CIS 575. Introduction to Algorithm Analysis
Material for January 29, 2024

Pitfalls In Algorithm Analysis

1 Can The Worst Case Always Happen?

For the examples given so far in this set of notes, worst case has been equal to best case, in that the running times (even of subloops) have depended only on the size of the input, and not at all on the particular input of the given size.

On the other hand, consider the program

```
for i ← 1 to n
   if A[i] = x
      for j ← i to n − 1
         A[j] ← A[j + 1]
```

which looks for x in a given array A that is assumed **strictly** increasing, and if x is found then it is removed from A and the subsequent elements are shifted to the left. Assuming we are interested in **worst case** running time, we may be tempted to reason as follows:

1. in the worst case (when we have to shift), the body of the outer loop takes time proportional to $n - i + 1$
2. hence, in the worst case, the total running time is given as the sum

$$\sum_{i=1}^{n}(n - i + 1) \text{ which equals } \sum_{i=1}^{n} i$$

   which by *Howell*'s Theorem 3.28 is in $\Theta(n \cdot n) = \Theta(n^2)$.

But that analysis has a major flaw: since x occurs at most once in A (as A is strictly increasing), only one iteration of the outer loop can have worst case behavior; all other iterations will execute in constant time! Instead, we must reason as follows:

1. the total time used to shift the array elements is in $O(n)$
2. apart from shifting the array elements, the running time is (even in the best case) in $\Theta(n)$

and we infer that the total running time (best case as well as worst case) is actually in $\mathbf{\Theta(n)}$.

Applying Thm 3.28 to Insertion Sort Recall our implementation of insertion sort:

```
for i ← 2 to n
    j ← i
    while j > 1 and A[j] < A[j − 1]
        A[j] ↔ A[j − 1]; j ← j − 1
```

To analyze (a bit more formally than the first week) its running time, we may reason as follows:

1. in the worst case, the body of the outer loop will take time in $\Theta(i)$
2. that worst case may happen *every* iteration (when the array is sorted in reverse order)
3. hence the worst case total running time is proportional to the sum

$$\sum_{i=1}^{n} i$$

4. by *Howell*'s Theorem 3.28 we see that the worst case total running time is in $\Theta(n \cdot n) = \mathbf{\Theta(n^2)}$.

2 Elementary Instructions

Recall that in the first week, we presented an iterative implementation of the fibonacci function:

```
i, j ← 1, 1
for k ← 1 to n − 1
    i, j ← j, i + j
return j
```

which surely runs faster than the recursive implementation which naively executed has exponential running time. And indeed, it may appear that the above algorithm runs in time $\Theta(n)$.

But let us examine what happens in the k'th iteration. Then we must add two numbers, and those two numbers will be increasingly large as k increases; it's not hard to see that it will require $\Theta(k)$ bits to represent i and j. Before long, this will exceed the number of bits (32 or 64) used for a standard representation of integers. When that happens, we may rely on "large integers" if implemented in our system, or write our own implementation of large integers, but we must expect the addition of such integers to take time proportional to the number of bits. Hence the running time of the k'th iteration will be not a constant, but in $\Theta(k)$; we now see (again using *Howell*'s Theorem 3.28) that the total running time is in $\Theta(n^2)$ (rather than in $\Theta(n)$.

Still, for most programs it is not unrealistic to assume that arithmetic operations, and assignments, can be executed in *constant* time. To avoid unnecessary complications, we shall make that assumption throughout this course (unless otherwise mentioned).