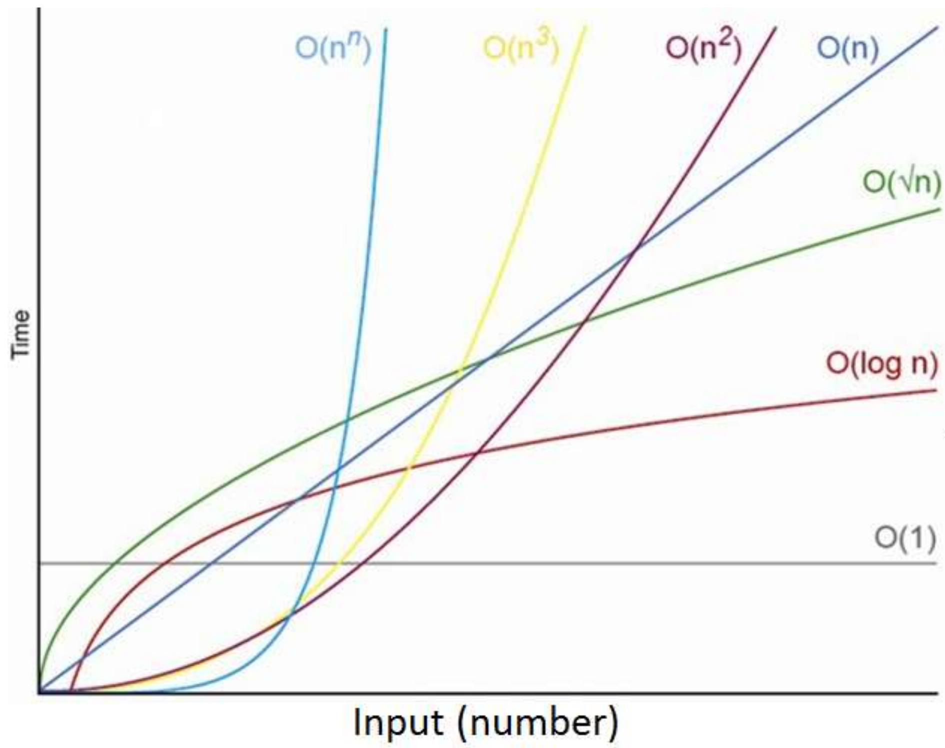


LECTURE 4

Which g(n)- Inherent imprecision of the big-O

The big-O notation is **inherently imprecise** as there can be infinitely many functions ***g*** for a **given function *f*.**

- For example, the $f(n)=2\,n^2 + 3n+1$ is big-O not only of $\boldsymbol{n^2}$, but also of $\boldsymbol{n^3}$**, . . . ,** $\boldsymbol{n^k}$**,** . . . for any $k \geq 2$.
- **To avoid this embarrassment of riches, the smallest function *g* is chosen, n^2 in this case.**

Real World Example:

- If a child, ask what comes after **class 1**
  - Answer could be class 2nd , 3rd ,4th, 5th , … but one should say **2nd**

EXAMPLES
BASIC LOOP ORDERS

Simple Loop Orders

Example 0

for (i=0;i<n;i=i++)

Loop will run approximately **n** times … O(n)

Example 1

for (i=0;i<n;i=i+k)

Loop will run approximately **n/k** times … O(n)

Example 2

for (i=n;i>0;i=i-k)

Loop will run approximately **n/k** times … O(n)

Example 3

for (i=1;i<n; i=i*k)

Loop will run approximately $\mathbf{\log_k n}$ times … O($\mathbf{\log_k n}$)

Nested Loops Orders

Example

```
for(i=0;i<n; i++)
        for (j=i; j<n;++j)
```

Nested loop approximately run **n(n+1)/2** times. $O(n^2)$

Example

```
for (i=0;i<n;++i)
        for (j=0;j<m;++j)
```

Nested loop approximately run **n*m** times. …

$O(n^2)$ given n>=m

Nested Loops Orders 1

Example 5

```
for(i=1;i<=n;++i)
        for (j=1;j<=i;++j)
```

$$\sum_{i=1}^{n} i = \frac{n(n+1)}{2} = O(n^2)$$

Nested loop approximately runs **n(n+1)/2** times.

(Arithmetic Series) $O(n^2)$

Nested Loops Orders 2

Example

```
for( i = 1; i < n; ++i )
        for( j = 1; j < n * n; ++j )
```

Approximately runs n^3 times. $O(n^3)$

	No of times loop runs
for(i = 1; i <= n; ++i)	$\sum_{i=1}^{n} 1 = n$
for(j = 1; j < i * i; ++j)	$\sum_{i=1}^{n} i^2$

$O(n^3)$ … **Arithmetic Series**

Nested Loop Orders

```
for(i=1; i<=n; i=i*2)
      for (j=1;j<=i;++j)
            sum+=1;
```

Nested loop run appr. 2n-1 times.

- Outer loop runs **lgn** times
- Inner loop runs 1 2 4 8 16 32 64 … times
- We need to sum up 1+2+4+8+16+32+64
- This forms a **Geometric series** sum up to lgn
- $1+2^1+2^2+2^3+2^4+2^5+2^6 \ldots\ldots 2^{lgn}$

$$\sum_{k=0}^{n} x^k = 1 + x + x^2 + \ldots + x^n = \frac{x^{n+1}-1}{x-1} (x \neq 1)$$

$$\sum_{x=1}^{lgn} 2^x = 1 + 2^1 + 2^2 + 2^3 + \ldots 2^{lgn} = \frac{2^{lgn+1} - 1}{2-1} = O(n)$$

$$\frac{2^{lg_2{}^n} - 1}{2-1} = \text{n-1}$$

Nested Loops Orders

Example

```
for(i=1;i<=n; i=i*2)
        for (j=1;j<=n;++j)
```

Outer loop runs O(lgn) times

Inner loop runs n times for each i … n+n+n+… +n … lgn times

Nested loop approximately run **O(nlgn)** times.

Nested Loops Orders

```
for(i=1;i<=n; i++)
        for (j=1;j<=i; j*2)
```

Outer loop runs n time and
inner lg1+lg2+lg3+…lgn times ..this is ***arithmetic series of lg***

$$\sum_{k=1}^{n} \lg k = n\mathbf{lg_2 n}$$

Nested Loop approx. runs **O(nlgn)** times.

Nested Loops Orders

Example

```
for(i=1;i<=n; i*2)
        for (j=1;j<=n; j*2)
```

Nested Loop approximately runs **$O(\lg_2 n)^2$** times.

Outer loop runs lgn time and inner lgn for each i

Linear Search

```
int LinearSearch(const int a[], int key, int n){
      for (int i = 0; i < n && a[i]!= key; i++);
      if (i == n)
              return -1;
      return i;
}
```

Unsuccessful Search: ➔ O(n)

Successful Search:

Best-Case: *item* is in the first location of the array ➔O(1)

Worst-Case: *item* is in the last location of the array ➔O(n)

Average-Case: The number of key comparisons 1, 2, ..., n

$$\frac{\sum_{i=1}^{n} i}{n} = \frac{(n^2+n)/2}{n}$$

➔ O(n)

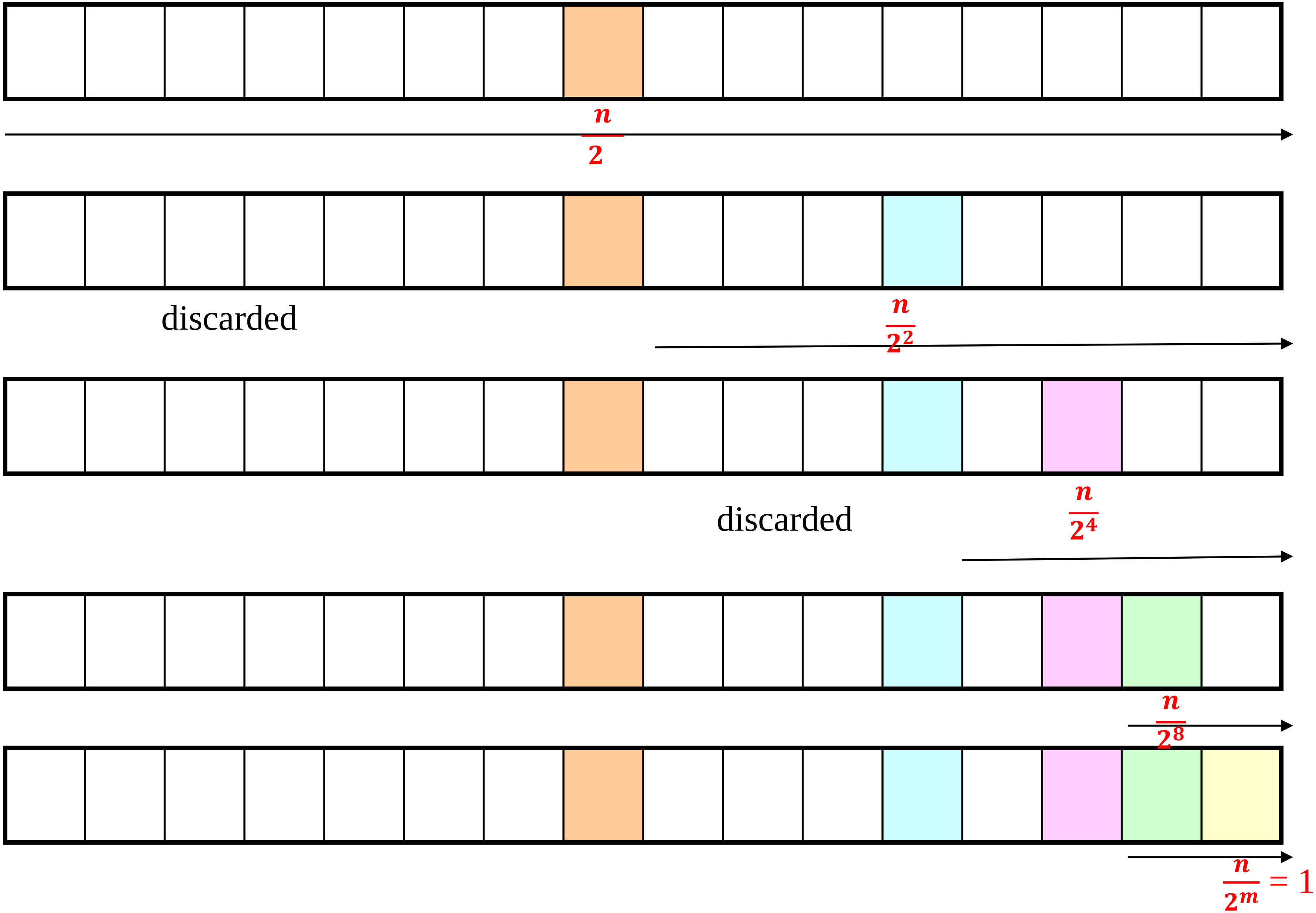
Types of Analysis

- **Worst case**
  - Provides an upper bound on running time (maximum number of steps)
  - An absolute guarantee that the algorithm would not run longer, no matter what the inputs are
- **Best case**
  - Provides a lower bound on running time (number of steps is the smallest)
  - Input is the one for which the algorithm runs the fastest

$$\boxed{\textit{Lower Bound} \leq \textit{Running Time} \leq \textit{Upper Bound}}$$

- **Average case**
  - Provides a prediction about the running time
  - Assumes that the input is random

EXAMPLE Binary search



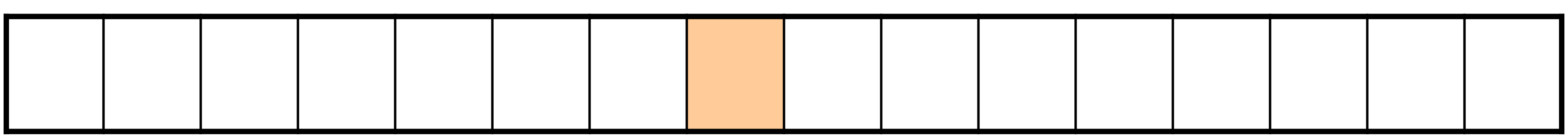
Binary Search – Analysis

- For an unsuccessful search:
  - The number of iterations in the loop is $\lfloor \log_2 n \rfloor + 1$

    $\rightarrow$ $O(\log_2 n)$
- For a successful search:
  - ***Best-Case:*** The number of iterations is 1. $\rightarrow$ $O(1)$
  - ***Worst-Case:*** The number of iterations is $\lfloor \log_2 n \rfloor + 1$ $\rightarrow$ $O(\log_2 n)$
  - ***Average-Case:*** The avg. # of iterations $< \log_2 n$ $\rightarrow$ $O(\log_2 n)$

0	1	2	3	4	5	6	7	$\leftarrow$ an array with size 8
3	2	3	1	3	2	3	4	$\leftarrow$ # of iterations

The average # of iterations = $21/8 < \log_2 8$



BUBBLE SORT

```
void BubbleSort(int arr[], int n) {

    bool done = false;
    for(i = 1; i < n; i++)        //repeat a pass of bubble sort
    {
      for (j=0; j <n-i; j++)//inner loop swaps consecutive items
      {
        if (arr[j+1] < arr[j]){
            swap(arr[j+1],arr[j])
         }
       }
    }
}
```

BUBBLE SORT

```
void BubbleSort(int arr[], int n) {
    bool done = false;
    for(i = 1; (i < n) && !done; i++){ //repeat a pass of bubble sort
        done = true;
        for (j=0; j <n-i; j++){   //inner loop swaps consecutive items

            if (arr[j+1] < arr[j]){
                swap(arr[j+1],arr[j])
                done = false;    //a swap is made and so sorting continues
            }
        }
    }
}
```

Best case $O(n)$

Worst case $O(n^2)$

SELECTION SORT

```
void SelectionSort(int arr[], int n) {
    for (i=0;i<n;++i){
        maxIndex = FindMaxIndex(arr,i,n-1);
        swap(arr[i],arr[maxIndex]);
    }
}
```

```
//finds the maximum item in the partial array startIndex to endIndex
int FindMaxIndex(int arr[], int startIndex, int endIndex){
        int max = startIndex
        for (i= startIndex; i<= endIndex;++i)
            if (a[max] < arr[i]) max =i;
      return max;
}
```

SPACE COMPLEXITY

- Space complexity is the amount of memory a program needs to run to completion
  - If program uses array of size n -> O(n) Space
  - IF program uses 2D array of size n*n -> O(n^2) Space

- Time complexity is the amount of computer time a program needs to run to completion

- [..\2. Lists.pptx](..\2. Lists.pptx)

Practice Questions

Euclids GCD

- 1 long gcd(long m, long n)
  - *2* {
  - *3* while(n != 0)
  - *4* {
    - *5* long rem = m % n;
    - *6* m = n;
    - *7* n = rem;
  - *8* }
  - *9* return m;
- *10 }*

EXAMPLES

- Summing an array of size n: $O(n)$
- Printing a matrix of size nxn: $O(n^2)$
- Summing two matrices of size nxn: $O(n^2)$
- Product of two matrices of size nxn: $O(n^3)$
- Linear search in array of size n: $O(n)$
- Binary search in array of size n: $O(\log n)$
- Printing all numbers that can be represented by n bits: $O(2^n)$
- Printing all subsets of numbers in an array of size n: $O(2^n)$
- Printing all permutations of numbers in array of size n: $O(n!)$