RANDOMIZED ALGORITHMS

A short list of categories

- Algorithm types we will consider include:
  - Simple recursive algorithms
  - Backtracking algorithms
  - Divide and conquer algorithms
  - Dynamic programming algorithms
  - Greedy algorithms
  - Branch and bound algorithms
  - Brute force algorithms
  - Randomized algorithms
    - Also known as Monte Carlo algorithms or stochastic methods

Randomized algorithms

- A randomized algorithm is just one that depends on random numbers for its operation

- These are randomized algorithms:
  - Using random numbers to help find a solution to a problem
  - Using random numbers to improve a solution to a problem

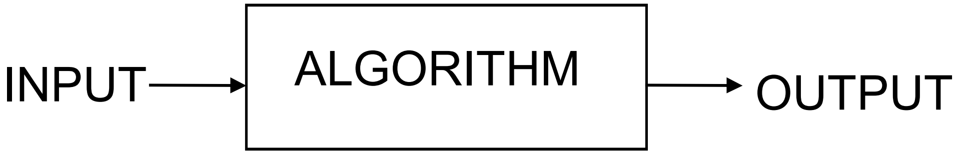
Pseudorandom numbers

- The computer is *not capable* of generating truly random numbers
  - The computer can only generate pseudorandom numbers--numbers that are generated by a formula
  - Pseudorandom numbers *look* random, but are perfectly predictable if you know the formula
    - Pseudorandom numbers are good enough for most purposes, but not all--for example, not for serious security applications
  - Devices for generating truly random numbers do exist
    - They are based on radioactive decay

Generating random numbers

- Perhaps the best way of generating “random” numbers is by the linear congruential method:
  - r = (a * r + b) % m;
    where a and b are large prime numbers, and m is 2^{32} or 2^{64}
  - The initial value of r is called the seed
  - If you start over with the same seed, you get the same sequence of “random” numbers
- One advantage of the linear congruential method is that it will (eventually) cycle through all possible numbers

Deterministic Algorithms



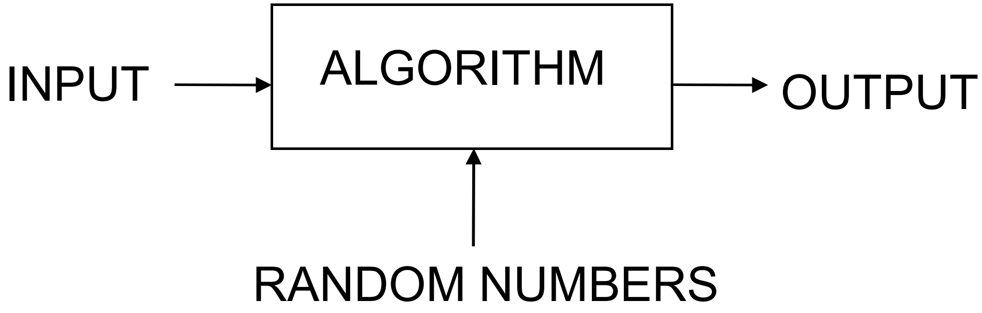
INPUT

ALGORITHM

OUTPUT

Goal: Prove for all input instances the algorithm solves the problem correctly and the number of steps is bounded by a polynomial in the size of the input.

Randomized Algorithms



- In addition to input, algorithm takes a source of random numbers and makes random choices during execution;
- Behavior can vary even on a fixed input;
- A randomized algorithm is defined as an algorithm that is allowed to access a source of independent, unbiased random bits, and it is then allowed to use these random bits to influence its computation

Randomized algorithm

- A randomized algorithm can be defined as one that receives, in addition to its input, a stream of random bits that it can use in the course of its action for the purpose of making random choices.

- An algorithm is randomized algorithm if its behavior is determined not only by its input but also by values produced by a random-number generator.

- A randomized algorithm may give different results when applied to the same input in different runs.

- Deterministic Algorithm: Identical behavior for different runs for a given input.

- Randomized Algorithm : Behavior is generally different for different runs for a given input

Thoughts

- For some problems, it is better to randomly choose, instead of taking the time to take the best choice.
- Factors:
  - How much difference are between random choices and the best choice
  - How long does it take to calculate the best choice
- Probabilistic algorithms might return different answers on the same problem instance

Randomized algorithms

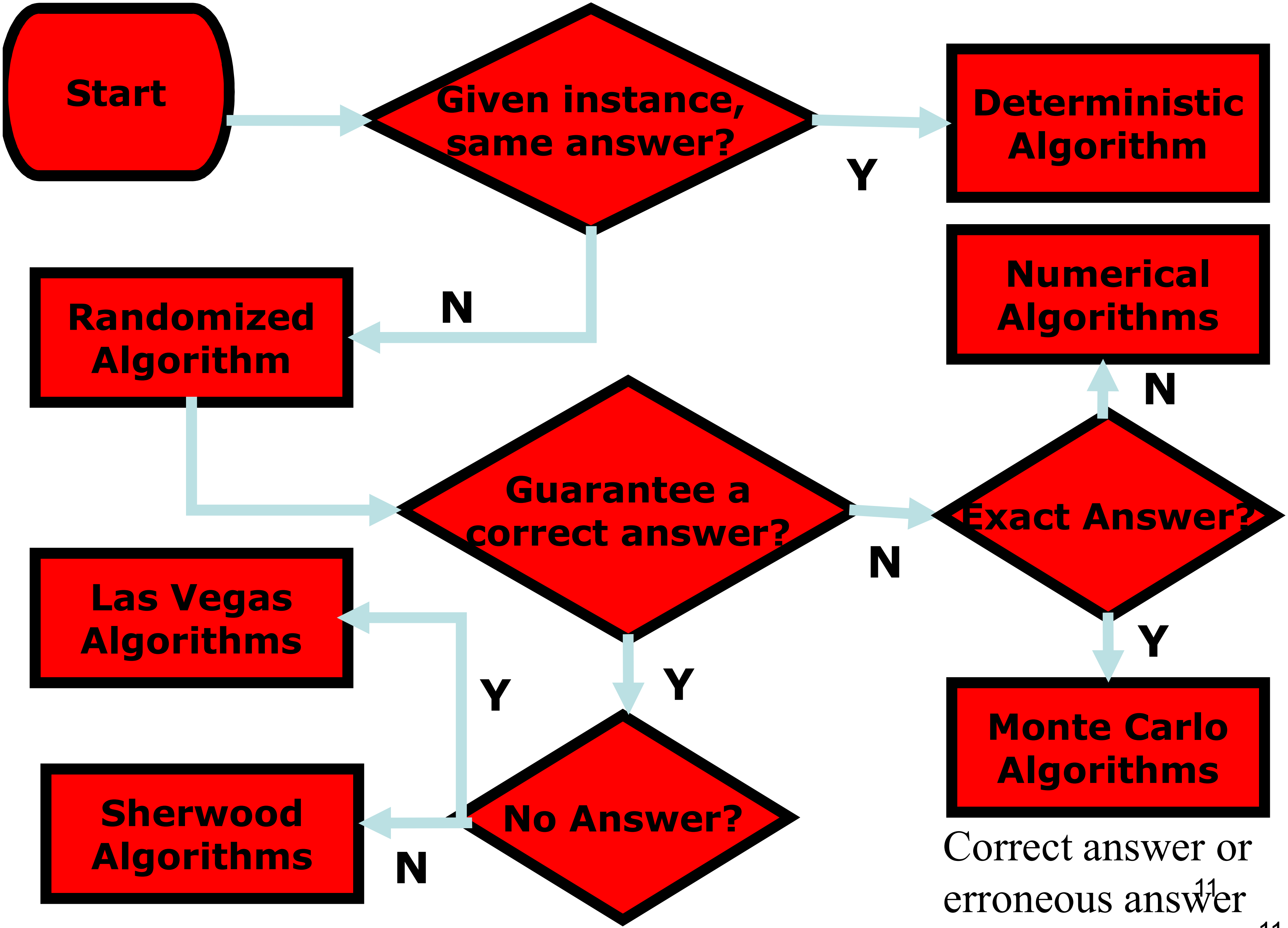
- Main characteristic:

  The same algorithm may behave differently when it is applied twice to the same instance

  Execution time and even the result obtained may vary considerably from one use to the next

Randomized algorithms fall into four main design categories:

1. Numerical probabilistic algorithms
2. Randomizations of deterministic algorithms (Sherwood algorithms)
3. Las Vegas algorithms
4. Monte Carlo algorithms



Start

Given instance, same answer?

Deterministic Algorithm

Randomized Algorithm

Numerical Algorithms

Guarantee a correct answer?

Exact Answer?

Las Vegas Algorithms

Monte Carlo Algorithms

No Answer?

Sherwood Algorithms

Correct answer or erroneous answer

Facetious Illustration

- How a randomized algorithm would answer the question “When did the WWWII begin?”
- Numerical algorithm (5 calls):

  {1940 ± 2; 1939 ± 2; 1937 ± 2;1915 ± 2; 1942 ± 2} in a confidence interval with probability *p*%
- Monte Carlo algorithm (10 calls):

  {1939; 1939; 1939; 1941;1939; 1939; 1939; 1939; 56BC; 1939} ← is correct with probability *p*%
- Las Vegas algorithm (10 calls): {1939; 1939; Fail!;1939;1939; 1939; 1939; 1939; Fail !; 1939} ← may fail with probability ε%
- Sherwood algorithm (10 calls): {1939; 1939; 1939;1939;1939; 1939; 1939; 1939; 1939; 1939} ← success 100%

For example

- **The hiring problem**

  Suppose that you need to hire a new office assistant. Your previous attempts at hiring have been unsuccessful, you are committed to having, at all times, the best possible person for the job. Therefore, you decide that, after interviewing each applicant, if that applicant is better qualified than the current office assistant, you will fire the current office assistant and hire the new applicant. You are willing to pay the resulting price of this strategy, but you wish to estimate what that price will be.

```
HireAssistant($n$)
$best \leftarrow 0$   // candidate 0 is a least-qualified dummy candidate
**for** $i \leftarrow 1$ to $n$ **do**
    interview candidate $i$
    **if** candidate $i$ is better than candidate $best$ **then**
        $best \leftarrow i$
        hire candidate $i$.
```

- Interviewing has a low cost, say c_i, whereas hiring is expensive, costing c_h. Let m be the number of people hired. Then the total cost associated with this algorithm is $O(nc_i + mc_h)$.
- Worst-case /Best-case analysis

Randomized hire algorithms

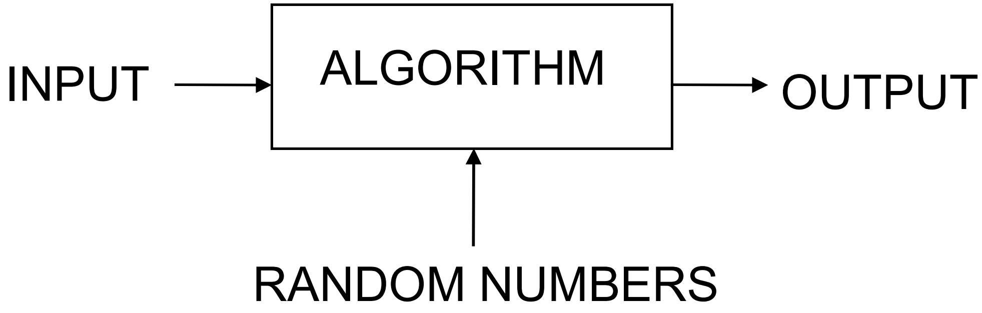
```
RandomizedHireAssistant($n$)
randomly permute the list of candidates
$best \leftarrow 0$  //candidate 0 is a least-qualified dummy candidate
for $i \leftarrow 1$ to $n$ do
    interview candidate $i$
    if candidate $i$ is better than candidate $best$ then
        $best \leftarrow i$
        hire candidate $i$
```

Monte Carlo and Las Vegas

There are two kinds of randomized algorithms:

- Monte Carlo:
  - A Monte Carlo algorithm runs for a fixed number of steps for each input and produces an answer that is correct with a bounded probability.
  - **These algorithms always give an answer, but may occasionally produce an answer that is incorrect.**
- Las Vegas:
  - A Las Vegas algorithm always produces the correct answer, but its runtime for each input is a random variable whose expectation is bounded.
  - **constitute those randomized algorithms that always give a correct solution, or do not have any correct solution at all**

Las Vegas Randomized Algorithms



Goal: Prove that for all input instances the algorithm solves the problem correctly and the expected number of steps is bounded by a polynomial in the input size.

Note: The expectation is over the random choices made by the algorithm**.**

Las Vegas

- Always gives the true answer.
- Sometimes will return no answer at all.
- Running time is random.
- Running time is bounded.
- Quick sort is a Las Vegas algorithm.

- The longer these algorithm run, the higher their probability of success .

- It will randomly make decision and then check to see if they are resulted in successful answer.
- Program that uses this alg would repeatedly calling until the algorithm initialized success.

Monte Carlo

- It may produce incorrect answer!
- We are able to bound its probability.
- By running it many times on independent random variables, we can make the failure probability arbitrarily small *at the expense of running time*.
- **2 ways to improve the results**
  - To increase the amount of time it runs
  - To call it multiple times
    - Only be used with algorithm is consistent.
- Make several calls to it and choose the answer that appears most frequently.

Advantages of Randomized Algorithm

❑ There are two main advantages of randomized algorithms:

- First, often the execution time or space requirement is smaller than that of the best deterministic algorithm.

- Second, randomized algorithms are extremely simple to comprehend and implement.

$\Theta(n\log n)$

Randomized Quicksort

- The running time of quicksort is $\Theta(n\log n)$ on the average, provided that all permutations of the input elements are equally likely.

This is not the case in many $\Theta(n^2)$ applications.

If the input is already sorted, then its running time is $\Theta(n^2)$

$\Theta(n\log n)$

Randomized Quicksort

❑ One approach to guarantee a running time of $\Theta(n\log n)$ on the average is to introduce a preprocessing step with purpose to permute the elements to be sorted randomly.

This step can be performed in $\Theta(n)$ time.

❑ Another approach is to introduce an element of randomness into the algorithm.

This can be done by selecting the pivot on which to split the elements randomly.

The result of choosing the pivot randomly is to relax the assumption that all permutations of the input elements are equally likely.

Computational Complexity of A Randomized Algorithm

- If *A* is a randomized algorithm, then its running time on a fixed instance I of size n may vary from one execution to another.

- Therefore, a more natural measure of performance is the expected running time of *A* on a fixed instance I.

- This is the mean time taken by algorithm *A* to solve the instance I over and over. Thus, it is natural to talk about the worst case expected time and the average case expected time.

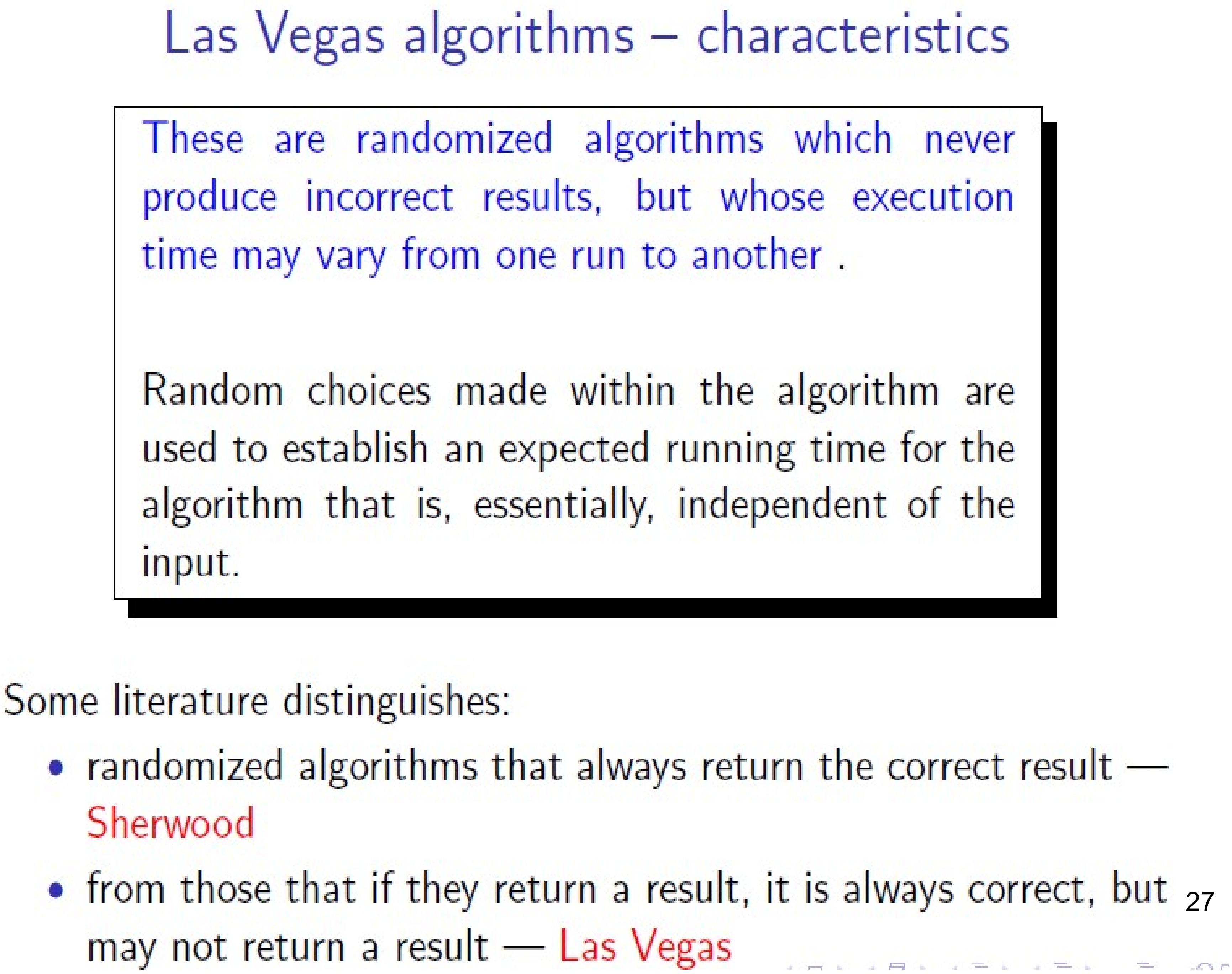
expected versus average time.

- Average time of a deterministic algorithm
  - The average time taken by the algorithm when each possible instance of a given size is considered equally likely.
  - Average case analysis can be misleading if in fact some instances are more likely than others.
- Expected time of a probabilistic algorithm is defined on each individual instance
  - Mean time that it would take to solve the same instance over and over again.
- Worst case expected time of a probabilistic algorithm
  - The expected time taken by the worst possible instance of a given size, not the time incurred if the worst possible probabilistic choices are unfortunately taken.

- Las vegas algorithm is more efficient than deterministic ones,but only w.r.t expected time. the expected behaviour of Las vegas algorithm can be much better than that of deterministic algorithms.

- Quick sort
  - Worst case expected time –O (n log n)
  - Worst case to sort n elements- Ω (n2)

Las Vegas Algorithms



Las Vegas algorithms – characteristics

These are randomized algorithms which never produce incorrect results, but whose execution time may vary from one run to another .

Random choices made within the algorithm are used to establish an expected running time for the algorithm that is, essentially, independent of the input.

Some literature distinguishes:

- randomized algorithms that always return the correct result — Sherwood
- from those that if they return a result, it is always correct, but may not return a result — Las Vegas

Summary

- Introduction
- *n Queens Problems*
- Factorizing Large Integers

Introduction

- A Las Vegas algorithm is a type of randomized algorithm that uses a random value to make randomized choices and never produces an incorrect answer.
- The choices made during computation attempt to guide the algorithm to the desired solution faster than other methods.

- Make randomized choices to get to an answer quicker on average
- Never returns a wrong answer,but no answer is possible.
- Note: This is different than guaranteeing a right answer

Deterministic vs. Las Vegas

Possible Outcomes

- Correct Answer

Running Time

- Fast for a few
- Slow for most

Possible Outcomes

- Correct Answer
- No answer

Running Time

- The slow cases have a probability of going fast
- The fast cases have a probability of slowing down

Las Vegas algorithm

The answer obtained is always correct but sometime no answer.

Modified deterministic algorithm by using randomness in the decision

When dead-end restart the algo.

Average running time analysis assumes some distribution of problem instances.

Robinhood effect

LV “steal” time from the “rich” instance -- instances that were solved quickly by deterministic algo. -- to give it to the “poor” instance.

Reduce the difference between good and bad instances.

- Las Vegas algorithms exist for problems with no known efficient deterministic algorithms
- Las Vegas algorithms have good expected performance for any instance to be solved since Deterministic algorithms could be vulnerable to an unexpected probability distribution of instances that may focus on the catastrophic worst-case instances

- Las Vegas algorithms have a $p(x)$ probability of solving an instance x.
- The $p(x)$ value tells us on average how many times we need to run to get an answer
- A correct Las Vegas algorithms should have

  $p(x)>0$ for all input instances, in more strong sense, $p(x)\geqslant\delta$, $\delta>0$.

```
RepeatLV($x$)
repeat
    LV($x$, $y$, success)
until success
return $y$
```

Where, *y* is a solution to problem. Due to $p(x)>0$, Las Vegas can always find a solution within enough long running time.

Running time analysis

RepeatLV each LV call success p(x); expected time t(x) before repeatLV is successful.

Must consider success and failure separately

RepeatLV each LV call success p(x); expected time t(x) before repeatLV is successful.

Must consider success and failure separately

p(x) : first call LV(x) succeeds after expected time s(x)

1-p(x) : first call LV(x) fails after expected time f(x) then restart, total expected time f(x) + t(x)

$t(x) = p(x)\, s(x) + (1- p(x))(f(x) + t(x))$

$t(x) = s(x) + (1-p(x))/p(x) \quad f(x)$

This equation guides how to “tune” various parameters.

Probabilistic selection and sorting

The problem of k-th smallest element using divide and conquer, the nearer the pivot is to the median of the element the more efficient.

Simple approach : pivot the first element

linear time in average, quadratic in worst case.

```
SelectionLV(T[1..n],s)
i = 1; j = n
repeat
  p = T[ uniform(i.. j) ]
  pivotbis(T[i..j], p, k, l )
  if s <= k then j = k
  else if s >= l then i = l
  else return p
```

Expected run time is linear

- Using random pivot, the execution time is independent of the instance.

- The expected time taken by probabilistic selection algorithm is linear, independently of the instance to be solved.

- It is always possible that some execution will take quadratic time but the prob. will be small if n is large.

- using Probabilistic approach, transform alg into Las Vegas alg.-efficient with high probability, whatever the instance considered.

- Expected run time is linear on all instances with a small hidden constant.

```
quicksortLV(T[i..j] )
if j-i is sufficiently small then insertsort(T[i..j] )
else
  p = T[ uniform(i..j) ]
  pivotbis( T[i..j], p, k, l )
  quicksortLV(T[i..k] )
  quicksortLV(T[l..j] )
```

Worst-case expected time O(n log n)

LV running time is independent of specific instances.

Probabilistic approach

Deterministic algo. that has excellent average execution time on all instances of some particular size except certain instances.

Turn that into LV that is efficient with high probability whatever the instance considered.

Quicksort - Randomized

Randomized Quick Sort

- In traditional Quick Sort,
  - we will always pick the first element as the pivot for partitioning.
  - The worst case runtime is O(n^2) while the expected runtime is O($n$log$n$) over the set of all input.
  - Therefore, some input are born to have long runtime, e.g., an inversely sorted list.
- In randomized Quick Sort,
  - we will pick randomly an element as the pivot for partitioning.
  - The expected runtime of any input is O($n$log$n$).

Randomized Quicksort

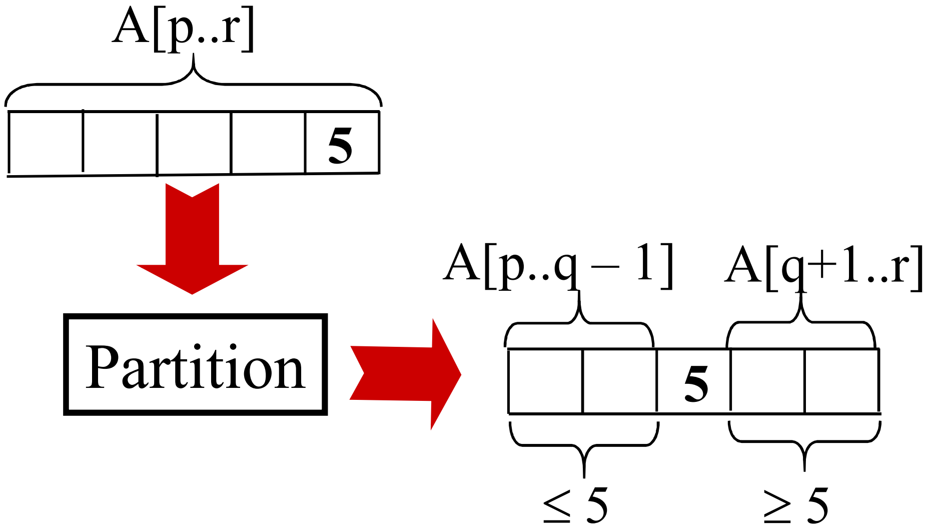
- Want to make running time independent of input ordering.
- How can we do that?
  - Make the algorithm randomized.
  - Make every possible input equally likely.
    - Can randomly shuffle to permute the entire array.
    - For quicksort, it is sufficient if we can ensure that every element is equally likely to be the *pivot*.
    - So, we choose an element in $A[p..r]$ and exchange it with $A[r]$.
    - Because the *pivot* is randomly chosen, we expect the partitioning to be well balanced on average.

Randomized Version

Want to make running time independent of input ordering.

```
Randomized-Quicksort(A, p, r)
if p < r then
        q := Randomized-Partition(A, p, r);
        Randomized-Quicksort(A, p, q – 1);
        Randomized-Quicksort(A, q + 1, r)
```

```
Randomized-Partition(A, p, r)
i := Random(p, r);
A[r] ↔ A[i];
Partition(A, p, r)
```



```
Partition(A, p, r)
x := A[r];
     i := p – 1;
for j := p to r – 1 do
        if A[j] ≤ x then
                i := i + 1;
                A[i] ↔ A[j]
A[i + 1] ↔ A[r];
```

Avg. Case Analysis of Randomized Quicksort

Let RV X = number of comparisons over all calls to Partition.

Suffices to compute E[X]. **Why?**

Notation:

- Let $z_1, z_2, \ldots, z_n$ denote the list items (in sorted order).
- Let $Z_{ij} = \{z_i, z_{i+1}, \ldots, z_j\}$.

Let RV $X_{ij} = \begin{cases} 1 & \text{if } z_i \text{ is compared to } z_j \\ 0 & \text{otherwise} \end{cases}$

X_{ij} is an **indicator random variable**.
X_{ij}=I{z_i is compared to z_j}.

Thus, the total no. of comparisons performed by the algorithm

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

Analysis (Continued)

We have:

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

$$= \sum_{i=1}^{n-1}\sum_{j=i+1}^{n} E[X_{ij}]$$

$$= \sum_{i=1}^{n-1}\sum_{j=i+1}^{n} P[z_i \text{ is compared to } z_j]$$

Note:

$E[X_{ij}] = 0 \cdot P[X_{ij}=0] + 1 \cdot P[X_{ij}=1]$
$\qquad\quad = P[X_{ij}=1]$

This is a nice property of indicator RVs. (Refer to notes on Probabilistic Analysis.)

So, all we need to do is to compute P[z_i is compared to z_j].

The set Z_{ij} has j-i+1 elements ,the probability that any given element is the first one chosen as pivot is 1/(j-i+1)

Analysis (Continued)

z_i and z_j are compared iff the first element to be chosen as a pivot from Z_{ij} is either z_i or z_j.

Exercise: Prove this.

So,
$$\begin{aligned} P[z_i \text{ is compared to } z_j] &= P[z_i \text{ or } z_j \text{ is first pivot from } Z_{ij}] \\ &= P[z_i \text{ is first pivot from } Z_{ij}] \\ &\quad + P[z_j \text{ is first pivot from } Z_{ij}] \\ &= \frac{1}{j-i+1} + \frac{1}{j-i+1} \\ &= \frac{2}{j-i+1} \end{aligned}$$

Analysis (Continued)

Therefore,

$$\begin{aligned}
E[X] &= \sum_{i=1}^{\text{n-1}} \sum_{j=i+1}^{n} \frac{2}{j-i+1} \\
&= \sum_{i=1}^{\text{n-1}} \sum_{k=1}^{\text{n-i}} \frac{2}{k+1} \\
&< \sum_{i=1}^{\text{n-1}} \sum_{k=1}^{n} \frac{2}{k} \\
&= \sum_{i=1}^{\text{n-1}} O(\lg n) \\
&= \boxed{O(n \lg n).}
\end{aligned}$$

Substitute $k = j - i.$

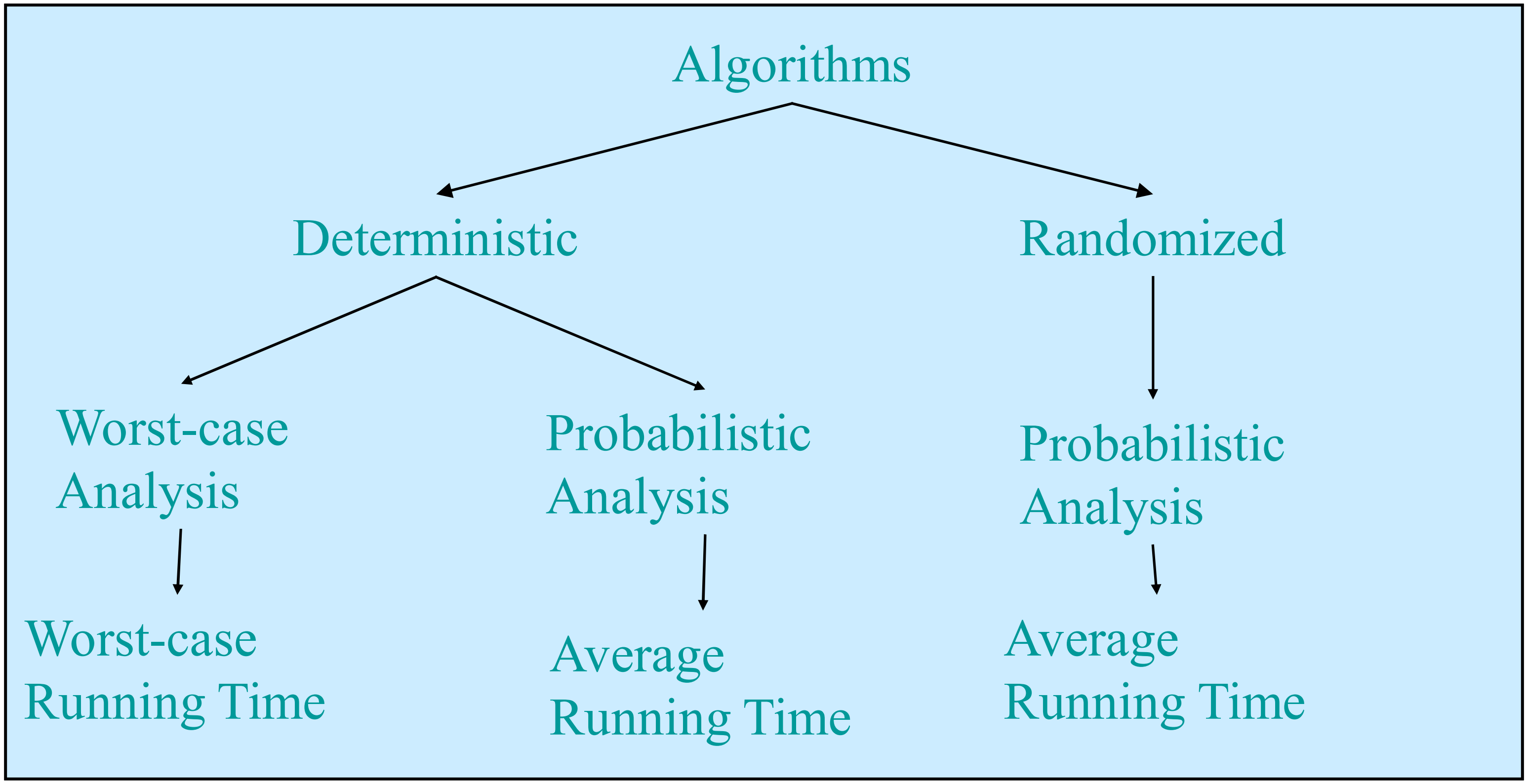
$$\sum_{k=1}^{n} \frac{1}{k} = H_n (n^{\text{th}} \text{ Harmonic number})$$

$$H_n = \ln n + O(1), \text{ Eq.}(A.7)$$

Randomized QS is a Las Vegas algorithm.

Deterministic vs. Randomized Algorithms

- Deterministic Algorithm : Identical behavior for different runs for a given input.
- Randomized Algorithm : Behavior is generally different for different runs for a given input.

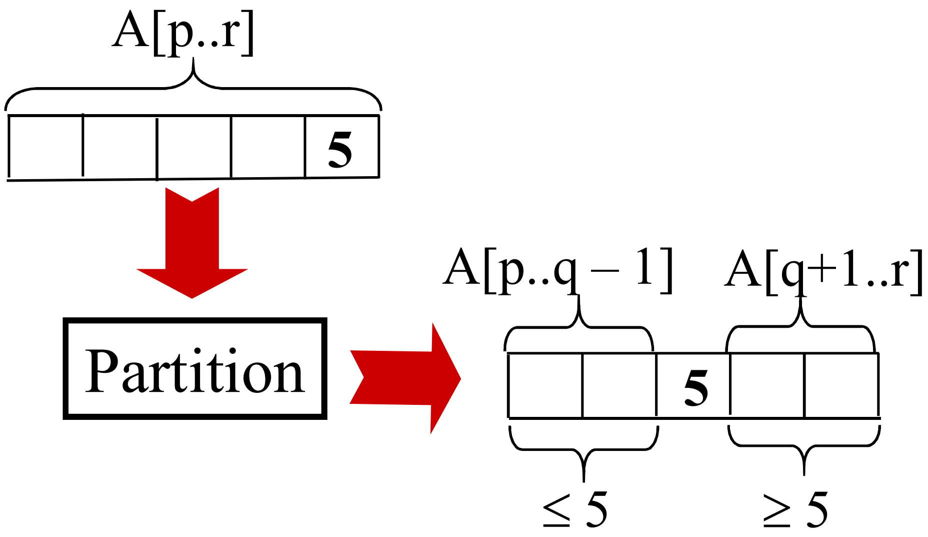


Quicksort - Randomized

Quicksort: review

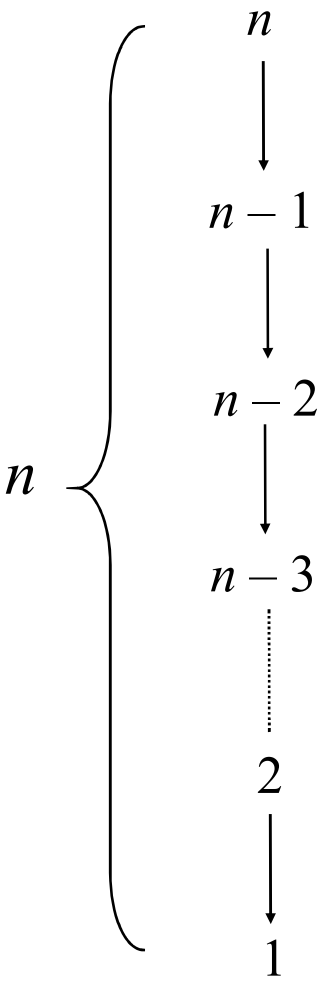
```
Quicksort(A, p, r)
if p < r then
        q := Partition(A, p, r);
        Quicksort(A, p, q – 1);
        Quicksort(A, q + 1, r)
fi
```

```
Partition(A, p, r)
x, i  := A[r], p – 1;
for j := p to r – 1 do
        if A[j]  ≤  x then
                i := i + 1;
                A[i] ↔ A[j]
        fi
od;
A[i + 1] ↔ A[r];
return i + 1
```



Worst-case Partition Analysis

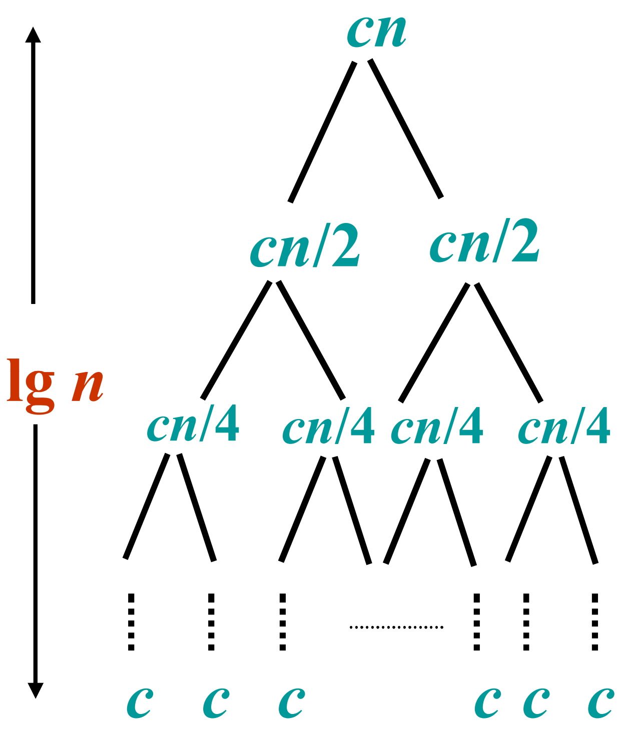
Recursion tree for worst-case partition



Split off a single element at each level:

$$
\begin{aligned}
T(n) &= T(n-1) + T(0) + \text{PartitionTime}(n)\\
&= T(n-1) + \Theta(n)\\
&= \textstyle\sum_{k=1 \text{ to } n} \Theta(k)\\
&= \Theta(\textstyle\sum_{k=1 \text{ to } n} k)\\
&= \Theta(n^2)
\end{aligned}
$$

Best-case Partitioning



- Each subproblem size ≤ *n*/2.
- Recurrence for running time
  - $T(n) \leq 2T(n/2) +$ PartitionTime(n)
    $= 2T(n/2) + \Theta(n)$
- $\boldsymbol{T(n) = \Theta(n \lg n)}$

Variations

- Quicksort is not very efficient on small lists.
- This is a problem because Quicksort will be called on lots of small lists.
- **Fix 1:** Use Insertion Sort on small problems.
- **Fix 2:** Leave small problems unsorted. Fix with one final Insertion Sort at end.
  - **Note:** Insertion Sort is very fast on almost-sorted lists.

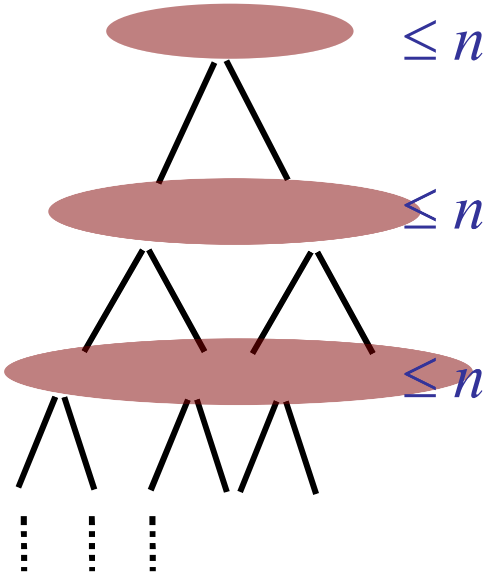
Unbalanced Partition Analysis

What happens if we get poorly-balanced partitions,
e.g., something like: $T(n) \le T(9n/10) + T(n/10) + \Theta(n)$?
Still get $\Theta(n \lg n)$!! (As long as the split is of constant proportionality.)

Intuition: Can divide n by $c > 1$ only $\Theta(\lg n)$ times before getting 1.

$$\left.\begin{array}{l} n \\ \downarrow \\ n/c \\ \downarrow \\ n/c^2 \\ \downarrow \\ \vdots \\ \downarrow \\ 1 = n/c^{\log_c n} \end{array}\right\}$$

Roughly $\log_c n$ levels;
Cost per level is $O(n)$.

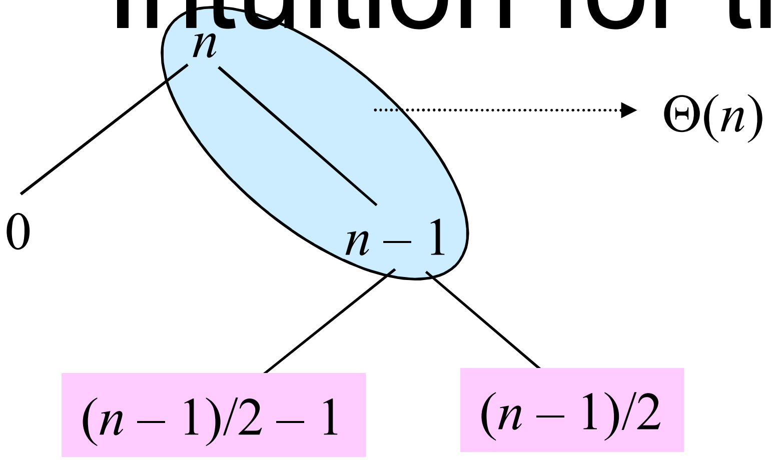


(**Remember:** Different base logs are related by a constant.)

Intuition for the Average Case

- Partitioning is unlikely to happen in the same way at every level.
  - Split ratio is different for different levels. (Contrary to our assumption in the previous slide.)
- Partition produces a mix of "good" and "bad" splits, distributed randomly in the recursion tree.
- What is the running time likely to be in such a case?

Intuition for the Average Case

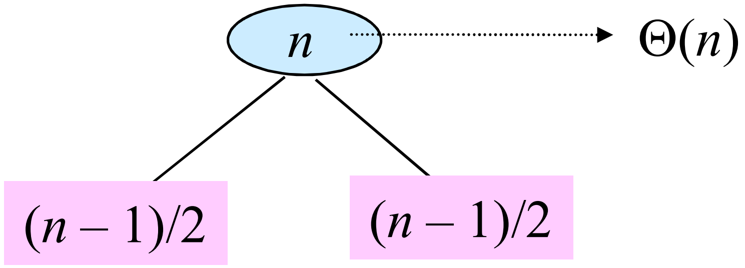


Bad split followed by a good split:

Produces subarrays of sizes 0, $(n-1)/2-1$, and $(n-1)/2$.

Cost of partitioning :

$$\Theta(n) + \Theta(n\text{-}1) = \Theta(n).$$



Good split at the first level:

Produces two subarrays of size $(n-1)/2$.

Cost of partitioning :

$$\Theta(n).$$

Situation at the end of case 1 is not worse than that at the end of case 2.

When splits alternate between good and bad, the cost of bad split can be absorbed into the cost of good split.

Thus, running time is $O(n \lg n)$, though with larger hidden constants.

Variations (Continued)

- Input distribution may not be uniformly random.
- **Fix 1:** Use “randomly” selected pivot.
  - We’ll analyze this in detail.
- **Fix 2:** Median-of-three Quicksort.
  - Use median of three fixed elements (say, the first, middle, and last) as the pivot.
  - To get O(n^2) behavior, we must continually be unlucky to see that two out of the three elements examined are among the largest or smallest of their sets.

Analysis of Randomized QS

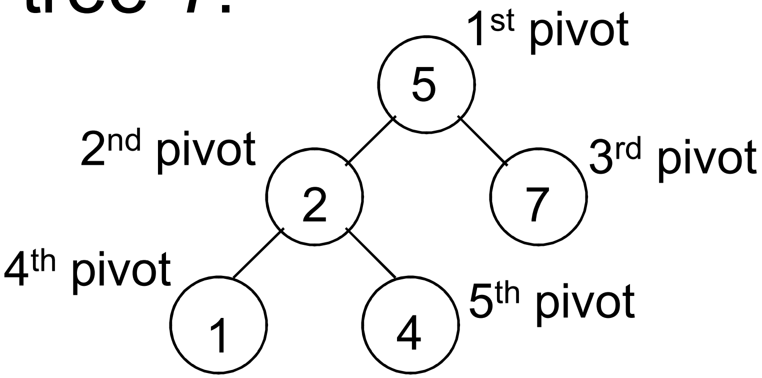
- Let $s(i)$ be the i^{th} smallest element in the input list S.
- X_{ij} is a random variable such that $X_{ij} = 1$ if $s(i)$ is compared with $s(j)$; $X_{ij} = 0$ otherwise.
- Expected runtime t of randomized QS is:

$$t = E[\sum_{i=1}^{n} \sum_{j \geq i} X_{ij}] = \sum_{i=1}^{n} \sum_{j \geq i} E[X_{ij}]$$

- $E[X_{ij}]$ is the expected value of X_{ij} over the set of all random choices of the pivots, which is equal to the probability p_{ij} that $s(i)$ will be compared with $s(j)$.

Analysis of Randomized QS

- We can represent the whole sorting process by a binary tree *T*:



- Notice that *s*(*i*) will be compared with *s*(*j*) where i<j if and only if *s*(*i*) or *s*(*j*) is the first one among the set {*s*(*i*), *s*(*i*+1), …, *s*(*j*)} to be selected as the pivot.
- Note that p_{ij} = 2/(*j*-*i*+1). Why?

Analysis of Randomized QS

Therefore, the expected runtime *t*:

$$t \quad ¿\sum_{i=1}^{n}\sum_{j\geq i} E[X_{ij}] \quad ¿\sum_{i=1}^{n}\sum_{j\geq i} p_{ij}$$

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

$$¿2n\ln n$$

Note that $1+\frac{1}{2}+\frac{1}{3}+\ldots+\frac{1}{n}\,\square\ln n$