5CCS2FC2: Foundations of Computing II

Recursive Algorithms & Solving Recursion Relations

Week 6

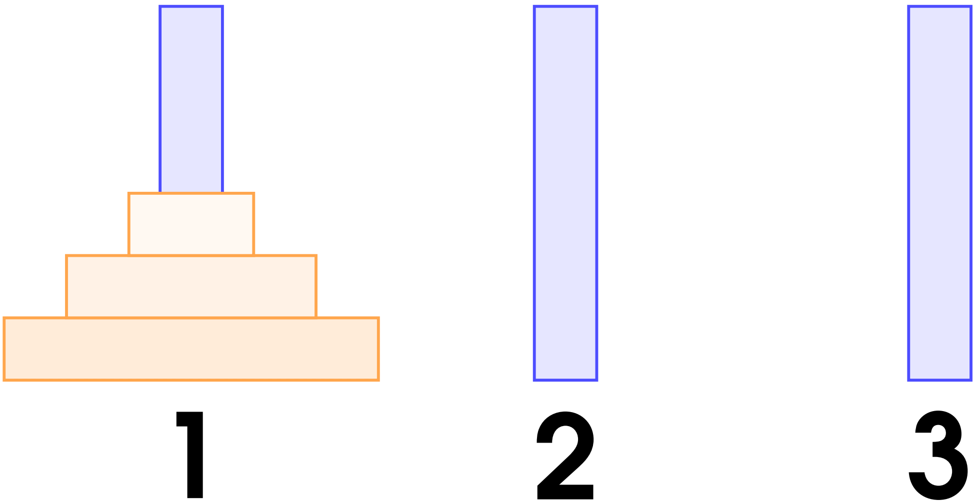
Dr Christopher Hampson

Department of Informatics

King's College London

Warm-up: The Towers of Hanoi

- Move the Tower of Blocks from **Post 1** to **Post 3**
- Can only move one block at a time,
- Cannot place any block on top of a smaller block.



- **The Towers of Hanoi**
  - A legend tells of **64 golden disks** stacked on **three posts** and sealed away in a monestry in Hanoi,
  - Acting out the command of an **ancient prophecy**, monks dilligently move the tower (one disk at a time) from one post to another, following the aforementioned prescribed rules,
  - Upon completion of their task...

...The world will end!

(the legend is likely fabricated by Édouard Lucas who devised the puzzle)

When will the World End?

Objectives for Today

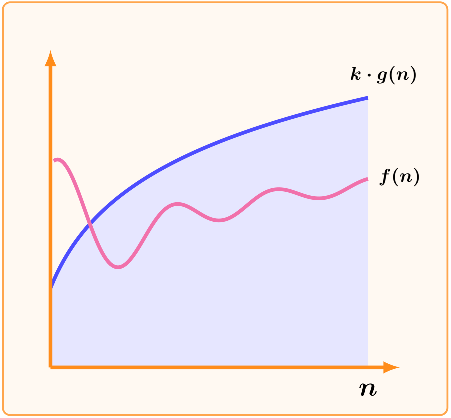
- To solve the **Towers of Hanoi** problem for $n = 64$,
- Be able to solve problems recursively using the **Divide-and-Conquer** technique,
- Be able to solve recurrence relations **inductively**
- Be able to solve common recurrence relations using **The Master Theorem**,

Asymptotic Notation

- **Big Oh Notation (upper bounds)**
  - Let $f(n)$ and $g(n)$ be any real-valued function. We say that g **eventually dominates** f if there is some constant $k > 0$ such that

$$f(n) \leq k \cdot g(n) \qquad \text{for all `large' } n$$

$$O(g(n)) \;=\; \left\{ \begin{array}{c} \text{All functions } f(n) \\ \text{that are \textit{eventually}} \\ \text{\textit{dominated} by } g(n) \end{array} \right\}$$

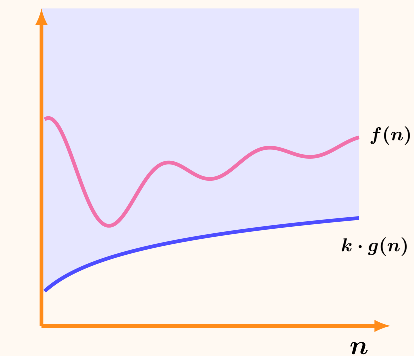


Asymptotic Notation – Big Omega

- **Big Omega Notation (lower bounds)**
  - Let $f(n)$ and $g(n)$ be any real-valued function. We say that g **eventually dominates** f if there is some constant $k > 0$ such that

$$f(n) \leq k \cdot g(n) \qquad \text{for all `large' } n$$

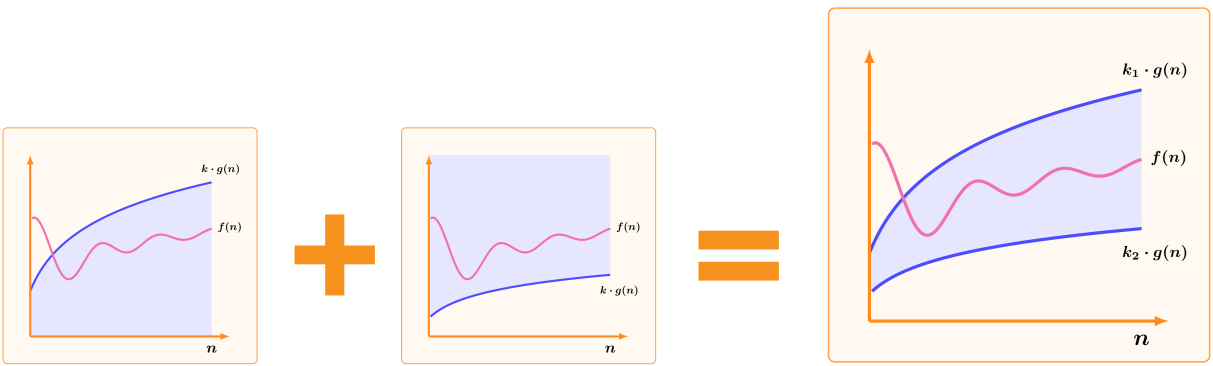
$$\Omega(g(n)) = \left\{ \begin{array}{c} \text{All functions } f(n) \\ \text{that } \textit{eventually} \\ \textit{dominate } g(n) \end{array} \right\}$$



Asymptotic Notation – Big Theta

- **Big Theta Notation (exact bounds)**
  - A function $f(n)$ belongs to $\Theta(g(n))$ if it is eventually **bounded above** *and* **below** by contant multiples of $g(n)$.

$$\boxed{\Theta(g(n)) \;=\; O(g(n)) \;\cap\; \Omega(g(n))}$$

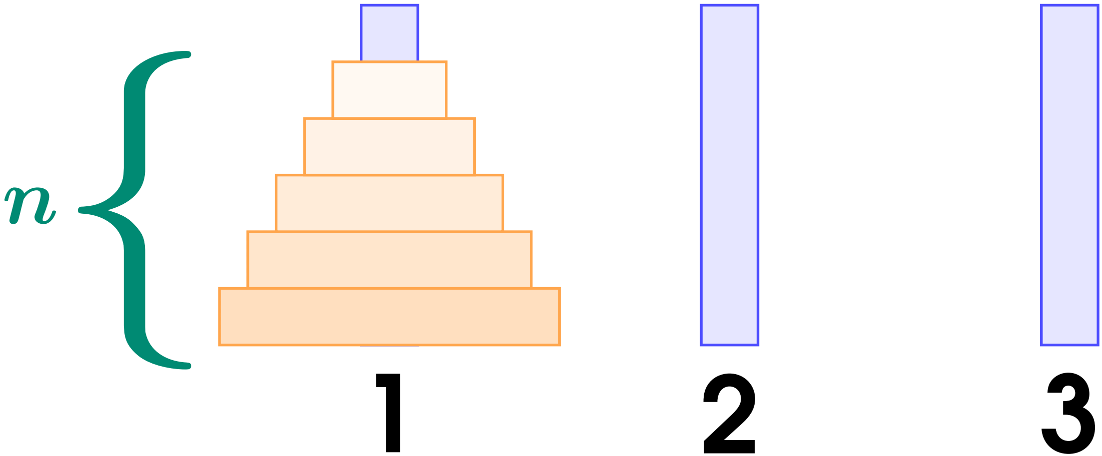


Solving Problems Recursively

The Towers of Hanoi

We would like an **general algorithm** that solves the Hanoi Tower problem for **any number of blocks**:

$$\boxed{\textsc{Move-Tower}\ (n,\ \text{post 1},\ \text{post 3})}$$



Divide-and-Conquer

- The Divide-and-Conquer technique is a useful technique for **designing** and **understanding** algorithms by diving them into easier sub-problems.

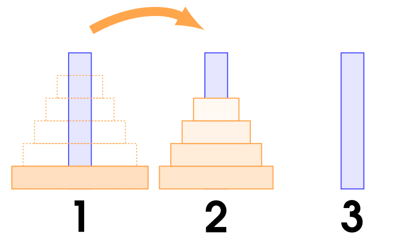
> **Divide-and-Conquer Technique**
>
> **Divide)** Divide the problem into several 'self-similar' but smaller sub-problems.
>
> **Conquer)** Solve these sub-problems recursively
>
> **Combine)** Recombine the sub-problems into a solution for the whole problem

Divide-and-Conquer

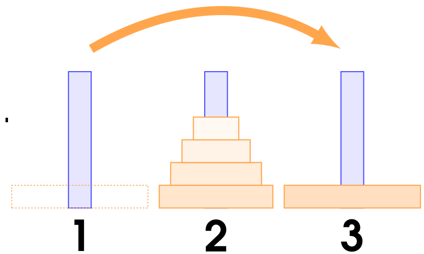
MOVE-TOWER $(n,\ i,\ j)$:

Step 1) MOVE-TOWER $(n-1,\ i,\ k)$

(move the top part of the tower out of the way)

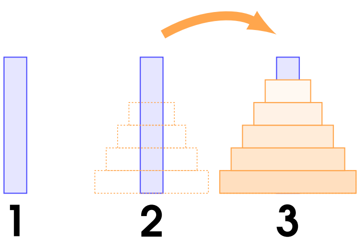


Step 2) Move the base of the tower from i to j.



Step 3) MOVE-TOWER $(n-1,\ k,\ j)$

(replace the top part of the tower)



Divide-and-Conquer

- How long does our MOVE-TOWER algorithm take?

$$\boxed{T(n) \;=\; \text{time to solve MOVE-TOWER } (n,\ i,\ j)}$$

(for any posts i and j)

- We can find a recurrence relation for $T(n)$ by examining the structure of the algorithm:

$$\begin{aligned} T(1) &= 1 \\ T(n) &= \underbrace{T(n-1)}_{\textbf{Step } 1} + \underbrace{1}_{\textbf{Step } 2} + \underbrace{T(n-1)}_{\textbf{Step } 3} \\ &= 2\,T(n-1) + 1 \end{aligned}$$

(to get the next value of $T(n)$ we multiply by 2 and add 1)

Divide-and-Conquer

- We can start to get an idea about the running time by **iterating** the first few values of $T(n)$

n	$T(n)$	
1	1	$\times 2 + 1$
2	3	$\times 2 + 1$
3	7	$\times 2 + 1$
4	15	$\times 2 + 1$
5	31	
$\vdots$	$\vdots$	
n	$2^n - 1$	

Divide-and-Conquer

- This gives us an **exact formula** for the running time...

$$\boxed{T(n) \;=\; 2^n - 1}$$

- ...but part that important for **scalability** is the 2^n,

$$\boxed{T(n) \;\in\; \Theta(2^n)}$$

(since $2^n - 1 < 2(2^n)$ for sufficiently large values of n)

- The running time for this algorithm quickly becomes **infeasible** to run!

(it takes 'exponential time' to solve)

How long until the end of the world?

- If we have **64 golden disks** to move,

$$\boxed{n = 64}$$

- The number of moves required to complete the puzzle is, therefore:

$$\begin{aligned} T(64) &= 2^{64} - 1 \\ &= 18,446,744,073,709,551,615 \end{aligned}$$

(or 1000 moves every second for 5 billion years...)

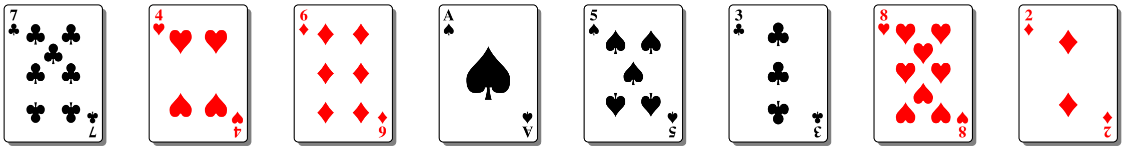
The world is safe for now!

Sorting Arrays with Divide-and-Conquer

Sorting Algorithms

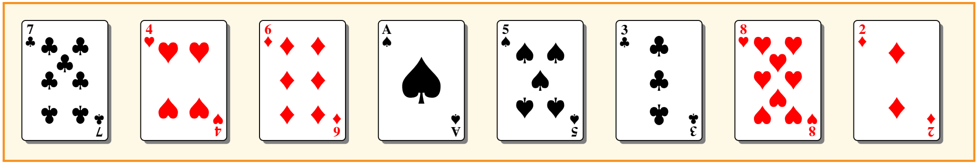
- Suppose we have an **array of integers (or cards)** of length n that we want to sort **ascending order**

(we are ignoring the suit)



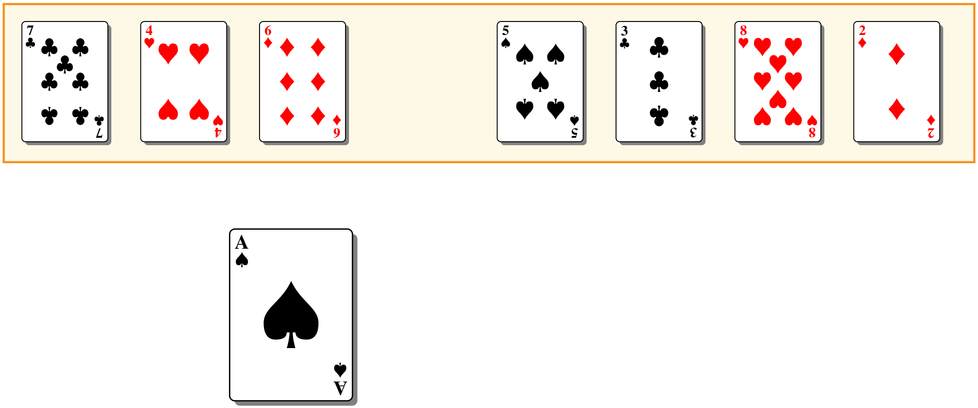
Naïve Sorting

- A **naïve sorting algorithm** would be to locate each card **in order** placing then into a new pile,



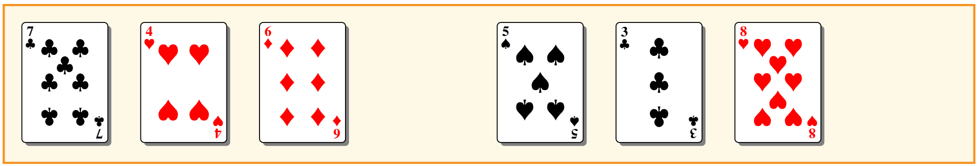
Naïve Sorting

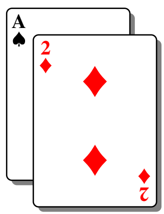
- A **naïve sorting algorithm** would be to locate each card **in order** placing then into a new pile,



Naïve Sorting

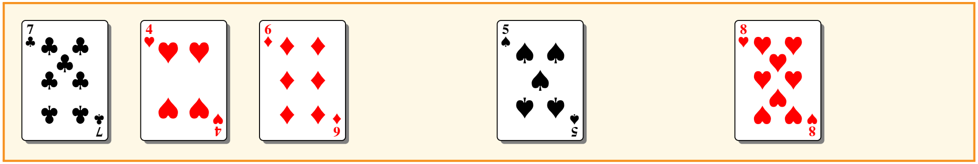
- A **naïve sorting algorithm** would be to locate each card **in order** placing then into a new pile,

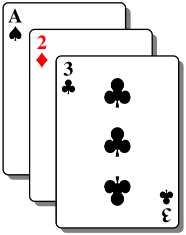




Naïve Sorting

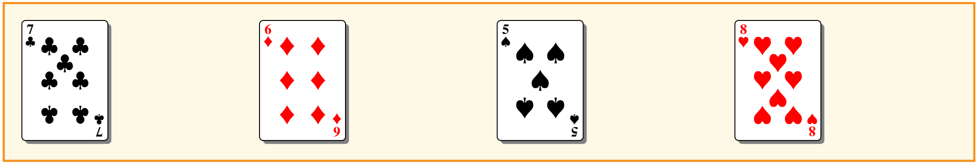
- A **naïve sorting algorithm** would be to locate each card **in order** placing then into a new pile,

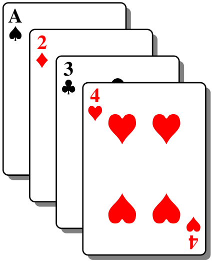




Naïve Sorting

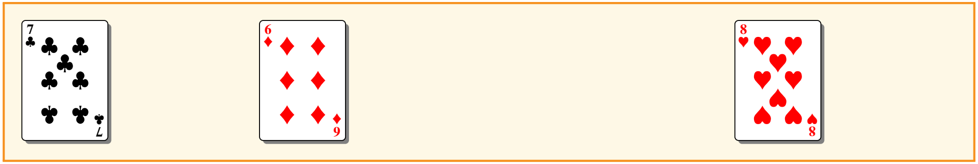
- A **naïve sorting algorithm** would be to locate each card **in order** placing then into a new pile,

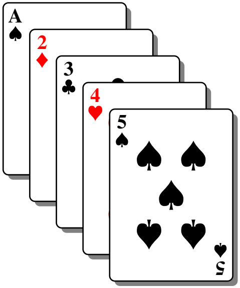




Naïve Sorting

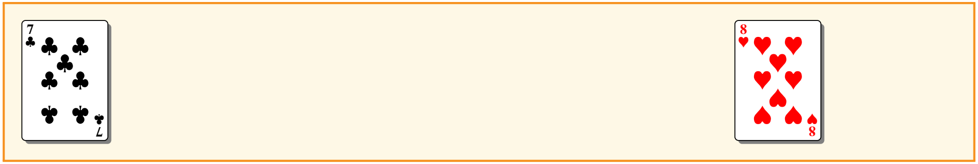
- A **naïve sorting algorithm** would be to locate each card **in order** placing then into a new pile,

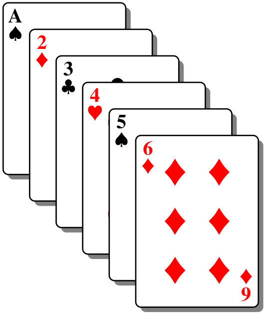




Naïve Sorting

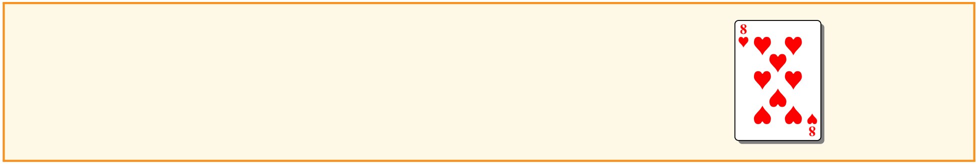
- A **naïve sorting algorithm** would be to locate each card **in order** placing then into a new pile,

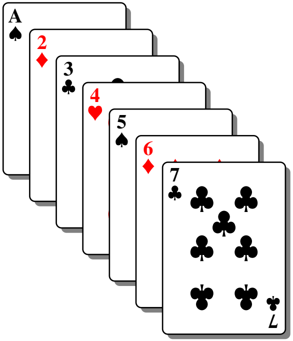




Naïve Sorting

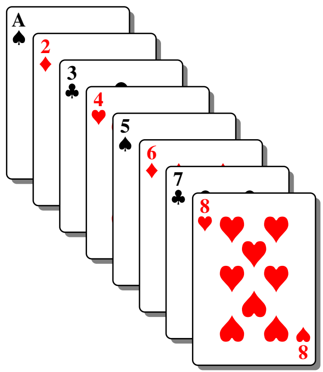
- A **naïve sorting algorithm** would be to locate each card **in order** placing then into a new pile,





Naïve Sorting

- A **naïve sorting algorithm** would be to locate each card **in order** placing then into a new pile,



Naïve Sorting

- What is the **(worst case) running time** for the naïve sorting algorithm?

Step 1) Locating the first card may take at most n steps,

Step 2) Adding to the new pile takes 1 step,

(depending on the data structure)

Step 3) Locating the next card takes at most $(n - 1)$ steps plus 1 to add to the new pile,

(there is one fewer card to search through)

Step 4) *etc.*

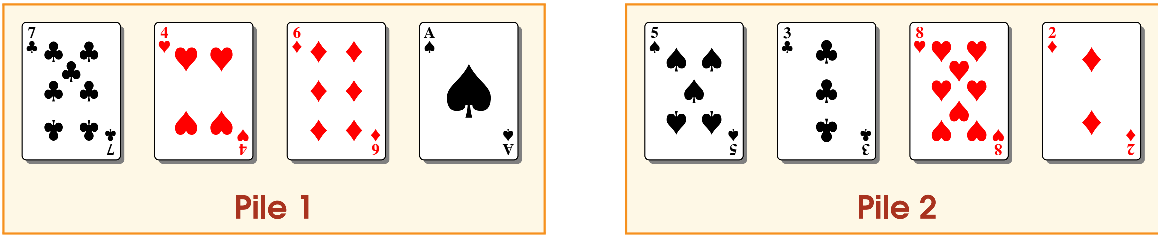
$$\begin{aligned} T(n) &\approx n + (n-1) + \cdots + 3 + 2 + 1 \\ &= \tfrac{1}{2}(n^2 + n) = \Theta(n^2) \end{aligned}$$

The Merge Sort Algorithm

MERGE-SORT($X[1:n]$):

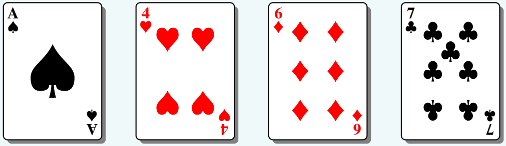
Step 1) Divide into two (roughly) even piles

(cannot divide perfectly if n is odd!)

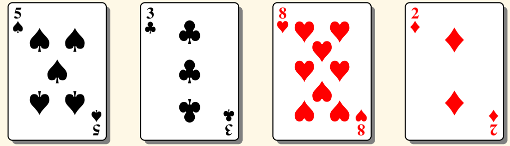


The Merge Sort Algorithm

Step 2) $\textsc{Merge-Sort}(\boldsymbol{X}[1 : \frac{1}{2}\boldsymbol{n}])$ (sort the first pile recursively)

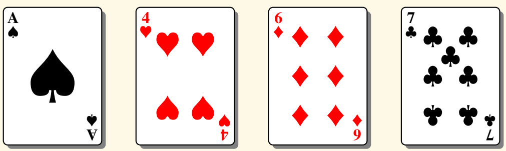


Sorted ✓

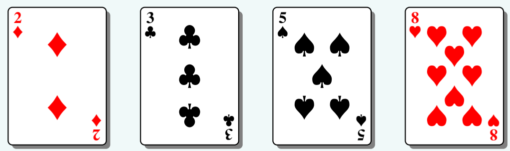


Pile 2

Step 3) $\textsc{Merge-Sort}(\boldsymbol{X}[\frac{1}{2}\boldsymbol{n} : \boldsymbol{n}])$ (sort the second pile recursively)



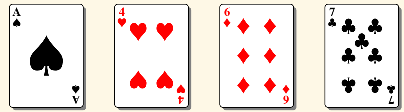
Sorted ✓



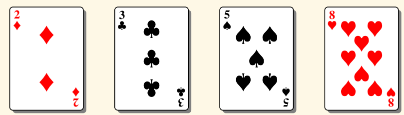
Sorted ✓

The Merge Sort Algorithm

Step 4) Merge the two sorted piles,



Pile 1

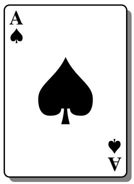


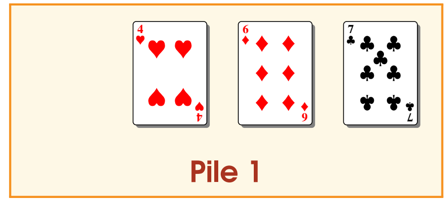
Pile 2

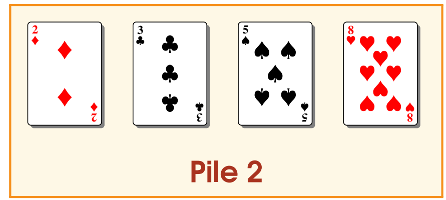
(we need only compare the first cards in each pile)

The Merge Sort Algorithm

Step 4) Merge the two sorted piles,



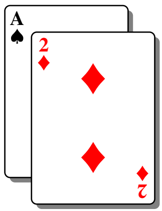


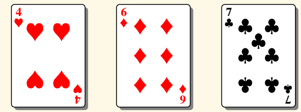


(we need only compare the first cards in each pile)

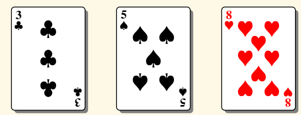
The Merge Sort Algorithm

Step 4) Merge the two sorted piles,





Pile 1

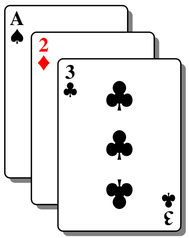


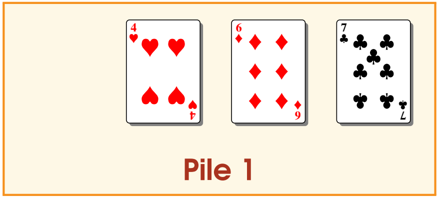
Pile 2

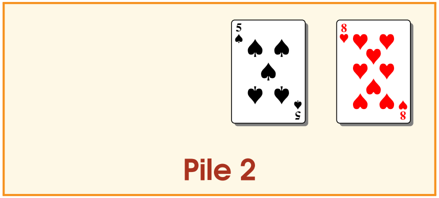
(we need only compare the first cards in each pile)

The Merge Sort Algorithm

Step 4) Merge the two sorted piles,



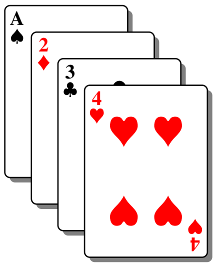


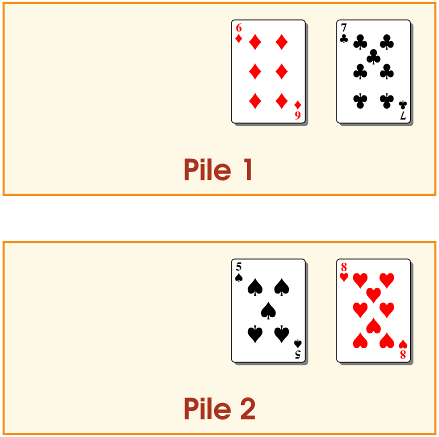


(we need only compare the first cards in each pile)

The Merge Sort Algorithm

Step 4) Merge the two sorted piles,

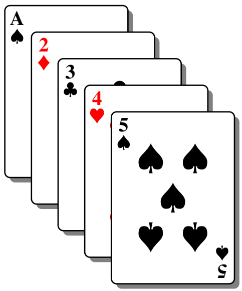


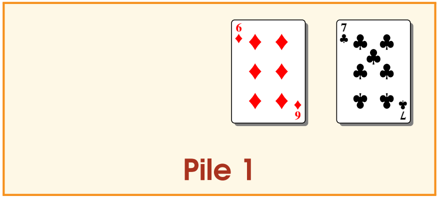


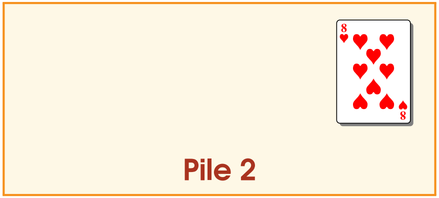
(we need only compare the first cards in each pile)

The Merge Sort Algorithm

Step 4) Merge the two sorted piles,



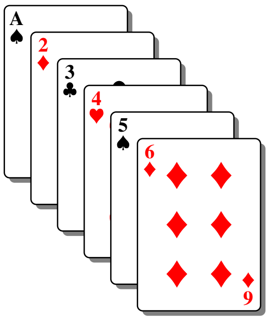


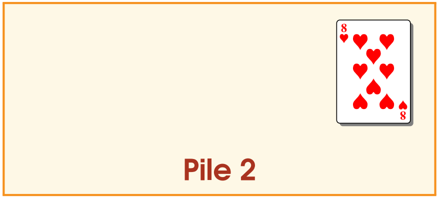


(we need only compare the first cards in each pile)

The Merge Sort Algorithm

Step 4) Merge the two sorted piles,

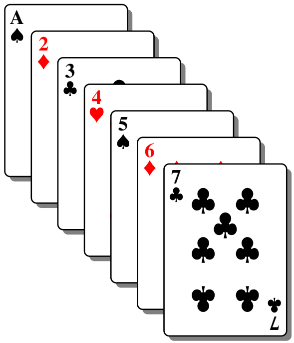


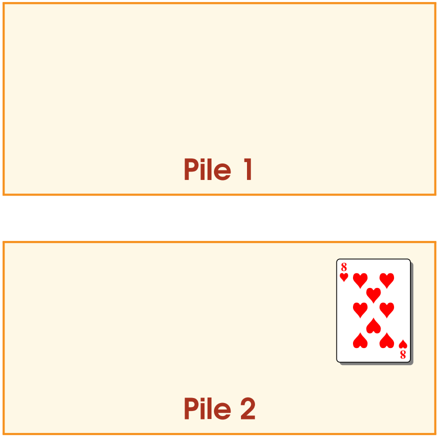


(we need only compare the first cards in each pile)

The Merge Sort Algorithm

Step 4) Merge the two sorted piles,

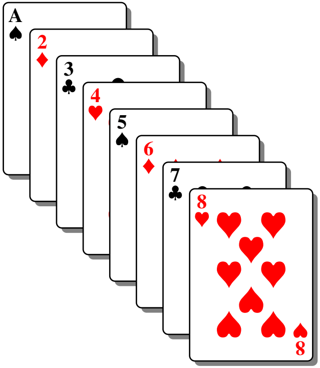


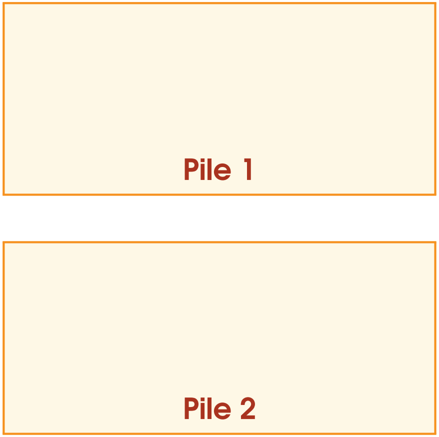


(we need only compare the first cards in each pile)

The Merge Sort Algorithm

Step 4) Merge the two sorted piles,





(we need only compare the first cards in each pile)

The Merge Sort Algorithm

- Again, we can ask: how long does our algorithm take?

$$\boxed{\boldsymbol{T(n)} \;=\; \text{time to solve } \textsc{Merge-Sort}(\boldsymbol{X})}$$

(for any array X of length n)

The Merge Sort Algorithm

- Again, we can ask: how long does our algorithm take?

$$\boldsymbol{T}(\boldsymbol{n}) \;=\; \text{time to solve } \textsc{Merge-Sort}(\boldsymbol{X})$$

(for any array X of length n)

- We can find a **recurrence relation** for $\boldsymbol{T}(\boldsymbol{n})$ by examining the structure of the algorithm:

$$\boldsymbol{T}(1) \;=\; 1$$

The Merge Sort Algorithm

- Again, we can ask: how long does our algorithm take?

$$\boxed{T(n) \;=\; \text{time to solve } \textsc{Merge-Sort}(X)}$$

(for any array X of length n)

- We can find a **recurrence relation** for $T(n)$ by examining the structure of the algorithm:

$$\begin{aligned} T(1) &= 1 \\ T(n) &\approx \underbrace{n}_{\text{Step 1}} + \underbrace{T(\lceil n/2 \rceil)}_{\text{Step 2}} + \underbrace{T(\lceil n/2 \rceil)}_{\text{Step 3}} + \underbrace{n}_{\text{Step 4}} \end{aligned}$$

The Merge Sort Algorithm

- Again, we can ask: how long does our algorithm take?

$$\boxed{T(n) \;=\; \text{time to solve } \textsc{Merge-Sort}(X)}$$

(for any array X of length n)

- We can find a **recurrence relation** for $T(n)$ by examining the structure of the algorithm:

$$\begin{aligned} T(1) &= 1 \\ T(n) &\approx \underbrace{n}_{\textbf{Step } 1} + \underbrace{T(\lceil n/2 \rceil)}_{\textbf{Step } 2} + \underbrace{T(\lceil n/2 \rceil)}_{\textbf{Step } 3} + \underbrace{n}_{\textbf{Step } 4} \\ &= 2\,T(\lceil n/2 \rceil) + 2n \end{aligned}$$

The Merge Sort Algorithm

- Again, we can ask: how long does our algorithm take?

$$\boldsymbol{T}(\boldsymbol{n}) \;=\; \text{time to solve } \textsc{Merge-Sort}(\boldsymbol{X})$$

(for any array X of length n)

- We can find a **recurrence relation** for $\boldsymbol{T}(\boldsymbol{n})$ by examining the structure of the algorithm:

$$\begin{aligned}
\boldsymbol{T}(1) &= 1 \\
\boldsymbol{T}(\boldsymbol{n}) &\approx \underbrace{\boldsymbol{n}}_{\textbf{Step } 1} + \underbrace{\boldsymbol{T}(\lceil \boldsymbol{n}/2 \rceil)}_{\textbf{Step } 2} + \underbrace{\boldsymbol{T}(\lceil \boldsymbol{n}/2 \rceil)}_{\textbf{Step } 3} + \underbrace{\boldsymbol{n}}_{\textbf{Step } 4} \\
&= 2\,\boldsymbol{T}(\lceil \boldsymbol{n}/2 \rceil) + 2\boldsymbol{n}
\end{aligned}$$

(where $\lceil x \rceil$ is the *ceiling function* of x)

The Merge Sort Algorithm

- Again, we can **iterate** the first few values of $T(n)$

(easier if we look only at powers of two!)

n	$T(n)$	
1	1	
		$\times 2 + 4$
2	6	
		$\times 2 + 8$
4	20	
		$\times 2 + 16$
8	56	
		$\times 2 + 32$
16	144	
		$\times 2 + 64$
32	352	
$\vdots$	$\vdots$	

(it is not quite so easy to see what the growth-rate is...)

Solving Recurrence Relations

The Substitution Method

Proof by Induction

Base Case) Show that your solution holds for $n = 1$,

Inductive Case) **(i)** Assume your result holds for $n = k$,

(ii) Substitute to confirm that it also holds for $n = (k + 1)$.

- Like knocking over an **infinite stack of dominoes**:
  - The *base case* knocks over the **first domino**,
  - The *inductive case* shows that the dominoes are **spaced close enough** that the kth domino always knocks down the $(k + 1)$st domino!

(therefore ALL dominoes will fall!)

The Substitution Method

- **Example:** Show that the solution to the recurrence relation

$$\boxed{\begin{aligned} T(1) &= 1 \\ T(n) &= 2\,T(n-1) + 1 \end{aligned}}$$

is given by $T(n) = 2^n - 1$.

Base Case) We just need to check that our formula gives the correct value for $n = 1$.

$$\boxed{T(1) = 1 = 2^1 - 1}$$

The Substitution Method

Induction Case) Assume that

$$\boxed{T(k) = 2^k - 1 \qquad \text{for some } k \geq 1}$$

(this is known as the *'Induction Hypothesis'*)

We can substitute into the recurrence relation to find $T(k+1)$

$$\begin{aligned} T(k+1) &= 2 \cdot T(k) + 1 \\ &= 2 \cdot \boxed{(2^k - 1)} + 1 \\ &= 2 \cdot 2^k - 2 + 1 \\ &= 2^{k+1} - 1 \end{aligned}$$

Conclusion) Since this has the same form as the Induction Hypothesis, the formula must hold for **ALL** values of n. **Q.E.D**

The Substitution Method

- How about a more complicated recurrence relation such as

$$\boxed{\begin{aligned} T(1) &= 1 \\ T(n) &= 2\,T(\lceil n/2 \rceil) + 2n \end{aligned}}$$

(this was the approximate running time for the merge-sort algorithm)

- Since $T(n)$ depends on $T(n/2)$ rather than the immediate predecessor $T(n-1)$, we need a **slightly stronger** version of induction!
  - It is not enough to consider the spacing between **neighbouring dominos**,

The Substitution Method

Proof by Induction (Strong)

Base Case) Show that your solution holds for $n = 1$,

Inductive Case) **(i)** Assume your result holds for *all* $m \leq k$ for some k,

(ii) Substitute to confirm that it also holds for $n = (k + 1)$.

- We may rely not just on the previous 'domino' but on all those that have fallen before!
- When is this useful?
  - If your recurrence relation does not depend on the **previous value**,
  - Or if your recurrence relation involves **multiple calls** to itself, *e.g.*

$$\boxed{F(n) \;=\; F(n-1) + F(n-2)}$$

(this recurrence relation generates the Fibonacci numbers)

The Substitution Method

- **Example:** Show that the solution to the recurrence relation

$$\begin{aligned} T(1) &= 1 \\ T(n) &= 2\,T(\lceil n/2 \rceil) + 2n \end{aligned}$$

bounded above by $T(n) \geq n \log_2 n$, for $n \geq 1$

Base Case) Again, we just need to check that our formula gives the correct value for $n = 1$.

$$\boxed{T(1) \;=\; 1 \;\geq\; 0 \;=\; 1 \log_2 1}$$

The Substitution Method

Induction Case) Assume that

$$\boxed{T(m) \geq m \log_2 m \qquad \text{for all } m \leq k \text{ for some } k \geq 1}$$

We can substitute into the recurrence relation to find $T(k+1)$

$$\begin{aligned}
T(k+1) &= 2 \cdot T\left(\left\lceil \frac{k+1}{2} \right\rceil\right) + 2(k+1) \\
&\geq 2 \cdot T\left(\frac{k+1}{2}\right) + 2(k+1) \\
&\geq 2 \cdot \boxed{\left(\frac{k+1}{2}\right) \log_2 \frac{k+1}{2}} + 2(k+1) \\
&= (k+1)\Big[\log_2(k+1) - 1\Big] + 2(k+1) \\
&= (k+1)\log_2(k+1) + (k+1) \\
&\geq (k+1)\log_2(k+1)
\end{aligned}$$

The Substitution Method

Conclusion) Hence, it follows that

$$\boxed{T(n) \;\geq\; n \log_2 n}$$

for **ALL** values of $n \geq 1$.

Q.E.D

- Hence it follows that the Merge-sort algorithm runs belongs to the class

$$\boxed{T(n) \;=\; \Omega(n \log_2 n)}$$

(we can similarly, show that $T(n) = \Theta(n \log_2 n)$, as well)

The Substitution Method

- **Remarks on Proof by Induction:**
    - Often easier to establish **upper** and **lower** bounds than to prove an exact formula.
    - You need to **correctly 'guess'** the correct formula before you start!
        - If the algorithm is similar to one whose growth-rate is known, try that!
        - If your first guess does not work, adjust accordingly!

          (if you can't bound above by a quadratic, try a cubic, *etc*..)
        - If the first few values **'misbehave'**, use a bigger base case!

The Master Theorem

The Master Theorem

Let $T(n)$ be a *monotonically increasing* recurrence relation such that

$$\boxed{T(n) \;=\; a\,T\left(\frac{n}{b}\right) + f(n)}$$

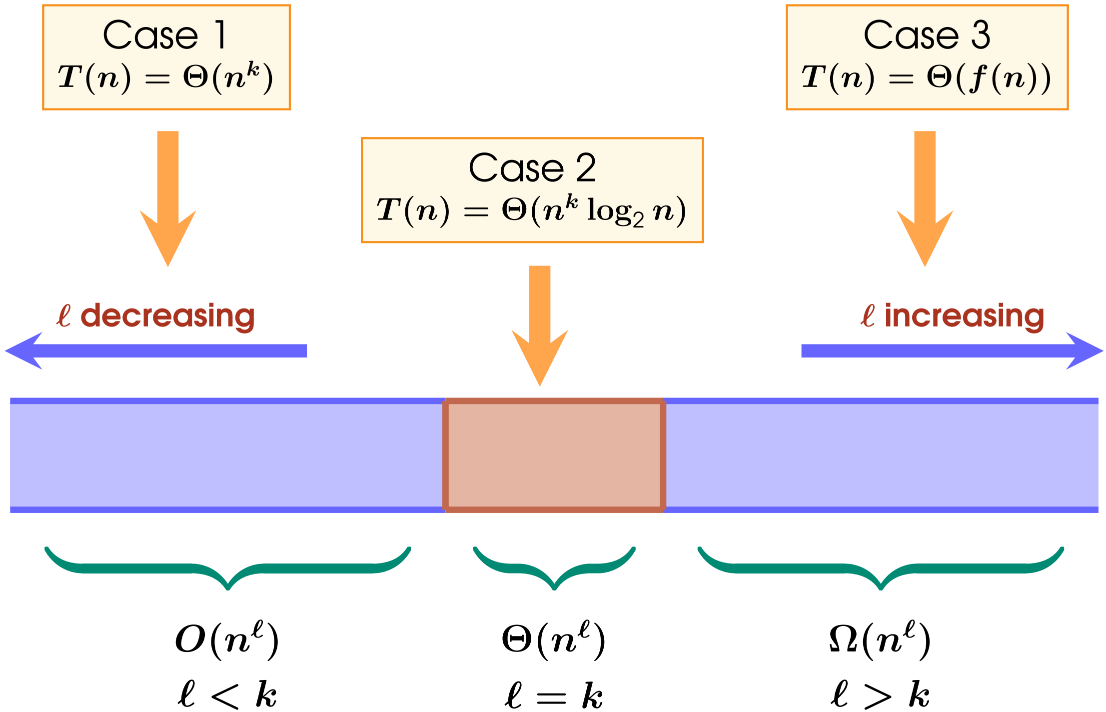
(for some constants $a \geq 1$, $b \geq 2$.)

Then

$$\boxed{T(n) \;\in\; \begin{cases} \Theta(n^k) & \text{if } f(n) \in O(n^\ell) \text{ for } \ell < k \\ \Theta(n^k \log_2 n) & \text{if } f(n) \in \Theta(n^\ell) \text{ for } \ell = k \\ \Theta(f(n)) & \text{if } f(n) \in \Omega(n^\ell) \text{ for } \ell > k \end{cases}}$$

where $\boxed{k = \log_b a \;=\; \log_{10} a / \log_{10} b}$

The Master Theorem



The Master Theorem : Some Examples

- **Example 1:** Let $T(n)$ be given by the following recurrence relation

$$\boxed{T(n) \;=\; 4\,T\left(\frac{n}{2}\right) + 2^n}$$

Step 1) Identify the parameters:

$$\boxed{a = 4, \quad b = 2, \quad \text{therefore} \quad k = \log_2 4 = 2}$$

Step 2) Identify the growth rate of $f(n)$

$$\boxed{f(n) \;=\; 2^n \;\in\; \Omega(n^3)}$$

(bounded below by a cubic, and $k < 3$)

Step 3) Therefore, Case 3 applies, and we have:

$$\boxed{T(n) \;\in\; \Theta(f(n)) \;=\; \Theta(2^n)}$$

The Master Theorem : Some Examples

- **Example 2:** Let $T(n)$ be given by the following recurrence relation

$$T(n) \;=\; 2\,T\left(\frac{n}{4}\right) + \sqrt{n+1}$$

Step 1) Identify the parameters:

$$a = 2, \quad b = 4, \quad \text{therefore} \quad k = \log_4 2 = 0.5$$

Step 2) Identify the growth rate of $f(n)$

$$f(n) \;=\; \sqrt{n+1} \;\in\; \Theta(\sqrt{n})$$

(bounded above and below by a square root, and $k = 0.5$)

Step 3) Therefore, Case 2 applies, and we have:

$$T(n) \;\in\; \Theta(n^k \log_2 n) \;=\; \Theta(\sqrt{n}\,\log_2 n)$$

The Master Theorem : Some Examples

- **Example 3:** Let $T(n)$ be given by the following recurrence relation

$$\boxed{T(n) \;=\; 3\,T\left(\frac{n}{2}\right) + \log_2 n}$$

Step 1) Identify the parameters:

$$\boxed{a = 3, \quad b = 2, \quad \text{therefore} \quad k = \log_2 3 \approx 1.5849}$$

Step 2) Identify the growth rate of $f(n)$

$$\boxed{f(n) \;=\; \log_2 n \;\in\; \Theta(n)}$$

(bounded above by a linear function, and $k > 1$)

Step 3) Therefore, Case 1 applies, and we have:

$$\boxed{T(n) \;\in\; \Theta(n^k) \;=\; \Theta(n^{1.5849})}$$

The Master Theorem : Some Examples

WARNING! WARNING!

- There is a **mistake** is the previous slide!
- Every approximation, no matter how accurate, will **eventually diverge!**

$$\boxed{\Theta(n^{1.58}) \neq \Theta(n^{1.584}) \neq \Theta(n^{1.5849}) \neq \cdots \neq \Theta(n^{\log_2 3})}$$

(we cannot use approximations when writing grown rates)

- The **correct** growth-rate for $T(n)$ should be

$$\boxed{T(n) \;\in\; \Theta(n^{\log_2 3})}$$

Next Time...

- SAT Solving
  - Greedy Algorithms,
  - The DPLL Algorithm
- Solving simple instances of SAT.

End of Slides!