TIME COMPLEXITY OF MIN. FINDING

$= n-1 = \theta(n)$.

TIME COMPLEXITY OR RUN TIME OF SELECTION SORT $= \frac{n(n-1)}{2}$

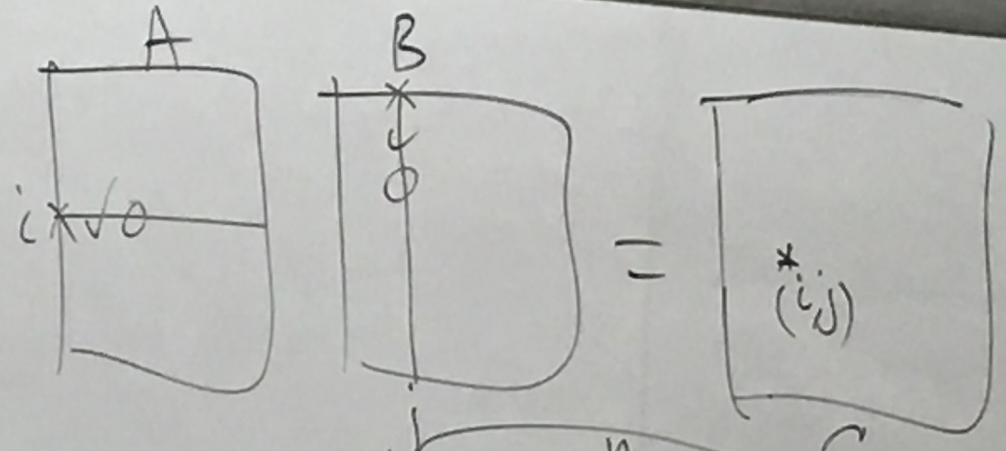
$= \frac{n^2}{2} - \frac{n}{2} = \theta(n^2)$.

PROBLEM:

INPUT: TWO MATRICE $A_{n \times n}$ and $B_{n \times n}$

OUTPUT: $C = AB$.

```
for i = 1 to n do
    for j = 1 to n do
        C[i,j] = 0.0;
        for k = 1 to n do
            C[i,j] = C[i,j] + A[i,k] * B[k,j];
```



$$C_{ij} = \sum_{k=1}^{n} A_{ik} B_{kj}$$

RUN TIME

$= n^2(2n-1) = \Theta(n^3)$

STANDARD ALGORITHM DESIGN TECHNIQUES:

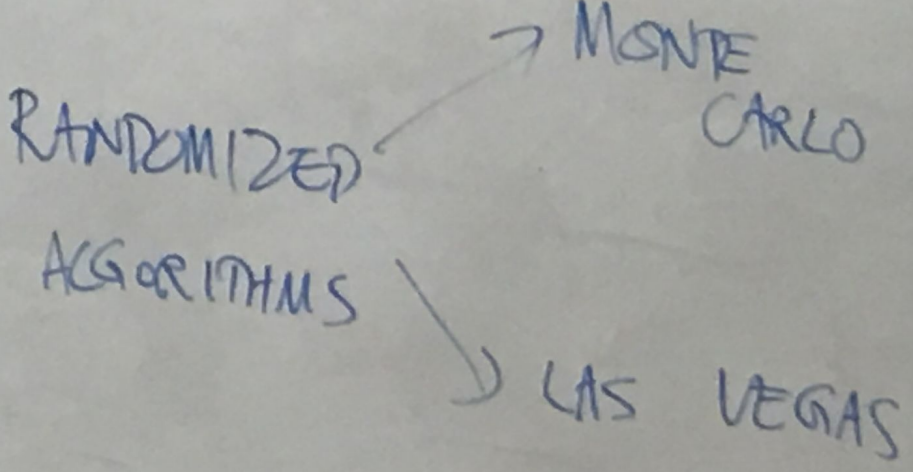
1. DIVIDE-AND-CONQUER
2. GREEDY
3. DYNAMIC PROGRAMMING
4. BACKTRACKING ...

ASSUME that 10^9 operations take 1 sec.

n	n	n^2	n^{10}	2^n
10	10^{-8}	10^{-7}	1	10^{-6}
100	10^{-7}	10^{-5}	10^{11} sec.	10^{33} sec. $\approx 3\times10^{25}$ yr.
10,000	10^{-5}	10^{-1} sec.		

RANDOMIZED ALGORITHMS.

A RANDOMIZED ALG. IS ONE WHERE CERTAIN DECISIONS ARE MADE BASED ON OUTCOMES OF COIN FLIPS.



A MONTE CARLO ALG.
RUNS FOR A PRESPECIFIED
AMOUNT OF TIME.
The output is correct
with a HIGH PROBABILITY.

A LAS VEGAS ALG.
ALWAYS OUTPUTS the
CORRECT ANSWER.
the RUN TIME IS A
RANDOM VARIABLE.

BY A HIGH PROBABILITY WE MEAN A PROB. OF AT LEAST $(1 - n^{-\alpha})$, where n is the INPUT SIZE, and α is A PROB. PARAMETER, ASSUMED TO BE A CONSTANT ≥ 1.

ADVANTAGES:

1. SIMPLICITY
2. BETTER PERFORMANCE (Theoretically as well as in PRACTICE).

Let $n = 10{,}000$; $\alpha = 100$.

$n^{-\alpha} = 10^{-400}$.

PROBLEM:

INPUT: $X = k_1, k_2, \ldots, k_n$.

X has $\frac{n}{2}$ COPIES OF ONE ELEMENT; the other elements are distinct.

OUTPUT: The REPEATED ELEMENT.

$X = 3,5,7,4,5,2,5,5$

DETERMINISTIC ALG.

1. ITERATE through X: $\Theta(n^2)$
2. SORTING: $\Theta(n \log n)$.
3. MEDIAN FINDING: $\Theta(n)$.
4. GROUP X into groups of SIZE 3 each; Look for

Repetitions within the groups.
$\Rightarrow \theta(n)$.

FACT: ANY DET. ALG. HAS TO SPEND $\geq \frac{n}{2}+2$ steps.

PROOF:

Consider an adversary who knows the alg. The adv. can make sure that the FIRST $(\frac{n}{2}+1)$ elements looked at by the algorithm are DISTINCT. □

A LAS VEGAS ALG.

REPEAT

BASIC STEP

Flip an n-sided coin to get i;
Flip an n-sided coin to get j;
if $k_i = k_j$ and $i \neq j$ then
Output k_i and QUIT;

FOREVER

ANALYSIS:

PROB. OF SUCCESS IN ONE BASIC STEP

$$= \frac{\frac{n}{2}\left(\frac{n}{2}-1\right)}{n^2}.$$

This prob. is $\geq \frac{1}{5}$ $\forall n \geq 10$.

$\Rightarrow$ Prob. of Failure in one basic step is $\leq \left(\frac{4}{5}\right)$.

CSE 3500 Algorithms and Complexity – Fall 2016
Lecture 2: September 1, 2016

Asymptotic Functions

- Asymptotic functions O, Ω, and Θ were introduced in the last lecture. When we analyze the run times of algorithms, it is desirable to express the run times using Θ.
- The run time of the minimum finding algorithm is $\Theta(n)$ and the run time of the selection sort algorithm is $\Theta(n^2)$.
- Consider the following matrix multiplication algorithm. (Input: two $n \times n$ matrices A and B; Output: $C = AB$.)

```
for 1 ≤ i, j ≤ n do
    C[i, j] = 0.0;
for i = 1 to n do
    for j = 1 to n do
        for k = 1 to n do
            C[i, j] = C[i, j] + A[i, k] * B[k, j];
```

  In the above algorithm we perform n multiplications and $n-1$ additions for every element in the output. There are n^2 elements in the output. Therefore, the total run time is $n^2(n+n-1) = 2n^3 - n^2 = \Theta(n^3)$.

- **Claim:** If $f(n)$ is a non-negative integer function of n such that $f(n) = a_k n^k + a_{k-1}n^{k-1} + \cdots + a_2 n^2 + a_1 n + a_0$, then $f(n) = \Theta(n^k)$. Here $a_k, a_{k-1}, \ldots, a_2, a_1, a_0$ are constants and k is a positive integer constant. (*Prove this*).

Algorithm Design

- Unfortunately, there are no standard recipes for designing optimal algorithms.
- There are some commonly used algorithm design techniques: divide-and-conquer, greedy, dynamic programming, backtracking, etc.
- There is no guarantee that using a subset of these techniques will always yield optimal algorithms.

- In any algorithm there is typically an idea that does not fall under any of these techniques. This idea is very much dependent on the problem and the algorithm developer.

Randomized Algorithms

- For many algorithms (such as quicksort) the average run time is much better than the worst case run time. Average run times are computed assuming a specific (typically uniform) distribution on the input space. However, this assumption may not hold in practice. Is it possible to develop algorithms whose run times will be as good as the average run times but where no assumptions are made on the input space? The answer is yes and such algorithms employ coin flips.
- A randomized algorithm is one where certain decisions are made based on outcomes of coin flips made in the algorithm. The analysis of a randomized algorithm is done without assuming anything about the input distribution. The analysis is done in the space of all possible outcomes for coin flips made in the algorithm (rather than in the space of all possible inputs).
- There are two kinds of randomized algorithms: Monte Carlo and Las Vegas.
- A Monte Carlo algorithm typically pertains to decision problems (i.e., problems for which the answer is either "yes" or "no"). The output of a Monte Carlo algorithm is correct with a very high probability.
- A Las Vegas algorithm always outputs the correct answer and its run time is a random variable. Ideally, we would like to prove that the run time of a Las Vegas algorithm is "small" with a very high probability. Such run time bounds proven are referred to as "high probability bounds".
- Randomized algorithms offer simplicity and better performance relative to their deterministic counterparts. For a number of fundamental problems in computing (such as sorting, selection, convex hull, etc.) the best known algorithms happen to be randomized. These algorithms have the best asymptotic run times and perform the best in practice.
- By a high (or very high) probability we mean a probability of at least $(1 - n^{-\alpha})$, where n is the input size and α is a probability parameter (assumed to be a constant ≥ 1). In the analysis we keep α as though it is a variable. By a low probability we mean a probability of $\leq n^{-\alpha}$. For instance, if $n = 1,000$ and $\alpha = 100$, then, $n^{-\alpha} = 10^{-300}$.

Identification of the Repeated Element

- Consider the following problem. Input: $X = k_1, k_2, \ldots, k_n$. It is known that X has $\frac{n}{2}$ copies of one element and the other elements are distinct (i.e., they occur exactly once

each). Output: the repeated element.

- We can solve the above problem deterministically in a number of ways: 1) We can sort the sequence X and scan through the sorted list. Copies of the repeated element will be found in successive positions. The run time of this algorithm will be $\Theta(n \log n)$ (if we use merge sort, for example); 2) We can use a median finding algorithm to identify the repeated element. There exist $\Theta(n)$ time algorithms for median finding; 3) There is a relatively simple linear time (i.e., $\Theta(n)$ time) algorithm for finding the repeated element: Partition X into groups of size 3 each and look for the repeated element in the individual groups. By pigeon-hole principle, it follows that at least one of the groups will have at least two copies of the repeated element. For each group we can check if it has two or more copies of any element using 3 comparisons. Therefore, we spend a total of n comparisons.

- We can argue that any deterministic algorithm for solving the above problem needs $\frac{n}{2} + 2$ time as follows: Consider an adversary who has perfect knowledge about the algorithm to be used and who is picking the input. Specifically, the adversary knows the order in which the sequence elements are accessed by the algorithm. In this case the adversary can make sure that the first $\frac{n}{2} + 1$ elements accessed by the algorithm are distinct, forcing the algorithm to access at least one more element.

- We can develop a Las Vegas algorithm that takes only $O(\log n)$ time with a high probability.

  repeat
    Flip an n-sided coin to get i;
    Flip an n-sided coin to get j;
    if $i = j$ and $k_i = k_j$ **then** output k_i and quit;
  forever

- **Claim:** The run time of the above algorithm is $\widetilde{O}(\log n)$.