Course Notes for CS310 – Run Time Analysis

Nurit Haspel (notes partially adapted from Prof. Carl Offner)

Reading Material for this Class

- K&T chapter 5, S&W chapter 1.4 (runtime analysis).
- Remember what you learned about recursions in CS210.

Logarithms

This is a reminder. You should have seen logarithms in Math 140 if not before. Please refresh your memory, this is important. Logarithms are involved in many important runtime results: Sorting, binary search etc. We will see many examples today and later on in the course. 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 \( \log 1G = 30 \).

Things to Remember:

- 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.
- \( \log(nm) = \log(n) + \log(m) \)
- \( \log(n/m) = \log(n) - \log(m) \)
- \( \log(n^K) = k \log(n) \)
- \( \log_a(b) = \frac{\log_b}{\log_a} \)
- If the base of log is not specified, assume it is base 2 (although for runtime analysis it doesn’t matter. Why?)
- \( \log \): base 2
- \( \ln \): base e

Computers work in binary, so in order to calculate how many numbers a certain amount of memory can represent we use \( \log_2 \). When it comes to runtime, the base is not important (see homework). So in runtime calculations I will just use \( \log(N) \) with no base. You may assume it is \( \log_2 \).
Computers Work in Binary

- 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 = \(2^{30+2} = 2^{30} \times 2^2 = 4G\). (many of today’s operating systems address space).
- 64 bits?? (most of today’s computers address space).

Runtime

Definitions

The basics of big-oh are hopefully covered in CS210. We will do a considerable amount of runtime analysis here.

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 (It is not always 100% clear what the “size of the problem” is. More on that later). \(T\) measures a resource: Runtime, CPU cycles, disk space, memory etc.

Definition 1: Big-O (read – big Oh)

\(T(N)\) is \(O(F(N))\) if there are positive constants \(c\) and \(N_0\) such that \(T(N) \leq c \times F(N)\) for all \(N \geq N_0\). In other words, \(T(N)\) is bounded by a multiple of \(F(N)\) from above for every big enough \(N\). See Figure 1 (a).

Definition 2: Big-Ω (read – big Omega)

\(T(N)\) is \(Ω(F(N))\) if there are positive constants \(c\) and \(N_0\) such that \(T(N) \geq c \times F(N)\) for all \(N \geq N_0\). In other words, \(T(N)\) is bounded by a multiple of \(F(N)\) from below for every big enough \(N\). See Figure 1 (b).

For a good estimate on the runtime it’s good to have both the \(O\) and the \(Ω\) estimates (upper and lower bounds).

Definition 3: Big-Θ (read – big Theta)

\(T(N)\) is \(Θ(F(N))\) if there are positive constants \(c_0, c_1\) and \(N_0\) such that \(c_0 \times F(N) \leq T(N) \leq c_1 \times F(N)\) for all \(N \geq N_0\). In other words, \(T(N)\) is bounded both from above and from below by a multiple of \(F(N)\) for every big enough \(N\). It does NOT mean that they are equal, but that they are in some way equivalent.

Example: Show that \(2N + 4 = O(N)\). To solve this, you have to actually give two constants, \(c\) and \(N_0\) such that \(2N + 4 \leq c \times N\) for every \(N \geq N_0\). Obviously, there are many possible solutions. For example, \(c = 4\) and \(N_0 = 2\) are good constants since \(2N + 4 < 4N\) for every \(N \geq 2\). Similarly, \(c = 10\) and \(N_0 = 1\) can also be used. Notice that the bound does not have to be tight, as long as it holds for any large enough \(N\).

Order of growth can be important. For example – sorting algorithms can perform quadratically or as \(n \times log(n)\). Very big difference for large inputs (do the math!). We care less about constants, so \(100N = O(N)\). \(100N + 200 = O(N)\). The constant can be important when choosing between two similar run-time algorithms. For example – quicksort vs. mergesort.
Important Rules

Computer Programs: We assume that all the atomic operations (basic arithmetic operations, if-statements, assignments, comparisons etc.) take $O(1)$ (constant) time. We don’t care exactly how much time they really take, and we make the simplifying (and generally incorrect) assumption they all take the same amount of time. The reason the specific times are not important is that they usually depend on the machine specs, but more importantly – they do not depend on the input size (which is the actual meaning of $O(1)$!). We just say they all take at most $C$ time, where $C$ is a large enough constant for this assumption to be true.

Polynomials: 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.

Common Functions You Should Remember: Polynomials always grow faster than logarithms. Exponents always grow faster than polynomials. See Figure 2 and the following table:

<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>
</tbody>
</table>

Table 1 is a useful example that shows actual runtimes as a function of $n$ and $f(n)$. Remember that the absolute runtime are not as important here as the concept of runtime growth:
Figure 2: Growth of several important functions

<table>
<thead>
<tr>
<th>$n$</th>
<th>$f(n)$</th>
<th>$\lg n$</th>
<th>$n$</th>
<th>$n \lg(n)$</th>
<th>$n^2$</th>
<th>$2^n$</th>
<th>$n!$</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td></td>
<td>0.003µs</td>
<td>0.01µs</td>
<td>0.033µs</td>
<td>0.1µs</td>
<td>1µs</td>
<td>3.63 ms</td>
</tr>
<tr>
<td>20</td>
<td></td>
<td>0.004µs</td>
<td>0.02µs</td>
<td>0.086µs</td>
<td>0.4µs</td>
<td>1ms</td>
<td>77.1 years</td>
</tr>
<tr>
<td>30</td>
<td></td>
<td>0.005µs</td>
<td>0.03µs</td>
<td>0.147µs</td>
<td>0.9µs</td>
<td>1 sec</td>
<td>$8.4 \times 10^{15}$ yrs</td>
</tr>
<tr>
<td>40</td>
<td></td>
<td>0.005µs</td>
<td>0.04µs</td>
<td>0.213µs</td>
<td>1.6µs</td>
<td>18.3 min</td>
<td></td>
</tr>
<tr>
<td>50</td>
<td></td>
<td>0.006µs</td>
<td>0.05µs</td>
<td>0.282µs</td>
<td>2.5µs</td>
<td>13 days</td>
<td></td>
</tr>
<tr>
<td>100</td>
<td></td>
<td>0.007µs</td>
<td>0.1µs</td>
<td>0.644µs</td>
<td>10µs</td>
<td></td>
<td>$4 \times 10^{14}$ yrs</td>
</tr>
<tr>
<td>$10^4$</td>
<td></td>
<td>0.010µs</td>
<td>1µs</td>
<td>9.966µs</td>
<td>1ms</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$10^4$</td>
<td></td>
<td>0.013µs</td>
<td>10µs</td>
<td>130µs</td>
<td>100ms</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$10^5$</td>
<td></td>
<td>0.017µs</td>
<td>100µs</td>
<td>1.67ms</td>
<td>10 sec</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$10^6$</td>
<td></td>
<td>0.020µs</td>
<td>1ms</td>
<td>19.93ms</td>
<td>16.7 min</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$10^7$</td>
<td></td>
<td>0.023µs</td>
<td>0.01 sec</td>
<td>0.23 sec</td>
<td>1.16 days</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$10^8$</td>
<td></td>
<td>0.027µs</td>
<td>0.1 sec</td>
<td>2.66 sec</td>
<td>115.7 days</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$10^9$</td>
<td></td>
<td>0.030µs</td>
<td>1 sec</td>
<td>29.9 sec</td>
<td>31.7 yrs</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 1: Runtime vs. input size of various input sizes.
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: \( \frac{N^2(N-1)}{2} \).

**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;
            }
}
```

To calculate the runtime 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: \( \frac{n(n-1)}{2} = O(n^2) \) – see above! I told you this sum will show up a lot.

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**

That’s a slightly trickier one, but not much more so. In recursive functions we don’t have all the work done in just one call, as is the case in iterative functions. Therefore, we can’t just count the number of operations as we did in the example before. There are some nice tricks that can help us figure it out.

Let us define \( T(n) \) as a function that measures the runtime. \( T(n) \) may not be given explicitly in closed form, especially in recursive functions, so we don’t know what it is yet. It can be polynomial,
logarithmic, exponential etc. We have to find a way to derive the closed form from the recursive function.

**Example:** factorial

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

Now all we have to do is translate the Java program above into a mathematical formula which expresses its runtime. Let us analyze, line by line, what the function does. It’s quite easy, since recursive functions are usually very short.

- The if statement takes $O(1)$.
- What about the rest? The statement return $n$*factorial(n-1) performs one multiplication followed by a recursive call. In other words, it calls the same function but on an input of size $n-1$.
- Since we defined $T(n)$ as the function that expresses the runtime of factorial(n), then $T(n-1)$ expresses the runtime of factorial(n-1).

Rearranging a bit, we can divide the operations into two parts:

1. The operations done explicitly in the function itself – the if statement and the multiplication. They all take a constant amount of time. Since the exact number is not important, we can bundle them all under a big enough constant $C$.
2. The recursive call – we don’t have an explicit runtime for this part yet, but as mentioned above, we can express it as $T(n-1)$.

Putting it all together, the runtime of factorial can be expressed as $T(n) = T(n-1) + C$. It is not the final answer yet, but we’re getting there. We can apply the same logic to analyzing the runtime of factorial(n-1) and so: $T(n-1) = c + T(n-2)$ ⇒ $T(n) = 2c + T(n-2)$.

After $n$ such equations we reach $T(1) = k$ (just the if-statement. Notice that $k$ is not the same as $C$. Minor detail, but can be important sometimes). Eventually, $T(n) = (n-1) * c + k = O(n)$. The iterative function performs the same. This is not always the case.

This is an example of a linear function. Let’s stop for a second and thing: 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? To answer the question, let’s go back to the definition of Big-O: $f(n) = O(n)$ ⇒ $f(n) = c * n$ for some $c$. This means $f(2n) \approx c * 2n$. In other words, if the function is linear, doubling the input size roughly doubles the runtime. The exact time 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? Let us do the same trick as before: $f(n) = O(n^2)$ ⇒ $f(n) = c * n^2$. This means $f(2n) \approx c * (2n)^2 = 4cn^2$. Doubling the input size increases the runtime 4-fold vs. 2-fold for a linear function! This goes to show why runtime is important. It may not look much for small input, but think of a function whose input is in the millions or more.

6
A Problematic Example:  The Fibonacci series. The well known Fibonacci series, where each number is the sum of the previous two numbers: 0 1 1 2 3 5 8 13 ...  

The formula is: $f(n) = f(n - 1) + f(n - 2)$, where the boundary conditions are $f(0) = 0, f(1) = 1$

This is a recursive definition, and this is the way we are used to thinking about it. So, it’s only natural 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);  
}
```

We can visualize it as follows:

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 (see Figure 3). Double recursion is not bad, as long as we split the work too! For example: Merge sort sorts recursively two halves of an array and merge. When we call merge sort recursively we do it twice, but on different input! The work is split between recursive calls in a smart way that does not involve any redundant calls.

We are not going to do an exact runtime analysis at this point, but the runtime is exponential. More accurately – $O(1.618^n)$. Why this weird number? More on that later in the course if time allows. In the case of fibonacci we easily make it more efficient by going against our instincts and write an iterative function

```c
int fib2(int n)  
{  
    int f1 = 0;  
    int f2 = 1;  
    int fi;  
    if(n == 0) return 0;  
    if(n == 1) return 1;  
    for(int i = 2 ; i <= n ; i++ )  
    {  
        fi = f1 + f2;  
    }
```
\[ f_1 = f_2; \]
\[ f_2 = f_i; \]
\}
\}
\text{return } f_i; \}

What is the runtime now? It is an iterative function, therefore we don’t need a recurrence formula, just counting operations.

**Binary Search**

**Definition:** Search for an element in a sorted array.
Return array index where the element is found or a negative value if not found. Start in the middle of the array. If the element is smaller than that, search in the smaller (left) half. Otherwise – search in the larger (right) half. Where I come from it’s sometimes called ”lion in the desert” algorithm (due to some obscure CS/mathematicians’ joke):

Q: How do you catch a lion in the desert?
A: Cut the desert into two equal halves with a lion-proof fence.
Pick the half which has the lion in it and recursively catch the lion in that half of the desert.

Illustration:

```
<table>
<thead>
<tr>
<th>Key</th>
<th>List</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>1 2 3 4 5 6 7 8 9</td>
</tr>
<tr>
<td>8&gt;4</td>
<td>1 2 3 4 5 6 7 8 9</td>
</tr>
<tr>
<td>8&gt;6</td>
<td>1 2 3 4 5 6 7 8 9</td>
</tr>
<tr>
<td>8=8</td>
<td>1 2 3 4 5 6 7 8 9</td>
</tr>
</tbody>
</table>
```

**Binary Search Implementation**
It is implemented in Java as part of the Collections API.

```java
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.

    /**
     * Performs the standard binary search
     * using two comparisons per level.
     * This is a driver that calls the recursive method.
     * @return index where item is found or NOT_FOUND if not found.
     */
    public static <AnyType extends Comparable<? super AnyType>> int binarySearch( AnyType[] a, AnyType x )
    {
        return binarySearch( a, x, 0, a.length -1 );
    }

    /**
     * Hidden recursive routine.
     */
    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;
    }

The Comparable <T super T> 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 solve it by now... The answer is: \( T(N) = O(\log N) \). I expect you to be able to figure it out yourselves, though.

**Recurrence Formula:**

\[
T(n) = \begin{cases} 
C & \text{If } n \text{ is 1} \\
T\left(\frac{n}{2}\right) + c & \text{Otherwise}
\end{cases}
\]

Notice that \( c \) and \( C \) are not the same constant!

**Mergesort**

You probably discussed MergeSort in CS210. It is a divide-and-conquer method to sort an array.
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
/**
 * Mergesort algorithm.
 * @param a an array of Comparable items.
 */
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.
 * @param a an array of Comparable items.
 * @param tmpArray an array to place the merged result.
 * @param left the left-most index of the subarray.
 * @param right the right-most index of the subarray.
 */
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 );
    }
}

/**
 * Internal method that merges two sorted halves of a subarray.
 * @param a an array of Comparable items.
 * @param tmpArray an array to place the merged result.
 * @param leftPos the left-most index of the subarray.
 * @param rightPos the index of the start of the second half.
 * @param rightEnd the right-most index of the subarray.
 */
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: We get two sorted halves and merge them. See Figure 4:

Recurrence Formula:
\[
T(n) = C \quad \text{If } n \text{ is 1}
\]
\[
T(n) = 2 \cdot T\left(\frac{n}{2}\right) + cn \quad \text{Otherwise}
\]

Again, c and C are not the same constant!

Remember how we analyzed recursive functions in the beginning of the semester. We first separate the function into the explicit (non-recursive) part and the recursive part. We then define a function \( T(N) \) which describes the runtime of the function on an input of size \( N \). We calculate the runtime as a function of the explicit and recursive parts, and get an equation of \( T(N) \), which we then try to solve. In this case, the top-level code does the following:

1. Boundary condition: \( O(1) \)
2. Two recursive calls to half the input, \( 2T\left(\frac{N}{2}\right) \)
3. Merge. Looking at the merge code, we loop over the two sorted halves, advancing two pointers and copying one value to the merged array each time. Overall we perform \( O(N) \) operations.

So, the runtime can be expressed as:
\[
T(N) = 2 \cdot T\left(\frac{N}{2}\right) + O(N)
\]
\[
=2 \cdot (2 \cdot T\left(\frac{N}{4}\right) + O(N/2)) + O(N)
\]
\[
=4 \cdot T\left(\frac{N}{4}\right) + O(N) + O(N)
\]
\[
=4 \cdot (2 \cdot T\left(\frac{N}{8}\right) + O(N/4)) + O(N) + O(N)
\]
\[
=8 \cdot T\left(\frac{N}{8}\right) + O(N) + O(N) + O(N)
\]
\[
=\ldots = 2 \log N \cdot T(1) + O(N) + O(N) + \ldots + O(N)
\]
\[
=N \cdot O(1) + O(N) + O(N) + \ldots + O(N).
\]
The terms are expanded logN times, each produces an O(N). log N terms of O(N) = O(N log N). This kind of formula is very common in divide-and-conquer algorithms.

This is another Identity that comes up frequently in algorithmic analysis. One basic way to solve it is to form a recursion tree. We saw an illustration of another example above (Figure 3). The recursion tree for the MergeSort formula is shown below in Figure 5. If \( N = 2^p \) then there are \( p \) rows with \( cn \) on the right, and one last row with \( dn \) 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.

**Best, Average, Worst Case**

When analyzing the runtime of an algorithm, we are usually interested in the following:

- **the worst-case time** This is an upper-bound to the run time of an algorithm. The worst-case time is useful because it gives a guarantee: you know that no matter what the input is, you will certainly do at least that well.

- **the best-case time** You usually don’t get the best-case time in practice. But it does tell something – it tells you that using this algorithm, you will never do better than the best-case time.
Figure 5: Recurrence tree for MergeSort.
the average-case time Average the times that the algorithm takes over all possible inputs of length $n$. To do this, we need some assumption of the statistical distribution of the inputs. (For instance, if we know that for our particular application certain inputs will never occur, we can ignore them in figuring out the average.)

Average-case analysis is the most difficult to figure out in general, but it is also the most useful. In many cases – binary search, mergesort, finding maximum/minimum etc., the average runtime and the worst-case runtime are the same. In other cases (quicksort is probably the best known example to you) the average is better than the worst-case. It basically tells us, indirectly, that the worst case is quite unlikely to occur.