1 Algorithm Analysis

When you look at an algorithm, you could ask "How good is it"? This can be measured several ways:

- Does it do what its suposed to do?
- How fast does it run?
- How much memory does it use?

Algorithm analysis gives us a way to answer the second two questions. We use a hand-waving description on time and space, "too much", "a lot", "a little", "basically none" described in a mathematical format called Big-O notation. Big-O gives us a worst case run time or run space for code. In later classes you will see more complex forms of analysis which describe the best case run time (Big-Ω) and average case (Big-Θ).

Nearly all of the following rules, patterns, and examples were seen in class. Use this guide as a verification of your own notes.

2 Simple code

The following line of code does **one** thing:

```
x = y;  // x and y are integers
```

It does one thing if y is the largest possible integer or the smallest. We say this happens in **O(1)** time because it does not depend on the value of any variables. This is "basically no time".

A function call does work inside a function and so is *not* automatically **O(1)**. Here are some examples:

```
x = 3; // O(1)

y = x + 3; // O(1)

z = Math.sqr(x) // not automatically O(1)

while (x < 9) { // the test is O(1)
   ..        // the body might not be O(1)
} // The whole loop might not be O(1)

if (isAlive(x,y)) { // the test might not be O(1)
  x = 2; // O(1)
} // the whole selection depends on the test and the body so we dont
  // know yet
```

Rule: Simple actions take O(1) time to complete.

3 Sequential code

The next code lines each do **one** thing:

```
x = y;    O(1)
y = 3;    O(1)
```

[1] Adapted and Borrowed from Professor Wittie

Since they happen sequentially, the whole thing takes **O(1) + O(1) = O(1+1) = O(2)** time. “basically none” and more “basically none” is still “basically none” so together these lines are all **O**(1).
Rule: Sequential actions get added. O(1) + O(1).
Rule: When you add Big-O, you add inside the O(). O(1) + O(1) = O(1+1) = O(2).
Rule: O(c) = O(1) as long as c is a constant number, not a variable.

Therefore the runtime of the above example is O(1) + O(1) = O(1+1) = O(2) = O(1).

Technically math operations and assignment are all individual steps.

```
x = y + 4 - 7;
```

Therefore this code actually does **O(1) + O(1) + O(1) = O(1+1+1) = O(3) = O(1)** work. It's all **O(1)** since it is constant, finite, and simple.

If you are asked to do an analysis of code, it is best to show all steps of your work. In this case, you would apply all of the rules in sequence just as described above. Do not skip over steps or it is impossible to know why you reach the conclusions you reach.

4 Code with loops

Sometimes we have loops where the number of repetitions depends on a variable.

4.1 $O(1)$ internal work, n repetitions

```
for (int i = 0; i<n; i++) // n repetitions
  some O(1) action // O(1) work
```

The loop repeats n times and that may be a many or few depending on the value of n. Therefore, the loop repeats **n** times. Each time it does **O(1)** work so we multiply **n * O(1) = O(n*1) = O(n)**.
Rule: Loop repetitions get multiplied by the loop contents. n repetitions of O(1) work is O(n*1) overall.

4.2 $O(1)$ internal work, n-c repetitions where c is some constant

```
for (int i = 0; i<n-2; i++)  // n-2 repetitions
  some O(1) action    // O(1) work
```

Here the loop only repeats n-2 times. **(n-2)*O(1) = O(n-2) = O(n)** because the constant would have little effect on n if n was large.

(Note: unless specified, log of n means log base 2 of n. Computer Science uses base 2 very heavily.)
Rule: The possible order levels can be ordered from fastest to slowest.
O(1) $<$ O(lg n) $<$ O(n) $<$ O(n * lg n) $<$ O(n^2) $<$ O(n^3) $<$ O(n^4).
Rule: Smaller added/subtracted terms do not count. O(n^2 + lg n - n - 4 + 1000) = O(n^2).

4.3 $O(n)$ internal work, n-c repetitions where c is some constant

```
for (int i = 0; i<n; i++)   // n repetitions
  some O(n) action          // O(n) work
```

In this example, we have **n** repeats of **O(n)** work. Using the loop rule, overall we have **n * O(n) = O(n^2)**.

4.4 $O(log_2 n)$ internal work, n-c repetitions where c is some constant

```
for (int i = 0; i<n; i++)  // n repetitions
  some O(lg n) action    // O(lg n) work
```

In this example we have **n * O(lg n) = O(n * lg n)**. We cannot reduce this because multiplication magnifies the effect the lg n has on the n.
Rule: Smaller multiplied/divided terms DO count. O(n) * O(lg n) = O(n * lg n).

4.5 $O(i)$ internal work, n-c repetitions where c is some constant

```
for (int i = 0; i<n; i++) // n repetitions
  some O(i) action // O(i) work
```

For this one the work varies from loop to loop. We know it's at worst n since we stop when $i >= n$ so this loop is at worst $O(n^2)$ but maybe we can be more accurate.

We'll make a chart comparing each loop to the work in that particular loop.

i	work
0	0
1	1
...	...
n-1	n-1
n	n

The total work is the sum of $(0+1+...+n-1+n)$. We'll use a math trick to solve this. $(0+1+...+n-1+n) = (n*(n+1))/2 = (n^2+n)/2 = O(n^2)$. In this case, this was as accurate as we were getting but sometimes it might make the results better so use a chart when the work varies from loop to loop.

5 Selection Statements

A selection (or *if*) statement frequently has 2 or more options, only one of which will actually execute.

```
if (x < 3)
    // some O(n) task
else
    // some O(1) task
```

The test in this selection is **O(1)**. It will be computed no matter which branch we take. If the test is true, we'll take the first branch and the total will be **O(1)+O(n) = O(n+1) = O(n)**. If the test is false, we'll take the second branch and the total will be **O(1)+O(1) = O(1+1) = O(2) = O(1)**. If you do not know which branch will be taken, then the worst case is the slower one, **O(n)**. If the selection is inside a loop, you will need to consider how many of the loops will use each branch.
Rule: A branch for selection includes the time to run all tests that happen before the branch even if the test fails.
Rule: Selection gives us several different run times, one for each branch.

In the next example, there is a selection inside a loop.

```
for (int i = 0; i<n; i++) {
   if (i < 10)    // first branch O(n)
      // some O(n) task
   else           // second branch O(1)
      // some O(1) task
}
```

For the first 10 loops, we will choose the first branch. For the remaining (n-10) loops we will choose the second branch. In Big-O notation, we'll do 10 repetitions of the first branch plus (n-10) repetitions of the second branch. From the calculations above, we already know that the first branch is **O(n)** and the second branch is **O(1)**. So this code runs in **O(10) * O(n) + O(n-10) * O(1)**
= O(10 * n) + O(n-10) * O(1)
= O(n) + O(n-10) * O(1)
= O(n) + O(n) * O(1)
= O(n) + O(n * 1)
= O(n) + O(n)
= O(2 * n)
= O(n) time. When you get comfortable with this system, you will be able to do the math in your head. Until then, it is best to write down each and every step.

One way to keep all the pieces straight is to label each line with its Big-O notation as you learn it.

```
for (int i = 0; i<n; i++) { // loop is O(n)
   if (i < 10)    // first branch O(n)
      // some O(n) task
   else           // second branch O(1)
      // some O(1) task
}
```

6 Calls to methods

To figure out the Big-O run time for a call of the method, first figure out the Big-O for each piece of the method.

```
public int foo() {
   return 5 + 2;         // O(1) work
}
```

The line in the foo function runs in **O(1)** time therefore the function is **O(1)**.

```
public int bar(int n) {
   return n + 2;          O(1) work reguardless of n's value
}
```

The bar function also runs in **O(1)** time. Notice that there is one math operation and the work is still O(1) no matter how big n is.

```
public int more(int n) {
   // an O(n) task
}
```

The more function runs is **O(n)** time so a call to it, **more(x);** , runs in **O(x)** time. (x and n are just variable names, we could also say **O(n)** time since there is only 1 variable involved). However, the call **more(3);** runs in **O(3)=O(1)** time since we've removed the variable.

7 Recursive methods

For recursive methods, see if the value we recurse on goes down by a constant or halves. In general down by a constant (as in factorial) means **O(n)** times the work. Down by half means **O(lg n)** times the work. Down by two overlapping halves means O(n lg n) times the work.

7.1 Example with O(1) internal work and recurse on n-c

```
public int foo(int n) {
   if (n == 0) return 0;   // O(1) work
   return  n + foo(n-1);   // O(1) work + a call to foo
}
```

First find the work for everything but the recursive call.

Then, make a table and list the work done in each call to the method, ignoring the recursive call. Start from a non base-case value (n) and do each variable as the previous recursive call would have had it.

In our case, the variable is n and each time n goes down 1. The work in each call is 1. Our base case is 0 with 1 work.

variable is	work is
n	O(1)
...	...
3	O(1)
2	O(1)
1	O(1)
0	O(1)

How many ones are there? Each time n goes down by one and one is removed. How many times can we take one out of n before we run out of ones? ... n times. So there are n ones.

Then add the column of work to see the total work for **foo(n);** . Overall we add n ones and get **O(n)**.

Pattern: In $n, n-1, ..., 2, 1$ there are n terms.

7.2 Example with O(n) internal work and recurse on n-c

```
public void foo(int n) {
   if (n == 0) return;  // O(1) work

   else {
    ... // some O(n) task
    foo(n-1);  // recursive call
  } total: O(n) + recursive call
}
```

Our variable is n, each call it goes down by 1, each time the work is n. The base case of 0 has 1 work.

variable is	work is
n	n
n-1	n-1
...	...
2	2
1	1
0	0

The overall runtime is 1 + 2 + 3 + ... + n. The sum of 1 to n is $(n*(n+1))/2 = 1/2*(n^2+n) = O(n^2)$.

Pattern: 1 + 2 + 3 + ... + n = O(n^2)

7.3 Example with O(1) internal work and recurse on n/c

```
public void foo(int n) {
   if (n <= 1) return;   O(1) work

   else {
    ... // some O(1) task
    foo(n/2);  // recursive call
  } total: O(1) + recursive call
}
```

Here n goes down by n/2. Each recursive call does O(1) internal work. The base case of 1 does O(1) work.

variable is	work is
n	1
n/2	1
n/4	1
n/8	1
...	...
1	1

How many ones are there? Each time n is halved and a multiple of 2 is removed. We can rephrase our question as "how many multiples of two are there before we run out of twos?". This is the same as "how many twos go

into n?". In math terms, $2^x = n$ and we want to know x. Logarithms in base 2 give us the answer: $log_2 x = y$. Therefore there are $log_2 n$ ones. $O(1 * log_2 n) = O(log_2 n)$.
Pattern: In $n, n/2, n/4, n/8, ..., 1$ there are $log_2 n$ terms.

7.4 Example with O(n) internal work and recurse on n/2

Dividing the problem in half does not always give us an answer with $O(log_2 n)$.

```
public void foo(int n) {
   if (n <= 1) return;   O(1) work

   else {
    ... // some O(n) task
    foo(n/2);  // recursive call
  } total: O(n) + recursive call
}
```

Here n goes down by n/2. Each recursive call does O(n) internal work. The base case of 1 does O(1) work.

variable is	work is
n	n
n/2	n/2
n/4	n/4
n/8	n/8
...	...
1	1

We need to sum up $n + n/2 + n/4 + n/8 + ... + 1$. If we pull out an n from each term we get $n * (1 + 1/2 + 1/4 + 1/8 + .. + 1/n)$. If you think about it, $1 + 1/2 + 1/4 + 1/8 + .. + 1/n$ never reaches 2 so it is $O(1)$. Therefore this function is $O(n * 1) = O(n)$.
Pattern: n + n/2 + n/4 + n/8 + ... + 1 = O(n)
Pattern: 1 + 1/2 + 1/4 + 1/8 + .. + 1/n = O(1)

8 Selection Sort

Let's look back at selection sort.

```
void sort(int[] a) {
  for (int i=0; i<a.length -1; i++) {
    int minPosition = minPos(a,i);
    swap(a,minPos,i);
  }
}

int minPos(int[] a, int from) {
  int minPosition = from;
  for (int i=from+1; i<a.length -1; i++) {
    if (a[i] < a[minPosition])
       minPosition = i;
  }
  return minPosition;
}

void swap(int[] a, int i, int j) {
```

```
  int temp = a[i];
  a[i] = a[j];
  a[j] = temp;
}
```

8.1 Analyzing swap

Sort depends on swap and minPos. We'll use array length as n. Let's do swap first.

```
void swap(int[] a, int i, int j) {
  int temp = a[i]; // O(1)
  a[i] = a[j]; // O(1)
  a[j] = temp; // O(1)
} // total: O(1) + O(1) + O(1) = O(1+1+1) = O(3) = O(1)
```

8.2 Analyzing minPos

```
int minPos(int[] a, int from) {
  int minPosition = from;// O(1)
  for (int i=from+1; i<a.length -1; i++) { // repeats n-1-from times
    if (a[i] < a[minPosition]) // O(1)
       minPosition = i;  // O(1) if we do it, O(0) if not
  } // worst case, O(n-from-1) * O(1) = O(n-from)
    // both n and from are unknown and thus we are stuck with them
  return minPosition; // O(1)
} // total: O(1) + O(n-from) + O(1) = O(n-from)
```

8.3 Analyzing sort

```
void sort(int[] a) {
  for (int i=0; i<a.length -1; i++) { // n repeats
    int minPosition = minPos(a,i); // O(n-i)
    swap(a,minPos,i); // O(1)
  }
}
```

Inside the loop O(1) + O(n-i) = O(n-i). Since i has an effect on the runtime, we'll look at the big picture

i	runtime
0	O(n)
1	O(n-1)
2	O(n-2)
...	...
n-2	O(2)
n-1	O(1)

So the loop is sum of 1 + 2 + 3 + ... + n = $O(n^2)$.

Therefore Selection Sort runs in $O(n^2)$ time.

What about space? Sort is called once. It calls minPos n times and swap n times. MinPos and swap dont make any other calls. Therefore the runspace is O(n) + O(n) = O(n+n) = O(2n) = O(n) stack frames. We used only one array with n spaces so O(n) memory (local variables don't count, just things made with new.

8.4 Overall

Runtime	O(n^2)
Stack frames	O(n)
Memory	O(n)

This algorithm is good if you have to write the code yourself and you don't care how slow it is.

9 Merge Sort

Your book does an out-of-place merge sort because it creates whole new arrays. It uses a recursive sort method and a merge helper method.

```
void sort(int[] a) {
  if (a.length <= 1) return;

  int[] first = new int[a.length/2];
  int[] second = new int[a.length - first.length];
  System.arraycopy(a, 0, first, 0, first.length);
  System.arraycopy(a, first.length, second, 0, second.length);

  sort(first);
  sort(second);
  merge(a, first, second);
}

// merge first and second into a.
void merge(int[] a, int[] first, int[] second) {
  int iFirst = 0;  // position in first
  int iSecond = 0; // position in second
  int iA = 0;      // position in a

  // merge till one goes empty
  while (iFirst<first.length && iSecond<second.length) {
    if (first[iFirst] < second[iSecond]) {
      a[iA] = first[iFirst];
      iA++; iFirst++; // both spots done
    }
    else {
      a[iA] = second[iSecond];
      iA++; iSecond++; // both spots done
    }
  }

  // copy over the other (one is empty)
  System.arraycopy(first, iFirst, a, iA, first.length - iFirst);
  System.arraycopy(second, iSecond, a, iA, second.length - iSecond);
}
```

9.1 Analyzing merge

We'll start with merge. We'll use p as the length of a.

```
// merge first and second into a.
void merge(int[] a, int[] first, int[] second) {
  int iFirst = 0;  // position in first     O(1)
  int iSecond = 0; // position in second    O(1)
  int iA = 0;      // position in a         O(1)

  // merge till one goes empty
  while (iFirst<first.length && iSecond<second.length) {
  // O(length+length) = O(p) repeats
    if (first[iFirst] < second[iSecond]) { // O(1) to test
      a[iA] = first[iFirst];               // O(1)
      iA++; iFirst++; // both spots done      O(1)
    } // total: O(1+1+1) = O(1)
    else {                                 // O(1) from if test
      a[iA] = second[iSecond];             // O(1)
      iA++; iSecond++; // both spots done     O(1)
    } total: O(1+1+1) = O(1)
  } total: O(p) repeats * Max(1,1) work = O(p)

  // copy over the other (one is empty)
  System.arraycopy(first, iFirst, a, iA, first.length - iFirst);
  System.arraycopy(second, iSecond, a, iA, second.length - iSecond);
  // We know one is empty and the other might be full
  // at worst O(p)
}// total: O(1+1+1+p+p) = O(p) where p is the length of a
```

9.2 Analyzing sort

We'll use n for the length of the original array.

```
void sort(int[] a) {
  if (a.length <= 1) return;

  int[] first = new int[a.length/2]; // O(1)
  int[] second = new int[a.length - first.length];// O(1)
  System.arraycopy(a, 0, first, 0, first.length); // O(n)
  System.arraycopy(a, first.length, second, 0, second.length); // O(n)

  sort(first); // length is halved
  sort(second);// length is halved
  merge(a, first, second); // O(length a)
} // total: O(1+1+n+n) + first sort + second sort + O(n) = O(n) + first sort + second sort
```

Since sort depends on the length, we'll make another chart (a tree) showing each stack frame.

```
                               length n
          length n/2                                 length n/2
   length n/4      length n/4           length n/4        length n/4
len n/8  len n/8  len n/8 len n/8    len n/8  len n/8     len n/8   len n/8


                            ....
len 1 len 1 len 1 len 1 len 1 len 1 len 1 len 1 len 1 len 1 len 1 len 1 len 1
```

Pattern: If the levels in a tree of stack frames are n, n-1, n-2, ..., 1 then there are n levels
Pattern: If the levels are n, n/2, n/4, n/8, ..., 1 then there are $log_2 n$ levels.

Since n is halved each time, we have $log_2 n$ levels. The work at each frame is O(length at that frame). Each level therefore adds up to O(n).

```
O(n/2 + n/2) = O(n)
O(n/4 + n/4 + n/4 + n/4) = O(n)
O(n/8 + n/8 + n/8 + n/8 + n/8 + n/8 + n/8 + n/8) = O(n)
```

So $log_2 n$ levels * $O(n)$ work at each = $O(nlog_2 n)$.

Another way to analyze this is using a math recurrence formula as described in BJ chapter 14.

9.3 Space Analysis

This merge sort is recursive. If we look at the tree of stack frames above, we can count the number of frames per level. $1+2+4+8+...+n$. Using a pattern or some math, we find this is $O(n^2)$ stack frames.

This merge sort is not in place so it makes two new arrays every time it splits. Each array length can be seen in the tree of stack frames. If we add them up, we find there are a total of n spaces in the arrays for each level and we already saw that the tree had $log_2 n$ levels. That means this algorithm uses $O(nlog_2 n)$ space. An in-place version of this algoritm would use only $O(n)$ space.

9.4 Overall

Runtime	O($nlog_2 n$)
Stack frames	$O(n^2)$
Memory	O($nlog_2 n$)

This algorithm is faster than Selection Sort but uses a lot of space. It also has more code and thus takes longer to write.

Note: other implementations of merge sort might use less memory and stack space.

10 Quicksort and Analysis

This is an in-place (space saving), recursive version of quicksort. It works by taking an array adn picking a pivot element. It sorts the list into elements that are smaller than the pivot and elements that are larger. Then it recursively sorts the smaller and larger sides.

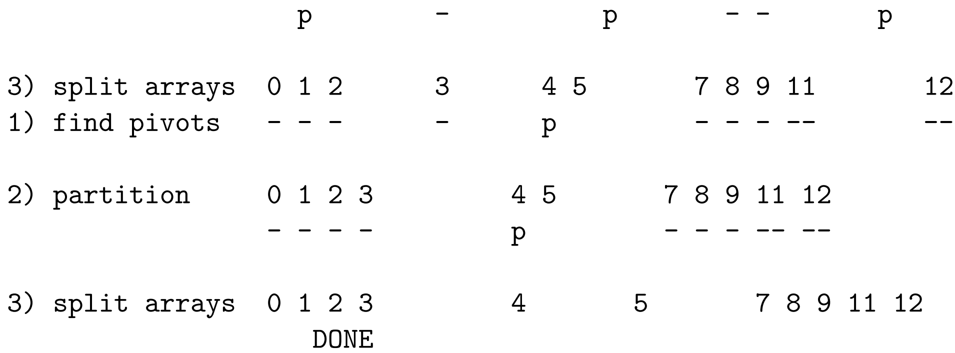
```
/**
    * Sorts an input array of integers using the quicksort algorithm. Pre: the
    * array 'a' contains the collection of integers to be sorted, 'first' and
    * 'last' contains the bounds of the section. Post: the array 'a' has been
    * sorted
    *
    * @param a[]
    *            the array to be sorted
    * @param first
    *            index of the start of the region to be sorted
    * @param last
    *            index of the end of the region to be sorted
    */
   void sort(int[] a, int first, int last) {
       if (first >= last)
           return;
       int pivot = partition(a, first, last);
       sort(a, first, pivot - 1);
```

```
        sort(a, pivot + 1, last);
    }

    /**
     * Partitions a region of the array around a pivot element. The pivot is
     * chosen to be the first element in the array. Everything in the first
     * partition is <= the pivot element. Everything in the second partition is
     * greater than the pivot. The pivot element gets moved to its final
     * destination in the sorted array.
     *
     * @param a[]
     *            the array to be partitioned
     * @param first
     *            index of the start of the region to be partitioned
     * @param last
     *            index of the end of the region to be partitioned
     * @return the index of the partition (pivot) element
     */
    int partition(int[] a, int first, int last) {
        // Use first as our pivot
        int lastSmall = first;
        for (int i = first + 1; i <= last; i++) {
            if (a[i] <= a[first]) {
                lastSmall++;
                swap(a, lastSmall, i);
            }
        }
        swap(a, first, lastSmall);
        return lastSmall;
    }
```

10.1 Running the algorithm

We'll walk thru the process with an example. I'll mark pivots with a p. I'll underline numbers that are in their final sorted spot. I'll show partitions by leaving spaces between parts of the array

```
                       8 3 9 1 7 2 12 4 5 11 0
1) find pivot          p

2) partition           3 1 7 2 4 5 0 8 9 12 11
                                     p

3) split array     3 1 7 2 4 5 0       8         9 12 11
1) find pivots     p                   -         p

2) partition       1 2 0 3 7 4 5 0     8         9 12 11
                         p             -         p

3) split arrays  1 2 0      3      7 4 5       8 9       12 11
1) find pivots   p          -      p           - -       p

2) partition     0 1 2      3      4 5 7       8 9       11 12
```



10.2 Analysis

This algorithm is slightly more complex to analyze. It's run time in the worst case (the numbers are already sorted) is $O(n^2)$ but if the pivot truly divides the array in half each time, we get an $O(nlog_2n)$ run time. It is in-place (needs only the one array) so it uses $O(n)$ space. It is recursive so like the recursive merge sort, it makes $O(nlog_2n)$ method calls.

The in-place versions of merge sort tend to have long, complicated code so this algorithm is both fast and uses little memory.

We can improve our algorithm by choosing a pivot intelligently to get a better split on the array. If you look above, our pivots were often the first or last number and thus not a half/half split. In lab, you will see another option and run some tests to see quicksort improve.

The basic idea of analysis for quicksort:

10.3 Analyzing partition

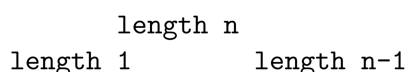
```
int partition(int[] a, int first, int last) {
    // Use first as our pivot
    int lastSmall = first; // O(1)
    for (int i = first + 1; i <= last; i++) { O(last-first) repeats
        if (a[i] <= a[first]) {// O(1) even if false
            lastSmall++;           // O(1)
            swap(a, lastSmall, i); // O(1)
        } // total: O(1)
    } // total: O(last-first)
    swap(a, first, lastSmall); // O(1)
    return lastSmall; // O(1)
} total: O(last-first)
```

10.4 Analyzing sort

```
void sort(int[] a, int first, int last) {
    if (first >= last) // O(1) even if false
        return;        // O(1)
    int pivot = partition(a, first, last); // O(last-first)
    sort(a, first, pivot - 1);
    sort(a, pivot + 1, last);
}
```

We'll use a tree of stack frames. In the worst case, each split produces an array of size 1 and an array that is only 1 smaller.

Worst case:



```
                    length 1    length n-2
                           length 1    length n-3
                                  length 1    length n-4
                           ...
                                            length 1 length 1
```

This tree has O(n) levels and the levels do the following work

level	work
n	n
n-1	n-1
n-2	n-2
...	...
1	1

The total work is n + n-1 + n-2 + ... + 1 = $O(n^2)$.

In the average case, we split the array in half.

Average case:

```
                            length n
        length n/2                                  length n/2
   length n/4       length n/4           length n/4         length n/4
len n/8  len n/8  len n/8 len n/8    len n/8  len n/8     len n/8   len n/8


                           ....
len 1 len 1 len 1 len 1 len 1 len 1 len 1 len 1 len 1 len 1 len 1 len 1 len 1
```

This has $log_2 n$ levels according to one of our rules. Each stack frame does work for its length, so each level does a total of n work. That gives us an $O(nlog_2 n)$ runtime.

10.5 Overall

Runtime	O($nlog_2 n$)
Stack frames	$O(n^2)$
Memory	O($nlog_2 n$)

But you only get this run time if you use a good pivot. The method shown here chooses a bad pivot.

11 What do I expect from you?

You can analyze $O(1), O(n), O(n^2), O(n^3), ...$ code bits and methods. You can deal with simple statements, sequential statements, selection statements, and method calls.

You can draw a chart of loop variables and the work for each loop iterations. You can calculate the total work for a loop either by the *work*repeats* formula or by adding up all the work for the loop iterations. You can do this for the *for, while,* and *do while* loops.

You can draw a chart or tree for the stack frames when recursion is involved. You can figure out how many levels there are in the tree. You can add up the work done in each level and find the runtime of your algorithm using *work in a level * number of levels*.

12 Summary of rules and patterns

Rule: Simple actions take O(1) time to complete.
Rule: Sequential actions get added. O(1) + O(1).
Rule: When you add Big-O, you add inside the O(). O(1) + O(1) = O(1+1) = O(2).
Rule: O(c) = O(1) as long as c is a constant number, not a variable.

Rule: Loop repetitions get multiplied by the loop contents. N repeats of O(1) work is O(n*1) overall.
Rule: The possible order levels can be ordered from fastest to slowest. O(1) $<$ O(lg n) $<$ O(n) $<$ O(n * lg n) $<$ O(n^2) $<$ O(n^3) $<$ O(n^4).
Rule: Smaller added/subtracted terms do not count. O(n^2 + lg n - n - 4 + 1000) = O(n^2).
Rule: Smaller multiplied/divided terms DO count. O(n) * O(lg n) = O(n * lg n).
Rule: A branch for selection includes the time to run all tests that happen before the branch even if the test fails.
Rule: Selection gives us several different run times, one for each branch.
Pattern: 1 + 2 + 3 + ... + n = O(n^2)
Pattern: n + n/2 + n/4 + n/8 + ... + 1 = O(n)
Pattern: 1 + 1/2 + 1/4 + 1/8 + .. + 1/n = O(1)
Pattern: If the levels in a tree of stack frames are n, n-1, n-2, ..., 1 then there are n levels
Pattern: If the levels in a tree of stack frames are n, n/2, n/4, n/8, ..., 1 then there are $log_2 n$ levels.