Recitation 11

Analysis of Algorithms and Inductive Proofs
Review: Big O definition

\[ f(n) \text{ is } O(g(n)) \]

iff

There exists \( c > 0 \) and \( N > 0 \) such that:

\[ f(n) \leq c \cdot g(n) \text{ for } n \geq N \]
Example: \( n+6 \) is \( O(n) \)

\[
n + 6 \quad \text{---this is } f(n) \\
\leq \quad \text{<if } 6 \leq n, \text{ write as}> \\
\quad n + n \\
= \quad \langle \text{arith} \rangle \\
\quad 2n \\
\quad \langle \text{choose } c = 2 \rangle \\
\quad c \cdot n \quad \text{---this is } c \cdot g(n)
\]

So choose \( c = 2 \) and \( N = 6 \)

\( f(n) \) is \( O(g(n)) \): There exist \( c > 0, N > 0 \) such that:

\[
f(n) \leq c \cdot g(n) \quad \text{for } n \geq N
\]
Review: Big O

Is used to classify algorithms by how they respond to changes in input size $n$.

**Important vocabulary:**
- Constant time: $O(1)$
- Logarithmic time: $O(\log n)$
- Linear time: $O(n)$
- Quadratic time: $O(n^2)$
- Exponential time: $O(2^n)$
## Review: Big O

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

mS is mergeSort for readability
Runtime of merge sort

/** 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);
}

mS is mergeSort for readability

- 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 segment of size $n$
Runtime of merge sort

/** 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);
}

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);   \( T(e+1-h) \) comparisons = \( T(n/2) \)
    mS(b, e+1, k); \( T(k-e) \) comparisons = \( T(n/2) \)
    merge(b, h, e, k); How long does merge take?
}

Runtime of merge sort
Runtime of merge

**pseudocode for merge**

/** Pre: b[h..e] and b[e+1..k] are already sorted */

merge(Comparable[] b, int h, int e, int k)

Copy both segments

While both copies are non-empty

- Compare the first element of each segment
- Set the next element of b to the smaller value
- Remove the smaller element from its segment

One comparison, one add, one remove

k-h loops must empty one segment

Runtime is $O(k-h)$
/** 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);
    T(e+1-h) comparisons = T(n/2)
    mS(b, e+1, k);
    T(k-e) comparisons = T(n/2)
    merge(b, h, e, k);
    O(k-h) comparisons = O(n)
}

Recursive Case:
T(n) = 2T(n/2) + O(n)
We determined that
\[
T(1) = 0
\]
\[
T(n) = 2T(n/2) + n \quad \text{for } n > 1
\]

We will prove that
\[
T(n) = n \log_2 n \quad \text{(or } n \log n \text{ for short)}
\]
Recursion tree

Merge Sort

$\lg n$ levels * $n$ comparisons is $O(n \log n)$
Proof by induction

To prove $T(n) = n \lg n$,
we can assume true for smaller values of $n$ (like recursion)

\[
T(n) = 2T(n/2) + n \\
= 2(n/2)\lg(n/2) + n \\
= n(\lg n - \lg 2) + n \\
= n(\lg n - 1) + n \\
= n\lg n - n + n \\
= n\lg n
\]

Property of logarithms

$log_2 2 = 1$
Heap Sort
Heap Sort

Very simple idea:
1. Turn the array into a max-heap
2. Pull each element out

/** Sort b */
public static void heapSort(Comparable[] b) {
    heapify(b);
    for (int i = b.length - 1; i >= 0; i--) {
        b[i] = poll(b, i);
    }
}
/** Sort b */
public static void heapSort(Comparable[] b) {
    heapify(b);
    for (int i = b.length - 1; i >= 0; i--) {
        b[i] = poll(b, i);
    }
}

Why does it have to be a max-heap?
Heap Sort runtime

/** Sort b */
public static void heapSort(Comparable[] b) {
    heapify(b);
    for (int i = b.length - 1; i >= 0; i--) {
        b[i] = poll(b, i);
    }
}

Total runtime:
\( O(n \log n) + n*O(\log n) = O(n \log n) \)