Recitation 11

Analysis of Algorithms and Inductive Proofs

---

Review: Big O definition

\[ f(n) = O(g(n)) \]

if

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

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

---

Example: \( n+6 \) is \( O(n) \)

\[
\begin{align*}
& n + 6 \quad \text{---this is } f(n) \\
\leq & \quad \text{<if } 6 \leq n, \text{ write as}> \\
& n + n \\
= & \quad \text{<arith>} \\
& 2^n \\
\text{choose } c = 2 \\
= & \quad \text{c} \cdot n \\
\text{---this is } c \cdot g(n) \\
\text{So choose } c = 2 \text{ and } N = 6
\end{align*}
\]

---

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. \( n \cdot \log(n) + n \) is \( n \cdot \log(n) \)
5. \( n^2 + 2n + 6 \) is \( O(n^2) \) (quadratic)
6. \( n^3 + n^2 \) is \( O(n^3) \) (cubic)
7. \( 2^n + n^5 \) is \( O(2^n) \) (exponential)

---

Merge Sort

---
/** Sort \([h..k]\). */
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.

Use \(T(n)\) for the number of array element comparisons that \texttt{mergeSort} makes on an array of size \(n\).

\(T(0) = 0\)
\(T(1) = 0\)

\(T(e+1-h)\) comparisons = \(T(n/2)\)
\(T(k-e)\)   comparisons = \(T(n/2)\)

merge(b, h, e, k); How long does merge take?

Recursive Case:
\(T(n) = 2T(n/2) + O(n)\)

Runtime

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

Proof by induction

To prove \( T(n) = n \log_2 n \), we can assume true for smaller values of \( n \) (like recursion)

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

Property of logarithms

\[ \log_2 2 = 1 \]

Heap Sort

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

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

Heap Sort runtime:

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