Analyzing algorithms

Yihan Sun

Course announcement

- **Entrance Exam due this weekend**
  - Try to start working on it soon if you haven't
  - Come to the office hours if you need help
- **Course policy test: due next Tuesday**
  - 1 point to your final grade, and required
  - Resubmit-able multiple choices problems
- **Regarding course-related logistics**
  - Contact Zijin for non-homework-related questions
  - Contact Xiangyun for homework-related questions

Collaboration: the “whiteboard” policy

- You are welcome to chat with each other (also welcome to come to OHs), but you come with nothing and leave with nothing
- When you type your answers / code, it must be done on your own. It must be close-book. It must be typed by you, word by word.
- Any violation may result in severe outcome. Usually -100% current/all homework assignment score, fail the course, report to the university, the university may make further decisions
- Must cite if any idea is from other sources, including people, books, websites, AI, etc.

Analyzing algorithms

Yihan Sun

Analyzing algorithms

- **Predict how your algorithm performs in practice**

- **Criteria:**
  - Running time ← Time complexity
  - Space usage
  - Cache I/O
  - Disk I/O
  - Network bandwidth
  - Power consumption
  - Lines of codes

What are good algorithms?

- **Tale about Guass**

One day Gauss's teacher asked his class to add together all the numbers from 1 to 100, assuming that this task would occupy them for quite a while. He was shocked when young Gauss, after a few seconds thought, wrote down the answer 5050. (source: https://nrich.maths.org/2478)

Other students:

```
sum = 0;
for (int i = 1; i <=n; i++)
  sum += i;
```

$O(n)$ time complexity

Guass:

```
sum = (1+n)*n/2;
```

$O(1)$ time complexity

$$1 + 2 + 3 + 4 + 5 + 6 + 7 + 8 + 9 + 10 = ?$$

$$(1+10) + (2+9) + (3+8) + (4+7) + (5+6) = ?$$

Computational Model

What is time complexity?

```
s = 0
for (i = 1 to n) {
  s = s + A[i]
}
```

```
sum(A, n) {
  if (n == 1) return A[0];
  L = sum(A, n/2);
  R = sum(A + n/2, n-n/2);
  return L+R;
}
```

$O(n)$ time complexity

$O(n)$ time complexity

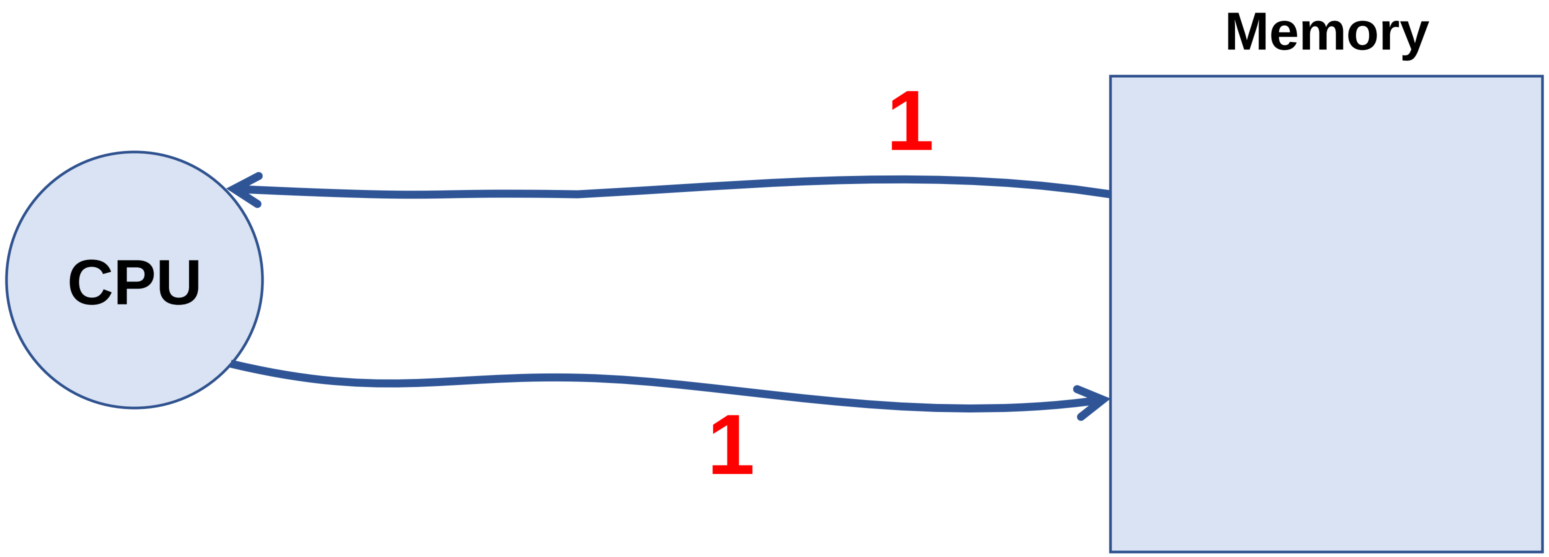
- **Why the time complexity is O(n)? What are we counting?**

What is “time complexity”?

- **Count the number of “instructions” in the algorithm**
- **Random Access Machine (RAM) model**
  - We have an arbitrarily large memory
  - We can
    - do arithmetic calculation
    - Read/write to a memory location given the address
  - Every operation takes unit time

Random-Access Machine (RAM)

- **Unit cost for:**
  - Any instruction on w-bit words (how large do we need for w?)
  - Read/write a single memory location from an infinite memory
- **The cost measure: time complexity**

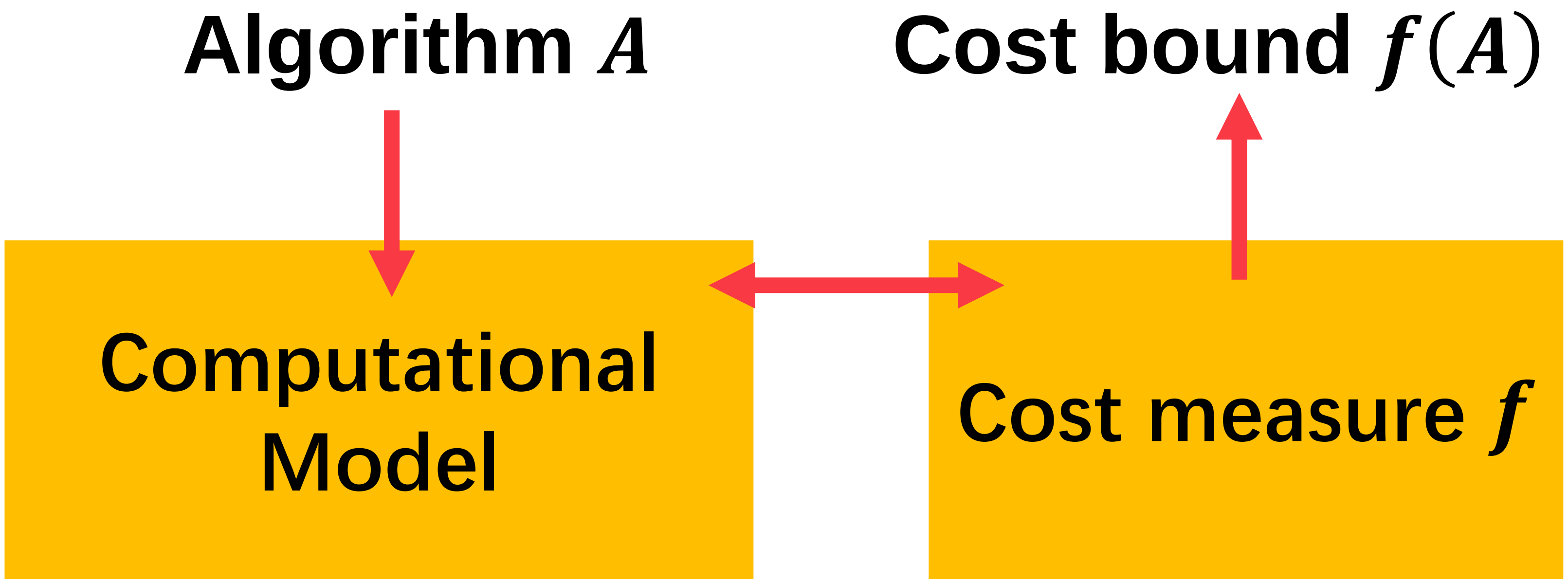


What is a computational model?

- What resource do we have?
- What operations are we allowed to do?
- What is the cost of each operation?

What is "time complexity"?

- To estimate the time needed for an algorithm
- To define the "cost" of an algorithm, we first need to define what "costs" time
- **Computational model**



What is a computational model?

- What resource do we have?
- What operations are we allowed to do?
- What is the cost of each operation?

What is time complexity?

```
s = 0
for (i = 1 to n) {
  s = s + A[i]
}
```

```
sum(A, n) {
  if (n == 1) return A[0];
  L = sum(A, n/2);
  R = sum(A + n/2, n-n/2);
  return L+R;
}
```

$O(n)$ time complexity

$O(n)$ time complexity

- **Why the time complexity is O(n)? What are we counting?**
  - The number of operations
- **Why the time complexity is O(n)? Why we omit the constants and lower-order terms? Why we use "asymptotic analysis"?**

Anything else we need?

- **Count the number of “instructions” in the algorithm**
- **Random Access Machine (RAM) model**
  - We have an arbitrarily large memory
  - We can
    - do arithmetic calculation
    - Read/write to a memory location given the address
  - Every operation takes unit time

```
sum = 0;
for (int i = 1; i <=n; i++)
  sum += i;
```

```
sum = (1+n)*n/2;
```

3n+2 operations?

3 operations?

What is “time complexity”?

- **Random Access Machine (RAM) model**
  - Every operation, including memory access, arithmetic operations, etc., takes unit time
- **To make our life easier, we only analyze order of the cost, and omit**
  - Lower-order terms
  - Constants
- **We care about how faster a function grows for large n**

Asymptotic Analysis

Asymptotic notation

```
sum = 0;
for (int i = 1; i <=n; i++)
  sum += i;
```

3n+2 operations?

$O(n)$ operations

```
sum = (1+n)*n/2;
```

3 operations?

$O(1)$ operations

- **OK, then what does big-O mean?**

Asymptotic notations

- **Big-O: asymptotically** **smaller than or equal to** $\leq$

  larger than or equal to $\geq$

  smaller than $<$

  larger than $>$

  equal to $=$

- n is $O(n)$
- $3n + 2$ is $O(n)$
- $\log n + \sqrt{n} + 4$ is also $O(n)$

- What happens if we want to say other cases?

Analogy to real numbers

Functions	Real numbers
$f(n) = O(g(n))$	$a \leq b$
$f(n) = \Omega(g(n))$	$a \geq b$
$f(n) = \Theta(g(n))$	$a = b$
$f(n) = o(g(n))$	$a < b$
$f(n) = \omega(g(n))$	$a > b$

Popular Classes of Functions

		Example
• **Constant:**	$f(n) = \Theta(1)$	$1, 10, 10000000$
• **Logarithmic:**	$f(n) = \Theta(\log(n))$	$\log n, \log n + 3\log\log n$
• **Poly-logarithmic:**	$f(n) = O(\log^k n)$	$\log^2 n, \log^9 n + 8$
• **Sublinear:**	$f(n) = o(n)$	$\log n, \sqrt{n}, n^{1/5}$
• **Linear:**	$f(n) = \Theta(n)$	$n, 5n + \log n$
• **Super-linear:**	$f(n) = \omega(n)$	$n^3, n\log n$
• **Quadratic:**	$f(n) = \Theta(n^2)$	$n^2, 3n^2 + n$
• **Polynomial:**	$f(n) = O(n^k)$	$n^3 + 2n^2 + 4, 4n^5$
• **Exponential:**	$f(n) = \Theta(k^n)$	2^n

($k \geq 0$ is a constant)

Compare two functions

- $\lim_{n\to\infty} \frac{f(n)}{g(n)}$
- 0: $f(n) = o\big(g(n)\big)$
- **Constant** $c > 0$: $f(n) = \Theta\big(g(n)\big)$
- ∞: $f(n) = \omega\big(g(n)\big)$

$O\big(g(n)\big)$

$\Omega\big(g(n)\big)$

Commonly-used functions

For large enough $\boldsymbol{n}$, we have (asymptotically):

$c > 0$ $c_1 > 1$ $0 < c_2 < 1$ $c_3 > 1$ $c_4 > 1$

- $c < \log\log n < \log n < \log^{c_1} n < n^{c_2} < n < n\log n < n^{c_3} < c_4^n < n^n$

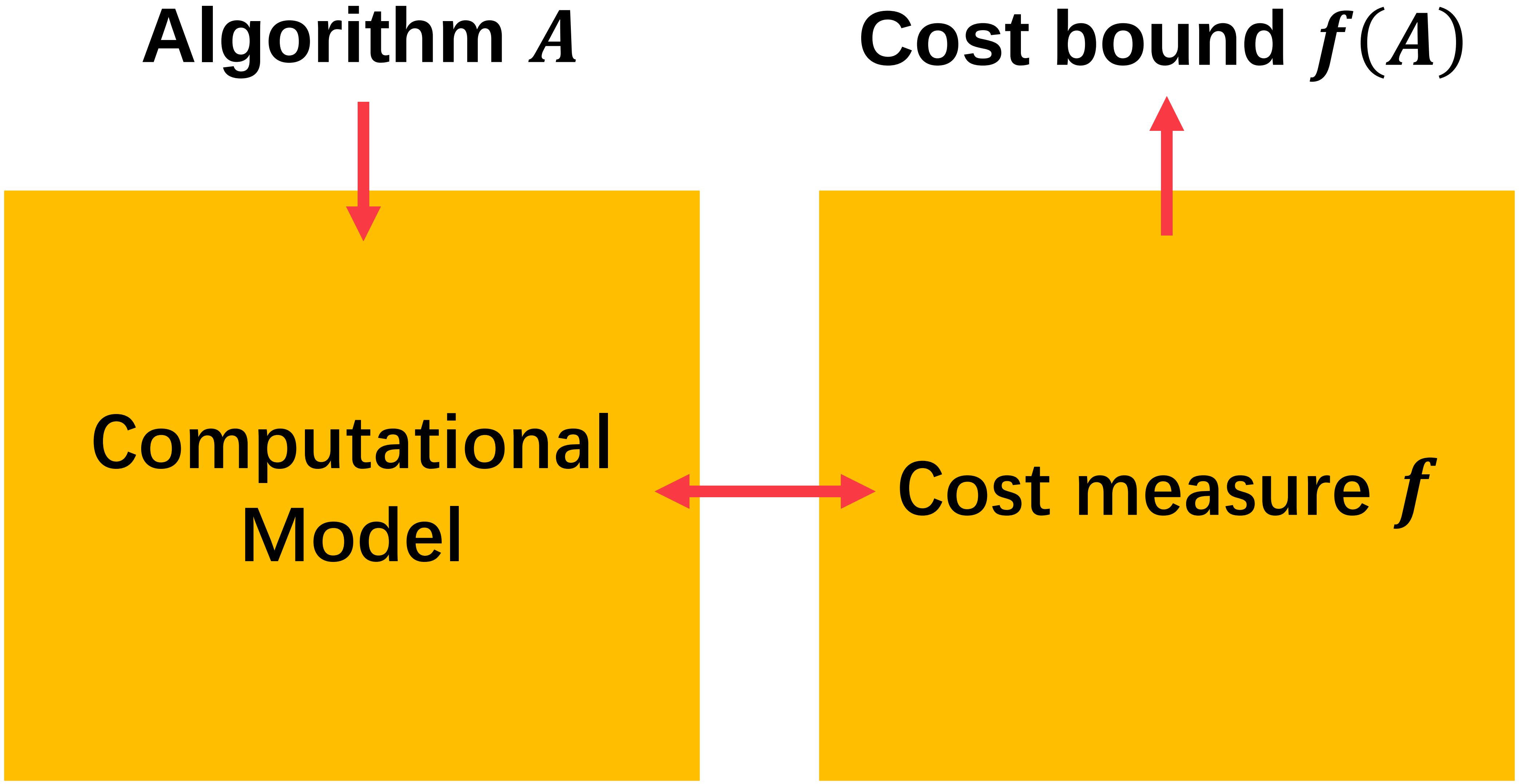
Analyze running time

- **Random Access Machine (RAM) model**
  - Every operation, including memory access, arithmetic operations, etc., takes unit time
- **We only care about the order of the cost for simplicity, and omit**
  - Constants
  - Lower-order terms
- **We usually consider worst-case running time for general input**
  - In some cases, we also analyze bounds with probabilistic guarantees (e.g., average running time for randomized algorithms)

If you are not familiar with these notations

- **Read CLRS Chapter 3: “Growth of Functions”**
  - Definitions of the asymptotic notation
  - How to compare the growth of two functions
  - What are the classic “classes” of functions

How does the asymptotic notation relate to the computational model (algorithm analysis)?


Algorithm A
Cost bound $f(A)$
Computational Model
Cost measure f

Questions (true or false):

- **The goal of defining computational models is to allow for asymptotic analysis**

- **We have to use asymptotic notations when analyzing algorithms**
  - It is almost true for time complexity, since it is too sloppy to give a distinct weight to each operation

Is RAM model perfect?

$n = 10^9$

```
for (int i = 0; i < n; i++) A[i] = A[i] + 1;
```
0.141072s

```
for (int i = 0; i < n; i++) A[i] = (long long)(i)*4323 % n + 1;
```
0.580809s

```
for (int i = 0; i < n; i++) A[i] = A[(long long)(i)*4323 % n] + 1;
```
3.25008s

How long would the other for-loop take?

A. 0.14s	D. 1.0s
B. 0.2s	E. 3.0s
C. 0.6s	F. 5.0s

Not all CPU operations are created equal

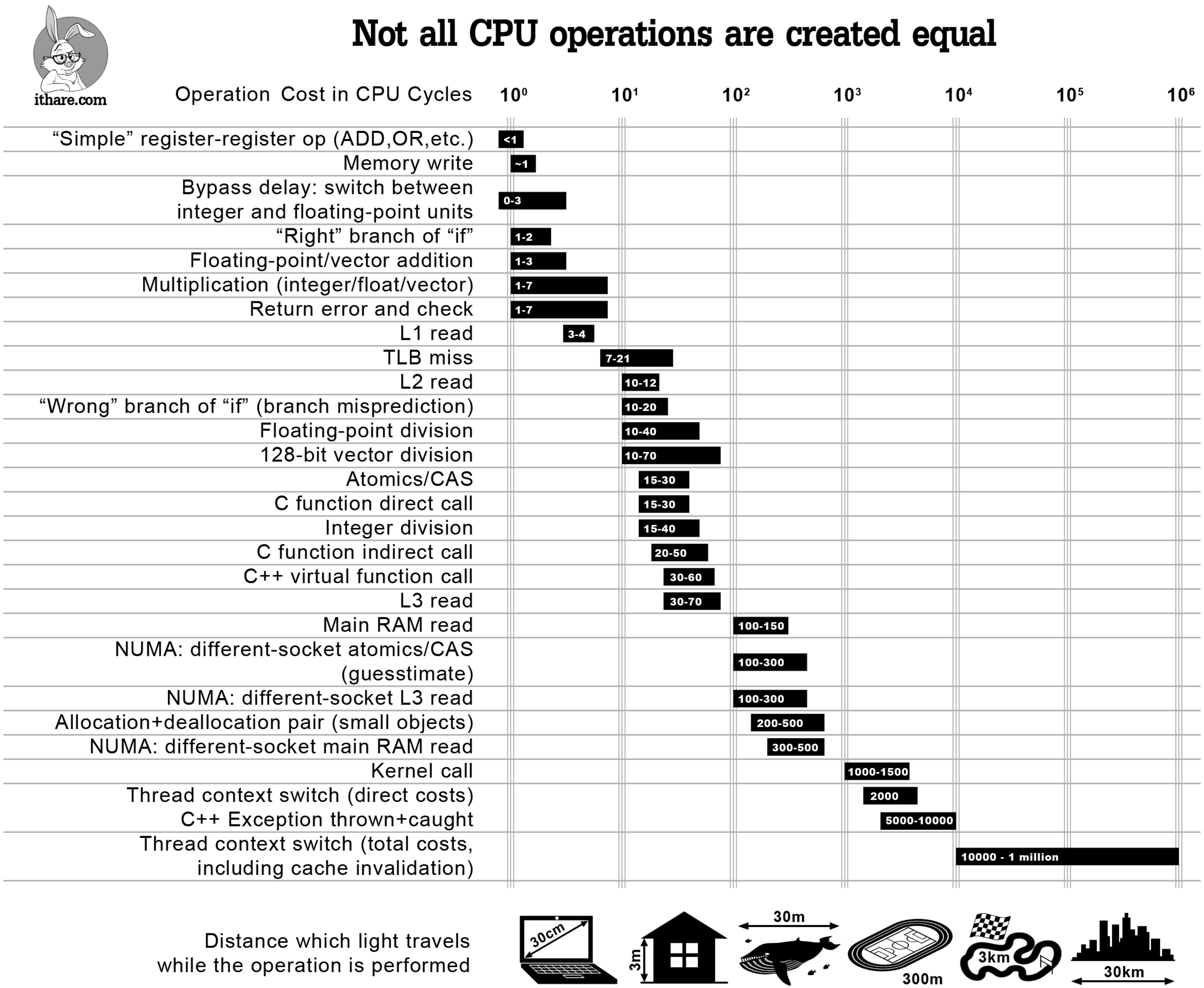
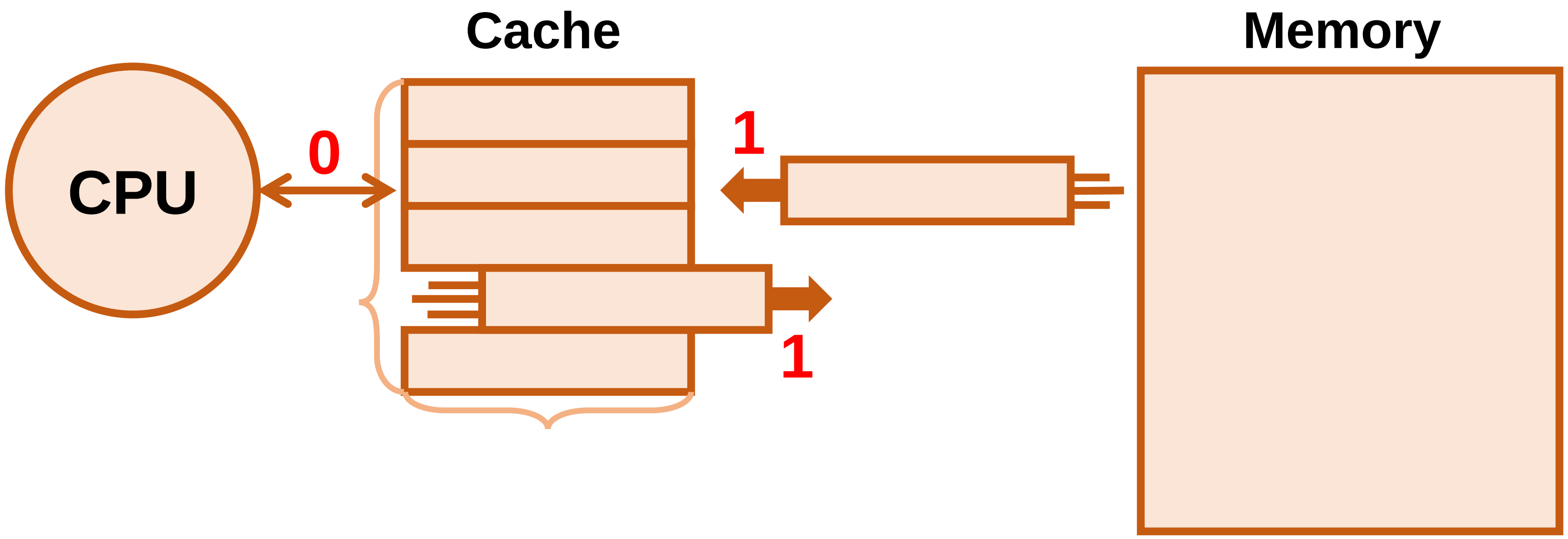


Image from ithare.com: http://ithare.com/infographics-operation-costs-in-cpu-clock-cycles/

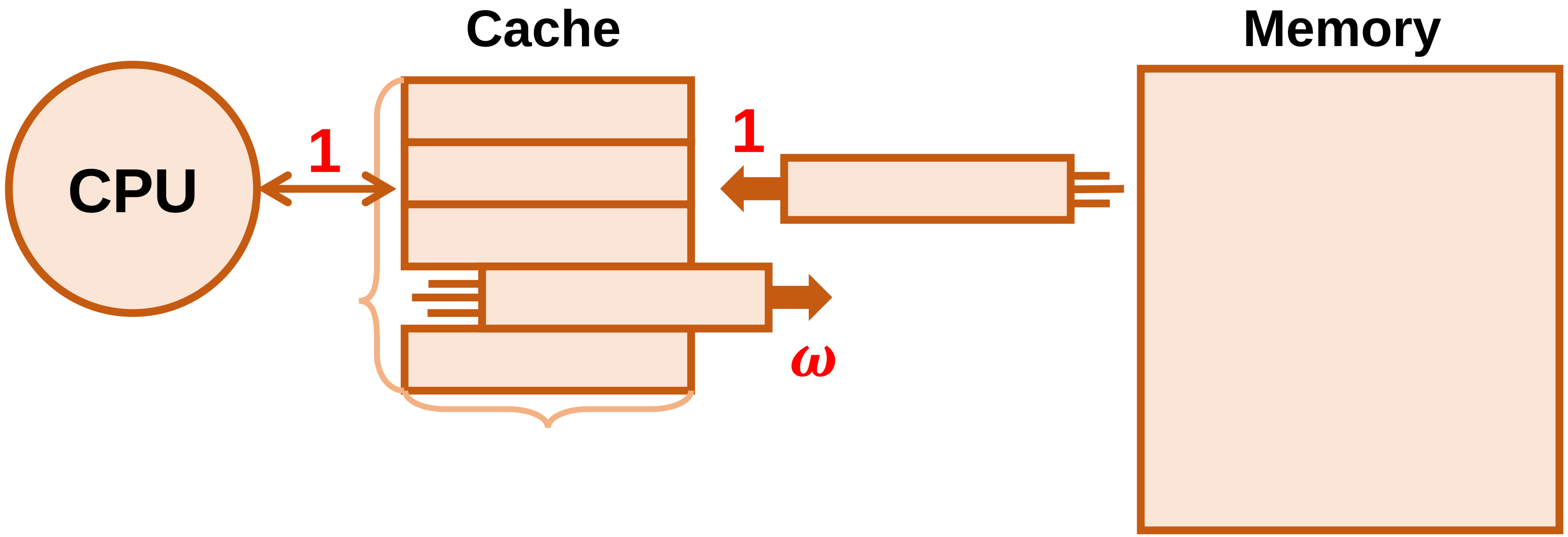
I/O model

- **Access memory is expensive**
- **But we have cache in our machine!**
- **We count the cost only if it is a "cache miss"**
  - Accessing data in the cache is free
  - # of transfers between cache and memory



Asymmetric RAM model

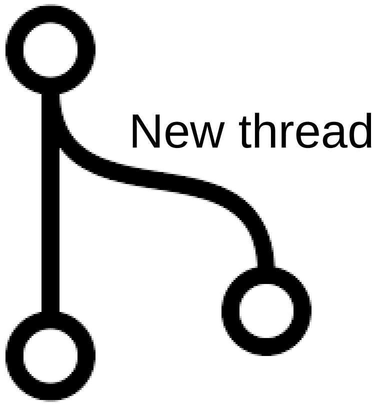
- Write to the memory is more expensive than read



Parallel computational models

- Multiple "threads/processes" can do computation together, and share a memory
- Computation starts with one thread
- Each thread is like a regular RAM
- Each thread can also "create" another thread to run in parallel

- What should we count to evaluate the running time in this case?



So.. Why are we still using RAM model?

- In the sequential setting, it actually provide a way to analyze algorithms nicely (e.g., the comparison of quicksort and selection sort)

- Many other important models are based on that (e.g., a parallel model would assume multiple threads/processors, each behaving like a RAM)

Analysis

- **Time complexity and RAM model**
- **Other models exist!**

- **Analyzing algorithms => time complexity, other cost (space, I/O, etc.), how efficient an algorithm is**

- **Analyzing PROBLEMS => lower bound of a problem, how hard an algorithm is?**
  - Will be covered next week

Lower bound analysis

Analyzing "problems" – lower bound analysis

- **Given a problem, what is the smallest cost we need to pay?**

- **Given an array, what is the "best complexity" you can get to sort it?**
  - We know algorithms of running time $O(n \log n)$
  - But is it the "best"? Does there exist better algorithms? If not, why?

- **Given a sorted array and an element x, what is the "best complexity" can you get to find the rank of x?**
  - We know binary search needs $O(\log n)$ time
  - But is it the "best"? Does there exist better algorithms? If not, why?

Example about lower bound proof: Finding the max of an array

- **Input:** an array of distinct elements
- **Model:** we only care about comparisons: each comparison is a unit cost
- **How many comparisons at least do you need to find the max of the array?**

- **Let's first consider some algorithms for this.**
- **Idea 1: always compare to the champion**

```
ans = a[1];
for (i = 2 to n) {
  if (a[i] > max) max = a[i];
}
```

$n - 1$ comparisons

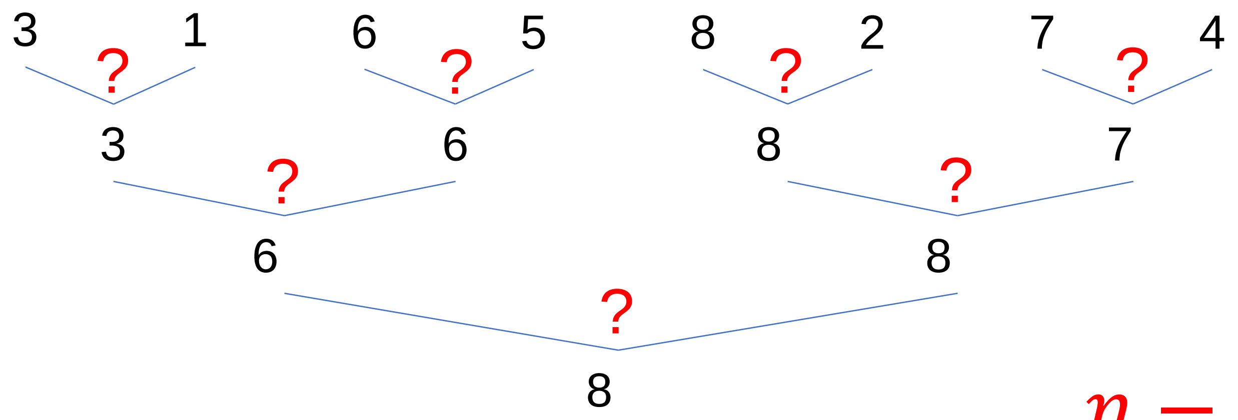
Example about lower bound proof: Finding the max of an array

- **Idea 1: always compare to the champion**

```
ans = a[1];
for (i = 2 to n) {
  if (a[i] > max) max = a[i];
}
```

$n - 1$ comparisons

- **Idea 2: single-elimination tournament?**



$n - 1$ comparisons

Finding the max of an array

- **Can we use fewer than $n - 1$ comparisons to find the max of an array?**
- **Principle:**
  - If an element is not compared to others, we cannot guarantee we find the max
  - If an element loses any comparison, it cannot be the max
  - If an element never lose a comparison, it's possible to be the max
- **Proof:**
  - Call an element "free" if it hasn't been compared to anyone (need to verify!)
  - Call an element "bad" if it loses at least one comparison (rule it out!)
  - Call an element "top" if it never loses any comparison (a candidate!)
  - Initial status: 0 top, n free, 0 bad
  - Final status: 1 top, 0 free, $n - 1$ bad
  - Only when we reached this final status, we find the max

Finding the max of an array – lower bound proof

- **Preliminary**
  - "top": never loses any comparison
  - "bad": lost at least one comparison
  - "free": never compared to others
  - Initial status: 0 top, n free, 0 bad
  - Final status: 1 top, no free, $n-1$ bad
- **What happens when we compare two elements?**

	#top	#bad	#free	
top vs. top	-1	+1	=	
top vs. bad	=	=	=	(top wins)
	-1	+1	=	(bad wins)
top vs. free	=	+1	-1	(top wins)
	-1+1	+1	-1	(free wins)
bad vs. bad	=	=	=	
bad vs. free	=	+1	-1	(bad wins)
	+1	=	-1	(free wins)
free vs. free	+1	+1	-2	

Each comparison increases at most 1 bad!
=> We need at least $n-1$ comparisons to get the final status ($n-1$ bad)

Finding the max of an array – lower bound proof

- **To get an optimal algorithm, we have to guarantee to increase #bad by 1 in every comparison**
- **Only compare**
  - top to top, or
  - free to free, or
  - top to free
  - (guarantee to increase #bad by 1)
- **Algorithm:**
  - If there are more than two **free or top** elements, **compare** them
  - Until there is only one top

	#top	#bad	#free	
top vs. top	-1	+1	=	
top vs. bad	=	=	=	(top wins)
	-1	+1	=	(bad wins)
top vs. free	=	+1	-1	(top wins)
	-1+1	+1	-1	(free wins)
bad vs. bad	=	=	=	
bad vs. free	=	+1	-1	(bad wins)
	+1	=	-1	(free wins)
free vs. free	+1	+1	-2	

Each comparison increases at most 1 bad!
=> We need at least $n-1$ comparisons to get the final status ($n-1$ bad)

Example about lower bound proof: Finding the max of an array

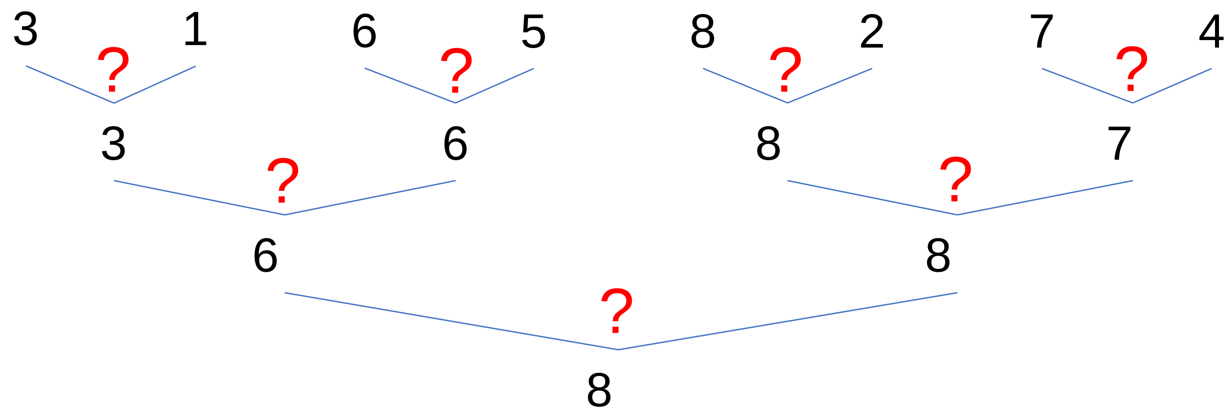
- Idea 1: always compare to the champion

Every comparison is top to free!

```
ans = a[1];
for (i = 2 to n) {
   if (a[i] > max) max = a[i];
}
```

$n - 1$ comparisons

- Idea 2: single-elimination tournament?



First level: free to free

Others: top to top

$n - 1$ comparisons

Upper bound vs. lower bound

- **An upper bound $f(n)$ of the cost of a problem means there exists an algorithm takes at most $f(n)$ steps on any input of size n**
  - So, given this problem, we can just run this algorithm to get an answer
  - $f(n)$ is guaranteed to be sufficient: we don't need more than $f(n)$ costs
- **A lower bound of $g(n)$ means for any algorithm, there exists an input for which all algorithm takes at least $g(n)$ steps on that input**
  - Whatever algorithm you use, you cannot get better than $g(n)$!!
- **When upper bound meets lower bound...**
  - An algorithm has cost $f(n)$, and the best you can do is $g(n) = f(n)$
  - That's an optimal algorithm!

Upper bound vs. lower bound

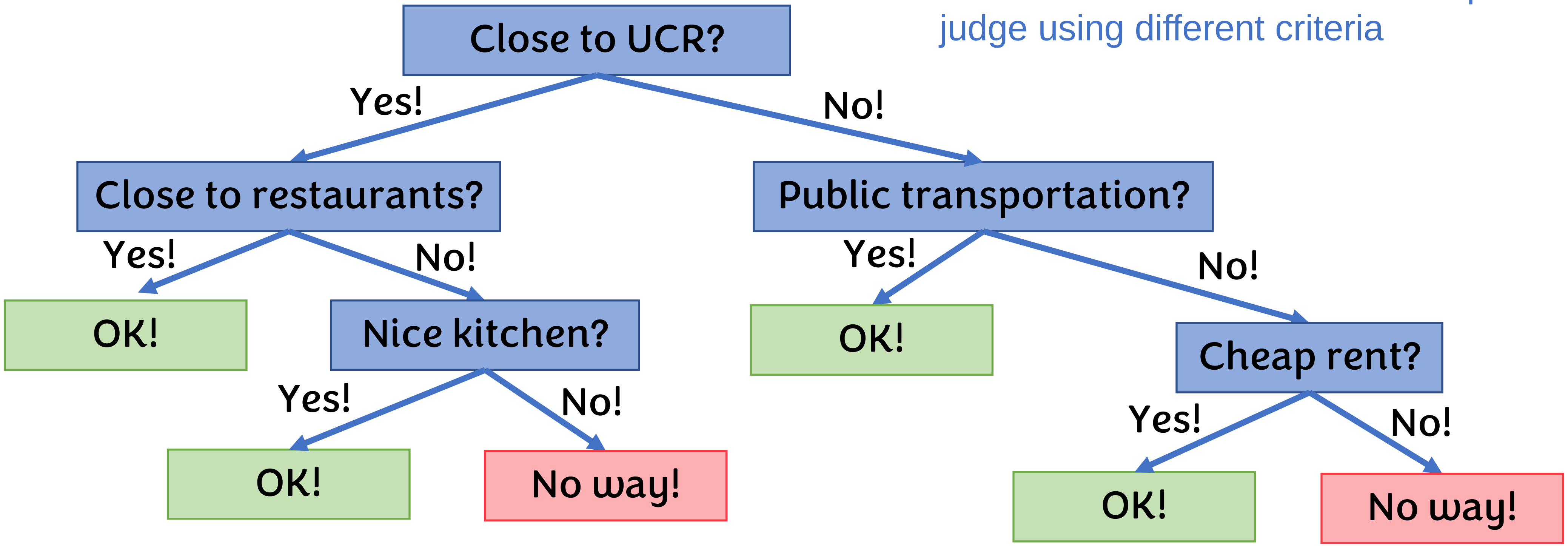
- **Finding max of an array**
  - Lower bound: $n-1$ comparisons (we just proved that)
    - If you use fewer than $n-1$ comparisons, you cannot find the answer
  - Upper bound: the compare-to-champion algorithm does exactly $n-1$ comparisions
    - If you want to find the answer, you don't need more than $n-1$ comparison
  - Upper bound meets lower bound!
  - Comparison-to-champion is an optimal algorithm (w.r.t. #comparisons) to find the max

Decision trees and lower bound proof

Decision trees: partition the decision space and conquer

- You want to rent a house in Riverside, you finally find a candidate, and you need to decide…

Similar to divide-and-conquer:
based on the answer to the first question, we judge using different criteria



See how AI uses a decision-tree-like idea to guess what you are thinking: http://20q.net

What question will you ask?

elephant	TV	bee	clock	chair	Joshua tree	rose	computer

- **Is it a plant?**
  - Yes: Joshua tree, rose. No: TV, clock, chair, elephant, bee, computer
- **Does is use electricity?**
  - Yes: TV, clock, computer. No: elephant, bee, chair, Joshua tree, rose
- **Can you find it in this classroom?**
  - Yes: chair, computer. No: elephant, TV, bee, clock, Joshua tree, rose
- **Does it have legs?**
  - Yes: elephant, chair, bee. No: TV, clock, Joshua tree, rose, computer
- **Is it alive?**
  - Yes: element, bee, Joshua tree, rose. No: TV, clock, chair, computer

Lower bound proof using a "decision tree"

Sorting lower bound

- **What is the minimum number of operations we need to sort n elements?**
- **This question is too vague to be answered**
  - We know $O(n \log n)$ sorting algorithms: they are “almost” the best sorting algorithms we know of…
  - But can we use $O(n)$ to do this? Well, if your elements are integers in [1,n]…
    - `for (i = 1 to n) count[A[i]]++;`
    - `for (x = 1 to n) {`
      - `for (j = 1 to count[x])`
        - `Print x;`
    - `}`
  - **How should we formalize the question??**

We need a formal approach

- **Look at computational models which specify exactly which operations may be performed on the input, and what they cost**
  - E.g., performing a comparison, or swapping a pair of elements

- **An upper bound of $f(n)$ means the algorithm takes at most $f(n)$ steps on any input of size n**

- **A lower bound of $g(n)$ means for any algorithm there exists an input for which the algorithm takes at least $g(n)$ steps on that input**

We need a formal approach

- **Look at computational models which specify exactly which operations may be performed on the input, and what they cost**
  - E.g., performing a comparison, or swapping a pair of elements
- **For sorting algorithms, we usually use the "comparison model"**
  - We only count the number of comparisons used in the algorithm
  - (similar to the finding-max problem, we also used comparison model)
- **Why we study comparison-based sort?**
  - General: no constraint on input type (integer, real, string, positive or negative, pair, complicated struct, key range, hashable or not)… as long as comparable!

Sorting in the Comparison Model

- In the **comparison model**, we have n items in some initial order
- An algorithm may compare two items (asking: is $a_i > a_j$?) at a cost of 1
  - Moving the items is free
- No other operations allowed, such as XORing, hashing, etc.
- Sorting: given an array $a = [a_1, \dots, a_n]$, output a permutation π so that $[a_{\pi(1)}, \dots, a_{\pi(n)}]$ in which the elements are in increasing order
- A sorting algorithm based on comparisons is called **comparison sort**

Lower bound for sorting

- **Of course 1 is a lower bound…**
  - You cannot guarantee to sort the entire array using 1 comparison!

- **Of course n-1 is a lower bound…**
  - We just proved that just finding the maximum value needs n-1 comparisons

- **But… Can we show "better" lower bounds?**
- **We are usually interested in tight lower bounds (the tighter, the better)**
- **For sorting, we can actually show that $\Omega(n \log n)$ is a lower bound**

Sorting lower bound in the Comparison Model

- **Theorem: Any deterministic comparison sort algorithm must perform at least $\Omega(n \log n)$ comparisons to sort $\boldsymbol{n}$ elements in the worst case**
  - More precisely, for any sorting algorithm $\boldsymbol{A}$ with size $\boldsymbol{n} \geq \mathbf{2}$, the #comparisons needed is $\log_2(n!)$
  - i.e., for any sorting algorithm, there exists an input $\boldsymbol{I}$ of size $\boldsymbol{n}$ so that $\boldsymbol{A}$ makes $\geq \log_2 \boldsymbol{n}! = \boldsymbol{\Omega}(\boldsymbol{n} \log \boldsymbol{n})$ comparisons to sort $\boldsymbol{I}$

- **Proof is information-theoretic**

Lower bound proof outline

- In total, there are $n!$ possible inputs (permutations) for an array of size n

- We need to identify the case for the input

- What can we do via a comparison?

All permutations of ranks

a_1	a_2	a_3	a_4
1	2	3	4
1	2	4	3
1	3	2	4
1	3	4	2
1	4	2	3
1	4	3	2
2	1	3	4
……			
4	3	2	1

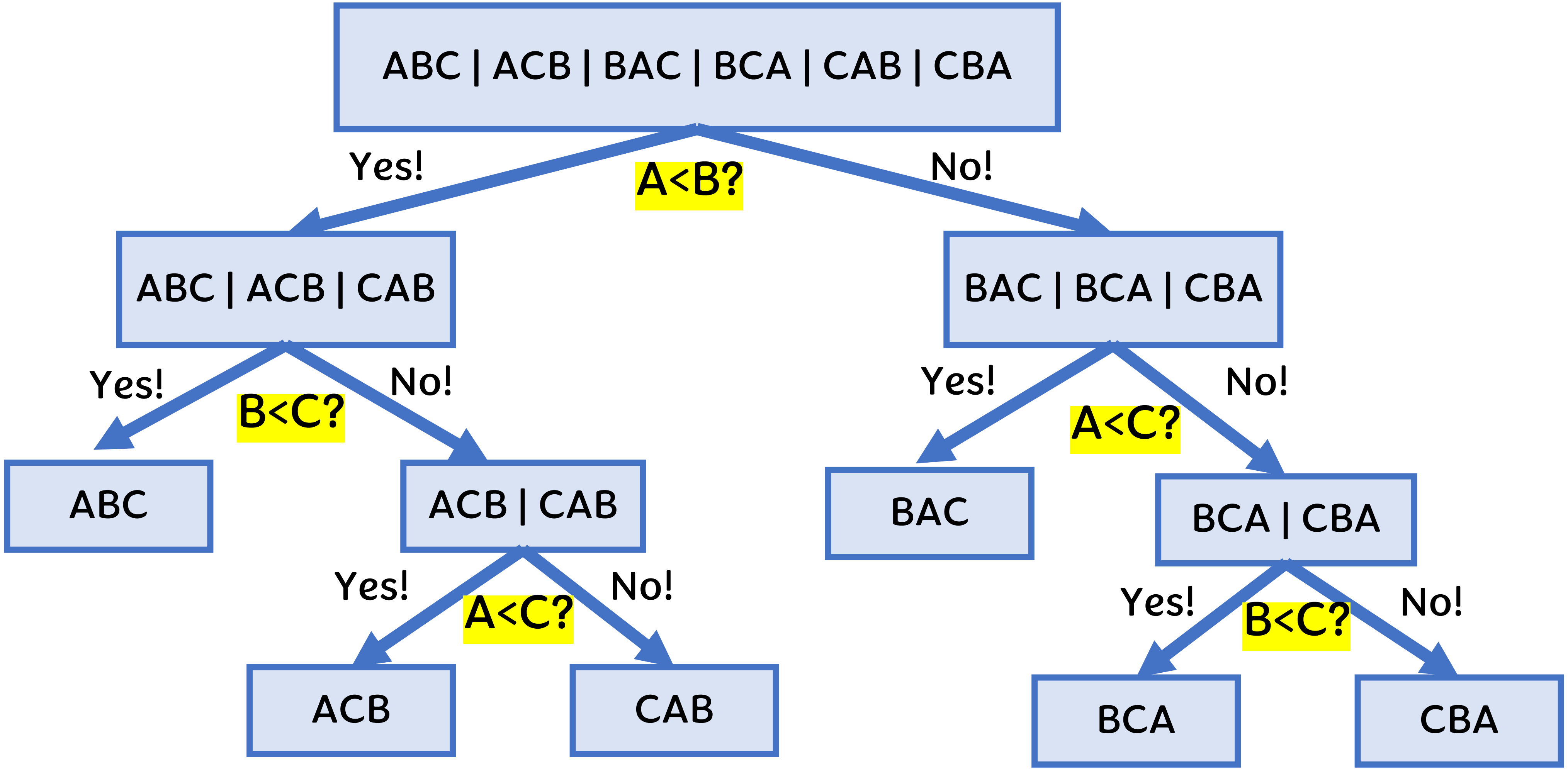
Lower bound proof outline

- **In total, there are $n!$ possible inputs (permutations) for an array of size n**
- **We need to identify the case for the input**
- **What can we do via a comparison?**
  - We can rule out some of the input cases
  - We have to repeat this computation until one case is left

All permutations of ranks

a_1	a_2	a_3	a_4
1	2	3	4
1	2	4	3
~~1~~	~~3~~	~~2~~	~~4~~
1	3	4	2
~~1~~	~~4~~	~~2~~	~~3~~
~~1~~	~~4~~	~~3~~	~~2~~
2	1	3	4
……			
~~4~~	~~3~~	~~2~~	~~1~~

$$a_2 < a_3$$

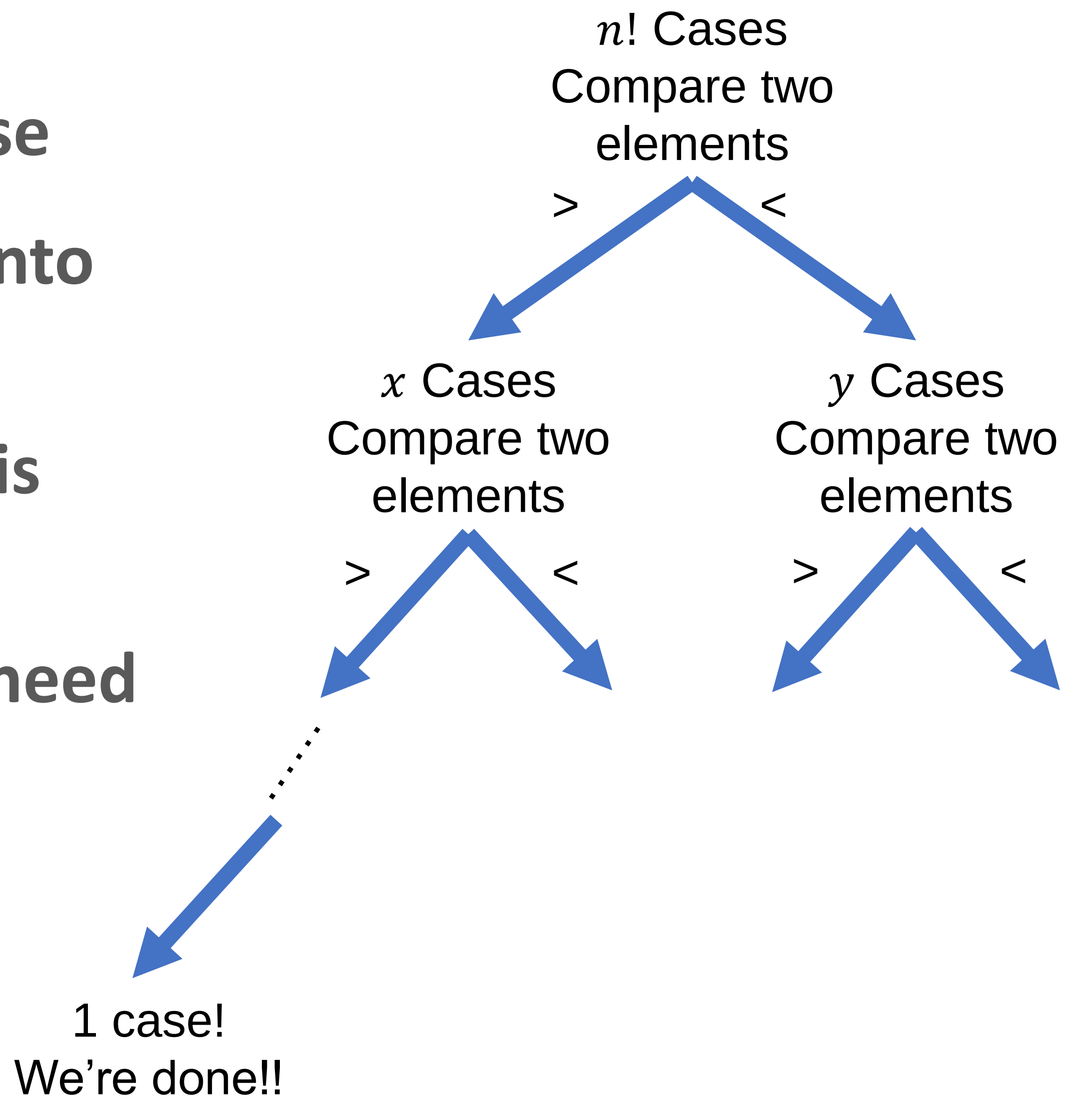


To find the correct solution for any input, we need to distinguish all possible $n!$ Input cases!

Lower bound for sorting algorithms

- We need to identify the input case
- Every comparison split all cases into two parts
- We need to have $n!$ leaves for this decision tree
- What's the number of levels we need for the deepest branch?

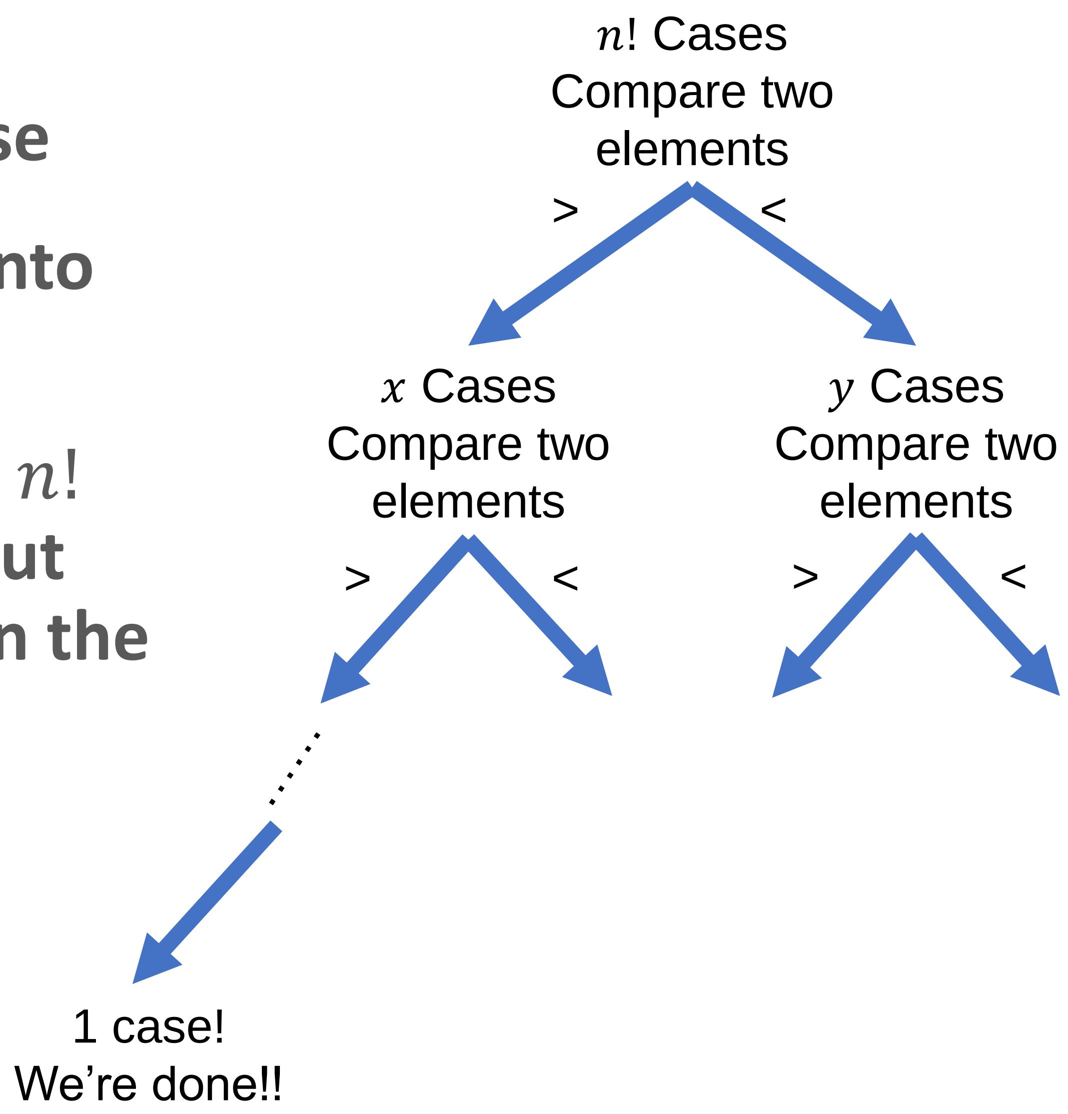
$$\log_2 n! = \Theta(n \log n)$$



Lower bound for sorting algorithms

- We need to identify the input case
- Every comparison split all cases into two parts
- Information-theoretic: need $\log_2 n!$ bits of information about the input before we can correctly decide on the output

$$\log_2 n! = \Theta(n \log n)$$



Why $\log_2 n! = \Theta(n \log n)$?

- $\log_2 n! = \log_2 n + \log_2(n-1) + \cdots + \log_2 1 < n \log_2 n = O(n \log n)$
- $\log_2 n! > \log_2 n + \log_2(n-1) + \cdots + \log_2 \frac{n}{2} > \frac{n}{2} \log_2 \frac{n}{2} = \Omega(n \log n)$

- **We can also use the Stirling's formula:**

$$n! = \sqrt{2\pi n}\left(\frac{n}{e}\right)^n + O(n)$$

- **So** $\log_2 n! = n \log_2 n - n \log_2 e + O(\log_2 n) = \Theta(n \log n)$

Summary for sorting lower bound

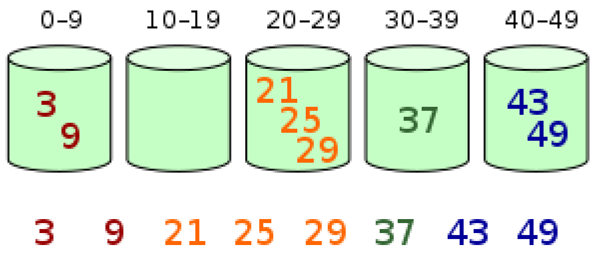
- **A lower bound of $g(n)$ means for any algorithm there exists an input for which the algorithm takes at least $g(n)$ steps on that input**
  - If that matches with the upper bound (time complexity), it means this algorithm is optimal, and the upper bound is tight
- **For sorting algorithm, we use the comparison model that assumes an algorithm compares two items with cost 1, and all other operations are free**
- **We can show that to distinguish $n!$ possible inputs, we need at least $\log_2 \boldsymbol{n}! = \boldsymbol{\Omega}(\boldsymbol{n} \log \boldsymbol{n})$ comparisons, indicating that quicksort and mergesort are optimal comparison-sort algorithms**

Sorting algorithms

- **Comparison-based sorts:**
  - Bubble sort: compare adjacent elements
  - Selection sort: compare to find the smallest, 2nd smallest, …
  - Quicksort: compare to pivots to partition
  - Merge sort: compare in merge
  - Bogosort (permutation sort, stupid sort or slowsort, based on infinite monkey theorem)
- **Non-comparison-based sorts:**
  - Counting sort: only positive integers in small range
  - Bucket sort: have to know key-range (create buckets)
  - Radix sort: only integers (sorting based on the bits)
  - Sleep sort: hmmm…

Bogosort:

```
while not is_sorted(data) {
    shuffle(data);
}
return data;
```



Sleep sort:

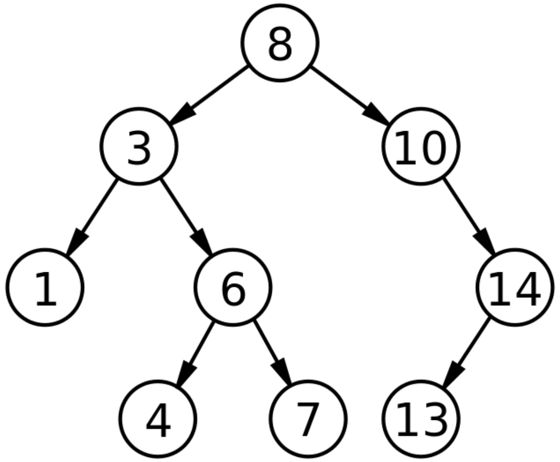
```
printNumber(n) {
    sleep n seconds;
    print n; }

// start n threads
parallel_for (i=1 to n)
  printNumber(A[i]);
wait for all threads to finish
```

What sorting algorithm should I use?

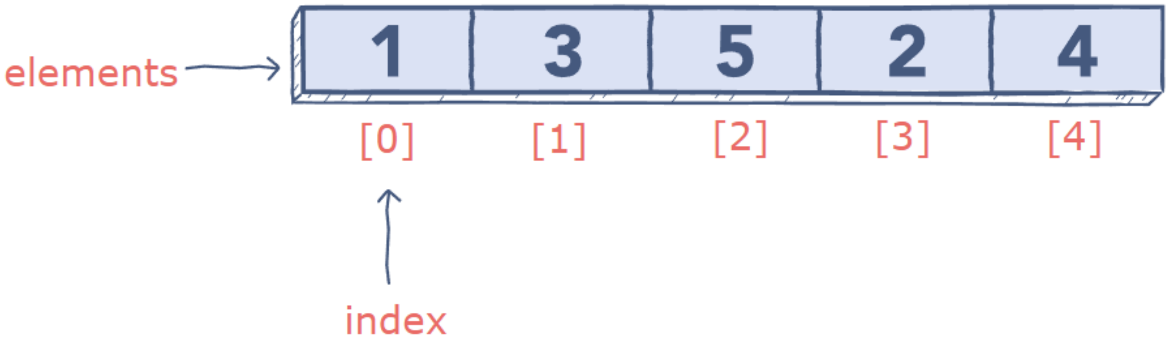
- **Merge sort and quicksort both are $O(n \log n)$**
  - In expectation for quicksort
- **Which one is faster?**
  - In practice, quicksort usually shows better performance
  - (that's why it's called quicksort. std::sort in STL also uses quicksort)
  - Why? Because quicksort is in-place (no extra space used), while merge requires extra space to hold the merging result temporarily (and then write back)
  - Also is more cache-friendly
- **Time complexity help you roughly predict the performance**
- **Many other practical considerations may affect performance**
- **We can develop more theoretical models, get more experience in coding, … to better understand the performance**

Pointer machine model

- You cannot random access the memory based on address (i.e., not assume contiguous memory)
- You can access the memory only use pointers



You can access a memory location by a pointer

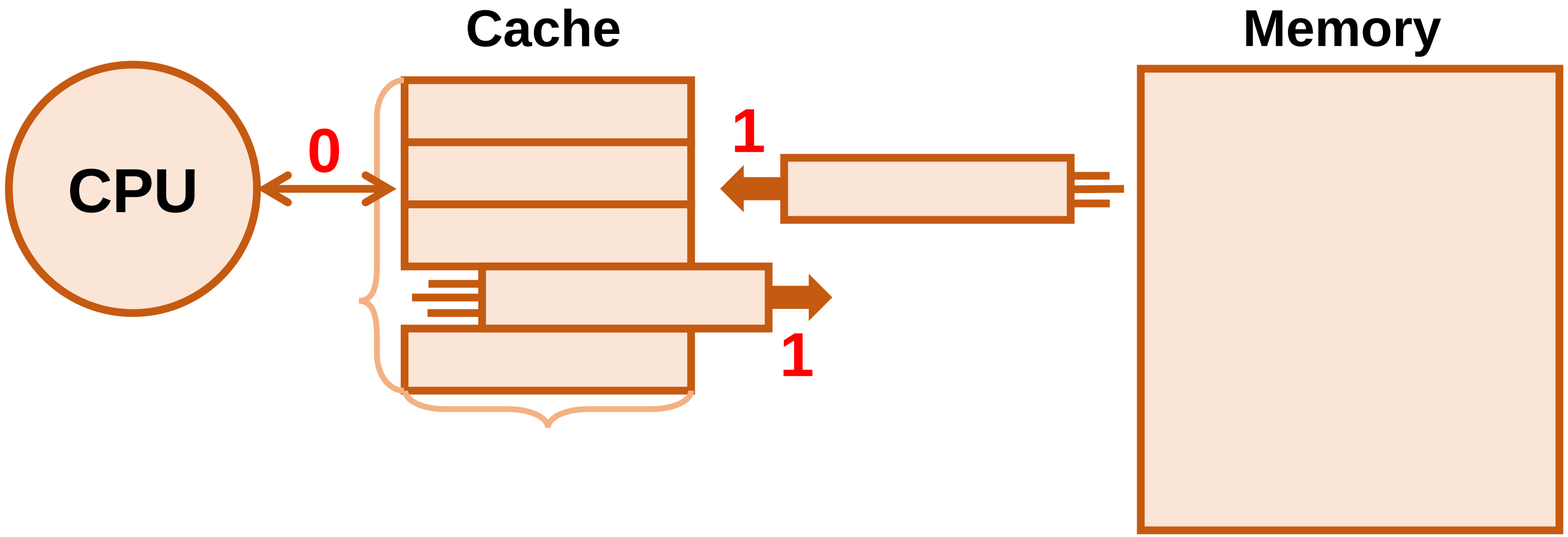


You **cannot** find the i-th element in an array by using a[i]

- Other models: cell-probe model, counter-machine model, etc.

I/O model

- Measures the number of reads and writes of an algorithm



- Examples of lower-bound proofs on the I/O model:
  Improved Parallel Cache-Oblivious Algorithms for Dynamic Programming and Linear Algebra

Upper bound vs. lower bound

- **An upper bound $f(n)$ of the cost of a problem means there exists an algorithm takes at most $f(n)$ steps on any input of size n**
  - So, given this problem, we can just run this algorithm to get an answer
  - $f(n)$ is guaranteed to be sufficient: we don't need more than $f(n)$ costs
- **A lower bound of $g(n)$ means for any algorithm, there exists an input for which all algorithm takes at least $g(n)$ steps on that input**
  - Whatever algorithm you use, you cannot get better than $g(n)$!!
- **When upper bound meets lower bound…**
  - An algorithm has cost $f(n)$, and the best you can do is $g(n) = f(n)$
  - That's an optimal algorithm!

Summary for lower bounds

- **The minimum "cost" to solve a PROBLEM using ANY algorithms**
- **Lower bounds are for problems, not algorithms**
- **Commonly used models for analyzing lower bounds: comparison model, pointer machine model, I/O model, cell-probe model, counter-machine model**
- **Analyzing or at least knowing the lower bound is helpful**
  - Showing the NP-hard / NP-completeness can be considered as a "special" lower bound

Lower bound proof

- **Further reading**
  - https://courses.cs.vt.edu/~cs4104/shaffer/Spring2007/bounds.pdf