COMP20007 Workshop Week 7

1	Discussion: the Dijkstra's algorithm & Qu
2	The Master Theorem: Q7.2
3	Closest Pairs: Q7.3
4	Horspool's Algorithm, Q 7.4-7.6
	Understand the lab's graph module and implement some graph algorithms:
LAB	BFS
	Dijkstra's
	Prim's

ATTN SVP!

Interesting, Important, & Hard Topic

Next Workshop:

Dynamic Programming.

Remember at least to attend/watch lectures before the workshop.

Group Work: Q7.0 & Q7.1 on the Dijkstra's Algo

Q7.0:

- *Time complexity of DA if **using adjacency lists** = ?*
- *Why DA doesn't work with negative weights? Give an example.*

Q7.1: *DA, 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.

```
function Dijkstra(G=<V,E>,s)
  for each u ∈ V
    dist[u] ← 0, prev[u] ← nil
  dist[s] ← 0
  build PQ with all (u, dist[u])
  while PQ ≠ ∅
    u ← deletmin(PQ)
    for each (u,v) ∈ E
      if dist[u]+w(u,v) < dist[v]
        dist[v] ← dist[u]+w(u,v)
        prev[v] ← u
        update dist[v] in PQ
```

Notes

good implementation of PQ offers:

- O(n) for building PQ of n elems
- O(log n) for delemin & update

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

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

Note:

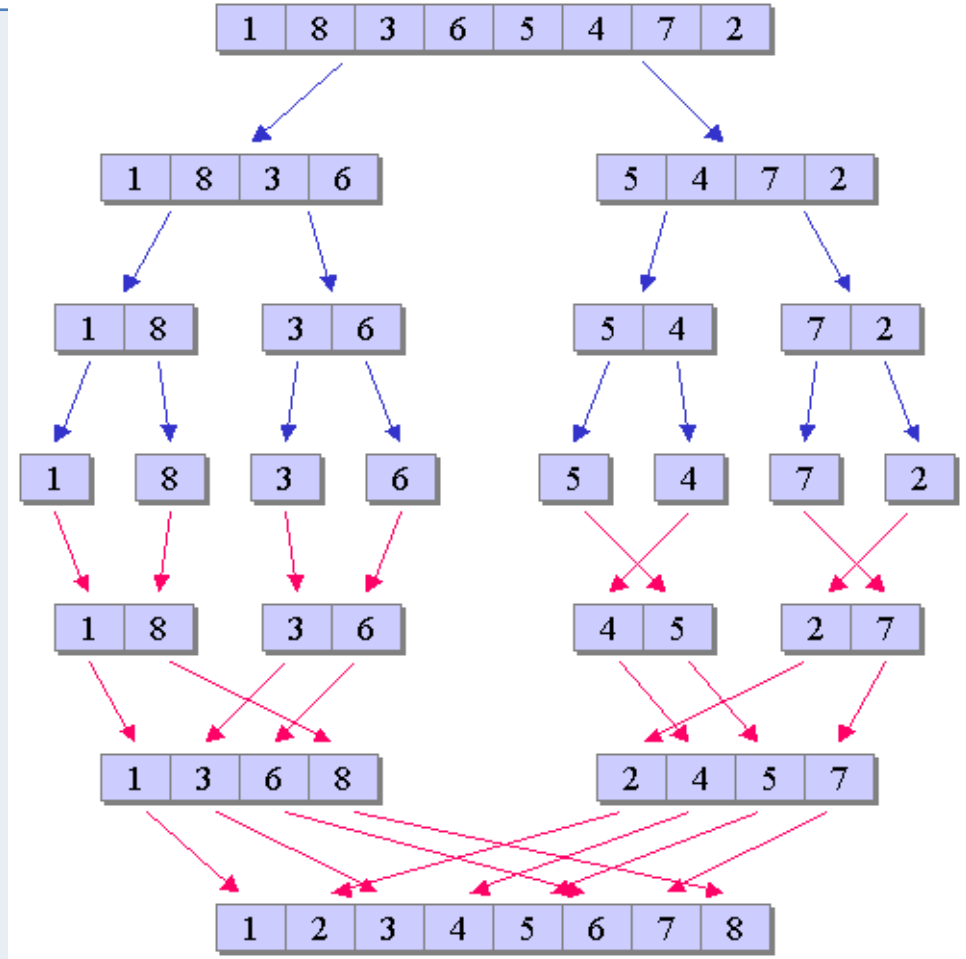
- Similar results hold for O and **Ω**
- Also OK if $T(1) = 0$

Q7.2: 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)=

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



The Master Theorem:

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

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

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

Q7.2: Find time complexity

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

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

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

we can compare $\log_b a$ with d

(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) $T(n) = 2T(n-1) + 1,\ T(1)=1$

(a) a=9 b= 3, d= 3 log_b a= 2<d → T(n^3)

(b) a= 64, b= 4 log= 3 >, d= 1 T(n^3)

(c) T(

(d) a=b= 2 log> T(n) d= -inf T( ) O()

(e) a= 2, b= T() O()

Q7.3: 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 table 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)).

Q7.4: Closest-pair and element-distinction

It has been proved that if we disallow the usage of a hash table 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)).

- 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

```
function ElemDistinction(?)
```

- 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)).

```
?
```

String Searching

Input:

A (long) text `T[0..n-1]`. Example: `T`= “SHE SELLS SEA SHELLS”, with `n`=20

A (short) pattern `P[0..m-1]`. Example: `P`=“HELL”, `m`=4.

Output:

index `i` such that `T[i..i+m-1]=P[0..m-1],` or NOTFOUND

Algorithms:

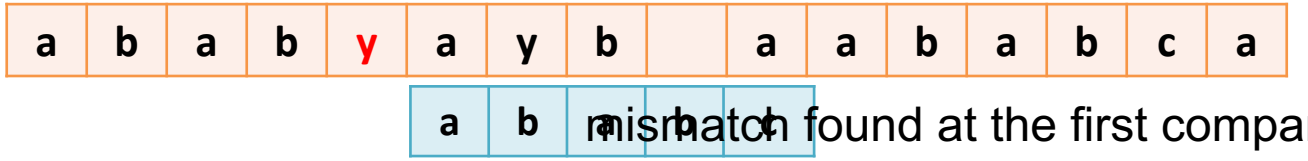
Naïve: brute force, complexity O(nm) (max= (n-m+1)*m character comparison)

- shift pattern left to right on the text, 1 position each time
- compare pattern with text from left to right

Horspool’s: also O(mn) but practically fast:

- shift pattern left to right on the text, *at least* 1 position each time
- compare pattern with text **from right to left**

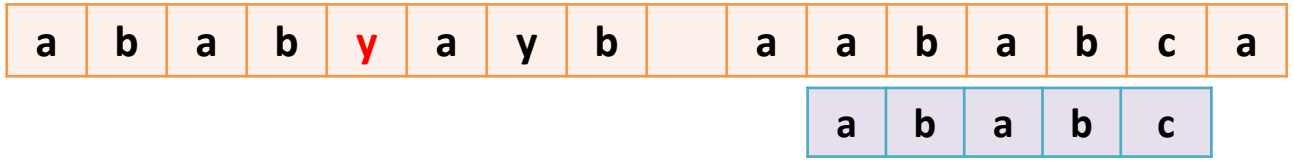
How to run Horspool's *manually*



mismatch found at the first comparison
no matter where mismatch happens, the shift is totally decided by the rightmost examined char of T, **y**

SHIFT m because **y** not in P

Shift until having the first match of character on P with that rightmost **y** (here, no match found)



The number of positions to shift totally depend on the pattern `P` and can be easily pre-computed!

Horspool's Algorithm Review

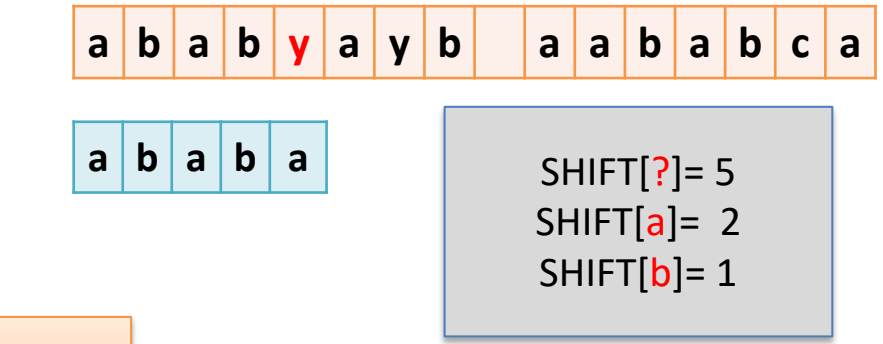
The task: Searching for a pattern `P` (such as "ababa" that has length m=5) in a text `T` (such as "ababyayb aababca", having length n=16).

Stage 1: build `SHIFT[x]` for all `x`, by:

1. set `SHIFT[x]= m` for all `x`, then
2. for each `x` in `P`, *except for the last one*: `SHIFT(x)=` distance from the last appearance of `x` to the end of `P`

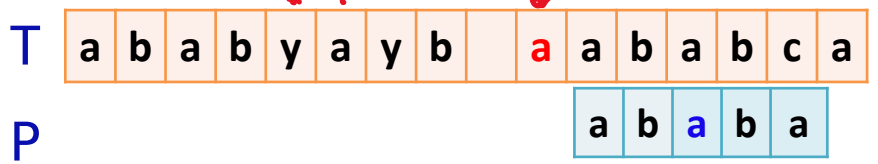
Stage 2: searching, by first set `i=m–1`, then

1. slide `P` so that `P[m–1]` aligned with `T[i];`
2. compare characters *backwardly* from the last character of `P` until the start or until finding a mismatch:
3. if no mismatch found: return solution which is `i–m+1`
4. otherwise, set `i= i+ SHIFT[c]`, where `c=T[i]`, back to step 1



`i= m–1` : making the first alignment

`i= i+SHIFT['y']` : shift pattern to the left



Horspool's Algorithm Review

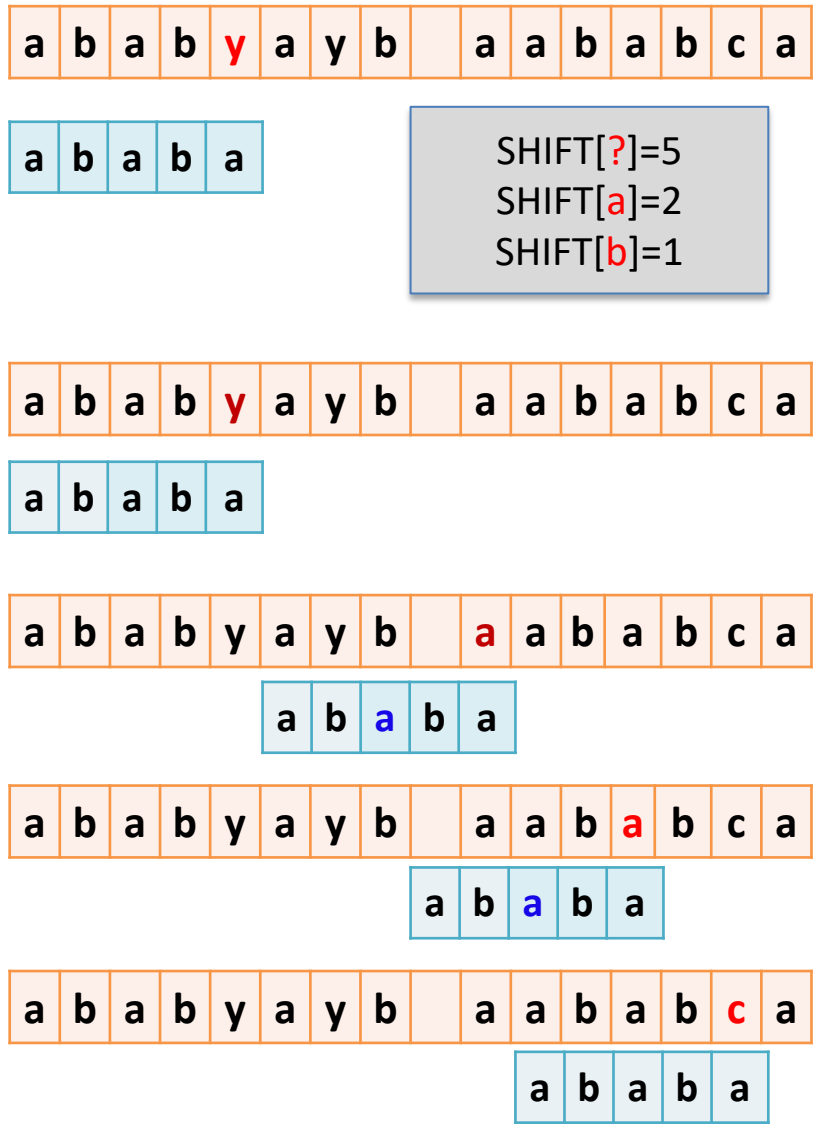
The task: Searching for a pattern `P` (such as "ababa" that has length m=5) in a text `T` (such as "ababyayb aababca", having length n=16).

Stage 1: build `SHIFT[x]` for all `x`, by:

1. set `SHIFT[x]= m` for all `x`, then
2. for each `x` in `P`, *except for the last one*: `SHIFT(x)=` distance from the last appearance of `x` to the end of `P`

Stage 2: searching, by first set `i=m−1`, then

1. set `c= T[i]`, align `P` with `T` so that `P[m−1]` aligned with `T[i];`
2. compare characters *backwardly* from the last character of `P` until the start or until finding a mismatch:
3. if no mismatch found: return solution which is `i−m+1`
4. otherwise, set `i= i+ SHIFT[c]`, back to step 1



Horspool's Algorithm: Group Work

Q7.4: Use Horspool's algorithm to search for the pattern GORE in the string ALGORITHM

```
ALGORITHM
   GORE
```

Q7.5: How many character comparisons will be made by Hor-spool's algorithm in searching for each of the following patterns it the binary text of one million zeros?

```
   (a) 01001      (b) 00010        (c) 01111
T = 000000000..0
a)   01001  S(0)=1   S(1)=3
b) 00010 S(0)= 2, S(1)= 1
c)
```

Q7.6 [possible homework]: Using Horspool's method to search in a text of length n for a pattern of length m, what does a worst-case example look like?

Horspool's Algorithm: Review your answers

Q7.4: Use Horspool's algorithm to search for the pattern GORE in the string ALGORITHM

ALGORITHM **GORE**

S(*)= 4, S(R)=1, S(O)=2, S(G)= 3, total: 2 alignments, 2 char comparisons

Q7.5: How many character comparisons will be made by Hor-spool's algorithm in searching for each of the following patterns it the binary text of one million zeros?

T= 000000000..0 **P=** (a) 01001 (b) 00010 (c) 01111

a) S(0)=1, S(1)=3 char comparisons= 999,996

b) S(0)=2, S(1)=1 char comparisons= 999,996

b) S(0)=4, S(1)=1 char comparisons= 249,999

Q7.6 [possible homework]: Using Horspool's method to search in a text of length n for a pattern of length m, what does a worst-case example look like?

something like: T=

LAB

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

ATTN SVP!

Interesting, Important, & Hard Topic

Next Workshop:

Dynamic Programming.

Remember at least to attend/watch lectures before the workshop.

- 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`, but using a queue instead of recursion
- for a queue
    - you can quickly implement a queue module using the list module, or
    - directly use a linked list as a queue, but
    - please do not used the priority 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 build 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 red 3 lines in `.h` files is a good convention.

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

queue.c	Notes

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

LAB: Task 2 – implement Prim's

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

Step 2.1: Write `prim(…)` and add a function call in `main()`. At first, `prim(…)` just builds up arrays `cost[]` and `prev[]`. Note that:

- it would be convenient to have an array `visited[]` so that `visited[u]= true` iif the `u` already added to the MST,
- when inserting to `PQ`, you should insert both node `u` and distance `cost[u]`
- when modifying `cost[v]`, remember to modify the corresponding priority in the `PQ`. What is the complexity of that `priority_queue_update` ?

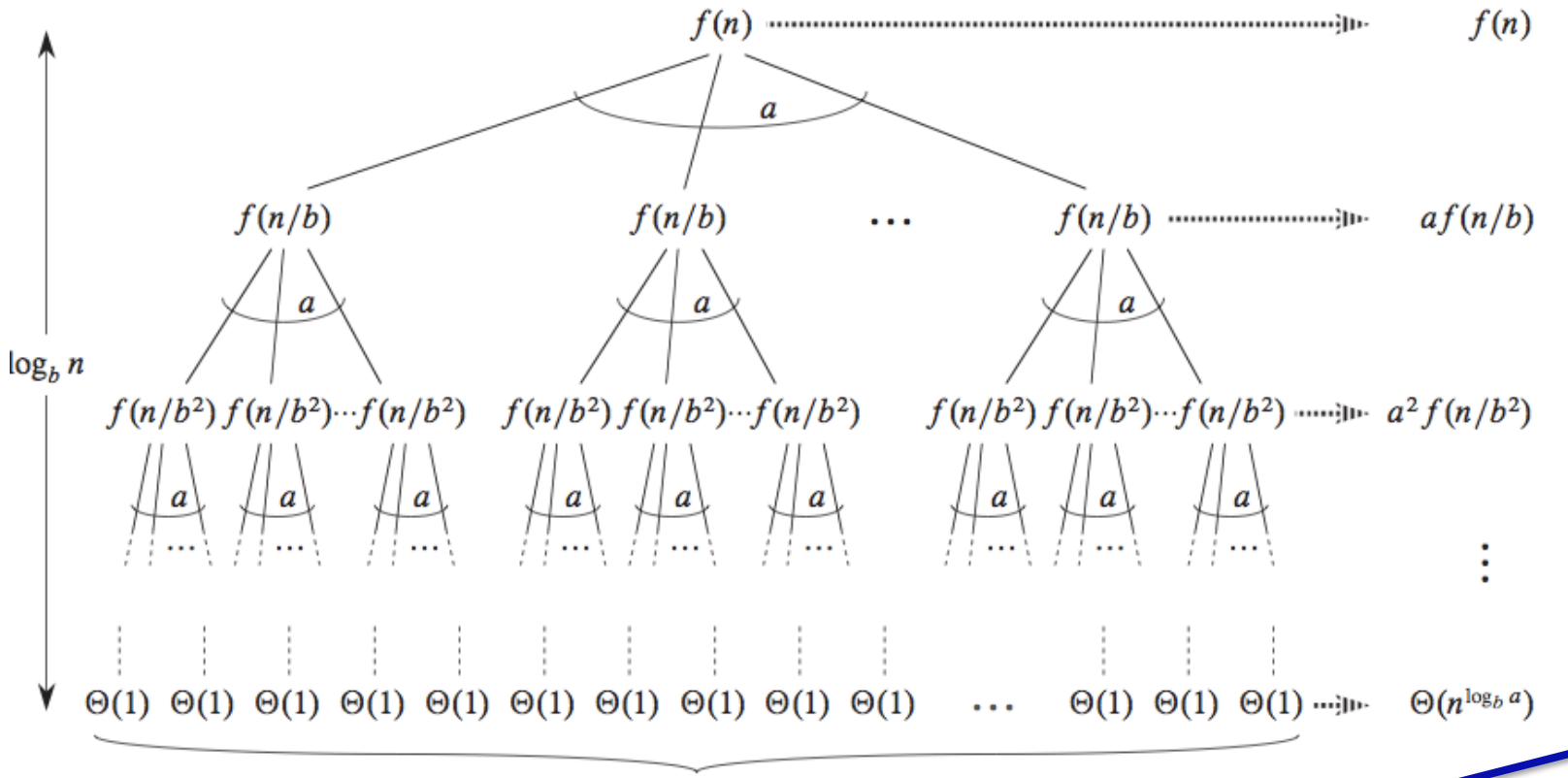
Step 2.2: What should `prim()` output?

Additional Slides

Master Theorem

$T(n)= aT(n/b)+ f(n), \quad f(n)= \theta(n^d)$

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



n^d

$+$

$a(n/b)^d$

$+$

$a^2(n/b^2)^d$

$+ \ \ldots$

$+$

$a^h(n/b^h)^d, \ n/b^h=1$

The Divider: $\theta(n\log_b a)$

Notes: number of *leaf nodes* = $a^h = a^{\log_b n} = n^{\log_b a}$

Master Theorem

The Conqueror

The Divider

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

$\log_b n + 1 = \Theta(\log n)$ members

and the winner is

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