# Course organization

- Introduction ( Week 1-2)
  - Course introduction
  - A brief introduction to molecular biology
  - A brief introduction to sequence comparison
- Part I: Algorithms for Sequence Analysis (Week 3 - 11)
  - Chapter 1-3, Models and theories
    - » Probability theory and Statistics (Week 4)
    - » Algorithm complexity analysis (Week 5)
    - » Classic algorithms (Week 6)
    - » Lab: Linux and Perl
  - Chapter 4, Sequence alignment (week 7)
  - Chapter 5, Hidden Markov Models ( week 8)
  - Chapter 6. Multiple sequence alignment (week 10)
  - Chapter 7. Motif finding (week 11)
  - Chapter 8. Sequence binning (week 11)
- Part II: Algorithms for Network Biology (Week 12 - 16)

# Chapter 2:
# Algorithm Complexity Analysis

Chaochun Wei

Fall 2014

# Contents

- Reading materials
- Why do we need to analyze the complexity of an algorithm?
  - Examples
- Introduction
  – Algorithm complexity
  – “Big O” notation: O()

# Reading

Cormen book:

Thomas, H. ,Cormen, Charles, E., Leiserson, and Ronald, L., Rivest . Introduction to Algorithms, The MIT Press.

(read Chapter 1 and 2, page 1-44).

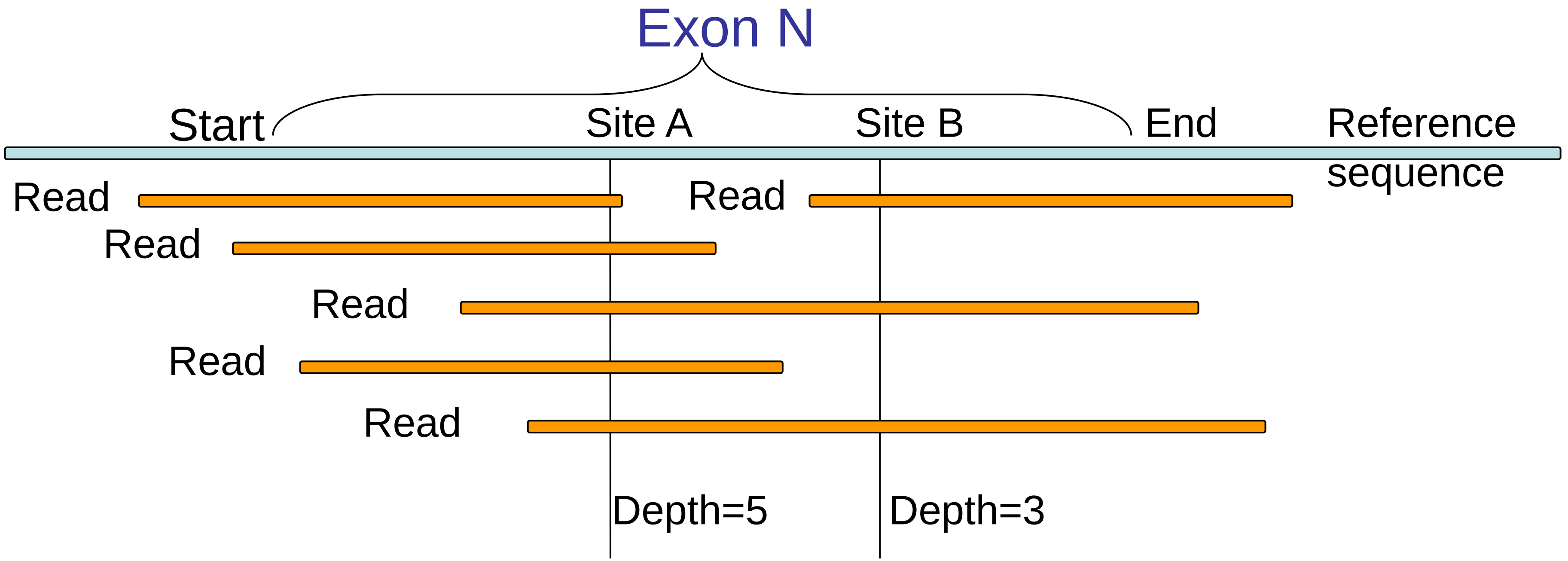
# A real example: Exon-capture data analysis

There are ~60 millions of short reads sequenced from exon regions of a human genome. We need to figure out the how many exons were covered with at least 10 reads.

Steps:

1. Reads are aligned to the genome;
2. Each alignment is checked to see the exon it covers;
3. For each exon, check the number of reads cover the exon;
4. For all exons, filter out those with read number < 10.

# A real example: Exon-capture data analysis



# A real example: Exon-capture data analysis

1 days later

Student: I have created a program to do the analysis. It’s running.

Teacher: Cool. Let me know when your analysis finishes.

# A real example: Exon-capture data analysis

6 days later…

Student: My program has been running for 5 days, and it keeps on running. I have no idea about what is happening and what to do with it.

Teacher: Its core is a sorting algorithm with a complexity of at most O(N*lgN). It should be done within a few minutes!

Student: What?.....

# Algorithm and its complexity

An **algorithm** is any well-defined computational procedure that takes in some **inputs** and produces some **outputs**.

Example: Sort an array of numbers

3, 2, 4, 5, 7, 1, 6 → 1, 2, 3,4, 5,6,7

# Algorithm and its complexity

An algorithm is any well-defined computational procedure that takes in some inputs and produces some outputs.

## Complexity: a function of input size

- Time complexity: the running time
- Space complexity: the memory size required

# Algorithm and its complexity

## Input size

- Number of items in the input
  - Sorting problem
  - FFT
- Total number of bits needed to represent the input
  - Arithmetic operation (+,-,x,/)
- The value of input
  - Factorial (N!)

## Multiple input sizes

- Need to specify which input size is used
  - Graph operation (number of Vertices, and edges)

# Algorithm and its complexity

## Before we start

- we use a generic one-processor, random-access machine.
  No parallel

# Algorithm and its complexity

Example: Sort an array of numbers
5, 2, 4, 6, 1, 3 → 1, 2, 3,4, 5,6

```
Insertion sort (A)
 for j = 2 to length(A)
      do key = A[j]
            /*insert A[j] into the sorted sequence A[1…j-1]
            i=j-1
             while i>0 and A[i]>key
                  do A[i+1]=A[i];
                        i=i-1;
                   A[i+1]=key;
```

# Algorithm and its complexity

Example: Sort an array of numbers

5, 2, 4, 6, 1, 3 → 1, 2, 3,4, 5,6

| | | | | | | |
|---|---|---|---|---|---|---|
| 5 | 2 | 4 | 6 | 1 | 3 | |
| 2 | 5 | 4 | 6 | 1 | 3 | |
| 2 | 4 | 5 | 6 | 1 | 3 | |
| 2 | 4 | 5 | 6 | 1 | 3 | |
| 1 | 2 | 4 | 5 | 6 | 3 | |
| 1 | 2 | 3 | 4 | 5 | 6 | ☺ |

# Algorithm and its complexity

Example: Sort an array of numbers

5, 2, 4, 6, 1, 3 → 1, 2, 3,4, 5,6

```
Insertion sort (A)
for j = 2 to length(A)
    do key = A[j]
        /*insert A[j] into the sorted sequence A[1…j-1]
        i=j-1
         while i>0 and A[i]>key
            do A[i+1]=A[i];
               i=i-1;
          A[i+1]=key;
```



Algorithm time complexity: $O(N^2)$

# Worst-case and average-case analysis

Example: Sort an array of numbers

5, 2, 4, 6, 1, 3 → 1, 2, 3,4, 5,6

```
Insertion sort (A)
for j = 2 to length(A)
      do key = A[j]
            /*insert A[j] into the sorted sequence A[1…j-1]
            i=j-1
            while i>0 and A[i]>key
                  do A[i+1]=A[i];
                       i=i-1;
                  A[i+1]=key;
```

**Can repeat from 0 to j times**



Algorithm time complexity: $O(N^2)$

# Order of growth

Example: Sort an array of numbers
5, 2, 4, 6, 1, 3 → 1, 2, 3,4, 5,6

## Insertion sort:

Algorithm run time complexity: $O(N^2)$
Order of growth: 2

# O-notation (big-O notation): Asymptotic upper bound

$O(g(n)) = \{f(n)$: there exist positive constants c and $n_0$ such that $0 \le f(n) \le c\, g(n)$ for all $n \ge n_0\}$

Note about O-notation operations:
$O(k_1 * N^2 + k_2 * N^3) = O(N^3)$ for constants $k_1$, $k_2$

# O-notation (big-O notation):
## Asymptotic upper bound

Example: Sort an array of numbers
5, 2, 4, 6, 1, 3 → 1, 2, 3,4, 5,6

Insertion sort:
algorithm time complexity: $O(N^2)$

# Sorting with time complexity of O(N*logN)

Example: Sort an array of numbers
5, 2, 4, 6, 1, 3 → 1, 2, 3,4, 5,6

```
Sort (A)
for j = 2 to length(A)
      do key = A[j]
            /*Use binary search to insert A[j]
            /*into the sorted sequence A[1…j-1]
            i=j-1

                Binary_search(A[j], A[1…j-1],)
```

# Sorting

Example: Sort an array of numbers

5, 2, 4, 6, 1, 3 → 1, 2, 3,4, 5,6

There are a lot of sorting algorithms:

- Heap sort (O(N*logN))
- Merge sort (O(N*logN))
- *Quick sort (worst-case $O(N^2)$, average O(N*logN))

# Merge sort

```
Merge-Sort (A, p, r)
        if p<r
                then q=[(p+r)/2]
                        Merge-Sort(A, p, q)
                        Merge-Sort(A,q+1,r)
                        Merge(A, p, q, r)
```

Time Complexity: $T(N) = \begin{cases} O(1); if\ N = 1 \\ 2T(N/2) + O(N); if\ N > 1 \end{cases}$

Solve it: T(N) = O(N*logN)

# Space complexity

Example: Sort an array of numbers

5, 2, 4, 6, 1, 3 → 1, 2, 3,4, 5,6

Need an array of size N: A[1…N], and 3 temporary variables
O(N)

Example: Sequence alignment

Need a two-dimension array of size N*M, and a constant number of temporary variables
O(N*M) or O(max(N, M))

# Other issues

- Output size
  - Blast: output can be a problem
- Input/Output method/place/mode
  - Speed
    - screen << hard disk << memory
- Programming language
  - Speed
    - Perl < java < C++ <C