Lecture 2:
Recursion & Performance analysis
BRUTE-FORCE SOLUTION: \( O(n!) \)

DYNAMIC PROGRAMMING ALGORITHMS: \( O(n^2 2^n) \)

SELLING ON EBAY: \( O(1) \)

STILL WORKING ON YOUR ROUTE?

SHUT THE HELL UP.
System Life Cycle

• Programs pass through a period called ‘system life cycle’, which is defined by the following steps:

  • **1. Requirements**: It is defined by the *inputs* given to the program and *outputs* that will be produced by the program.

  • **2. Analysis**: The first job after defining the requirements is to analyze the problem. There are two ways for this:
    • Bottom-up
    • Top-down
System Life Cycle (cont’)

• **3. Design:** Data objects and the possible operations between the objects are defined in this step. Therefore, abstract data objects and the algorithm are defined. They are all independent of the programming language.

• **4. Refinement and Coding:** Algorithms are used in this step in order to do operations on the data objects.

• **5. Verification:** Correctness proofs, testing and error removal are performed in this step.
Cast of Characters

- **Programmer** needs to develop a working solution.
- **Client** wants to solve problem efficiently.
- **Student** might play any or all of these roles someday.
- **Theoretician** wants to understand.
Recursion
Recursion

- Recursion is all about breaking a big problem into smaller occurrences of that same problem.
  - Each person can solve a small part of the problem.
    - What is a small version of the problem that would be easy to answer?
    - What information from a neighbor might help me?
Recursive Algorithm

- Number of people behind me:
  - If there is someone behind me, ask him/her how many people are behind him/her.
    - When they respond with a value $N$, then I will answer $N + 1$.
  - If there is nobody behind me, I will answer 0.
Recursive Algorithms

- Functions can call themselves (direct recursion)
- A function that calls another function is called by the second function again. (indirect recursion)
Recursion

• Consider the following method to print a line of * characters:

```java
// Prints a line containing the given number of stars.
// Precondition: n >= 0
void printStars(int n) {
    for (int i = 0; i < n; i++) {
        System.out.print("*");
    }
    System.out.println(); // end the line of output
}
```

• Write a recursive version of this method (that calls itself).
  • Solve the problem without using any loops.
  • Hint: Your solution should print just one star at a time.
A basic case

• What are the cases to consider?
  • What is a very easy number of stars to print without a loop?

```java
public static void printStars(int n) {
    if (n == 1) {
        // base case; just print one star
        System.out.println("*");
    } else {
        ...
    }
}
```
Handling more cases

• Handling additional cases, with no loops (in a bad way):

```java
public static void printStars(int n) {
    if (n == 1) {
        // base case; just print one star
        System.out.println("*");
    } else if (n == 2) {
        System.out.print("*");
        System.out.println("*");
    } else if (n == 3) {
        System.out.print("*");
        System.out.print("*");
        System.out.println("*");
    } else if (n == 4) {
        System.out.print("*");
        System.out.print("*");
        System.out.print("*");
        System.out.println("*");
    } else ...
    
    }
```
Handling more cases 2

- Taking advantage of the repeated pattern (somewhat better):

```java
public static void printStars(int n) {
    if (n == 1) {
        // base case; just print one star
        System.out.println("*");
    } else if (n == 2) {
        System.out.print("*");
        printStars(1); // prints "*
    } else if (n == 3) {
        System.out.print("*");
        printStars(2); // prints "**
    } else if (n == 4) {
        System.out.print("*");
        printStars(3); // prints "***
    } else ...
}
```
Using recursion properly

- Condensing the recursive cases into a single case:

```java
public static void printStars(int n) {
    if (n == 1) {
        // base case; just print one star
        System.out.println("*");
    } else {
        // recursive case; print one more star
        System.out.print("*");
        printStars(n - 1);
    }
}
```
Even simpler

• The real, even simpler, base case is an n of 0, not 1:

```java
public static void printStars(int n) {
    if (n == 0) {
        // base case; just end the line of output
        System.out.println();
    } else {
        // recursive case; print one more star
        System.out.print("*");
        printStars(n - 1);
    }
}
```
Recursive tracing

• Consider the following recursive method:

```java
public static int mystery(int n) {
    if (n < 10) {
        return n;
    } else {
        int a = n / 10;
        int b = n % 10;
        return mystery(a + b);
    }
}
```

• What is the result of the following call?
  mystery(648)
A recursive trace

mystery(648):
• int a = 648 / 10;  // 64
• int b = 648 % 10;  // 8
• return mystery(a + b);  // mystery(72)

mystery(72):
• int a = 72 / 10;  // 7
• int b = 72 % 10;  // 2
• return mystery(a + b);  // mystery(9)

mystery(9):
• return 9;
Consider the following recursive method:

```java
public static int mystery(int n) {
    if (n < 10) {
        return (10 * n) + n;
    } else {
        int a = mystery(n / 10);
        int b = mystery(n % 10);
        return (100 * a) + b;
    }
}
```

What is the result of the following call?

`mystery(348)`
A recursive trace 2

mystery(348)

- int a = mystery(34);
  - int a = mystery(3);
    return (10 * 3) + 3; // 33
  - int b = mystery(4);
    return (10 * 4) + 4; // 44
  - return (100 * 33) + 44; // 3344

- int b = mystery(8);
  return (10 * 8) + 8; // 88
  - return (100 * 3344) + 88; // 334488

- What is this method really doing?
Binary Search

• A value is searched in a sorted array. If found, the index of the item is returned, otherwise -1 is returned.

```c
int binsearch(int list[], int searchnum, int left, int right)
{
    int middle;
    while(left <= right){
        middle = (left + right)/2;
        switch(compare(list[middle], searchnum)){
            case -1 : left = middle + 1; break;
            case 0; return middle;
            case 1: right = middle -1;
        }
    }
    return -1;
}
```

• Macro for ‘COMPARE’:

```c
#define COMPARE(x,y) (((x)<(y))? -1 ((x)==(y))?0:1)
```
Binary Search

Let’s search for 18:

```plaintext
4 13 18 19 24 27 33 36 39 40
```

 screenings: 4 13 18 19 24 27 33 36 39 40

 screenings: 4 13 18 19 24 27 33 36 39 40

 screenings: 4 13 18 19 24 27 33 36 39 40
int binsearch(int list[], int searchnum, int left, int right)
{
    int middle;
    if(left<=right){
        middle = (left+right)/2;
        switch(COMpare(list[middle], searchnum)){
            case -1: return binsearch(list, searchnum, middle+1, right);
            case 0: return middle;
            case 1: return binsearch(list, searchnum, left, middle-1);
        }
    }
    return -1;
}
Binary Search (revisited)
Permutation problem

- Finding all the permutations of a given set with size \( n \geq 1 \).
  - Remember there are \( n! \) different sequences of this set.
  - Example: Find all the permutations of \{a,b,c\}
Permutation problem

- Example: Find all the permutations of \{a,b,c,d\}

- All the permutations of \{b,c,d\} follow ‘a’
- All the permutations of \{a,c,d\} follow ‘b’
- All the permutations of \{a,b,d\} follow ‘c’
- All the permutations of \{a,b,c\} follow ‘d’
#include <stdio.h>
#include <stdlib.h>
#define MAX_SIZE 10
#define SWAP(x,y,t) ((t)==(x), (x)==(y), (y)==(t))

void perm(char*, int, int);

void main(void)
{
    int n;
    char list[MAX_SIZE];
    printf("Enter the values:");
    for(n=0; (list[n]=getchar())!="\n"; n++)
        if(n>=MAX_SIZE){
            printf("\nOverflow error!\n");
            exit(1);
        }
    perm(list,0,n-1);
}
void perm(char* list, int i, int n) 
{
    int j, temp;
    if(i==n){
        for(j=0; j<=n; j++)
            printf("%c", list[j]);
        printf("\n");
    }
    else{
        for(j=i; j<=n; j++){
            SWAP(list[i], list[j], temp);
            perm(list, i+1, n);
            SWAP(list[i], list[j], temp);
        }
    }
}
Factorial Function

• $n! = n \times (n-1) \times (n-2) \times \ldots \times 2 \times 1$ for any integer $n > 0$

  $0! = 1$

• **Iterative Definition:**

```c
int fval=1;
for(i=n; i>=1; i--)
    fval = fval*i;
```
Factorial Function

Recursive Definition

• To define \( n! \) recursively, \( n! \) must be defined in terms of the factorial of a smaller number.

• Observation (problem size is reduced):

\[
n! = n \times (n - 1)!
\]

• Base case:

\[
0! = 1
\]

• We can reach the base case by subtracting 1 from \( n \) if \( n \) is a positive integer.
Factorial Function
Recursive Definition

Recursive definition

\[ n! = 1 \quad \text{if } n = 0 \]
\[ n! = n \times (n - 1)! \quad \text{if } n > 0 \]

```c
int fact(int n)
{
    if(n==0)
        return (1);
    else
        return (n*fact(n-1));
}
```

This fact function satisfies the four criteria of a recursive solution.
Four Criteria of a Recursive Solution

1. A recursive function calls itself.
   - This action is what makes the solution recursive.

2. Each recursive call solves an identical, but smaller, problem.
   - A recursive function solves a problem by solving another problem that is identical in nature but smaller in size.

3. A test for the base case enables the recursive calls to stop.
   - There must be a case of the problem (known as base case or stopping case) that is handled differently from the other cases (without recursively calling itself.)
   - In the base case, the recursive calls stop and the problem is solved directly.

4. Eventually, one of the smaller problems must be the base case.
   - The manner in which the size of the problem diminishes ensures that the base case is eventually is reached.
Fibonacci Sequence

• It is the sequence of integers:
  
  \[
  t_0 \quad t_1 \quad t_2 \quad t_3 \quad t_4 \quad t_5 \quad t_6 \\
  0 \quad 1 \quad 1 \quad 2 \quad 3 \quad 5 \quad 8 \quad \ldots
  \]

• Each element in this sequence is the sum of the two preceding elements.

• The specification of the terms in the Fibonacci sequence:

\[
\begin{align*}
  t_n &= n \quad \text{if } n \text{ is } 0 \text{ or } 1 \text{ (i.e. } n < 2) \\
  t_n &= t_{n-1} + t_{n-2} \quad \{\text{otherwise}\}
\end{align*}
\]
Fibonacci Sequence

- Iterative solution

```c
int Fib(int n)
{
    int prev1, prev2, temp, j;
    if(n==0 || n==1)
        return n;
    else{
        prev1=0;
        prev2=1;
        for(j=1;j<=n;j++)
        {
            temp = prev1+prev2;
            prev1=prev2;
            prev2=temp;
        }
        return prev2;
    }
}
```
Fibonacci Sequence

- Recursive solution

```cpp
int fibonacci(int n) {
    if(n<2)
        return n;
    else
        return (fibonacci(n-2)+fibonacci(n-1));
}
```
Performance Analysis
Running Time

"As soon as an Analytic Engine exists, it will necessarily guide the future course of the science. Whenever any result is sought by its aid, the question will arise—By what course of calculation can these results be arrived at by the machine in the shortest time?" — Charles Babbage (1864)

how many times do you have to turn the crank?

Analytic Engine
Observations

Example-1

3-SUM. Given $N$ distinct integers, how many triples sum to exactly zero?

```
% more 8ints.txt
8
30 -40 -20 -10 40 0 10 5
% java ThreeSum 8ints.txt
4
```

<table>
<thead>
<tr>
<th></th>
<th>a[i]</th>
<th>a[j]</th>
<th>a[k]</th>
<th>sum</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>30</td>
<td>-40</td>
<td>10</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>30</td>
<td>-20</td>
<td>-10</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>-40</td>
<td>40</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>-10</td>
<td>0</td>
<td>10</td>
<td>0</td>
</tr>
</tbody>
</table>

Context. Deeply related to problems in computational geometry.
Observations
Example-1: 3-SUM brute force algorithm

```java
class ThreeSum {
    public static int count(int[] a) {
        int N = a.length;
        int count = 0;
        for (int i = 0; i < N; i++)
            for (int j = i+1; j < N; j++)
                for (int k = j+1; k < N; k++)
                    if (a[i] + a[j] + a[k] == 0)
                        count++;
        return count;
    }

    public static void main(String[] args) {
        In in = new In(args[0]);
        int[] a = in.readAllInts();
        StdOut.println(count(a));
    }
}
```
Empirical Analysis

Run the program for various input sizes and measure running time.

<table>
<thead>
<tr>
<th>( N )</th>
<th>time (seconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td>250</td>
<td>0.0</td>
</tr>
<tr>
<td>500</td>
<td>0.0</td>
</tr>
<tr>
<td>1,000</td>
<td>0.1</td>
</tr>
<tr>
<td>2,000</td>
<td>0.8</td>
</tr>
<tr>
<td>4,000</td>
<td>6.4</td>
</tr>
<tr>
<td>8,000</td>
<td>51.1</td>
</tr>
<tr>
<td>16,000</td>
<td>?</td>
</tr>
</tbody>
</table>
Data Analysis

Standard plot. Plot running time $T(N)$ vs. input size $N$. 
Total running time: sum of cost $\times$ frequency for all operations.
- Need to analyze program to determine set of operations.
- Cost depends on machine, compiler.
- Frequency depends on algorithm, input data.

In principle, accurate mathematical models are available.
## Cost of Basic Operations

**Challenge.** How to estimate constants.

<table>
<thead>
<tr>
<th>operation</th>
<th>example</th>
<th>nanoseconds †</th>
</tr>
</thead>
<tbody>
<tr>
<td>integer add</td>
<td>$a + b$</td>
<td>2.1</td>
</tr>
<tr>
<td>integer multiply</td>
<td>$a \times b$</td>
<td>2.4</td>
</tr>
<tr>
<td>integer divide</td>
<td>$a / b$</td>
<td>5.4</td>
</tr>
<tr>
<td>floating-point add</td>
<td>$a + b$</td>
<td>4.6</td>
</tr>
<tr>
<td>floating-point multiply</td>
<td>$a \times b$</td>
<td>4.2</td>
</tr>
<tr>
<td>floating-point divide</td>
<td>$a / b$</td>
<td>13.5</td>
</tr>
<tr>
<td>sine</td>
<td>Math.sin(theta)</td>
<td>91.3</td>
</tr>
<tr>
<td>arctangent</td>
<td>Math.atan2(y, x)</td>
<td>129.0</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

† Running OS X on Macbook Pro 2.2GHz with 2GB RAM
Cost of Basic Operations

**Observation.** Most primitive operations take constant time.

<table>
<thead>
<tr>
<th>operation</th>
<th>example</th>
<th>nanoseconds</th>
</tr>
</thead>
<tbody>
<tr>
<td>variable declaration</td>
<td>int a</td>
<td>$c_1$</td>
</tr>
<tr>
<td>assignment statement</td>
<td>a = b</td>
<td>$c_2$</td>
</tr>
<tr>
<td>integer compare</td>
<td>a &lt; b</td>
<td>$c_3$</td>
</tr>
<tr>
<td>array element access</td>
<td>a[i]</td>
<td>$c_4$</td>
</tr>
<tr>
<td>array length</td>
<td>a.length</td>
<td>$c_5$</td>
</tr>
<tr>
<td>1D array allocation</td>
<td>new int[N]</td>
<td>$c_6 \cdot N$</td>
</tr>
<tr>
<td>2D array allocation</td>
<td>new int[N][N]</td>
<td>$c_7 \cdot N^2$</td>
</tr>
</tbody>
</table>

**Caveat.** Non-primitive operations often take more than constant time.

novice mistake: abusive string concatenation
Example: 1-Sum

Q. How many instructions as a function of input size $N$?

```
int count = 0;
for (int i = 0; i < N; i++)
    if (a[i] == 0)
        count++;
```

N array accesses

<table>
<thead>
<tr>
<th>operation</th>
<th>frequency</th>
</tr>
</thead>
<tbody>
<tr>
<td>variable declaration</td>
<td>2</td>
</tr>
<tr>
<td>assignment statement</td>
<td>2</td>
</tr>
<tr>
<td>less than compare</td>
<td>$N + 1$</td>
</tr>
<tr>
<td>equal to compare</td>
<td>$N$</td>
</tr>
<tr>
<td>array access</td>
<td>$N$</td>
</tr>
<tr>
<td>increment</td>
<td>$N$ to $2N$</td>
</tr>
</tbody>
</table>
Example: 2-Sum

Q. How many instructions as a function of input size $N$?

```java
int count = 0;
for (int i = 0; i < N; i++)
    for (int j = i+1; j < N; j++)
        if (a[i] + a[j] == 0)
            count++;
```

Pf. [n even]

\[
0 + 1 + 2 + \ldots + (N-1) = \frac{1}{2}N(N-1) = \binom{N}{2}
\]

\[
0 + 1 + 2 + \ldots + (N-1) = \frac{1}{2}N^2 - \frac{1}{2}N
\]
Example: 2-Sum

Q. How many instructions as a function of input size $N$?

```java
int count = 0;
for (int i = 0; i < N; i++)
    for (int j = i+1; j < N; j++)
        if (a[i] + a[j] == 0)
            count++;
```

$$0 + 1 + 2 + \ldots + (N - 1) = \frac{1}{2} N (N - 1) = \binom{N}{2}$$

<table>
<thead>
<tr>
<th>operation</th>
<th>frequency</th>
</tr>
</thead>
<tbody>
<tr>
<td>variable declaration</td>
<td>$N + 2$</td>
</tr>
<tr>
<td>assignment statement</td>
<td>$N + 2$</td>
</tr>
<tr>
<td>less than compare</td>
<td>$\frac{1}{2} (N + 1) (N + 2)$</td>
</tr>
<tr>
<td>equal to compare</td>
<td>$\frac{1}{2} N (N - 1)$</td>
</tr>
<tr>
<td>array access</td>
<td>$N (N - 1)$</td>
</tr>
<tr>
<td>increment</td>
<td>$\frac{1}{2} N (N - 1)$ to $N (N - 1)$</td>
</tr>
</tbody>
</table>

tedious to count exactly
Simplifying the Calculations

“It is convenient to have a measure of the amount of work involved in a computing process, even though it be a very crude one. We may count up the number of times that various elementary operations are applied in the whole process and then given them various weights. We might, for instance, count the number of additions, subtractions, multiplications, divisions, recording of numbers, and extractions of figures from tables. In the case of computing with matrices most of the work consists of multiplications and writing down numbers, and we shall therefore only attempt to count the number of multiplications and recordings.” — Alan Turing

ROUNDING-OFF ERRORS IN MATRIX PROCESSES

By A. M. TURING
(National Physical Laboratory, Teddington, Middlesex)
[Received 4 November 1947]

SUMMARY
A number of methods of solving sets of linear equations and inverting matrices are discussed. The theory of the rounding-off errors involved is investigated for some of the methods. In all cases examined, including the well-known ‘Gauss elimination process’, it is found that the errors are normally quite moderate: no exponential build-up need occur.
Simplification: cost model

Cost model. Use some basic operation as a proxy for running time.

```java
int count = 0;
for (int i = 0; i < N; i++)
    for (int j = i+1; j < N; j++)
        if (a[i] + a[j] == 0)
            count++;
```

\[ 0 + 1 + 2 + \ldots + (N-1) = \frac{1}{2} N (N - 1) = \binom{N}{2} \]

<table>
<thead>
<tr>
<th>operation</th>
<th>frequency</th>
</tr>
</thead>
<tbody>
<tr>
<td>variable declaration</td>
<td>(N + 2)</td>
</tr>
<tr>
<td>assignment statement</td>
<td>(N + 2)</td>
</tr>
<tr>
<td>less than compare</td>
<td>(\frac{1}{2} (N + 1) (N + 2))</td>
</tr>
<tr>
<td>equal to compare</td>
<td>(\frac{1}{2} N (N - 1))</td>
</tr>
<tr>
<td><strong>array access</strong></td>
<td>(N (N - 1))</td>
</tr>
<tr>
<td>increment</td>
<td>(\frac{1}{2} N (N - 1) ) to (N (N - 1))</td>
</tr>
</tbody>
</table>

Cost model = array accesses
(we assume compiler/JVM do not optimize any array accesses away!)
Asymptotic Notation: 2-SUM problem

Q. Approximately how many array accesses as a function of input size $N$?

```c
int count = 0;
for (int i = 0; i < N; i++)
    for (int j = i+1; j < N; j++)
        if (a[i] + a[j] == 0)
            count++;
```

```
0 + 1 + 2 + ... + (N - 1) = \frac{1}{2} N(N - 1) = \binom{N}{2}
```

A. $\sim N^2$ array accesses.
Q. Approximately how many array accesses as a function of input size $N$?

```java
int count = 0;
for (int i = 0; i < N; i++)
    for (int j = i+1; j < N; j++)
        for (int k = j+1; k < N; k++)
            if (a[i] + a[j] + a[k] == 0)
                count++;
```

A. $\sim \frac{1}{2} N^3$ array accesses.

$$\binom{N}{3} = \frac{N(N-1)(N-2)}{3!} \sim \frac{1}{6} N^3$$
Invariant. If key appears in array a[], then a[lo] ≤ key ≤ a[hi].

```java
public static int binarySearch(int[] a, int key) {
    int lo = 0, hi = a.length - 1;
    while (lo <= hi) {
        int mid = lo + (hi - lo) / 2;
        if (key < a[mid]) hi = mid - 1;
        else if (key > a[mid]) lo = mid + 1;
        else return mid;
    }
    return -1;
}
```

why not mid = (lo + hi) / 2?

one "3-way compare"
Proposition. Binary search uses at most \( 1 + \lg N \) key compares to search in a sorted array of size \( N \).

Def. \( T(N) = \# \) key compares to binary search a sorted subarray of size \( \leq N \).

Binary search recurrence. \( T(N) \leq T(N/2) + 1 \) for \( N > 1 \), with \( T(1) = 1 \).

Pf sketch. [assume \( N \) is a power of 2]

\[
\begin{align*}
T(N) & \leq T(N/2) + 1 \quad \text{[ given ]} \\
& \leq T(N/4) + 1 + 1 \quad \text{[ apply recurrence to first term ]} \\
& \leq T(N/8) + 1 + 1 + 1 \quad \text{[ apply recurrence to first term ]} \\
& \quad \vdots \\
& \leq T(N/N) + 1 + 1 + \ldots + 1 \quad \text{[ stop applying, } T(1) = 1 \text{ ]} \\
& = 1 + \lg N \quad \text{[ stop applying, } T(1) = 1 \text{ ]}
\end{align*}
\]
Types of Analyses

**Best case**: Lower bound on cost.
  - Determined by “easiest” input.
  - Provides a good for all inputs.

**Worst case**: Upper bound on cost.
  - Determined by “most difficult” input.
  - Provides a guarantee for all inputs.

**Average case**: Expected cost for random input.
  - Need a model for “random” input.
  - Provides a way to predict performance.

slide by R. Sedgewick and K. Wayne
Types of Analyses

**Ex 1.** Array accesses for brute-force 3-SUM.
- Best: $\sim \frac{1}{2} N^3$
- Average: $\sim \frac{1}{2} N^3$
- Worst: $\sim \frac{1}{2} N^3$

**Ex 2.** Compares for binary search.
- Best: $\sim 1$
- Average: $\sim \lg N$
- Worst: $\sim \lg N$
Mathematical Models for Running Time

In principle, accurate mathematical models are available.

In practice,

- Formulas can be complicated.
- Advanced mathematics might be required.
- Exact models best left for experts.

\[ T_N = c_1 A + c_2 B + c_3 C + c_4 D + c_5 E \]

- \( A \) = array access
- \( B \) = integer add
- \( C \) = integer compare
- \( D \) = increment
- \( E \) = variable assignment

Bottom line. We use approximate models in this course: \( T(N) \sim c N^3 \).
Asymptotic notation: $O (\text{Big Oh})$

$f(n) = O(g(n))$ iff there exist positive constants $c$ and $n_0$ such that 
\[ f(n) \leq cg(n) \text{ for all } n, n \geq n_0 \]

\[
\begin{align*}
3n + 2 &= O(n) & \text{as } 3n + 2 \leq 4n & \text{for all } n \geq 2 \\
10n^2 + 4n + 2 &= O(n^2) & \text{as } 10n^2 + 4n + 3 \leq 11n^2 & \text{for } n \geq 5 \\
6 \times 2^n + n^2 &= O(2^n) & \text{as } 6 \times 2^n + n^2 \leq 7 \times 2^n & \text{for } n \geq 4 \\
3n + 3 &= O(n) & \text{Correct, OK.} \\
3n + 3 &= O(n^2) & \text{Correct, NO!}
\end{align*}
\]
Question

How many array accesses does the following code fragment make as a function of $N$?

```c
int count = 0;
for (int i = 0; i < N; i++)
    for (int j = i+1; j < N; j++)
        for (int k = 1; k < N; k = k*2)
            if (a[i] + a[j] >= a[k])
                count++;
```

A. $\sim 3N^2$
B. $\sim 3/2 N^2 \lg N$
C. $\sim 3/2 N^3$
D. $\sim 3 N^3$
E. *I don't know.*
Performance Analysis

- There are various criteria in order to evaluate a program.
- The questions:
  - Does the program meet the requirements determined at the beginning?
  - Does it work correctly?
  - Does it have any documentation on the subject of how it works?
  - Does it use the function effectively in order to produce the logical values: TRUE and FALSE?
  - Is the program code readable?
  - Does the program use the primary and the secondary memory effectively?
  - Is the running time of the program acceptable?
Performance Analysis

Performance Evaluation

Space Complexity  
Time Complexity
Space Complexity

- Memory space needed by a program to complete its execution:
  - A fixed part $\rightarrow c$
  - A variable part $\rightarrow S_p(\text{instance})$

$$S(P) = c + S_p(\text{instance})$$

\begin{verbatim}
float abc(float a, float b, float c){
    return a+b+b*c+(a+b+c)/(b+c);
}
\end{verbatim}

\begin{verbatim}
float sum(float list[], int n){
    float tempsum=0;
    int i;
    for(i=0;i<n;i++){
        tempsum += list[i];
    }
    return tempsum;
}
\end{verbatim}

$S_{abc}(I)=0$

$S_{sum}(n)=0$
Space Complexity

• Recursive function call:

```
float rsum(float list[], int n){
    if(n)
        return rsum(list, n-1)+list[n-1];
    return 0;
}
```

• What is the total variable memory space of this method for an input size of 2000?
Time Complexity

- Total time used by the program is: \( T(P) \)
- Where \( T(P) = \text{compile time} + \text{run (execution) time} \) \( (T_P) \)

```c
float sum(float list[], int n){
    float tempsum=0;  1
    int i;
    for(i=0;i<n;i++){
        tempsum += list[i];  n
    }
    return tempsum;  1
}
```

Total step number = 2n+3
Time Complexity

- What is the number of steps required to execute this method?

```c
float rsum(float list[], int n){
    if(n)
        return rsum(list, n-1)+list[n-1];
    return 0;
}
```
Time Complexity

- What is the number of steps required to execute this method?
O (Big Oh) Notation

- Time complexity = algorithm complexity
- \( f(n) < c g(n) \) \( \rightarrow \) \( f(n) = O(g(n)) \)

- \( T_{\text{sum}}(n) = 2n + 3 \) \( \rightarrow \) \( T_{\text{sum}}(n) = O(n) \)

- \( T_{\text{rsum}}(n) = 2n + 2 \) \( \rightarrow \) \( T_{\text{sum}}(n) = O(n) \)

- \( T_{\text{add}}(\text{rows, cols}) = 2\text{rows}*\text{cols} + 2\text{rows} + 1 \) \( \rightarrow \) \( T_{\text{add}}(n) = O(\text{rows}*\text{cols}) = O(n^2) \)
Asymptotic Notation

• Example: $f(n)=3n^3+6n^2+7n$

• What is the algorithm complexity in O notation?
“Magic Square” Problem

• A magic square is a square grid where the numbers in each row and in each column add up to the same number.

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>15</td>
<td>8</td>
<td>1</td>
<td>24</td>
<td>17</td>
</tr>
<tr>
<td>16</td>
<td>14</td>
<td>7</td>
<td>5</td>
<td>23</td>
</tr>
<tr>
<td>22</td>
<td>20</td>
<td>13</td>
<td>6</td>
<td>4</td>
</tr>
<tr>
<td>3</td>
<td>21</td>
<td>19</td>
<td>12</td>
<td>10</td>
</tr>
<tr>
<td>9</td>
<td>2</td>
<td>25</td>
<td>18</td>
<td>11</td>
</tr>
</tbody>
</table>
“Magic Square” Problem
--- Based on Coxeter’s rule

```c
#include <stdio.h>
#include <stdlib.h>
#define MAX_SIZE 15

void main(void)
{
    static int square[MAX_SIZE][MAX_SIZE];
    int i, j, row, column;
    int count;
    int size;

    printf("\n Enter the square size: ");
    scanf(" %d ",&size);
    //Check the square size

    for(i=0; i<size; i++)
        for(j=0; j<size; j++)
            square[i][j]=0;
    square[0][((size-1)/2)]=1; //ilk satırın orta elemanı
    i=0;                     //ilk satır
    j=(size-1)/2;            //orta kolon
```
“Magic Square” Problem
--- Based on Coxeter’s rule

```c
for(count=2; count<= size*size; count++){
    row=(i-1 < 0) ? (size-1) : (i-1);
    column=(j-1 < 0) ? (size-1) : (j-1);

    if(square[row][column])
        i=(++i)%size;
    else{
        i=row;
        j=column;
    }
    square[i][j]=count;
}
printf(“Magix Square size:%d :n\n “,size);
for(i=0; i<size; i++)
    for(j=0; j<size; j++)
        printf(“%5d“, square[i][j]);
    printf(“\n“);
}
```
Question

• Example: If the processor can execute 1 billion commands in 1 second, calculate the time required for the ‘magic square’ problem solved by brute force for n=5.
## Practical Complexities

<table>
<thead>
<tr>
<th>Complexity (time)</th>
<th>Name</th>
<th>1</th>
<th>2</th>
<th>4</th>
<th>8</th>
<th>16</th>
<th>32</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>constant</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>( \log n )</td>
<td>logarithmic</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>( n )</td>
<td>linear</td>
<td>1</td>
<td>2</td>
<td>4</td>
<td>8</td>
<td>16</td>
<td>32</td>
</tr>
<tr>
<td>( n \log n )</td>
<td>log linear</td>
<td>0</td>
<td>2</td>
<td>8</td>
<td>24</td>
<td>64</td>
<td>160</td>
</tr>
<tr>
<td>( n^2 )</td>
<td>quadratic</td>
<td>1</td>
<td>4</td>
<td>16</td>
<td>64</td>
<td>256</td>
<td>1024</td>
</tr>
<tr>
<td>( n^3 )</td>
<td>cubic</td>
<td>1</td>
<td>8</td>
<td>64</td>
<td>512</td>
<td>4096</td>
<td>32768</td>
</tr>
<tr>
<td>( 2^n )</td>
<td>exponential</td>
<td>2</td>
<td>4</td>
<td>16</td>
<td>256</td>
<td>65536</td>
<td>4294967296</td>
</tr>
<tr>
<td>( n! )</td>
<td>factorial</td>
<td>1</td>
<td>2</td>
<td>24</td>
<td>40326</td>
<td>20922789888000</td>
<td>26313*10^{33}</td>
</tr>
</tbody>
</table>
Practical Complexities
<table>
<thead>
<tr>
<th>n</th>
<th>$f(n)=n$</th>
<th>$f(n)=\log_2 n$</th>
<th>$f(n)=n^2$</th>
<th>$f(n)=n^3$</th>
<th>$f(n)=n^4$</th>
<th>$f(n)=n^{10}$</th>
<th>$f(n)=2^n$</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>0.01μ</td>
<td>0.03μ</td>
<td>0.1μ</td>
<td>1μ</td>
<td>10μ</td>
<td>10 sec</td>
<td>1μ</td>
</tr>
<tr>
<td>20</td>
<td>0.02μ</td>
<td>0.09μ</td>
<td>0.4μ</td>
<td>8μ</td>
<td>160μ</td>
<td>2.84 hr</td>
<td>1 ms</td>
</tr>
<tr>
<td>30</td>
<td>0.03μ</td>
<td>0.15μ</td>
<td>0.9μ</td>
<td>27μ</td>
<td>810μ</td>
<td>6.83 d</td>
<td>1 sec</td>
</tr>
<tr>
<td>40</td>
<td>0.04μ</td>
<td>0.21μ</td>
<td>1.6μ</td>
<td>64μ</td>
<td>2.56ms</td>
<td>121.36 d</td>
<td>18.3 min</td>
</tr>
<tr>
<td>50</td>
<td>0.05μ</td>
<td>0.28μ</td>
<td>2.5μ</td>
<td>125μ</td>
<td>6.25ms</td>
<td>3.1 yr</td>
<td>13 d</td>
</tr>
<tr>
<td>100</td>
<td>0.10μ</td>
<td>0.66μ</td>
<td>10μ</td>
<td>1ms</td>
<td>100ms</td>
<td>3171 yr</td>
<td>4*10^{13} yr</td>
</tr>
<tr>
<td>1000</td>
<td>1μ</td>
<td>9.96μ</td>
<td>1ms</td>
<td>1sec</td>
<td>16.67min</td>
<td>3.17*10^{13} yr</td>
<td></td>
</tr>
<tr>
<td>10,000</td>
<td>10μ</td>
<td>130.03μ</td>
<td>100ms</td>
<td>16.67min</td>
<td>115.7d</td>
<td>3.17*10^{23} yr</td>
<td></td>
</tr>
<tr>
<td>100,000</td>
<td>100μ</td>
<td>1.66ms</td>
<td>10sec</td>
<td>11.57d</td>
<td>3171 yr</td>
<td>3.17*10^{33} yr</td>
<td></td>
</tr>
<tr>
<td>1,000,000</td>
<td>1ms</td>
<td>19.92ms</td>
<td>16.67min</td>
<td>31.71yr</td>
<td>3.17*10^{7} yr</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1,000,000,000</td>
<td>1ms</td>
<td>19.92ms</td>
<td>16.67min</td>
<td>31.71yr</td>
<td>3.17*10^{43} yr</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Survey of Common Running Times
Tilde notation

- Estimate running time (or memory) as a function of input size $N$.
- Ignore lower order terms.
  - when $N$ is large, terms are negligible
  - when $N$ is small, we don't care

Ex 1. $\frac{1}{6}N^3 + 20N + 16 \sim \frac{1}{6}N^3$
Ex 2. $\frac{1}{6}N^3 + 100N^{4/3} + 56 \sim \frac{1}{6}N^3$
Ex 3. $\frac{1}{6}N^3 - \frac{1}{2}N^2 + \frac{1}{2}N \sim \frac{1}{6}N^3$

discard lower-order terms
(e.g., $N = 1000$: 166.67 million vs. 166.17 million)

Technical definition. $f(N) \sim g(N)$ means $\lim_{N \to \infty} \frac{f(N)}{g(N)} = 1$
Tilde Notation ($\sim$) (same as $\theta$-notation)

- Estimate running time (or memory) as a function of input size $N$.
- Ignore lower order terms.
  - when $N$ is large, terms are negligible
  - when $N$ is small, we don't care

<table>
<thead>
<tr>
<th>operation</th>
<th>frequency</th>
<th>tilde notation</th>
</tr>
</thead>
<tbody>
<tr>
<td>variable declaration</td>
<td>$N + 2$</td>
<td>$\sim N$</td>
</tr>
<tr>
<td>assignment statement</td>
<td>$N + 2$</td>
<td>$\sim N$</td>
</tr>
<tr>
<td>less than compare</td>
<td>$\frac{1}{2} (N + 1) (N + 2)$</td>
<td>$\sim \frac{1}{2} N^2$</td>
</tr>
<tr>
<td>equal to compare</td>
<td>$\frac{1}{2} N (N - 1)$</td>
<td>$\sim \frac{1}{2} N^2$</td>
</tr>
<tr>
<td>array access</td>
<td>$N (N - 1)$</td>
<td>$\sim N^2$</td>
</tr>
<tr>
<td>increment</td>
<td>$\frac{1}{2} N (N - 1)$ to $N (N - 1)$</td>
<td>$\sim \frac{1}{2} N^2$ to $\sim N^2$</td>
</tr>
</tbody>
</table>
Asymptotic notation: $\Omega$ (Omega)

$f(n) = O(g(n))$ iff there exist positive constants $c$ and $n_0$ such that 
\[ f(n) \geq cg(n) \text{ for all } n, n \geq n_0 \]

\[
\begin{align*}
3n+2 &= \Omega(n) & \text{as } 3n+2 \geq 3n & \text{for all } n \geq 1 \\
10n^2+4n+2 &= \Omega(n^2) & \text{as } 10n^2 + 4n + 2 \geq n^2 & \text{for } n \geq 1 \\
6 \times 2^n + n^2 &= \Omega(2^n) & 6 \times 2^n + n^2 \geq 2^n & \text{for } n \geq 1
\end{align*}
\]

\[
\begin{align*}
3n+3 &= \Omega(n) & \text{Correct, OK.} \\
3n+3 &= \Omega(1) & \text{Correct, NO!}
\end{align*}
\]
Asymptotic notation: $\Theta$ (theta) (same as “$\sim$” notation)

$f(n) = O(g(n))$ iff there exist positive constants $c_1$, $c_2$, and $n_0$ such that

$$c_1g(n) \leq f(n) \leq c_2g(n) \text{ for all } n, n \geq n_0$$

$$3n+2 = \Theta(n)$$  as $3n+2 \geq 3n$  for all $n \geq 1$

$$10n^2+4n+2 = \Theta(n^2)$$  as $10n^2 + 4n + 2 \geq n^2$  for $n \geq 1$

$$6*2^n + n^2 = \Theta(2^n)$$  $6*2^n + n^2 \geq 2^n$  for $n \geq 1$

$$3n+3 = \Omega(n)$$  Correct, OK.

$$3n+3 = \Omega(2n)$$  Correct, not used!
Linear time: $O(n)$

Linear time. Running time is proportional to input size.

Computing the maximum. Compute maximum of $n$ numbers $a_1, \ldots, a_n$.

```plaintext
max ← a_1
for i = 2 to n {
    if (a_i > max)
        max ← a_i

}
Linear time: $O(n)$

**Merge.** Combine two sorted lists $A = a_1, a_2, \ldots, a_n$ with $B = b_1, b_2, \ldots, b_n$ into sorted whole.

\[
\text{Merged result} \quad \begin{array}{c}
| a_i \quad \text{A} \\
| b_j \quad \text{B} \\
\end{array}
\]

\[
i = 1, \ j = 1 \\
\text{while } (\text{both lists are nonempty}) \{
\quad \text{if } (a_i \leq b_j) \text{ append } a_i \text{ to output list and increment } i \\
\quad \text{else} \quad \text{append } b_j \text{ to output list and increment } j \\
\}
\text{append remainder of nonempty list to output list}
\]

**Claim.** Merging two lists of size $n$ takes $O(n)$ time.

**Pf.** After each compare, the length of output list increases by 1.
Linearithmic time: $O(n \log n)$

**$O(n \log n)$ time.** Arises in divide-and-conquer algorithms.

**Sorting.** Mergesort and heapsort are sorting algorithms that perform $O(n \log n)$ compares.

**Largest empty interval.** Given $n$ time-stamps $x_1, \ldots, x_n$ on which copies of a file arrive at a server, what is largest interval when no copies of file arrive?

**$O(n \log n)$ solution.** Sort the time-stamps. Scan the sorted list in order, identifying the maximum gap between successive time-stamps.
Quadratic time: $O(n^2)$

Ex. Enumerate all pairs of elements.

Closest pair of points. Given a list of $n$ points in the plane $(x_1, y_1), \ldots, (x_n, y_n)$, find the pair that is closest.

$O(n^2)$ solution. Try all pairs of points.

```plaintext
min ← (x_1 - x_2)^2 + (y_1 - y_2)^2
for i = 1 to n {
    for j = i+1 to n {
        d ← (x_i - x_j)^2 + (y_i - y_j)^2
        if (d < min)
            min ← d
    }
}
```

Remark. $\Omega(n^2)$ seems inevitable, but this is just an illusion. [see Chapter 5]
Cubic time: $O(n^3)$

Cubic time. Enumerate all triples of elements.

Set disjointness. Given $n$ sets $S_1, \ldots, S_n$ each of which is a subset of $1, 2, \ldots, n$, is there some pair of these which are disjoint?

$O(n^3)$ solution. For each pair of sets, determine if they are disjoint.

```
foreach set $S_i$ {
    foreach other set $S_j$ {
        foreach element $p$ of $S_i$ {
            determine whether $p$ also belongs to $S_j$
        }
        if (no element of $S_i$ belongs to $S_j$)
            report that $S_i$ and $S_j$ are disjoint
    }
}
```
Independent set of size $k$. Given a graph, are there $k$ nodes such that no two are joined by an edge?

**$O(n^k)$ solution.** Enumerate all subsets of $k$ nodes.

```plaintext
foreach subset $S$ of $k$ nodes {
    check whether $S$ in an independent set
    if (S is an independent set)
        report $S$ is an independent set
}
```

- Check whether $S$ is an independent set takes $O(k^2)$ time.
- Number of $k$ element subsets $\binom{n}{k} = \frac{n(n-1)(n-2) \times \cdots \times (n-k+1)}{k(k-1)(k-2) \times \cdots \times 1} \leq \frac{n^k}{k!}$
- $O(k^2 \frac{n^k}{k!}) = O(n^k)$.

poly-time for $k=17$, but not practical
Exponential time

**Independent set.** Given a graph, what is maximum cardinality of an independent set?

**O\((n^2 2^n)\) solution.** Enumerate all subsets.

```
S* ← ∅
foreach subset S of nodes {
    check whether S in an independent set
    if (S is largest independent set seen so far)
        update S* ← S
}
```
Sublinear time

Search in a sorted array. Given a sorted array $A$ of $n$ numbers, is a given number $x$ in the array?

$O(\log n)$ solution. Binary search.

```python
lo ← 1, hi ← n
while (lo ≤ hi) {
    mid ← (lo + hi) / 2
    if (x < A[mid]) hi ← mid - 1
    else if (x > A[mid]) lo ← mid + 1
    else return yes
}
return no
```
References

- Kevin Wayne, “Analysis of Algorithms”
- Sartaj Sahni, “Analysis of Algorithms”
- BBM 201 Notes by Mustafa Ege
- Marty Stepp and Helene Martin, Recursion