1 The growth of functions

- Suppose two functions, \( s(n) = 80000 + n \) and \( t(n) = 4n^2 \), represent the performance (time spent) of two algorithms to index a set of \( n \) books.
- How do we determine which algorithm to use?
- Although \( s(n) \) is greater than \( t(n) \) for small values of \( n \), \( t(n) \) will quickly become greater – does that help us decide?
- The constant factors only truly depend on the individual machine the algorithms are run on – if we purchased a faster machine, we would find that the constant 80000 would likely decrease, but the behavior of the algorithm would still be proportional to \( n \).

2 Big-Oh notation

- Big-Oh notation allows us to compare a function to a set of functions that are bounded by some other function, allowing us to describe the performance of a particular algorithm.
- The notation omits information about constant factors, since those are not important to us when analyzing the performance of programs on large input sets.
- We say that a function \( s(n) \in O(f(n)) \) if \( |s(n)| \leq K|f(n)| \) for all \( n > n_0 \) where \( n_0 \) and \( K \) are constants.
- This allows us to bound the function \( s(n) \) by \( f(n) \), in a sense, saying that \( s(n) \) grows at most as fast as \( f(n) \).
- With the example above, we could say that \( s(n) \in O(n^2) \) and that \( t(n) \in O(n^2) \), however, we could also be more specific (provide a “tighter” bound) and say that \( s(n) \in O(n) \).
- Typically the function we use for comparison is a simple function such as \( n \) or \( n^2 \), simply because the constant factor does not matter when using Big-Oh notation.
- Note that \( O(f(n)) \) is the set of all functions for which the above condition is satisfied, hence the set notation.
- We often loosely say that Big-Oh represents an upper bound.

3 Big-Omega notation

- Big-Omega notation is the opposite of Big-Oh – it represents a lower bound.
- We say that a function \( s(n) \in \Omega(f(n)) \) if \( |s(n)| \geq K|f(n)| \) for all \( n > n_0 \) where \( n_0 \) and \( K \) are constants.
4 Big-Theta notation

- If a function is bounded both above and below, some sense of equality must be indicated.
- We have seen Big-Oh and Big-Omega, which provide upper and lower bounds, respectively. If \( s(n) \in \mathcal{O}(f(n)) \) and \( s(n) \in \Omega(f(n)) \) then we can say that \( s(n) \in \Theta(f(n)) \).
- We say that a function \( s(n) \in \Theta(f(n)) \) if \(|s(n)| = K|f(n)|\) for all \( n > n_0 \) where \( n_0 \) and \( K \) are constants.

5 Little-Oh and Little-Omega notation

- Similar to their Big counterparts, but provide strict bounds – \( n \in o(n^2) \) is true, but \( n \in o(n) \) is false.
- We say that a function \( s(n) \in o(f(n)) \) if \(|s(n)| < K|f(n)|\) for all \( n > n_0 \) where \( n_0 \) and \( K \) are constants.
- We say that a function \( s(n) \in \omega(f(n)) \) if \(|s(n)| > K|f(n)|\) for all \( n > n_0 \) where \( n_0 \) and \( K \) are constants.

6 Examples

- Constant time – \( \Theta(1) \) – array access, other atomic operations.
- Logarithmic – \( \Theta(lg(n)) \) – binary search, heap operations.
- Linear – \( \Theta(n) \) – linked-list access, tree traversal.
- Linear Logarithmic – \( \Theta(nlg(n)) \) – fast sorting.
- Quadratic – \( \Theta(n^2) \) – slow sorting.
- Cubic – \( \Theta(n^3) \) – matrix-multiplication.
- Exponential – \( \Theta(2^n) \) – travelling-salesperson problem.

7 Analyzing algorithms

Examples

In \( O(lg \ n) \), \( n \) is \( inp.length \):

```java
static int binSearch(int[] s, int v, int b, int e) {
    if (b == e)
        return -1;
    int m = (e-b)/2;
    if (v == s[m])
        return m;
    else if (v > s[m])
        return binSearch(s, v, m+1, e);
    return binSearch(s, v, b, m-1);
}
```

In \( O(n) \), \( n \) is \( inp.length \):

```java
...
static int[] copyArray(int[] inp) {
    int[] out = new int[inp.length];
    for (int i = 0; i < inp.length; i++)
        out[i] = inp[i];
    return out;
}

In O(n), n is length of stack:

static int maxPos(IntList stack, int limit) {
    int max = stack.data;
    int maxpos = 0;
    IntList cur = stack.next;
    for (int i = 1; i < limit; i++, cur = cur.next)
        if (cur.data > max) {
            max = cur.data;
            maxpos = i;
        }
    return maxpos;
}

In O(n lg(n)), n is number of elements:

static BinaryHeap buildHeap(Comparable[] inp) {
    BinaryHeap h = new BinaryHeap();
    for (int i = 0; i < inp.length; i++)
        h.insert(inp[i]);
}

In O(n), n is number of elements:

BinaryHeap(Comparable[] inp) {
    h = new Vector(Arrays.asList(inp));
    for (int i = h.size()/2-1; i >=0; i--)
        reHeapifyDown(i);
}

8 Question

Analyze flipSort

public void flipSort() {
    flipSort(size);
}

private void flipSort(int limit) {
    if (limit <= 1)
        return;
flip(maxPos(limit)+1); // O(size)
flip(limit);
flipSort(limit-1);
}

White: O(n)
Blue: O(lg n)
Yellow: O(n lg n)
Green: O(n^2)
Pink: O(n^3)

Green and Pink are both correct, since we know that maxPos and flip both take O(n) time, and that flipSort calls itself n times, so we are doing O(n) operations n times, yielding n^2. Since Big-Oh notation gives us an upper bound, O(n^2) and O(n^3) are both correct for flipSort.