COMP20007 Workshop Week 7

1	More on the Dijkstra's algorithm: Problem 1
2	The Master Theorem: Problem 2 & 3
3	Closest Pairs: Problem 4
LAB	Understand the lab's graph module and implement some graph algorithms: BFS Dijkstra's Prim's [optional]

Complexity of Dijkstra's Algorithm [graph=Adj. List]

```
function Dijkstra(<V,E,W>,s)
  for each u ∈ V do
    dist[u] ← 0, pred[u] ← nil
  dist[s] ← 0
  build PQ with all (u, dist[u])    #cost= O(n)
  while (PQ is not empty) do
    u ← deletmin(PQ)
    for each (u,v) ∈ E do
      if (dist[u]+w(u,v) < dist[v]) then
        pred[v] ← u
        dist[v] ← dist[u]+w(u,v)
        #also change priority in PQ
```

- *Complexity of Dijkstra's algorithm? Supposing |V|=n, |E|= m*

PQ Implementation	Complexity of PQ operation deletemin	change priority	Complexity of DA (for AdjList graph)
List (array, linked list)	O(n)	O(n)	
Binary Heap	O(log n)	O(log n)	
2-3 or Fibonacci Heap	O(log n)	O(1)	

Complexity of Dijkstra's Algorithm [graph=Adj. List]

Suppose that if PQ has n elem,ents, the complexity of:

- `deletemin(PQ)` is O(f(n))
- `change_priority_of_a_key(PQ, key, new_weight)` is O(g(n)

Step	Big-O Complexity Single line	Combined
`function Dijkstra(<V,E,W>,s)`		
`  for each u ∈ V do`	`O(n)`	
`    dist[u] ← 0, pred[u] ← nil`	`O(1)`	
`  dist[s] ← 0`	`O(1)`	`O( )`
`  build PQ with all (u, dist[u])`	`O(n)`	
`  while (PQ is not empty) do`	`O(?)`	
`    u ← deletmin(PQ)`	**`O(?)`**	
`    for each (u,v) ∈ E do`	`O(?)`	
`      if (dist[u]+w(u,v) < dist[v]) then`	`O(1)`	`O( )`
`        pred[v] ← u`	`O(1)`	
`        dist[v] ← dist[u]+w(u,v)`	`O(1)`	
`        change priority of v in PQ`	**`O(?)`**	

Complexity of DA [graph=Adj. List]: check your answer

```
function Dijkstra(<V,E,W>,s)
  for each u ∈ V do
    dist[u] ← 0, pred[u] ← nil
  dist[s] ← 0
  build PQ with all (u, dist[u])    #cost= O(n)
  while (PQ is not empty) do
    u ← deletmin(PQ)
    for each (u,v) ∈ E do
      if (dist[u]+w(u,v) < dist[v]) then
        pred[v] ← u
        dist[v] ← dist[u]+w(u,v)
        change_weight(PQ, v, dist[v])
```

- *Complexity of Dijkstra's algorithm? Supposing |V|=n, |E|= m*

PQ Implementation	Complexity of PQ operation deletemin | change priority		Complexity of DA (for AdjList graph)
List (array, linked list)	O(n)	O(n)	O((n+m).n)= O(mn)
Binary Heap	O(log n)	O(log n)	O((n+m)log n)= O(mlog n)
2-3 or Fibonacci Heap	O(log n)	O(1)	O(nlogn + m)

More on Dijkstra's Algorithm

- *Dijkstra's Algorithm can't handle negative weight. Why? Give an example.*

```
function Dijkstra(<V,E,W>,s)
  for each u ∈ V do
    dist[u] ← 0, pred[u] ← nil
  dist[s] ← 0
  build PQ from all (u, dist[u])
  while (PQ is not empty) do
    u ← deletmin(PQ)
    for each (u,v) ∈ E do
      if (dist[u]+w(u,v) < dist[v]) then
        pred[v] ← u
        dist[v] ← dist[u]+w(u,v)
        change_weight(PQ, v, dist[v])
```

Class Work: Problem 1

T2: *Dijkstra's algorithm, unmodified, can't handle some graphs with negative edge weights. Your friend has come up with a modified algorithm for finding shortest paths in a graph with negative edge weights:*

1. Find the largest negative edge weight, call this weight **`-w`**.
2. Add **`w`** to the weight of all edges in the graph. Now, all edges have non-negative weights.
3. Run Dijkstra's algorithm on the resulting non-negative-edge-weighted graph.
4. For each path found by Dijkstra's algorithm, compute its true cost by subtracting **`w`** from the weight of each of its edges.

Will your friend's algorithm work? Why? Give an example.

The Master Theorem

If

$$T(n) = aT\left(\frac{n}{b}\right) + \Theta\left(n^d\right)$$

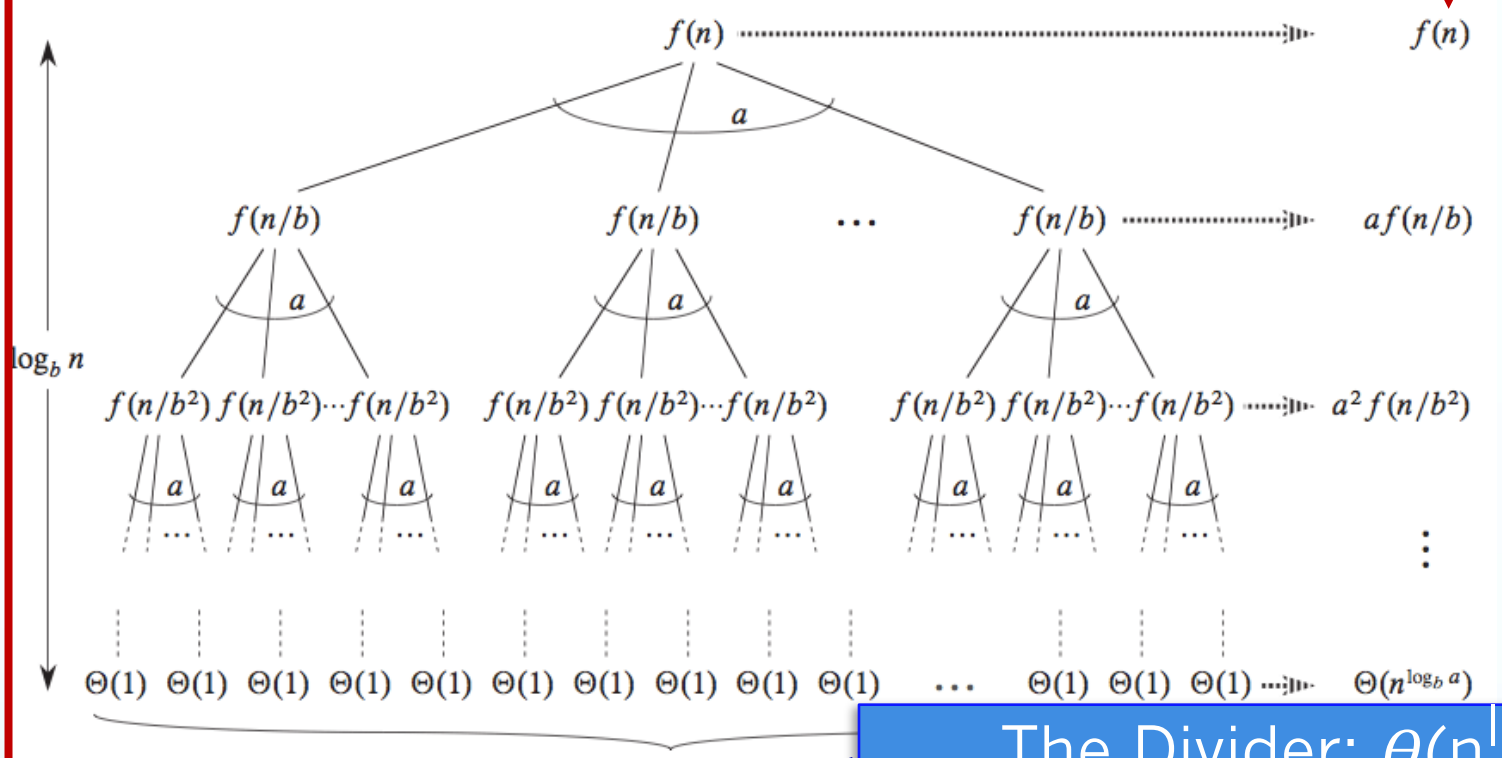
$$T(1) = \Theta(1)$$

where $a \geq 1$, $b > 1$, then

$$T(n) = \begin{cases} \Theta(n^d) & \text{if } a < b^d \\ \Theta(n^d \log n) & \text{if } a = b^d \\ \Theta(n^{\log_b a}) & \text{if } a > b^d \end{cases}$$

Master Theorem

The Conqueror: $f(n)=\theta(n^d)$



The Divider: $\theta(n^{\log_b a})$

$$leaves = a^h = a^{\log_b n} = n^{\log_b a}$$

Total time:

$$n^d + a\,(n/b)^d + a^2\,(n/b^2)^d + \ldots + a^h\,\underbrace{(n/b^h)^d}_{=1}$$

Master Theorem

The Conqueror

The Divider

$$n^d + a\,(n/b)^d + a^2\,(n/b^2)^d + \ldots + n^{\log_b a}$$

$$\log_b n + 1 = \boldsymbol{\theta}(\log n) \text{ members}$$

and the winner is

Winner	Condition	Equivalent condition	Time complexity
Conqueror	$a < b^d$	$\log_b a < d$	$\Theta(n^d)$
Divider	$a > b^d$	$\log_b a > d$	$\Theta(n^{\log_b a})$
none	$a = b^d$	$\log_b a = d$	$\Theta(n^d \log n)$

$$T(n) = aT\left(\frac{n}{b}\right) + \Theta\left(n^d\right)$$

$$T(1) = \Theta(1)$$

$$T(n) = \begin{cases} \Theta(n^d) & \text{if } a < b^d \\ \Theta(n^d \log n) & \text{if } a = b^d \\ \Theta(n^{\log_b a}) & \text{if } a > b^d \end{cases}$$

T1: Find time complexity (big- or big-O) for:

(a) $T(n) = 9T\left(\frac{n}{3}\right) + n^3,\ T(1) = 1$

(b) $T(n) = 64T\left(\frac{n}{4}\right) + n + \log n,\ T(1) = 1$

(c) $T(n) = 2T\left(\frac{n}{2}\right) + n,\ T(1) = 1$

(d) $T(n) = 2T\left(\frac{n}{2}\right),\ T(1) = 1$

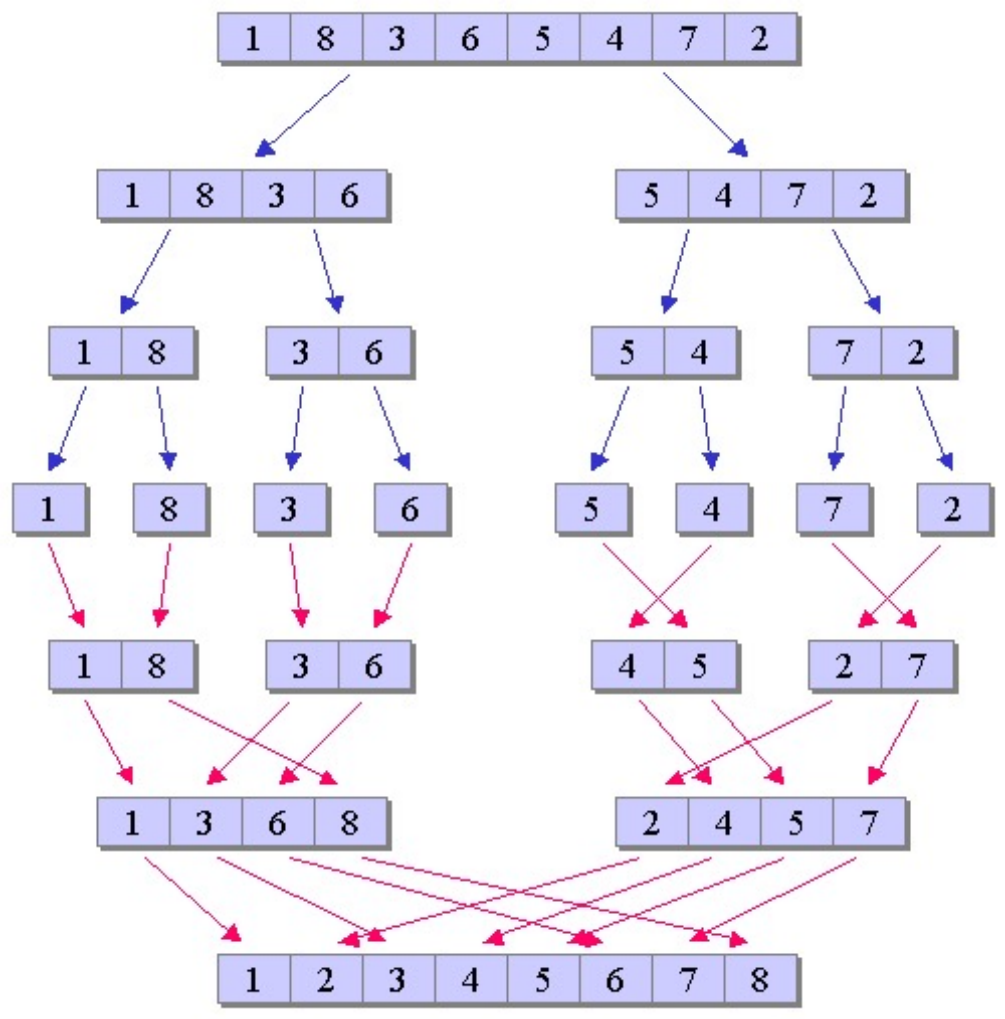
(e) T(n) = 2T(n-1) + 1

T3: Mergesort Time Complexity

Mergesort is a divide-and-conquer sorting algorithm made up of three steps (in the recursive case):

1. Sort the left half of the input (usin mergesort)
2. Sort the right half of the input (using mergesort)
3. Merge the two halves together (using a merge operation)

- Construct a recurrence relation to describe the runtime of mergesort . Explain where each term in the recurrence relation comes from.
- Use the Master Theorem to find the time complexity of Mergesort in Big Theta terms.

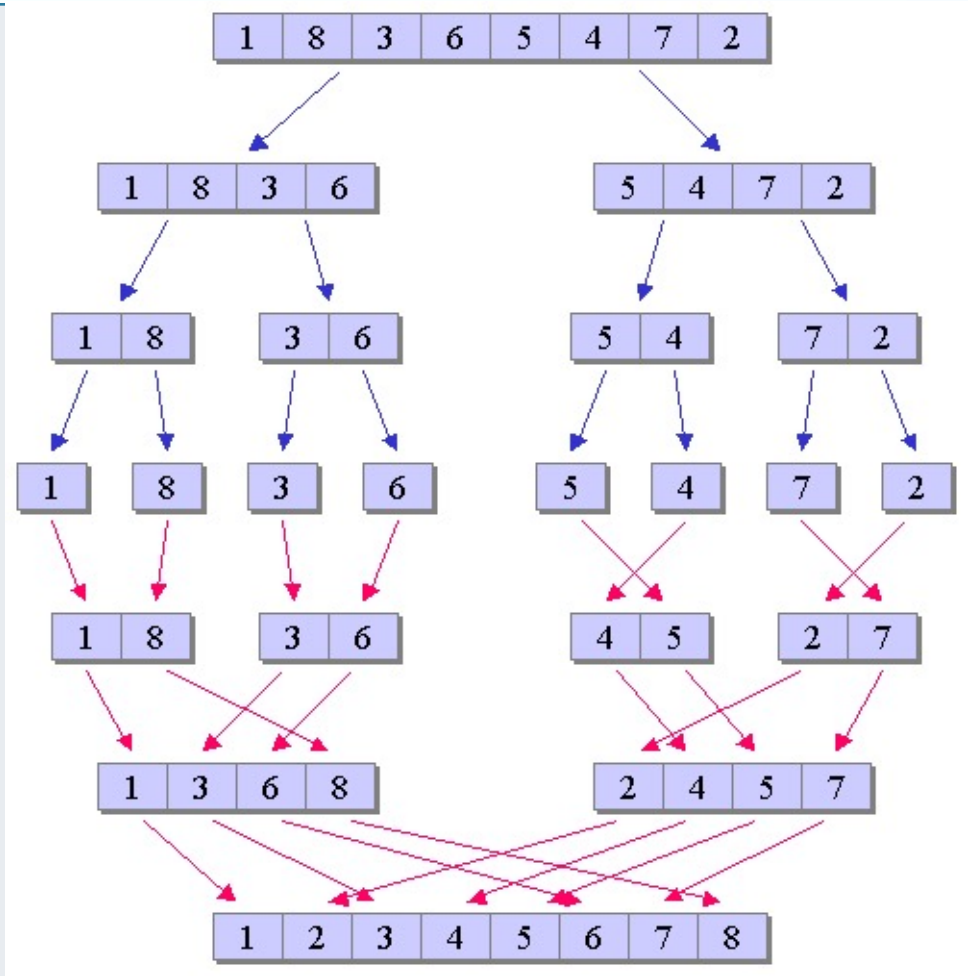


Problem 3: Mergesort Time Complexity

- Construct a recurrence relation to describe the runtime of mergesort . Explain where each term in the recurrence relation comes from.

  T(n)=
  T(1)=

- Use the Master Theorem to find the time complexity of Mergesort in Big-Theta terms.



$$T(n) = aT\left(\frac{n}{b}\right) + \Theta\left(n^d\right)$$

$$T(1) = \Theta(1)$$

$$T(n) = \begin{cases} \Theta(n^d) & \text{if } a < b^d \\ \Theta(n^d \log n) & \text{if } a = b^d \\ \Theta(n^{\log_b a}) & \text{if } a > b^d \end{cases}$$

Problem 4: Closest-pair and element-distinction

Lower bound for the Closest Pairs problem The closest pairs problem takes n points in the plane and computes the Euclidean distance between the closest pair of points.

The algorithm provided in lectures to solve the closest pairs problem applies the divide and conquer strategy and has a time complexity of O(n log n).

The element distinction problem takes as input a collection of n elements and determines whether or not all elements are distinct. It has been proved that if we disallow the usage of a hash table1 then this problem cannot be solved in less than n log n time (i.e., this class of problems is Ω(n log n)).

Describe how we could use the closest pair algorithm from class to solve the element distinction problem (where the input is a collection of floating point numbers), and hence explain why this proves that the closest pair problem must not be able to be solved in less that nlogn time (and is thus Ω(n log n)).

Problem 4: Closest-pair and element-distinction

It has been proved that if we disallow the usage of a hash table1 then the element distinction problem cannot be solved in less than n log n time (i.e., this class of problems is Ω(n log n)).

a) Describe how we could use the closest pair algorithm from class to solve the element distinction problem (where the input is a collection of floating point numbers), and hence **b)** explain why this proves that the closest pair problem must not be able to be solved in less that nlogn time (and is thus Ω(n log n)).

E is a set of numbers, returns YES if all numbers are different

```
function ElemDistinction(E=(e1, e2, …, en))
```

LAB

Download lab_files.zip and unzip. Try:

```
make
./main < graph-01.txt
./main < graph-02.txt
```

Then, read `main.c` to understand the overall logic.

Task 1: Write BFS, return the array of visited order, using supplied DFS as a guide.

- change `main.c` to add a call to, and output for BFS
- explore function `dfs` in `graphalgs.c` and design a similar BFS
- your top level of `bfs` could be almost similar to that of `dfs`
- your `bfs_explore` could be similar to `dfs_explore`, but note that you need to use a queue instead of a recursive function
- you can quickly implement a queue module using the list module
- alternatively, you can directly use a linked list as a queue

LAB: the list module

The list module (`lish.h` and `list.c`).

Read `list.h` to know the list interface (ie. data types & functions), make sure that you know how to:

- declare a list and create an empty list
- insert an element to the start or the end of a list
- remove the first or the last element of a list
- test if a list is empty, if a list contains a specific data
- iterate through the list (and, say, print out each element), using a `ListIterattor` and related function (you can explore `graph.c` to see examples of how to use `ListIterator`)
- understand why `ListIterator` is useful

LAB: building a queue module

Suppose we want to buils a queue module (`queue.h` and `queue.c`) in a least effort manner, using the list module.

queue.h	Notes

`#ifndef _QUEUE_H_`
`#define _QUEUE_H_`
`#include "list.h"`
`typedef List Queue;`
 	
`Queue *new_queue();`
`void free_queue(Queue *q);`
 	
`void enqueue(Queue *q, int data);`
`int dequeue(Queue *q);`
`int queue_is_empty(Queue *q);`
`#endif`
	Any `.h` file needs to have the first 2 and the last 1 red lines. Why?
	
	Using these 3 lines in `.h` files is a good convention.

The `queue.c` should be simple, for example:

queue.c	Notes

`void enqueue(Queue *q, int data) {`
`  list_add_end(q, data);`
`}`
	with the call `list_add_end`, `q` will be auto-cast to type `List *`

LAB: Task 2 – implement Dijkstra's

We need priority queue! Examine `priorityqueue.h` to know the interface (the data type, the functions). Then

- **Step 2.1:** Write `diskstra(…)` to find shortest path from a source s, and add a function call in `main()`. At first, `dikstra(…)` just builds up arrays `dist[]` and `pred[]`. Note that:
  - it would be convenient to have an array `visited[]` so that `visited[u]= true` iif the the shortest path for u already found,
  - when inserting to `PQ`, you should insert both node `u` and distance `dist[u]`
  - when modifying `dist[v]`, remember to modify the corresponding priority in the `PQ`. What is the complexity of that `priority_queue_update` ?
- **Step 2.2:** What should `dijkstra()` output? Modify your function to store the paths from the start node to each other node in an array of linked lists or an array of arrays. Which data structure will be easier to work with?