CS310 - Advanced Data Structures and Algorithms

Fall 2017 – Runtime

October 2, 2017
Kleinberg and Tardos, chapter 2, Sedgewick and Wayne, chapter 1.4.

Remember recursion, basic runtime analysis.
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 \( \log_2 1024 = 10 \).
  - \( 2^{20} = 1M \), so \( \log 1M = 20 \).
  - \( 2^{30} = 1G \) so \( \log 1G = 30 \).
Logarithms

- 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 a certain amount of memory can represent we use $\log_2$
16 bits of memory can represent $2^{16}$ different numbers
$= 2^{10+6} = 2^{10} \times 2^6 = 64K$.

32 bits of memory can represent $2^{32}$ different numbers
$= 2^{30+2} = 2^{30} \times 2^2 = 4G$ – see previous slide. (many of today’s operating systems address space).

64 bits?? (most of today’s computers address space).
Useful Logarithm Rules

- \( \log(nm) = \log(n) + \log(m) \)
- \( \log(n/m) = \log(n) - \log(m) \)
- \( \log(n^k) = k \log(n) \)
- \( \log_a(b) = \frac{\log b}{\log a} \)

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

- \( \log: \) base 2
- \( \ln: \) 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 $n \cdot \log(n)$. Very big difference for large inputs.

We care less about constants, so $100N = O(N)$.

$100N + 200 = O(N)$.

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.
- $T(N)$ is $O(F(N))$ if there are positive constants $c$ and $N_0$ such that $T(N) \leq c \times F(N)$ for all $N \geq N_0$.
- $T(N)$ is bounded by a multiple of $F(N)$ from above for every big enough $N$.
- Example – Show that $2N + 4 = O(N)$
- This means – find actual $c$ and $N_0$ (there is more than one correct answer).
• $T(N)$ is $\Omega(F(N))$ if there are positive constants $c$ and $N_0$ such that $T(N) \geq c \times F(N)$ for all $N \geq N_0$.
• $T(N)$ is bounded by a multiple of $F(N)$ from below 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).
When the runtime is estimated as a polynomial we care about the leading term only.

Thus $3n^3 + n^2 + 2n + 17 = O(n^3)$ 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).

$\Theta$ is both upper and lower bound – if $f(n) = \Theta(g(n))$ then they are equivalent as far as runtime is concerned.

It does NOT mean that they are equal!
### Useful Nomenclature

<table>
<thead>
<tr>
<th>Function</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>(c)</td>
<td>Constant</td>
</tr>
<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>(N \log N)</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>
<tr>
<td>n</td>
<td>$f(n)$</td>
</tr>
<tr>
<td>-----</td>
<td>--------</td>
</tr>
<tr>
<td>10</td>
<td></td>
</tr>
<tr>
<td>20</td>
<td></td>
</tr>
<tr>
<td>30</td>
<td></td>
</tr>
<tr>
<td>40</td>
<td></td>
</tr>
<tr>
<td>50</td>
<td></td>
</tr>
<tr>
<td>100</td>
<td></td>
</tr>
<tr>
<td>$10^3$</td>
<td></td>
</tr>
<tr>
<td>$10^4$</td>
<td></td>
</tr>
<tr>
<td>$10^5$</td>
<td></td>
</tr>
<tr>
<td>$10^6$</td>
<td></td>
</tr>
<tr>
<td>$10^7$</td>
<td></td>
</tr>
<tr>
<td>$10^8$</td>
<td></td>
</tr>
<tr>
<td>$10^9$</td>
<td></td>
</tr>
</tbody>
</table>
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.

- **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**: 
  \[(n^2 + 2n + 17) \times (2n^2 + n + 17) = O(n^2 \times n^2) = O(n^4)\]. (Remember to ignore all but the leading term).

- If we sum over a large number of terms, we multiply the number of terms by the estimated size of one term.

- **Example**: Sum of \( i \) from 1 to \( N \). Average size of an element: \( \frac{N}{2} \). There are \( N \) terms so the sum is \( O(N^2) \). Exact term: \( \frac{N \times (N-1)}{2} \).
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;
    for(i = 0; i < n-1; i++) /* n passes of loop */
        /* n-i passes of loop */
        for(j = n-1; j > i; j--)
                temp = A[j-1];
                A[j] = temp;
            }
}
```
Loops

- Work from inside out:
  - Calculate the body of inner loop (constant an if statement and three assignments).
  - Estimate the number of passes of the inner loop: n-i passes.
  - Estimate the number of passes of the outer loop: n passes. Each pass counts \( n, n-1, n-2, \ldots, 1 \).
  - Overall \( 1 + 2 + 3 + \ldots + n \) passes of constant operations:
    \[ \frac{n(n-1)}{2} = O(n^2). \]

- 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.

```c
int factorial (int n)
{
    if(n<=1) return 1;
    return n*factorial(n-1);
}
```
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.
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: $T(n - 1) = c + T(n - 2) \Rightarrow T(n) = 2c + T(n - 2)$.

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.
The well known Fibonacci series, where each number is the sum of the previous two numbers: 0 1 1 2 3 5 8 13 ...

\[ f(n) = f(n - 1) + f(n - 2), \text{ where } f(0) = 0, f(1) = 1 \]

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):

```c
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

\[ T(n) \]

\[ T(n - 1) \quad T(n - 2) \]

\[ T(n - 2) \quad T(n - 3) \quad T(n - 3) \quad T(n - 4) \]
The problem is the double recursion which runs on the same input so we 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 $O(1.618^n)$. The full analysis is beyond the scope for now.

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

How do we fix the fibonacci program?
- 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

Key: 8
List: 1 2 3 4 5 6 7 8 9

8 > 4
Key: 8
List: 1 2 3 4 5 6 7 8 9

8 > 6
Key: 8
List: 1 2 3 4 5 6 7 8 9

8 = 8
Key: 8
List: 1 2 3 4 5 6 7 8 9
static <T> int binarySearch(T[] a, T 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.
private static <AnyType extends Comparable<? super AnyType>>
int binarySearch( AnyType [] a, AnyType x, int low, int high )
{
    if( low > high )
        return NOT_FOUND;

    int mid = ( low + high ) / 2;

    if( a[ mid ].compareTo( x ) < 0 )
        return binarySearch( a, x, mid + 1, high );
    else if( a[ mid ].compareTo( x ) > 0 )
        return binarySearch( a, x, low, mid - 1 );
    else
        return mid;
}
What is that `<superT>` clause?

The `Comparable` `<superT>` specifies that `T ISA Comparable < Y >`, where `Y` is `T` or any superclass of it.

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 subclass elements.

For example, we commonly use a unique id for equals, `hashCode` and `compareTo` across a hierarchy, and only want to implement it once in the base class.
You should be able to guess this one out by now (I hope):

\[ T(N) = T(N/2) + O(1) \]

\[ T(N) = O(\log N) \]
Mergesort Recurrence Formula

\[ T(n) = \begin{cases} 
C & \text{if } n \text{ is 1} \\
2 \times T\left(\frac{n}{2}\right) + cn & \text{otherwise}
\end{cases} \]

Notice that c and C are not the same constant!
Identities like this come up frequently in algorithmic analysis.

It’s important to have ways of solving them. We’ll see a couple.

One basic way is to form a recursion tree.

Mergesort is a good example:

1. If the array has at most one item – return.
2. Split it in half, call merge sort recursively on each half.
3. Merge the two sorted halves.
The Mergesort Algorithm

```java
public static <AnyType extends Comparable<? super AnyType>>
    void mergeSort(AnyType[] a)
{
    AnyType[] tmpArray = (AnyType[]) new Comparable[a.length];
    mergeSort(a, tmpArray, 0, a.length - 1);
}

// Internal method that makes recursive calls.
private static <AnyType extends Comparable<? super AnyType>>
    void mergeSort(AnyType[] a, AnyType[] tmpArray,
                   int left, int right)
{
    if( left < right ) {
        int center = ( left + right ) / 2;
        mergeSort( a, tmpArray, left, center );
        mergeSort( a, tmpArray, center + 1, right );
        merge( a, tmpArray, left, center + 1, right );
    }
}
```
private static <AnyType extends Comparable<? super AnyType>>
void merge(AnyType[] a, AnyType[] tmpArray,
           int leftPos, int rightPos, int rightEnd) {
    int leftEnd = rightPos - 1;
    int tmpPos = leftPos;
    int numElements = rightEnd - leftPos + 1;
    // Main loop
    while (leftPos <= leftEnd && rightPos <= rightEnd)
        if (a[leftPos].compareTo(a[rightPos]) <= 0)
            tmpArray[tmpPos++] = a[leftPos++];
        else tmpArray[tmpPos++] = a[rightPos++];
    while (leftPos <= leftEnd) // Copy rest of first half
        tmpArray[tmpPos++] = a[leftPos++];
    while (rightPos <= rightEnd) // Copy rest of right half
        tmpArray[tmpPos++] = a[rightPos++];
    // Copy tmpArray back
    for (int i = 0; i < numElements; i++, rightEnd--)
        a[rightEnd] = tmpArray[rightEnd];
}
Linear-time Merging of Sorted Arrays

1 13 24 26
1 13 24 26
1 13 24 26
1 13 24 26
1 13 24 26
1 13 24 26
1 13 24 26
1 13 24 26

2 15 27 38
2 15 27 38
2 15 27 38
2 15 27 38
2 15 27 38
2 15 27 38
2 15 27 38
2 15 27 38

1
1
1
1
1
1
1
1

1 13 24 26
1 13 24 26
1 13 24 26
1 13 24 26

2 15 27 38
2 15 27 38
2 15 27 38
2 15 27 38

1 2 13 15
1 2 13 15
1 2 13 15
1 2 13 15

...
$T(N) = 2 \times T(N/2) + O(N)$

$= 2 \times (2 \times T(N/4) + O(N/2)) + O(N)$

$= 4 \times T(N/4) + O(N) + O(N)$

$= 4 \times (2 \times T(N/8) + O(N/4)) + O(N) + O(N)$

$= 8 \times T(N/8) + O(N) + O(N) + O(N)$

$= \ldots = 2 \log N \times T(1) + O(N) + O(N) + \ldots + O(N)$

$= N \times O(1) + O(N) + O(N) + \ldots + O(N).$

The terms are expanded $\log N$ times, each produces an $O(N)$. $\log N$ terms of $O(N) = O(N \log N)$
If $N = 2^p$ 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 $cN \log N + dN$. 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.
- Say a linear algorithm runs for 5 seconds on an input of size 10. How much time will it (approximately) run on an input of size 20?
Another Way to Look at Runtime

- What does ”linear runtime” really mean?
- A linear function (program, algorithm) requires resources that scale \textit{linearly} with the input size.
- Say a linear algorithm runs for 5 seconds on an input of size 10. How much time will it (approximately) run on an input of size 20?
- \( f(n) = O(n) \Rightarrow f(n) = c \times n \) for some \( c \). This means \( f(2n) \approx c \times 2n \).
- Doubling the input size roughly doubles the runtime.
- The exact runtime depends on the constant, the machine specs etc.
- If a quadratic algorithm \( f(n) = O(n^2) \) runs for 5 seconds on an input of size 10. How much time will it (approximately) run on an input of size 20?
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: $O(\log n)$