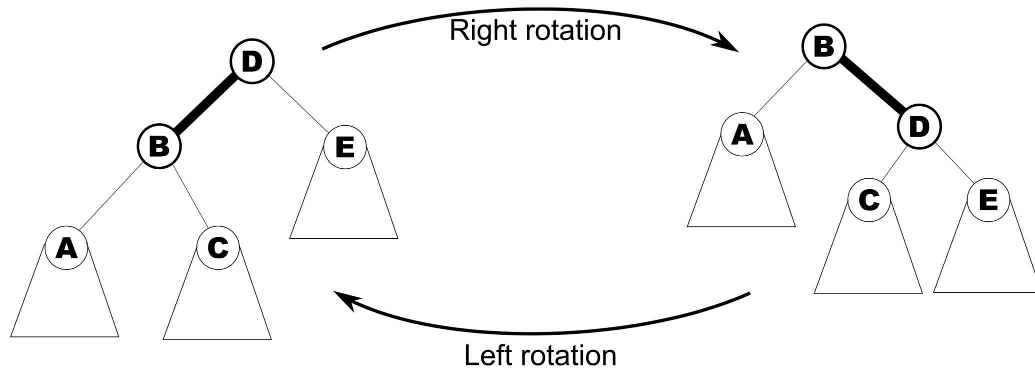
3. Lazy Forest
   - **path compression heuristic** - Merge two sets as in a deep forest, but when you walk up the tree to find the root do it a second time and set all intermediate nodes to point to the root
   - Rank isn't adjusted during path compression and is therefore an upper bound on the height
   - `get_set_with` walk to root twice, first time to find it, second to update all intermediate pointers
   - `merge` attach one root to the other, which only involves updating a single pointer
   - Aggregate analysis, m operations on n elements, $O(m\alpha_n)$
     * α_n is an integer sequence, that increases very slowly and can be ignored for big-O notation
     * This gives an amortized complexity per operation of $O(1)$

4.11 Binary Search Trees

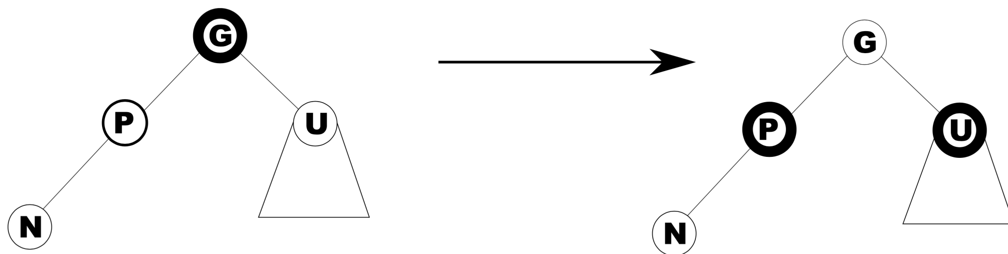
- Binary search trees are always stored in such a way as to satisfy the **binary-search-tree property**
- Let x be a node in a binary tree. If y is a node in the left subtree of x, the y.key $\leq$ x.key. If y is a node in the right subtree of x, the y.key $\geq$ x.key
- To find a node's successor
  1. If it has a right subtree then the successor must be the smallest value in it
  2. If not, go to its parent, grandparent, etc. until you go up-and-right rather than up and left, that will its successor
  3. If you reach the root before going up-and-right, then the node is the largest in the tree
- If a node n has two children then its successor has no left child
- To insert, search for the item until you reach a leaf then attach the node
- To delete, there are three cases
  1. The node is a leaf - Just remove it
  2. The node has one child - Delete and replace with its child
  3. The node has two children - Replace with its successor (which will not have a left subtree)
     (a) Replace the successor with its right child
     (b) Then replace the node to be deleted with its successor
- On average look up has a complexity of $O(lg\,n)$, but a worst-case complexity of $O(n)$

4.12 Red-Black Trees

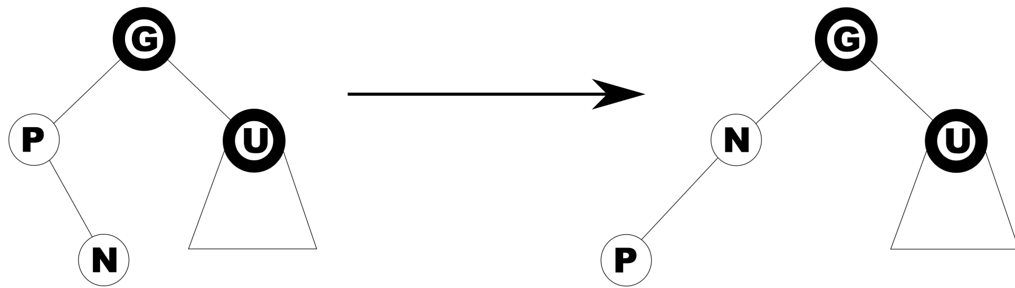
- Red-black trees are special binary search trees, that are guaranteed to be reasonably balanced, at most a height of $2lg(n+1)$
- Satisfies the following five invariants
  1. Every node is either red or black
  2. The root is black
  3. All leaves are black and never contain key-value pairs
  4. If a node is red, both its children are black
  5. For every node, all paths from that node to descendant leaves contain the same number of black nodes
- All functions that do not change the structure of the tree, e.g. `get()`, `min()`, etc. are the same as in a normal binary tree and have a complexity of $O(lg\, n)$
- However for functions that change the tree's structure rotations must be used to maintain the invariants
  - A rotation is a local transformation of a BST that changes the shape of the BST but preserves the BST property
  - Rotations have a fixed cost, independent of tree size
  - Rotations are often thought of rotating the parent node, but is really a rotation of an edge



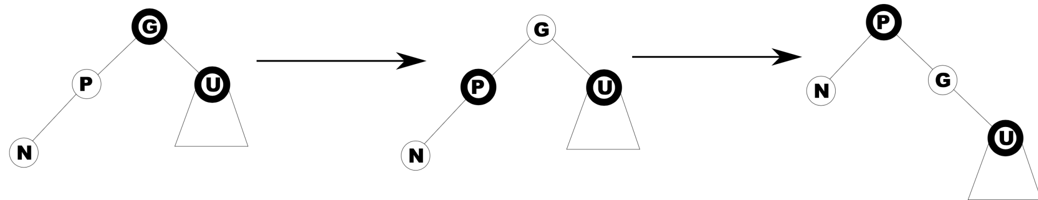
  - This does not have to be done on the root node, D can have a parent
- Implementing red-black trees, on inserting a node n
  - If the tree is empty, set n as being black and make it the root
  - If n's parent is black then make n red
  - If n's parent is red then we insert a red node
    * n's parent is red and therefore it's grandparent is black
    * Let p be n's parent, g be n's grandparent, and u be n's uncle
      1. u is red



2. u is black and the path $g \to p \to n$ is zig-zagged



3. u is black and the path $g \to p \to n$ is straight



4.13 2-3-4 Trees

- In a 2-3-4 trees there are three different classes of node, those with 2, 3, and 4 pointers to children. e.g.

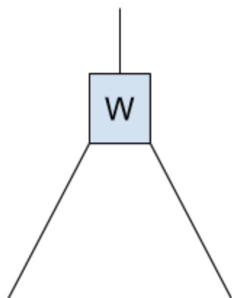


Figure 1: A 2 node

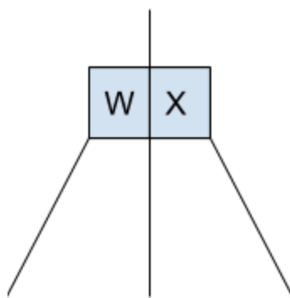


Figure 2: A 3 node

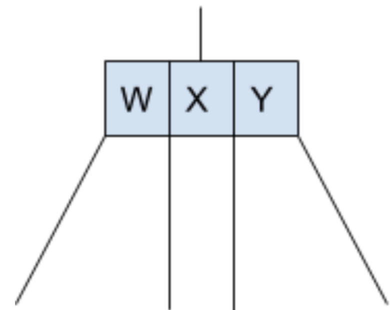
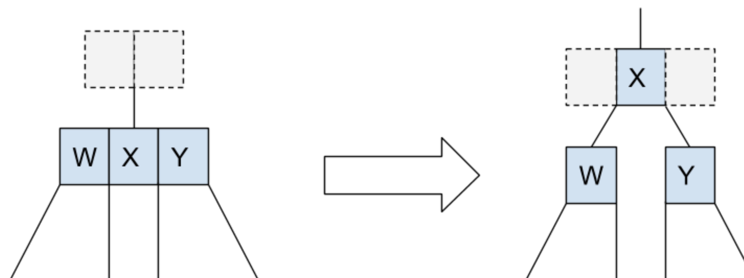


Figure 3: A 4 node

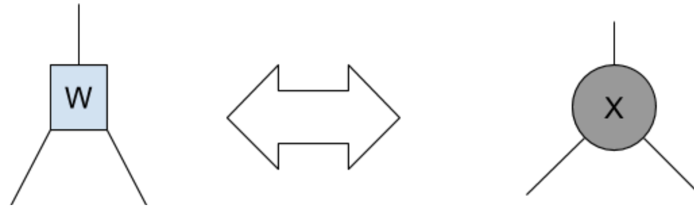
- On insertion you add the node to its parent, if it is a 2 or 3 node then it is added. If the parent is a 4 node it must be split



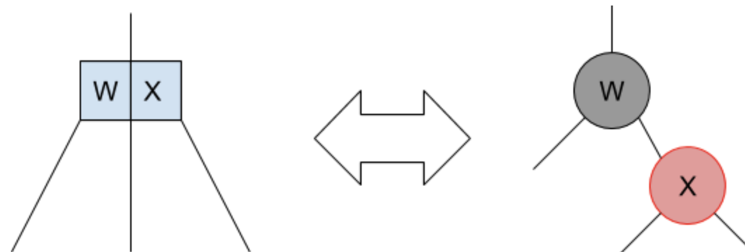
  - The central node moves up and joins the parent

  - To ensure it can join the parent without it having to split all 4 nodes are split when finding the insertion position
  - Because depth is only ever added at top of the tree, it remains balanced - to form another level the extra node must bubble to the top of the tree and become root

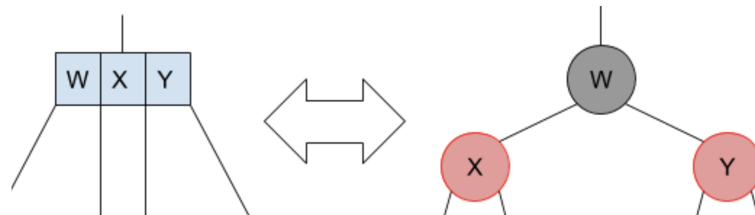
- 2-3-4 trees are isomorphic to red-black trees, like so
  - 2 Node



  - 3 Node



  - 4 Node



4.14 B-Trees

- Designed for storing data on disk, each node taking up a block or similar
- Nodes have a high branching factor, and are therefore shallow
- For a tree a **minimum degree**, t, is defined, where $t \geq 2$
  - Each node must have must have between t and $2t$ pointers, that is $t-1$ and $2t-1$ keys
- The formal rules can be stated as follows
  1. There are internal nodes (with keys and payloads and children) and leaf nodes (without keys or payloads or children)
  2. For each key in a node, the node also holds the associated payload
  3. All leaf nodes are at the same distance from the root
  4. All internal nodes have at most $2t$ children; all internal nodes except the root have at least t children
  5. A node has c children iff it has $c-1$ keys
- If a node is too full then it will be split in half, if it too small it is combined

- **Inserting**
  - Look for the key in the normal way
  - On the way down split full nodes in two, putting the median key in the parent, pointing to the two new nodes
  - When you reach the leaf node it will have space, or will have been split
  - If the root is full, split in half and promote the median to a new root node
- **Deleting**
  - Three different operations for deletion
    1. **Merge** - Merge two adjacent brother nodes and the key that separates then from the parent node. The parent loses one key
    2. **Split** - This splits a node into three, a left brother, a separating key and a right brother. The separating key is send up to the parent
    3. **Redistribute** - Redistributes the keys among two adjacent sibling nodes. It may be thought of as a merge followed by a split in a different place
  - Pseudocode for the algorithm

```
1  def delete(k):
2      if k is in the bottom node B:
3          if B can lose key without becoming too small:
4              delete k from B locally
5          else:
6              refill B
7              delete k from B locally
8      else:
9          swap k with its successor
10         # ASSERT: Now k is in a bootom node
11         delete k from the bottom node with a recursive invocation
12
13 def refill(B):
14     if either the left or right of B can afford to lose any keys:
15         redistribute keys between B and that sibling
16     else:
17         # ASSERT: B and its siblings all have the minimum number of keys, t − 1
18         merge B with either of its siblings
19         # ... this may require recursively refilling the parent of B
20         # because it will lose a key during the merge
```

4.15 Hash Tables

- Implements the dictionary ADT
- A hash function $h(k)$ maps a key of variable length onto an integer between 0 and $m-1$ so some size m
- The key-value pair (k, v) is then stored at location $h(k)$ in the array
- Collisions occur when two distinct keys k_1 and k_2 both map to the same value, that is $h(k_1) = h(k_2)$
- There are two methods for dealing with collisions
  1. Chaining
     - The locations in the array hold linked lists
     - Each time a value is hashed to an address it is added to the linked list
     - For n items the average length of each list will be $\lceil n/m \rceil$
  2. Open Addressing

    - If the location $h(k)$ is already taken a succession of other probes are made of the hash table according to some rule until either the key or an empty space is found
    - The simplest method of collision resolution is to try successive array locations, wrapping around at the end of the array
    - The table can become full
    - If the table is nearly full performance decreases significantly
    - Implementations will typically double the size of the array once occupancy goes above a certain threshold

- Hash tables can have very bad worst-case complexities
- Hash tables have constant average costs
- Deletion in open addressing
  - Mark the key-value pair as having been deleted
  - When searching, skip over this in the sequence of probing
  - When inserting, treat this as a free space
- Probing Sequences
  - A probe sequence can be described as a function taking a key k and an attempt number j

```
int probe(Key k, int j);
// Behaviour: Return the array index to be probed at attempt j for key k
```

  - **Linear Probing**
    * Of the form $(h(k) + j)$ mod m
    * Leads to primary clustering - Many failed attempts hit the same slot and spill over to the same follow-up slots, resulting in longer and longer runs of occupied cells, increasing search time
  - **Quadratic Probing**
    * Of the form $(h(k) + cj + dj^2)$ mod m, for constants c and d
    * Much better than linear probing, providing c and d are chosen well
    * Leads to secondary clustering - Two keys that hash to the same value will yield the same probing sequence
  - **Double Hashing**
    * Of the form $(h_1(k) + j \cdot h_2(k))$ mod m, for hash functions h_1 and h_2
    * Even functions with the same hash (under h_1) will have different hash sequences
    * This is the best for spreading probes across all slots
    * Each access costs an extra hash function
- Rehashing
  - When a hash table becomes too full searching becomes prohibitively costly
  - Rehashing will create a new hash-table twice as long and a new hash function
  - Every non-deleted item in the original table is hashed with the new function and stored in the new table
  - The original table it deleted

4.16 Binary Heaps

- A binary tree that satisfies two additional invariants
  1. It is almost full - All but the lowest level is full
  2. It obeys the heap property - Each node has a key less than or equal to those of its children
- In a min-heap the root of a tree will always be the smallest element in the tree
- Can be used to implement a priority queue
  - `first` - Just query the root ($O(1)$)
  - `insert` - Add the item to the end of the heap and bubble it up ($O(ln\, n)$)
  - `extractMin` - Read it out, then replace it with the last element in the heap and let it sink down to its correct position ($O(ln\, n)$)
  - `decreaseKey` - Change the item's key and let it bubble up to the correct position ($O(ln\, n)$)
  - `delete` - Decrease the key to $-\infty$ and then delete root ($O(ln\, n)$)
- A good choice to implement a priority queue
- Merging Methods
  - Can just extract each element from the smaller queue and insert it, has a complexity of $O(n\, lg\, n)$
  - Can append the arrays and call heapify, has a complexity of $O(n)$

4.17 Binomial Heaps

```
ADT BinomialHeap extends PriorityQueue {
    void Merge(BinomialHeap h):
    // Behaviour: combine the current heap with the supplied heap h. In the process, make the supplied heap h empty and incorporate all its elements into the current heap
```

- Has the advantage of allowing the merging of two priority queues in $O(lg\, n)$
- A binomial heap is a forest of binomial trees
- A **binomial tree of order 0** is a single node, containing one `Item`, it has height 0
- A **binomial tree of order k** is a tree obtained by combining two binomial trees of order $k - 1$
  - By induction it contains 2^k nodes
  - By induction the root has k nodes
  - By induction tree has a depth of k
- A **binomial heap** is a collect of binomial trees all obeying the *heap property*
- The trees are sorted in increasing size
- If a heap contains n nodes, it contains $O(ln\, n)$ binomial trees, with the largest tree with degree $O(lg\, n)$
- Can be used to implement a priority queue
  - `first` - Scan the roots of all the binomial trees in the heap ($O(lg\, n)$)
  - `extractMin` - ($O(lg\, n)$)
    * First find the root with the smallest value
    * Remove this from the tree and its children form a binomial heap
    * Merge this with the current binomial heap

- `merge` - ($O(lg\, n)$)
  - Start from order 0 and move to higher orders
  - At each order combine the trees of that order from the two heaps and any carry from the previous combinations
- `insert` - To insert a new element, consider the element as a binomial heap with one element and merge it and the heap ($O(lg\, n)$)
- `decreaseKey` - To decrease the key of an item, change it's key and let it bubble up through the tree ($O(lg\, n)$)

4.18 Fibonacci Heaps

- A Fibonacci heap is a lazy data structure, it does no cleanup while doing `push` or `decreaseKey`, each $O(1)$, but does clean up on `popmin`
- Nodes that lose a child are marked as a loser and if they lose two children are promoted to root, ensures that trees don't become too sparse
- The `push` method

```
def push(Value v, Key k):
    create a new heap n consisting of a single node (k, v)
    add n to the list of roots
    update minroot if n.key < minroot.key

```

  - Node just added to the list of roots, as a tree of order 0
  - Process has a constant complexity

- The `popmin` method

```
def popmin():
    mark all minroot's children as loser = False
    take note of minroot.value and minroot.key
    delete the minroot node, and promote all its children to be roots

    # clean up roots
    while there are two roots with the same degree:
        merge those two roots, by making the smaller root a child of the larger
    update minroot to point to the root with smallest key
    return the value and key noted on line 3

```

  - All clean up done

- The `decreasekey` method

```
def decreasekey(Value v, Key newk):
    let n be the node where this value is stored
    n.key = newk
    if n still satisfies the heap condition: return
    while True:
        p = n.parent
        remove n from p.children
        insert n into the list of roots, updating minroot if necessary
        n.loser = False
        if p.loser = True:
            n = p
        else:
            if p is not the root: p.loser = True
            break

```

  - If the new key doesn't violate the heap property no other work is done
  - Otherwise the node becomes the root of a tree, and all of its ancestors that have lost two children are moved to the root as well

- The `delete` method
  - Decrease the node's key to negative infinity and then popmin

- The `merge` method
  - Just join the two heaps root lists

- Actual implementation, store each node as a structure with the following
  - A pointer to its parent, siblings and children
  - The key and payload
  - A flag to indicate if it a loser
  - The degree of the tree that is it root for

4.18.1 Analysis

Define the following potential function

$$\Phi(\mathcal{S}) = \text{number of roots in } \mathcal{S} + 2 \cdot (\text{number of loser nodes in } \mathcal{S})$$

- `push`
  - True cost of $O(1)$
  - Increases Φ by 1
  - Gives an amortized cost of $O(1)$

- `popmin`
  - Method composed of two parts
    1. Promotes its children to root list and marks them as not losers
       * Takes $O(D)$ elementary work
       * Increases Φ by $O(D)$
    2. It then merges some number of trees, t
       * Takes $O(t)$ elementary work
       * Decreases Φ by $O(t)$
  - This gives an amortized cost of $O(D)$

- `decreasekey`
  - Two possibilities
    * New key doesn't violate the heap property
      · Takes $O(1)$ elementary work to decrease key
      · Φ doesn't change
      · Amortized cost is $O(1)$
    * New key violates the heap property, three stages
      1. Move node to root list
         · True cost $O(1)$
         · If node loser then Φ decrease by 1, else increases by 1

2. Move loser ancestors to root list
   · True cost of $O(1)$
   · New node, so Φ increases by 1, but no longer loser, so Φ decrease by 2 $\implies$ Φ decrease by 1
   · Total amortized cost of zero
3. Mark ancestor as loser
   · True cost of O(1)
   · Φ increases by 2

  * This gives an amortized cost of $O(1)$

- The method has a total amortized cost of $O(1)$

- D is the maximum number of children in any of the trees.
  - D can be bounded using the following theorem

    Theorem. *In the Fibonacci heap, if a node had d children, then the total number of nodes in the subtree rooted at the node is at least F_{d+2}, the $(d+2)^{nd}$ Fibonacci number*

  - F_d grows exponentially: $F_{d+2} \leq \varphi^d$, where $\varphi = \frac{1+\sqrt{5}}{2}$, the golden ratio

    Corollary. *In a Fibonacci heap with n items, $D = O(\log n)$*

  - The above theorem is proved as follows

    Proof. Consider some node in the Fibonacci heap, x, at a time T it has d children, $y_1, \ldots, y_k, \ldots, y_d$. Take the child, y_k, that was added at time T_k, at this time x must have had at least $k-1$ nodes. Two trees are only merged if they have the same number of nodes, therefore at time T_k, y_k must also have had at least $k-1$ children. If a node loses two children it is promoted to root, therefore at time T y_k can have lost at most one child, and as a result must have at least $k-2$ children.
    Let N_d be the minimum number of nodes in a tree whose root has degree d.

$$
\begin{aligned}
N_0 &= 1 && \text{(A node with no children)} \\
N_1 &= 2 && \text{(A node with one child and no grandchildren)} \\
N_d &= 1 + N_0 + N_1 + \cdots + N_{d-1} && \text{(The node itself + it's children)} \\
N_d &= N_{d-2} + N_{d-1}
\end{aligned}
$$

    this is the defining equation for the Fibonacci numbers, only offset by 2. □

5 Graphs

- A graph is a set of vertices and edges between them
- A graph is referred to as $G = (V,\ E)$, where V is the set of vertices and E the set of edges
- Graphs can be directed or undirected
- Not allowed multiple edges between the same pair of vertices
- Will allow edges from a vertex back to itself
- Path is a sequence of vertices connected by edges
- Cycle is a path from a vertex back to itself
- A graph is connected if there is an path between each pair of vertices
- DAG (directed acyclic graph)

- If $v_1 \to v_2$ then v_1 is v_2's parent and v_2 is v_1's child

- An undirected, connected acyclic graph is a tree, if it unconnected it is a forest
- Can be represented in one of two ways, depends on the density of the graph, density $= \frac{E}{V^2}$
  1. An adjacency list
     - An array of vertices, each with a linked list of adjacent vertices
     - Takes up $O(V + E)$ space
  2. An adjacency matrix
     - 2 dimensional array, marking adjacent vertices
     - Takes up $O(V^2)$ space

5.1 Depth First Search

Using a stack

```
def dfs(g, s):
    for v in g.vertices:
        v.seen = False
    toExplore = Stack([s])
    s.seen = True

    while not toExplore.isEmpty():
        v = toExplore.popright()
        # Do work on vertex v
        for w in v.neighbours:
            if w.seen: continue
            toExplore.pushright(w)
            w.seen = True
```

Using recursion

```
def dfs(g, s):
    for w in g.vertices:
        w.visited = False
    visit(s)

def visit(v):
    v.visited = True
    # Do work on the vertex v
    for w in v.neighbours:
        if w.visited: continue
        visit(w)
```

- To find the path between two nodes
  - For each visited vertex store the vertex's parent
  - Once the graph has been explored begin at the goal and follow the parent pointers to the start
- Has a complexity of $\Theta(V + E)$

5.2 Topological Sort

- A DAG can represent a total ordering
- In such a graph $v_1 \to v_2$ represents "The user said they prefer v_1 to v_2"
- We use a depth-first search to find the ordering, the order is the reverse of the order that the function returns from a vertex

- Use the following algorithm

```
def toposort(g):
    for v in g.vertices:
        v.visited = False
        # v.colour = White
    totalorder = []
    for v in vertices:
        if v.visited: continue
        visit(v, totalorder)
    return totalorder

def visit(v, totalorder):
    v.visited = True
    # v.colour = Grey
    for w in v.neighbours:
        if w.visited: continue
        visit(w, totalorder)
    totalorder.prepend(v)
    # v.colour = Black
```

- Proof - Assume there are two vertices v_1 and v_2, with an edge $v_1 \to v_2$, consider line 13 (v_1 becomes Grey) there are three possibilities
  - v_2 is black - Then v_2 has already been prepended to the ordering and v_1 has yet to be, so v_1 will be earlier in the ordering than v_2
  - v_2 is white - Then it has not already been visited and it will be visited as one of v_1's neighbours and will therefore be prepended before v_2, and v_1 will therefore appear earlier than v_2 in the ordering
  - v_2 is grey - Then must be in v_2's call stack and there is therefore a path from v_2 to v_1 and a path from v_1 to v_2; we have assumed that g is a DAG this is a contradiction

5.3 Breadth First Search

```
def bfs(g, s):
    for v in g.vertices:
        v.seen = False
    toExplore = Queue([s])
    s.seen = True

    while not toExplore.isEmpty():
        v = toExplore.popright()
        # Do what you want at the node

        for w in v.neighbours:
            if w.seen: continue
            toExplore.pushleft(w)
            w.seen = True
```

- Designed to find the shortest path
- Keep track of an ever expanding *frontier*
  1. Contains just the start
  2. Contains all vertices one hop from the start
  3. Contains all vertices two hops from the start
  4. etc.
- As soon as the frontier reaches the goal vertex the shortest path has been found
- Has a complexity of $O(E + V)$

5.4 Dijkstra's Algorithm

```
def dijkstra(g, s):
    for v in g.vertices:
        v.distance = infinity
    s.distance = 0
    toExplore = PriorityQueue([s], sortkey = lambda v: v.distance)

    while not toexplore.isEmpty():
        v.toExplore.popmin()
        # ASSERT: v.distance is the true shortest distance from s to v
        # ASSERT: v is never put back into toExplore
        for (w, edgecost) in v.neighbours:
            dist_w = v.distance + edgecost
            if dist_w >= w.distance: continue
            w.distance = dist_w
            if w in toExplore:
                toExplore.decreasekey(w)
            else:
                toExplore.push(w)
```

- For a graph weighted with the cost of each edge, want to find the path with the minimum cost - the **shortest path problem**
- We will visit vertices in the order of distance from the start vertex
- May come across a vertex multiple times with different costs, so a vertex is added to the frontier if it we've never come across it or our new path to it is shorter than the old path
- Could just initially put all vertices (with a distance of infinity) in to the queue
- When implemented with a Fibonacci heap it has a complexity of $O(E + V \, lg \, V)$
  - V only put into and popped from the queue once (by assertion line 10), giving $O(V \, lg \, V)$
  - Lines 12-18 only run once per edge, giving $O(E)$

Theorem. *When run on a graph as described in the problem statement, the algorithm terminates. When it terminates then, for every vertex v, v.distance is equal to the distance from s to v.*

Proof. The assertion on line 10 ensures termination, this assertion follows from that on line 9. A vertex is only added to the queue if a shorter path to it is found, if the assertion on line 9 holds then once a vertex is removed from the queue no shorter path to it exists and it is therefore impossible for it to be put back into the queue.

Assume the assertion on line 9 fails, for some vertex v. Now consider the shortest path from the start vertex, s, to that vertex v

$$\underbrace{s \to \cdots \to u}_{\text{Have been popped}} \;\to\; \underbrace{w \to}_{\text{Not popped}} \;\; \cdots \to v$$

Because the assertion failed,

$$\text{distance}(s \to v) \neq \text{v.distance}$$

also because the algorithm only sets the distance of visited vertices

$$\text{distance}(s \to v) \leq \text{v.distance}$$

combined these give,

$$\text{distance}(s \to v) < \text{v.distance}$$

Both w and v were in the priority queue and v was popped first, therefore

$$\text{v.distance} \leq \text{w.distance}$$

The vertex u has already been popped and examined. w was considered as u's neighbour, as a result

$$\text{w.distance} \leq \text{u.distance} + \text{cost}(u \to w)$$

the assertion did not fail when u was popped, therefore `u.distance` is the correct distance, and

$$\text{u.distance} + \texttt{cost}(u \to w) = \text{distance}(s \to u) + \text{cost}(u \to w)$$

given that s, u and w lie on the shortest path to v

$$\text{distance}(s \to u) + \text{cost}(u \to w) \leq \text{distance}(s \to v)$$

because the first inequality was strict,

$$\text{distance}(s \to v) < \text{distance}(s \to v)$$

a contradiction, therefore the assertion holds □

5.5 Bellman-Ford's Algorithm

```
def bf(g, s):
    for v in g.vertices:
        v.minweight = infinity
    s.minweight = 0

    repeat len(g.vertices) - 1 times:
        for (u, v, c) in g.edges:
            v.minweight = min(u.minweight + c, v.minweight)

    for (u,v,c) in g.edges:
        if u.minweight + c < v.minweight:
            throw "Negative cycle detected"
```

- Can find the shortest distance in a graph with negative weights
- Detects and throws an error if there are any negative cycles in the graph
- Uses the same rule as Dijkstra's algorithm

$$\text{minweight from s to t} \leq \text{minweight from s to u} + \text{weight}(u \to v)$$

- This has to be applied a V times to find the shortest path
- The algorithm therefore has a complexity of $O(EV)$

Theorem. *The algorithm correctly solves the problem statement. In the case (i) it terminates successfully or (ii) it throws an error*

Proof. Let $w(v)$ be the true minimum weight along all paths from s to v, with $w(v) = -\infty$ if no path exists

Consider case (i), that is assume that there are no negative weight cycles in the graph.

Consider some vertex, v, the shortest path from s to v can not, by assumption, contain any negative cycles, nor can it contain any non-negative cycles, as these could be removed and the path shortened. Consequently the path has at most $V - 1$ edges, let this shortest path be

$$s = v_0 \to v_1 \to \cdots \to v_k = v$$

during the first pass of the algorithm consider the edge $v_0 \to v_1$, this will give

$$\begin{aligned} v_1.\text{minweigth} &= v_0 + \text{weight}(v_0 \to v_1) \\ &= w(v_1) \end{aligned}$$

on the second pass, we will achieve

$$v_2.\text{minweight} = w(v_2)$$

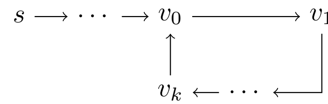
and so on, on the ith pass,

$$v_i.\text{minweight} = w(v_i)$$

After $V-1$ iterations, it has found $v.\text{minweight} = w(v)$ for all v

The test on line 11 must fail for all vertices because overwise we could construct a shorter path from s to v

Consider case (ii), that is assume that there is a negative cycle in the graph



where $\text{weight}(v_0 \to v_1) + \cdots + \text{weight}(v_k \to v_0) < 0$

If the algorithm didn't throw an error then all these edges passed the test on line 11

$$\begin{aligned}
v_0.\text{minweight} + \text{weight}(v_0 \to v_1) &\geq v_1.\text{minweight} \\
v_1.\text{minweight} + \text{weight}(v_1 \to v_2) &\geq v_2.\text{minweight} \\
&\vdots \\
v_k.\text{minweight} + \text{weight}(v_k \to v_0) &\geq v_0.\text{minweight}
\end{aligned}$$

putting these equations together,

$$v_0.\text{minweight} + \text{weight}(v_0 \to v_1) + \cdots + \text{weight}(v_k \to v_0) \geq v_0.\text{minweight}$$

thus the cycle has weight ≥ 0, a contradiction, and therefore one of the edges must fail the test $\square$

5.6 Johnson's Algorithm

- Used to find the shortest paths between all pairs of vertices in a graph which can have negative weights
- If you add some constant to each vertex in a graph the minimum path is not preserved (unless all paths have the same number of vertices)
- Instead assign some weight, d, to each vertex, and redefine the vertex weights as follows

$$w'(u \to v) = d_u + w(u \to v) - d_v$$

consider an arbitrary path P from a to b. with length L, the new length, L' is given by

$$\begin{aligned}
L' &= \sum_{(u,v)\in P} d_u + w(u \to v) - d_v \\
&= d_a + \left(\sum_{(u,v)\in P} w(u \to v) \right) - d_b \\
&= d_a + L - d_b
\end{aligned}$$

this is independent of the path taken, that is the length of all paths from a to b are altered by the same amount, and as a result the minimum path is unchanged.

- Create another vertex, s, outside the graph and connect this to every other vertex in the graph with a vertex of length zero
- Run Bellman-Ford starting at s, this will find the lengths of the minimum paths from s to each vertex (remember negative weighted vertices, so won't necessarily be zero). These lengths become the vertices's weights

- Calculate the new weights for each vertex
- Run Dijkstra's algorithm from each vertex to give the shortest paths between all pairs of vertices
- To find the actual length of each path

$$d(u \to v) = d'(u \to v) - d_u + d_v$$

- The complexity of each stage, gives a total complexity of $O(V^2\, lg\, V + VE)$
  1. Running Bellman-Ford once - $O(EV)$
  2. Running Dijkstra's from each vertex - $O(V^2\, lg\, V + VE)$
  3. Doing clean up on the results (calculating real weights) - $O(V^2)$

Theorem. *The augmented edge weights, w', are non-negative for all edges*

Proof. Assume $w'(u \to v) < 0$
The augmented weights are defined as follows

$$\begin{aligned} w'(u \to v) &= d_u + w(u \to v) - d_v \\ \implies d_u + w(u \to v) - d_v &< 0 \\ \implies d_u + w(u \to v) &< d_v \end{aligned}$$

This implies there exists a shorter path from s to u, a contradiction, therefore the augmented weights must be non-negative for all edges. □

5.7 All-pairs Shortest Paths with Matrices

- Let W be a $V \times V$ matrix, defined by

$$W_{ij} = \begin{cases} 0, & i = j; \\ \text{weight}(i \to j), & \text{if there exists an edge } i \to j; \\ \infty, & \text{otherwise}; \end{cases}$$

- Defined the matrix $M^{(n)}$, such that

$$M_{ij}^{(n)} = \text{minweight from } i \text{ to } j\text{, along all paths with } \leq n \text{ edges}$$

- This can be calculated by a method analogous to matrix multiplication, where $a \wedge b \equiv \min(a, b)$

$$M_{ij}^{(n)} = M_{ij}^{(n-1)} \wedge \left[(M_{i1}^{(n-1)} + W_{1j}) \wedge (M_{i1}^{(n-1)} + W_{1j}) \wedge \cdots \wedge (M_{i|V|}^{(n-1)} + W_{|V|j}) \right]$$

  By definition, $W_{jj} = 0$, therefore $M_{ij}^{(n-1)} = M_{ij}^{(n-1)} + w_{jj}$

$$= (M_{i1}^{(n-1)} + W_{1j}) \wedge (M_{i1}^{(n-1)} + W_{1j}) \wedge \cdots \wedge (M_{i|V|}^{(n-1)} + W_{|V|j})$$

- This is matrix multiplication, but with scalar multiplication replaced with addition and addition replaced with finding the minimum ($\wedge$). This operation is symbolised by $\otimes$
- The operation $\otimes$ has a complexity of $O(V^3)$
- As in Bellman-Ford
  1. Compute $M^{(V-1)}$ and $M^{(V)}$

2. If they are different, the graph has negative-weight cycles
3. If they are equal, these are the minimum weights

- $M^{(V)}$ can be found in $log\,V$ operations, by using the following rule

$$M^{(2a)} = M^{(a)} \otimes M^{(a)}$$

- If there are no negative cycles, for some $x \geq V$,

$$M^{(V)} = M^{(x)}$$

- Using the above trick, the complexity of the algorithm is $O(V^3\,lg\,V)$

5.8 Prim's Algorithm

```
def prim(g, s):
    for v in g.vertices:
        v.distance = infinity
        v.in_tree = False
    s.come_from = None
    s.distance = 0
    toExplore = PriorityQueue([s], lambda v : v.distance)

    while not toExplore.isEmpty():
        v = toExplore.popmin()
        v.in_tree = True

        for (w, edgeweight) in v.neighbours:
            if w.in_tree or edgeweight > w.distance: contine
            w.distance = edgeweight
            w.come_from = v
            if w in toExplore:
                toExplore.decreasekey(w)
            else:
                toExplore.push(w)
```

- The **minimum spanning tree** is a tree that connect all vertices in the graph, and has minimum weight among all spanning trees
  - Where the weight of a tree is the sum of all edge weights in the tree
- A **cut** of a graph is an assignment of its vertices into two non-empty sets, an edge crosses the cut if its two ends are in different sets
- Uses a greedy method
- Procecdure
  1. Pick an arbitrary vertex to be the start of the tree
  2. The frontier is the set of vertices which aren't yet in the tree, but are neighbours of some vertex in the tree
  3. Pick the edge from the tree to the frontier with the lowest weight
  4. Repeat until the tree spans the entire graph
- This is just a slight alteration of Dijkstra's algorithm and therefore it has the same complexity $O(E+V\,lg\,V)$

  Lemma. *Let f be a subset of the edges of an MST of some graph g, and consider a cut of the vertices of g such that no edge in f crosses the cut. Let e be the minimum weight edge that crosses the cut. Then, $f \cup e$ is also a subset of the MST.*

Proof. Let $\bar{f}$ be the minimum spanning tree that f is a subset of.

Assume $e \notin \bar{f}$, and let $e = v \to u$

Now consider a path from v to u in $\bar{f}$, it must use some edge, e', to cross the cut.

$$e.\text{weight} \leq e'.\text{weight}$$

Now consider the graph f', with all edges in $\bar{f}$, but with e' replaced with e

$$\text{weight}(f') \leq \text{weight}(\bar{f})$$

it can easily be shown that f' is a spanning tree □

5.9 Kruskal's Algorithm

```
1  def kruskal(g):
2      tree_edges = []
3      partition = DisjointSet()
4      for v in g.vertices:
5          partition.addsingleton(v)
6      edges = sorted(g.edges, sortkey = lambda u, v, edgeweight: wdgeweight)
7
8      for (u, v, edgeweight) in edges:
9          p = partion.getsetwith(u)
10         q = partion.getsetwith(v)
11         if p == q: continue
12         tree_edges.append((u,v))
13         partition.merge(p, q)
14     return tree_edges
```

- Kruskal's algorithm builds up the MST by agglomerating smaller subtrees together.
- At each stage the algorithm joins up the two fragments that have the lowest-weight edge between them
- The algorithm has a complexity of $O(E\, lg\, V)$
  - Cost of $O(E\, lg\, E)$ to sort edges
  - All disjoint set operations are O(1), giving $O(V + 3E) = O(E)$ for the rest
- The maximum number of edges in a graph is $\frac{V(V-1)}{2}$, therefore $lg\, E = O(lg\, V)$
- Correctness
  - When merging p and q, consider the cut of g, into sets p and not-p
  - The algorithm then chooses the minimum weighted edge across this cut
  - The Prim's Lemma applies and the algorithm produces a minimum spanning tree

5.10 Ford-Fulkerson Algorithm

```
1  def ford_fulkerson(g, s, t):
2      for u→v in g.edges:
3          f(u→v) = 0
4      while true:
5          # Find an augmenting path
6          S = set([s]) # Set of vertices to which flow can be increased
7          while there are vertices v ∈ S and u ∉ S with f(v→u)<c(v→u) or f(u → v) > 0:
8              S.add(u)
9          if t not in S: break
10         pick any path p from s to t made up of vertex pairs discovered in line 7
11         δ = ∞
12
```

```
        # Finding the bottleneck in the path
        for each edge(v_i, v_{i+1}) along p:
            if v_i →v_{i+1} is an edge in g:
                # Forward edge, find the smallest excess capacity in the path
                δ = min(c(v_i →v_{i+1}) − c(v_i →v_{i+1}), δ)
            else:
                # Backward edge, find the smallest capacity that can be augmented, as above
                δ = min(f(v_{i+1} →v_i, δ)

        # Augment the path, to remove that bottleneck
        for each edge(v_i, v_{i+1})  along p:
            if v_i →v_{i+1} is an edge in the graph:
                # Forward edge, add the extra capacity
                f(v_i →v_{i+1}) = f(v_i →v_{i+1}) + δ
            else:
                # Backward edge, subtract capacity
                f(v_i →v_{i+1}) = f(v_i →v_{i+1}) − δ
```

- A graph with weighted edges representing capacity, a source, s, and a sink, t.
  - Capacities represented by $c(u \to v)$, with $c(u \to v) \geq 0$
- A flow is a set of edge weights, $f(u \to v)$, such that $0 \geq f(u \to v) \geq c(u \to v)$
- Flows maintain the flow property, in all vertices other than s and t, flow is conserved, ie. as much stuff comes in as goes out

$$\forall v \in V \backslash \{s,t\}. \left(\sum_{u:u \to v} f(u \to v) = \sum_{w:v \to w} f(v \to w) \right)$$

- A naive greedy algorithm fails
- Instead one must find an augmenting path, that is one that satisfies one of the following
  1. Non-full forward edge
  2. Non-empty backwards edge
- General algorithm

```
def ford_fulkerson(g, s, t):
    initialise all flows to zero
    while an augmenting path can be found:
        compute the bottleneck excess capacity
        Augment each edge by this amount

```

- Each edge must remain in equilibrium, add to forward edge, subtract from backward, because
  - Input to backwards edge start node stays constant
  - But input of backwards edge's end vertex increases
  - Output of start vertex also increases
- Algorithm's complexity - $O(Ef)$
  - Find path in $O(E)$ time
  - Each time increase flow by at least one
  - If flow large, large running time
- If no edge exists between two vertices then the flow between those vertices is 0
- The net flow is the total flow out of the source

  Lemma. *The net flow out of the source is equal to the net flow into the sink*

Proof.

$$\begin{aligned}\text{volume of flow} &= \sum_{v\in V} f(s\to v) - \sum_{v\in V} f(v\to s)\\ &= \sum_{v\in V} f(s\to v) - \sum_{v\in V} f(v\to s) + \sum_{w\in V\setminus\{s,t\}}\left[\sum_{v\in V} f(w\to v) - \sum_{v\in V} f(v\to w)\right]\end{aligned}$$

Flow is conserved for all vertices except s and t

$$\begin{aligned}&= \sum_{v\in V} f(v\to t) - \sum_{v\in V} f(t\to v)\\ &= \text{Net flow into } t\end{aligned}$$

□

Theorem. *If all capacities are integers then the algorithm terminates, and the resulting flow in each edge is an interger*

Proof. Initially the flow is an integer, namely 0. We have integer capacities, and as a result δ the minimum difference between the flow and capacities is a positive integer. All flows are then augmented by this integer, yielding an integer flow. As a result of δ always being a positive integer the maximum flow is always increased by an integer amount. The value of the flow can never exceed the sum all capacities, so the algorithm must terminate. □

- A cut partitions the graph into two sets, $(S,\ S^c)$, with $s \in S$, the $t \in S^c$
- The capacity of a cut is the sum of edge capacities from one set to the other

Lemma. *For any flow and any cut, the value of the flow is less than or equal to the capacity of the cut*

Proof.

Let f be the flow and (S, S^C) be a cut

$$\begin{aligned}\text{volume of flow} &= \sum_{u\in V} f(s\to u) - \sum_{u\in V} f(u\to s)\\ &= \sum_{w\in S}\left(\sum_{u\in V} f(v\to u) - \sum_{u\in V} f(u\to v)\right)\end{aligned}$$

Split the set of vertices V into the sets S and S^c

$$= \sum_{v\in S, u\in S} f(s\to u) - \sum_{v\in S, u\in S} f(u\to s) + \sum_{v\in S, u\notin S} f(s\to u) - \sum_{v\in S, u\notin S} f(u\to s)$$

The first two sums are identical and cancel

$$= \sum_{v\in S, u\notin S} f(s\to u) - \sum_{v\in S, u\notin S} f(u\to s)$$

Because the $f \geq 0$

$$\leq \sum_{v\in S, u\notin S} f(s\to u) \tag{1}$$

Because the flow is always less than the capacity

$$\leq \sum_{v \in S, u \notin S} c(s \to u) \tag{2}$$
$$\leq \text{capacity of cut}$$

□

Theorem. *If the algorithm terminates, then the value of the flow it has found is equal to the size of the cut found in lines 6 and 8.*

Proof. Let f be the flow found and (S, S^c) be the cut it produced by line 7's condition.

$f(u \to v) = 0$ for all $v \in S,\ u \notin S$, so inequality (1) is equality.

By the same condition, $f(u \to v) = c(u \to v)$ for all $v \in S,\ u \notin S$, so inequality (2) is equality.

Thus value of f = capacity of (S, S^c) □

5.11 Matchings

- A Bipartite graph is one in which the vertices are split into two sets, and all edges have one end in one set and the other in the other set
- A matching of a bipartite graph is a selection of some or all of a graph's vertices, such that no vertex is connect to more than one edge in this selection
- A maximum matching is a matching with the largest possible number of edges, it does not have to be unique
- A good way to find a maximum matching is to turn it into a maximum flow problem
  1. Add a source with edges to each left-hand vertices
  2. Add a sink with edges from all right-hand vertices
  3. Turn the bipartite graphs original edges into directed edges from left to right
  4. Give all edges capacity 1
- Because given integer capacities a max flow will always have integer flows, the flow must be 0 or 1
  - All edges with a flow of one are part of the matching

Theorem. *The matching obtained in this way is a maximum matching*

1. Any matching can be translated into a flow
   - It is non-negative
   - The definition of a matching implies the flow conservation equation
2. The size of the matching (number of selected edges) is equal to the value of the flow
3. If there were a larger matching than that obtained by Ford-Fulkerson, it would correspond to a larger flow which cannot be.

6 Geometric Algorithms

6.1 Line Segment Intersection

- First test which side of a line a point q lies on
  - For a line that passes through the origin and a point p, (p_x, p_y)
  - Let the point p^T be be a point on the line made by rotating the line containing p by 90° anticlockwise
  - p^T is given by $(-p_y, p_x)$
  - Take the dot product of the point q and p^T

$$\begin{aligned} p^T \cdot q > 0 &: \quad \text{q is on the left, as you travel along the line in the direction } 0 \to p \\ p^T \cdot q = 0 &: \quad \text{q lies on the line} \\ p^T \cdot q < 0 &: \quad \text{q is on the right} \end{aligned}$$

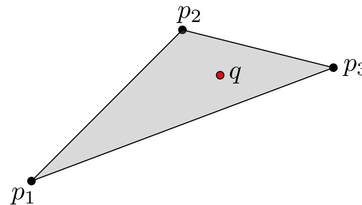
  - Gives a test to decide if two line segments $r - s$ and $t - u$ intersect
    1. If t and u are both on the same side of the line in the direction $r \to s$, then the two lines don't intersect
       * i.e. $(s-r)^T \cdot (t-r)$ and $(s-r)^T \cdot (u-r)$ have the same sign
    2. If r and s are both on the same side of the extension $t \to u$ then the lines don't intersect
    3. Otherwise they do intersect

6.2 Javis's March

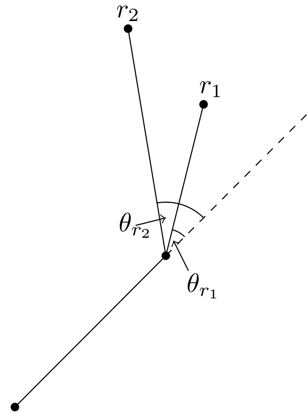
- A convex combination of a collection of points $p_1, \ldots, p_n$, is a vector q, such that

$$q = \alpha_1 p_1 + \cdots + \alpha_n p_n, \qquad \alpha_i \geq 0, \quad \sum_i \alpha_i = 0$$

- A convex hull is the set of all convex combinations
  - For some convex combination q in the convex hull



- **Pick the point r with the smallest θ_r**
- Algorithm
  1. Start with the point p_0, with the lowest y-coordinate, and if there are several, with the largest x-coordinate
  2. Chose the second point p_1 using standard procedure and a left $\to$ right reference line
  3. Draw a line between the two last points to be added
  4. Then add the point with the smallest angle θ_r
     - If two points have the small θ_r then chose the point furthest along the line
  5. Repeat until you return to the start point



- Can be made faster using the fact

$$\theta_{r_1} < \theta_{r_2} \iff r_2 \text{ on left of } \overrightarrow{Or_1}$$

  - Finding the dot product is much faster than measuring angles or trigonometry

- Has a complexity of $O(n\,h)$, where n is the number of points and h is the number of points in the convex hull

6.3 Graham's Scan

- Algorithm
  1. Start with the point p_0, with the lowest y-coordinate, and if there are several, with the largest x-coordinate
  2. Sort all remaining points, in terms of their angle from the positive x-axis
  3. Add p_0 to the stack, as the most recently visited point
  4. Iterate through the sorted list, beginning with the point with the smallest angle
  5. Draw a segment from the most recent point to the next unvisited point
     (a) if the path turns left with respect to the previous one, push point onto stack and continue
     (b) else (including if they are on the same line) pop the most recent point and repeat 5
- The points can be sorted using the fact $\theta_{r_1} < \theta_{r_2} \iff r_2$ on left of $\overrightarrow{Or_1}$, as a comparison
- Has a complexity of $O(n\,lg\,n)$
  - Requires $O(n\,lg\,n)$ to sort the points
  - Requires $O(n)$ to iterate through all points