

Algorithm Analysis (Review from CS2420)

Program Performance

- Typically the performance of a program is dependent on the size of the input data
- Performance: memory and time required
 - Performance analysis: analytical
 - Performance measurement: experimental
- Space complexity (usually less important):
 - Space may be limited
 - May use to determine largest problem size we can solve
- Time complexity:
 - Real time constraints
 - May need to interact with user
- Components of space complexity:
 - At seats, draw a picture of how recursion works – space-wise.
 - Instruction space – needed to store the compiled version
 - Data space – variables and constants
 - Environment stack space – for recursion and return values

Log Review

- For a balanced binary tree with 63 nodes, how many levels are there?
- If I have an array of size 120, how many times can I split the array in half?
- Log values:
 - $\log_2 32 = 5$
 - $\log_3 27 = 3$
 - $\log_5 1 = 0$
 - $\log_a 1 = 0$
 - $\log_a a = 1$
 - $a^{\log_a x} = x$
- Components of time complexity:
 - Amount of time spent in each operation - difficult to measure
 - Estimate of number of times a key operation is performed
 - In small programs, actual elapsed time is difficult (for accuracy)

Operation Counts

- Operation count: how many times you add, multiply, compare, etc.

- Step counts: attempt to account for time spent in all parts of the program/function as a function of the characteristics of the program.
- See Figure 1.
- We can also assign counts on a per statement basis. See Figure 2. The total is $2(\text{rows} \times \text{col}) + 2\text{rows} + 1$
- Key reason for operation or step counts is to compare two programs that compute the same results.

Asymptotics

What is an asymptote?

- Asymptotics – study of functions of n as n gets large (without bound)
- If the running time of an algorithm is proportional to n , when we double n , we double the running time.
- If the running time is proportional to $\log n$ ($c \log n$), when we double n we only change the running time by c (c is constant of proportionality).

$$c \log 2n = c(\log 2 + \log n) = c(1 + \log n) = c + c \log n$$

- Since original time was $c \log n$, doubling n only increased the time by c .
- What if something is proportional to n^2 (running time is cn^2)? When n doubles, how does the time differ?
- $c(2n)^2 = c(4n^2) = 4(cn^2)$
- What if something is proportional to n^3 (running time is cn^3)? When n doubles, how does the time differ?
- $c(2n)^3 = c(8n^3) = 8(cn^3)$

- List examples of something being proportional to something else:

- Earnings are proportional to hours worked.
- Grades (may seem like) are proportional to log of work.
- For some, like is proportional to cost^2 .
- Area of square is proportional to base^2 .
- Volume of square is proportional to base^3 .

```

template<class T>
void Add( T a[][MAX], T b[][MAX],
         T c[][MAX], int rows, int cols)
{ // Add matrices a and b to obtain matrix c.
  for (int i = 0; i < rows; i++) {
    count++; // preceding for loop
    for (int j = 0; j < cols; j++) {
      count++; // preceding for loop
      c[i][j] = a[i][j] + b[i][j];
      count++; // assignment
    }
    count++; // last time of j for loop
  }
  count++; // last time of i for loop
}

```

Figure 1 Operation Count

```

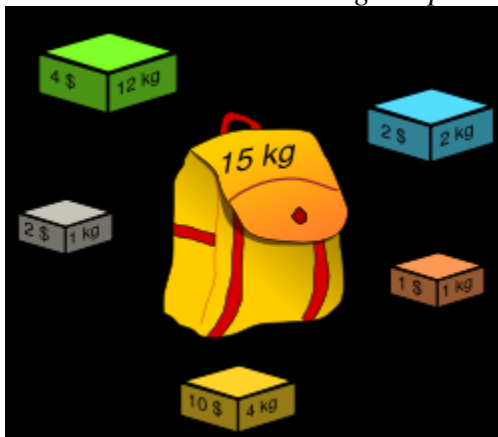
void Add( T **a, T **b, T **c, int rows, int cols)
{
  for (int i = 0; i < rows; i++)           rows+1
    for (int j = 0; j < cols; j++)       rows(cols+1)
      c[i][j] = a[i][j] + b[i][j];     rows*cols
}

```

Figure 2 Per Statement Count

O Notation (Big Oh)

- We want to give an *upper bound* on the amount of time it takes to solve a problem.
- Definition: $v(n) = O(f(n)) \leftrightarrow \exists$ constant c and n_0 such that $|v(n)| \leq c|f(n)|$ whenever $n \geq n_0$.
 - The growth rate of $T(N)$ is less than or equal to that of $F(N)$
- Termed complexity but has nothing to do with difficulty of coding or understanding, just the time to execute.
- Important tool for analyzing and describing the behavior of algorithms.
- Is an $O(n^2)$ algorithm always better than an $O(n^3)$ algorithm?
 - No, it depends on the specific constants.
 - But if n is large enough, an $O(n^3)$ algorithm is always slower than an $O(n^2)$ algorithm.
- The different complexity classes are: $O(1)$ (constant), $O(\log n)$ (logarithmic), $O(n)$ (linear), $O(n \log n)$ ($n \log n$), $O(n^2)$ (quadratic), $O(n^3)$ (cubic), $O(2^n)$ (exponential), $O(n!)$ (factorial).
- For small n , c may dominate.
- Insertion sort is an $O(n^2)$ algorithm. Look at the difference between sorting 1,000 elements and 10,000 elements. $O(10^6)$ vs. $O(10^8)$
- Intractable – all known algorithms to solve a problem are of exponential complexity.
- **What kinds of problems have an exponential solution?** What about the knapsack problem? *Given a set of items, each with a cost and a value, then determine the number of each item to include in a collection so that the total cost is less than some given cost and the total value is as large as possible.*



Other Notations

Name	Expression	Growth Rate	Similar to
Big-Oh	$T(N) = O(F(N))$	Growth of $T(N)$ is \leq growth of $F(N)$	Less than or Equal (Upper bound)
Big-Omega	$T(N) = \Omega(F(N))$	Growth of $T(N)$ is \geq growth of $F(N)$	Greater than (lower bound)
Big-Theta	$T(N) = \Theta(F(N))$	Growth of $T(N)$ is $=$ growth of $F(N)$	Equal to (tight bound)
Little-Oh	$T(N) = o(F(N))$	Growth of $T(N)$ is $<$ growth of $F(N)$	Less than (sub)

So why not just use Big-Theta? Wouldn't we rather have a tight bound?

Consider the difference between showing:

1. I can solve a problem in n^2 time. (There exists a solution I can show you.)
2. No solution will be any better (I can reason about ALL possible solutions.)

Clearly the first is easier, which is like Big Oh. Big Theta is like the second option, and is much more difficult.

Measuring Complexity

1. Additive Property – If two statements follow one another, the complexity of the two of them is the larger complexity.
 - a. In Figure 3, there are two values that effect the complexity: m and n .
 - b. The first statement has complexity $O(n)$. The second statement has complexity $O(m)$.
 - c. Therefore, the additive property indicates: $O(n) + O(m) = O(\max(n, m))$.

```
for (i=0; i < n; i++) x++
for (j=0; j < m; i++) x++
```

Figure 3 Additive

2. If/Else: For the fragment in Figure 4:
 - a. The complexity is the running time of the cond plus the larger of the complexities of S1 and S2.

```
if ( cond ) S1
else S2
```

Figure 4 If/Else

3. Multiplicative Property:
 - a. For Loops: the complexity of a for loop is at most the complexity of the statements inside the for loop times the number of iterations.
 - b. However, if each iteration is different in length, it is more accurate to sum the individual lengths rather than just multiply.

Nested For Loops

- Analyze nested for loops inside out. The total complexity of a statement inside a group of nested for loops is the complexity of the statement multiplied by the product of the size of each for loop.
- See Figure 5 for Example 1.
 - $O(mn)$
 - Consider a pictorial view of the work shown in Figure 6. Let each row represent the work done in one iteration of the outermost loop. The number of rows represents the number of times the outermost loop executes. The area of the figure will then represent the total work.
- See Figure 7 for Example 2.
 - In this example, our complexity picture is triangular.
 - The outermost loops executes m times, but since each time the j loop is called it is has a different beginning location, the rows are of different length.
 - The complexity is $O(m^2)$
- See Figure 8 for another example
 - In this example, the complexity picture looks like an upside down pyramid.
 - The outermost loop executes n times, but the number of times the j loop executes halves each time.
 - The complexity is $O(n \log n)$.

```
for (int i = 0; i < m; i++)
  for (int j = 0; j < n; j++)
    x++;
```

Figure 5 Example 1

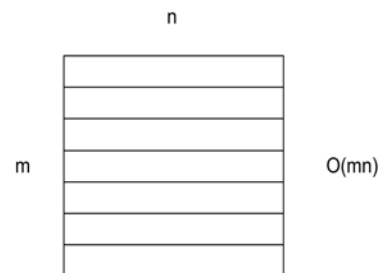


Figure 6 Nested for Loops

```
for (i = 1; i < m; i++)
  for (j = i; j < m; j++)
    x++;
```

Figure 7 Example 2

```
for ( int i = 0; i < n; i++ )
  for ( int j = 0; j < n / 2 )
    x++
```

Figure 8 Another Example

What about searching

static searching: array is not changed.

What is complexity of sequential search? binary search?

Interpolation search is like a binary search except you pick a more intelligent choice for the next search point. Based on the value you are looking for, you estimate which fraction of the remaining array you should skip, convert that to a number, and add to the low bound.

$$\text{next} = \text{low} + (\text{high} - \text{low} - 1) * (x - a[\text{low}]) / (a[\text{high}] - a[\text{low}])$$

The complexity is difficult to estimate, but likely it isn't much better than binary.

1. If distribution isn't consistent, we could actually take more time as our estimates are bad.
2. With approximately even distribution of values, estimates say the search is $O(\log \log n)$.
3. Point is – changing complexity classes is difficult. Just being a little smarter likely doesn't change complexity classes.

Recursive Algorithms

- The complexity for recursive algorithms requires additional techniques.
- See Figure 9 for Example 3.
- If we let $T(n)$ represent the time to solve `doit(n)`, the running time is represented recursively as $T(n) = n + 2T(n/2)$.
- In other words, the time for method `doit` to execute when n is the parameter is n (because of the for loop) plus two times the running time of $T(n/2)$ (since `doit` is called twice recursively with a parameter of $n/2$).
- Since T is defined in terms of T , this is called a *recurrence relation*.
- In our pictorial view (Figure 10), we let each line represent a layer of recursions (The first call is the first row, the two calls at the second level (`doit` calls `doit`) comprise the second row, the four third level calls (`doit` calls `doit` calls `doit`) represent the third row. The length of the row represents the call itself (ignoring costs incurred by the recursive calls). In other

```
void doit(int n)
{ if (n==1) return;
  for (int i=0; i < n; i++)
    x = x + i;
  doit(n/2);
  doit(n/2);
}
```

Figure 9 Example 3

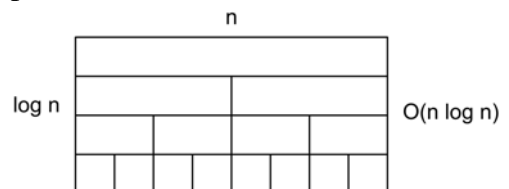


Figure 10 Pictorial of Example 3

words, to determine the size of the first row, measure how much work is done in that call not counting the recursive calls it makes.

- The number of rows is determined by $\log(n)$.
- The overall complexity is $O(n \log n)$.
- See Figure 11 for Example 4.
- If we let $T(n)$ represent the time to solve this problem, the time is represented recursively as $T(n) = T(n/2) + n$.
- In our pictorial view (shown below in Figure 12), we let each line represent a layer of recursions (The first call is the first row, the one call at the second level (doit calls doit) is the second row, the one third level call (doit calls doit calls doit) represents the third row).

```
void doit(int n)
{ if (n<=0) return;
  for (int i=0; i < n; i++)
    x = x + i
  doit(n/2)
}
```

Figure 11 Example 4

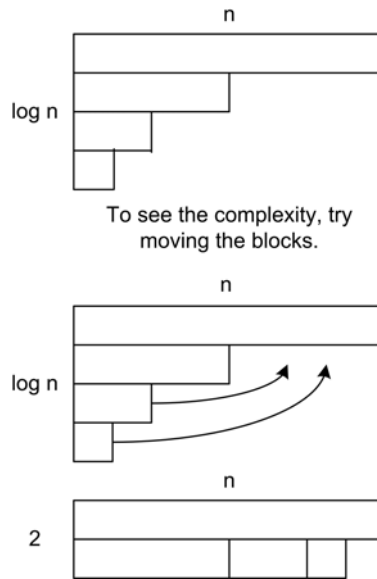


Figure 12 Pictorial of Figure 11

- The length of the row represents the call itself (ignoring costs incurred by the recursive calls). In other words, to determine the size of the first row, measure how much work is done in that call not counting the recursive calls it makes.
- The complexity of this example is $O(2n)$ (really $O(n)$).
- See Figure 13 for Example 5.
- If we let $T(n)$ represent the time to solve this problem, the time is represented recursively as $T(n) = T(n/2) + 1$.
- In this case, a single call to `doit(n)` (ignoring recursive calls) takes constant time. We draw that as a square of length 1. Since there are $\log n$ levels representing the $\log n$ calls, the picture looks like Figure 13 below.

```
void doit( int n )
{ if (n <=1) return;
  int x = x + i;
  doit(n/2);
}
```

Figure 13 Example 5

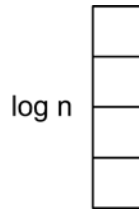


Figure 14 Pictorial of Figure 13

- The complexity of this example is $O(\log n)$.
- See Figure 14 for Example 6.
- If we let $T(n)$ represent the time to solve this problem, the time is represented recursively as $T(n) = 2T(n/2) + 1$.
- Again, the time to execute `doit(n)` ignoring recursive calls is constant. However, the number of calls required at each level doubles. Thus the work is $1 + 2 + 4 + \dots + n = 2n - 1$.
- Our picture is shown below in Figure 15.

```
void doit( int n )
{ if (n <=1) return;
  int x = x + i;
  doit(n/2);
  doit(n/2);
}
```

Figure 15 Example 6

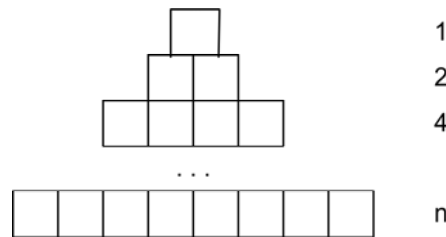


Figure 16 Pictorial for Example 6

A Formula Approach

- Mathematicians have developed a formula approach to determining complexity.
- Theorem: Assume $T(n) = aT(n/b) + O(n^k)$ is the time for the function.
 - If $a > b^k$, the complexity is $O(n^{\log_b a})$.
 - If $a = b^k$, the complexity is $O(n^k \log n)$.
 - If $a < b^k$ the complexity is $O(n^k)$.

For the above:

- a represents the number of recursive calls you make.
- b represents the number of pieces you divide the problem into.
- k represents the power on n which represents the work you do in breaking the problem into two pieces or putting them back together again. In other words, n^k represents the work you do independent of the recursive calls you make. Note this corresponds to one row in our pictorial representation.
- Let's use the theorem to revisit the same problems we solved pictorially.
- Example 3 (Figure 8).
 - There are two recursive calls made: $a=2$.

- Each recursive call does half of the work: $b=2$.
- The work done in a single call is n .
- k is the power on n : $k=1$ (as work is n).
- Since $a = b^1$, we are in the “equals” case.
- If $a = b^k$ then $O(n^k \log n) = O(n^1 \log n) = O(n \log n)$, which is exactly what our pictures told us.
- Example 4 (Figure 10).
 - There is one recursive calls made: $a=1$.
 - The recursive call does half of the work: $b=2$.
 - The work done in a single call is n .
 - k is the power on n : $k=1$ (as work is n).
 - Since $a < b^1$, we are in the “less than” case.
 - If $a < b^k$, then $O(n^1) = O(n^k) = O(n)$, which is exactly what our pictures told us.
- Example 5 (Figure 12.)
 - There is one recursive calls made: $a=1$.
 - The recursive call does half of the work: $b=2$.
 - The work done in a single call is independent of n .
 - k is the power on n : $k=0$ (as work is 1).
 - Since $a = b^k$ as $1 = 2^0$, we are in the “equals” case.
 - If $a = b^k$, then $O(n^k \log n) = O(n^0 \log n) = O(\log n)$, which is exactly what our pictures told us.
- Example 6 (Figure 14).
 - There are two recursive calls made: $a=2$.
 - The recursive call does half of the work: $b=2$.
 - The work done in a single call is independent of n .
 - k is the power on n : $k=0$ (as work is 1).
 - Since $a > b^k$, we are in the “greater than” case.
 - If $a > b^k$ then the complexity is $O(n^{\log_b a}) = O(n^{\log_2 2}) = O(n^1)$ which is exactly what our pictures told us.

Recurrence Relations

- In analyzing algorithms, we often encounter progressions. Most calculus books list the formulas below, but you should be able to derive them.

Arithmetic Progression

- Example: $S = 3 + 5 + 7 + 9 + \dots + (2n - 1) + (2n + 1)$
- Writing the same sum backwards:
 $S = (2n + 1) + (2n - 1) + (2n - 3) + (2n - 5) + \dots + 5 + 3$

- If we add the two S 's together,

$$S = 3 + 5 + 7 + 9 + \dots + (2n-1) + (2n+1)$$

$$S = (2n+1) + (2n-1) + (2n-3) + (2n-5) + \dots + 5 + 3$$

$$2S = (2n+4) + (2n+4) + (2n+4) + (2n+4) + \dots + (2n+4) + (2n+4)$$

$$2S = n(2n+4)$$

$$S = n(n+2)$$

Geometric Progression

- Example:

$$S = 2 + 4 + 8 + 16 + 32 + \dots + 2^n$$
- Multiplying both sides by the base:

$$2S = 4 + 8 + 16 + 32 + \dots + 2^n + 2^{n+1}$$
- Subtracting S from $2S$ we get

$$S = 2^{n+1} - 2$$

Determining Complexity from Experimental Evidence

- At times we may want to verify the complexity of written code.
- To do this we can either use:
 - *Actual running times*: this can be done using system timing commands. The only downside is that sometimes the timing methods are too crude to give accurate information unless huge amounts of data are used.
 - *Counts of operations performed*: This is done by placing a counter at key points throughout the program (inside all loops, inside if/else constructs) to determine (roughly) how many operations are performed.
- To be able to determine complexity from experimental evidence, you must be able to have data for several different problem sizes. Since we do not know the constant, we can determine nothing from a single data point.
- For example, if our runtime information consists of

n	T(n)
32	800

- The complexity could be anything, such as:
 - $O(1)$ with $c = 800$
 - $O(n)$ with $c = 25$
 - $O(\log n)$ with $c = 160$
 - $O(n^2)$ with $c = .78$
- Even with two pieces of data it is not completely determined, as the timing doesn't have to be exact (you could have been lucky and finished faster).
- The easiest way to visually see the complexity is to have four or five data points in which the problem size keeps doubling.

- Consider the following timings obtained from running code. What is the complexity?
- Experimental Table 1

n	T(n)
2	10
4	10
8	10
16	11
32	8

This complexity is $O(1)$ – the run time is basically constant.

- Experimental Table 2

n	T(n)
2	10
4	17
8	32
16	66
32	130

This complexity is $O(n)$ with a c of 5. There is a bit of variability, however.

- Experimental Table 3

n	T(n)
2	10
4	10
8	15
16	20
32	25

This is $O(\log n)$ with a $c = 5$. The first entry doesn't fit the pattern, but remember that it may take a while for the pattern to emerge.

- Experimental Table 4

n	T(n)
2	8
4	32
8	500
16	65,600
32	4,294,967,300

This is $O(2^n)$ with $c = 2$.

- Experimental Table 5

n	T(n)
2	2
4	8
8	25
16	64
32	161

This is $O(n \log n)$ with $c = 1$.

- How do you figure out the complexity from experimental data when you know neither the constant nor the complexity?
- I “eyeball it” to come up with a good guess. Then I figure out the constant for one entry. Next I see if that constant works for all the entries. This is basically my “approximate-then-finalize” approach.
- How do I guess?
 - $O(1)$ is basically constant.
 - $O(\log n)$ grows slowly, by a constant between entries.
 - $O(n)$ doubles between entries.
 - $O(n \log n)$ slightly more than doubles between entries .
 - $O(n^2)$ quadruples between entries.
 - $O(2^n)$ grows exponentially .

Another way to check if the complexity is correct is to divide the experimental time by the complexity. Below represents three attempts at finding the complexity. The first is an overestimate. Notice how the ratio goes to zero. The second is correct, showing the constant is approximately 4.3. The third is an underestimate. Notice how the ratio increases.

n	Actual time	n3	ratio	n2	ratio	n log n	ratio
4	75	64	1.17	16	4.7	8	9.4
8	219	512	0.43	64	3.4	24	9.1
16	1104	4096	0.27	256	4.3	64	17.2
32	4398	32768	0.13	1024	4.3	160	27.5
64	17632	262144	0.07	4096	4.3	384	45.9

Practical Complexities

- Assume program P is $\Theta(n)$ for some constant n_1 and program Q is $\Theta(n^2)$ for some constant n_2 . Since $cn \leq dn^2$ for some $n \geq c/d$, program P is faster when $n \geq \max(n_1, n_2, c/d)$.
- Note, the coefficients matter in this comparison. Eventually the lower complexity wins out, but n might be quite large.
- Study the table below. Note that even for small n (1000), time is measured in years for $O(2^n)$.

$\log n$	n	$n \log n$	n^2	n^3	2^n
0	1	0	1	1	2
1	2	2	4	8	4
2	4	8	16	64	16
3	8	24	64	512	256
4	16	64	256	4096	65,536
5	32	160	1024	32,788	4,294,967,296

- One problem that concerns us is, “Is the n^2 always worse than $n \log n$, no matter what the constants?” The following table helps us answer that question:

N	$n \log n$	$10n \log n$	$100n \log n$	n^2
1	0	0	0	1
2	2	20	200	4
4	8	80	800	16
8	24	240	2,400	64
16	64	640	6,400	256
32	160	1,600	16,000	1,024
64	384	3,840	38,400	4,096
128	896	8,960	89,600	16,384
256	2,048	20,480	204,800	65,536
512	4,608	46,080	460,800	262,144
1,024	10,240	102,400	1,024,000	1,048,576
2,048	22,528	225,280	2,252,800	4,194,304
4,096	49,152	491,520	4,915,200	16,777,216
8,192	106,496	1,064,960	10,649,600	67,108,864
16,384	229,376	2,293,760	22,937,600	268,435,456
32,768	491,520	4,915,200	49,152,000	1,073,741,824
65,536	1,048,576	10,485,760	104,857,600	4,294,967,296

- Notice that while the constants make $n \log n$ worse than n^2 for a while, eventually (for large enough n) the higher complexity will be worse.