Logarithms

- Involved in many important runtime results: Sorting, binary search etc.
- Logarithms grow slowly, much more slowly than any polynomial but faster than a constant.
- Definition: \( \log_B N = K \) if \( B^K = N \). B is the base of the log.
- Examples:
  - \( \log_2 8 = 3 \) because \( 2^3 = 8 \).
  - \( \log_{10} 100 = 2 \) because \( 10^2 = 100 \).
  - \( 2^{10} = 1024 \) (1K), so \( \log_2 1024 = 10 \).
  - \( 2^{20} \approx 1M \), so \( \log_2 1M \approx 20 \).
  - \( 2^{30} \approx 1G \) so \( \log_2 1G = 30 \).

Useful Logarithm Rules

- It requires \( \log_N K \) digits to represent \( K \) numbers in base \( N \).
- It requires approx. \( \log_2 K \) multiplications by 2 to get from 1 to \( N \).
- It requires approx. \( \log_2 K \) divisions by 2 to get from \( N \) to 1.
- Computers work in binary, so in order to calculate how many numbers can represent a certain amount of memory we use \( \log_2 \).

Useful Logarithm Rules

- 16 bits of memory can represent \( 2^{16} \) different numbers = \( 2^{10} \cdot 2^6 = 64K \).
- 22 bits of memory can represent \( 2^{22} \) different numbers = \( 2^{20} \cdot 2^2 \approx 4G \) (see previous slide). Many of 1990’s operating systems’ addresses were 32 bits, so only 4 GB of addressable address space, became cramped.
- 64 bits (most of today’s computers’ addresses).

Useful Logarithm Rules

- \( \log(\text{nm}) = \log(n) + \log(m) \)
- \( \log(n/m) = \log(n) − \log(m) \)
- \( \log(n^k) = k \log(n) \)
- \( \log_b(a) = \frac{\log_e(a)}{\log_e(b)} \)

If the base of \( \log \) is not specified, assume it is base 2 (although for runtime analysis it doesn’t matter)

- \( \log \): base 2
- In: base e
When we develop an algorithm, we want to know how many resources it requires.

Let $T$ and $N$ be positive numbers. $N$ is the size of the problem, and $T$ measures a resource: Runtime, CPU cycles, disk space, memory etc.

Order of growth can be important. For example—sorting algorithms can perform quadratically or as $n \log(n)$. Very big difference for large inputs.

We care less about constants, so $100N = O(N)$, $100N + 300 = O(N)$. Very big difference for large inputs.

Constant can be important when choosing between two similar run-time algorithms. Example—quicksort.

* It is not always 100% clear what the "size of the problem" is. More on that later.

---

**Runtime Analysis**

- $T(N)$ is $O(F(N))$ if there are positive constants $c$ and $N_0$ such that $T(N) \leq c F(N)$ for all $N \geq N_0$.
- $T(N)$ is $\Omega(F(N))$ if there are positive constants $c$ and $N_0$ such that $T(N) \geq c F(N)$ for all $N \geq N_0$.
- i.e., $T(N)$ is bounded by a multiple of $F(N)$ from above for every big enough $N$.
- Example—Show that $2N + 4 = \Omega(N)$
- This means—find actual $c$ and $N_0$ (there is more than one correct answer).

---

**Runtime Analysis — Big-O Notation**

- $T(N)$ is $O(F(N))$ if there are positive constants $c$ and $N_0$ such that $T(N) \leq c F(N)$ for all $N \geq N_0$.
- Example—Show that $2N + 4 = \Omega(N)$
- This means—find actual $c$ and $N_0$ (there is more than one correct answer).

---

**Runtime Analysis — \Omega Notation**

- $T(N) = \Omega(F(N))$ if there are positive constants $c$ and $N_0$ such that $T(N) \geq c F(N)$ for all $N \geq N_0$.
- Example—Show that $2N + 4 = \Omega(N)$
- This means—find actual $c$ and $N_0$ (there is more than one correct answer).

---

**~ Notation in S&W**

- Note that although $O$ is the standard notation for expressing growth of functions in C.S., S&W prefer the applied mathematician’s $\sim$ notation.
- How are they related?
- Very simply. In the usual case that $O(F(N)) = O(F(N)) = \Omega(N)$, the case of an "asymptotically tight bound", we can say $F(N) \sim \Omega(N)$.
- For our previous example, we had $T(N) = 2N + 4 = \Omega(N)$ and $T(N) = \Omega(N)$ also, so we can say $T(N) \sim N$.

---

**Can \sim do the whole job?**

- How about an example that doesn’t qualify for $\sim$...
- Here’s one: $T(N) = N \cdot \text{sin}(N)$. Since $\text{sin}(N) \leq 1$ for all $N$, we see that $T(N) = O(N)$. But $\text{sin}(N)$ goes 0 and negative periodically, so no (positive) function qualifies as $\Omega(N)$.
- You can see that we had to reach pretty far to get an example. All polynomial functions have asymptotically tight bounds, as do other combinations of logs and powers.
- The function has to wiggle or jump around a lot to avoid asymptotic expression.
Runtime Analysis

- When the runtime is estimated as a polynomial we care about the leading term only.
- Thus \( 3n^2 + 2n + 17 = O(n^2) \) because eventually the leading cubic term is bigger than the rest.
- For a good estimate on the runtime it’s good to have both the \( O \) and the \( \Omega \) estimates (upper and lower bounds).
- But we often settle for the big-\( O \) bound, or \(-\) expression.

Useful Examples

<table>
<thead>
<tr>
<th>Function</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \log N )</td>
<td>Logarithmic</td>
</tr>
<tr>
<td>( \log^2 N )</td>
<td>Log-squared</td>
</tr>
<tr>
<td>( N )</td>
<td>Linear</td>
</tr>
<tr>
<td>( N \log N )</td>
<td>Linearithmic</td>
</tr>
<tr>
<td>( N^2 )</td>
<td>Quadratic</td>
</tr>
<tr>
<td>( N^3 )</td>
<td>Cubic</td>
</tr>
<tr>
<td>( 2^N )</td>
<td>Exponential</td>
</tr>
</tbody>
</table>

(N \( \log N \) is called “linearithmic”, page 187)

Adding and Multiplying Functions

- Rule for sums (e.g.: two consecutive blocks of code): If \( T_1(N) = O(F(N)) \) and \( T_2(N) = O(G(N)) \) then \( T_1 + T_2 = O(\max(F(N), G(N))) \). The biggest contribution dominates the sum. Same for • here and below.
- Rule for products (e.g.: an inner loop run by an outer loop): If \( T_1(N) = O(F(N)) \) and \( T_2(N) = O(G(N)) \) then \( T_1 \times T_2 = O(F(N) \times G(N)) \).

Example:

\( n^2 + 2n + 17 = 2n^2 + n + 17 = O(n^2) \).

(Example: Sum \( i \) from 1 to \( N \). Average size of an element: \( \frac{2}{3} \times N \). There are \( N \) terms so the sum is \( O(N^2) \). Exact form: \( \frac{N(N+1)}{2} \).)

Loops

- The runtime of a loop is the runtime of the statements in the loop • number of iterations.

Example: bubble sort

```c
/* sort array of ints in A[0] to A[n-1] */
int bubblesort(int A[], int n)
{
    int i, j, temp;
    /* \( n \) passes of loop */
    for (i = 0; i < n-1; i++) /* \( n-1 \) passes of loop */
        for (j = 0; j < n-i-1; j++) /* \( n-i-1 \) passes of loop */
               A[j] = A[j+1];
               A[j+1] = temp; */
```

Runtime Table

<table>
<thead>
<tr>
<th>( n )</th>
<th>( n \log n )</th>
<th>( n^2 )</th>
<th>( n^3 )</th>
<th>( 2^n )</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>2</td>
<td>4</td>
<td>8</td>
<td>4</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>9</td>
<td>27</td>
<td>8</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>16</td>
<td>64</td>
<td>16</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
<td>25</td>
<td>125</td>
<td>32</td>
</tr>
<tr>
<td>10</td>
<td>10</td>
<td>100</td>
<td>1000</td>
<td>1024</td>
</tr>
</tbody>
</table>

Loops

- Work from inside out:
  - Calculate the body of inner loop (constant – one statement and three assignments).
  - Estimate the number of passes of the inner loop: \( m \) passes.
  - Estimate the number of passes of the outer loop: \( n \) passes.
  - Each pass counts \( n \times m \), \( n \times m \) passes of constant operations:
  \( \sum_{i=0}^{n} (m - i) \times \sum_{j=0}^{m} (n - j) \times c = O(n^2 m^2) \).

- This is not the fastest sorting algorithm but it’s simple and works in-place. Good for small size input.

- We’ll talk a bit about sorting later on (but only briefly. It was CS210 material).
Recursive Functions

- Recursive functions perform some operations and then call themselves with a different (usually smaller) input.
- Example: factorial.

```java
int factorial(int n)
{
  if(n==0) return 1;
  return n*factorial(n-1);
}
```

Recursive Analysis

- Let us define \( T(n) \) as a function that measures the runtime.
- \( T(n) \) can be polynomial, logarithmic, exponential etc.
- \( T(n) \) may not be given explicitly in closed form, especially in recursive functions (which lend themselves easily to this kind of analysis).
- We have to find a way to derive the closed form from the recurrence formula.

```
T(n) = T(n-1) + T(n-2)
```

\( T(1) = 1 \) and \( T(2) = 1 \)

Let us denote the run-time on input \( n \) as some function \( T(n) \) and analyze \( T(n) \).

\( O(1) \) operations before recursive call – if statement and a multiplication.

The recursive part calls the same function with \( n - 1 \) as input, so this part runs \( T(n-1) \). 

So: \( T(n) = c + T(n-1) \).

Similarly: \( T(n-1) = c + T(n-2) \) \( T(n) = 2c + T(n-2) \).

After \( n \) such equations we reach \( T(1) = k \) (just the if-statement).

\( T(n) = (n - 1) \cdot c + k = O(n) \).

Iterative function performs the same.

A Problematic Example

- The well known Fibonacci series, where each number is the sum of the previous two numbers: 0 1 1 2 3 5 8 13 ...
- \( f(n) = f(n-1) + f(n-2) \), where \( f(0) = 0 \), \( f(1) = 1 \)
- This is a recursive definition.
- The following recursive program calculates the \( n \)th term in the Fibonacci series (assume \( n \) is non-negative and the first term is the zero-th):

```java
int fib(int n)
{
  if(n == 0) return 0;
  if(n == 1) return 1;
  return fib(n-2)+fib(n-1);
}
```

What is the problem here?

Ill-Behaved Recursion – Illustration

The problem is the double recursion which runs on the same input and do a lot of redundant work.

The call tree looks like a big binary tree.

Two or more recursive calls are not necessarily bad, as long as we split the work too.

Example: Merge sort – sort recursively two halves of an array and merge.

Call recursively twice, but on different input! The work is split between recursive calls in a smart way.

The exact runtime is \( O(n \log n) \). The full analysis is beyond the scope for now.

But it is exponential! (remember the illustration above).

How do we fix the fibonacci program?

Ill-Behaved Recursion

- The problem is the double recursion which runs on the same input and do a lot of redundant work.
- The call tree looks like a big binary tree.
- Two or more recursive calls are not necessarily bad, as long as we split the work too.
- Example: Merge sort – sort recursively two halves of an array and merge.
- Call recursively twice, but on different input! The work is split between recursive calls in a smart way.
- The exact runtime is \( O(n \log n) \). The full analysis is beyond the scope for now.
- But it is exponential! (remember the illustration above).
- How do we fix the fibonacci program?
**Binary Search**

- **Definition**: Search for an element in a sorted array.
- **Return**: array index where element is found or a negative value if not found.
- **Implemented in Java as part of the Collections API.**
- **Start**: in the middle of the array.
- **If** the element is smaller than that, search in the smaller half. Otherwise, search in the larger half.

## Binary Search Example

<table>
<thead>
<tr>
<th>Key</th>
<th>List</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>3/4</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
</tr>
</tbody>
</table>

---

**Binary Search Implementation**

```java
// Hidden recursive routine.
private static <T extends Comparable<? super T>> int binarySearch(T[] a, int low, int high)
{
    if (low > high)
        return NOT_FOUND;
    int mid = (low + high) / 2;
    if (a[mid].compareTo(x) < 0)
        return binarySearch(a, mid + 1, high);
    else if (a[mid].compareTo(x) > 0)
        return binarySearch(a, low, mid - 1);
    else
        return mid;
}
```

**Binary Search**

- **What is that <(?super?)> clause?**
- **The Comparable<?)super? > specifies that T ISA Comparable<Y>, where Y is T or any superclass of T.**
- **This allows the use of a compareTo implemented at the top of an inheritance hierarchy (i.e., in the base class) to compare elements of an array of sub class elements.**
- **For example, we commonly use an unique id for equals, hashCode and compareTo across a hierarchy, and only want to implement it once in the base class.**

**Binary Search Algorithm Runtime**

You should be able to guess this one out by now (I hope):  
$$T(N) = T(N/2) + O(1)$$  
$$T(N) = O(logN)$$

---

**Binary Search Implementation**

```java
static <T> int binarySearch(T[] a, Key, Comparator<? super T> c)
static int binarySearch(Object[] a, Object key)
```

The version without the Comparator uses "natural order" of the array elements, i.e., calls compareTo of the element type to compare elements.

Thus the elements need to be Comparable -- the element type implements Comparable<ElementType> in the generics setup.

Or the old Comparable works here too.
MergeSort Performance

- T(1) = 1, T(2) = 1
- T(n) = T(n/2) + T(n/2) + O(n)
- ceiling(log n)

MergeSort Algorithm

1. If n <= 1, return.
2. Divide the array into two halves.
3. Recursively sort the halves.
4. Merge the sorted halves.

Recurrence

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

Analysis

- The recurrence is log n terms.
- The master theorem applies.
- T(n) = O(nlog n)
Recursion Tree

- If \( N = 2^n \) then there are \( p \) rows with \( c_n \) on the right, and one last row with \( d_n \) on the right.
- Since \( p = \log n \), this means that the total cost is \( cv \log N + dv \).
  In other words, this is what we call an \( O(N \log N) \) algorithm.

Another Way to Look at Runtime

- What does "linear runtime" really mean?
- A linear function (program, algorithm) requires resources that scale linearly with the input size.

Another Way to Look at Runtime

- What does "linear runtime" really mean?
- A linear function (program, algorithm) requires resources that scale linearly with the input size.

Best, Worst, and Average-Case Analysis

- Best case: the minimum time for any instance of size \( n \)
- Worst case: the maximum time for any instance of size \( n \)
  - Unless otherwise specified, \( O(f(n)) \) means the worst case runtime
- Average case: the average time for all instances of size \( n \)
- Successful sequential search:
  - Average case: \( O(n) \)
  - Worst case: \( O(n) \)
- Unsuccessful sequential search: \( O(n) \)
- Successful binary search:
  - Average case: \( O(\log n) \)
  - Worst case: \( O(\log n) \)
- Unsuccessful binary search: \( O(\log n) \)