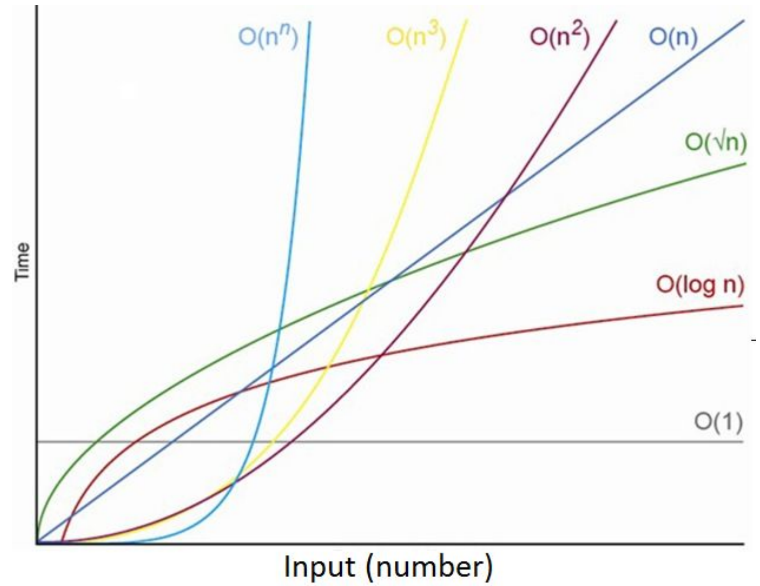


LECTURE 4

Example: Geometric series

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

Nested loop approximately run 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
- $1+2^1+2^2+2^3+2^4+2^5+2^6 \ldots\ldots 2^{\lg n}$

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

EXAMPLES
BASIC LOOP ORDERS

Simple Loop Orders

Example 0

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

Loop will run approximately **n** times

Example 1

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

Loop will run approximately **n/k** times

Example 2

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

Loop will run approximately **n/k** times

Example 3

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

Loop will run approximately $\mathbf{log_k n}$ times

Nested Loops Orders 1

Example 4

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

for (j=0;j<m;++j)

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

Example 5

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

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

$$\sum_{i=1}^{n}\sum_{j=1}^{i} 1 \quad \textit{...no of times inner loop runs}$$

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

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

Example 6

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

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

Nested loop approximately run **n(n+1)/2** times. **(Arithmetic Series)**

Nested Loops Orders 2

Example 11

```
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 Loops Orders 2

Example 7

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

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

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

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

Outer loop runs lgn time and inner loop runs $1 + 2^1 + 2^2 + 2^3 + \ldots 2^{lgn}$ …. *summation from 1 uptill lgn of geometric series*

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

Example 8

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

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

Outer loop runs 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 3

Example 8

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

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

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

Outer loop runs lgn time and inner lgn for each i

Example 9

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

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

Nested Loop approximately runs **(nlgn)** times. *Outer loop runs n time and inner lg1+lg2+lg3+…lgn times ..this is **arithmetic series of lg***

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

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

$$\boxed{\text{Lower Bound} \le \text{Running Time} \le \text{Upper Bound}}$$

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

Best, Worst and Average

- The *worst case* is when an algorithm requires a maximum number of steps
- The *best case* is when the number of steps is the smallest.
- The *average case* falls between these extremes.

- In simple cases, the average complexity is established by considering possible inputs to an algorithm, determining the number of steps performed by the algorithm for each input, adding the number of steps for all the inputs, and dividing by the number of inputs. This definition, however, assumes that the probability of occurrence of each input is the same, which is not always the case.

Linear Search

```
int LinearSearch(const int a[], int item, int n){
    for (int i = 0; i < n && a[i]!= item; 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

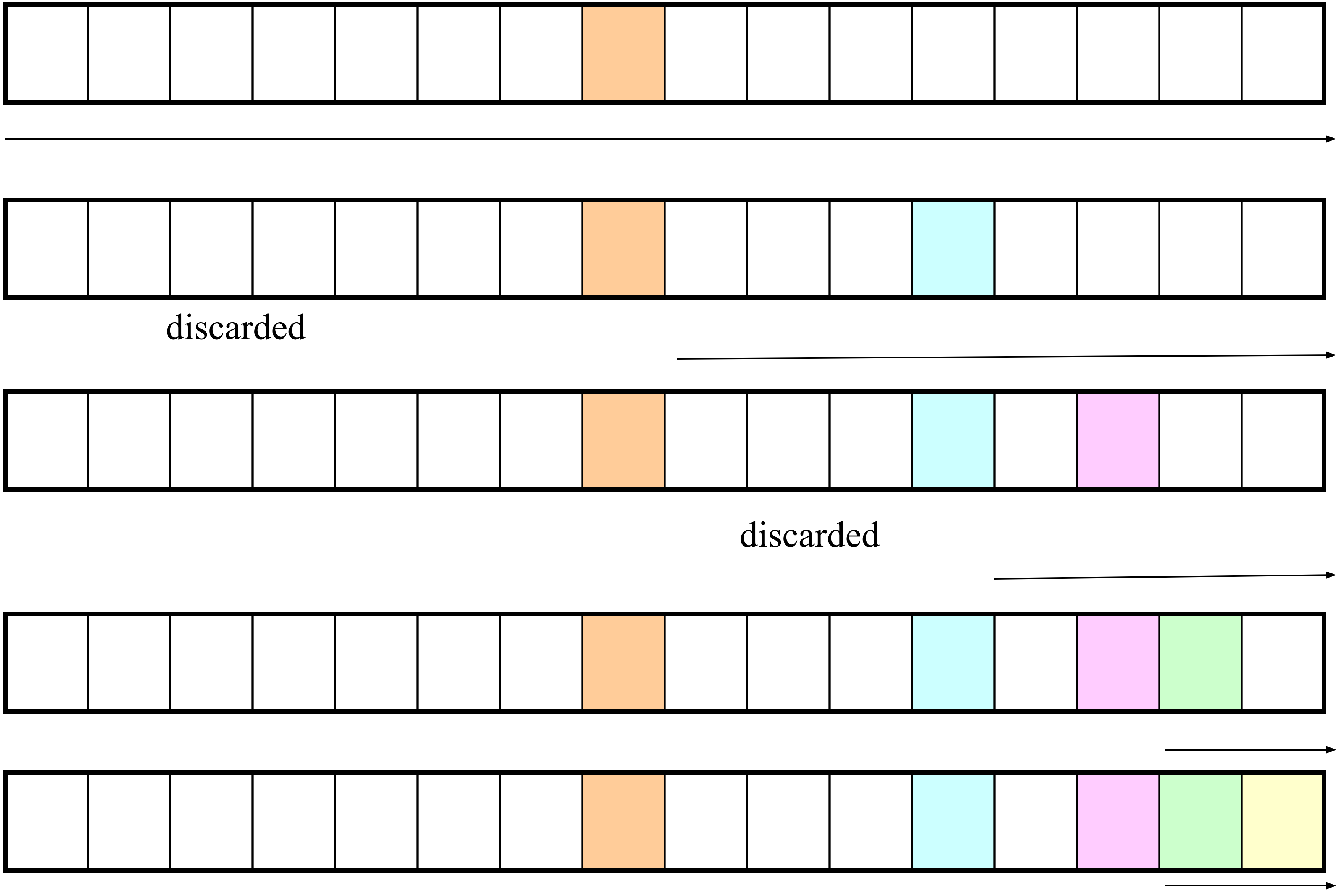
$$\frac{\sum_{i=1}^{n} i}{n} = \frac{(n^2+n)/2}{n}$$ □ O(n)

Binary Search- Best and worst

- template<class T>
- int binarySearch(const T arr[], int arrSize, const T& key) {
  - int lo = 0, mid, hi = arrSize-1;
  - while (lo <= hi) {
    - mid = (lo + hi)/2;
    - if (key < arr[mid])
      - hi = mid - 1;
    - else if (arr[mid] < key)
      - lo = mid + 1;
    - else return mid; // success: return the index of the cell occupied by key;
  - } return -1; // failure: key is not in the array;
- }

- If `key` is in the middle of the array, the loop executes only one time.
- How many times does the loop execute in the case where `key` is not in the array?
- *k* is not in the array can be determined after lg *n* iterations of the loop.

EXAMPLE Binary search



Binary Search – Analysis

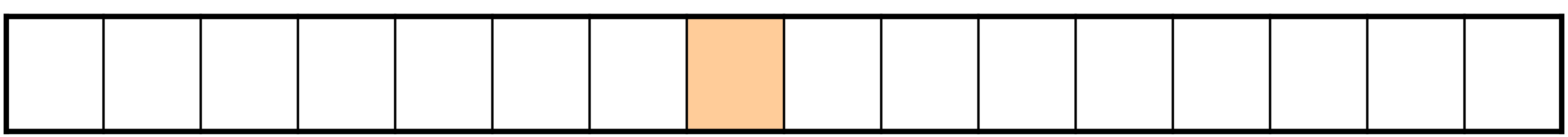
- For an unsuccessful search:
  - The number of iterations in the loop is $\lfloor \log_2 n \rfloor + 1$

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

```
0   1   2   3   4   5   6   7
```
□ an array with size 8

```
3   2   3   1   3   2   3   4
```
□ # of iterations

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



BUBBLE SORT

```
 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])
        }//end of  if
    }//end of inner for
}//end of outer for
```

BUBBLE SORT

```
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
        }//end of  if
    }//end of inner for
}//end of outer for
```

Best case $O(n)$

Worst case $O(n^2)$

SELECTION SORT

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

```
//FindMaxIndex(int arr[],int startIndex,int endIndex)
//finds the maximum item in the partial array
//from start index to end index
```

what is the complexity of the above???

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