CS 61B Asymptotic Analysis Fall 2018

Disclaimer: This discussion worksheet is fairly long and is not designed to be finished in a single section. Some of these questions of the level that you might see on an exam and are meant to provide extra practice with asymptotic analysis.

Best and Worse Case

Before we begin it is important to understand what we mean by the best case and worst case runtimes of a function, especially as it pertains to asymptotically bounding functions. First, we remind ourselves that when considering asymptotics we will always be concerned with the runtime of the program as the size of the input grows very large (e.g. as the length of an array increases to infinity). This means that the best or worst case should never be defined by considering $N = 1$ or some other small value of N. For example you should never say something to the extent of "the best case is when the array is length 1 and then the function runs in constant time".

Instead when we describe the best and worst case we should be seeking different configurations of large inputs which might cause the runtime to somehow change. Also it is typical that we will be able to provide a tight bound using Θ notation, as we are essentially saying is "if we have this configuration of the input what will the runtime be?". When we condition on the configuration of the input like this we can often be more precise in our runtime bounds than we could be without conditioning. Additionally when we do this we will often have two bounds (one for the best case and one for the worst case) and might say something to the effect of "the function `f` is $\Theta(N)$ in the best case and $\Theta(N^2)$ in the worst case.

Sum Important Sums

In runtime problems there are several sums or classes of sums that will appear fairly often. Here we will take a moment to briefly explain how they can be derived.

- **Sum of the first N integers:** For this we will make the admissible assumption that N is even (for asymptotic analysis this will not change our results as we are concerned with approximately how the function grows, not a precise calculation). When we do this we can group the first term with the last term, the second term with the second to last term, etc. In doing so we will form $N/2$ pairs which will each sum to $N+1$. Thus we have:

$$1 + 2 + 3 + \cdots + (N-2) + (N-1) + N$$

$$= (1+N) + (2 + (N-1)) + (3 + (N-2)) + \cdots$$
$$= (N+1) + (N+1) + (N+1) + \cdots$$
$$= \frac{N(N+1)}{2} \in \Theta(N^2)$$

- **Sum of the powers of 2 up to N:** We will use a general technique to solve for the value of this geometric series. First we will set the series equal to a variable S, then we will factor out a multiple of the ratio, in this case 2, from all but the first term of the series. Next we will perform a substitution, and solve for the value of S. Thus we have:

$$1+2+4+8+\cdots+N/2+N=S$$
$$1+2(1+2+4+\cdots+N/4+N/2)=S$$
$$1+2(S-N)=S$$
$$S=2N-1\in\Theta(N)$$

- **General geometric series:** The above was a specific case of a geometric series, but we will now consider the generic case where the ratio is r and the last term is some function of N, $f(N)$. We will follow the same basic process as above:

$$1+r+r^2+r^3+\cdots+f(N)/r+f(N)=S$$
$$1+r(1+r+r^2+\cdots+f(N)/r^2+f(N)/r)=S$$
$$1+r(S-f(N))=S$$
$$(r-1)S=rf(N)-1$$
$$S=\frac{rf(N)}{r-1}\in\Theta(f(N))$$

This is a powerful and general result that means that for any geometric series we see in a runtime problem, the runtime will run with respect to whatever the last term is. In order to apply this result it is important to understand where it comes from and when it should be applied, so as to prevent using it when it does not actually apply.

Recursive Runtime Approaches

Runtimes of recursive functions are typically a bit harder to visualize. Especially when you are first learning one of the most helpful things to understand the runtime of a recursive problem is to consider a tree which represents all of the function calls. in doing this you might wish to determine the following:

1. Determine the height of the tree. There are various ways in which to do this which we be shown throughout the problems below.
2. Determine the branching factor. This is typically the number of recursive function calls that are made from each call of the function. You can also use the branching factor in determining the number of nodes at any given layer of the tree.
3. Determine the amount of work done at each node relative to the input size. We should be careful here as this may or may not be the same amount of work being done at every node in a given level of the tree.
4. Calculate the entire amount of work being done in the entire function call by:

$$\sum_{\text{layers in the tree}} \frac{\text{\# nodes}}{\text{layer}} \cdot \frac{\text{amount of work}}{\text{1 node}}$$

1 More Running Time

Give the worst case and best case running time in $\Theta(\cdot)$ notation in terms of M and N.

(a) Assume that `slam()` $\in \Theta(1)$ and returns a boolean.

```java
public void comeon(int M, int N) {
    int j = 0;
    for (int i = 0; i < N; i += 1) {
        for (; j < M; j += 1) {
            if (slam(i, j))
                break;
        }
    }

    for (int k = 0; k < 1000 * N; k += 1) {
        System.out.println("space jam");
    }
}
```

- **Best case:** We can see that there is a doubly nested for loop. The inner loop contains an if statement and breaks out of the inner loop if `slam(i,j)`. The best case behaviour will be when the inner loop always immediately breaks (this will happen if `slam(i,j)` always returns true) causing the inner loop to always run in constant time. In this case we will still need to iterate through the entire loop from $\texttt{i} = 0$ to `i=N`, which will take linear time. Additionally there is a second loop that will also run in linear time regardless of the conditions we are imposing on `slam(i,j)`. Therefore the runtime of `comeon(M,N)` will be $\Theta(N)$ in the best case.
- **Worst case:** Contrary to the above, the worst case behavior will correspond to when the function `slam(i,j)` always returns false, which will cause the loop to never break early. For this we notice that the variable `j` is declared outside of the for loop, instead of inside as we would typically expect. This means that `j` will only take on the values from 0 to `M` and similarly `i` will only take on the values from 0 to `N`. One way to visualize the total amount of work done is to look at the number of times the variables `i` or `j` are incremented in this worst case. The varibale `i` will be incremented `N` times and the varibale `j` will be incremented `M` times. This means the total amount of work done by the first part of the code will be $\Theta(M+N)$. We additionally will still have the second loop which will run in linear time with respect to `N`. Therefore the runtime of `comeon(M, N)` will be $\Theta(M+N)$ in the worst case.

Answer: For `comeon(M,N)` the runtime is $\Theta(N)$ in the best case and $\Theta(M+N)$ in the worst case.

(b) *Exam Practice:* Give the worst case and best case running time in $\Theta(\cdot)$ notation in terms of N for `find`.

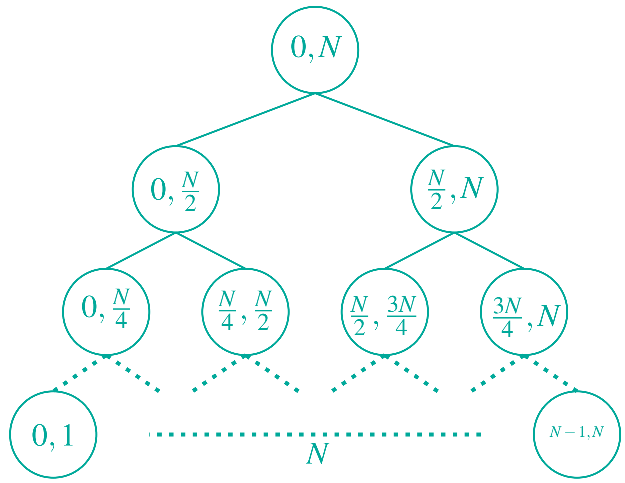
```java
public static boolean find(int tgt, int[] arr) {
    int N = arr.length;
    return find(tgt, arr, 0, N);
}
private static boolean find(int tgt, int[] arr, int lo, int hi) {
    if (lo == hi || lo + 1 == hi) {
        return arr[lo] == tgt;
    }
    int mid = (lo + hi) / 2;
    for (int i = 0; i < mid; i += 1) {
        System.out.println(arr[i]);
    }
    return arr[mid] == tgt || find(tgt, arr, lo, mid)
                           || find(tgt, arr, mid, hi);
}
```

Note: This questions is perhaps poorly placed relative to the difficulty of this question. It is recommended to first understand some of the other questions following this question, before returning to work on this question.

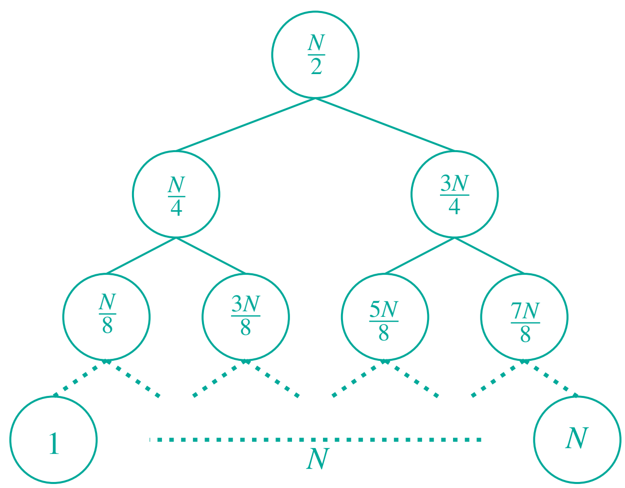
This question is fairly tricky, and for most intents and purposes this code would be considered buggy, but nonetheless we can consider the runtime of this question. First we need to define what we should consider our argument size to be. We see that for each call we are passing in the same array, so this will always be a fixed size N and does not represent the argument size of a function call. The amount of the array to be iterated over is determined by the variable `mid = (hi + lo) / 2`. We would expect the size of the chunk of the array that we are looking at to be defined by the difference between `lo` and `hi`, but as we stated above the code is buggy and the for loop begins at position 0 not at position `lo` as would typically be expected. As such there is a linear amount of work done at each call to this function with respect to the value of `mid`.

Now we see that in each function call there will be up to two recursive calls and the difference between `lo` and `hi` will be geometrically decreased by a factor of two each time. Depending on the evaluation of the expression `arr[mid] == tgt`, the boolean expression might be short circuited, causing neither of the two recursive calls to be evaluated. Thus the worst case behavior will be when the boolean expression is never short circuited, and the best case behavior will be when the boolean expression is short circuited immediately (on the first call to `find`). In the worst case the height of the tree will be $\log_2(N)$. A more detailed explanation for how this can be derived is given in the following questions.

We will begin with the worst case when the short circuiting never happens and there are always two recursive calls made. At the first function call we will have `mid` $= (0+N)/2 = N/2$, so we will do $N/2$ work. At each function call we will make two recursive calls, one with `hi = hi` and the other with `hi = (lo + hi) / 2`. So at the next level we will have two nodes, with $N/4$ and $3N/4$ work being done. We can envision a tree of recursive function calls as below where the nodes contain values of both `lo` and `hi` separated by commas.



We can translate from this tree into a tree where the nodes correspond to value of `mid` or the amount of work done in each of the nodes/recursive calls which leaves us with the following tree. This can be created by computing the `mid` value based off of the `lo` and `hi` values in the tree above.



With this in mind we can now extract the total amount of work being done out of this tree, by looking at each level of the tree and determining the amount of work being done at each level. By summing the amounts per each layer then we will end up with the total amount of work being done. We can summarize the amount of work done at each level in the following table:

Layer Number	Sum	Total Work
1	$\frac{N}{2}$	$\frac{N}{2}$
2	$\frac{N}{4}+\frac{3N}{4}$	N
3	$\frac{N}{8}+\frac{3N}{8}+\frac{5N}{8}+\frac{7N}{8}$	$2N$
4	$\frac{N}{16}+\frac{3N}{16}+\frac{5N}{16}+\frac{7N}{16}+\frac{9N}{16}+\frac{11N}{16}+\frac{13N}{8}+\frac{15N}{16}$	$4N$
$\vdots$	$\vdots$	$\vdots$
$\log_2(N)$	$1+2+3+\cdots+N$	$\Theta(N^2)$

We notice in the last row of our table, which corresponds to the lowest level of the tree we will have N function calls, where the first one will look at only the first element, the second will look at the first two, etc. This continues until the last function call which will look at all N elements in the array, so this level corresponds to the summation of the first N integers which we found above to be in $\Theta(N^2)$. To find the total amount of work done we simply have to sum the last column which leaves us with the following.

$$N/2+N+2N+4N+\cdots+N^2$$

$$=N/2+N(1+2+4+\cdots+N)\in\Theta(N^2)$$

For this last step we also make use of summation of the powers of two up to N we found above. All of this results in worst case $\Theta(N^2)$ runtime.

We will now consider the best case runtime. As specified above the best case will be when there are no recursive calls made as the boolean expression of function calls short circuits after the evaluation of `arr[mid] == tgt`. In this case we still must iterate through the array starting at 0 and ending at `mid = (lo + hi) / 2`. Since we are looking at the first call to `find` we will have that `lo = 0` and `hi = N` which means that `mid = (0 + N) / 2 = N / 2`. This means that this loop will do a linear amount of work with respect to `N / 2`, which will be a $\Theta(N)$ operation. Again in the best case `arr[mid] == tgt` and we short circuit, so we will not make either of the recursive calls. This linear operation is all of the work done in the best case, thus `find` is $\Theta(N)$ in the best case.

Answer: For `find` the best case runtime is $\Theta(N)$ and the worst case runtime is $\Theta(N^2)$.

2 Recursive Running Time

For the following recursive functions, give the worst case and best case running time in $\Theta(\cdot)$ notation.

(a) Give the running time in terms of N.

```java
public void andslam(int N) {
    if (N > 0) {
        for (int i = 0; i < N; i += 1) {
            System.out.println("bigballer.jpg");
        }
        andslam(N / 2);
    }
}
```

We will follow the process as described above.

1. First of all we can see that the argument to the recursive call is halved every time so that the calls will have values $N, N/2, N/4, \ldots, 1$. We are diving the original argument `N` by 2 until the argument passed in becomes 1 (technically the base case will be one function call after this when the argument passed in is 0 as 1 / 2 = 0 with Java integers). This means that the height of the tree will be $\log_2(N)$. Another way to see this is if we define h as the height of the tree, then we can see that $N/2^h = 1$ (if we know the value at the top of the tree is N and the bottom of the tree is 1, then how many times should we divide N by 2 to reach 1). Solving for h we have $N = 2^h \implies h = \log_2(N)$.
2. We can see that in the `andslam` function there is only one recursive function call made, so the branching factor will be 1.
3. Inside each function call there is a single for loop that will do linear work with respect to the input argument, e.g. the first function call will do N work, the second call will do $N/2$ work, then $N/4$ work, etc. From the branching factor, the height of the tree, and the observation of how the input size is changing we can draw the following tree to represent the amount of work being done. Here the values inside the nodes are the size of the arguments passed into each successive function call.

4. The total amount of work being done is the summation of all of the work being done in this tree. We notice that the amount of work being done in a given node is the same as the argument passed in (as a linear amount of work is being done in each of the function calls). This means that the values in the above tree will also correspond to the work being done in each node. With a branching factor of 1 we will have the total amount of work being done as $N+N/2+N/4+\cdots+4+2+1$. From the above work we know that this summation is $\Theta(N)$. For this function we see that regardless of any configuration of the input size the function will perform the same amount of work; there is no difference between the best and worst case. Thus we can say that `andslam` is $\Theta(N)$ or the function is best and worst case $\Theta(N)$.

Answer: For `andslam(N)` the runtime is $\Theta(N)$ in the best and worst case.

(b) Give the running time for `andwelcome(arr, 0, N)` where N is the length of the input array `arr`.

```java
public static void andwelcome(int[] arr, int low, int high) {
    System.out.print("[ ");
    for (int i = low; i < high; i += 1) {
         System.out.print("loyal  ");
    }
    System.out.println("]");
    if (high - low > 0) {
        double coin = Math.random();
        if (coin > 0.5) {
            andwelcome(arr, low, low + (high - low) / 2);
        } else {
            andwelcome(arr, low, low + (high - low) / 2);
            andwelcome(arr, low + (high - low) / 2, high);
        }
    }
}
```

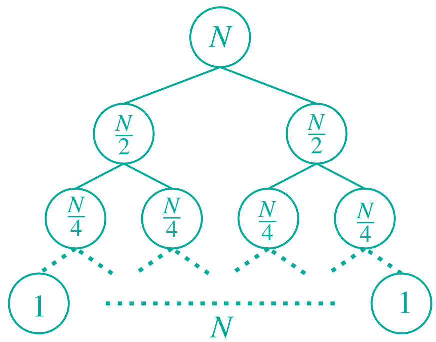
For this question we will have to determine the best case and the worst case of the function when calculating the runtime of the program. The condition of what will comprise of the best and worst case will be conditioned on the behavior of the calls to `Math.random()`. The result, stored as `coin` will determine whether there will be either one or two recursive calls. The best case will correspond to the probabilisitcally unlikely outcome that every single call to `Math.random` will produce a result that is greater than 0.5, thus there will always be only recursive call made. The worst case behavior corresponds to the equally unlikely case that every single call to `Math.random` will produce a result that is less than 0.5, and as such there will always be two recursive calls made. We will analyze the best and worst case below:

- **Worst case:**
  1. This function takes an array, and looks at smaller and smaller chunks of the array as passed defined by the arguments to `low` and `high`. We can see that the size of the chunk considered at each level is geometrically decreasing, so we will consider the full array, then two halves, then four quarters, etc. At the deepest level of the tree, we can think of this being the furthest we can split apart an array, into elements, so

the size of the chunks at the bottom level will be 1. The height of this tree can be calculated as above to be $\log_2(N)$.

2. In the worst case as described above we will always make two recursive calls, which will mean that the branching factor will be 2. Here we can notice that the total number of nodes at a given depth i will be 2^i (where the root node corresponds to depth 1). Thus there will be 1 node at the top level, two nodes at the next, four at the next, etc.

3. The amount of work being done in each of the function calls will be done by the for loop in the beginning of the function. This loop will iterate through the array starting at position `low` and ending at the position `high`, so the amount of work done will be linear with respect to `high - low`. We can visualize the recursive calls using the following tree. Here the values inside the nodes will also correspond to the size of the arguments (for this it will correspond to `high - low`).



4. To summarize the tree we can also construct a table to analyze the amount of work done.

# Nodes	Work per Node	Total Work
1	N	N
2	N / 2	N
4	N / 4	N
⋮	⋮	⋮
N	1	N

Here we can see that at each level of the tree there will be N work being done. We also remember that the height of the tree is $\log(N)$, so the total amount of work being done is $\Theta(N\log(N))$. We could also achieve this result using summations as follows:

$$\sum_{\text{layers in the tree}} \frac{\text{\# nodes}}{\text{layer}} \cdot \frac{\text{amount of work}}{\text{1 node}}$$

$$\sum_{i=0}^{\log(N)} 2^i \cdot \frac{N}{2^i} = \sum_{i=0}^{\log(N)} N \in \Theta(N\log(N))$$

- **Best Case:** We notice that if only one recursive function call is performed the runtime analysis will actually be identical to what we did for the function `andslam`, so in the best case this function will be $\Theta(N)$

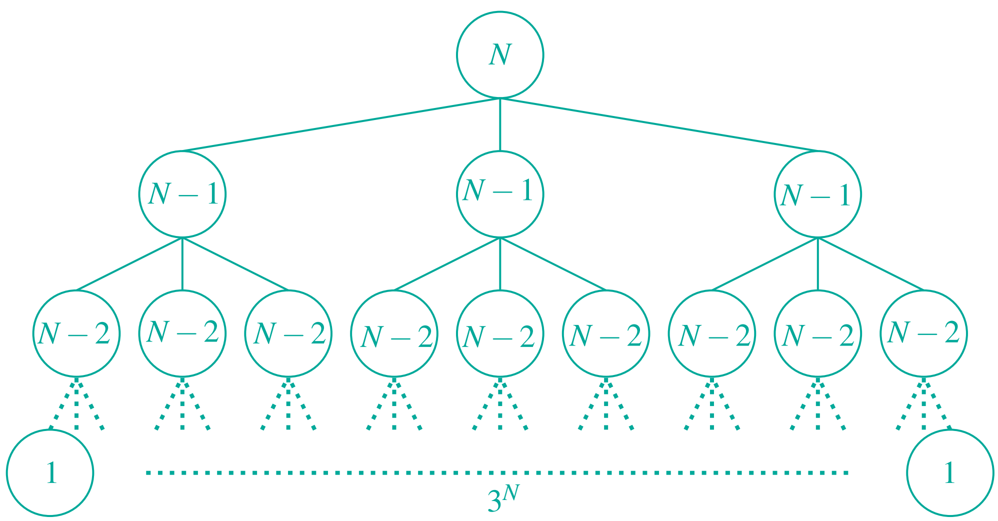
Answer: For `andwelcome(arr, 0, N)` the runtime is $\Theta(N)$ in the best case and $\Theta(N \log N)$ in the worst case.

(c) Give the running time in terms of N.

```
public int tothe(int N) {
    if (N <= 1) {
        return N;
    }
    return tothe(N - 1) + tothe(N - 1) + tothe(N - 1);
}
```

We notice here that regardless of what our input size is, the function will behave the same. As such the best case and the worst case will be the same. We will perform the same steps as we did above.

1. The argument passed into each of the recursive calls will be decremented by one each time. To reach the base case we will have to decrement the argument N times to reach a number that is less than or equal to 1. Therefore the height of the tree will be N.
2. Each function call will make three recursive calls, so we will have a branching factor of three. Similar to above we can see that the number of nodes at a given depth i will be 3^i (where the root node corresponds to depth 1). We can visualize the tree of recursive calls as follows (the values inside of the nodes will be the size of the input, not the amount of work being done).



3. The only work being done in a call to the function will be the checking of the base case, the function calls, and the addition of the results. This will be a constant amount of work, so the overall runtime of the function can be broken down into determining the number of nodes in the tree.
4. We can again first summarize the information from the tree in the following table

Argument	# Nodes	Work per Node	Total Work
N	1	1	1
N - 1	3	1	3
N - 2	9	1	9
$\vdots$	$\vdots$	$\vdots$	$\vdots$
1	3^N	1	3^N

From here we can see that the total amount of work being done is $1+3+9+\cdots+3^N$. Using the above general formula for geometric series we can see that $1+3+9+\cdots+3^N \in \Theta(3^N)$. We can also acheive this result through summations as follows:

$$\sum_{\text{layers in the tree}} \frac{\text{\# nodes}}{\text{layer}} \cdot \frac{\text{amount of work}}{\text{1 node}}$$

$$\sum_{i=1}^{N} 3^i \cdot 1 = 1+3+9+\cdots+3^N \in \Theta(3^N)$$

Answer: For `tothe(N)` the runtime is $\Theta(3^N)$ in the best and worst case.

(d) Give the running time in terms of N

```java
public static void spacejam(int N) {
    if (N == 1) {
        return;
    }
    for (int i = 0; i < N; i += 1) {
        spacejam(N-1);
    }
}
```

This runtime questions is a bit more involved. The first thing that we notice is that the branching factor changes depending on the value of the the input size. We will proceed carefully through the same steps as above.

1. Similar to `tothe` function above, the input size to each recursive call will be decremented by one. This means that the height of the tree will be N.
2. The branching factor will be linear with respect to the input argument. This means that the branching factor for the first call will be N, then $N-1$ for the second, $N-2$ for the third, etc.
3. For each of the function calls the amount of work being done will be linear with respect to the size of the argument, as the for loop will iterate from `i = 0` to `i = N` where N would correspond to the input argument, not necessarily the original value N passed in.
4. For this question we will not draw the tree, as it quickly becomes quite too large to be managed. We will instead use the following table to explain the information we would have represented in the tree. There are a few general trends to notice in the table. First the argument size will be decremented by 1 each time. The number of nodes at any level is equal to the number of nodes at the previous level times the branching factor of the previous

level (we also know that there will be only one node at the top level). Lastly the work done per node will be equivalent to the argument. Thus we have the following:

Argument	# Nodes	Work per Node	Total Work
N	1	N	N
N - 1	N	$N-1$	$N(N-1)$
N - 2	$N(N-1)$	$N-2$	$N(N-1)(N-2)$
$\vdots$	$\vdots$	$\vdots$	$\vdots$
3	$N!/3$	3	$N!/2$
2	$N!/2$	2	$N!$
1	$N!$	1	$N!$

From here we can see that the total amount of work being done is as follows:

$$N+N(N-1)+N(N-1)(N-2)+\cdots+\frac{N!}{1\cdot 2\cdot 3}+\frac{N!}{1\cdot 2}+\frac{N!}{1}+\frac{N!}{1}$$

$$=\frac{N!}{(N-1)!}+\frac{N!}{(N-2)!}+\frac{N!}{(N-3)!}+\cdots+\frac{N!}{3!}+\frac{N!}{2!}+\frac{N!}{1!}+\frac{N!}{0!}$$

$$=\sum_{i=0}^{N-1}\frac{N!}{i!}$$

$$=N!\cdot\sum_{i=0}^{N-1}\frac{1}{i!}$$

We notice that after factoring out the $N!$ we will be left with a sum which corresponds to the power series for e^x with $x=1$ ($e^x=\sum_{i=0}^{\infty}\frac{x^i}{i!}$). This means that as N increases to infinity this sum will converge to e which results in the total amount of work being done as follows:

$$\text{total work}=N!\cdot\sum_{i=0}^{N-1}\frac{1}{i!}=N!\cdot e\in\Theta(N!)$$

Final answer: For `tothe(N)` the runtime is $\Theta(N!)$ in the best and worst case.

Note: This runtime question asks a bit more as it requires you to know what the power series for e^x is to provide a tight bound. For tests, we would likely either provide you the summation necessary, or expect you to use other techniques in order to produce a looser bound.

3 Hey you watchu gon do?

For each example below, there are two algorithms solving the same problem. Given the asymptotic runtimes for each, is one of the algorithms **guaranteed** to be faster? If so, which? And if neither is always faster, explain why. Assume the algorithms have very large input (so N is very large).

(a) Algorithm 1: $\Theta(N)$, Algorithm 2: $\Theta(N^2)$

(b) Algorithm 1: $\Omega(N)$, Algorithm 2: $\Omega(N^2)$

(c) Algorithm 1: $O(N)$, Algorithm 2: $O(N^2)$

(d) Algorithm 1: $\Theta(N^2)$, Algorithm 2: $O(\log N)$

(e) Algorithm 1: $O(N \log N)$, Algorithm 2: $\Omega(N \log N)$

(a) Algorithm 1: $\Theta(N)$ - straight forward, Θ gives tightest bounds

(b) Neither, something in $\Omega(N)$ could also be in $\Omega(N^2)$

(c) Neither, something in $O(N^2)$ could also be in $O(1)$

(d) Algorithm 2: $O(\log N)$ - Algorithm 2 cannot run SLOWER than $O(\log N)$ while Algorithm 1 is constrained on best and worst case by $\Theta(N^2)$.

(e) Neither, Algorithm 1 CAN be faster, but is not guaranteed - it is guaranteed to be "as fast as or faster" than Algorithm 2.

Why did we need to assume that N was large? Asymptotic bounds often only make sense as N gets large, because constant factors may result in a function with a smaller order of growth growing faster than a faster one. For example, take the functions $1000n$ and n^2. n^2 is asymptotically larger than $1000n$, but for small n, it will seem that $1000n$ is larger than n^2.

4 Big Ballin' Bounds

1. Prove the following bounds by finding some constant $M > 0$ and input $N > 0$ for $M \in \mathbb{R}, N \in \mathbb{N}$ such that f and g satisfy the relationship.

   (a) $f \in O(g)$ for $f = 2n$, $g = n^2$
   (b) $f \in \Omega(g)$ for $f = 0.1n$, $g = 40$
   (c) $f \in \Theta(g)$ for $f = \log(n)$, $g = \log(n^a)$, for $a > 0$.

   (a) Let $M = 2$, $N = 1$. Then we see for all $n \geq N$, $|f(n)| \leq |Mg(n)|$.
   (b) Let $M = \frac{1}{400}$, $N = 1$. Then we see for all $n \geq N$, $f(n) \geq |Mg(n)|$. Alternatively, $M = 1$ and $N = 400$ also satisfies the bound.
   (c) For $M = \frac{1}{a}$, $N = 2$, we have $|\log(n)| = |\frac{1}{a} \times \log(n^a)|$. It should be noted that this is an equality, which proves that these functions are asymptotically the same.

2. Answer the following claims with true or false. If false, provide a counterexample.

   (a) If $f(n) \in O(g(n))$, then $500f(n) \in O(g(n))$.
   (b) If $f(n) \in \Theta(g(n))$, then $2^{f(n)} \in \Theta(2^{g(n)})$.

(a) True.

(b) False, consider $f(n) = n$, $g(n) = 2n$. Then $2^{f(n)} = 2^n$, but $2^{g(n)} = 2^{2n} = (2^2)^n = 4^n$.