# COMP3711: Design and Analysis of Algorithms

Tutorial 2

HKUST

# Question 1

Give asymptotic upper bounds for $T(n)$ by recursion tree approach. Make your bounds as tight as possible.

(a)

$$\begin{aligned} T(1) &= 1 \\ T(n) &= T(n/2) + n \qquad \text{if } n > 1 \end{aligned}$$

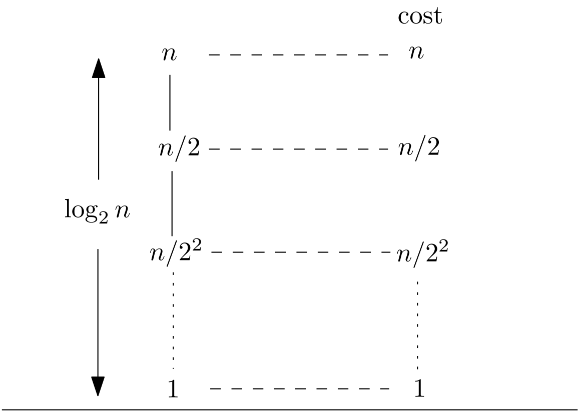
(b)

$$\begin{aligned} T(1) &= T(2) = 1 \\ T(n) &= T(n-2) + 1 \qquad \text{if } n > 2 \end{aligned}$$

(c)

$$\begin{aligned} T(1) &= 1 \\ T(n) &= T(n/3) + n \qquad \text{if } n > 1 \end{aligned}$$

# Solution 1 (a)



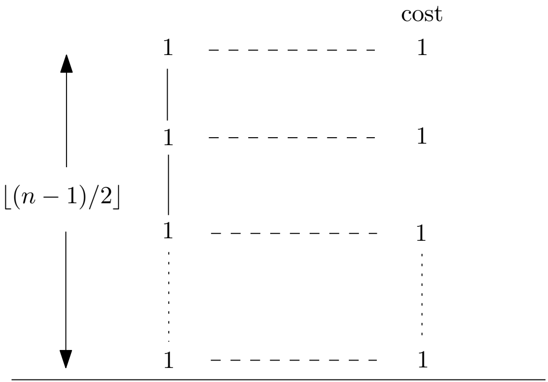

$$\text{total} = n + n/2 + n/2^2 \cdots + 1$$
$$= O(n)$$

# Solution 1 (a)

Set $h = \log_2 n$

$$
\begin{aligned}
T(n) &= n + T(n/2) \\
&= n + n/2 + T(n/2^2) \\
&= n + n/2 + n/2^2 + T(n/2^3) \\
&\dots \\
&= n + n/2 + n/2^2 + \cdots + n/2^{h-2} + n/2^{h-1} + T(n/2^h) \\
&= n(1 + 1/2 + 1/2^2 + \cdots + 1/2^{h-2} + 1/2^{h-1}) + T(n/2^h) \\
&\leq n(1 + 1/2 + 1/2^2 + \cdots + 1/2^{h-1} + \dots) + T(n/2^h) \\
&= 2 \cdot n + T(1) \\
T(n) &= O(n)
\end{aligned}
$$

# Solution 1 (b)

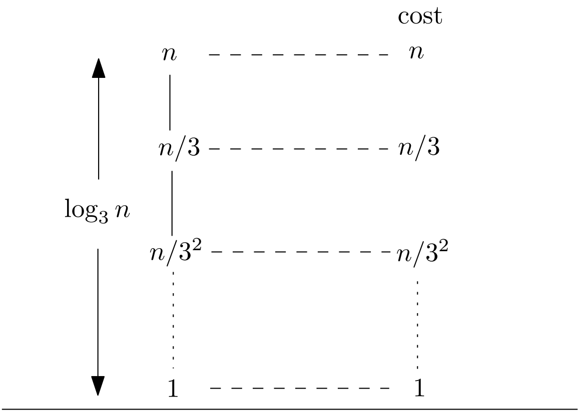



$$\text{total} = 1 + \lfloor (n-1)/2 \rfloor$$
$$= O(n)$$

# Solution 1 (b)

$$
\begin{aligned}
T(n) &= T(n-2)+1 \\
&= T(n-2\cdot 2)+2 \\
&= T(n-3\cdot 2)+3 \\
&\dots \\
&= T(n-\lfloor (n-1)/2 \rfloor \cdot 2)+\lfloor (n-1)/2 \rfloor \\
T(n) &= 1+\lfloor (n-1)/2 \rfloor = \lceil (n/2) \rceil = O(n)
\end{aligned}
$$

# Solution 1 (c)




$$\text{total} = n + n/3 + n/3^2 \cdots + 1$$
$$= O(n)$$

# Solution 1 (c)

Set $h = \log_3 n$

$$
\begin{aligned}
T(n) &= n + T(n/3) \\
&= n + n/3 + T(n/3^2) \\
&= n + n/3 + n/3^2 + T(n/3^3) \\
&\dots \\
&= n + n/3 + n/3^2 + \cdots + n/3^{h-2} + n/3^{h-1} + T(n/3^h) \\
&= n(1 + 1/3 + 1/3^2 + \cdots + 1/3^{h-2} + 1/3^{h-1}) + T(n/3^h) \\
&\leq n(1 + 1/3 + 1/3^2 + \cdots + 1/3^{h-1} + \dots) + T(n/3^h) \\
&= 3n/2 + T(1) \\
T(n) &= O(n)
\end{aligned}
$$

# Question 2

Given a sorted array $A[1..n]$ of $n$ distinct integers (positive or negative), give an algorithm to find the index $i$ such that $A[i] = i$, if such an index exists. If there are many such indices, the algorithm may return any one of them. Solve this problem in $O(\log n)$ time.

# Solution 2

```
INDEX-SEARCH(A, s, t)
    if (s = t)   //  O(1)
        if (A[s] = s)
            return s;
        else
            return -1;
    m ← ⌊(s+t)/2⌋;
    if (A[m] = m)  return m;   //  O(1)
    if (A[m] > m)
        return INDEX-SEARCH(A, s, m);   //  T(⌊n/2⌋)
    else
        return INDEX-SEARCH(A, m + 1, t);   //  T(⌈n/2⌉)
```

# Solution 2

If $A[m] > m$, any $i > m$ will have $A[i] > i$, since the array is sorted and all numbers are distinct. So the latter half of the array cannot possibly contain a desired index. Similarly, if $A[m] < m$, any $i < m$ will have $A[i] < i$. In either case, we can throw away half of the array and recursively solve the problem for the other half. The running time of the algorithm has the recurrence $T(n) = T(n/2) + O(1)$, which solves to $T(n) = O(\log n)$.

# Question 3

Let $A[1..n]$ be an array of $n$ elements. A *majority element* of $A$ is any element occuring more than $n/2$ times (e.g., if $n = 8$, then a majority element should occur at least 5 times). Your task is to design an algorithm that finds a majority element, or reports that no such element exists.

(a) Suppose that you are not allowed to order the elements, the only way you can access the elements is to check whether two elements are equal or not. Design an $O(n \log n)$-time algorithm for this problem.

(b) Design an $O(n)$ algorithm for this problem. Similar to (a), you are still only allowed to use equality tests on the elements.

# Solution 3 (a)

Divide $A$ into two parts $A[1..n/2]$ and $A[n/2+1..n]$. Since a majority element in $A$ must be a majority in at least one of the halves, we recursively find a majority in $A[1..n/2]$ and $A[n/2+1..n]$. If $A[1..n/2]$ returns a majority element $e$, we scan the entire $A$ to count its occurrences. If it's more than $n/2$, we return it. We do the same thing for the majority returned from $A[n/2+1..n]$ if it returns one. If we cannot find a majority after this, we return "no majority exists". The base case is when $n=1$, we simply return the only element as the majority. The running time of the algorithm satisfies $T(n) = 2T(n/2) + O(n)$, which solves to $T(n) = O(n \log n)$.

# Solution 3 (b)

Initially set $e = NULL$ and a counter $c = 0$. Then for $i = 1$ to $n$ we do the following: If $c = 0$, we set $e = A[i]$. If $c > 0$, we check if $e = A[i]$. If so, we increment $c$ by 1; else we decrement $c$ by 1. We claim that in the end, $e$ is the only possible majority if there exists one. Then we scan $A$ again to count the actual number of occurrences of $e$ and decide if it is indeed a majority.