Recitation on analysis of algorithms

Formal definition of $O(n)$

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

Let $f(n)$ and $g(n)$ be two functions that tell how many statements two algorithms execute when running on input of size $n$.

$f(n)$ is $O(g(n))$ iff there is a positive constant $c$ and a real number $N$ such that:

$$f(n) \leq c \times g(n) \quad \text{for } n \geq N$$

Example:

Let $f(n) = n + 6$ and $g(n) = n$.

We show that $n+6$ is $O(n)$.

So choose $c = 2$ and $N = 6$.

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 $n$.

$f(n)$ is $O(g(n))$ iff there is a positive constant $c$ and a real number $N$ such that:

$$f(n) \leq c \times g(n) \quad \text{for } n \geq N$$

Example:

Let $f(n) = n + 6$ and $g(n) = n$.

We show 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 $O(n)$.

What’s the difference between executing 1,000,000 steps and 1,000,0006? It’s insignificant.

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 $3 \times n$ are $O(n)$
4. $n \times \log(n) + n$ is $n \times \log(n)$ (quadratic)
5. $n^2 + 2 \times n + 6$ is $O(n^2)$ (cubic)
6. $2^n + n5$ is $O(2^n)$ (exponential)

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 $3 \times n$ are $O(n)$
4. $n \times \log(n) + n$ is $n \times \log(n)$ (quadratic)
5. $n^2 + 2 \times n + 6$ is $O(n^2)$ (cubic)
6. $2^n + n5$ 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. $O(n)$.

Some Notes on $O()$

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

• 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: $O(n^2) + O(n) = O(n^2)$

Some Notes on $O()$

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

• 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: $O(n^2) + O(n) = O(n^2)$
/** Sort b[h..k]. */
public static void mS(Comparable[] b, int h, int k) {
if (h >= k) return;

int e = (h+k)/2;
merge(b, h, e, 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 mS 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);
}

Run, me of MergeSort
Recursion:
T(n) = 2 * T(n/2) + comparisons made in merge

/** 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

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);
}

Use T(n) for the number of array element comparisons that mS 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);
}

Runtime

public static void merge (Comparable b[], int h, int e, int k) {
Comparable[] c = copy(b, h, e);
int i = h;
int j = e+1;
int m = 0;

for (i = h; i != k+1; i++) {
if (j <= k && (m > e-h || b[j].compareTo(c[m]) <= 0)) {
b[i] = b[j]; j++;
}
else {
b[i] = c[m]; m++;
}
}

Simplify calculations: assume n is a power of 2.

Simplify: use the size of the array segment O(k-h) time

Thus: T(n) < 2 T(n/2) + n, with T(1) = 0

// We show how to do an analysis, assuming n is a power of 2
(just to simplify the calculations)

Use T(n) for number of array element comparisons to mergesort an array segment 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);
}

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

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

Proof \( \text{by recursion tree of } T(n) = n \lg n \)

\[
T(n) = 2^{\lg n} + n, \text{ for } n > 1, \text{ a power of } 2, \text{ and } T(1) = 0
\]

Proof by induction:

Base case: \( n = 1 \): \( P(1) \) is \( T(1) = 1 \) \( \lg 1 \)

\( T(1) = 0 \), by definition.

\( 1 = 2^0 \), so \( 1 \) \( \lg 1 = 0 \).

\( \lg n \) means \( \log_2 n \)

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

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

\[
\begin{align*}
\lg(2n) - 1 &= <\text{definition of } n> \\
\lg(2^{2^k}) - 1 &= <\text{arith}> \\
\lg(2^2) + \lg(2^k) - 1 &= <\text{property of } \lg> \\
1 + \lg(2^k) - 1 &= <\text{arith}> \\
\lg n &= <\text{arith, definition of } n>
\end{align*}
\]

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

\( \lg n = k \)

\( \lg n \) means \( \log_2 n \)

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” \( O(n \lg n) \) runtime
  - MergeSort always has \( O(n \lg n) \) 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.