Recitation on analysis of algorithms

What does it mean? Let $f(n)$ and $g(n)$ be two functions that tell how many statements two algorithms execute when running on input of size $n$. $f(n) = O(g(n))$ means that as $n$ gets larger and larger, any constant $c$ that you use becomes meaningless in relation to $n$, so throw it away.

We showed that $n+6$ is $O(n)$. In fact, you can change the 6 to any constant $c$ you want and show that $n+c$ is $O(n)$.

An algorithm that executes $O(n)$ steps on input of size $n$ is called a linear algorithm.

Understand? Then use informally

1. $\log(n) + 20$ is $O(\log(n))$ (logarithmic)
2. $n + \log(n)$ is $O(n)$ (linear)
3. $n/2$ and $3 \cdot n$ are $O(n)$
4. $n \cdot \log(n) + n$ is $n \cdot \log(n)$ (quadratic)
5. $n^2 + 2n + 6$ is $O(n^2)$ (cubic)
6. $n^3 + n^2$ is $O(n^3)$ (cubic)
7. $2^n + n5$ is $O(2^n)$ (exponential)

Once you fully understand the concept, you can use it informally. Example:

An algorithm takes $(7n + 6) / 3 + \log(n)$ steps. It’s obviously linear, i.e. $O(n)$.

Oft-used execution orders

In the same way, we can prove these kinds of things:

1. $\log(n) + 20$ is $O(\log(n))$ (logarithmic)
2. $n + \log(n)$ is $O(n)$ (linear)
3. $n/2$ and $3 \cdot n$ are $O(n)$
4. $n \cdot \log(n) + n$ is $n \cdot \log(n)$ (quadratic)
5. $n^2 + 2n + 6$ is $O(n^2)$ (cubic)
6. $n^3 + n^2$ is $O(n^3)$ (cubic)
7. $2^n + n5$ is $O(2^n)$ (exponential)

Some Notes on $O()$

- Why don’t logarithm bases matter?
  - For constants $x, y$: $O(\log_x n) = O(\log_y n)$
  - Since $(\log_y n)$ is a constant, $O(\log_x n) = O(\log_y n)$

- Usually: $O(f(n)) \times O(g(n)) = O(f(n) \times g(n))$
  - Such as if something that takes $g(n)$ time for each of $f(n)$ repetitions . . . (loop within a loop)

- Usually: $O(f(n)) + O(g(n)) = O(\max(f(n), g(n)))$
  - “$\max$” is whatever’s dominant as $n$ approaches infinity

Example: $O((n^2 - n)/2) = O(1/2)n^2 + (-1/2)n = O((1/2)n^2)$

Formal definition of $O(n)$

We give a formal definition and show how it is used:

$f(n)$ is $O(g(n))$ iff

There is a positive constant $c$ and a real number $x$ such that:

$f(n) \leq c \cdot g(n)$ for $n \geq x$

Example:

$f(n) = n + 6$
$g(n) = n$

We show that $n+6$ is $O(n)$.
/** Sort b[h..k]. */
public static void mS(Comparable[] b, int h, int k) {
    int e= (h+k)/2;
    mS(b, h, e);
    mS(b, e+1, k);
    merge(b, h, e, k);
}

We will count the number of comparisons mS makes
Use T(n) for the number of array element comparisons that mergeSort makes on an array of size n

/** Sort b[h..k]. */
public static void mS(Comparable[] b, int h, int k) {
    if (h >= k) return;
    int e= (h+k)/2;
    mS(b, h, e);
    mS(b, e+1, k);
    merge(b, h, e, k);
}

Recursion: T(n) = 2 * T(n/2) + T(e+1-h) comparisons made in merge
Simplify: use n is a power of 2

/** Sort b[h..k]. */
public static void mS(Comparable[] b, int h, int k) {
    if (h >= k) return;
    int e= (h+k)/2;
    mS(b, h, e);
    mS(b, e+1, k);
    merge(b, h, e, k);
}

Runtime

<table>
<thead>
<tr>
<th>T(0)</th>
<th>T(1)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Use T(n) for the number of array element comparisons that mergeSort makes on an array of size n

Loop body: O(1).

Number of array element comparisons is the size of the array segment – 1.
Simplify: use the size of the array segment O(k-h) time
Runtime

Thus, for any \( n \) a power of 2, we have

\[
T(1) = 0
\]

\[
T(n) = 2T(n/2) + n \quad \text{for } n > 1
\]

We can prove that

\[
T(n) = n \log_2 n
\]

\( \log n \) means log base 2.

---

Proof by recursion tree of \( T(n) = n \log n \)

\[
T(n) = 2^k T(n/2^k) + n
\]

\( n \) levels

merge time at level

Each level requires \( n \) comparisons to merge. \( \log n \) levels.

Therefore \( T(n) = n \log n \) MergeSort has time \( O(n \log n) \)

---

Proof of \( \log n = \log(2n) - 1 \), \( n \) a power of 2

Since \( n = 2^k \) for some \( k \):

\[
\log(2n) - 1
\]

\[
\log(2^k) - 1
\]

\[
\log(2^k + 2^k) - 1
\]

\[
\log n \text{ means } \log_2 n
\]

Thus, if \( n = 2^k \)

\[
\log n = k
\]

---

MergeSort vs QuickSort

- Covered QuickSort in Lecture
- MergeSort requires extra space in memory
  - The way we’ve coded it, we need to make that extra array \( c \)
  - QuickSort was done “in place” in class
- Both have “average case” \( O(n \log n) \) runtime
  - MergeSort always has \( O(n \log n) \) runtime
  - QuickSort has “worst case” \( O(n^2) \) runtime
- Let’s prove it!
Quicksort

- Pick some "pivot" value in the array
- Partition the array:
  - Finish with the pivot value at some index j
  - Everything to the left of j ≤ the pivot
  - Everything to the right of j ≥ the pivot
- Run Quicksort on the array segment to the left of j, and on the array segment to the right of j

Runtime of Quicksort

- Base case: array segment of 0 or 1 elements takes no comparisons
  \( T(0) = T(1) = 0 \)
- Recursion:
  - partitioning an array segment of \( n \) elements takes \( n \) comparisons to some pivot
  - Partition creates length \( m \) and \( r \) segments (where \( m + r = n - 1 \))
  - \( T(n) = n + T(m) + T(r) \)

Worst Case Runtime of Quicksort

- When \( T(n) = n + T(n-1) + T(0) \)
- Hypothesis: \( T(n) = n^2 - n/2 \)
- Base Case: \( T(1) = (1^2 - 1)/2 = 0 \)
- Inductive Hypothesis: assume \( T(k) = (k^2 - k)/2 \)
  - \( T(k+1) = k + (k^2 - k)/2 + 0 = (k^2 + k)/2 \)
  - \( = ((k+1)^2 - (k+1))/2 \)
- Therefore, for all \( n \geq 1 \):
  - \( T(n) = (n^2 - n)/2 = O(n^2) \)

Worst Case Space of Quicksort

You can see that in the worst case, the depth of recursion is \( O(n) \). Since each recursive call involves creating a new stack frame, which takes space, in the worst case, Quicksort takes space \( O(n) \). That is not good!

To get around this, rewrite QuickSort so that it is iterative but it sorts the smaller of two segments recursively. It is easy to do. The implementation in the java class that is on the website shows this.