

COMP202

Complexity of Algorithms

Sorting

Reading materials: Exercise 2.4, Chapter 7, Chapter 8 in CLRS.

Learning outcomes

- 1 Understand the sorting problem, and its fundamental importance in algorithms.
- 2 Have a wide knowledge of different types of sorting algorithms available (such as quick sort, counting sort, radix sort), as well as their algorithmic complexity (i.e. running time).
- 3 Comprehend the idea behind the divide-and-conquer methods of merge-sorting and quick sorting.
- 4 Be able to explain the mechanism behind the *randomized* quick sort algorithm, and *why* randomization is used.

Sorting

Sorting problem: Given a collection, C , of n elements (and a total ordering) arrange the elements of C into non-decreasing order, e.g.

I/P

45	3	67	1	5	16	105	8
----	---	----	---	---	----	-----	---

EXPECTED
O/P

1	3	5	8	16	45	67	105
---	---	---	---	----	----	----	-----

Sorting (cont.)

Sorting is a fundamental algorithmic problem in computer science.

Many algorithms perform sorting (as a subroutine) during their execution. Hence, efficient sorting methods are crucial to achieving good algorithmic performance.

We will investigate various methods that we can use to sort items. Why several methods?

We may not always require a fully sorted list (for example), so some methods might be more appropriate depending upon the exact task at hand.

Sorting algorithms might be directly adaptable to perform additional tasks and directly provide solutions in this fashion.

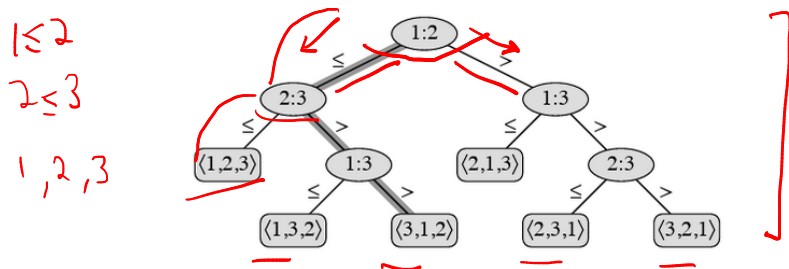
Comparison-based sorting

$$\underline{[x_i \geq x_j] ?}$$

We've already seen two sorting algorithms. These are MergeSort and HeapSort.

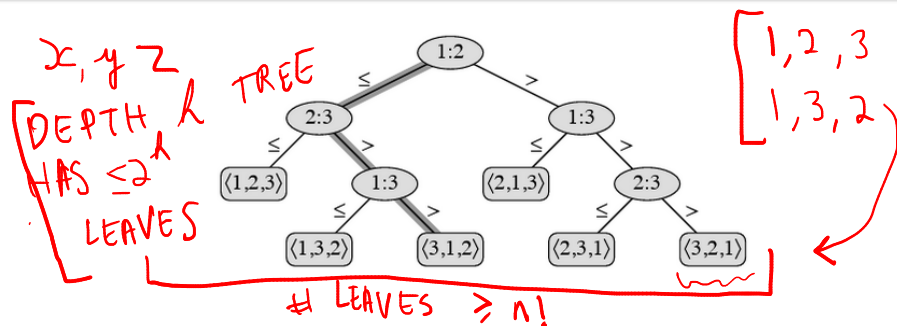
These are both comparison-based algorithms. Recall that they both had a time complexity of $O(n \log n)$. We will first show that no comparison-based sorting technique can have a better running time.

Decision tree lower bound



The above is an example of a sorting algorithm on an input list of 3 elements. The notation $i : j$ means a_i is compared with a_j .

Decision tree lower bound



The above is an example of a sorting algorithm on an input list of 3 elements. The notation $i : j$ means a_i is compared with a_j .

Internal nodes are unit cost, leaves correspond to the output of the algorithm on that path. The cost of the algorithm is at least the depth of the decision tree.

Total number of leaves must be at least 6 (more generally, at least $n!$. Why?).

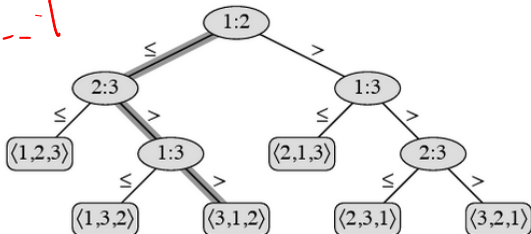
$$2^h \geq n! \Rightarrow h \geq \log(n!)$$

Decision tree lower bound

$$n \cdot (n-1) \cdot \dots \cdot 1$$

$$\log(n!)$$

$$\Theta(n \log n)$$



The above is an example of a sorting algorithm on an input list of 3 elements. The notation $i : j$ means a_i is compared with a_j . Internal nodes are unit cost, leaves correspond to the output of the algorithm on that path. The cost of the algorithm is at least the depth of the decision tree.

Total number of leaves must be at least 6 (more generally, at least $n!$. Why?). Thus, the depth must be at least $\log(n!) = \Omega(n \log n)$ (**see Stirling's approximation**).

Counting inversions

This example is inspired by (if not directly related to) some of the “ranking systems” that are becoming more popular on some websites.

Counting inversions

This example is inspired by (if not directly related to) some of the “ranking systems” that are becoming more popular on some websites.

Suppose that you’ve rated a set of films or books (for example). In particular, you’ve rated n films by ranking them from your most favorite (ranked at 1) to least favorite (ranked at n).

In order to give a recommendation to you, this website wants to compare your ratings of these films with those of other people (for the same films) to see how similar they are.

How can you do this?

Counting inversions (cont.)

In other words, how can you compare your rankings

1 2 3 4 5 6 7 8 9 10

to another ranking

2 7 10 4 6 1 3 9 8 5?

Or even to another person's rankings

8 9 10 1 3 4 2 5 6 7?

Which one of these is “closest” to your rankings?

Counting inversions (cont.)

One proposed way of measuring the similarity is to count the number of *inversions*.

Counting inversions (cont.)

One proposed way of measuring the similarity is to count the number of *inversions*.

Suppose that

$$a_1, a_2, a_3, \dots, a_n$$

denotes a permutation of the integers $1, 2, \dots, n$. The pair of integers i, j are said to form an inversion if $i < j$, but $a_i > a_j$.

(We can generalize this idea to any sequence of distinct integers.)

Counting inversions (cont.)

One proposed way of measuring the similarity is to count the number of *inversions*.

Suppose that

$$a_1, a_2, a_3, \dots, a_n$$



denotes a permutation of the integers $1, 2, \dots, n$. The pair of integers i, j are said to form an inversion if $i < j$, but $a_i > a_j$.

(We can generalize this idea to any sequence of distinct integers.)

We will count the number of inversions to measure the similarity of someone's rankings to yours.

Counting inversions (cont.)

For example, the permutation

1 2 4 3

contains one inversion (the 4 and the 3), while the permutation

1 4 3 2

has three (the 3, 4 pair, the 2, 3 and the 2, 4 pair).

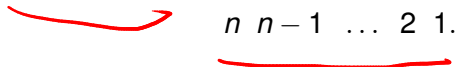
In other words, to find the number of inversions, we count the pairs $i \neq j$ that are *out of order* in the permutation.

Counting inversions (cont.)

The number of inversions can range from 0, for the permutation

$$1 \ 2 \ 3 \ \dots \ n,$$

up to $\binom{n}{2} = \frac{n(n-1)}{2}$ for the permutation


$$n \ n-1 \ \dots \ 2 \ 1.$$

Other examples:

2 1 3 4 5	has one inversion,
2 3 4 5 1	has four inversions,
5 2 3 4 1	has seven inversions, and
5 4 3 1 2	has nine inversions.

Counting inversions (cont.)

In terms of the ranking system describe earlier, *the number of inversions* for a permutation is a measure of how “out of order” it is as compared to the identity permutation

1 2 3 ... n

and hence could be used to measure the “similarity” to the identity permutation.

Counting inversions (cont.)

In terms of the ranking system describe earlier, *the number of inversions* for a permutation is a measure of how “out of order” it is as compared to the identity permutation

1 2 3 ... n

and hence could be used to measure the “similarity” to the identity permutation.

Note, however, that two different permutations can have the same number of inversions.

2 1 3 4 5

and

1 2 4 3 5

each have a single inversion, but are obviously not the same permutation.

Counting inversions: How do we do it?

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

So how do we count the number of inversions in a given permutation of n numbers?

The “naive” approach is to check all $\binom{n}{2}$ pairs to see if they form an inversion in the permutation. This gives an algorithm with $\Omega(n^2)$ running time.

Can we do better?

Counting inversions: How do we do it?

So how do we count the number of inversions in a given permutation of n numbers?

The “naive” approach is to check all $\binom{n}{2}$ pairs to see if they form an inversion in the permutation. This gives an algorithm with $\Omega(n^2)$ running time.

Can we do better?

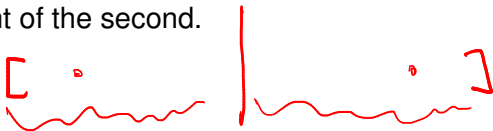
Claim: We can count inversions using a divide-and-conquer algorithm that runs in time $O(n \log n)$.

A divide-and-conquer way to count inversions

Idea:

As with similar divide-and-conquer algorithms, we divide the permutation into two (nearly equal) parts. Then we (recursively) count the number of inversions in each part.

This gives us most of the inversions. We then need to get the number of inversions that involve one element of the first list, and one element of the second.



A divide-and-conquer way to count inversions

Idea:

As with similar divide-and-conquer algorithms, we divide the permutation into two (nearly equal) parts. Then we (recursively) count the number of inversions in each part.

This gives us most of the inversions. We then need to get the number of inversions that involve one element of the first list, and one element of the second.

To do that we can *sort* each sublist and merge them into a single (sorted) list. As we merge them together into a single list, we can count the inversions from such pairs mentioned above.

A divide-and-conquer way to count inversions

Idea:

As with similar divide-and-conquer algorithms, we divide the permutation into two (nearly equal) parts. Then we (recursively) count the number of inversions in each part.

This gives us most of the inversions. We then need to get the number of inversions that involve one element of the first list, and one element of the second.

To do that we can *sort* each sublist and merge them into a single (sorted) list. As we merge them together into a single list, we can count the inversions from such pairs mentioned above.

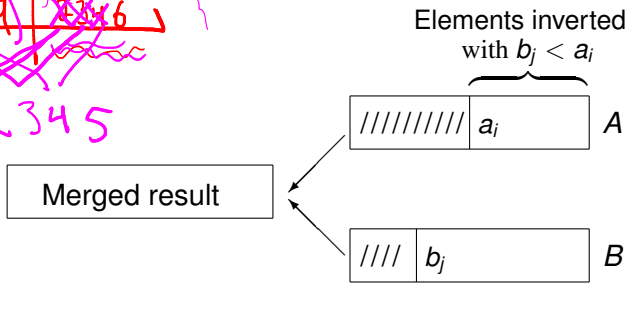
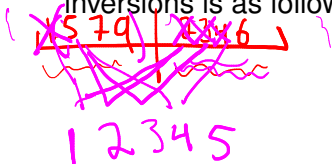
In other words, we're performing a modified MergeSort!

Divide-and-conquer for counting inversions

Suppose that we've divided the list into A (the first half) and B (the second half) and have counted the inversions in each.

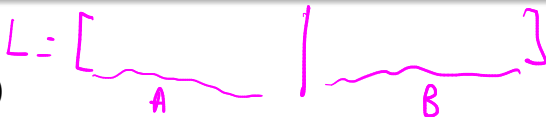
After sorting them, the idea for counting the additional

inversions is as follows:



As we merge the lists, every time we take an element from the list B , it forms an inversion with all of the *remaining* (unused) elements in list A .

A recursive algorithm for counting inversions



COUNTINVERSIONS(L)

- ▷ Input: A list, L , of distinct integers.
- ▷ Output: The number of inversions in L .

```
1  if  $L$  has one element in it then
2      there are no inversions, so Return (0,  $L$ )
3  else
4      ▷ Divide the list into two halves
5           $A$  contains the first  $\lfloor n/2 \rfloor$  elements
6           $B$  contains the last  $\lceil n/2 \rceil$  elements
7       $(k_A, A) = \text{COUNTINVERSIONS}(A)$ 
8       $(k_B, B) = \text{COUNTINVERSIONS}(B)$ 
9       $(k, L) = \text{MERGEANDCOUNT}(A, B)$ 
10     Return  $(k_A + k_B + k, L)$ 
```

Handwritten pink annotations: A bracket groups lines 7 and 8, and an arrow points from the result of line 8 to the final return statement in line 10.

The MERGEANDCOUNT method

MERGEANDCOUNT(A, B)

- 1 $Current_A \leftarrow 0$
- 2 $Current_B \leftarrow 0$
- 3 $Count \leftarrow 0$
- 4 $L \leftarrow$ empty list
- 5 **while** both lists (A and B) are non-empty
- 6 Let a_i and b_j denote the elements pointed to
 by $Current_A$ and $Current_B$.
- 7 Append the smaller of a_i and b_j to L .
- 8 **if** b_j is the smaller element **then**
- 9 Increase $Count$ by the number of elements
 remaining in A .
- 10 Advance the $Current$ pointer of the appropriate list.
- 11 Once one of A and B is empty, append the remaining
 elements to L .
- 12 Return ($Count, L$)

Counting inversions - The payoff

As mentioned earlier, this method for counting inversions is basically a modified version of the MergeSort algorithm.

Hence, we can count the number of inversions in a permutation in time $O(n \log n)$.

L=

2

3

7

8

|

1

4

6

5

23/78

14/65

23

78

14

65

2 3

7 8

1 4

6 5

23

78

14

56

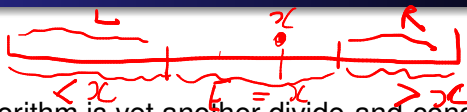
~~23~~78

~~14~~~~5~~~~6~~

INV+=1

6,5 1
(1,2) (1,3) + 4
(1,4) (1,5)
47, 48 + 2
57, 58 + 2
67, 68 + 2
11

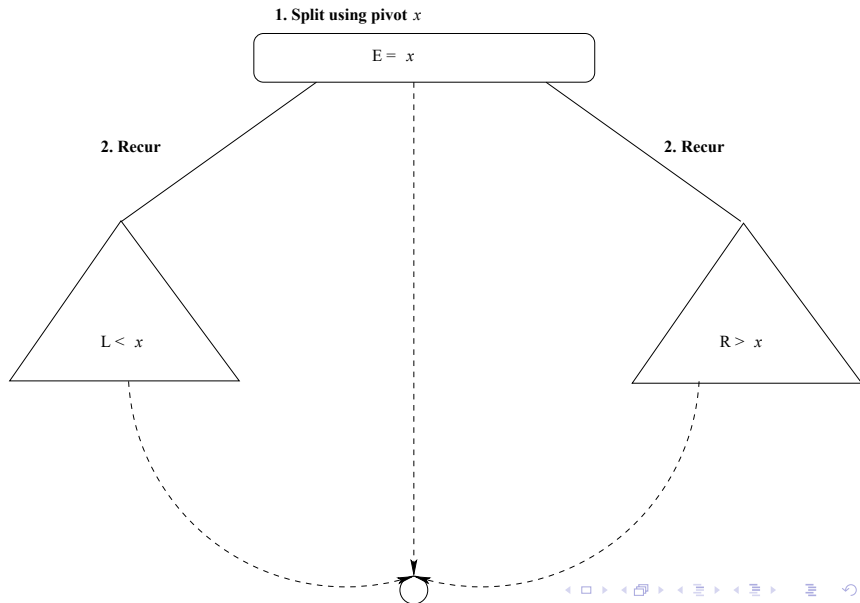
1 2 3 4 5 6



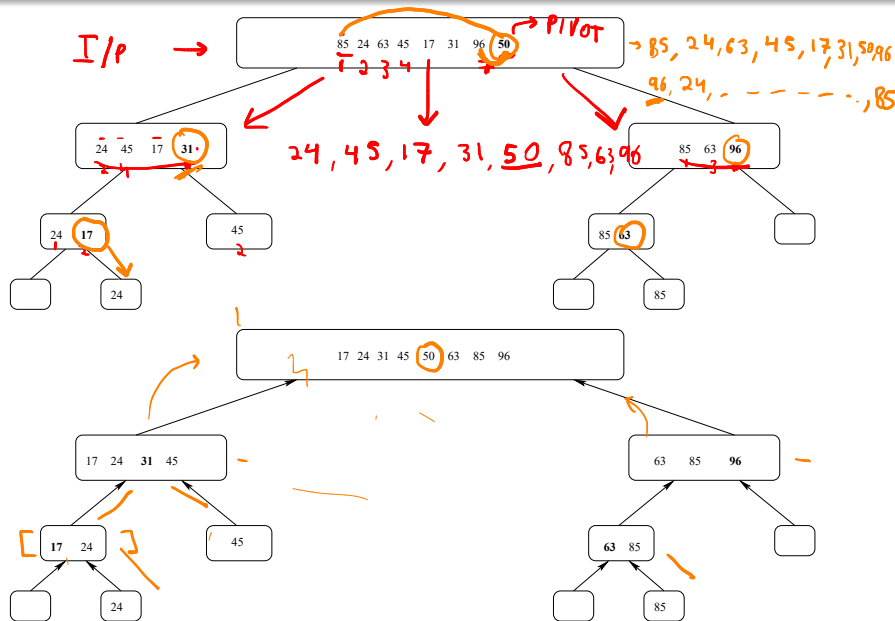
The QuickSort algorithm is yet another divide-and-conquer method for performing sorting. This method differs from MergeSort in that the steps for QuickSort are as follows:

- Divide: If $|S| > 1$, select a *pivot element* x in S and create three sequences: L , E , and G where
 - L stores elements in S that are less than x ,
 - E stores elements in S that are equal to x , and
 - G stores elements in S that are greater than x .
- *Recur*: Recursively sort the sequences L and G .
- Conquer: Put sorted elements from L , E , and G together to form a sorted list.

QuickSort Tree



QuickSort - Example



QuickSort

What is the running time of the QuickSort algorithm?

- Let s_i be the sizes of the input nodes (at depth i) in the tree.
- $s_i \leq n - i$. (Note that $s_i = n - i$, when use of *pivots* leads to a single empty sequence (for L or G).)
- The worst case complexity is bounded by

$$O(\sum_{i=0}^{n-1} s_i) = O(\sum_{i=0}^{n-1} n - i) = O(\sum_{i=0}^{n-1} i).$$

Since

$1 + 2 + \dots + n - 1 = \frac{n(n-1)}{2}$

$\sum_{i=0}^{n-1} i = \frac{(n-1)n}{2}$

the *worst case* running time of QuickSort is $O(n^2)$.

The worst case running time of QuickSort occurs when one of the sublists is in fact empty.

To avoid this worst case run time, we want to ensure that each of L and G are sufficiently large for each recursive call.

For any deterministic method of choosing a pivot element, we can design some list so that the running time is (almost) n^2 .

The worst case running time of QuickSort occurs when one of the sublists is in fact empty.

To avoid this worst case run time, we want to ensure that each of L and G are sufficiently large for each recursive call.

For any *deterministic* method of choosing a pivot element, we can design some list so that the running time is (almost) n^2 .

- Solution: Select a pivot element randomly!

- Probability theory is useful in many contexts in the theory of algorithms.

It can be used to analyze the *average-case* performance of algorithms (by *weighting* the running time according to a probability distribution on inputs).

- Probability theory is useful in many contexts in the theory of algorithms.

It can be used to analyze the *average-case* performance of algorithms (by *weighting* the running time according to a probability distribution on inputs).

Randomized algorithms (those that use random bits during their execution to make decisions) are also becoming more commonplace and useful in recent years. Analysis of such randomized algorithms also requires use of probability theory.

See Appendix C in [CLRS] for details on probability theory!

Sample space

- Sample space is the set of all possible *outcomes* from an *experiment*.

Examples:

- 1 Rolling a fair (6-sided) die. The sample space is $\{1, 2, 3, 4, 5, 6\}$.
- 2 Flipping a fair coin. The sample space is $\{H, T\}$.
- 3 Flipping a fair coin until it comes up heads. Here the sample space is *infinite*, with the i^{th} possible *outcome* being a sequence of $i - 1$ tails followed by a single heads, for $i = 1, 2, 3, 4, \dots$

$\{ \underline{H}, \underline{TH}, \underline{TTH}, \dots \}$

A *Probability space* is a sample space \mathcal{S} and a *probability function* Pr , which maps subsets of \mathcal{S} to $[0, 1]$.

Each subset, $\mathcal{A} \subseteq \mathcal{S}$, is called an *event*.

Probability function

$$S = \{s_1, s_2, \dots, s_n\}$$

A probability function, Pr , possesses the following properties:

① $\text{Pr}(\emptyset) = 0$,

② $\text{Pr}(S) = 1$,

③ $0 \leq \text{Pr}(\mathcal{A}) \leq 1$, for any $\mathcal{A} \subseteq S$,

④ If $\mathcal{A}, \mathcal{B} \subseteq S$ and $\mathcal{A} \cap \mathcal{B} = \emptyset$, then

$$\text{Pr}(\mathcal{A} \cup \mathcal{B}) = \text{Pr}(\mathcal{A}) + \text{Pr}(\mathcal{B}).$$

$$\text{Pr}[\{\}] = 0$$

→ A & B ARE "MUTUALLY EXCLUSIVE".

$\text{Pr}[\{s_1, s_2\}]$: PROB. THAT EITHER s_1 OR s_2 HAPPENS

An example of a probability space

Consider the sample space earlier, namely that of flipping a fair coin until it comes up heads.

We can represent the sample space as an infinite set of strings, namely $\{H, TH, TTH, TTTH, \dots\}$.

An example of a probability space

Consider the sample space earlier, namely that of flipping a fair coin until it comes up heads.

We can represent the sample space as an infinite set of strings, namely $\{H, TH, TTH, TTTH, \dots\}$.

The probability function on this sample space assigns these probabilities to events:

$$\begin{array}{rcl} Pr(H) & = & 1/2 \\ Pr(TH) & = & 1/4 \\ Pr(TTH) & = & 1/8 \\ & \dots & \end{array} \quad \begin{array}{l} \rightarrow \\ \rightarrow \\ \rightarrow \end{array} \quad \frac{1}{2} \cdot \frac{1}{2} = \frac{1}{4}$$

$$\rightarrow Pr(H \text{ or } TTH) = \frac{1}{2} + \frac{1}{8} = \frac{5}{8}$$

$\cap = \emptyset$

An example of a probability space

Consider the sample space earlier, namely that of flipping a fair coin until it comes up heads.

We can represent the sample space as an infinite set of strings, namely $\{H, TH, TTH, TTTH, \dots\}$.

The probability function on this sample space assigns these probabilities to events:

$$Pr(H) = 1/2$$

$$Pr(TH) = 1/4$$

$$Pr(TTH) = 1/8$$

...

$$Pr(H \text{ or } TTH) = 1/2 + 1/8 = 5/8$$

...

} INDEPENDENCE
OF EVENTS

(These statements make use of the *independence* of coin flips discussed on the next slide, and the last statement uses the final property of the probability function on the previous slide.)

Independent Events

- Two events are *independent* if $Pr(\mathcal{A} \cap \mathcal{B}) = Pr(\mathcal{A}) \cdot Pr(\mathcal{B})$.
- A collection of events $\{\mathcal{A}_1, \mathcal{A}_2, \dots, \mathcal{A}_n\}$ is *mutually independent* if

$$Pr(\mathcal{A}_{i_1} \cap \mathcal{A}_{i_2} \cap \dots \cap \mathcal{A}_{i_k}) = Pr(\mathcal{A}_{i_1}) \cdot Pr(\mathcal{A}_{i_2}) \cdot \dots \cdot Pr(\mathcal{A}_{i_k})$$

for any subcollection of events $\{\mathcal{A}_{i_1}, \dots, \mathcal{A}_{i_k}\}$.

Example

Let \mathcal{A} be the event that the roll of a fair die is 6, and let \mathcal{B} be the event that the roll of a second fair die is a 3, and let \mathcal{C} be the event that the sum of the two dice is 10.

Then \mathcal{A} and \mathcal{B} are independent events, but it's easy to check that \mathcal{C} is not independent with either \mathcal{A} or \mathcal{B} .

This follows since $\mathcal{C} = \{11, 12, 13, \dots, 16, 21, \dots, 26, \dots, 61, \dots, 66\}$

$$\begin{aligned} \rightarrow \left\{ \begin{aligned} Pr(\mathcal{A}) &= Pr(\mathcal{B}) = 1/6 \\ Pr(\mathcal{C}) &= 3/36 \end{aligned} \right. \quad \checkmark \end{aligned}$$

$$\mathcal{C} = \{46, 55, 64\}$$

$$Pr(\mathcal{A} \cap \mathcal{C}) = 1/36 = Pr(\mathcal{A} \cap \mathcal{B}), \text{ and}$$

$$Pr(\mathcal{B} \cap \mathcal{C}) = 0.$$

$$\begin{aligned} S_1 &= \{1, 2, 3, 4, 5, 6\} \\ S_2 &= \dots \end{aligned}$$

$$Pr[1] = \dots \quad Pr[\frac{6}{2}] = \frac{1}{6}$$

Conditional Probability

$$\rightarrow \underline{Pr[A \cap B]} = Pr[B] \cdot Pr[A|B]$$

The *conditional probability* of an event \mathcal{A} occurring, given another event \mathcal{B} , denoted $Pr(\mathcal{A}|\mathcal{B})$ is

$Pr(\mathcal{A}|\mathcal{B}) = Pr(\mathcal{A} \cap \mathcal{B}) / Pr(\mathcal{B})$, assuming that $Pr(\mathcal{B}) > 0$.

In other words, conditional probability talks about the probability of some event occurring, *given the knowledge that some other event has already occurred*.

Examples

- ① \mathcal{A} is event that two throws of a fair die sum to 10, \mathcal{B} is event that the first throw is 6.

$$Pr(\mathcal{B}) = 1/6 \text{ and } Pr(\mathcal{A} \cap \mathcal{B}) = 1/36.$$

Thus, $Pr(\mathcal{A}|\mathcal{B}) = (1/36)/(1/6) = 1/6.$

$Pr(\mathcal{A} \cap \mathcal{B}) / Pr(\mathcal{B})$

$$A = \{46, 55, 64\}$$

$$B = \{61, \dots, 66\}$$

$$A \cap B = \{64\}$$

Examples

- ① \mathcal{A} is event that two throws of a fair die sum to 10, \mathcal{B} is event that the first throw is 6.

$$Pr(\mathcal{B}) = 1/6 \text{ and } Pr(\mathcal{A} \cap \mathcal{B}) = 1/36.$$

$$\text{Thus, } Pr(\mathcal{A}|\mathcal{B}) = (1/36)/(1/6) = 1/6.$$

- ② Suppose that you have a red die and a black die.
Let \mathcal{C} be the event that after throwing the two dice they sum to 9, and \mathcal{D} be the event that the red die shows a 5 or 6.

Then (as you can verify, think about it!!!):

$$Pr(\mathcal{D}) = 2/6$$

$$Pr(\mathcal{C} \cap \mathcal{D}) = 2/36$$

Hence,

$$Pr(\mathcal{C}|\mathcal{D}) = Pr(\mathcal{C} \cap \mathcal{D})/Pr(\mathcal{D}) = (2/36)/(2/6) = 1/6.$$

Also, $Pr(\mathcal{C}) = 4/36$. Hence,

$$Pr(\mathcal{D}|\mathcal{C}) = Pr(\mathcal{D} \cap \mathcal{C})/Pr(\mathcal{C}) = (2/36)/(4/36) = 1/2.$$

Random variables

- A *random variable* is a function X that maps outcomes from some sample space to real numbers.

In algorithm analysis we often use a random variable X with a discrete set of outcomes to characterize the running time of a randomized algorithm.

- A *random variable* is a function X that maps outcomes from some sample space to real numbers.

In algorithm analysis we often use a random variable X with a discrete set of outcomes to characterize the running time of a randomized algorithm.

- The *expected value* of a discrete random variable X is defined as $\mathbb{E}(X) = \sum_i i \cdot \text{Pr}(X = i)$ where the summation is over the range of possible values of X .

The expected value is *linear*, meaning that for any two random variables X and Y we have $\mathbb{E}(X + Y) = \mathbb{E}(X) + \mathbb{E}(Y)$.

Example

Consider the event of rolling a fair die, and let X denote the value that occurs. Then we have

$$Pr(X = 1) = Pr(X = 2) = \dots = Pr(X = 6) = 1/6.$$

By definition, we have

$$\mathbb{E}(X) = 1 \cdot \frac{1}{6} + 2 \cdot \frac{1}{6} + \dots + 6 \cdot \frac{1}{6} = 3\frac{1}{2}.$$

$\sum i \cdot Pr[i]$
 i IS AN OUTCOME

Example (cont.)

Suppose that Y is the value obtained by the roll of another fair die. Then $X + Y$ represents the sum of the two dice.

We have

$$\underline{\mathbb{E}(X + Y) = \mathbb{E}(X) + \mathbb{E}(Y) = 3\frac{1}{2} + 3\frac{1}{2} = 7},$$

which can also be computed from the definition of expected value using the facts that

$$Pr(X + Y = 2) = 1/36$$

$$Pr(X + Y = 3) = 2/36$$

...

$$Pr(X + Y = 6) = 5/36$$

$$Pr(X + Y = 7) = 6/36$$

$$Pr(X + Y = 8) = 5/36$$

...

(1,1), ..., (6,6)
2, ..., 26
:
6, ..., 66

Another example

Consider the problem from before of flipping a fair coin until it comes up heads for the first time.

Let X denote the *number* of flips that must be made until the heads appears (including the final flip). Then X is a random variable that takes values in $\{1, 2, 3, \dots\}$.

We note that

$$\Pr(X = i) = \left(\frac{1}{2}\right)^i \text{ for } i \in \{1, 2, 3, \dots\}.$$

By definition, we then have

$$\mathbb{E}(X) = \sum_{i=1}^{\infty} i \cdot \Pr(X = i)$$

Using a bit of calculus (which I don't necessarily expect you to know), we can find that $\mathbb{E}(X) = 2$.

$$S = \underbrace{1}_{\text{H}} + \underbrace{2}_{\text{T H}} + \underbrace{3}_{\text{T T H}} + \dots$$

$$S = 1 \cdot \frac{1}{2} + 2 \cdot \frac{1}{4} + 3 \cdot \frac{1}{8} + \dots$$

$$2S = 1 + 2 \cdot \frac{1}{2} + 3 \cdot \frac{1}{4} + \dots$$

$$S = 1 + \frac{1}{2} + \frac{1}{4} + \frac{1}{8} + \dots$$

Another way to get the same result

We actually don't need to use calculus for this. If you consider the first coin toss, either it comes up heads (in which case the experiment is over), or it doesn't (which “costs” us one coin toss, plus however many more we need until the first heads shows up).

Thus we can write

$$\mathbb{E}(X) = \underbrace{\text{Law of}}_{\text{COND. EXP.}} \underbrace{\text{Pr}(X \text{ is tails})}_{\text{Pr}(X \text{ is tails})} \cdot (1 + \underbrace{\text{expected number of additional coin}}_{\text{tosses until first heads appears}}).$$

But the “expected number of additional coin tosses until the first heads appears” is exactly the same as $\mathbb{E}(X)$. (Having first obtained a “tails” the experiment “resets itself” and we're just interested in obtaining the first heads again.)

Thus we have

$$\mathbb{E}(X) = \underbrace{Pr(X \text{ is heads}) \cdot 1} + \underbrace{Pr(X \text{ is tails}) \cdot (1 + \mathbb{E}(X))}_{\text{G}}$$

Of course,

$$Pr(X \text{ is heads}) = Pr(X \text{ is tails}) = \frac{1}{2}, \text{ so}$$

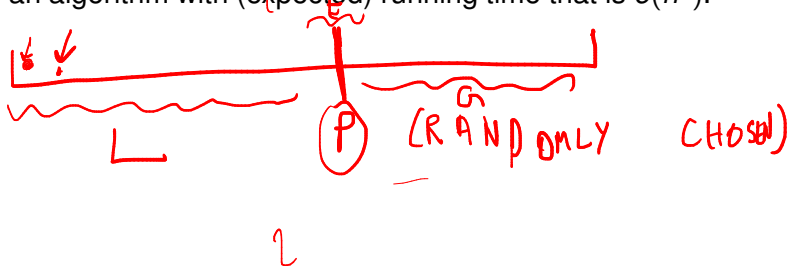
$$\mathbb{E}(X) = \frac{1}{2} \cdot 1 + \frac{1}{2} \cdot (1 + \mathbb{E}(X))$$

which upon solving for $\mathbb{E}(X)$ gives us the same result, namely $\mathbb{E}(X) = 2$.

Randomized QuickSort (cont.)

Recall that our proposed method for the randomized QuickSort algorithm is to choose the pivot element at random.

We can repeat the selection until we find a “good” pivot element (defined below). With a good pivot element during each step, we have an algorithm with (expected) running time that is $O(n^2)$.



Randomized QuickSort (cont.)

Recall that our proposed method for the randomized QuickSort algorithm is to choose the pivot element at random.

We can repeat the selection until we find a “good” pivot element (defined below). With a good pivot element during each step, we have an algorithm with (expected) running time that is $o(n^2)$.

Theorem The expected running time of randomized QuickSort (pivot chosen at random) is $O(n \log n)$.

Randomized QuickSort (cont.)

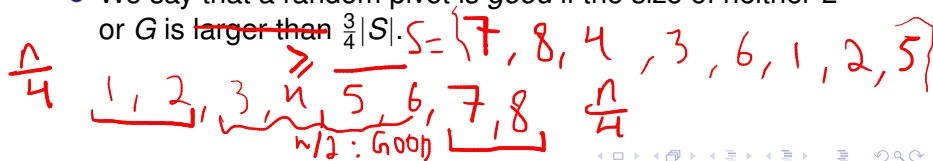
Recall that our proposed method for the randomized QuickSort algorithm is to choose the pivot element at random.

We can repeat the selection until we find a “good” pivot element (defined below). With a good pivot element during each step, we have an algorithm with (expected) running time that is $O(n^2)$.

Theorem The expected running time of randomized QuickSort (pivot chosen at random) is $O(n \log n)$.

Proof idea:

- We say that a random pivot is *good* if the size of neither L or G is larger than $\frac{3}{4}|S|$.



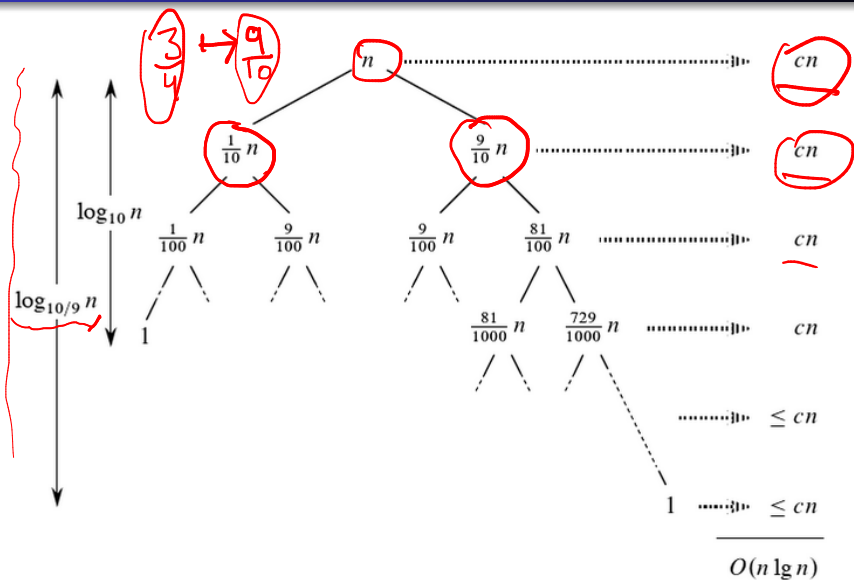
Randomized QuickSort (cont.)

$$|L| \leq \frac{3}{4}n$$

$$|R| \leq \frac{3}{4}n$$

- The probability of success of choosing a good pivot is $\frac{1}{2}$.
- Fact: The *expected* number of times that a *fair* coin must be tossed until it shows heads k times is $2k$. Choosing a good pivot is analogous to flipping a fair coin until it comes up heads. So, on average we need two attempts (random choices) to find a good pivot.
- Any path in the QuickSort algorithm can contain at most $\log_{4/3} n$ levels with good pivots.
- Hence, the expected length of each path will be $2 \log_{4/3} n$.

Randomized QuickSort



Linear-time algorithms

We will now see some examples of linear-time ($O(n)$) sorting algorithms.

Linear-time algorithms

We will now see some examples of linear-time ($O(n)$) sorting algorithms?

(Didn't we see an $\Omega(n \log n)$ lower bound earlier?)

Linear-time algorithms

We will now see some examples of linear-time ($O(n)$) sorting algorithms?

(Didn't we see an $\Omega(n \log n)$ lower bound earlier?)

The algorithms that we'll see will have the following 2 properties:

- They aren't comparison based, and
- They require a promise that the elements in the input aren't too large.

Counting sort



MAX VALUE $\leq k$,
ALL the INTEGERS
COST: $O(n + k)$

Counting sort assumes that each of the n input elements is an integer between 0 and k . When $k = O(n)$, counting sort runs in $O(n)$ time.

Input array is $A[1..n]$. Our output will be in $B[1..n]$, and we'll use $C[0..k]$ as temporary working storage.

Counting sort (cont.)

In one pass over A , fill C such that $C[k]$ contains the number of occurrences of k in A .

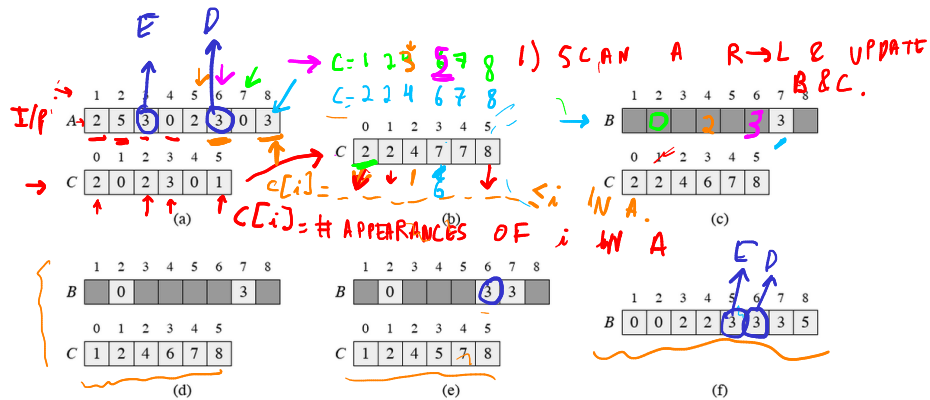
In another pass over C , update C such that $C[k]$ contains the number of elements *less than or equal to* k in A .

From the last to the first element of A , insert this element in the appropriate location in B and update C accordingly.

COUNTING-SORT(A, B, k)

```
1  let  $C[0..k]$  be a new array
2  for  $i = 0$  to  $k$ 
3       $C[i] = 0$ 
4  for  $j = 1$  to  $A.length$ 
5       $C[A[j]] = C[A[j]] + 1$ 
6  //  $C[i]$  now contains the number of elements equal to  $i$ .
7  for  $i = 1$  to  $k$ 
8       $C[i] = C[i] + C[i - 1]$ 
9  //  $C[i]$  now contains the number of elements less than or equal to  $i$ .
10 for  $j = A.length$  downto 1
11      $B[C[A[j]]] = A[j]$ 
12      $C[A[j]] = C[A[j]] - 1$ 
```

Example of counting sort



Loop invariant: before every iteration in the last **for** loop, $C[i]$ contains the list of numbers remaining in A that are at most i , for all $i \in \{0, 1, \dots, k\}$. Thus, B is updated correctly (why are there no collisions in B ?), so is C , and thus the loop invariant is maintained.

Counting sort is **stable**: numbers with the same value appear in the same order in the output array as they appear in the input array.

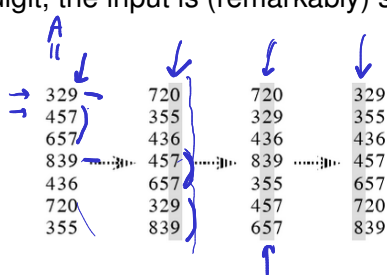
Stability is a practically-useful concept because there may be big chunks of data also associated with each number, and we may want the data's order to always be preserved.

More importantly for us, in order for radix sort (coming up) to work correctly, counting sort must be stable.

Radix sort

Used to be used by card-sorting machines that sorted punch cards containing data.

Counter-intuitive: First sort by the *least significant* digit. Next sort by second-least significant digit, and so on. After running through each digit, the input is (remarkably) sorted.



EACH OF
THE "DIGIT-SORT"
MUST USE
STABLE
ALG.

Analysis of radix sort

COUNTING SORT
R.TIME $O(n+k)$ \uparrow MAX VAL.

Running time: $O(n)$ per digit. Thus, if all inputs are d digits, radix sort runs in time $O(dn)$.

Correctness: Can be shown by induction (on which digit the algorithm looks at). Crucially requires each column to be sorted using a stable sort.

