Sorting

Data Structures and Algorithms

Nanjing University, Fall 2021
郑朝栋

Can we sort faster than $\Theta(n \log n)$?

Complexity of a problem:

Upper bound and Lower bound

- Consider a problem $\mathcal{P}$. (Such as the sorting problem.)
- **Upper bound:** how fast can we solve the problem?
  - The (worst-case) runtime of an algorithm $\mathcal{A}$ on input of size n is: $T_{\mathcal{A}}(n) = \max_{|I|=n}\{cost_{\mathcal{A}}(I)\}$.
  - $T_{\mathcal{A}}(n)$ *upper bounds* the complexity of solving problem $\mathcal{P}$.
  - Every valid algorithm gives an upper bound on the complexity of $\mathcal{P}$.
- **Lower bound:** how slow solving the problem has to be?
  - The worst-case complexity of $\mathcal{P}$ is the worst-case runtime of the *fastest* algorithm that solves $\mathcal{P}$: $T_{\mathcal{P}}(n) = \min_{\mathcal{A} \text{ solves } \mathcal{P}}\{\max_{|I|=n}\{cost_{\mathcal{A}}(I)\}\}$
  - $T_{\mathcal{P}}(n)$, usually in the form of $\Omega(f(n))$, *lower bounds* the complexity of solving problem $\mathcal{P}$.
  - $T_{\mathcal{P}}(n) = \Omega(f(n))$ means **any** algorithm has to spend $\Omega(f(n))$ time to solve problem $\mathcal{P}$.

Lower bound of a problem

- A **lower bound** of $\Omega(f(n))$ for a problem $\mathcal{P}$ means *any* algorithm that solves $\mathcal{P}$ has *worst-case* runtime $\Omega(f(n))$.

- Larger lower bound is a *stronger* lower bound. (On the other hand, smaller upper bound is better.)

- But how do we prove a lower bound?!
  - It is usually unpractical to examine all possible algorithms…
  - Instead, rely on structures/properties of the problem itself…

Techniques for proving lower bounds:

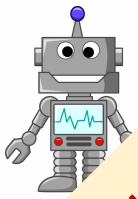
The Adversary Argument

- Imagine an adversary Eve that determines the input I.
- Whenever the algorithm $\mathcal{A}$ asks about the input, Eve answers.
- If $\mathcal{A}$ does not ask enough questions, then there are two inputs that are both *consistent* with Eve's answers, but result in *different* outputs. Say I_0 and I_1 are two such inputs.
- If $\mathcal{A}$ outputs $result(I_0)$ (resp., $result(I_1)$), then Eve can "reveal" that I was actually I_1 (resp., I_0), making $\mathcal{A}$'s output invalid.
- $\mathcal{A}$ is **not** an algorithm which solves the considered problem!
- $\mathcal{A}$ can be **any** algorithm: no restrictions posed on its behavior.
- Any algorithm that does solve the problem must ask

The adversary argument in action:

Lower bound for sorting

Assume we want to sort n integers.



The algorithm, which queries the input $n-1$ times, does not solve the problem.

Any algorithm which queries the input at most $n-1$ times does not solve the problem.

Solving the "sort n integers" problem has a time complexity of $\Omega(n)$.

"What -1?"

Busy ...busy...

What is a_1?"

...

Place a_2 at beginning or e

a_1 =1	a_2 =?	...	a_{n-1} =1	a_n =1

... (always say 1)

$n-1$ queries, $\Theta(n)$ work

a_2 <1	a_1 =1	...	a_{n-1} =1	a_n =1

sorry but $a_2 > 1$

a_1 =1	a_3 =1	...	a_n =1	a_2 >1

Sorting needs $\Omega(n)$ time.

Can we sort faster than $\Theta(n \log n)$?

No, at least not for a large class of algorithms...

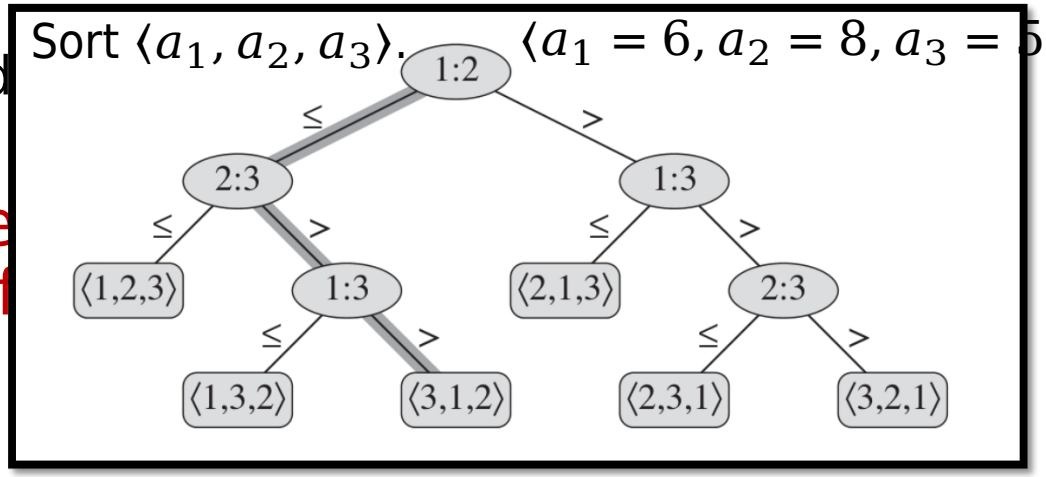
Comparison-based sorting lower bound

- A **comparison-based sorting algorithm** determines sorted order only based on *comparisons* between the input items.
- In a comparison sort, only comparisons between elements are used to gain order information about the input sequence.
  - Given a_i and a_j, only use "<", "≤", "=", "≥", or ">" to gain order info.
  - Particularly, the algorithm **cannot** inspect the values of input items.

Any comparison-based sorting algorithm has time complexity $\Omega(n \log n)$, in the worst case.

Decision Tree

- Decisions trees can be used to describe algorithms.
  - A decision tree is a tree.
  - Each internal node denotes a query the algorithm makes on input.
  - Outgoing edges denote the possible answers to that query.
  - Each leaf denotes an output.
- One execution of the algorithm is a path from root to a leaf.
  - At each internal nod
    next.
- The worst-case time
  of the longest path f
  of the tree!



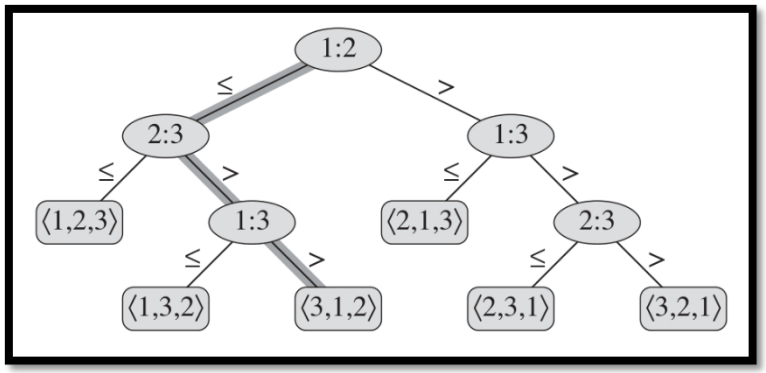
Decision tree in action:

Comparison-based sorting lower bound

Any comparison-based sorting algorithm has time complexity $\Omega(n \log n)$, in the worst case.

- Assume input items are distinct.
- Assume the algorithm only uses "≤" to do comparison.
- We can use a *binary comparison tree* to describe the alg.
  - Each internal node has two outgoing edges.
  - Each internal node denotes a query of the form "$a_i \leq a_j$".
- **The tree must have $\geq n!$ leaves.**
- The height of the tree must be $\geq \lg(n!)$, which is $\Omega(n \log n)$.

*: 基于信息论的下界证明 $2^x >= n!$



Sorting needs $\Omega(n)$ time.

Can we sort faster than $\Theta(n \log n)$?

No, at least not for comparison-based sorting...

Humm, maybe “non-comparison-based” sorting?

Bucket sort

- Assume we want to sort n integers, and we know each item is from the set $[10]$. Can we beat $\Theta(n \log n)$?
- Of course, very easy!
  - Create 10 empty lists. (These are the **buckets**.)
  - Scan through input, for each item, append it to the end of the corresponding list.
  - Concatenate all lists.
- This algorithm only takes $\Theta(n)$ time.
- This is **not** a comparison based algorithm.
  - No comparison between items are made.
  - Instead the algorithm uses actual values of the items.

Bucket sort

- In general, if the input items are all from set $[d]$, then we can use the following algorithm to sort them.

BucketSort(A, d):

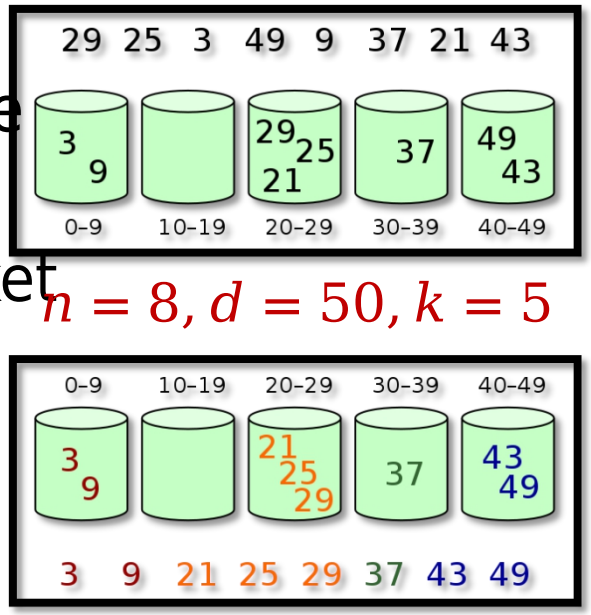
```
<L_1,L_2,...,L_d> = CreateBuckets(d)
for (i=1 to A.length)
  AssignToBucket(A[i])
CombineBuckets(L_1,L_2,...,L_d)
```

- Total time complexity is $\Theta(n + d)$.
  - $\Theta(d)$ time to create buckets.
  - $\Theta(n)$ time to assign items to buckets.
  - $\Theta(d)$ time to combine buckets.

What if $n \ll d$?
Say sort 1000 64-bit integers.

Bucket sort

- If the range of items' values is too large, allow each bucket to hold multiple values.
- Allocate k buckets each responsible for an interval of size d/k.
- But now we need to sort each bucket before combining them.



BucketSort(A, k):

<L_1,L_2,…,L_k> = CreateBuckets(k)
for (i=1 to A.length)
  AssignToBucket(A[i])
for (j=1 to k)
  SortWithinBucket(L_j)
CombineBuckets(L_1,L_2,…,L_k)

Bucket sort

- Runtime is $\Theta(n + k)$, plus cost for sorting within buckets.
- If items are uniformly distributed and we use insertion sort,
  expected cost for sorting is $O(k \cdot (n/k)^2) = O(n^2/k)$.
- Expected total runtime is $O(n + n^2/k)$,
  which is $O(n)$ when we
- BucketSort can be stabl

BucketSort(A, k):

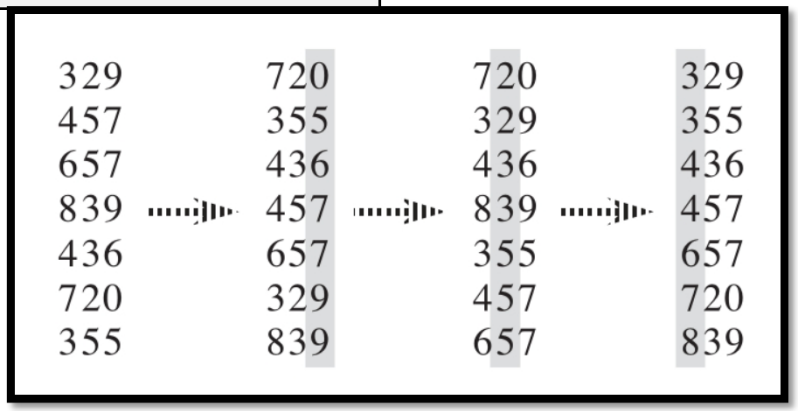
<L_1,L_2,...,L_k> = CreateBuckets(k)
for (i=1 to A.length)
  AssignToBucket(A[i])
for (j=1 to k)
  SortWithinBucket(L_j)
CombineBuckets(L_1,L_2,...,L_k)

Radix sort

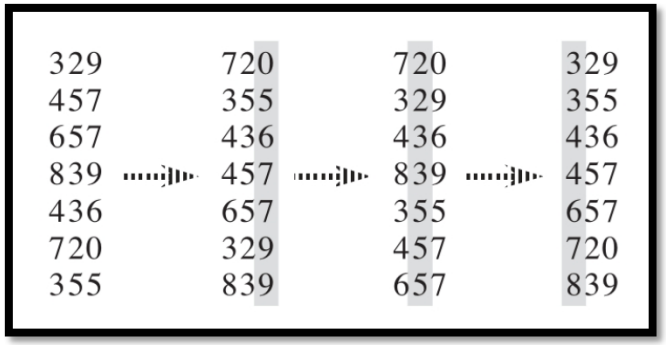
- Assume we want to sort n decimal integers each of d-digits.
- How about recursive bucket sort?
  - Based on most significant bit, assign items to 10 buckets.
  - Sort recursively in each bucket (i.e., use 2nd most significant bit).
- Valid but not for now…
- gnificant bit.

RadixSort(A, d):

```
for (i=1 to d)
  use-a-stable-sort-to-sort-A-on-digit-i
```



Radix sort



RadixSort(A, d):
for (i=1 to d)
  use-a-stable-sort-to-sort-A-on-digit-i

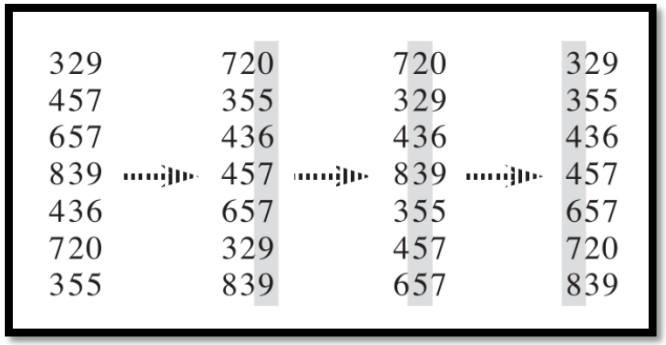
Why it works?!

Claim: after i^{th} iteration, items are sorted by their rightmost

Use induction to prove the claim.

- [*Basis*] Claim holds after the first iteration.
- [*Hypothesis*] Assume claim holds after first $k-1$ iterations.
- [*Inductive Step*] Consider two items a and b after k iterations.
  - W.l.o.g., assume a appears before b. Thus, $a[k] \leq b[k]$.
  - If $a[k] < b[k]$, then it must be $a[k...1] < b[k...1]$.
  - If $a[k] = b[k]$, since we use stable sort, it must be $a[(k-1)...1] \leq b[(k-1)...1]$. Again, $a[k...1] \leq b[k...1]$.

Radix sort



RadixSort(A, d):
for (i=1 to d)
  use-a-stable-sort-to-sort-A-on-digit-i

RadixSort(A, d):
for (i=1 to d)
  use-**bucket**-**sort**-to-sort-A-on-digit-i

How much time?

Since only considering decimal numbers, we only need $10 = \Theta(1)$ bucket

RadixSort can sort n decimal d-digits numbers in $O(dn)$ time

Summary

Lower Bounds:

- Sorting needs $\Omega(n)$ time. (adversary argument)
- Comparison-based sorting needs $\Omega(n \log n)$. (decision tree)

Upper Bounds:

- There are $O(n \log n)$ comparison-based sorting algorithms.
- BucketSort, RadixSort can be $\Theta(n)$ in many cases.

Reading

- [CLRS] Ch.8