0. (1 pt.) Have you thoroughly read the course policies on the webpage?

[We are expecting: The answer “yes.”]

SOLUTION: Yes.

1. (1 pt.) See the IPython notebook `HW1.ipynb` for Exercise 1. Modify the code to generate a plot that convinces you that \( T(x) = O(g(x)) \). Note: There are instructions for installing Jupyter notebooks in the pre-lecture exercise for Lecture 2.

[We are expecting: Your choice of \( c, n_0 \), the plot that you created after modifying the code in Exercise 1, and a short explanation of why this plot should convince a viewer that \( T(x) = O(g(x)) \).]

SOLUTION: We choose \( c = 4 \) and \( n_0 = 8 \). As we can see in the picture below, for all \( n > n_0 \) (that is, to the right of the yellow vertical line), we have \( cg(n) \geq T(n) \), meaning that the green dashed curve lies above the solid red curve.

![Picture that convinces you that \( T(x) = O(g(x)) \)](image)

2. (3 pt.) See the IPython notebook `HW1.ipynb` for Exercise 2, parts (a), (b) and (c).

(a) What is the asymptotic runtime of the function `numOnes(lst)` given in the Python notebook? Give the smallest answer that is correct. (For example, it is true that the runtime is \( O(2^n) \), but you can do better).

[We are expecting: Your answer in the form “The running time of `numOnes(lst)` on a list of size \( n \) is \( O(\ldots) \).”, and a short explanation of why this is the case.]
(b) Modify the code in HW1.ipynb to generate a picture that backs up your claim from Part (a).

[We are expecting: Your choice of \( c \), \( n_0 \), and \( g(n) \); the plot that you created after modifying the code in Exercise 2; and a short explanation of why this plot should convince a viewer that the runtime of \texttt{numOnes} is what you claimed it was.]

(c) How much time do you think it would take to run \texttt{numOnes} on an input of size \( n = 10^{15} \)?

[We are expecting: Your answer (in whichever unit of time is easiest to interpret) with a short explanation that references the runtime data you generated in part (b). You don’t need to do any fancy statistics, just a reasonable back-of-the-envelope calculation.]

SOLUTION:

(a) The running time of \texttt{numOnes(list)} on an input list of size \( n \) is \( O(n) \). This is because the for loop goes through \( n \) iterations, and in each iteration it does a constant number of operations: it tests if \( x == 1 \), and then it might increment a counter. Thus, the running time is \( O(n) \).

(b) See the picture below, where we have chosen \( c = 0.00035 \) and \( n_0 = 100 \). It seems in the picture that for all \( n > n_0 \) (that is, to the right of the yellow vertical line), the green dashed curve lies above the solid red curve, meaning that \( cg(n) \geq T(n) \), where \( g(n) = n \) and \( T(n) \) is the running time of \texttt{numOnes} on an input of size \( n \).

(c) Now that we’ve convinced ourselves through both parts (a) and (b) that the runtime is roughly linear, we can roughly model \( T(n) \) (the running time of \texttt{numOnes} as \( T(n) = a \cdot n \) for some slope \( a \)). In order to predict \( T(10^{15}) \) (without actually running it...) we should estimate \( a \). For this problem, eyeballing it is fine for full credit. However, we chose to solve a least squares problem to find the best slope, which in our case turned out to be \( a \approx 0.00028 \). (Of course, this will be different on different computers, and even different runs on the same computer!) Projecting out, we find that the number of milliseconds that this would take is \( 10^{15} \cdot a \), which works out to about 283 billion milliseconds, or 3280 days, or about 9 years. Good thing we didn’t try it!
3. (1 pt.) Which of the following functions are $O(n^2)$?

No explanation is required, but you might want to prove your answer to yourself to convince yourself that you are correct.

(a) $f_1(n) = 5n + 3$
(b) $f_2(n) = 5n^2 + 3$
(c) $f_3(n) = 5n^3 + 3$
(d) $f_4(n) = n \log(n)$
(e) $f_5(n) = \sin(n) + 5$
(f) $f_6(n) = 2^n$
(g) $f_7(n) = 2^{100}$

[We are expecting: A list of which functions are $O(n^2)$. No explanation is required and no partial credit will be given.]

SOLUTION: The functions $f_1, f_2, f_4, f_5,$ and $f_7$ are all $O(n^2)$.

4. (4 pt.) Plucky the Pedantic Penguin is trying to prove, from the definition of big-Oh, that $f(n) = O(g(n))$, where $f(n) = 4n$, $g(n) = n^2 - 2n$.

(a) In the definition of big-Oh, Plucky would really like to take $c = 1$. Give a proof that $f(n) = O(g(n))$ in which “$c$” is chosen to be 1.

(b) Plucky has changed their mind: now they don’t care what $c$ is, but would like to take $n_0 = 3$. Give a different proof that $f(n) = O(g(n))$ in which “$n_0$” is chosen to be 3.

[We are expecting: For both parts, a short but formal proof.]

SOLUTION:

(a) Let $c = 1$ and $n_0 = 6$. Then for all $n \geq n_0$, we have

\[
6 \leq n \\
6n \leq n^2 \\
4n \leq n^2 - 2n \\
f(n) \leq 1 \cdot g(n) = c \cdot g(n),
\]

where in the last line we plugged in the definition of $f$ and $g$. From the definition of big-Oh, $f(n) = O(g(n))$.  

3
(b) Let \( c = 4 \) and \( n_0 = 3 \). Then for all \( n \geq n_0 \), we have

\[
\begin{align*}
3 & \leq n \\
3n & \leq n^2 \\
12n & \leq 4n^2 \\
4n & \leq 4(n^2 - 2n) \\
f(n) & \leq 4 \cdot g(n),
\end{align*}
\]

where in the last line we have plugged in the definition of \( f \) and \( g \). From the definition of big-Oh, \( f(n) = O(g(n)) \).

**Notes.** For (a), any \( n_0 \geq 6 \) works, and for (b) and \( c \geq 4 \) works. The most common error was only verifying the inequality for \( n_0 \), rather than all \( n \geq n_0 \). While some argument can be made about the growth of the two functions that can complete this proof, it is unnecessarily complicated. It is also not a good practice to assume that once an inequality holds for \( n_0 \), it will hold for all \( n \geq n_0 \), since this is not always the case.

5. (3 pt.) Prove that \( 10^n \) is not \( O(2^n) \).

*We are expecting: A formal proof.*

**SOLUTION:** Suppose towards a contradiction that \( 10^n = O(2^n) \). Then from the definition of big-Oh, there is some \( n_0, c \) so that for all \( n \geq n_0 \), \( 10^n \leq c \cdot 2^n \). Taking logs of both sides and using the fact that \( \log_2(10) \geq 3 \), this implies that

\[
\forall n \geq n_0, \ 3n \leq n \cdot \log_2(10) \leq \log_2(c) + n,
\]

which after re-arranging implies that

\[
\forall n \geq n_0, \ n \leq \frac{\log_2(c)}{2}.
\]

However, this is clearly false: for example,

\[
n = \max\left\{\left\lfloor \frac{\log_2(c)}{2} \right\rfloor, n_0 \right\} + 1
\]

is an example of an \( n \geq n_0 \) for which this is not true. This is contradiction, and therefore \( 10^n \) is not \( O(2^n) \).

**Notes.** Another approach is to divide by \( 2^n \) before taking logs, which gives us \( 5^n \leq c \) for all \( n \geq n_0 \). The same type of argument follows afterwards.
Problems
You may talk with your fellow CS161-ers about the problems. However:

- Try the problems on your own before collaborating.
- Write up your answers yourself, in your own words. You should never share your typed-up solutions with your collaborators.
- If you collaborated, list the names of the students you collaborated with at the beginning of each problem.

6. [Fibonacci] (5 pt.)
The Fibonacci sequence $F_0, F_1, \ldots$, is defined by $F_0 = 0, F_1 = 1,$ and $F_n = F_{n-1} + F_{n-2}$ for $n \geq 2$. For example, the first several Fibonacci numbers are 0, 1, 1, 2, 3, 5, 8, . . . . Show by induction that

(a) $F_n = O(2^n)$
(b) $F_{2n} = \Omega(2^n)$.

Together, parts (a) and (b) imply that $F_n = 2^{\Theta(n)}$.

[We are expecting: For each part, a formal proof by induction. Make sure to clearly label your inductive hypothesis, base case, inductive step, and conclusion.]

SOLUTION:
(a) We will show by induction that $F_n \leq 2^n$ for all $n \geq 0$. This will imply that $F_n = O(2^n)$ by the definition of big-Oh, with $c = 1$ and $n_0 = 1$.

- **Inductive Hypothesis:** $F_n \leq 2^n$.
- **Base cases:** We establish the inductive hypothesis for $n = 0$ and $n = 1$: we have $F_0 = 0 \leq 2^0$ and $F_1 = 1 \leq 2^1$.
- **Inductive step:** We will show that, for any $k \geq 1$, if the inductive hypothesis is true for $n = 0, \ldots, k$, then it is true for $n = k + 1$. Suppose the inductive hypothesis is true for $n = k - 1$ and $n = k$. i.e., $F_{k-1} \leq 2^{k-1}$ and $F_k \leq 2^k$. Then

$$F_{k+1} = F_k + F_{k-1} \leq 2^{k-1} + 2^k \leq 2 \cdot 2^k = 2^{k+1}.$$ 

Hence, the inductive hypothesis is also true for $n = k + 1$.
- **Conclusion:** Therefore, the inductive hypothesis holds for all $n \geq 0$, which is what we wanted to show.

(b) We will show by induction that $F_{2n} \geq \frac{1}{2} \cdot 2^n$, for all $n \geq 1$. This will imply that $F_{2n} = \Omega(2^n)$ by the definition of big-Omega, with $c = \frac{1}{2}$ and $n = 1$.

- **Inductive Hypothesis:** $F_{2n} \geq \frac{1}{2} \cdot 2^n$.
- **Base cases:** We establish the inductive hypothesis for $n = 1$: we have that $F_2 = 1 = \frac{1}{2} \cdot 2^1$. 


• **Inductive step:** We will show that, for any $k \geq 1$, if the inductive hypothesis is true for $n = k$, then it is true for $n = k + 1$. Suppose the inductive hypothesis is true for $n = k$, i.e., $F_{2k} \geq \frac{1}{2} \cdot 2^k$. Then, using the fact that $F_{2k+1} \geq F_{2k}$, we have

$$F_{2(k+1)} = F_{2k+1} + F_{2k} \geq 2 \cdot F_{2k} \geq 2 \cdot \frac{1}{2} \cdot 2^k = \frac{1}{2} \cdot 2^{k+1}.$$  

Hence, the inductive hypothesis is also true for $n = k + 1$.

• **Conclusion:** By induction, the inductive hypothesis holds for all $n \geq 1$, which is what we wanted to show.

**Notes.** There are several ways to solve this problem. As noted in the hint that we posted, there are many options for $c$ and $n_0$ for each part. For instance, another natural option for part (b) is to take $c = 1$ and $n_0 = 3$. Like part (b), part (a) can also be modified to work with one base case by using the fact that $F_n$ is nondecreasing.

The most common error was including only one base case, but writing a proof of the inductive step that relied on two inductive hypotheses. An easy way to check that your inductive proofs are logically correct is to try to see how the inductive step works for the case immediately after the base case. If your assumptions would need to include something more than your base case, you may need to add another base case.
You’ve been communicating with some aliens, and learning about their arithmetic. Interestingly, they have the same base 10 numerical system, addition, and subtraction, but their system of multiplication is completely different. The symbol they use is $\otimes$, and they’ve sent you the 1-digit multiplication table that they memorize in grade school. Here are some of the highlights:

<table>
<thead>
<tr>
<th>$a$</th>
<th>$b$</th>
<th>$a \otimes b$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>19</td>
</tr>
<tr>
<td>3</td>
<td>5</td>
<td>49</td>
</tr>
<tr>
<td>9</td>
<td>9</td>
<td>243</td>
</tr>
</tbody>
</table>

and so on. After some time, you discover that their multiplication corresponds to the following in human arithmetic:

$$x \otimes y = x^2 + xy + y^2.$$  

You want to give your alien friends an efficient way to compute the alien product of two $n$-digit numbers, without referencing or explaining any human multiplication to them (even implicitly as a sum). For example, you can’t tell them to compute $x^2$, $xy$, and $y^2$ separately since they don’t understand what these terms mean.

Find an algorithm to compute the alien product of two $n$-digit numbers that runs in $O(n \log_3 3)$ time. For example, your algorithm might take as input $x = 1234$ and $y = 4321$, and it should return $x \otimes y = 25525911$.

You may assume that $n$ is a power of 2, and that the alien computers can shift digits, add, and subtract just like human computers can. You may also assume that the alien computers have easy access to the $\otimes$-multiplication table you received.

**[We are expecting]**: Pseudocode for your algorithm and a clear English description of what your algorithm is doing and why it is correct. You do not need to prove that your algorithm is correct.

**SOLUTION:** We will use the fact that, if $x = a \cdot 10^{n/2} + b$ and $y = c \cdot 10^{n/2} + d$, then

$$x \otimes y = (a \cdot 10^{n/2} + b)^2 + (a \cdot 10^{n/2} + b)(c \cdot 10^{n/2} + d) + (c \cdot 10^{n/2} d)^2$$

$$= (a^2 + ac + c^2)10^n + (2ab + 2cd + bc + ad)10^{n/2} + (b^2 + bd + d^2)$$

$$= (a \otimes c)10^n + (2ab + 2cd + bc + ad)10^{n/2} + (b \otimes d)$$

Ignoring the powers of 10 in the above expansion, we can find that

$$(a + b) \otimes (c + d) = (a + b)^2 + (a + b)(c + d) + (c + d)^2$$

$$= (a^2 + ac + c^2) + (2ab + 2cd + bc + ad) + (b^2 + bd + d^2)$$

$$= (a \otimes c) + (2ab + 2cd + bc + ad) + (b \otimes d)$$

$$\Rightarrow (2ab + 2cd + bc + ad) = (a + b) \otimes (c + d) - (a \otimes c) - (b \otimes d)$$

and so

$$x \otimes y = (a \otimes c)10^n + ((a + b) \otimes (c + d) - (a \otimes c) - (b \otimes d))10^{n/2} + (b \otimes d).$$

Thus, we can compute $x \otimes y$ using only three alien multiplications of $n/2$-digit integers, which means we can do the same thing that the Karatsuba algorithm does for human multiplication. Our pseudocode is:
Algorithm 1: AlienKaratsuba: finds $x \otimes y$.

**Input:** $n$-digit integers $x$ and $y$, where $n$ is a power of 2.

**Output:** $x \otimes y$

1. If $n \leq 1$ then
   - Look up the value of $x \otimes y$ in the table that all aliens are made to memorize in grade school and return it.
   - Write $x = a 10^{n/2} + b$ and $y = c 10^{n/2} + d$
   - $A_1 \leftarrow $ AlienKaratsuba$(a, c)$
   - $A_2 \leftarrow $ AlienKaratsuba$(b, d)$
   - $A_3 \leftarrow $ AlienKaratsuba$(a + b, c + d)$
2. Return $\sigma(A_1, n) + \sigma(A_3 - A_1 - A_2, n/2) + A_2$, where $\sigma(z, n)$ shifts the integer $z$ to the left by $n$ digits.

8. **[Skyline] (13 pt.)** You are handed a scenic black-and-white photo of the skyline of a city. The photo is $n$-pixels tall and $m$-pixels wide, and in the photo, buildings appear as black (pixel value 0) and sky background appears as white (pixel value 1). In any column, all the black pixels are below all the white pixels. In this problem, you will design an efficient algorithm that finds the location of a tallest building in the photo.\(^1\)

The input is an $n \times m$ matrix, where the buildings are represented with 0s, and the sky is represented by 1s. The output is an integer representing the location of a tallest building. For example, for the input $6 \times 5$ matrix below, a tallest building has height 5 and is in location 1 (assuming we are 0-indexing). Thus the output is 1.

\[
\begin{pmatrix}
1 & 1 & 1 & 1 & 1 \\
1 & 0 & 1 & 1 & 1 \\
1 & 0 & 1 & 0 & 0 \\
0 & 0 & 1 & 0 & 0 \\
0 & 0 & 1 & 0 & 0 \\
0 & 0 & 0 & 0 & 0
\end{pmatrix}
\]

(a) (5 pt.) Find an algorithm that finds a tallest building in time $O(m \log n)$.

**We are expecting:** Pseudocode, a clear English description of what your algorithm is doing, and a brief justification of the runtime. No proof of correctness is required.

(b) (5 pt.) Find an algorithm that finds a tallest building in time $O(m + n)$.

**We are expecting:** Pseudocode, a clear English description of what your algorithm is doing, and a brief justification of the runtime. No proof of correctness is required.

(c) (3 pt.) For some values of $(n, m)$ the algorithm from part (a) is more efficient, while for others, the algorithm from part (b) is more efficient. For each of the values of $n$ in terms of $m$ below, determine which of the above algorithms is more efficient (or that they are equally efficient) in terms of big-Oh notation. The case $n = m$ is filled in as an example. If it is helpful, the \LaTeX{}code for this table is provided at the end of this assignment.

<table>
<thead>
<tr>
<th>$n$ = ?</th>
<th>$\sqrt{m}$</th>
<th>$\frac{m}{\log m}$</th>
<th>$m$</th>
<th>$m \log m$</th>
<th>$m^2$</th>
<th>$2^m$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Runtime for (a)</td>
<td>$O(m \log m)$</td>
<td>$O(m)$</td>
<td>(b)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Runtime for (b)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Which is better?</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**We are expecting:** For each value of $n$ in terms of $m$, the best big-Oh runtime you can guarantee for each of your algorithms from (a) and (b), and a conclusion about which is asymptotically more

---

\(^1\)It could be that there are multiple tallest buildings that all have the same height; in this case, your algorithm should should return any one of them.
efficient. You do not need to give any formal proofs, but your runtimes should be in the simplest terms possible.

(d) (2 BONUS pt.) Combine your algorithms from parts (a) and (b) to make a better algorithm. That is, for any relationship between $m$ and $n$, your algorithm should be asymptotically no worse than either your algorithm from (a) or your algorithm from (b), and there should be some settings where it is asymptotically better. You should present your algorithm, its running time, and an example of a function $f$ so that when $n = f(m)$, your algorithm is asymptotically strictly better than both $O(n + m)$ and $O(m \log n)$.

[We are expecting: Nothing, this part is not required. To get the bonus points, you should include pseudocode, the best big-Oh running time for your algorithm that you can come up with, and a function $f$ as described above. All three parts should be accompanied by a clear English description/justification.]

SOLUTION:

(a) One solution is to use binary search on the columns to find the highest point $h$ where there are any buildings whose height is $h$. The pseudocode is as follows:

```python
def TallestTower( an n x m matrix A):
    if n == 1:
        for j = 0, ..., m-1:
            if A[0,j] == 0, return j
        return "There are no buildings in this city!"
    if A[n/2,:] has more than one zero:
        return TallestTower( A[:n/2, :) )
    else:
        return TallestTower( A[n/2:, :) )
```

We note that there are multiple natural ways to do this part. Another one is to do a binary search on each building to find the height, and then return the largest.

(b) The basic idea is as follows: Starting with the bottom left corner, move upwards until you see a 1, and then move right until we see a 0, and repeat until we reach the rightmost column. The tallest building is height $n - i - 1$ where $i$ is the row we finish on. Each time we move either upwards or rightwards, so at most $m + n$ steps are made.

The pseudocode is as follows:

```python
def TallestTower( an n x m matrix A):
    best = None
    j = 0
    for i in 0, ..., m-1:
        if A[i,j] == 0:
            best = i
        while A[i,j] = 0:
            j += 1
    return best
```

(c) The big-Oh running times are as follows:

<table>
<thead>
<tr>
<th>$n = ?$</th>
<th>100</th>
<th>$\sqrt{m}$</th>
<th>$\frac{m}{\log m}$</th>
<th>$m$</th>
<th>$m \log m$</th>
<th>$m^2$</th>
<th>$2^m$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Runtime for (a)</td>
<td>$O(m)$</td>
<td>$O(m \log m)$</td>
<td>$O(m)$</td>
<td>$O(m \log m)$</td>
<td>$O(m \log m)$</td>
<td>$O(m \log m)$</td>
<td>$O(m^2)$</td>
</tr>
<tr>
<td>Runtime for (b)</td>
<td>$O(m)$</td>
<td>$O(m)$</td>
<td>$O(m)$</td>
<td>$O(m)$</td>
<td>$O(m \log m)$</td>
<td>$O(m \log m)$</td>
<td>$O(m^2)$</td>
</tr>
<tr>
<td>Which is better?</td>
<td>same</td>
<td>(b)</td>
<td>(b)</td>
<td>(b)</td>
<td>(b)</td>
<td>(b)</td>
<td>(a)</td>
</tr>
</tbody>
</table>

(d) The intuition from part (c) is that the strategy from part (a) is more efficient when $n$ is much larger than $m$, and otherwise the strategy from part (b) is more efficient. Since each iteration of the algorithm from part (a) reduces $n$ by a factor of 2, we can get an
improved algorithm by using the binary search strategy from (a) until \( n \) is close to \( m \) (i.e., \( m \leq n \leq 2m \)), and then use the algorithm from part (b).
The pseudocode is as follows:

```python
def TallestTower( an n x m matrix A ):
    if n <= 2m:
        best = None
        j = 0
        for i in 0, ..., m-1:
            if A[i,j] == 0:
                best = i
            while A[i,j] == 0:
                j += 1
        return best
    else:
        if A[n/2, :] has more than one zero:
            return TallestTower( A[:n/2, :] )
        else:
            return TallestTower( A[n/2:, :] )
```

The binary search performs \( O(\log \frac{n}{m}) \) iterations, so the runtime for the first step is \( O(m \log \frac{n}{m}) \). The runtime for the second step is \( O(m) \), so the overall runtime is \( O(m \log \frac{n}{m}) \). This runtime is asymptotically better than \( O(n + m) \) and \( O(m \log n) \) when \( n = m \log m \).

**Note:** Strictly speaking, when \( n \leq m \), there are no iterations of the binary search, and the runtime above may be negative. To account for this, we can say that the overall runtime is \( O(\min\{m, m|\log \frac{n}{m}\}) \).

**Common Errors.** The most common errors were in part (c). The most common issue was not using the fact that \( O(\log \frac{n}{m}) = O(\log(\log m)) = O(\log m) \).

A number of students claimed to have an \( O(m + \log n) \) algorithm for part (d), which works like part (b), but then scales the buildings by using binary search instead of climbing one row at a time. While this may be effective in practice, it is not effective in the worst case when the difference between the heights of the buildings is small (e.g., increasing by 1 or 2 each time). In this case, the algorithm does nearly a full binary search in every iteration, so the worst case runtime is actually \( O(m \log n) \).

**Alternate Solutions.** Some students found other creative ways of solving part (d).

One approach similar to the solution above is to note that we can use the algorithm in part (b) to determine the height of the tallest building within any \( m \times m \) block in \( O(m) \) time. This means that we can do binary searches on blocks of size \( m \times m \), rather than just rows for the same cost. In the same way as the solution above, this results in an \( O(m \log \frac{n}{m}) \) algorithm.

Another solution is to proceed like part (b), but scale each building with steps that are increasing powers of 2 (note that this is different from a binary search). If the heights of the buildings are \( h_1, h_2, ..., h_m \) (where we suppose that the heights are increasing). With some effort, it can be shown that the work done to find the height of the \( k \)th building is \( O(\log(h_k - h_{k-1})) \), and using the concavity of \( \log x \), it can be shown that the overall work is \( O(m \log \frac{h_m}{h_1}) = O(m \log \frac{n}{m}) \). The details of this solution go beyond the scope of the class, but it goes to show that there can be a number of ways to solve a problem!
Feedback

There's no “correct” answer here, and it is completely anonymous.

9. (1 pt.) Please fill out the following poll, which asks about your expectations for the course:

https://forms.gle/qWDZLAg3p3JU2xbS6

Did you fill out the poll?

[We are expecting: The answer “yes.”]

\LaTeX code for problem 8(c)

Here is code that generates a table like the one in 8(c). Put your answers between the \&'s to fill in the table.

\begin{center}
\begin{tabular}{c|c|c|c|c|c|c|c}
\hline
$n = \ ?$ & 100 & $\sqrt{m}$ & $\frac{m}{\log m}$ & $m$ & $m \log m$ & $m^2$ & $2^m$ \\
\hline
Runtime for (a) & & & & $O(m \log m)$ & & & \\
Runtime for (b) & & & & $O(m)$ & & & \\
Which is better? & & & & (b) & & & \\
\end{tabular}
\end{center}

Note: if after filling in the table, it is too wide for the page, you can either use a command like \footnotesize before you start the table and \normalsize afterwards to put it in a smaller font; or you can edit the code above to break the table up across multiple lines.