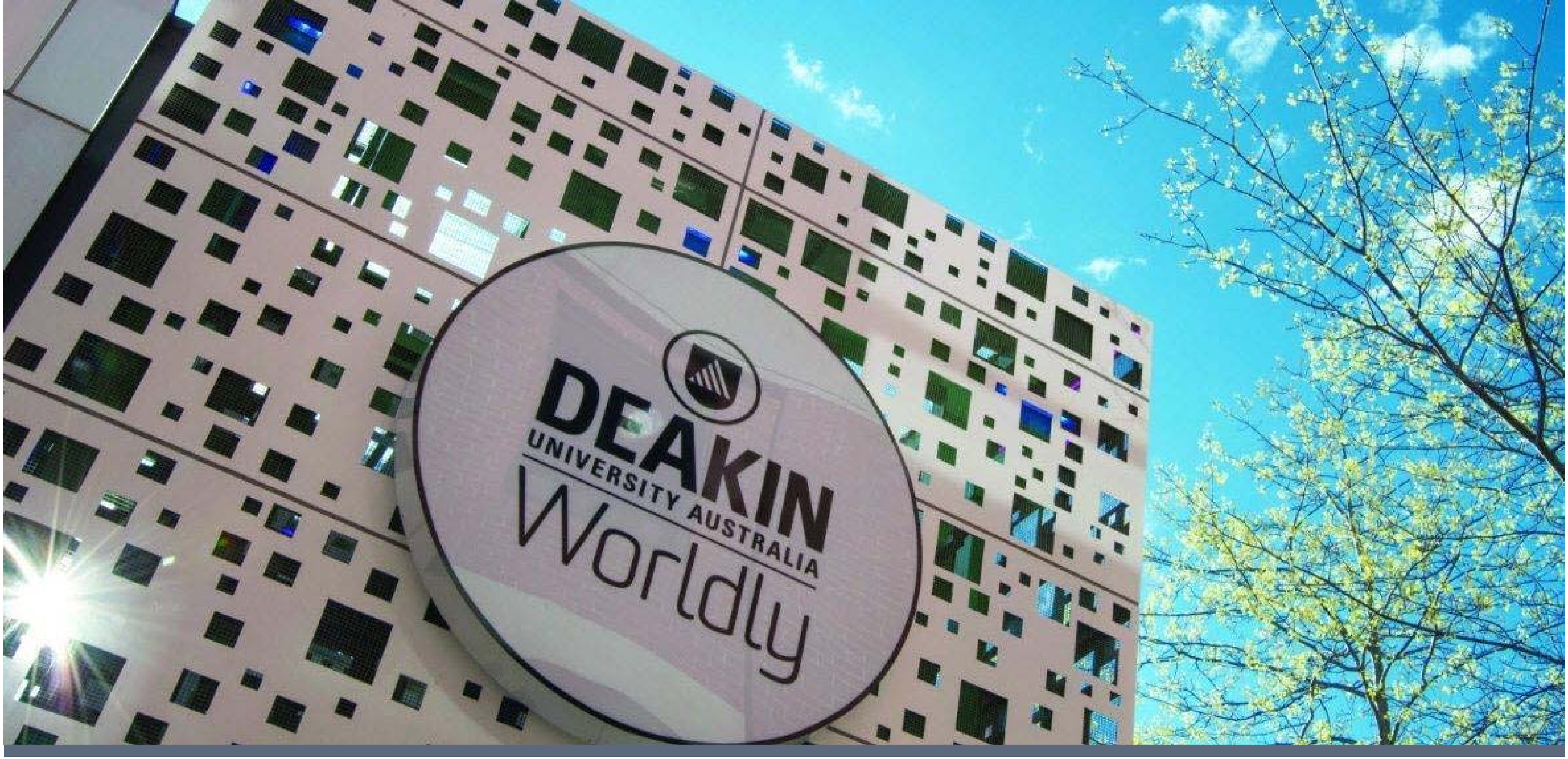
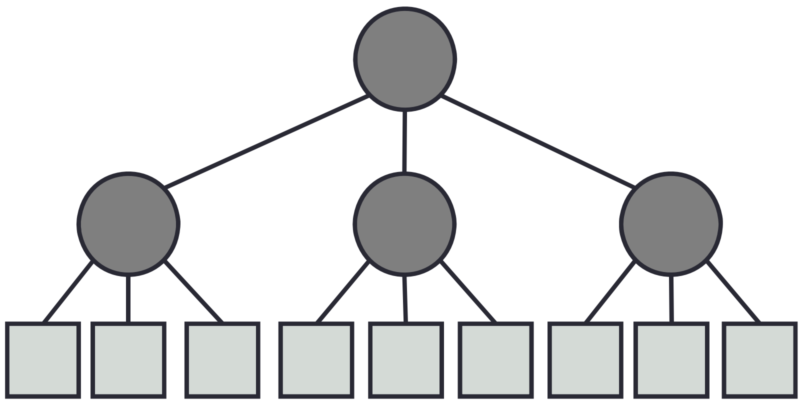


Lecture 3. Analysis of Recursive Algorithms. Binary Search. Recursive approach

SIT221 Data Structures and Algorithms

Divide and Conquer Paradigm

- Divide-and conquer is a general algorithm design paradigm:
  - Divide: divide the input data S in two or more disjoint subsets $S_1, S_2, \ldots$
  - Recur: solve the sub-problems recursively
  - Conquer: combine the solutions for $S_1, S_2, \ldots$ into a solution for S
- The base case for the recursion are sub-problems of constant size
- Analysis can be done using **recurrence equation**



Merge Sort: Review

Input: sequence S with n elements, comparator C

Output: sequence S sorted according to C

```
void MergeSort( S, C )
    if ( Size(S) > 1 ) {
        (S1, S2) ← Partition( S, n/2 )
        MergeSort( S1, C )
        MergeSort( S2, C )
        S ← Merge( S1, S2 )
    }
```

Merge Sort on an input sequence S with n elements consists of three steps:

- Divide: partition S into two sequences S_1 and S_2 of about $\frac{n}{2}$ elements each
- Recur: recursively sort S_1 and S_2
- Conquer: merge S_1 and S_2 into a unique sorted sequence

Recurrence Equation Analysis

- The conquer step of merge-sort consists of merging two sorted sequences, each with $n/2$ elements, and takes at most $b \cdot n$ steps for some constant b.
- Likewise, the basis case (n<2) will take at b most steps.
- Let $T(n)$ denote the running time of Merge Sort:

$$T(n) = \begin{cases} b & \text{if } n < 2 \\ 2T(n/2) + bn & \text{if } n \geq 2 \end{cases}$$

- We can therefore analyze the running time of Merge Sort by finding a **closed form solution** to the above equation.
- That is, a solution that has $T(n)$ only on the left-hand side.

Iterative Substitution

- In the iterative substitution technique, we iteratively apply the recurrence equation to itself and see if we can find a pattern:

$$
\begin{aligned}
T(n) &= 2T(n/2) + bn \\
&= 2(2T(n/2^2)) + b(n/2)) + bn \\
&= 2^2 T(n/2^2) + 2bn \\
&= 2^3 T(n/2^3) + 3bn \\
&= 2^4 T(n/2^4) + 4bn \\
&= \ldots \\
&= 2^i T(n/2^i) + ibn
\end{aligned}
$$

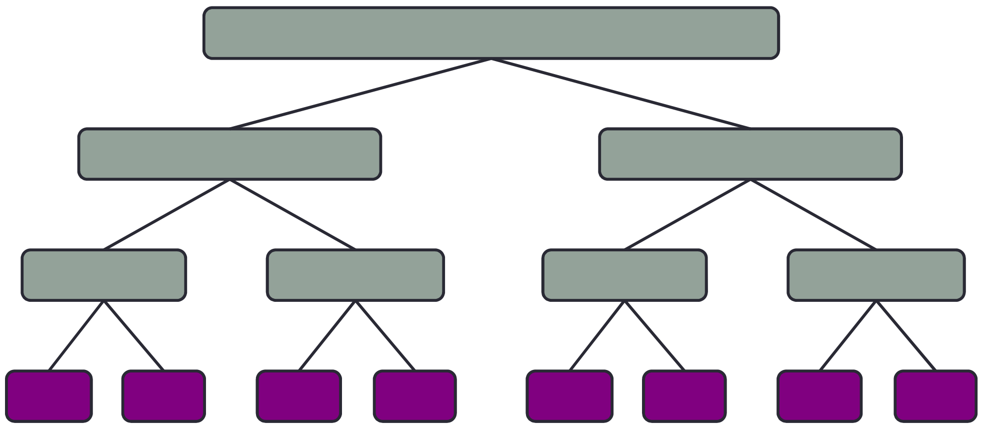
- Note that base, $T(n) = b$, case occurs when $2^i = n$. That is, $i = \log n$.
- So, $T(n) = bn + bn \log n$. Thus, $T(n)$ is $O(n \log n)$

The Recursion Tree

Draw the recursion tree for the recurrence relation and look for a pattern:

$$T(n) = \begin{cases} b & \text{if } n < 2 \\ 2T(n/2) + bn & \text{if } n \geq 2 \end{cases}$$

depth	T's	size	time
0	1	n	bn
1	2	$n/2$	bn
i	2^i	$n/2^i$	bn
…	…	…	…



Total time $T(n) = bn + bn \log n$

(last level plus all previous levels)

Guess-and-Test Method

- Recall the recurrence equation:

$$T(n) = \begin{cases} b & \text{if } n < 2 \\ 2T(n/2) + bn\log n & \text{if } n \geq 2 \end{cases}$$

- Guess: $T(n) < cn\log n$.

$$\begin{aligned} T(n) &= 2T(n/2) + bn\log n \\ &= 2(c(n/2)\log(n/2)) + bn\log n \\ &= cn(\log n - \log 2) + bn\log n \\ &= cn\log n - cn + bn\log n \end{aligned}$$

- Wrong: we cannot make this last line be less than $cn\log n$.
- In general, to use this method, you need to have a good guess and you need to be good at induction proofs.

General Tool for Solving Recursion: Master Theorem

- Solving recursive formulas can be complicated.
- Often requires induction proofs and a good guess about what to proof.
- Would be great to have a general tool for solving standard recursive formulas.
- The master Theorem provides a general way of solving recursion.

Master Method

Many divide-and-conquer recurrence equations have the form:

$$T(n) = \begin{cases} c & \text{if } n < d \\ aT(n/b) + f(n) & \text{if } n \geq d \end{cases}$$

The Master Theorem:

1. If $f(n) = O(n^k)$, where $k < \log_b a$, then $T(n) = \Theta\left(n^{\log_b a}\right)$
2. If $f(n) = \Theta(n^k \log^j n)$, where $k = \log_b a$, then $T(n) = \Theta\left(n^k \log^{j+1} n\right)$
3. If $f(n) = \Omega(n^k)$, where $k > \log_b a$, then $T(n) = \Theta\left(f(n)\right)$
   provided $af(n/b) \leq \delta f(n)$ for some $\delta < 1$.

- a is the number of sub-problems we have each time
- b defines the size of each sub-problem

Master Method

$$T(n) = \begin{cases} c & \text{if } n < d \\ aT(n/b) + f(n) & \text{if } n \geq d \end{cases}$$

The Master Theorem:

1. If $f(n) = O(n^k)$, where $k < \log_b a$, then $T(n) = \Theta\left(n^{\log_b a}\right)$
2. If $f(n) = \Theta(n^k \log^j n)$, where $k = \log_b a$, then $T(n) = \Theta\left(n^k \log^{j+1} n\right)$
3. If $f(n) = \Omega(n^k)$, where $k > \log_b a$, then $T(n) = \Theta\left(f(n)\right)$

Case1: If the work done at leaves is more, then leaves are the dominant part, and our result becomes the work done at leaves.

Case2: If work done at leaves and root is asymptotically same, then result becomes height multiplied by work done at any level.

Case3: If work done at root is asymptotically more, then our result becomes work done at root.

Master Method: Example 1

$$T(n) = \begin{cases} c & \text{if } n < d \\ aT(n/b) + f(n) & \text{if } n \geq d \end{cases}$$

The Master Theorem:

1. If $f(n) = O(n^k)$, where $k < \log_b a$, then $T(n) = \Theta\left(n^{\log_b a}\right)$
2. If $f(n) = \Theta(n^k \log^j n)$, where $k = \log_b a$, then $T(n) = \Theta\left(n^k \log^{j+1} n\right)$
3. If $f(n) = \Omega(n^k)$, where $k > \log_b a$, then $T(n) = \Theta\left(f(n)\right)$

$$T(n) = 4T(n/2) + n$$

Solution: $\log_b a = \log_2 4 = 2$, so case 1 says $T(n) = O(n^2)$.

Master Method: Example 2

$$T(n) = \begin{cases} c & \text{if } n < d \\ aT(n/b) + f(n) & \text{if } n \geq d \end{cases}$$

The Master Theorem:

1. If $f(n) = O(n^k)$, where $k < \log_b a$, then $T(n) = \Theta\left(n^{\log_b a}\right)$
2. If $f(n) = \Theta(n^k \log^j n)$, where $k = \log_b a$, then $T(n) = \Theta\left(n^k \log^{j+1} n\right)$
3. If $f(n) = \Omega(n^k)$, where $k > \log_b a$, then $T(n) = \Theta\left(f(n)\right)$

$$T(n) = 2T(n/2) + n \log n$$

Solution: $\log_b a = \log_2 2 = 1$, so case 2 says $T(n) = O(n\log^2 n)$.

Master Method: Example 3

$$T(n) = \begin{cases} c & \text{if } n < d \\ aT(n/b) + f(n) & \text{if } n \ge d \end{cases}$$

The Master Theorem:

1. If $f(n) = O(n^k)$, where $k < \log_b a$, then $T(n) = \Theta\left(n^{\log_b a}\right)$
2. If $f(n) = \Theta(n^k \log^j n)$, where $k = \log_b a$, then $T(n) = \Theta\left(n^k \log^{j+1} n\right)$
3. If $f(n) = \Omega(n^k)$, where $k > \log_b a$, then $T(n) = \Theta\left(f(n)\right)$

$$T(n) = T(n/3) + n \log n$$

Solution: $\log_b a = \log_3 1 = 0$, so case 3 says $T(n) = O(n \log n)$.

Master Method: Example 4

$$T(n) = \begin{cases} c & \text{if } n < d \\ aT(n/b) + f(n) & \text{if } n \geq d \end{cases}$$

The Master Theorem:

1. If $f(n) = O(n^k)$, where $k < \log_b a$, then $T(n) = \Theta\left(n^{\log_b a}\right)$
2. If $f(n) = \Theta(n^k \log^j n)$, where $k = \log_b a$, then $T(n) = \Theta\left(n^k \log^{j+1} n\right)$
3. If $f(n) = \Omega(n^k)$, where $k > \log_b a$, then $T(n) = \Theta\left(f(n)\right)$

$$T(n) = 8T(n/2) + n^2$$

Solution: $\log_b a = \log_2 8 = 3$, so case 1 says $T(n) = O(n^3)$.

Master Method: Example 5

$$T(n) = \begin{cases} c & \text{if } n < d \\ aT(n/b) + f(n) & \text{if } n \geq d \end{cases}$$

The Master Theorem:

1. If $f(n) = O(n^k)$, where $k < \log_b a$, then $T(n) = \Theta\left(n^{\log_b a}\right)$
2. If $f(n) = \Theta(n^k \log^j n)$, where $k = \log_b a$, then $T(n) = \Theta\left(n^k \log^{j+1} n\right)$
3. If $f(n) = \Omega(n^k)$, where $k > \log_b a$, then $T(n) = \Theta\left(f(n)\right)$

$$T(n) = 9T(n/3) + n^3$$

Solution: $\log_b a = \log_3 9 = 2$, so case 3 says $T(n) = O(n^3)$.

Master Method: Example 6

$$T(n) = \begin{cases} c & \text{if } n < d \\ aT(n/b) + f(n) & \text{if } n \geq d \end{cases}$$

The Master Theorem:

1. If $f(n) = O(n^k)$, where $k < \log_b a$, then $T(n) = \Theta\left(n^{\log_b a}\right)$
2. If $f(n) = \Theta(n^k \log^j n)$, where $k = \log_b a$, then $T(n) = \Theta\left(n^k \log^{j+1} n\right)$
3. If $f(n) = \Omega(n^k)$, where $k > \log_b a$, then $T(n) = \Theta\left(f(n)\right)$

$$T(n) = 2T(n/2) + \log n$$ (heap construction)

Solution: $\log_b a = \log_2 2 = 1$, so case 1 says $T(n) = O(\log n)$.

Correctness of Algorithms

We want to have algorithms that are **correct** and **efficient**

Correctness has highest priority.

You want to be sure that your program does what you want.

Invariants

- Invariants are a powerful tool to show correctness of an algorithm/program.
- An invariant is a property of an algorithm that holds during the execution of the program.

  Define: **Preconditions**, **Invariants**, and **Postconditions**
- It is often used to show correctness of loops.
- Often it is non-trivial to find an invariant.

Invariants: Example

Consider the following while loop

```
int x = 10;
while ( x < 20 ) {
        x = x + 1;
}
```

Precondition: Before entering the while-loop $x < 20$ holds.

Invariant: During the execution of the while-loop $x \leq 20$ holds.

Postcondition: After execution of the while-loop $x \leq 20$,
but not $x < 20$ holds.
This implies that $x = 20$ holds.

Binary Search

- There is a strategy you can use to limit the number of guesses that you have to make.
- It only works if the array of elements where you are searching for an item is sorted.
- It is an example of Divide and Conquer.

Binary Search: Let's play a game

$a = \{1, \dots, 15\}$ consists of all integers from $1, \dots, 15$.

Player 1 picks a secret number x of a.

Player 2 has to guess x querying in each step a number y.

Answer of Player 1 is either

- found if $x = y$
- x is greater than y
- x is smaller than y

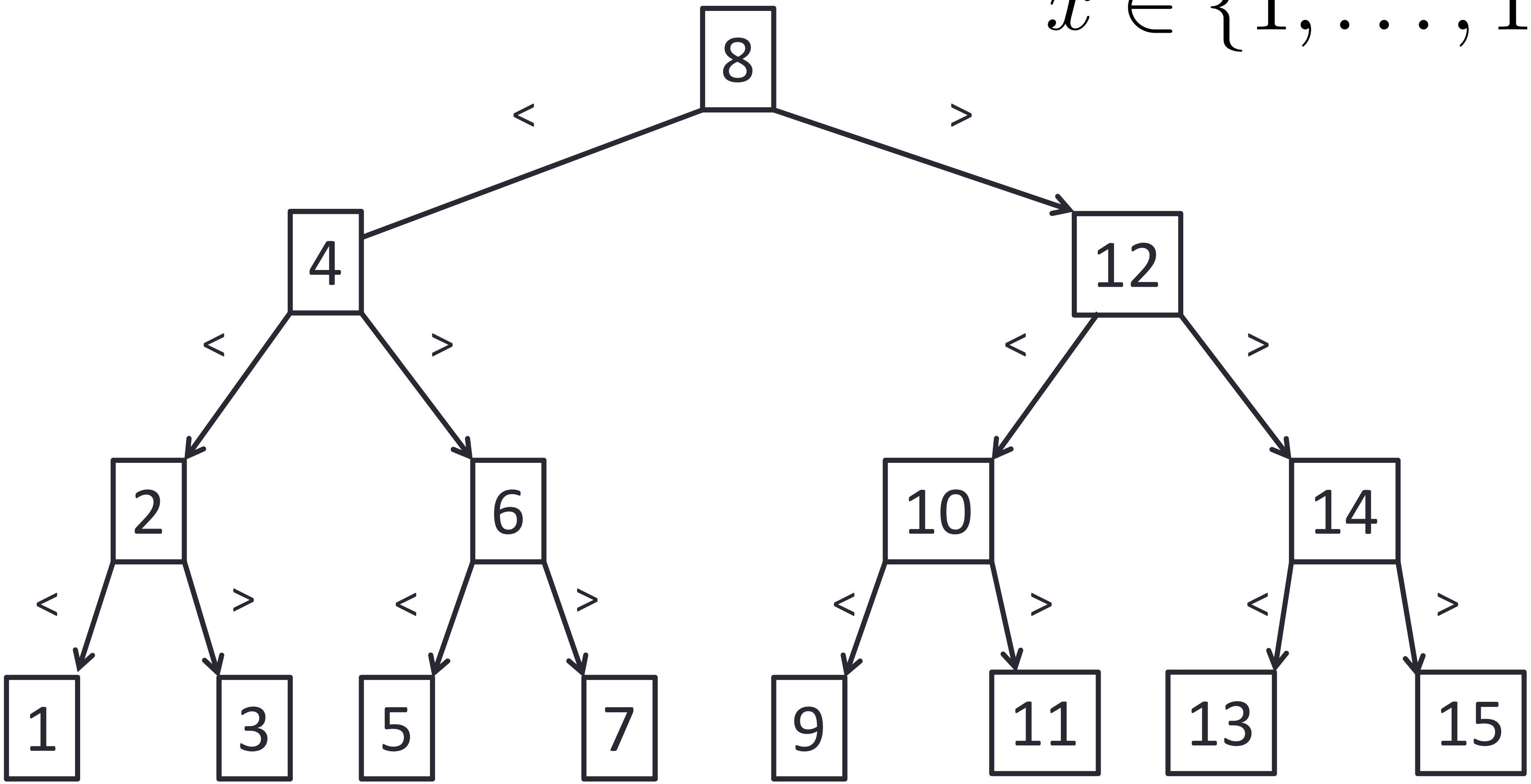
If you are Player 2:
What is your strategy to use the smallest number of queries to reveal x?

Binary Search: Strategy

1. Start with a sorted list
2. Find the midpoint of the part your are searching
3. Compare our search value to the midpoint
   - if the midpoint is our search value we stop!
   - if our number is smaller we now only look between the start of the list and this point
   - if our number is bigger we now only look between this point and the end of the list.
4. If the part we are searching is not empty, go back to step 2
5. If the part we are searching is empty, the value is not in the list

Binary Search: Example

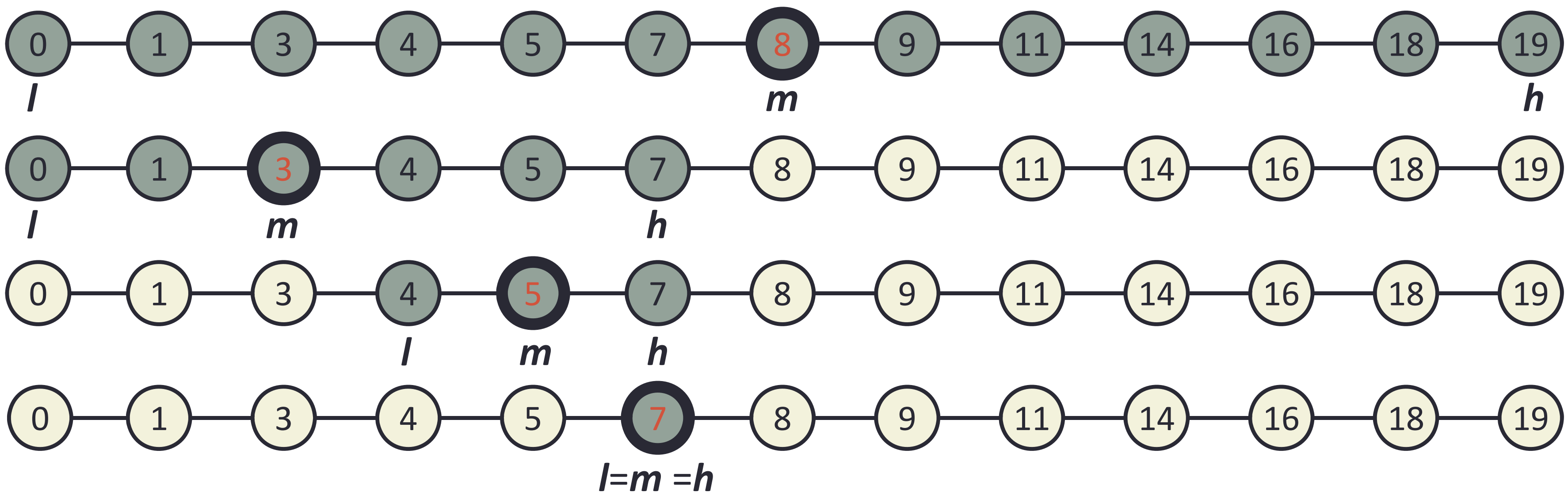
$$x \in \{1, \ldots, 15\}$$



At most 3 comparisons to determine x

Binary Search: Example

Binary search can perform operation find(k) on an array-based sequence, sorted by key, in O(log n) steps



Example: find(7)

Binary Search: Problem Statement

- Given: A sorted array $a[1 \dots n]$ of pairwise distinct elements, i.e.

  $a[1] < a[2] < \cdots < a[n]$, and an element x

  $a[0] = -\infty$ and $a[n+1] = \infty$
- Find: Index i such that $a[i-1] < x \leq a[i]$

Binary Search: Procedure

- Choose index $m \in [1 \dots n]$
- Compare x with $a[m]$
- If $x = a[m]$ we are done
- If $x < a[m]$, search in the part of the array before $a[m]$.
- If $x > a[m]$, search in the part of the array after $a[m]$.

Binary Search: Implementation

- Use two indices l and r.
- Maintain the invariant

$$(I)\ 0 \leq l < r \leq n + 1 \text{ and } a[l] < x < a[r]$$

- Start with $l = 0$ and $r = n + 1$.
- Choose m in the middle of the interval defined by l and r.
- If $x \neq a[m]$, change l or r accordingly.
- If l and r are consecutive indices then x is not contained in the array.

Binary Search: Pseudocode

$(\ell, r) := (0, n+1)$
while *true* **do**
  invariant I // i.e., invariant (I) holds here
  if $\ell + 1 = r$ **then return** "$a[\ell] < x < a[\ell+1]$"
  $m := \lfloor (r+\ell)/2 \rfloor$ // $\ell < m < r$
  $s := compare(x, a[m])$ // -1 if $x < a[m]$, 0 if $x = a[m]$, $+1$ if $x > a[m]$
  if $s = 0$ **then return** "x *is equal to* $a[m]$";
  if $s < 0$
    then $r := m$ // $a[\ell] < x < a[m] = a[r]$
    else $\ell := m$ // $a[\ell] = a[m] < x < a[r]$

Choose the middle of the current interval

Binary Search: Invariant Part 1

$$0 \le l < r \le n + 1$$

- Loop is entered with $0 \le l < r \le n + 1$
- If $l + 1 = r$, we stop.
  Otherwise, $l + 2 \le r$ and hence $l < m < r$.
  Implies that m is a legal array index
- If $x = a[m]$, we stop.
  Otherwise we set either $r = m$ or $l = m$ and hence
  $0 \le l < r \le n + 1$ at the end of the loop.

Binary Search: Invariant Part 2

$$a[l] < x < a[r]$$

- Loop is entered with $a[l] < x < a[r]$
- If $l + 1 = r$, we stop.
  Otherwise, $l + 2 \leq r$ and hence $l < m < r$.
- If $x = a[m]$, we stop.
- If $x < a[m]$, we set $r = m$ which implies $a[l] < x < a[r]$ at the end of the loop.
- If $x > a[m]$, we set $l = m$ which implies $a[l] < x < a[r]$ at the end of the loop.

Binary Search: Termination

- If an iteration is not the last one, we either increase l or decrease r.
- Hence $r - l$ decreases.
- Implies that the search terminates.

Binary Search: Runtime Complexity

$$T(n) = \begin{cases} c & \text{if } n < d \\ aT(n/b) + f(n) & \text{if } n \geq d \end{cases}$$

The Master Theorem:

1. If $f(n) = O(n^k)$, where $k < \log_b a$, then $T(n) = \Theta\left(n^{\log_b a}\right)$
2. If $f(n) = \Theta(n^k \log^j n)$, where $k = \log_b a$, then $T(n) = \Theta\left(n^k \log^{j+1} n\right)$
3. If $f(n) = \Omega(n^k)$, where $k > \log_b a$, then $T(n) = \Theta\left(f(n)\right)$

Recurrence formula for binary search:

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

Solution: $\log_b a = \log_2 1 = 0$, so case 2 says $T(n) = O(\log n)$.

Binary Search: Program

```
// return the position of value in the array A or -1 if it is not present
int binary_search(int A[], int length, int value)
{
    // work out the first and last array indexes to search
    int imin = 0 ; int imax = length - 1;

    // search while [imin,imax] is not empty
    while (imax >= imin)                          // Why isn't this a for loop?
    {
        // calculate the midpoint using integer division
        int midpt = (imin + imax) / 2;
        if(A[midpt] == value)
        {
            return midpt; // We found it!
        }
        // Now have to deal with subarrays.
        if (A[midpt] < value)
        {
            imin = midpt + 1; // change min index to search upper subarray
        } else
        {
            imax = midpt - 1; // change max index to search lower subarray
        }
    }
    // value was not found - why does this work?
    return -1;
}
```

Thinking recursively

Finding the recursive structure of the problem is the hard part.

- **Common patterns:**
  - divide in half, solve one half
  - divide in sub-problems, solve each sub-problem recursively, “merge”
  - solve one or several problems of size n-1
  - process first element, recurse on the remaining problem
- **Recursion**
  - functional: function computes and returns result
  - procedural: no return result (function returns void)
    The task is accomplished during the recursive calls.
- **Recursion**
  - exhaustive
  - non-exhaustive: stops early

Recursive algorithm: Template

To solve a problem recursively

- break into smaller problems;
- solve sub-problems recursively;
- assemble sub-solutions.

```
recursive-algorithm(input) {
// base case
if (isSmallEnough(input))
   compute the solution and return it
else
// recursive case
   break input into simpler instances input1, input 2,...
   solution1 = recursive-algorithm(input1)
   solution2 = recursive-algorithm(input2)
   ...
   figure out solution to this problem from solution1, solution2,...
   return solution
}
```

Exercise: Blob Check

Problem: you have a 2-dimensional grid of cells, each of which may be filled or empty. Filled cells that are connected form a “blob” (for lack of a better word).

Objective: Write a recursive method that returns the size of the blob containing a specified cell (i,j).

	0	1	2	3	4
0				x	x
1				x	
2	x	x			
3	x	x	x		x
4	x	x			x

BlobCount(0,3) = 3

BlobCount(0,4) = 3

BlobCount(3,4) = 2

BlobCount(4,0) = 7

Exercise: Blob Check

Solution: essentially you need to check the current cell, its neighbours, the neighbours of its neighbours, and so on.

When calling BlobCheck(i,j)

- (i,j) may be outside of grid
- (i,j) may be EMPTY
- (i,j) may be FILLED

When you write a recursive method, always start from the base case

Given a call to BlobCkeck(i,j): when is there no need for recursion, and the function can return the answer immediately?

Exercise: Blob Check

```
blobCheck(i,j):
    if (i,j) is FILLED          -> add 1 (for the current cell)
                                -> count its 8 neighbours

// first check base cases
if (outsideGrid(i,j)) return 0;
if (grid[i][j] != FILLED) return 0;
blob_counter = 1;
for (l = -1; l <= 1; l++)
    for (k = -1; k <= 1; k++)
        // skip of middle cell
        if (l==0 && k==0) continue;
        // count neighbours that are FILLED
        if (grid[i+l][j+k] == FILLED) blob_counter ++;
```

Does this work?

Exercise: Blob Check

```
blobCheck(i,j):
    if (i,j) is FILLED        -> add 1 (for the current cell)
                              -> count its 8 neighbours

// first check base cases
if (outsideGrid(i,j)) return 0;
if (grid[i][j] != FILLED) return 0;
blob_counter = 1;
for (l = -1; l <= 1; l++)
    for (k = -1; k <= 1; k++)
        // skip of middle cell
        if (l==0 && k==0) continue;
        // count neighbors that are FILLED
        if (grid[i+l][j+k] == FILLED) blob_counter ++;
```

- It does not count the neighbours of the neighbours, and their neighbours, and so on.
- Instead of adding +1 for each neighbour that is filled, need to count its blob recursively.

Exercise: Blob Check

```
blobCheck(i,j):
    if (i,j) is FILLED      -> add 1 (for the current cell)
                            -> count blobs of its 8 neighbours

// first check base cases
if (outsideGrid(i,j)) return 0;
if (grid[i][j] != FILLED) return 0;
blob_counter = 1
for (l = -1; l <= 1; l++)
    for (k = -1; k <= 1; k++)
        if (l==0 && k==0) continue;  // skip of middle cell
        blob_counter += blobCheck(i+k, j+l);
```

Does this work?

Exercise: Blob Check

```
blobCheck(i,j):
    if (i,j) is FILLED      -> add 1 (for the current cell)
                            -> count blobs of its 8 neighbours

// first check base cases
if (outsideGrid(i,j)) return 0;
if (grid[i][j] != FILLED) return 0;
blob_counter = 1
for (l = -1; l <= 1; l++)
    for (k = -1; k <= 1; k++)
        if (l==0 && k==0) continue;  // skip of middle cell
        blob_counter += blobCheck(i+k, j+l);
```

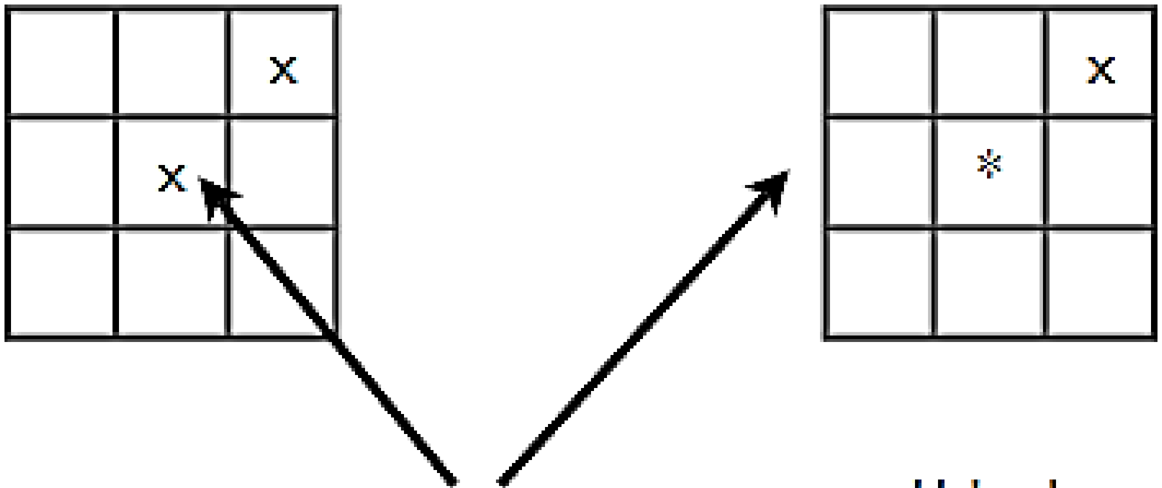
- **Example:** blobCheck(1,1)
  - blobCount(1,1) calls blobCount(0,2)
  - blobCount(0,2) calls blobCount(1,1)

- **Problem:** infinite recursion because of the multiple counting of the same cell.

Exercise: Blob Check

Idea: once you count a cell, mark it so that it is not counted again by its neighbours.

blobCheck(1,1)



count it and mark it

blobc=1

then find counts of neighbors, recursively

+ blobCheck(0,0)
+ blobCheck(0,1)
+blobCheck(0,2)
...

Exercise: Blob Check

- **blobCheck(i,j) works correctly if the cell (i,j) is not filled**
- **blobCheck(i,j) works correctly if the cell (i,j) is filled**
  - mark the cell
  - the blob of this cell is 1 plus the blobCheck of all neighbours
  - because the cell is marked, the neighbours will not see it as FILLED

    => a cell is counted only once
- **Why does this stop?**
  - blobCheck(i,j) will generate recursive calls to neighbours
  - recursive calls are generated only if the cell is FILLED
  - when a cell is marked, it is NOT FILLED anymore,
    so the size of the blob of filled cells is one smaller

    => the blob when calling blobCheck(neighbor of i,j) is smaller that blobCheck(i,j)

Summary

- Divide and conquer is an important concept in algorithmics.
- Master theorem is a general tool for solving standard recursive formulas.
- Invariants are an important tool to show correctness of algorithms/programs.
- Binary Search is effective to locate elements in a sorted array.
  - Algorithm maintains two invariants.
  - It halves the problem size in each iteration.
  - This implies $O(\log n)$ comparisons.

Other references and things to do

- Have a look at the attached references in CloudDeakin.
- Read chapters 4.4, 12.1.3-12.1.4, and 5.1.3 in Data Structures and Algorithms in Java. Michael T. Goodrich, Irvine Roberto Tamassia, and Michael H. Goldwasser, 2014.