COMP 3711

Tutorial 2

Version of Feb 18, 2019

Question 1

Derive asymptotic upper bounds for $T(n)$.
Make your bounds as tight as possible.

(a) You may assume n is a power of 2.

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

(b)

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

(c) You may assume n is a power of 3.

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

Question 1 (cont)

Derive asymptotic upper bounds for $T(n)$.
Make your bounds as tight as possible.

(d) You may assume n is a power of 2.

$$\begin{aligned} &T(1) = 1 \\ &T(n) = 4T(n/2) + n \qquad \text{if } n > 1 \end{aligned}$$

(e) You may assume n is a power of 2.

$$\begin{aligned} &T(1) = 1 \\ &T(n) = 3T(n/2) + n^2 \qquad \text{if } n > 2 \end{aligned}$$

(f) You may assume n is a power of 2.

$$\begin{aligned} &T(1) = 1 \\ &T(n) = T(n/2) + \log_2 n \quad \text{if } n > 1 \end{aligned}$$

Question 1 (cont)

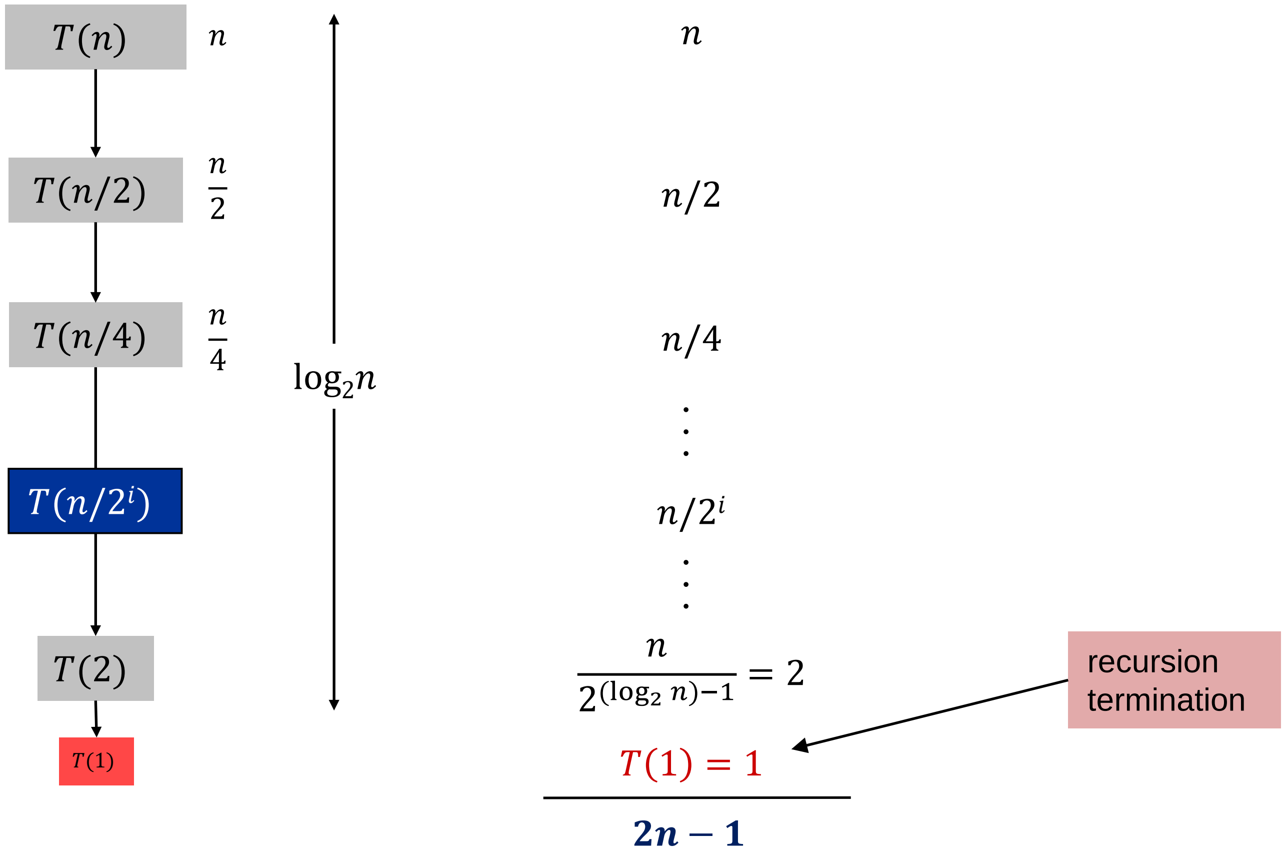
Derive asymptotic upper bounds for $T(n)$.
Make your bounds as tight as possible.

(g) You may assume n is a power of 2.

$$T(1) = 1$$

$$T(n) = 2T(n/2) + n\log_2 n \qquad \text{if } \; n > 1$$

Solution 1(a): Recursion tree approach



$T(n)$ n

$T(n/2)$ $\frac{n}{2}$

$T(n/4)$ $\frac{n}{4}$

$T(n/2^i)$

$T(2)$

$T(1)$

$\log_2 n$

n

$n/2$

$n/4$

$\vdots$

$n/2^i$

$\vdots$

$\frac{n}{2^{(\log_2 n)-1}} = 2$

$T(1) = 1$

$\mathbf{2n - 1}$

recursion termination

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

Assumption is that n is a power of 2.

Left side is the call to $T(\,)$ on that level;

Right side is amount of (non-recursive) work performed by problem on that level.

Note that there will only be one problem on each such level (so tree is a path)

Solution 1(a): Expansion method

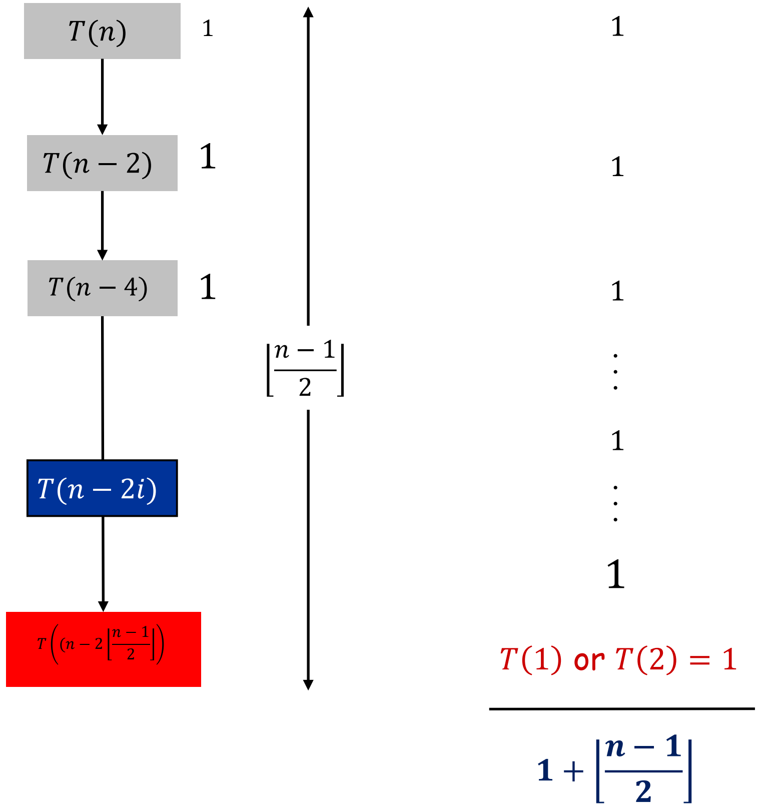
Set $h = \log_2 n$

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

$$\boxed{\Rightarrow\ T(n) = O(n)}$$

Solution 1(b): Recursion tree approach

$$T(n) = \begin{cases} 1 & n = 1,2 \\ T(n-2)+1 & if\ n > 2 \end{cases}$$



$T(n)$
1
$T(n-2)$
1
$T(n-4)$
1
$\left\lfloor \frac{n-1}{2} \right\rfloor$
$T(n-2i)$
$T\left(\left(n-2\left\lfloor \frac{n-1}{2} \right\rfloor\right)\right)$
1
1
1
⋮
1
⋮
1
$T(1)$ or $T(2) = 1$
$\mathbf{1 + \left\lfloor \frac{n-1}{2} \right\rfloor}$

Left side is the call to $T(\,)$ on that level.

Right side is amount of (non-recursive) work performed on that level.
Note that there will only be one problem on each such level (so tree is a path).

Solution 1(b): Expansion method

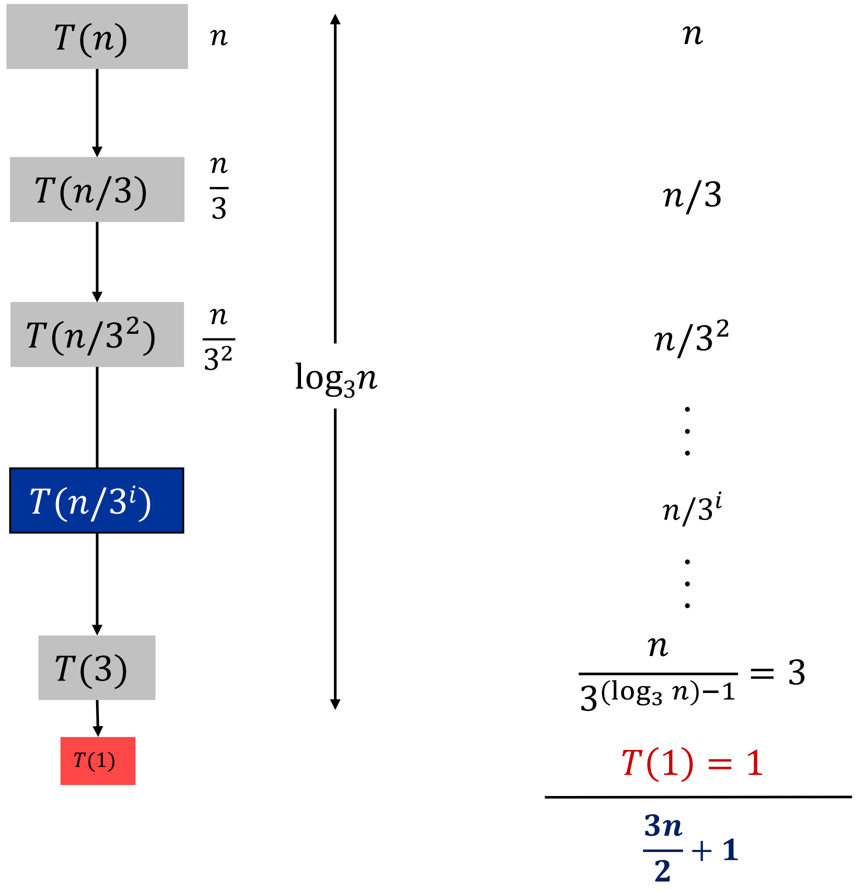
$$
\begin{aligned}
T(n) &= T(n-2)+1 \\
&= T(n-2\cdot 2)+2 \\
&= T(n-3\cdot 2)+3 \\
&\dots \\
&= T\left(n-\left\lfloor \frac{n-1}{2} \right\rfloor \cdot 2\right)+\left\lfloor \frac{n-1}{2} \right\rfloor
\end{aligned}
$$

Since $n-2\left\lfloor \frac{n-1}{2} \right\rfloor = 1, 2$ depending upon whether n is odd, even and $T(1)=T(2)=1$ we get

$$T(n) = 1 + \left\lfloor \frac{n-1}{2} \right\rfloor = \lceil (n/2) \rceil = O(n)$$

Solution 1(c): Recursion tree approach

$T(n) = T(n/3) + n$ if $n > 1$;
$T(1) = 1$.
Assumption is that n
is a power of 3.



$T(n)$ n

$T(n/3)$ $\frac{n}{3}$

$T(n/3^2)$ $\frac{n}{3^2}$

$T(n/3^i)$

$T(3)$

$T(1)$

$\log_3 n$

n

$n/3$

$n/3^2$

$\vdots$

$n/3^i$

$\vdots$

$\frac{n}{3^{(\log_3 n)-1}} = 3$

$T(1) = 1$

$\frac{3n}{2} + 1$

Left side is the call to $T(\,)$ on that level.

Right side is amount of (non-recursive) work performed on that level.
Note that there will only be one problem on each such level (so tree is a path).

Solution 1(c): Expansion method

Set $h = \log_3 n$

$$T(n) = n + T\left(\frac{n}{3}\right)$$

$$= n + \frac{n}{3} + T\left(\frac{n}{3^2}\right)$$

$$= n + \frac{n}{3} + \frac{n}{3^2} + T\left(\frac{n}{3^3}\right)$$

$$\ldots$$

$$= n + \frac{n}{3} + \frac{n}{3^2} + \cdots + \frac{n}{3^{t-2}} + \frac{n}{3^{t-1}} + T\left(\frac{n}{3^t}\right)$$

$$\ldots$$

$$= n + \frac{n}{3} + \frac{n}{3^2} + \cdots + \frac{n}{3^{h-2}} + \frac{n}{3^{h-1}} + T\left(\frac{n}{3^h}\right)$$

$$= n\left(1 + \frac{1}{3} + \frac{1}{3^2} + \cdots + \frac{1}{3^{h-2}} + \frac{1}{3^{h-1}}\right) + T\left(\frac{n}{3^h}\right)$$

$$\leq n \sum_{i=0}^{\infty} \left(\frac{1}{3}\right)^i + T\left(\frac{n}{3^h}\right)$$

$$= 3n/2 + T(1)$$

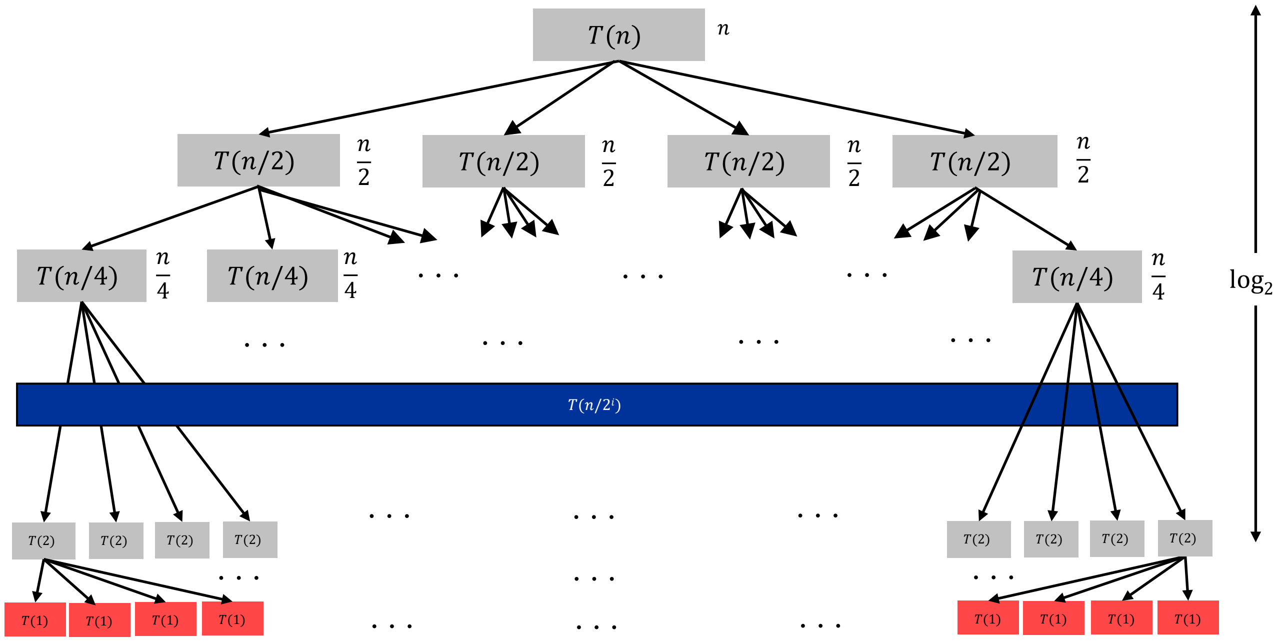
$$T(n) = O(n)$$

Solution 1(d): Recursion tree approach

$T(n) = n + 4T(n/2)$ if $n > 1$;

$T(1) = 1$.

Assume n is a power of 2.



n

$4 \cdot n/2$

$4^2 \cdot n/2^2$

⋮

$4^i \cdot n/2^i$

⋮

$4^{\log_2 n-1}\frac{n}{2^{\log_2 n-1}} = \frac{n^2}{2}$

$n^2 T(1) = n^2$

$\boldsymbol{n^2 + n(n-1)}$

Left side is the call to $T(\,)$ on that level.

Right side is amount of (non-recursive) work performed on that level.
Note that tree is NOT a path.

Solution 1(d): Expansion method

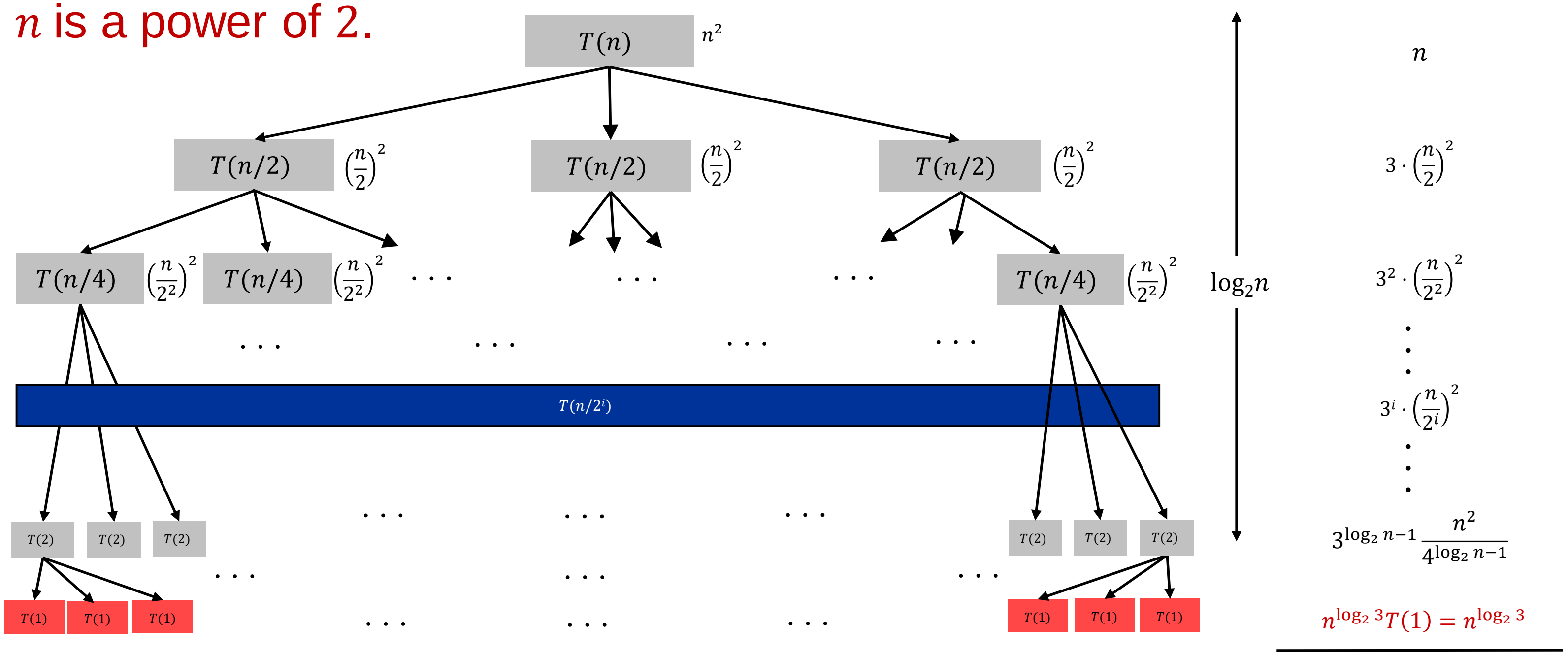
Set $h = \log_2 n$. Then $4^h = (2^2)^h = 2^{2h} = \left(2^h\right)^2 = n^2$.

$$
\begin{aligned}
T(n) &= n + 4T\left(\frac{n}{2}\right) \\
&= n + 4\left(\frac{n}{2} + 4T\left(\frac{n}{2^2}\right)\right) \\
&= n + 2n + 4^2T\left(\frac{n}{2^2}\right) \\
&\dots \\
&= n + 2n + 2^2n + \dots + 2^{t-1}n + 4^tT\left(\frac{n}{2^t}\right) \\
&\dots \\
&= n + 2n + 2^2n + \dots + 2^{h-1}n + 4^hT\left(\frac{n}{2^h}\right) \\
&= n(2^h - 1) + 4^h
\end{aligned}
$$

$$T(n) = n(n-1) + 4^h = O(n^2)$$

Solution 1(e): Recursion tree approach

$T(n) = n^2 + 3T(n/2)$ if $n > 1$;
$T(1) = 1$.
Assume n is a power of 2.



$T(n)$
n^2
$T(n/2)$
$\left(\frac{n}{2}\right)^2$
$T(n/2)$
$\left(\frac{n}{2}\right)^2$
$T(n/2)$
$\left(\frac{n}{2}\right)^2$
$T(n/4)$
$\left(\frac{n}{2^2}\right)^2$
$T(n/4)$
$\left(\frac{n}{2^2}\right)^2$
$T(n/4)$
$\left(\frac{n}{2^2}\right)^2$
$T(n/2^i)$
$T(2)$
$T(1)$
$\log_2 n$
n
$3 \cdot \left(\frac{n}{2}\right)^2$
$3^2 \cdot \left(\frac{n}{2^2}\right)^2$
$3^i \cdot \left(\frac{n}{2^i}\right)^2$
$3^{\log_2 n - 1} \frac{n^2}{4^{\log_2 n - 1}}$
$n^{\log_2 3} T(1) = n^{\log_2 3}$

$$\boldsymbol{n^{\log_2 3} + n^2 \sum_{i=0}^{\log_2 n - 1} \left(\frac{3}{4}\right)^i}$$

Left side is the call to $T(\,)$ on that level.

Right side is amount of (non-recursive) work performed on that level.

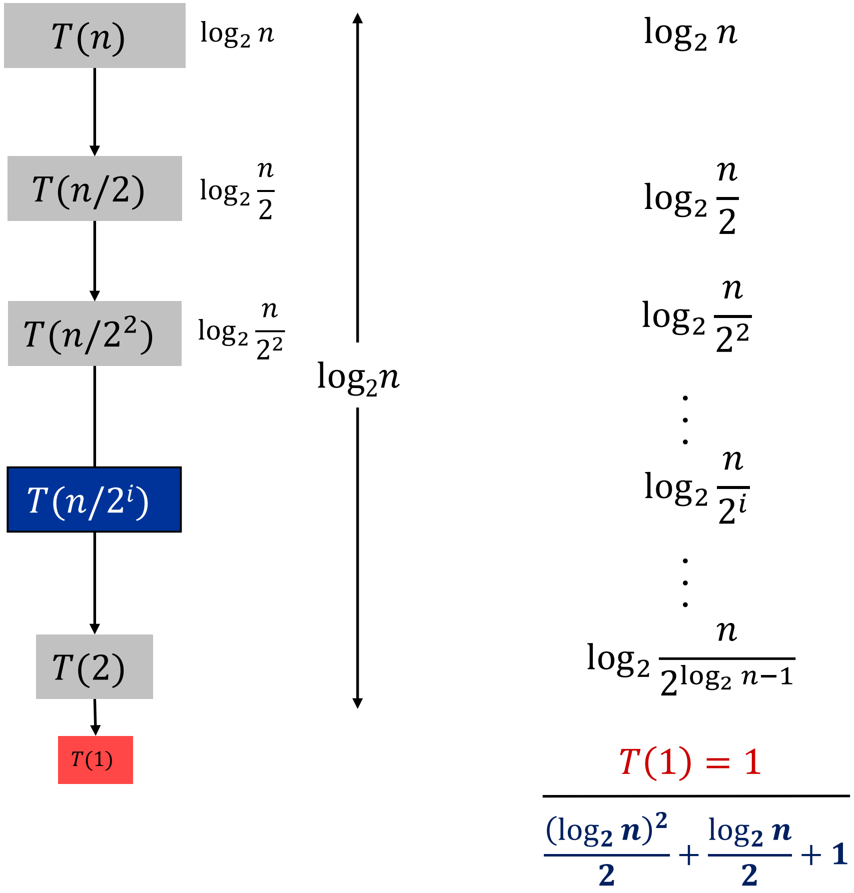
Solution 1(e): Expansion method

Set $h = \log_2 n$. Note that $3^h = 3^{\log_2 3\ \log_3 n} = n^{\log_2 3} = O(n^2)$

$$
\begin{aligned}
T(n) &= n^2 + 3T\left(\frac{n}{2}\right) \\
&= n^2 + 3\left(\left(\frac{n}{2}\right)^2 + 3T\left(\frac{n}{2^2}\right)\right) \\
&= n^2 + \frac{3}{4}n^2 + 3^2T\left(\frac{n}{2^2}\right) \\
&= n^2 + \frac{3}{4}n^2 + 3^2\left(\left(\frac{n}{2^2}\right)^2 + 3T\left(\frac{n}{2^3}\right)\right) \\
&\dots \\
&= n^2 + \frac{3}{4}n^2 + \left(\frac{3}{4}\right)^2 n^2 + \dots + \left(\frac{3}{4}\right)^{t-1} n^2 + 3^tT\left(\frac{n}{2^t}\right) \\
&\dots \\
&= n^2 + \frac{3}{4}n^2 + \left(\frac{3}{4}\right)^2 n^2 + \dots + \left(\frac{3}{4}\right)^{h-1} n^2 + 3^hT\left(\frac{n}{2^h}\right) \\
&= n^2 \textstyle\sum_{i=0}^{h-1} (3/4)^i + 3^h = O(n^2)
\end{aligned}
$$

Solution 1(f): Recursion tree approach

$T(n) = T(n/2) + \log_2 n$ if $n > 1$;
$T(1) = 1$.
Assume that n is a power of 2.



$T(n)$ $\log_2 n$

$T(n/2)$ $\log_2 \frac{n}{2}$

$T(n/2^2)$ $\log_2 \frac{n}{2^2}$

$T(n/2^i)$

$T(2)$

$T(1)$

$\log_2 n$

$\log_2 n$

$\log_2 \frac{n}{2}$

$\log_2 \frac{n}{2^2}$

$\vdots$

$\log_2 \frac{n}{2^i}$

$\vdots$

$\log_2 \frac{n}{2^{\log_2 n - 1}}$

$T(1) = 1$

$$\frac{(\log_2 n)^2}{2} + \frac{\log_2 n}{2} + 1$$

Left side is the call to $T(\,)$ on that level.

Right side is amount of (non-recursive) work performed on that level.
Note that there will only be one problem on each such level (so tree is a path)

Solution 1(f): Expansion method

Set $h = \log_2 n$

$$T(n) = \log_2 n + T\left(\frac{n}{2}\right)$$

$$= \log_2 n + \log_2 \frac{n}{2} + T\left(\frac{n}{2^2}\right)$$

$$\dots$$

$$= \log_2 n + \log_2 \frac{n}{2} + \dots + \log_2 \frac{n}{2^{t-1}} + T\left(\frac{n}{2^t}\right)$$

$$\dots$$

$$= \log_2 n + \log_2 \frac{n}{2} + \dots + \log_2 \frac{n}{2^{h-1}} + T\left(\frac{n}{2^h}\right)$$

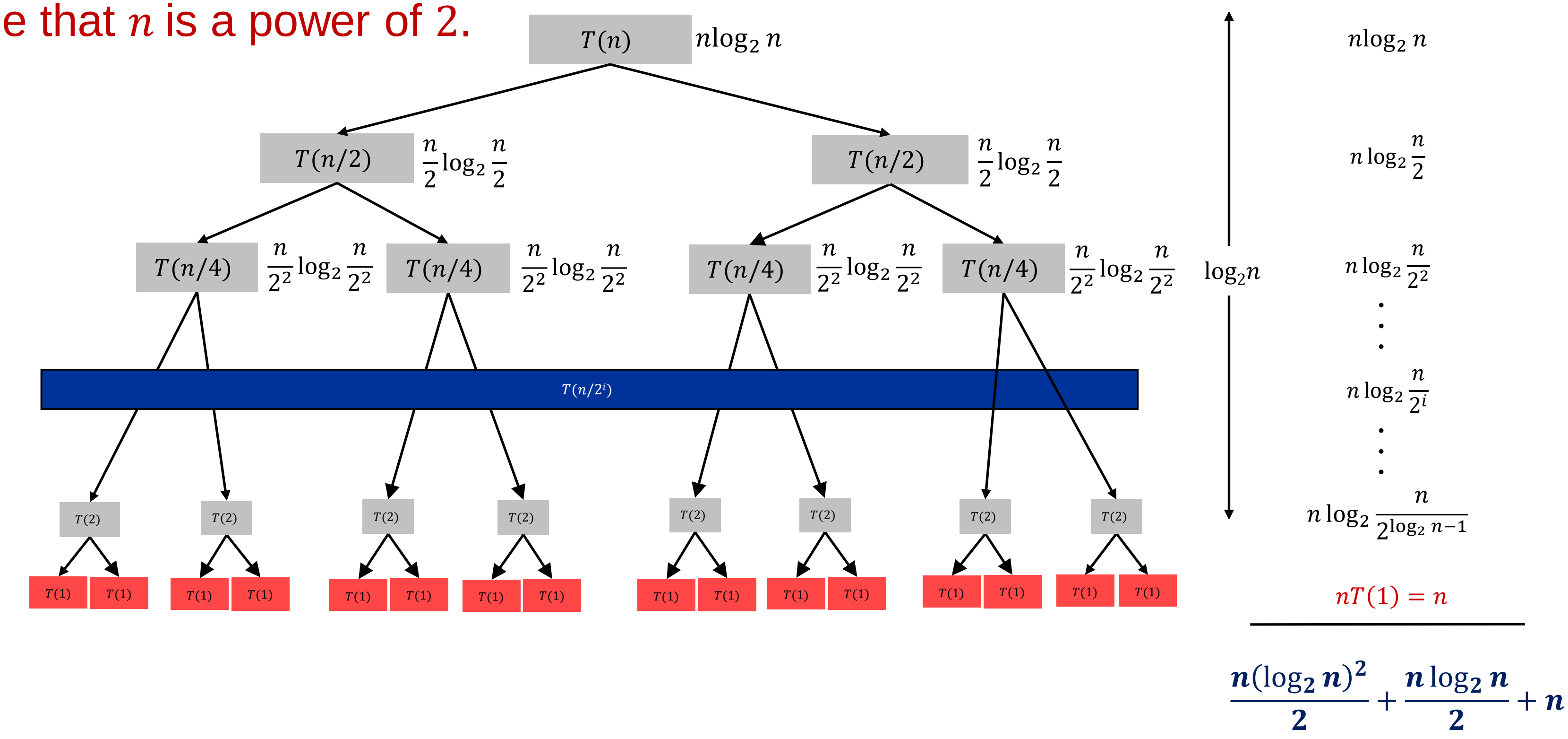
$$= T(1) + \sum_{i=0}^{h-1} \log_2 \frac{n}{2^i} = 1 + \sum_{i=0}^{h-1} (\log_2 n - i)$$

$$= T(1) + h\log_2 n - h\,(h-1)/2 = 1 + h^2 - h\,(h-1)/2$$

$$= O(h^2) = O(\log^2 n)$$

Solution 1(g): Recursion tree approach

$T(n) = n\log_2 n + 2T(n/2)$ if $n > 1$;
$T(1) = 1$.
Assume that n is a power of 2.



$T(n)$ $n\log_2 n$

$T(n/2)$ $\frac{n}{2}\log_2\frac{n}{2}$ $T(n/2)$ $\frac{n}{2}\log_2\frac{n}{2}$

$T(n/4)$ $\frac{n}{2^2}\log_2\frac{n}{2^2}$ $T(n/4)$ $\frac{n}{2^2}\log_2\frac{n}{2^2}$ $T(n/4)$ $\frac{n}{2^2}\log_2\frac{n}{2^2}$ $T(n/4)$ $\frac{n}{2^2}\log_2\frac{n}{2^2}$

$T(n/2^i)$

$T(2)$ $T(2)$ $T(2)$ $T(2)$ $T(2)$ $T(2)$ $T(2)$ $T(2)$

$T(1)$ $T(1)$ $T(1)$ $T(1)$ $T(1)$ $T(1)$ $T(1)$ $T(1)$ $T(1)$ $T(1)$ $T(1)$ $T(1)$ $T(1)$ $T(1)$ $T(1)$ $T(1)$

$\log_2 n$

$n\log_2 n$

$n\log_2\frac{n}{2}$

$n\log_2\frac{n}{2^2}$

$\vdots$

$n\log_2\frac{n}{2^i}$

$\vdots$

$n\log_2\frac{n}{2^{\log_2 n-1}}$

$nT(1) = n$

$$\frac{\boldsymbol{n}(\log_2 \boldsymbol{n})^2}{\mathbf{2}} + \frac{\boldsymbol{n}\log_2 \boldsymbol{n}}{\mathbf{2}} + \boldsymbol{n}$$

Left side is the call to $T(\,)$ on that level.

Right side is amount of (non-recursive) work performed on that level.

Solution 1(g): Expansion method

Set $h = \log_2 n$

$$T(n) = n\log_2 n + 2T\left(\frac{n}{2}\right)$$

$$= n\log_2 n + 2\left(\frac{n}{2}\log_2\frac{n}{2} + 2T\left(\frac{n}{2^2}\right)\right)$$

$$\dots$$

$$= n\log_2 n + n\log_2\frac{n}{2} + \dots + n\log_2\frac{n}{2^{t-1}} + 2^t T\left(\frac{n}{2^t}\right)$$

$$\dots$$

$$= n\log_2 n + n\log_2\frac{n}{2} + \dots + n\log_2\frac{n}{2^{h-1}} + 2^h T\left(\frac{n}{2^h}\right)$$

$$= nT(1) + n\sum_{i=0}^{h-1}\log_2\frac{n}{2^i} = n + n\sum_{i=0}^{h-1}(\log_2 n - i)$$

$$= n + n(h\log_2 n - h\,(h-1)/2) = n + n(h^2 - h(h-1)/2)$$

$$= O(nh^2) = O(n\log^2 n)$$

Question 2

Input: a sorted array $A[1..n]$ of n distinct integers

(Distinct means that there are no repetitions among the integers. The integers can be positive, negative or both).

Problem:
Design an $O(\log n)$ algorithm to return an index i such that $A[i] = i$, if such an i exists.

Otherwise, report that no such index exists.

As an example, in the array immediately below, the algorithm returns 4

i	1	2	3	4	5	6	7	8
$A[i]$	-3	0	1	4	12	17	20	22

while in the array below, the algorithm returns that no such index exists.

i	1	2	3	4	5	6	7	8
$A[i]$	-3	0	1	7	12	17	20	22

Question 2: Hint

Input: a sorted array $A[1..n]$ of n distinct integers

(Distinct means that there are no repetitions among the integers.
The integers can be positive, negative or both).

Problem:
Design an $O(\log n)$ algorithm to return an

index i such that $A[i] = i$, if such an i exists.

Otherwise, report that no such index exists.

Hint: $O(log\, n)$ often denotes some type of binary search.
Can you think of any question you might ask that permits you to throw away some constant fraction of the items.

Solution 2

The main observation is that

If $A[m] > m$, then $A[i] > i$ for all $i > m$.

Follows directly from the facts that the array is **sorted** and all integers are **distinct**.

- If $A[m] = m$, we have found solution and can stop
- If $A[m] > m$, from observation above, the top half of the array cannot contain desired index.
- If $A[m] < m$, similarly $A[i] < i$ for all $i < m$.
  If solution exists, must be in $A[m + 1..n]$

So, after checking $A[m]$ against m we can either
(i) stop or
(ii) throw away half of the array and solve the problem on the other half.

The running time of the algorithm has the recurrence
$T(n) = T(n/2) + O(1)$, which solves to $T(n) = O(\log n)$.

Solution 2

```
INDEX-SEARCH(A,s,t)
        if $(s = t)$ $\\ O(1)$
              if $(A[s] = s)$
                    return $s$;
                 else
                        return $-1$;
        $m \leftarrow \left\lfloor \frac{s+t}{2} \right\rfloor$;

        if $(A[m] = m)$ return $m$; $\\ O(1)$

         if $(A[m] > m)$
                 return INDEX-SEARCH(A,s,m) ;   $\\ O\left(T\left\lfloor\frac{n}{2}\right\rfloor\right)$
        else
                 return INDEX-SEARCH(A,m+1,t) ; $\\ O\left(T\left\lceil\frac{n}{2}\right\rceil\right)$
```

Question 3

Let $A[1..n]$ be an array of n elements. A majority element of A is any element occurring 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 exist

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

Note: Obviously a solution to (b) is also a solution to (a). The intent here is to use standard divide-and-conquer techniques to solve (a) and then see if you can get a faster solution in (b) by using more problem specific properties.

Question 3: Examples

In the array below, 2 is the majority item:

i	1	2	3	4	5	6	7	8
$A[i]$	2	3	2	1	2	5	2	2

In the array below there is no majority item:

i	1	2	3	4	5	6	7	8
$A[i]$	2	3	5	1	2	5	2	2

Solution 3(a)

The important observation is that

If A contains a majority element e,
=> e must be a maj. element in at least one of $A[1..n/2]$ and $A[n/2+1..n]$.

Algorithm is then

1. Find majority e_1, if it exists, in $A[1..n/2]$

2. Find majority e_2, if it exists, in $A[n/2+1..n]$

3. If e_1 exists, count how many times it occurs in A
   If more than $n/2$, report e_1.

   If e_2 exists, count how many times it occurs in A.
   If more than $n/2$, report e_2.

   If neither exist, report, "no majority".

The base case is when $n = 1$,
and we can simply return the only element as the majority.

Solution 3(a)

1. Find majority e_1, if it exists, in $A[1..n/2]$ $\mathrm{T}(n/2)$

2. Find majority e_2, if it exists, in $A[n/2+1..n]$ $T(n/2)$

3. If e_1 exists, count how many times it occurs in A $O(n)$

   If more than $n/2$, report e_1.

   If e_2 exists, count how many times it occurs in A $O(n)$

   If more than $n/2$, report e_2.

   If neither exist, report, “no majority”.

Algorithm’s correctness follows from the first observation on previous page.
Its running time satisfies

$$T(n) = 2T(n/2) + O(n), \quad \text{with } T(1) = 1$$

which solves to $T(n) = O(n\log n)$

Solution 3(a)

Extended Question: Suppose we want to find all elements that
appear more than 1/3 of the time?
appear more 1/4 of the time?
appear more than x% of the time?

How could the method on the previous slides be modified to solve the extension?

More about this problem and its practical uses can be found by googling Heavy Hitters in Data Streams

Solution 3(b) (O(N) algorithm. Extra. Not required for class)

Initially set $e = NULL$ and a counter $c = 0$.
Walk through the items one at a time as follows:

```
For $i = 1$ to $n$
    If $c = 0$,
        $e = A[i]$ and set $c = 1$.   % $e$ is current potential majority
    else
        If $e = A[i]$ set $c = c + 1$
        If $e \neq A[i]$ set $c = c - 1$
```

Claim: If a majority exists, it must be the value of e when procedure ends.

Assuming this claim is correct, we just scan the items one more time counting how often that e appears.
If it appears more than $n/2$ times, e is a majority; otherwise, none exists.

This is an $O(n)$ algorithm.

Solution 3(b): Example

```
For $i = 1$ to $n$
        If $c = 0$,
                $e = A[i]$ and set $c = 1$.        % $e$ is current potential majority
        else
                If $e = A[i]$ set $c = c + 1$
                If $e \neq A[i]$ set $c = c - 1$
```

Claim: If a majority exists, it must be the value of e when procedure ends.

i	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
$A[i]$	a	a	d	b	d	b	a	a	b	b	b	b	a	b	b
e	a	a	a	a	d	d	a	a	a	a	b	b	b	b	b
c	1	2	1	0	1	0	1	2	1	0	1	2	1	2	3

In this example, claim implies that if a majority element exists, it must be “b”.
After running the for loop, run through all items 1 more time, counting # of “b”’s

We find 8 “b”s out of 15 items so “b” is a majority element!

Solution 3(b): Proof of Claim

Claim: If a majority exists, it must be the value of e when procedure ends.

To prove the claim we *partition* the array into consecutive subarrays $S_1, S_2, \ldots . S_m$ by *labelling* each item $A[i]$ with the value of e after step i

Example

i	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
$A[i]$	a	a	d	b	d	b	a	a	b	b	b	b	a	b	b
e	a	a	a	a	d	d	a	a	a	a	b	b	b	b	b
c	1	2	1	0	1	0	1	2	1	0	1	2	1	2	3

Note that all items in the same subarray S_j have the same label, e_j

Consider subarray S_j which is not last subarray, i.e., $j \neq m$.

1st observation: Exactly ½ the items in S_j have the value e_j

This is because, in S_j,
c increments by 1 when it sees $\mathrm{A[i]} = e_j$
and decrements by 1 when it sees a different character $\mathrm{A[i]} \neq e_j$
and c starts and ends the subarray with value 0.

Solution 3(b): Proof of Claim

Claim: If a majority exists, it must be the value of e when procedure ends.

To prove the claim we *partition* the array into consecutive subarrays $S_1, S_2, \ldots . S_m$ by *labelling* each item $A[i]$ with the value of e after step i

Example

i	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
$A[i]$	a	a	d	b	d	b	a	a	b	b	b	b	a	b	b
e	a	a	a	a	d	d	a	a	a	a	b	b	b	b	b
c	1	2	1	0	1	0	1	2	1	0	1	2	1	2	3

Note that all items in the same subarray S_j have the same label, e_j

Consider subarray S_j which is not last subarray m..

1st observation: Exactly ½ the items in S_j have the value e_j

2nd observation: At least ½ the items in S_m have the value e_m

This follows from same reasoning
(except that at end of last subarray, c might be larger than 0).

Solution 3(b): Proof of Claim

Claim: If a majority exists, it must be the value of e when procedure ends.

To prove the claim we *partition* the array into consecutive subarrays $S_1, S_2, \ldots .S_m$ by *labelling* each item $A[i]$ with the value of e after step i

Consider subarray S_j which is not last subarray m..

1st observation: Exactly ½ the items in S_j have the value e_j

2nd observation: At least ½ the items in S_m have the value e_m

Now let x be any character, $x \neq e_m$

From observations, **at most** ½ the items in each S_j, $j \leq m$, have value x.

Adding this up over all subarrays shows

$\Rightarrow$ **at most** ½ the items in the entire array = x

$\Rightarrow$ x can not be a majority item

$\Rightarrow$ The only possible majority item is e_m, which is what the claim is stating!

Question 4

You have found a newspaper from the future that tells you the price of a stock over a period of n days next year. This is presented to you as an array $\mathrm{p}[1 \ldots n]$ where $p[i]$ is the price of the stock on day i.

Design an $O(n \log n)$time divide-and-conquer algorithm that finds a strategy to make as much money as possible, i.e., it finds a pair i, j with $1 \leq i < j \leq n$ such that $p[j] - p[i]$ is maximized over all possible such pairs

Note: you are only allowed to buy the stock once and then sell it later. You can not make multiple purchases and sales over time.

If there is no way to make money,
i.e., $p[j] - p[i] \leq 0$ for all pairs i, j with $1 \leq i < j \leq n$,
your algorithm should return ``no way".

Solution 4

Note: This problem can be solved using similar ideas as were used for the maximum contiguous subarray problem. Similar to that problem, there is also an $O(n)$ time linear scan solution. That is not what we are asking for here but it is a good exercise to see if you can find it.

To find the pair i, j in $p[a \dots b]$ with largest value $p[j] - p[i]$ note that,
if q is the midpoint of $a \dots b$, then exactly one of the following three cases occurs:

- Both of i, j are in $a \dots q$
- Both of i, j are in $q + 1 \dots b$
- i is in $a \dots q$ and j is in $q + 1 \dots b$

=> if $b - a > 1$ => optimal solution is the maximum of

- The best solution in $a \dots q$
- The best solution in $q + 1 \dots b$
- $j_{max} - i_{min}$ where $j_{max} = \max_{q+1 \le j \le b} p[j]$ and $i_{min} = \min_{a \le i \le q} p[i]$

if $b - a = 1$ => optimal solution is $\max\{p[b] - p[a], 0\}$

if $b - a = 0$ => optimal solution is $p[b] - p[a] = 0$

Solution 4

if $b - a > 1$, set $q = \left\lfloor \frac{a+b}{2} \right\rfloor$

=> optimal solution $opt(a,b)$ is the maximum of

- The best solution in $a \ldots q$ $(opt(a,q))$
- The best solution in $q + 1 \ldots b$ $(opt(q+1,b))$
- $j_{max} - i_{min}$ where $j_{max} = \max_{q+1 \le j \le b} p[j]$ and $i_{min} = \min_{a \le i \le q} p[i]$

if $b - a = 1$ => $opt(a,b) = \max\{p[b] - p[a], 0\}$

if $b - a = 0$ => $opt(a,b) = p[b] - p[a] = 0$

$$opt(a,b) = \begin{cases} 0 & if\ b - a = 0 \\ \max\{p[b] - p[a], 0\} & if\ b - a = 1 \\ \max\{opt(a,q), opt(q+1,b), j_{max} - i_{min}\} & if\ b - a > 1 \end{cases}$$

Code for finding $opt(a,b)$ in the non-terminal case would make two recursive calls of half the size and does $O(b-a)$ work in finding i_{min} and j_{max}

Running time satisfies $T(n) = 2T\left(\frac{n}{2}\right) + O(n)$ which we know implies $T(n) = O(n \log n)$.

Question 5: BubbleSort

See the Bubble Sort PowerPoint.