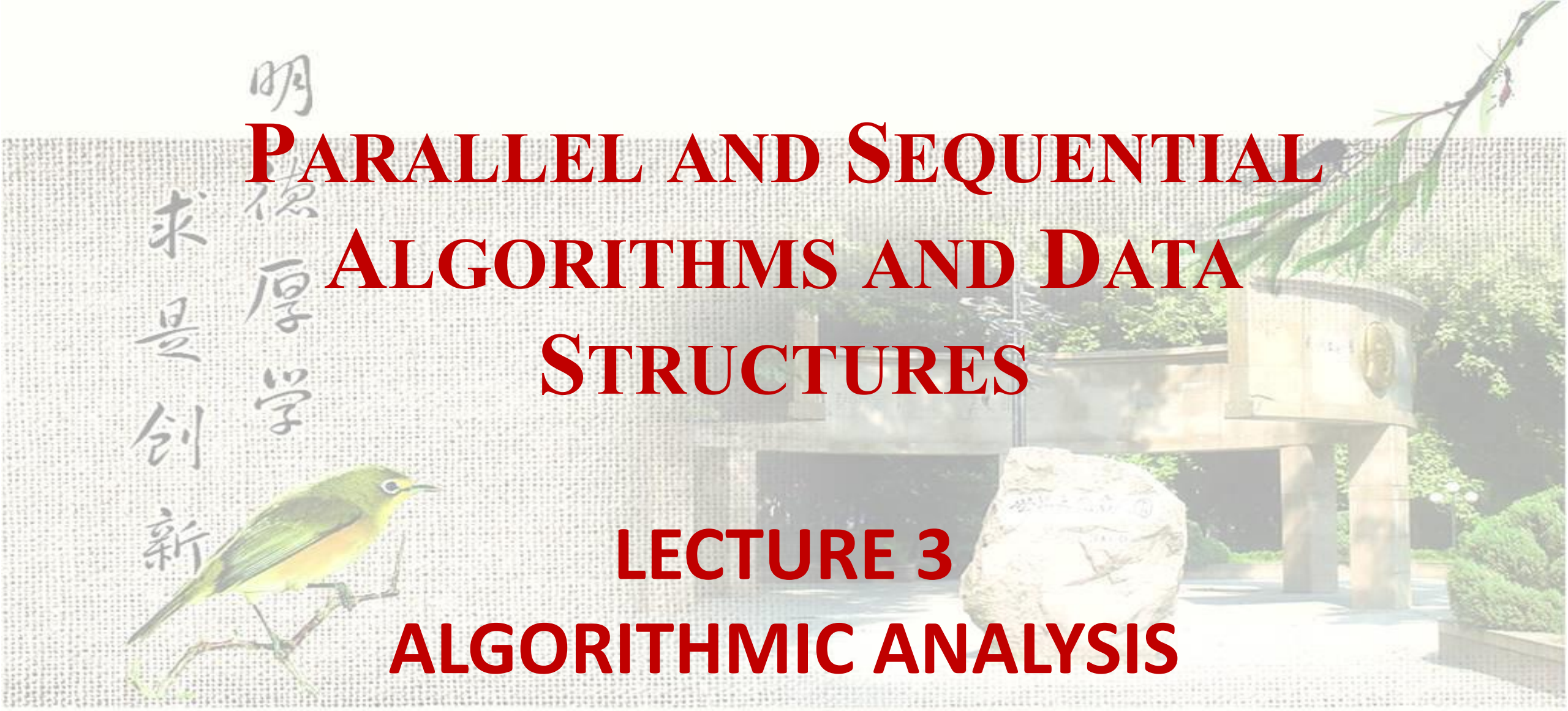


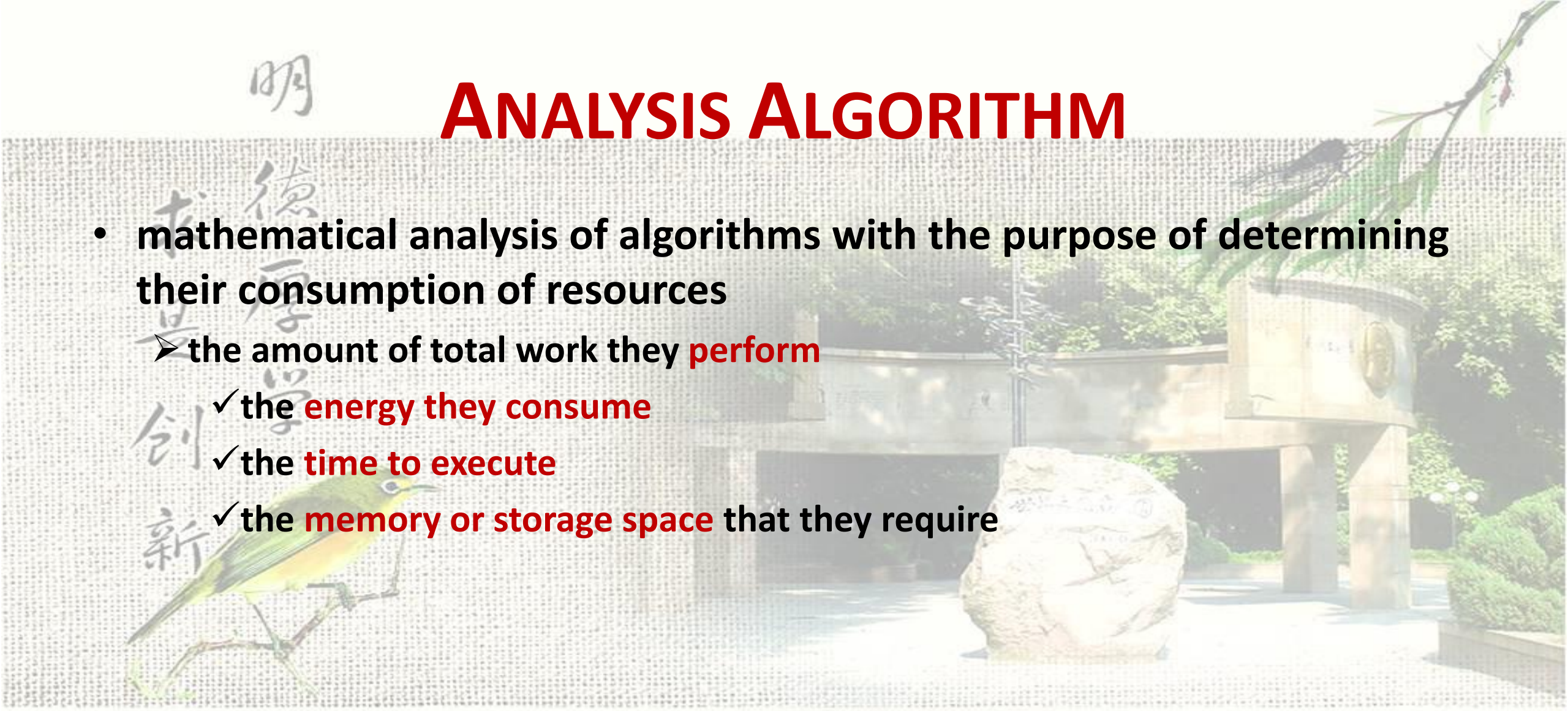
PARALLEL AND SEQUENTIAL ALGORITHMS AND DATA STRUCTURES

LECTURE 3
ALGORITHMIC ANALYSIS

華中科技大學
HUAZHONG UNIVERSITY OF SCIENCE AND TECHNOLOGY

ANALYSIS ALGORITHM

- mathematical analysis of algorithms with the purpose of determining their consumption of resources
  - ➢ the amount of total work they perform
    - ✓ the energy they consume
    - ✓ the time to execute
    - ✓ the memory or storage space that they require



ANALYSIS ALGORITHM

- to be precise enough
  - compare different algorithms
- to be abstract enough
  - don't have to look at minute details of compilers and computer architectures
- To balance between precision and abstraction, rely on:
  - asymptotic analysis
  - cost models (machine-based and language-based cost models)

華中科技大學
HUAZHONG UNIVERSITY OF SCIENCE AND TECHNOLOGY

SYNOPSIS

- **Asymptotics**
- **Cost Models**
- **Recurrences**

華中科技大學
HUAZHONG UNIVERSITY OF SCIENCE AND TECHNOLOGY

Asymptotics

- **Which one is better?**

$$W_A(n) = 2n \log n + 3n + 4 \log n + 5 \qquad W_A(n) = \Theta(n \log n)$$

$$W_B(n) = 6n + 7 \log^2 n + 8 \log n + 9. \qquad W_B(n) = \Theta(n)$$

- **Method**
  - **Ignores "constant factors" and "lower-order terms"**

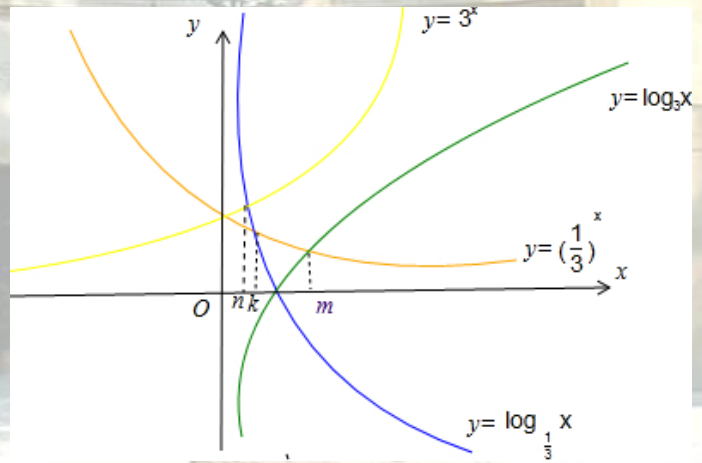
Asymptotics

- Let $f(\cdot)$ and $g(\cdot)$ be two numeric functions, we say that $f(\cdot)$ asymptotically dominates $g(\cdot)$ or that $g(\cdot)$ is asymptotically dominated by $f(\cdot)$ if there exists constants $c>0$ and $n_0>0$ such that for all for all $n \geq n_0$

$$g(n) \leq c \cdot f(n).$$

or

$$\lim_{n \to \infty} \frac{g(n)}{f(n)} \leq c .$$



- When a function $f(\cdot)$ asymptotically dominates $g(\cdot)$, we say that $f(\cdot)$ grows faster than $g(\cdot)$: the absolute value of $g(\cdot)$ does not exceed a constant multiple of $f(\cdot)$ for sufficiently large values

Asymptotics

- For two functions f and g it is possible neither dominates the other
  - $f(n)=n\sin(n)$ and $g(n)=n\cos(n)$ neither dominates since they keep crossing
  - $f(n)=n\sin(n)$ and $g(n)=n\cos(n)$ are dominated by $h(n)=n$
- the relation is
  - Transitive： if f dominates g, and g dominates h, then f dominates h
  - Reflexive： f dominates itself

Big-O, big-Omega, and big-Theta

- **O-notation (upper bounds)**
  - For a function *g(n)*, we say that $g(n) \in O(f(n))$ if there exist positive constants n_0 and *c* such that for all $n > n_0$, we have $g(n) < c \cdot f(n)$.

$$\text{EXAMPLE: } 2n^2 = O(n^3) \quad (c = 1, n_0 = 2)$$

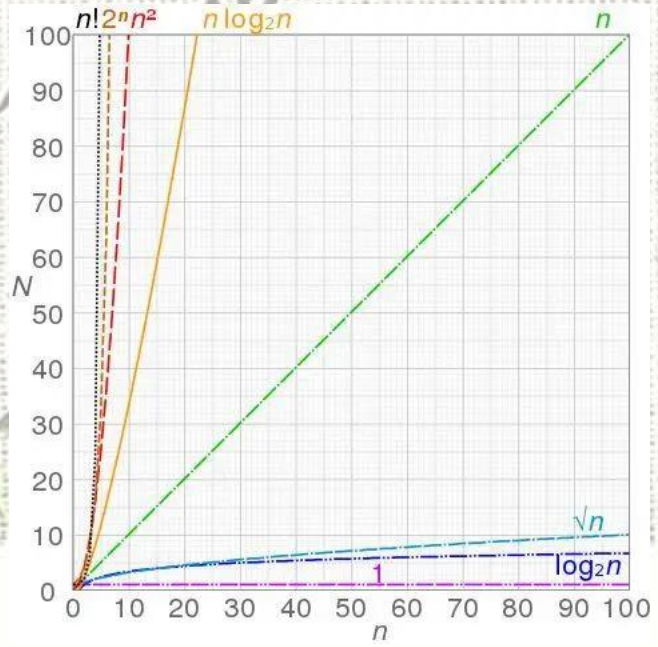
  - If *g(n)* is a finite function (*g(n)* is finite for all *n*), then it follows that there exist constants k_1 and k_2 such that for all $n \geq 1$,

$$g(n) \leq k_1 \cdot f(n) + k_2,$$

$$k_1 = c \text{ and } k_2 = \sum_{i=1}^{n_0} |g(i)|$$

Big-O, big-Omega, and big-Theta

- **Ω-notation (lower bounds)**
  - For a function $g(n)$, we say that $g(n) \in \Omega(f(n))$ if there exist positive constants n_0 and c such that for all $n > n_0$, we have $c \cdot f(n) \leq |g(n)|$.



EXAMPLE: $\sqrt{n} = \Omega(\lg n) \quad (c = 1, n_0 = 16)$

Big-O, big-Omega, and big-Theta

- **Θ-notation (tight bounds)**
  - For a function *g(n)*, we say that $g(n) \in \Theta(f(n))$ if there exist positive constants n_0, c_1 and c_2 such that for all $n > n_0$, we have $c_1 \cdot f(n) \leq |g(n)| \leq c_2 \cdot f(n)$.

EXAMPLE: $\frac{1}{2}n^2 - 2n = \Theta(n^2)$

Big-O, big-Omega, and big-Theta

- The key idea in asymptotic analysis is to understand how the growth rate of two functions compare on large input
- Consider the set of all numeric functions $\mathcal{F}$, and $f \in \mathcal{F}$

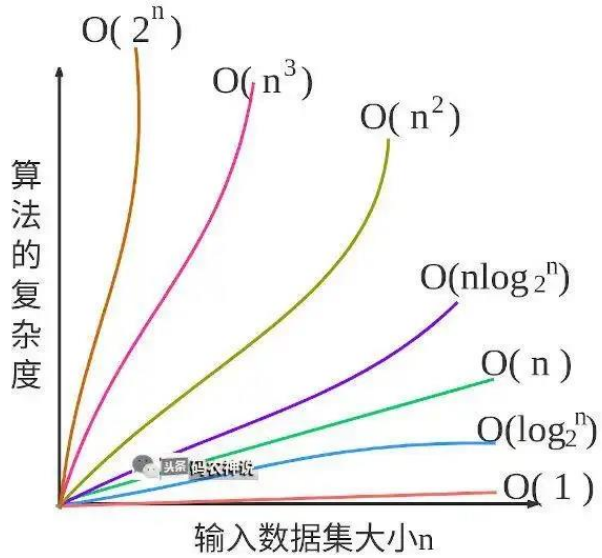
Name		Definition	Intuitively
big-O	:	$O(f) = \{g \in F$ such that f dominates $g\}$	$\leq f$
big-Omega	:	$\Omega(f) = \{g \in F$ such that g dominates $f\}$	$\geq f$
big-Theta	:	$\Theta(f) = O(f) \cap \Omega(f)$	$= f$
little-o	:	$o(f) = O(f) \setminus \Omega(f)$	$< f$
little-omega	:	$\omega(f) = \Omega(f) \setminus O(f)$	$> f$

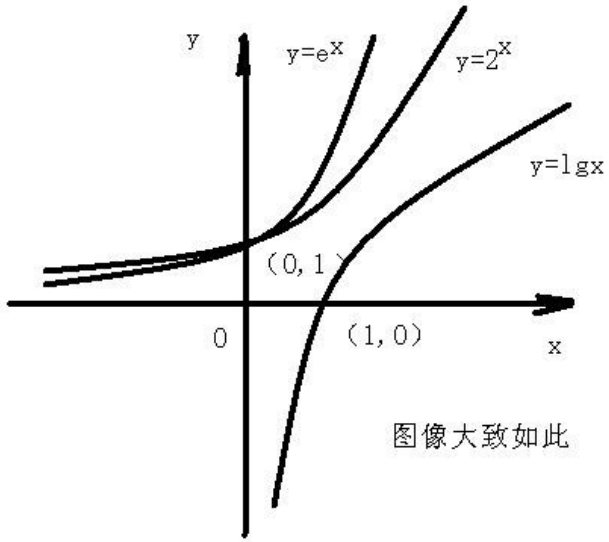
Here "\" means set difference

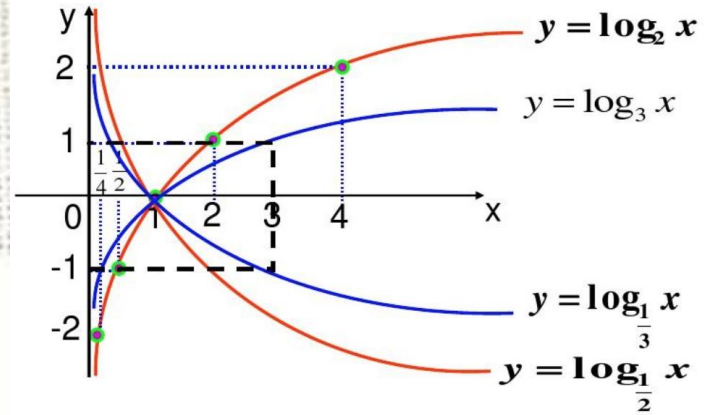
Some Conventions

- **Writing = Instead of ∈**
  - **write $g(n) = O(f(n))$ instead of $g(n) \in O(f(n))$ (or equivalently for Ω and Θ)**

linear	$: O(n)$
sublinear	$: o(n)$
quadratic	$: O(n^2)$
polynomial	$: O(n^k)$, for any constant k.
superpolynomial	$: \omega(n^k)$, for any constant k.
logarithmic	$: O(\lg n)$
polylogarithmic	$: O(\lg^k n)$, for any constant k.
exponential	$: O(a^n)$, for any constant $a > 1$.

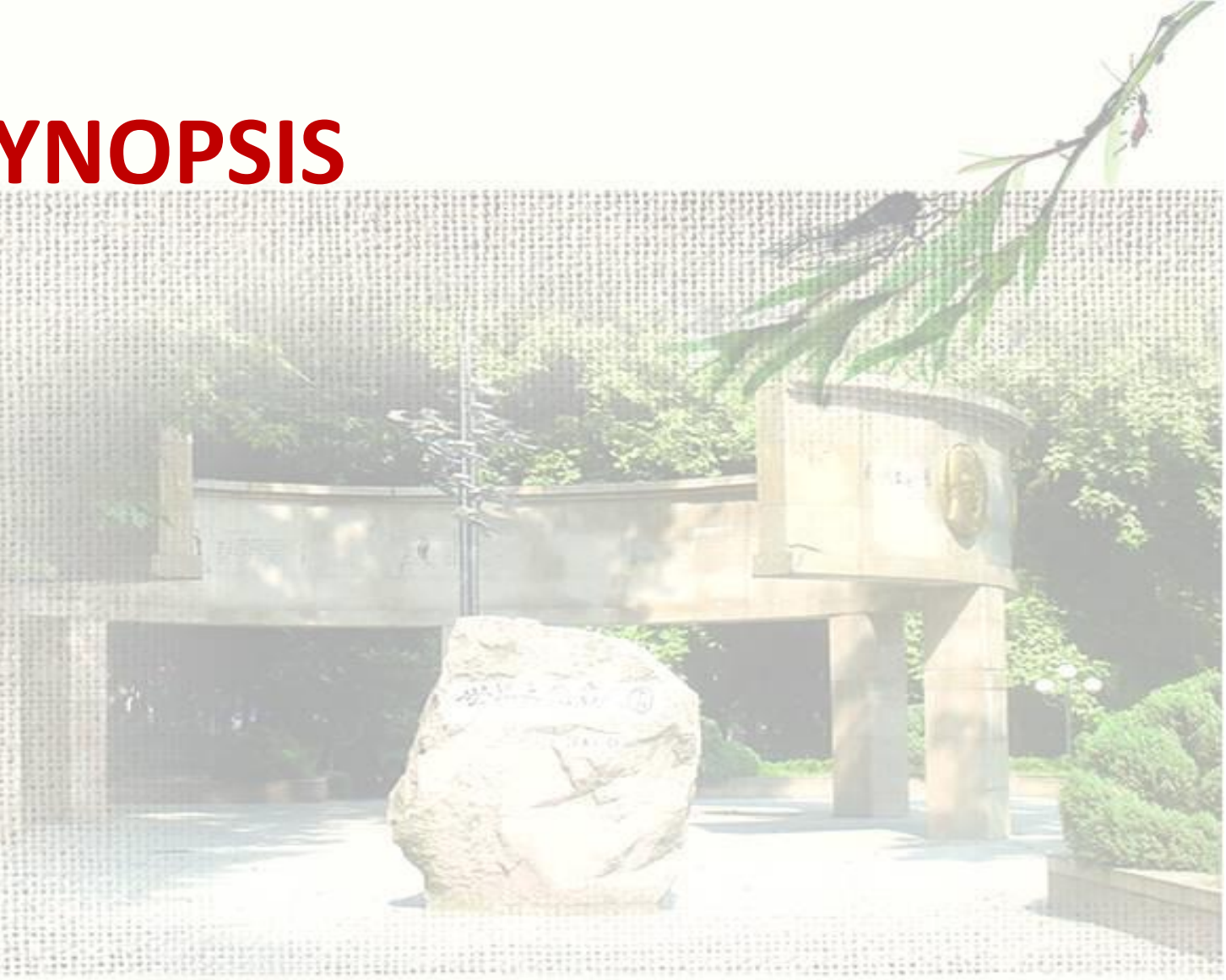






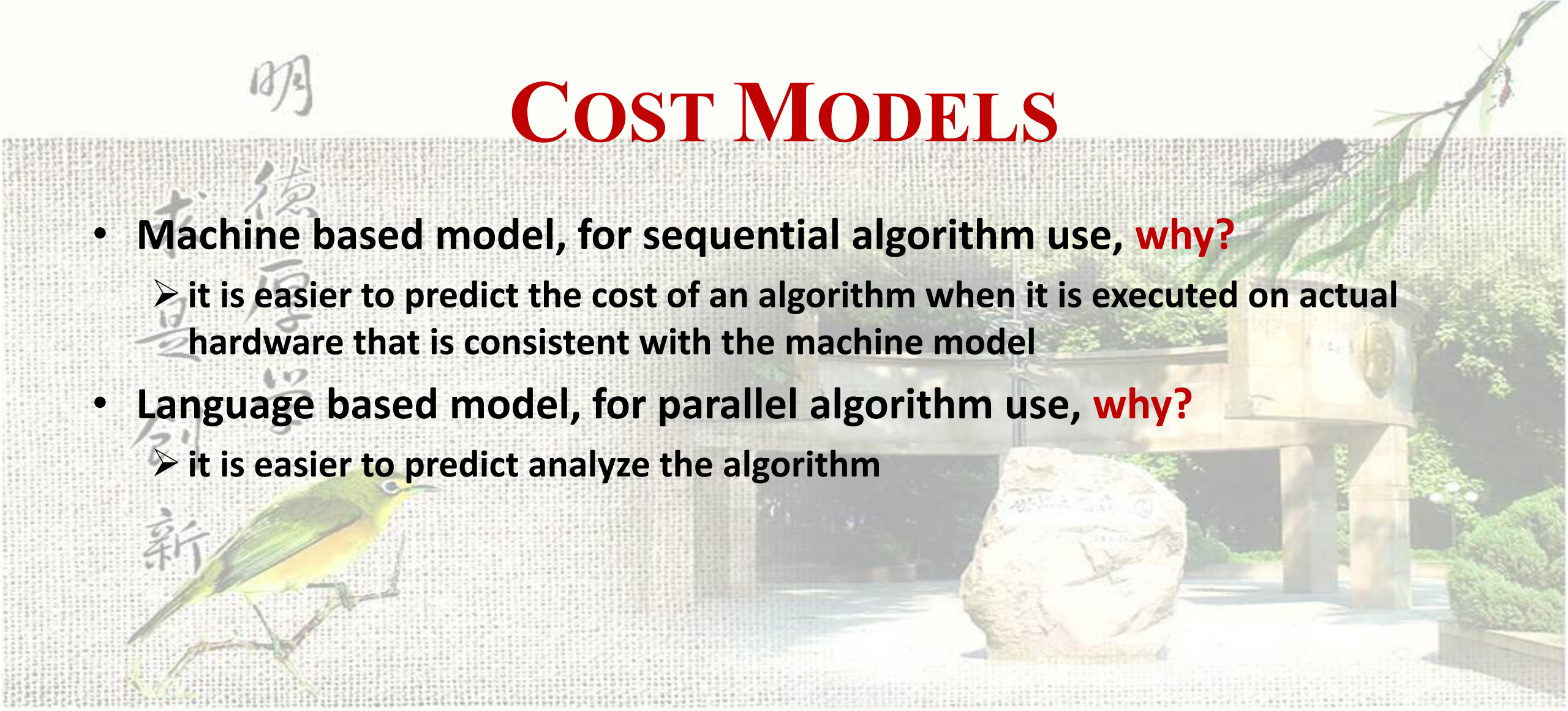
SYNOPSIS

- Asymptotics
- Cost Models
- Recurrences



COST MODELS

- **Machine based model, for sequential algorithm use, why?**
  - **it is easier to predict the cost of an algorithm when it is executed on actual hardware that is consistent with the machine model**
- **Language based model, for parallel algorithm use, why?**
  - **it is easier to predict analyze the algorithm**



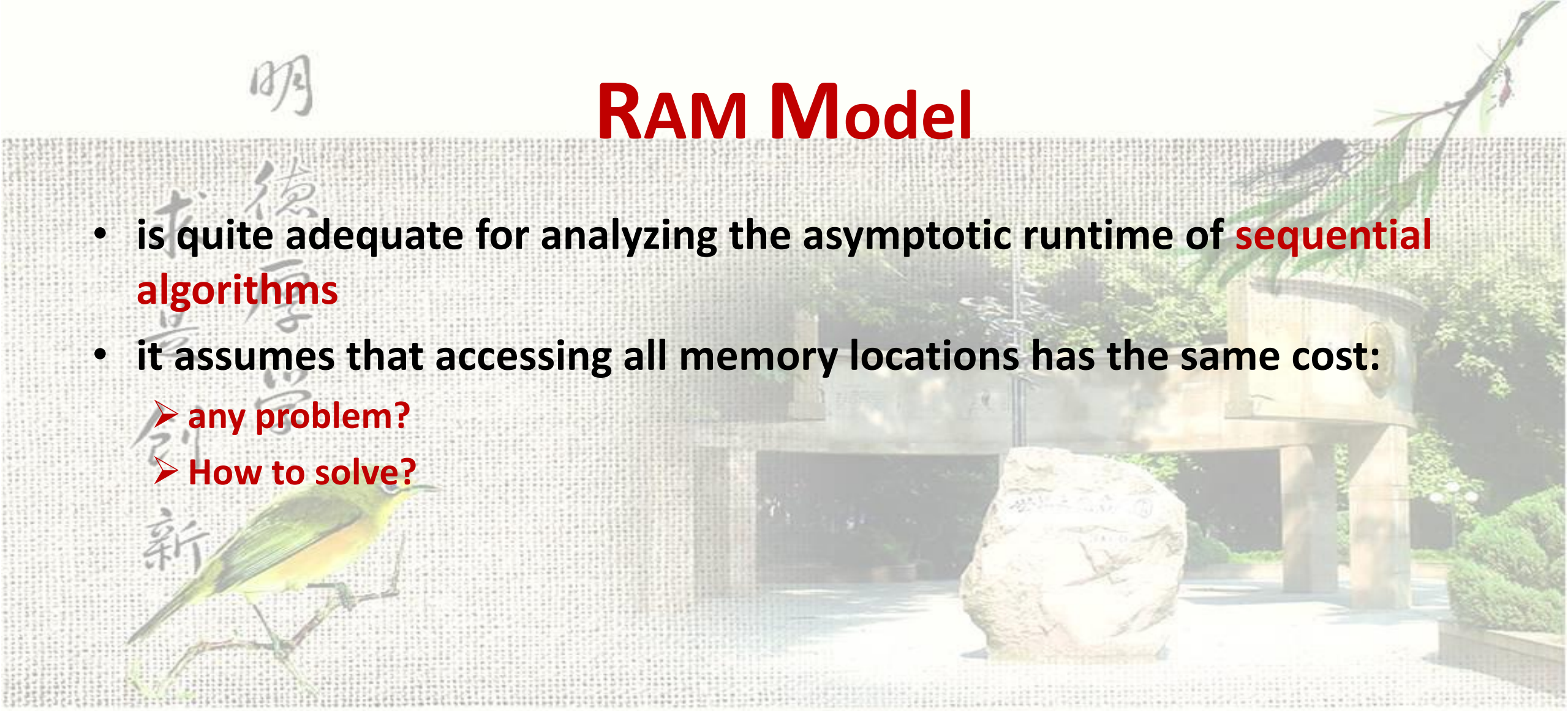
RAM Model

- Random Access Machine (RAM) model
  - assumes a single processor accessing unbounded memory indexed by the nonnegative integers
  - cost of a computation is measured in terms of the number of instructions executed by the machine, and is referred to as time
    - basic arithmetic and logical operations (e.g. +, -, *, and, or, not)
    - reads from and writes to arbitrary memory locations
    - conditional and unconditional jumps to other locations in the code

華中科技大學
HUAZHONG UNIVERSITY OF SCIENCE AND TECHNOLOGY

RAM Model

- is quite adequate for analyzing the asymptotic runtime of sequential algorithms
- it assumes that accessing all memory locations has the same cost:
  - any problem?
  - How to solve?

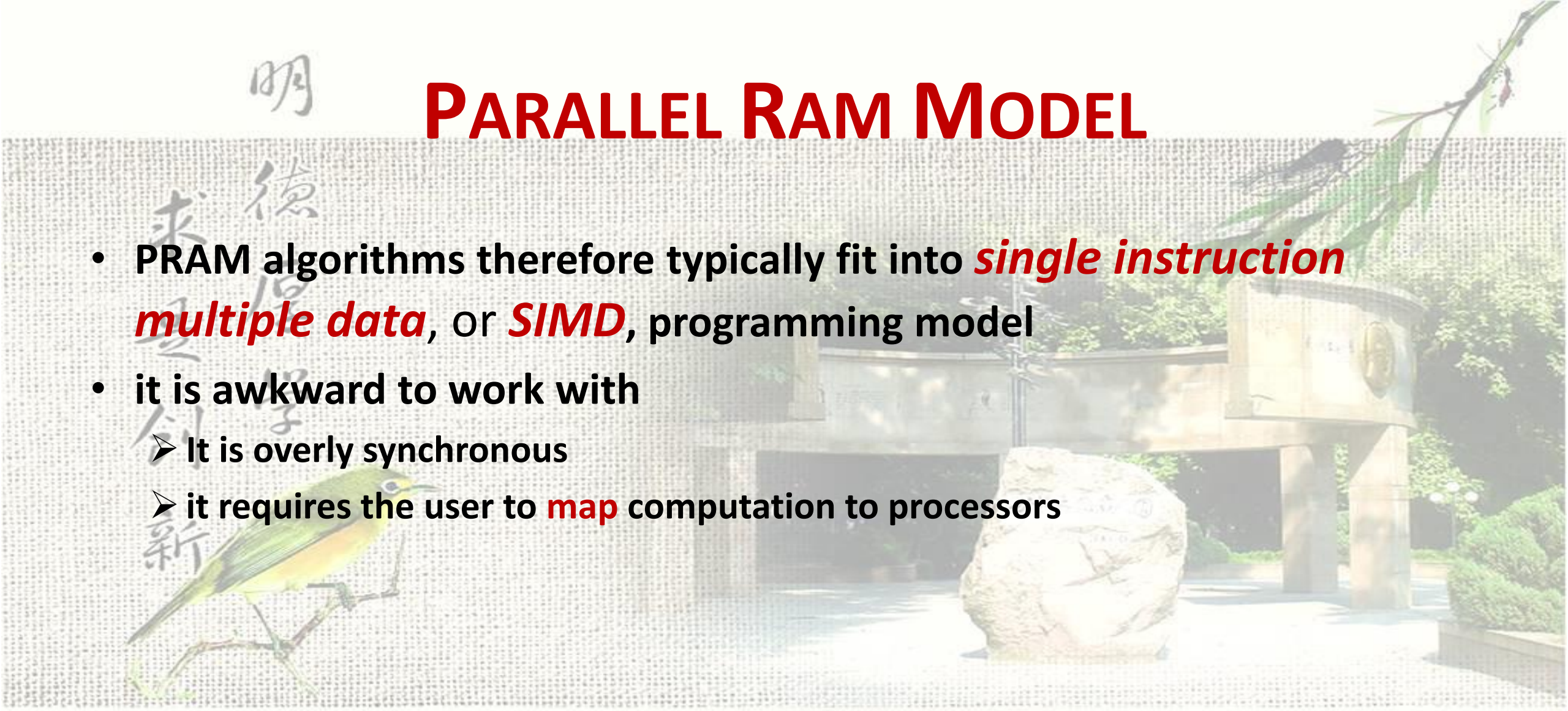


PARALLEL RAM MODEL

- **Parallel Random Access Machine (PRAM)**
  - ➢ **extend the RAM to allow parallelism is simply to use multiple processors which share the same memory**
  - ➢ **A *Parallel Random Access Machine* , or *PRAM* , consist of p sequential random access machines (RAMs) sharing the same memory**
    - ✓ **The number of processors, p, is a parameter of the machine, and each processor has a unique index in $\{0,\ldots,p-1\}$ called the *processor id***
    - ✓ **Processors in the PRAM operate under the control of a common clock and execute one instruction at each time step**
  - ➢ **The PRAM model is most usually used as a synchronous model, where all processors execute the same algorithm and operate on the same data structures**

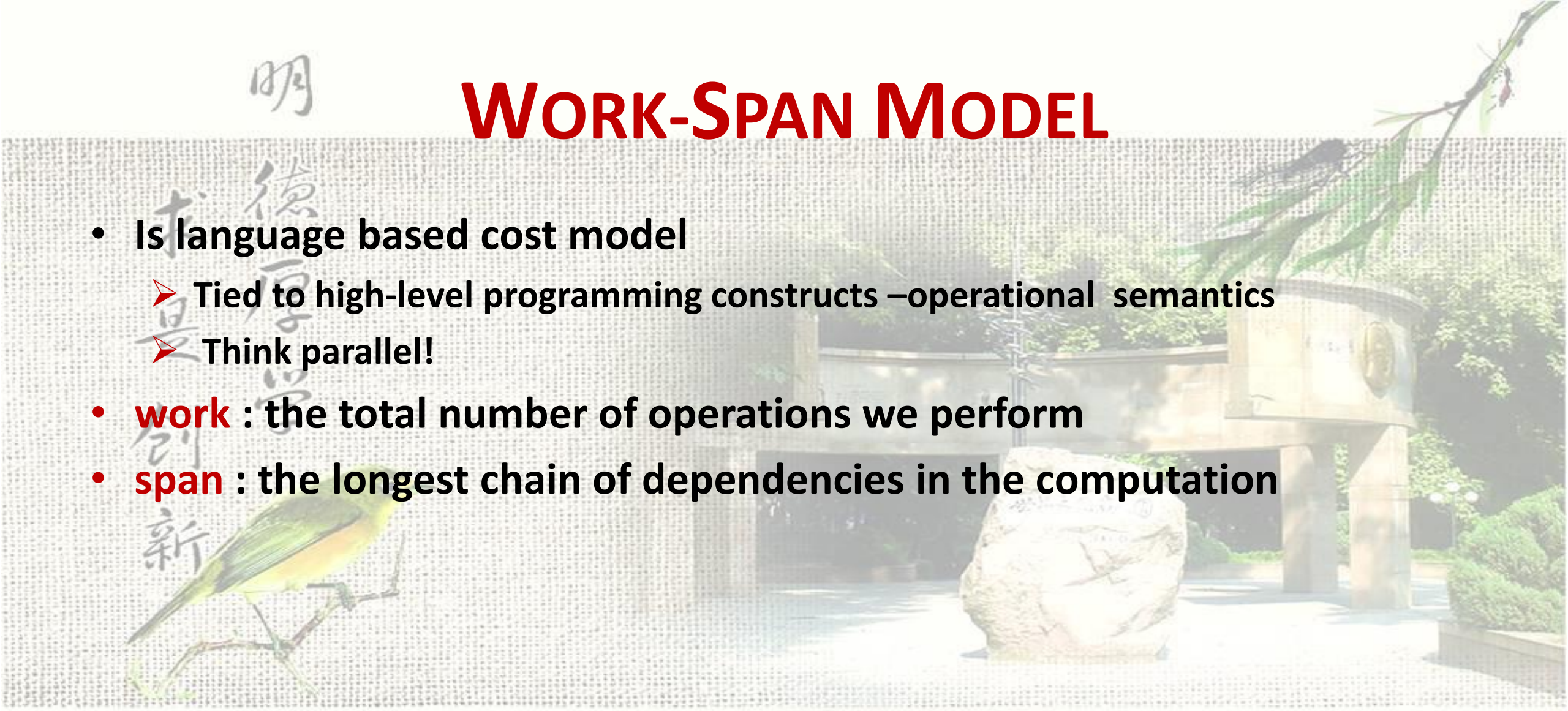
PARALLEL RAM MODEL

- PRAM algorithms therefore typically fit into *single instruction multiple data*, or *SIMD*, programming model
- it is awkward to work with
  - It is overly synchronous
  - it requires the user to map computation to processors

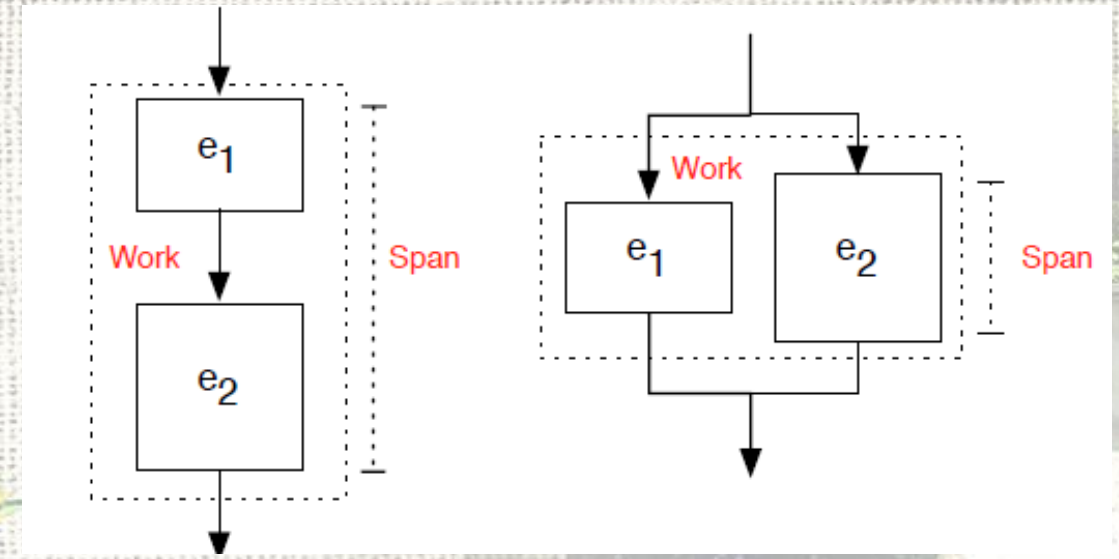


WORK-SPAN MODEL

- Is language based cost model
  - Tied to high-level programming constructs –operational semantics
  - Think parallel!
- **work** : the total number of operations we perform
- **span** : the longest chain of dependencies in the computation



WORK-SPAN MODEL



- ***Eval(e):*** **evaluates the expression *e* and returns the result**
- ***[v/x] e*** **: all free (unbound) occurrences of the variable *x* in the expression *e* are replaced with the value *v***

WORK-SPAN MODEL

$$
\begin{aligned}
W(v) &= 1 \\
W(\texttt{lambda}\ p\ .\ e) &= 1 \\
W(e_1\ e_2) &= W(e_1) + W(e_2) + W([\mathrm{Eval}(e_2)/x]\ e_3) + 1 \\
&\quad \text{where } \mathrm{Eval}(e_1) = \texttt{lambda}\ x\ .\ e_3 \\
W(e_1\ \texttt{op}\ e_2) &= W(e_1) + W(e_2) + 1 \\
W(e_1\ ,\ e_2) &= W(e_1) + W(e_2) + 1 \\
W(e_1 \parallel e_2) &= W(e_1) + W(e_2) + 1 \\
W\begin{pmatrix}\texttt{if}\ e_1 \\ \texttt{then}\ e_2 \\ \texttt{else}\ e_3\end{pmatrix} &= \begin{cases} W(e_1) + W(e_2) + 1 & \text{if } \mathrm{Eval}(e_1) = \texttt{true} \\ W(e_1) + W(e_3) + 1 & \text{otherwise} \end{cases} \\
W\begin{pmatrix}\texttt{let}\ x = e_1 \\ \texttt{in}\ e_2\ \texttt{end}\end{pmatrix} &= W(e_1) + W([\mathrm{Eval}(e_1)/x]\ e_2) + 1 \\
W((e)) &= W(e)
\end{aligned}
$$

WORK-SPAN MODEL

$$
\begin{array}{lcl}
S(v) & = & 1 \\
S(\texttt{lambda}\ p\ .\ e) & = & 1 \\
S(e_1\ e_2) & = & S(e_1) + S(e_2) + S([\mathrm{Eval}(e_2)/x]\ e_3) + 1 \\
 & & \text{where } \mathrm{Eval}(e_1) = \texttt{lambda}\ x\ .\ e_3 \\
S(e_1\ \mathrm{op}\ e_2) & = & S(e_1) + S(e_2) + 1 \\
S(e_1\ ,\ e_2) & = & S(e_1) + S(e_2) + 1 \\
S(e_1 \parallel e_2) & = & \max\left(S(e_1), S(e_2)\right) + 1 \\
S\begin{pmatrix}\texttt{if}\ e_1 \\ \texttt{then}\ e_2 \\ \texttt{else}\ e_3\end{pmatrix} & = & \begin{cases} S(e_1) + S(e_2) + 1 & \mathrm{Eval}(e_1) = \texttt{true} \\ S(e_1) + S(e_3) + 1 & \text{otherwise} \end{cases} \\
S\begin{pmatrix}\texttt{let}\ x = e_1 \\ \texttt{in}\ e_2\ \texttt{end}\end{pmatrix} & = & S(e_1) + S([\mathrm{Eval}(e_1)/x]\ e_2) + 1 \\
S((e)) & = & S(e)
\end{array}
$$

WORK-SPAN MODEL

- **Remark**
  - **there is no || construct in the ML**
  - **do not have the set notation**

$$\{f(x) : x \in A\}$$

$$W(\{f(x) : x \in A\}) = 1 + \sum_{x \in A} W(f(x))$$

$$S(\{f(x) : x \in A\}) = 1 + \max_{x \in A} S(f(x))$$

$$W(\text{map } f \langle s_0, \ldots, s_{n-1} \rangle) = 1 + \sum_{i=0}^{n-1} W(f(s_i))$$

$$S(\text{map } f \langle s_0, \ldots, s_{n-1} \rangle) = 1 + \max_{i=0}^{n-1} S(f(s_i))$$

Upper and Lower Bounds

- **Upper bound**: The maximum asymptotic work (and span) that a given algorithm needs for all inputs of size n
- **Lower bound**: The minimum asymptotic work (and span) that any algorithm for a problem needs for all inputs of size n

PARALLELISM

- For a given W and S, what is the maximum number of processors you can utilize?
- parallelism

$$\mathbb{P} = \frac{W}{S}$$

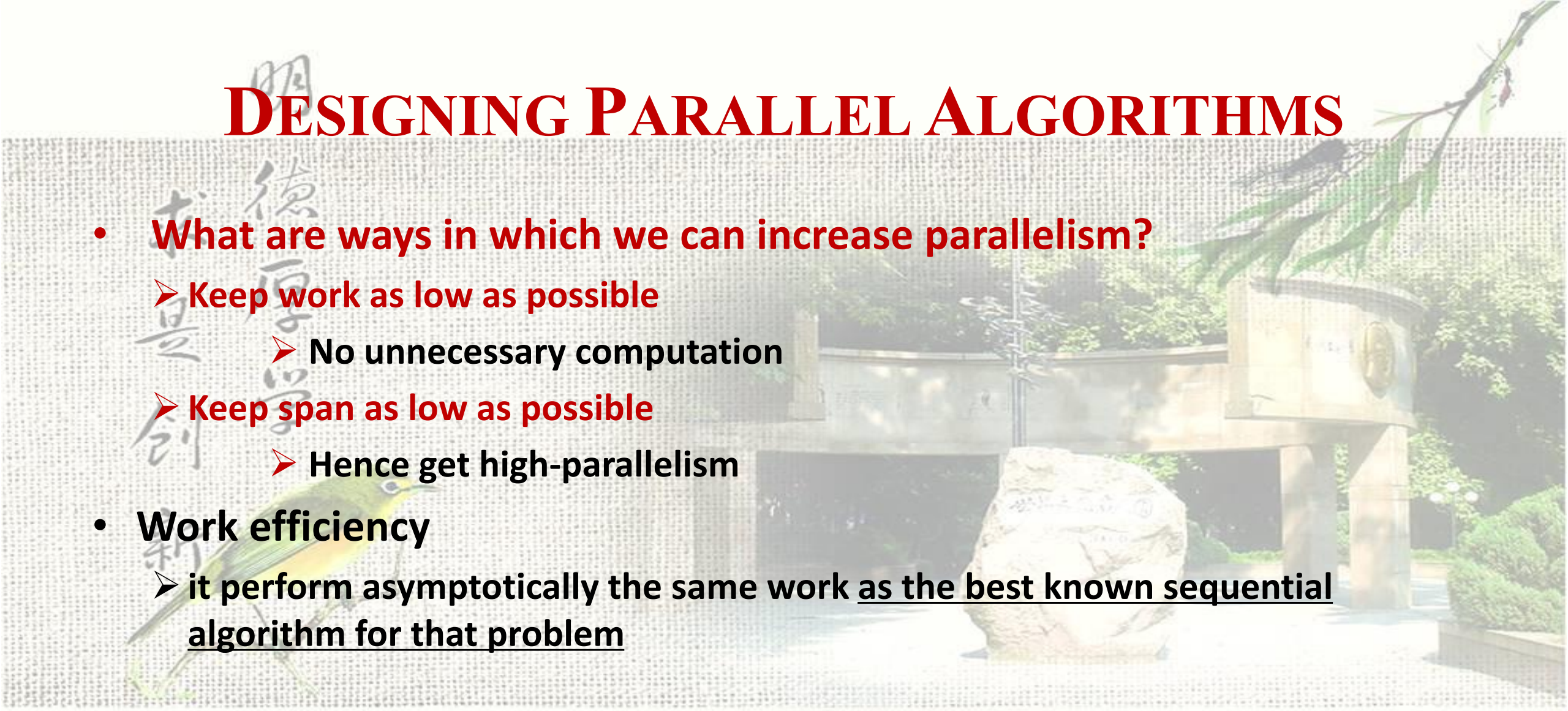
  - Mergesort has $W = \theta(n \log n)$ and $S = \theta(\log^2 n)$

$$\theta(n/\log n)$$

  - The larger the problem is, the higher the parallelism

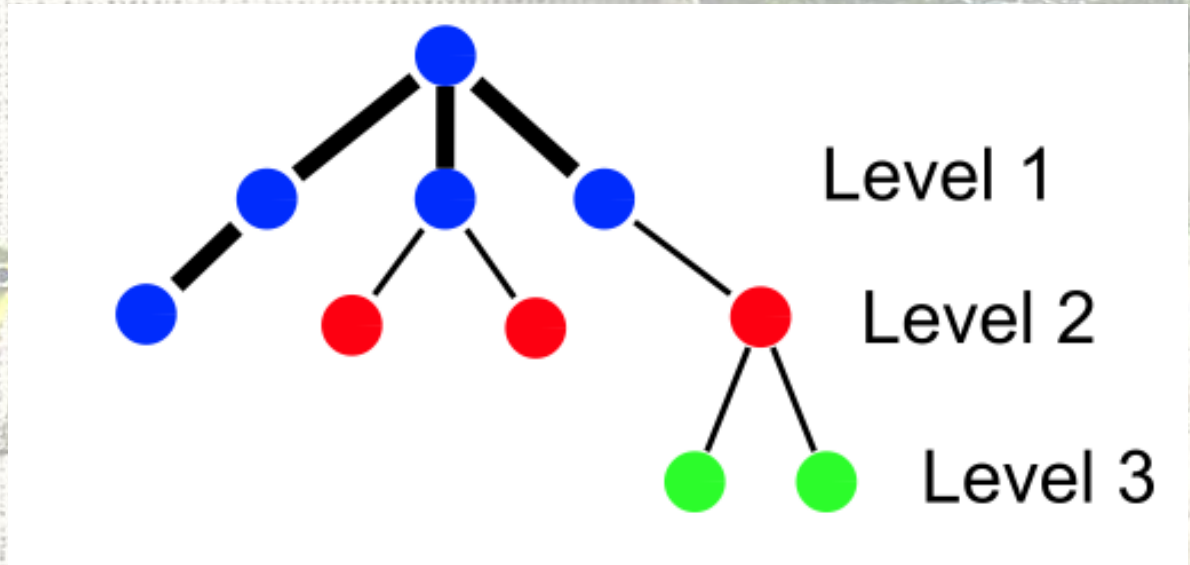
DESIGNING PARALLEL ALGORITHMS

- What are ways in which we can increase parallelism?
  - Keep work as low as possible
    - No unnecessary computation
  - Keep span as low as possible
    - Hence get high-parallelism
- Work efficiency
  - it perform asymptotically the same work as the best known sequential algorithm for that problem



TASK SCHEDULING

- Is scheduling a challenging task? Why?
  - Mapping from a computation graph to processors



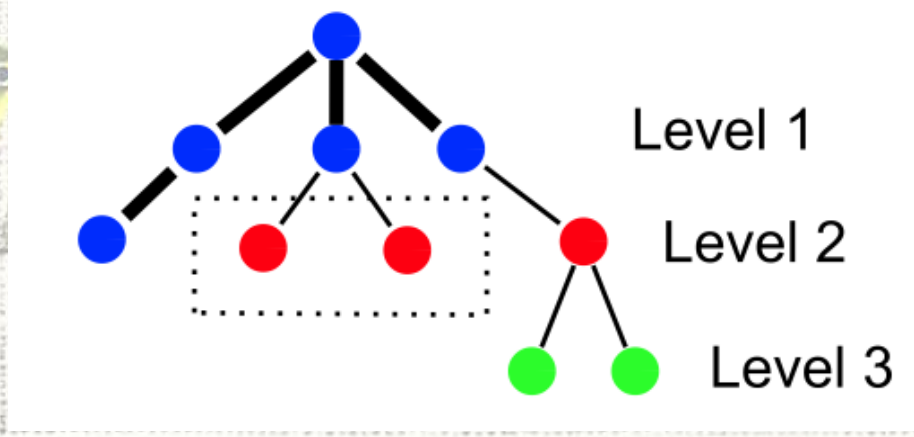
- Can you think of a scheduling algorithm?

TASK SCHEDULING

- The scheduler works by taking all parallel tasks, which are generated dynamically as the algorithm evaluates, and assigning them to processors
  - If only one processor is available, for example, then all tasks will run on that one processor.
  - If two processors are available, the task will be divided between the two
- Schedulers are typically designed to
  - minimize the execution time of a parallel computation
  - minimizing space usage

GREEDY SCHEDULING

- whenever there is a processor available and a task ready to execute, then it assigns the task to the processor and starts running it immediately
  - A greedy scheduler will schedule a ready task on an available processor



A Lower Bound

- Let *Tp* be the "time" needed when using p processors

$$\max(\frac{W}{P}, \quad S) \leq T_p$$

- Why?

An Upper Bound

- With p processors

$$T_p < \frac{W}{P} + S$$

- Why?

TYING THINGS TOGETHER

- **Speed-up is** $\frac{W}{T_p}$
- **Maximum possible speed-up is p.**

$$T_p < \frac{W}{p} + S$$

$$= \frac{W}{p} + \frac{W}{\mathbb{P}}$$

$$= \frac{W}{p}\left(1 + \frac{p}{\mathbb{P}}\right)$$

- **p >>** $\mathbb{P}$ **→ near perfect parallelism**

SYNOPSIS

- Asymptotics
- Cost Models
- Recurrences

Fibonacci

- Here is a recurrence written in SPARC

$$
\begin{aligned}
F(n) = \quad & \texttt{case } n \texttt{ of} \\
& 0 \Rightarrow 0 \\
& \mid 1 \Rightarrow 1 \\
& \mid \_ \Rightarrow F(n-1) + F(n-2) \ .
\end{aligned}
$$

- It has an exact closed form solution

$$F(n) = \frac{\varphi^n - (1-\varphi)^n}{\sqrt{5}}$$

Home work?

- We can write this in asymptotic notation as

$$F(n) = \Theta(\varphi^n)$$

Mergesort Recurrence

- Assuming that the input length is a power of 2, we can write the code for *parallel mergesort algorithm* as follows.

```
msort(A) =
  if |A| ≤ 1 then A
  else
    let (L, R) = msort(A[0 ... |A|/2]) || msort(A[|A|/2 ... |A|])
    in merge(L, R) end
```

Mergesort Recurrence

- write a recurrence for the work of mergesort as:

$$
\begin{aligned}
&W_{msort}(n) = \\
&\quad \texttt{if } n \leq 1 \texttt{ then } c_1 \\
&\quad \texttt{else} \\
&\qquad \texttt{let } (W_L, W_R) = (W_{msort}(n/2), W_{msort}(n/2)) \\
&\qquad \texttt{in } W_L + W_R + W_{merge}(n) + c_2 \texttt{ end}
\end{aligned}
$$

- where the ci are constants. Assuming $W\text{merge}(n)=c3n+c4$, this can be simplified to

$$
\begin{aligned}
W_{msort}(n) = \quad & \texttt{if } n \leq 1 \texttt{ then } c_1 \\
& \texttt{else } 2W_{msort}(n/2) + c_3 n + c_5
\end{aligned}
$$

Cost Analysis with Recurrences

- recurrences are especially common in recursive algorithms
- we can write the work of the merge-sort algorithm with a recurrence of the form

$$W(n) = 2W(n/2)+O(n)$$

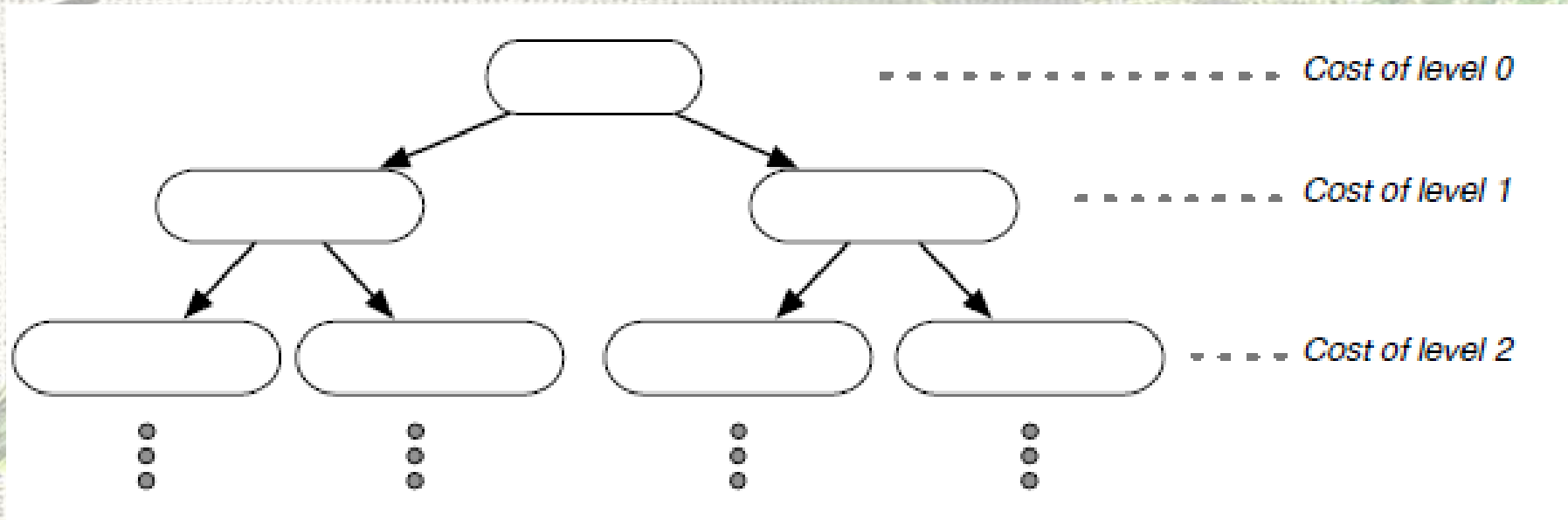
$$S(n) = \max(S(n/2), S(n/2)) + O(\log n) = S(n/2) + O(\log n)$$

（Why？）

- Solving recurrences
  - Tree method (Brick method)
  - Substitution method

THE TREE METHOD

- **Expand recurrence into a tree structure**



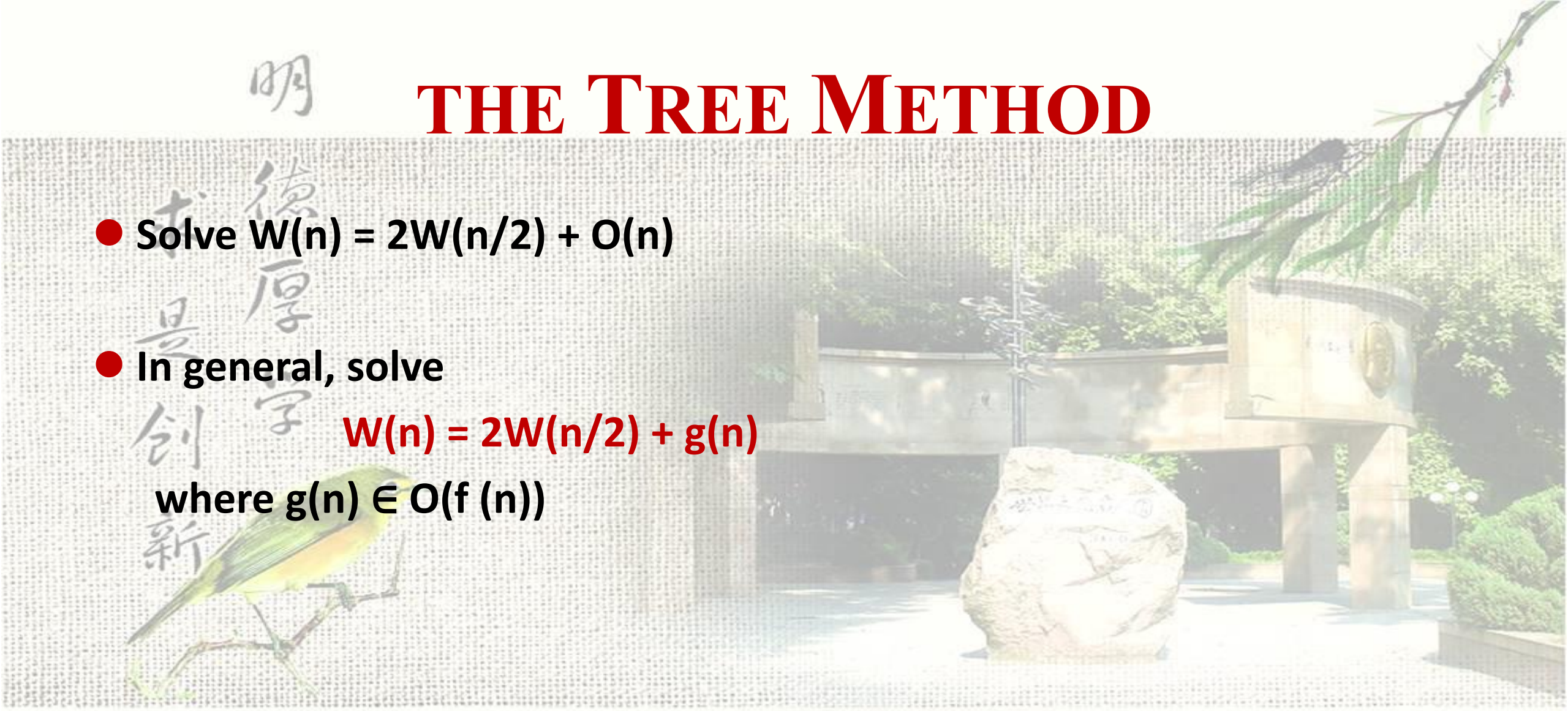
- **Add/Max costs at levels**

THE TREE METHOD

- Solve W(n) = 2W(n/2) + O(n)
- In general, solve

  W(n) = 2W(n/2) + g(n)

  where g(n) ∈ O(f (n))

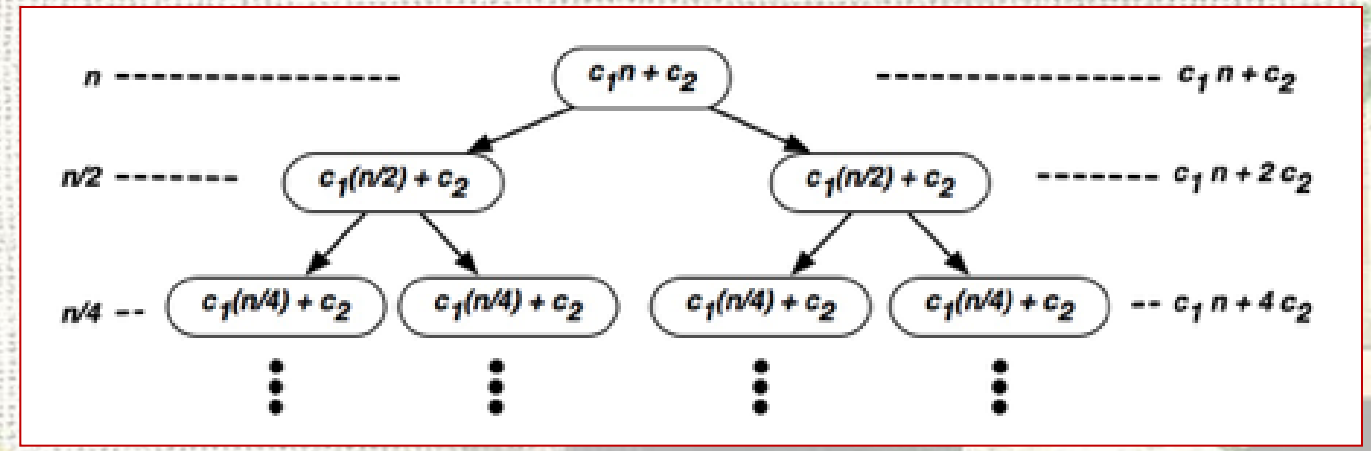


THE TREE METHOD

- $g(n) \in O(f(n)) \Rightarrow g(n) \le c \cdot f(n)$
  - For some $c > 0, N_0 > 0$ and $n \ge N_0$
- $g(n) \le c_1 \cdot f(n) + c_2$ for some c_1, c_2 and $n \ge 1$
  - e.g., $c_1 = c$ and $c_2 = \sum_{i=1}^{N_0} |g(i)|$ (Why?)
- **Solve $W(n) \le 2W(n/2) + c_1 \cdot n + c_2$**
  - f (n) = n in our case

THE TREE METHOD

- Solving $W(n) \leq 2W(n/2) + c_1 \cdot n + c_2$



- Questions:
  - Number of levels in the tree? $1 + \log n$
  - Problem size at level i? $n/2^i$
  - Cost for each node at level i? $c_1(n/2^i) + c_2$
  - Number of nodes at level i? 2^i
  - Total cost at level i?

THE TREE METHOD

- Total cost at level i is at most

$$2^i \cdot \left(c_1 \frac{n}{2^i} + c_2\right) = c_1 \cdot n + 2^i \cdot c_2.$$

- Total cost over all levels is

$$\begin{aligned} W(n) &\le \sum_{i=0}^{\lg n} \left(c_1 \cdot n + 2^i \cdot c_2\right) \\ &= c_1 n(1 + \lg n) + c_2\left(n + \tfrac{n}{2} + \tfrac{n}{4} + \cdots + 1\right) \\ &= c_1 n(1 + \lg n) + c_2(2n - 1) \\ &\in O(n \lg n), \end{aligned}$$

Why?

THE BRICK METHOD

- when the costs at each level grow geometrically, shrink geometrically, or stay approximately equal
  - By recognizing whether the recurrence conforms with one of these cases, we can almost immediately determine the asymptotic complexity of that recurrence
- Look at the cost structure at the levels of the cost tree
  - Root dominated
  - Leaves dominated
  - Balanced

華中科技大學
HUAZHONG UNIVERSITY OF SCIENCE AND TECHNOLOGY

Root-Dominated Cost Trees

- For some $\rho < 1$, for all levels i

$$cost_{i+1} \leq \rho \cdot cost_i$$

++++++++

++++++

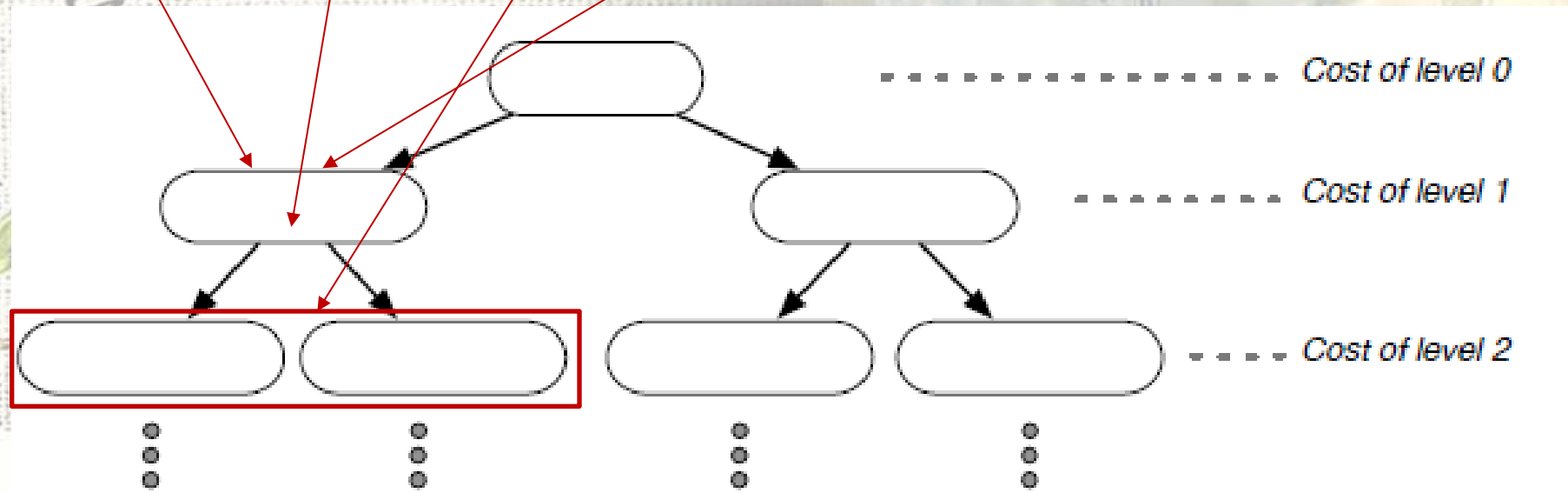
++++

++

- Overall cost is $O(cost_0)$ where d is the depth.

TREE METHOD

- Consider each node v of the recursion tree, let $M(v)$ denote its input size, $C(v)$ denote its cost, and $D(v)$ denote the set of its children.
- There exists constants $a \geq 1$ (base size), $\alpha > 1$ (grown/decay rate)



Root Dominated

- For all nodes v such that $N(v)>a$,

$$C(v) \geq \alpha \sum_{u \in D(v)} C(u),$$

  - i.e., the cost of the parent is at least a constant factor greater than the sum of the costs of the children.
- In this case, the total cost is dominated by the root, and is upper bounded by $(\alpha/(\alpha-1))$ times the cost of the root.

Consider each node v of the recursion tree, let $N(v)$ denote its input size, $C(v)$ denote its cost, and $D(v)$ denote the set of its children

Root dominated example

- Lets consider the recurrence

$$W(n) = 2W(n/2) + n^2.$$

- For a node in the recursion tree of size n we have that the cost of the node is n^2 and the sum of the cost of its children is $(n/2)^2+(n/2)^2=n^2/2$
  - the cost has decreased by a factor of two going down the tree, and hence the recurrence is root dominated
- only consider the cost of the root, and we have that $W(n)=O(n^2)$

LEAVES-DOMINATED COST TREES

- For some ρ > 1, for all levels i

$$cost_{i+1} \geq \rho \cdot cost_i$$

++

++++

++++++

++++++++

- Overall cost is $O(cost_d)$ where d is the depth.

Leaves Dominated

- For all v such that $N(v)>a$,

$$C(v) \leq \frac{1}{\alpha} \sum_{u \in D(v)} C(u),$$

  - i.e., the cost of the parent is at least a constant factor less than the sum of the costs of the children
- In this case, the total cost is dominated by the cost of the leaves, and is upper bounded by $(\alpha/(\alpha-1))$ times the sum of the cost of the leaves
- Most often all leaves have constant cost so we just have to count the number of leaves

Consider each node v of the recursion tree, let $M(v)$ denote its input size, $C(v)$ denote its cost, and $D(v)$ denote the set of its children

Leaf dominated example

- Lets consider the recurrence $W(n) = 2W(n/2) + \sqrt{n}$.
- For a node of size n we have that the cost of the node is √n and the sum of the cost of its two children is $\sqrt{(n/2)} + \sqrt{(n/2)} = \sqrt{2}\sqrt{n}$
  - the cost has increased by a factor of $\sqrt{2}$ going down the tree, and hence the recurrence is leaf dominated
  - Since each recursive call halves the input size, the depth of recursion is going to be $\lg n$
  - Now on each level the recursion is making two recursive calls, so the number of leaves will be $2^{\lg n} = n$
- therefore have that $W(n) = O(n)$

Balanced Cost Trees

- All levels have about the same cost

++++++++

++++++++

++++++++

++++++++

- Overall cost is $O(d \cdot \max_i cost_i)$ where d is the depth.

Balanced example

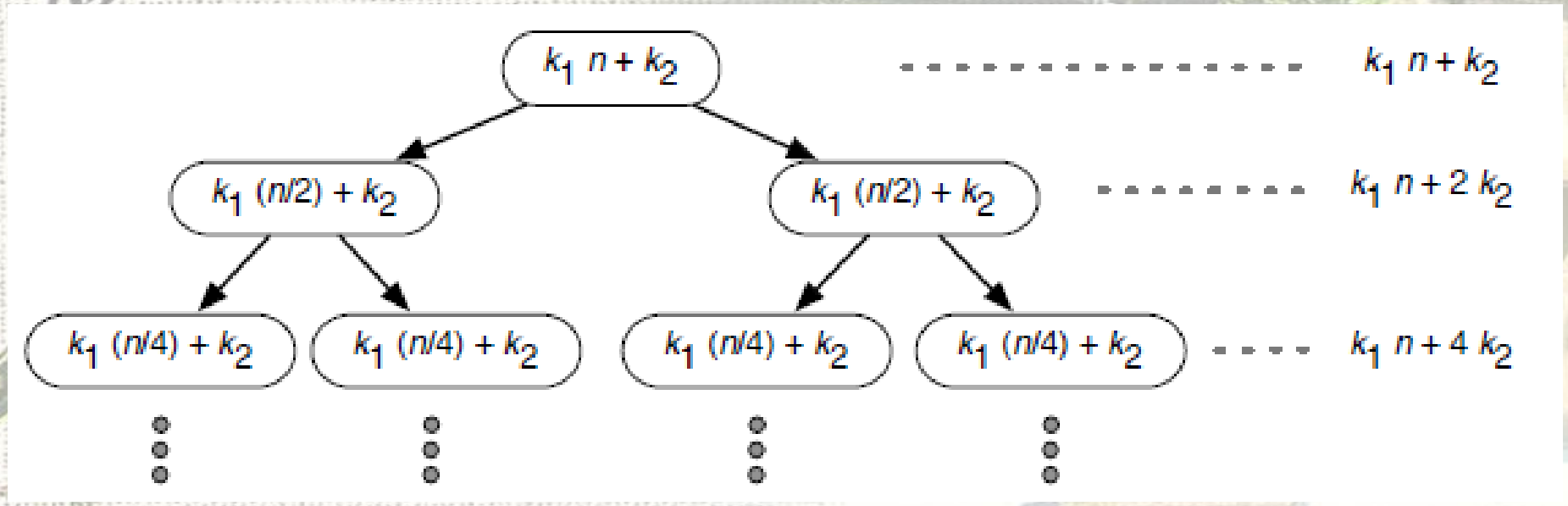
- consider the same recurrence we considered for the tree method

$$W(n) = 2W(n/2) + c_1 n + c_2.$$

- For all nodes we have that the cost of the node is $c1n+c2$ and the sum of the cost of the two children is $(c1n/2+c2)+(c1n/2+c2)=c1n+2c2$
  - The maximum cost of any level is upper bounded by $(c1+c2)n$, since there are at most n total elements across any level (for the $c1n$ term) and at most n nodes (for the $c2n$ term)
  - There are $1+\lg n$ levels, so the total cost is upper bounded by $(c1+c2)n(1+\lg n)$
- $O(n\lg n)$

THE BRICK METHOD

- What type of a cost tree is this?



SUMMARY

- Asymptotics
- Cost Models
- Recurrences

Exercise

1. Prove that for all k, $f(n)=n$ asymptotically dominates $g(n)=\ln^k n$.
2. Prove that asymptotic dominance is transitive.
3. Prove or disprove the following statement: if $g(n) \in \mathcal{O}(f(n))$ and $g(n)$ is a finite function ($g(n)$ is finite for all n), then it follows that there exist constants $k1$ and $k2$ such that for all $n \geq 1$,

$$g(n) \leq k_1 \cdot f(n) + k_2.$$

4. Describe the conditions under which a parallel algorithm would obtain near perfect speedups.

Exercise

5. For each of the following recurrences state whether it is leaf dominated, root dominated or balanced, and then solve the recurrence

$$
\begin{aligned}
W(n) &= 3W(n/2) + n \\
W(n) &= 2W(n/3) + n \\
W(n) &= 3W(n/3) + n \\
W(n) &= W(n-1) + n \\
W(n) &= \sqrt{n}W(\sqrt{n}) + n^2 \\
W(n) &= W(\sqrt{n}) + W(n/2) + n
\end{aligned}
$$