Randomized Algorithms:

Randomized Quicksort

Fatemeh Panahi
Department of Computer Science
University of Toronto

CSC263-Fall 2017
Lecture 6

Announcements

- Midterm next week,
  - LEC0101, LEC2003: Fri, Oct 27, 10:00-11:00, EX 300
  - LEC0201, LEC2000, LEC2201: Fri, Oct 27, 13:00-14:00, SS 1083, SS 1085, SS 1087.

- Tutorial: New tutorial TA, Sasa Milic,
  - She will discuss the quiz and some examples on expected running time.

Today

- Quick sort
- Randomized Quick sort
- Randomized algorithm in general
- Expected running time for randomized algorithms

Reading Assignments 

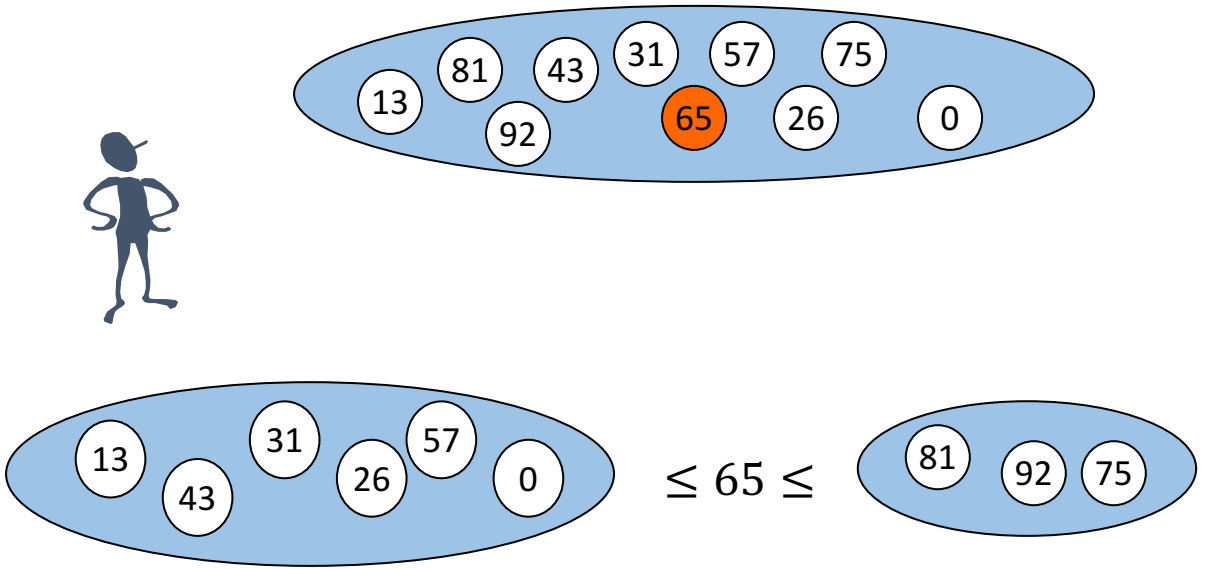
Chapter 7, 5.1, 5.2, 5.3

StQocikru

Very commonly used sorting algorithm. When implemented well, can be about 2-3 times faster than merge sort and heapsort.

Quick sort – idea

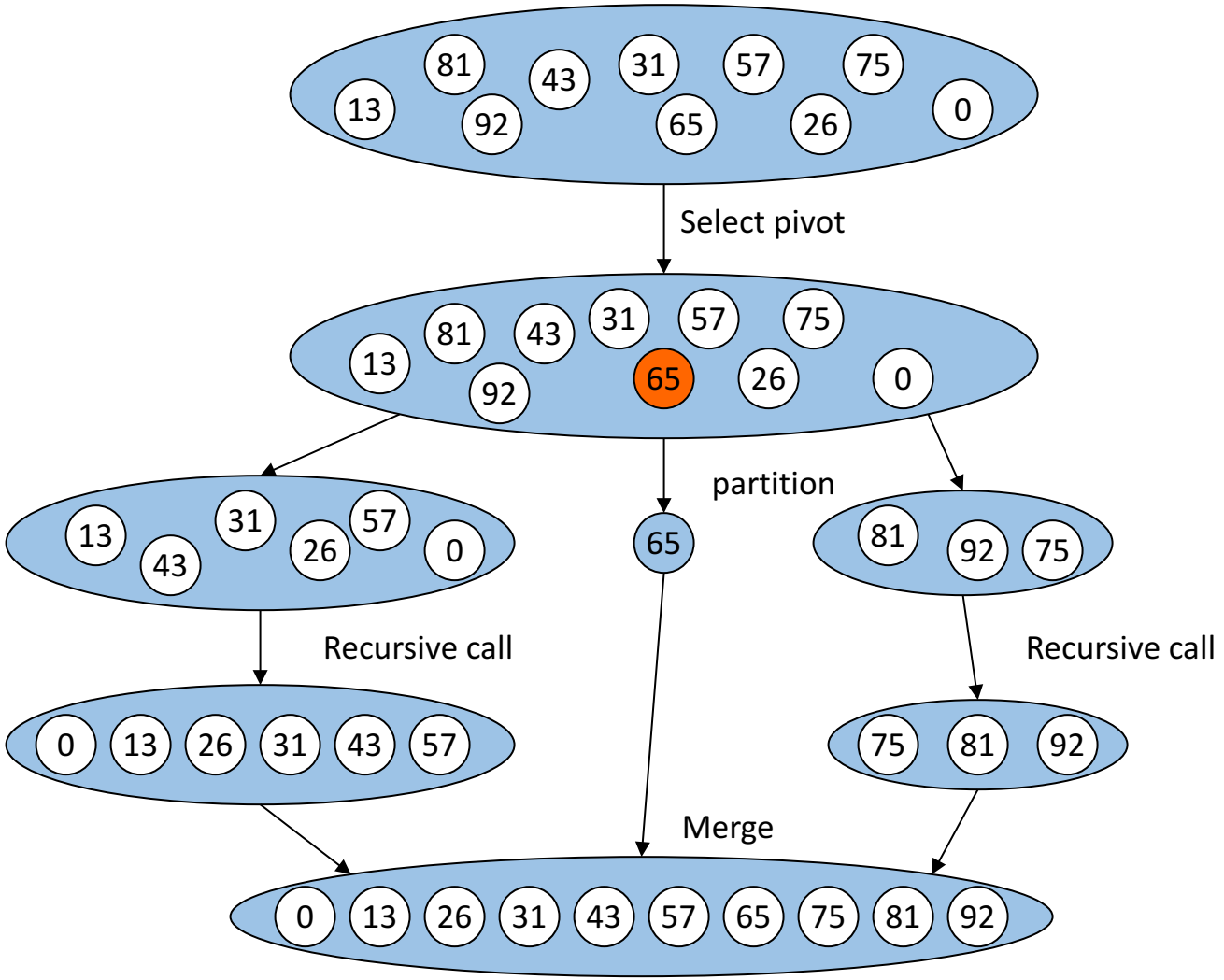
Partition set into two using a chosen pivot



sort the first partition.

sort the second part.

Quick sort – divide and conquer



Quick sort - main idea

- *Divide*: $A[p \ldots r]$ is partitioned (rearranged) into two nonempty subarrays $A[p \ldots q-1]$ and $A[q+1 \ldots r]$ s.t. each element of $A[p \ldots q-1]$ is less than or equal to each element of $A[q+1 \ldots r]$. Index q is computed here, called **pivot**.

- *Conquer*: two subarrays are sorted by recursive calls to quicksort.

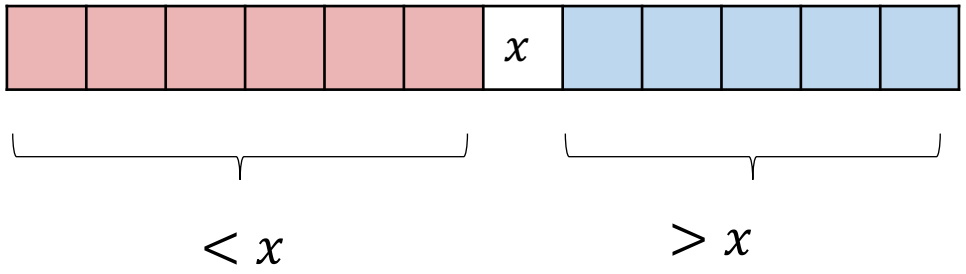
- *Combine*: unlike merge sort, no work needed since the subarrays are sorted in place already.

Quick sort - main idea

For simplicity, assume all elements are **distinct**

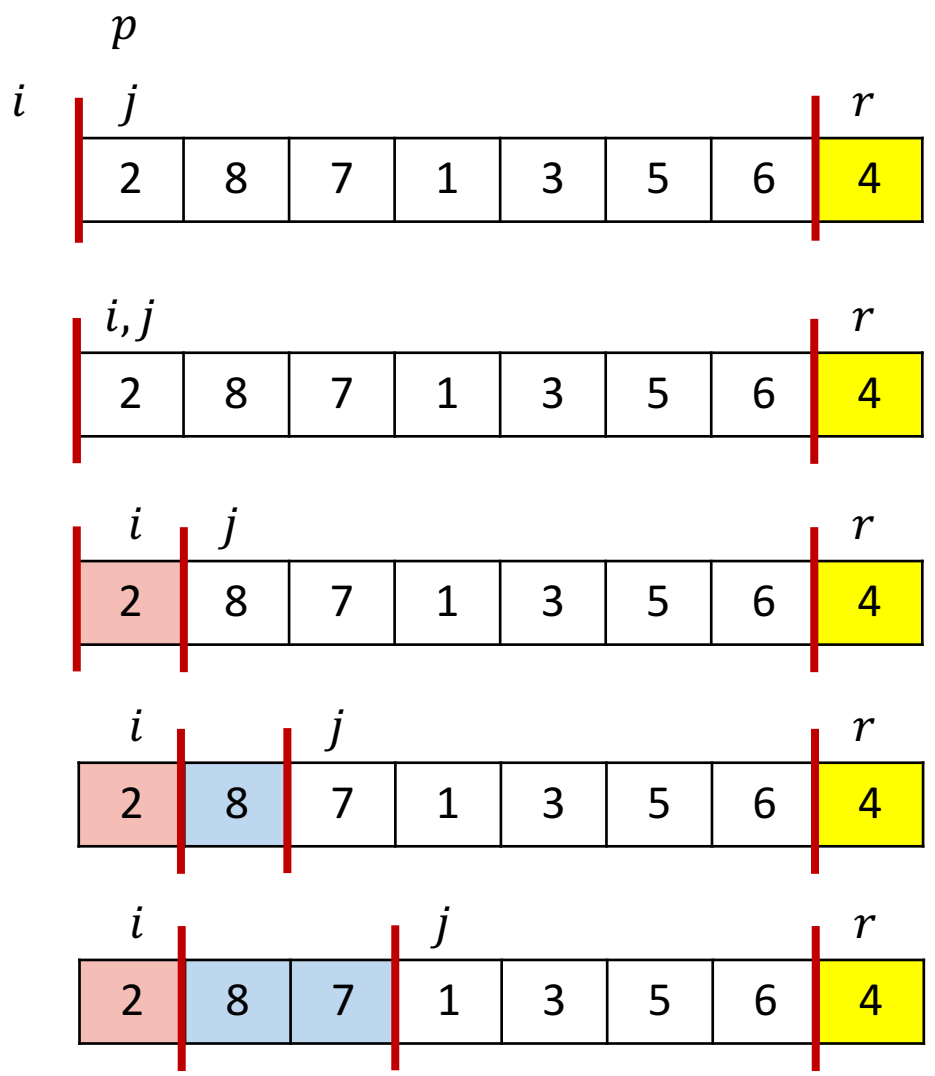
The basic algorithm to sort an array A consists of the following four easy steps:

- If the number of elements in A is 0 or 1, then return
- Pick any element $\boldsymbol{x}$ in A. This is called the pivot
- Partition $A - \{\boldsymbol{v}\}$ (the remaining elements in A) into two disjoint groups:
  - $L = \{\boldsymbol{a} \in A - \{\boldsymbol{x}\} | \boldsymbol{a} < x\}$, and
  - $E = \{\boldsymbol{a} \in A - \{\boldsymbol{x}\} | \boldsymbol{a} = \boldsymbol{x}\}$
  - $G = \{\boldsymbol{a} \in A - \{\boldsymbol{x}\} | \boldsymbol{a} > \boldsymbol{x}\}$
- return {quicksort(A_1) followed by A_2 followed by quicksort(A_3)}



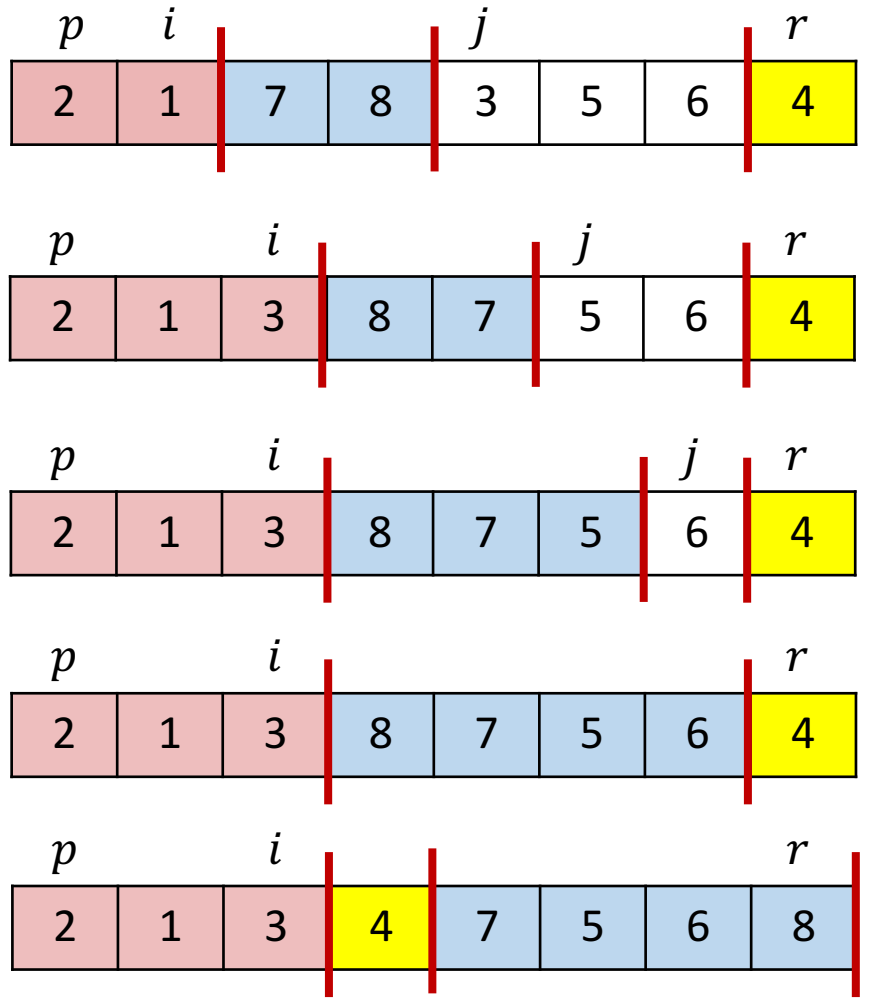
Partition – pseudocode

```
Partition(A, p, r)
{
   x = A[r]          // x the last element is pivot
   i = p − 1
   for j = p to r − 1
   {
     do if A[j] ≤ x
          then {
                    i = i + 1
                    exchange A[i] ↔ A[j]
                  }
   }
   exchange A[i + 1] ↔ A[r]
   return i + 1
}
```



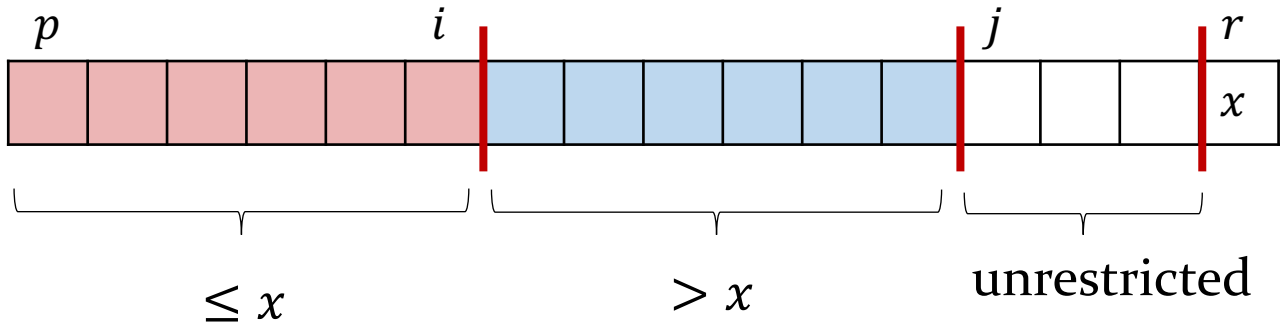
Partition – pseudocode

```
Partition(A, p, r)
{
   x = A[r]                    // x the last element is pivot
   i = p − 1
    for j = p to r − 1
    {
      do if A[j] ≤ x
           then {
                    i = i + 1
                    exchange A[i] ↔ A[j]
                  }
    }
   exchange A[i + 1] ↔ A[r]
   return i + 1
}
```



Quick sort - partition

- Clearly, all the action takes place in the **partition**() function
  - Rearranges the subarray in place
  - End result:
    - Two subarrays
    - All values in first subarray ≤ all values in second
  - Returns the **index** of the "pivot" element separating the two subarrays



Quick sort - pseudocode

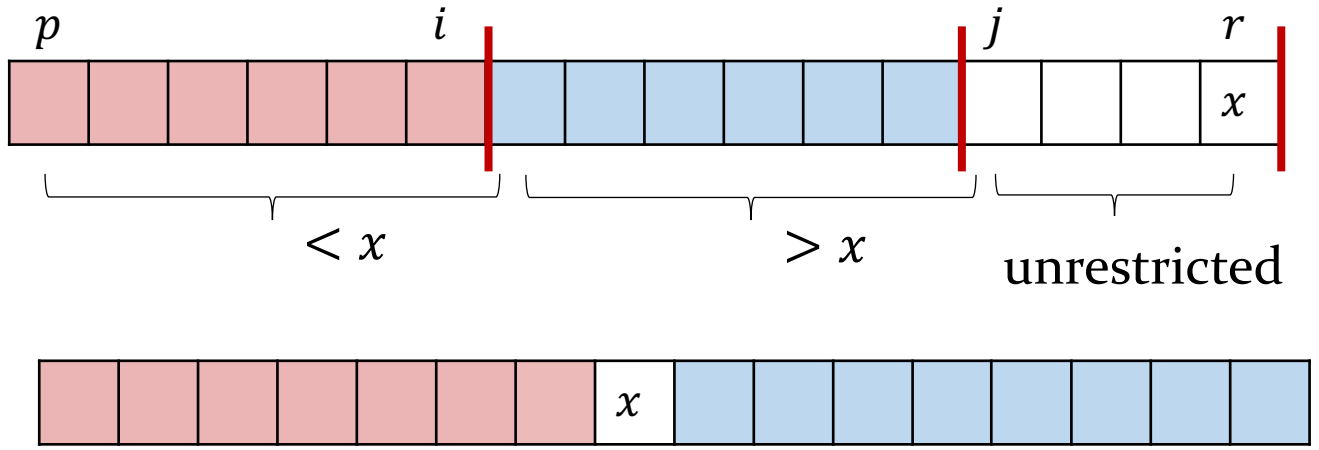
```
Quicksort(A, p, r)
{
    if (p < r)
    {
        q = Partition(A, p, r)
        Quicksort(A, p, q − 1)
        Quicksort(A, q + 1, r)
    }
}
```

- Initial call is **Quicksort**$(A, 1, n)$, where n in the length of A

Quick Sort
Running time Analysis

Proposition

Proposition: $T(n)$ total number of comparisons is no more than the total number of pairs.



Claim 1. Each element in A can be chosen as pivot at most once.

Claim 2. Elements are only compared to pivots.

Proposition – Cont'd

Claim 3.

- Any pair (a, b) in A, they are compared with each other at most once.
  - The only possible one happens when a or b is chosen as a pivot and the other is compared to it;
  - after being the pivot, the pivot one will be out of the market and never compare with anyone anymore.
  - So, the total number of comparisons is no more than the total number of pairs.

Worst case – upper bound

So, the total number of comparisons is no more than the total number of pairs.

$$T(n) \leq \binom{n}{2} = \frac{n(n-1)}{2}$$

$$T(n) \in O(n^2)$$

Next show $T(n) \in \Omega(n^2)$

Worst case – lower bound

- We want to show $T(n) \in \Omega(n^2)$

  How do you show the tallest person in the room is at least 1 meter?
  It is enough to show one person who is taller than 1 meter.

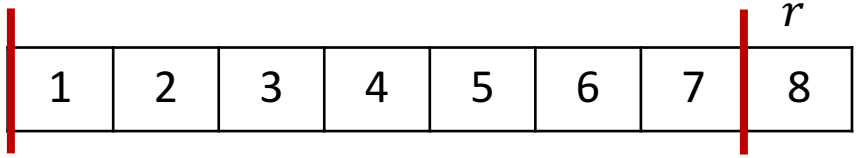
- So, just find one input for which the running time is some cn^2.

  For two partitions with size of n_1 and n_2 :
  $T(n) = T(n_1) + T(n_2) + \Theta(n)$

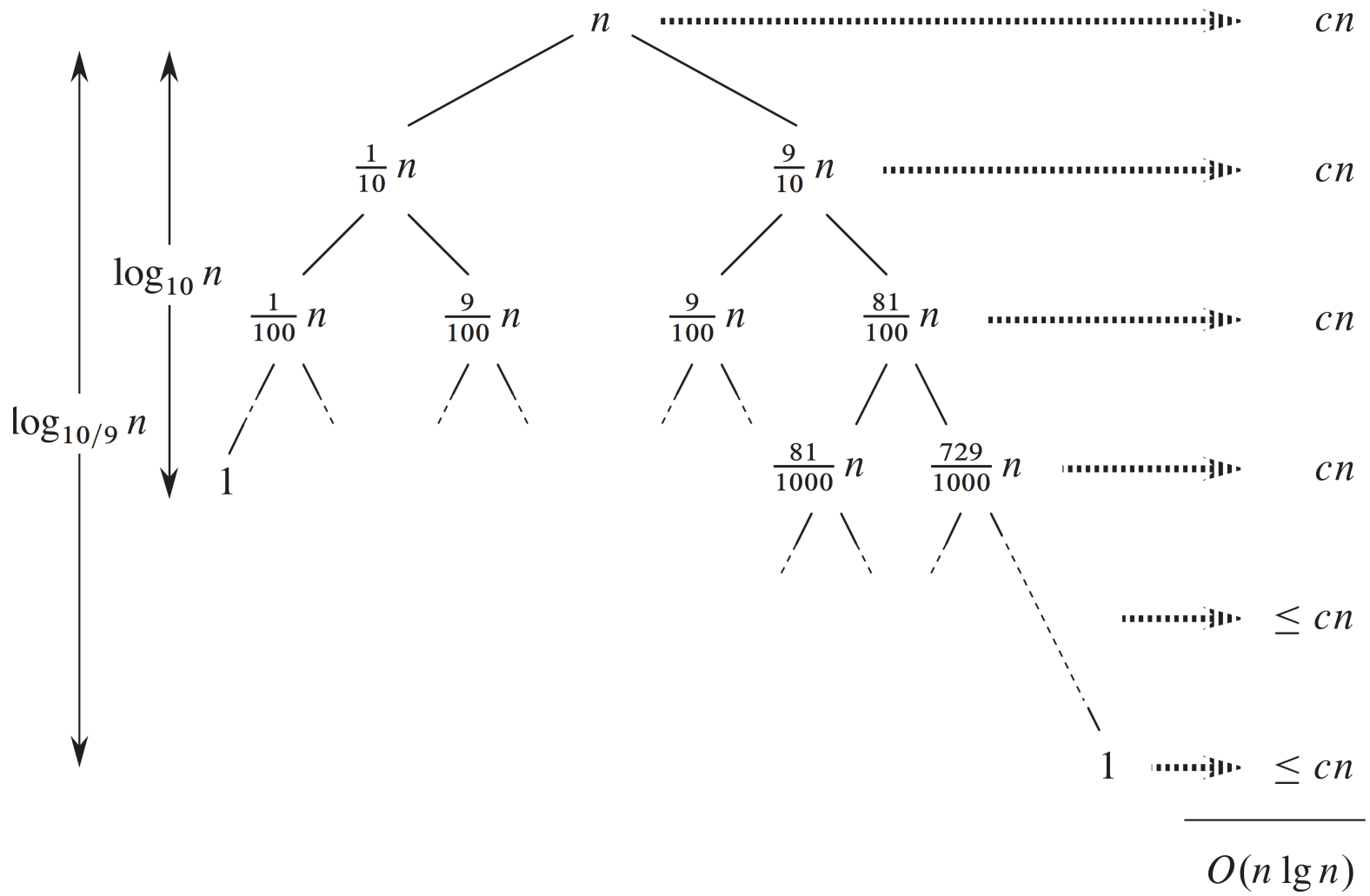
  Best case partition: $T(n) = 2T\left(\frac{n}{2}\right) + n - 1$

  Worst case partition: Find one input that results in awful partitions.
  $T(n) = T(n-1) + n - 1$

Remember that we always pick the last one as pivot.



Worst case – good partition



Worst case – lower bound

For input $[n,\ n-1,\ n-2,\ \ldots, 2, 1]$

$T(1) = 0$
$T(n) = T(n-1) + n - 1$
$T(n) = T(n-2) + (n-2) + n - 1$
$T(n) = 0 + 1 + 2 + \cdots + (n-2) + n - 1$

$$\rightarrow T(n) = \frac{n(n-1)}{2} = \Omega(n^2)$$

$$\rightarrow T(n) = \Theta(n^2)$$

Is QuickSort really "quick" ?

Yes, in average case!

in practice

Average case

- Sample space $S_n = \{$ all permutations of $[1, 2, \ldots, n] \}$, with uniform probability distribution (not necessarily a good assumption in general).

- We assume that there is no repeated element.

Q: Why is this reasonable?

A: Running time can only decrease with repeated values (they get put into E and never examined again), so assumption does not underestimate complexity for more general sample spaces.

- More general sample spaces tricky to define clearly and precisely.

Average case

Let X be the **random variable** representing the number of comparisons.

We want to compute $E[X]$.

We define an indicator random variable:

$$X_{ij} = \begin{cases} 1 & if\ value\ i\ and\ j\ are\ compared \\ 0 & otherwise \end{cases}$$

Only Xij

So the total number of comparisons:

$$X = \sum_{i=1}^{n} \sum_{j=i+1}^{n} X_{ij}$$

sum over all possible pairs

Average case

$$X = \sum_{i=1}^{n} \sum_{j=i+1}^{n} X_{ij} \rightarrow \quad E(X) = E\left[\sum_{i=1}^{n} \sum_{j=i+1}^{n} X_{ij}\right]$$

$$= \sum_{i=1}^{n} \sum_{j=i+1}^{n} E(X_{ij})$$

$$= \sum_{i=1}^{n} \sum_{j=i+1}^{n} E(X_{ij})$$

$$= \sum_{i=1}^{n} \sum_{j=i+1}^{n} \Pr(i \text{ and } j \text{ are compared})$$

Just need to figure this out!

Average case

$\Pr(i \text{ and } j \text{ are compared})$

no necessary ?

Think about the sorted sub-sequence $Q_{ij} : i, i+1, \ldots, j$

Claim: i and j are compared if and only if, among all elements in Q_{ij}, the first element to be picked as a pivot is either i or j.

Proof:

- $\Longrightarrow$ suppose the first one picked as pivot as some k that is between i and j,… then i and j will be separated into different partitions and will never meet each other.

- $\Longleftarrow$ if i is chosen as pivot (the first one among Q_{ij}), then j will be compared to pivot i for sure, because nobody could have possibly separated them yet! Similar argument for first choosing j

Average case

We proved: i and j are compared if and only if, among all elements in Q_{ij}, the first element to be picked as a pivot is either i or j.

$$\Pr(i \text{ and } j \text{ are compared}) = \Pr(i \text{ or } j \text{ is the first among } Q_{ij} \text{ chosin as pivot})$$

$$= \frac{2}{j-i+1}$$

$$\sum_{i=1}^{n} \sum_{j=i+1}^{n} \Pr(i \text{ and } j \text{ are compared})$$

$$= \sum_{i=1}^{n} \sum_{j=i+1}^{n} \frac{2}{j-i+1}$$

$$\leq \mathrm{n} \sum_{k=1}^{n} \frac{1}{k} \in O(n \log n)$$

There are $j - i + 1$ numbers in Q_{ij}, and each of them is equally likely to be chosen as the first pivot.

How to avoid the worst case?

The average-case runtime ($E[X]$) of QuickSort is $O(n \log n)$.
The worst-case runtime was $\Theta(n^2)$.

The assumption of uniform randomness is NOT really true, because it is often impossible for us to know what the input distribution really is.

Ever worse, if the person who provides the inputs is malicious, they can totally only provide worst-inputs and guarantees worst-case runtime.

How do we make sure to get average-case and avoid worst-case?

We do Randomization.

How can we get some guaranteed performance in real life?

- We shuffle the input array “uniformly randomly”, so that after shuffling the arrays look like drawn from a uniform distribution
- Even the malicious person’s always-worst inputs will be shuffled to be like uniformly distributed. This makes the assumption in the average-case analysis true
- So we are guaranteed the O(n log n) expected runtime

Average case

Randomize- QuickSort(A):
    permute A uniformly randomly
    QuickSort(A)

How exactly do we perform the permutation so that we can prove that it's going to be like uniform distribution? (Will be discussed in tutorial)

Is it the same that we choose the pivot randomly?

Quick sort – pros and cons

- Quicksort advantages:
  - Sorts in place
  - Sorts $O(n \lg n)$ in the average case
  - Very efficient in practice

- Quicksort disadvantages :
  - Sorts $O(n^2)$ in the worst case
  - not stable
    - Does not preserve the relative order of elements with equal keys
    - Sorting algorithm (stable) if 2 records with same key stay in original order

In practice, it's quick and the worst case doesn't happen often.

Randomized Algorithm

Randomization

Use randomization to guarantee expected performance

We do it everyday.

Randomized algorithm- Running time

We take the expected running time over the distribution of values returned by the random number generator. Recall that so far we had talked about three types of running time (best case, average case and worst case)

```
Randomized- Algorithm(A):
        . . .
         make random variables X
        . . .
        End
```

Randomized algorithm- Running time

- We discuss the average-case running time when the probability distribution is over the inputs to the algorithm,
- We discuss the expected running time when the algorithm itself makes random choices.

It means that we have also best case expected running time or worst case expected running time for randomized algorithms. Example:

```
Randomized- Algorithm(A):
        . . .
        Choose a random variable r ∈ {1, 2, ... n}
        . . .
        End
```

Worst case expected running time:

$\sum_{i=1}^{n} \frac{1}{n} X_i$ where X_i is the worst case running time for algorithm when $r = i$

Two types of randomized algorithms

- "Las Vegas" algorithm
  - Deterministic answer, random runtime

    always correct, often fast
- "Monte Carlo" algorithm
  - Deterministic runtime, random answer

    always fast, often correct

Randomized-QuickSort is a …

Las Vegas algorithm

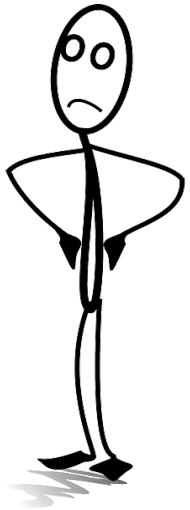
Monte Carlo Algorithm - Example

The problem: "Equality Testing"

Given two binary numbers x and y, decide whether x = y.

How many bits needs to be exchanged between A and B? $\Theta(n)$

Are you kidding me?



No kidding, what if the size of x and y are 10TB each?
The above code needs to compare $\sim 10^{14} \leq 2^{50}$ bits.
Can we do better?

Does it give the correct answer?

Idea: If (x mod p) ≠ (y mod p), then…
Must be x ≠ y, our answer is correct for sure.

If $(x \ mod \ p) \ = \ (y \ mod \ p)$, then…
Could be $x \ = \ y \ or \ x \ \neq \ y$, so our answer might be correct.

Correct with what probability?
What's the probability of a wrong answer?

The number of required bits

$p = \{2, \ldots n^2\}$ uniformly random
$|p| \leq \log(n^2) = 2 \log n$
$x \; mod$ p $\leq p$

Number of bits for p and $x \; mod \; p \; = \; 4 \log n \leq 200 \; bits$
n $= 10$ TB $\leq 2^{50}$

Prime number theorem

- In range $[1, m]$, there are roughly $m/\ln(m)$ prime numbers.
- So in range $[1, n^2]$, there are $n^2/\ln(n^2) = n^2/2\ln(n)$ prime numbers.

How many (bad) primes in $[1, n^2]$ satisfy $(x \bmod p) = (y \bmod p)$ even if $x \neq y$? At most n

why?

$(x \bmod p) = (y \bmod p) \Leftrightarrow |x - y|$ is a multiple of p, i.e.,
p is a divisor of $|x - y|$.
$|x - y| < 2^n$ (n-bit binary #) so it has no more than n prime divisors (otherwise it will be larger than 2^n).

Prime number theorem

So…
Out of the $n^2/2\ln(n)$ prime numbers we choose from, at most n of them are bad.
If we choose a good prime, the algorithm gives correct answer for sure.
If we choose a bad prime, the algorithm may give a wrong answer.
So the prob of wrong answer is less than

$$\frac{n}{\frac{n^2}{2\ln n}} = \frac{2\ln n}{n}$$

Error probability of our Monte Carlo algorithm

$$\Pr(error) \leq \frac{2\ln n}{n}$$

When $n = 10^{14}$ $(10TB)$

$$\Pr(error) \leq 0.000000000000644$$

Prime number theorem

Performance comparison ($n = 10TB$)

The regular algorithm $x == y$

- Perform 10^{14} comparisons
- Error probability: 0

The Monte Carlo algorithm $(x \ mod \ p) == (y \ mod \ p)$

- Perform $<$ 100 comparisons
- Error probability: 0.000000000000644

If your boss says: "This error probability is too high!"

Run it twice: Perform $<$ 200 comparisons

- Error prob squared: 0.000000000000000000000000215

Summary

Randomized algorithms

- Guarantees expected performance
- Make algorithm less vulnerable to malicious inputs

Monte Carlo algorithms

- Gain time efficiency by sacrificing some correctness.

Questions