WASHINGTON STATE UNIVERSITY

CPTS 223 Advanced Data Structure C/C++

Math Review: Basic

WSU

Why math?

- To **Analyze** data structures and algorithms
  - Deriving formulae for time and memory requirements
  - Will the solution scale (before we implement it)?
  - Quantify the results
  - Proving algorithm correctness

WSU

Examples: running time?

```
// Assume A is an
integer array of size n
```

Definition: (1) Let $T(n)$ denote the time take by an algorithm on an input of size n. (2) T(1)=1

```
Algorithm1 (A, n)
    max = infinity;
    for (i=1 to n) {
        if(A[i]>max) max=A[i];
    }
    Output max;
```

```
Algorithm2 (A, 1, n)
    if (n<2) return;
    mid = floor(n/2);
    if (condition#1)
        Algorithm2(A, 1, mid);
    else
        Algorithm2(A, mid+1, n);
```

```
Algorithm3 (A, 1, n)
    if (n<2) return;
    x = floor(n/2);
    T(1)=1;
    Algorithm3(A, 1, x)
    Algorithm3(A, x+1, n)
```

Is running time comparable?

WSU

Example: Algorithm 3

- Consider Algorithm3 that divides the input array in half and calls Algorithm3 recursively on each half

```
Algorithm3 (A, 1, n)
    if (n<2) return;
    x = floor(n/2);
    T(1)=1;
    Algorithm3(A, 1, x)
    Algorithm3(A, x+1, n)
```

T(n)

Constant time

T(n/2)

T(n/2)

T(n) = T(n/2) + T(n/2) + const
(but not a closed form)

Choose one:
A. n + const
B. n/2 + const
C. lg(n) + const
D. sqrt(n) + const
……

Example: Algorithm 3

- Consider Algorithm3 that divides the input array in half and calls Algorithm3 recursively on each half

```
Algorithm3 (A, 1, n)
    if (n<2) return;
    x = floor(n/2);
    T(1)=1;
    Algorithm3(A, 1, x)
    Algorithm3(A, x+1, n)
```

T(n)

Constant time

T(n/2)

T(n/2)

T(n)

= T(n/2) + T(n/2) + const

= 2T(n/2) + const

= 4T(n/4) + 2const + const

= 8 T(n/8) + (4+2+1) const

$= 2^K\ T\left(\frac{n}{2^K}\right) + const \sum_{k=0}^{K-1} 2^k$

Let $\frac{n}{2^K} = 1$ geometric series

= n T(1) + const * n

= n (1 + const) [Closed-form]

Examples: running time?

```
// Assume A is an
integer array of size n
```

Definition: (1) Let *T(n)* denote the time take by an algorithm on an input of size *n*. (2) T(1)=1

```
Algorithm1 (A, n)
    max = infinity;
    for (i=1 to n) {
        if(A[i]>max) max=A[i];
    }
    Output max;
```

```
Algorithm2 (A, 1, n)
    if (n<2) return;
    mid = floor(n/2);
    if (condition#1)
        Algorithm2(A, 1, mid);
    else
        Algorithm2(A, mid+1, n);
```

```
Algorithm3 (A, 1, n)
    if (n<2) return;
    x = floor(n/2);
    T(1)=1;
    Algorithm3(A, 1, x)
    Algorithm3(A, x+1, n)
```

T(n) = n (1 + const)

Is running time comparable?

WSU

Example: Algorithm 2

```
Algorithm2 (A, 1, n)
    if (n<2) return;
    mid = floor(n/2);
    if (condition#1)
        Algorithm2(A, 1, mid);
    else
        Algorithm2(A, mid+1, n);
```

- `if (n<2) return;` / `mid = floor(n/2);` → const
- `Algorithm2(A, 1, mid);` → T(n/2)
- `Algorithm2(A, mid+1, n);` → T(n/2)

T(n)
= T(n/2) + const
= T(n/4) + 2 const
= T(n/8) + 3 const
= $T\left(\frac{n}{2^K}\right) + const\ \sum_{k=0}^{K-1} 1$

Let $\frac{n}{2^K} = 1 \Leftrightarrow 2^K = n$

= T(1) + const * K
= 1 + const * K
= 1 + const * $log_2 n$

[Closed-form]

WSU

Examples: running time?

```
// Assume A is an
integer array of size n
```

Definition: (1) Let *T(n)* denote the time take by an algorithm on an input of size *n*. (2) T(1)=1

```
Algorithm1 (A, n)
    max = infinity;
    for (i=1 to n) {
        if(A[i]>max) max=A[i];
    }
    Output max;
```

```
Algorithm2 (A, 1, n)
    if (n<2) return;
    mid = floor(n/2);
    if (condition#1)
        Algorithm2(A, 1, mid);
    else
        Algorithm2(A, mid+1, n);
```

T(n) = 1 + const * $log_2 n$

```
Algorithm3 (A, 1, n)
    if (n<2) return;
    x = floor(n/2);
    T(1)=1;
    Algorithm3(A, 1, x)
    Algorithm3(A, x+1, n)
```

T(n) = n (1 + const)

Is running time comparable?

Example: Algorithm 1

```
Algorithm1 (A, n)
    max = infinity;
    for (i=1 to n) {
        if(A[i]>max) max=A[i];
    }
    Output max;
```

const

n

const

T(n)
= n T(1) + const
= n + const
[Closed-form]

Examples: running time?

```
// Assume A is an
integer array of size n
```

Definition: (1) Let *T(n)* denote the time take by an algorithm on an input of size *n*. (2) T(1)=1

```
Algorithm1 (A, n)
    max = infinity;
    for (i=1 to n) {
        if(A[i]>max) max=A[i];
    }
    Output max;
```

T(n) = n + const

```
Algorithm2 (A, 1, n)
    if (n<2) return;
    mid = floor(n/2);
    if (condition#1)
        Algorithm2(A, 1, mid);
    else
        Algorithm2(A, mid+1, n);
```

T(n) = 1 + const * $log_2 n$

```
Algorithm3 (A, 1, n)
    if (n<2) return;
    x = floor(n/2);
    T(1)=1;
    Algorithm3(A, 1, x)
    Algorithm3(A, x+1, n)
```

T(n) = n (1 + const)

Is running time comparable?

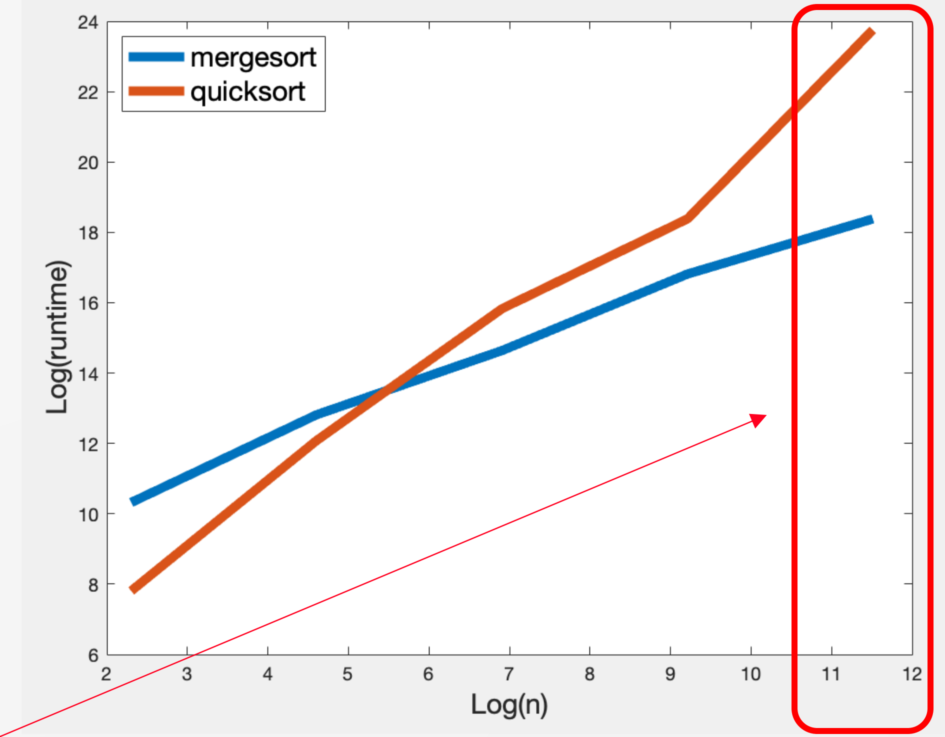
WSU

Comparison of running time

Recall the Mergesort V.S. Quicksort example:

n (input size)	Mergesort	Quicksort
10	30375	2458
100	367666	176750
1000	2280125	7493833
10000	20054042	96236458
100000	96236458	20707570875

Running time (in ns)



Our focus: scaled-up time

Floor and ceiling

- floor(x), denoted $\lfloor x \rfloor$, is the greatest integer ≤ x
- ceiling(x), denoted $\lceil x \rceil$, is the smallest integer ≥ x
- Normally used to divide input into integral parts
  - floor(N/2) + ceiling(N/2) = N

WSU

Exponents

- $X^A X^B = X^{A+B}$
- $X^A / X^B = X^{A-B}$
- $(X^A)^B = X^{AB}$
- $X^A + X^A = 2X^A \neq X^{2A}$
- $2^A + 2^A = 2^{A+1}$

WSU

Logarithms

- $\log_X B = A \Leftrightarrow X^A = B$ (logarithm of B base X)
- $\log_A B = \log_C B$ / $\log_C A$, where $A, B, C > 0, A \neq 1$
- $\log_X A + \log_X B = \log_X AB$, where $A, B > 0$,
- $\log_X \left(\frac{A}{B}\right) = \log_X A - \log_X B$
- $\log_X (A^B) = B \log_X A$
- $\log_X A < A$ for all $A > 0$
- $\lg A = \log_2 A$ (In Weiss book, log n → $\log_2 n$)
- $\ln A = \log_e A$ where $e = 2.7182\ldots$ (natural logarithm)

Logarithms

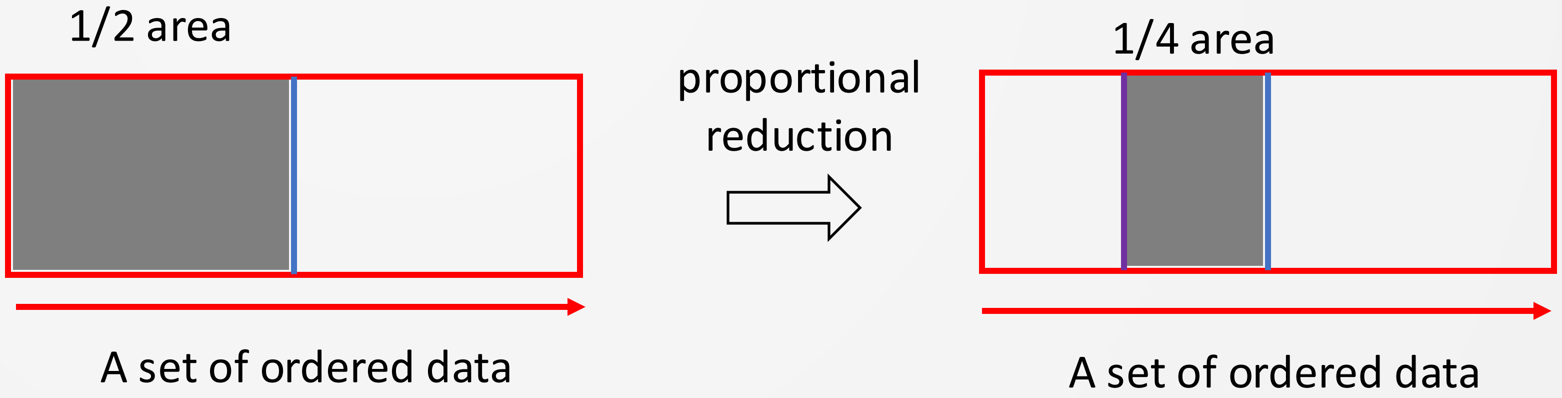
- What is the meaning of the log function?
  - For example, lg 1024 = 10
  - 2^10=1024

Logarithms

- How many times to halve an array of length n until its length is 1?

```
KeepHalving (n)
    i = 0
    while (n ≠ 1)
    {
        i = i + 1
        n = floor(n/2)
    }
    return i
```

What will be the value of i?



WSU

Factorials

- Definition $n! = \begin{cases} 1 \text{ if } n = 0 \\ n * (n-1)! \text{ if } n > 0 \end{cases}$
- Stirling's approximation: $n! \approx \sqrt{2\pi n}\left(\frac{n}{e}\right)^n (1 + \theta(1/n))$
- Physical explanation:
  - $n! =$ how many ways to order a set of n elements

    1 2 3
    1 3 2
    2 1 2
    3 1 3
    2 3 1
    3 2 1

$n! < n^n$

$n! = n * (n-1) * \cdots * 1$
$n^n = n * n * \cdots * n$

helps simplify $n!/n^n$ in complexity analysis

WSU

Modular Arithmetic

- $A\ mod\ N = A - N * \lfloor A/N \rfloor$ (remainder)
- $(A\ mod\ N) = B\ mod\ N \Rightarrow A \equiv B(mod\ N)$
  - A is congruent to B modulo N
  - e.g., $81 \equiv 61 \equiv 1(mod\ 10)$
    - $81 - 10 * \left\lfloor \frac{81}{10} \right\rfloor = 81 - 10 * \lfloor 8.1 \rfloor = 81 - 10 * 8 = 1$
    - $61 - 10 * \left\lfloor \frac{61}{10} \right\rfloor = 61 - 10 * \lfloor 6.1 \rfloor = 61 - 10 * 6 = 1$
- If $A \equiv B(mod\ N)$, then:
  - $A + C \equiv B + C(mod\ N)$
  - $AD \equiv BD(mod\ N)$

Basis of most encryption schemes: (Message mod Key)

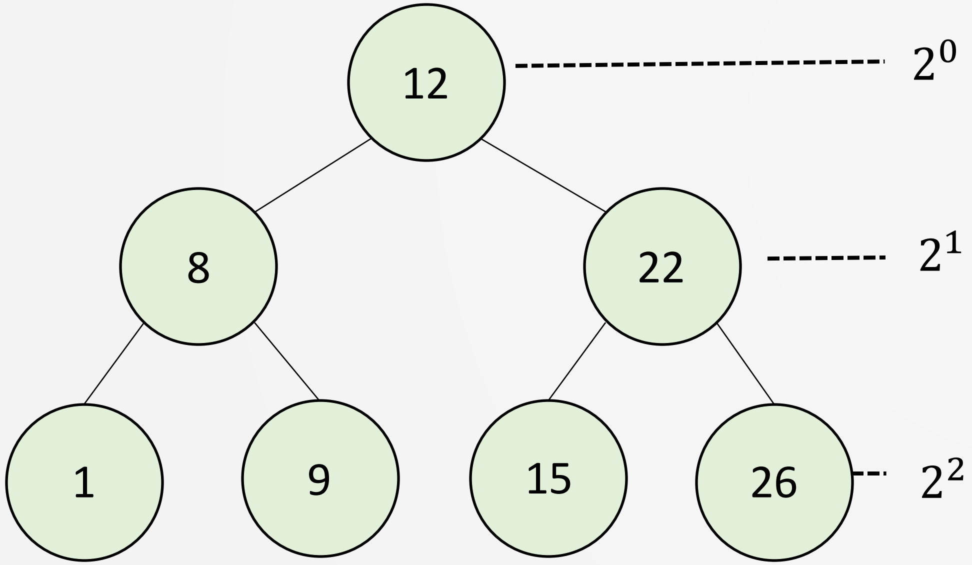
WSU

Series

- General
  - $\sum_{i=0}^{N} f(i) = f(0) + f(1) + \cdots + f(N)$
- Linearity
  - $\sum_{i=0}^{N}\big(cf(i) + g(i)\big) = c\sum_{i=0}^{N} f(i) + \sum_{i=0}^{N} g(i)$
- Arithmetic series
  - $\sum_{i=0}^{N} i = N(N+1)/2$
- Geometric series
  - $\sum_{i=0}^{N} A^i = \frac{A^{N+1}-1}{A-1}$
  - $\sum_{i=0}^{N} A^i \le \sum_{i=0}^{\infty} A^i = \frac{1}{1-A}$ for $0 \le A \le 1$

WSU

Example of geometric series



Total number of elements in a complete binary search tree (plug in $N = 2, A = 2$):

$$\sum_{i=0}^{N} A^i = \frac{A^{N+1} - 1}{A - 1} = \frac{2^{2+1} - 1}{2 - 1} = 7$$

Can be generalized to any integer N as the height