Today

- Analysis Techniques (cont)

Assignments

- Exercise 1 (due thur)
- Quiz 1 on thur
Example 2:

```cpp
bool duplicates(const int A[], int n)
{
    bool found = false;
    for (int i = 0; i < n; ++i)
        for (int j = 0; j < n; ++j)
            if (A[i] == A[j] and i != j)
                found = true;
    return found;
}
```

Assume primitives: assign/initialize, compare, increment, array access, logical and

Q: Is there a “best case” and “worst case”?

- Best case when array has unique values
- Worst case when all values in array are the same

Q: Give the best-case count as a function $T(n)$

\[ T(n) \geq 6n^2 + 4n + 3 \]

Q: Give the worst-case count as a function $T(n)$

\[ T(n) \leq 8n^2 + 3n + 3 \]

This is a (worst case) “quadratic time” algorithm (i.e., of the form $a \cdot n^2 + b \cdot n + c$)

\(*)\) For best and worst case we count the cost of “and”. Worst case, $i \neq j$ checked each iteration
Example 3:

```c
1:   bool duplicates(const int A[], int n)  
2:   { 
3:     for (int i = 0; i < n; ++i)  
4:       for (int j = i + 1; j < n; ++j)  
5:         if (A[i] == A[j])  
6:           return true;  
7:       return false;  
8:   }
```

Q: Is there a "best case" and "worst case"?

- Best case when \(A[0] == A[1]\)
- Worst case when no duplicates

Q: Only counting array comparisons (i.e., line 5), what is the worst-case as \(T(n)\)?

- Notice that:
  - \(i = 0\), inner loop performs \(n - 1\) comparisons
  - \(i = 1\), inner loop performs \(n - 2\) comparisons
  - \(\ldots\)
  - \(i = n-1\), inner loop performs \(n - n\) comparisons

- Giving (*):
  \[ T(n) \leq \sum_{i=0}^{n-1} i = \frac{(n-1)n}{2} = \frac{1}{2}n^2 - \frac{1}{2}n \]

This is a (worst case) "quadratic time" algorithm (i.e., \(a \cdot n^2 + b \cdot n + c\))

(*) Note: \(\sum_{i=1}^{n} i = \frac{n(n+1)}{2}\)
**Asymptotic Notation**

Two algorithms with same “growth rate” treated as same “complexity”
- e.g., two linear-time algorithms are of similar complexity
- even if they have different primitive operation count equations

Basic high-level idea:
1. suppress constant factors and lower order terms
   - too system-dependent
   - irrelevant for large inputs

Three main notations for comparing algorithm growth rates:
- “Big-O” \((O\text{-notation})\) ... “not slower than” (upper bound)
- “Big-\(\Omega\) (\(\Omega\text{-notation})\) ... “not faster than” (lower bound)
- “Big-\(\Theta\) (\(\Theta\text{-notation})\) ... “exactly” (tight bound)

**Primarily Big-O and Big-\(\Theta\) notation in this class**
- For example, saying an algorithm has a linear-time Big-O growth rate
- ... implies its worst case isn’t worse than linear time

**Note the three notations can be used in different contexts**
- We could say an algorithm’s average case has a linear-time Big-O growth rate
- ... implying its average case isn’t worse than linear-time

---

\(^1\)From: *Algorithms Illuminated*
Big-O Definition

\( T(n) \) is \( O(f(n)) \) iff there exist positive constants \( k \) and \( n_0 \) such that:

\[ T(n) \leq k \cdot f(n) \]

for all \( n \geq n_0 \)

Comments:

• showing \( T(n) \) is \( O(f(n)) \) involves finding a \( k \) and \( n_0 \)

• as an upper bound can be misleading ...

• e.g., \( T(n) \) being \( O(n) \) implies \( T(n) \) is also \( O(n^2) \)

Exercise: Show \( T(n) \leq 3n + 2 \) is \( O(n) \)

• note there are many possible \( k \) and \( n_0 \) values we could use ...

• \( 3n + 2 \leq 5n \) for \( n \geq 1 \) ... thus \( k = 5 \) and \( n_0 = 1 \)

• Which implies \( T(n) \) is \( O(n) \)
Big-$\Omega$

$T(n)$ is $\Omega(f(n))$ iff there exist positive constants $k$ and $n_0$ such that:

$$T(n) \geq k \cdot f(n)$$

for all $n \geq n_0$

Comments:

• showing $T(n)$ is $\Omega(f(n))$ involves finding a $k$ and $n_0$
• as a lower bound can be misleading ...
• e.g., $T(n)$ being $\Omega(n^2)$ implies $T(n)$ is also $\Omega(n)$

Exercise: Show $T(n) \geq 4n + 3$ is $\Omega(n)$

• $4n + 3 \geq 4n$ for $n \geq 1$ ... thus $k = 4$ and $n_0 = 1$
• Which implies $T(n)$ is $\Omega(n)$
Big-Θ

\[ T(n) \text{ is } \Theta(f(n)) \text{ iff there exist positive constants } k_1, k_2, \text{ and } n_0 \text{ s.t.:} \]
\[ k_1 \cdot f(n) \leq T(n) \leq k_2 \cdot f(n) \]

for all \( n \geq n_0 \)

Comments:

• showing \( T(n) \text{ is } \Theta(f(n)) \) often by showing \( \Omega(f(n)) \) and \( O(f(n)) \) for an \( n_0 \)

Example:

• Let \( 4n + 3 \leq T(n) \leq 5n + 3 \) and \( f(n) = n \)
• \( T(n) \text{ is } \Omega(n) \text{ and } O(n) \) for \( n_0 = 1 \)
• Which implies \( T(n) \text{ is } \Theta(n) \)
Some “Standard” Algorithm Growth Rates

\[ O(n) \] and even \( O(n \log n) \) algorithms are often considered “for free primitives”

- since they basically only require “reading all the input”
Properties of Big-O Notation

**Sum of two functions**

\[ O(f(n)) + O(g(n)) = O(\max(f(n), g(n))) \]

- e.g., duplicate check \( O(n^2) \) followed by membership check \( O(n) \) is \( O(n^2) \)

**Product of two functions**

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

- e.g., doing \( O(n) \) work \( O(\log n) \) times is \( O(n \log n) \)
- usually requires more careful analysis to ensure really a product (more later)
**Things to watch-out for with Asymptotic Analysis**

**Algorithm versus Problem**
- Bubble sort is an $O(n^2)$ (comparison) sorting algorithm
- But the problem of (comparison) sorting is a worst-case $O(n \log n)$ problem
- Thus comparison sorting can’t be achieved with a lower worst-case cost

**Tight Upper and Lower Bounds**
- An algorithm that is $O(n)$ is also $O(n^2)$
- Similarly an algorithm that is $\Omega(n)$ is also $\Omega(\log n)$
- However, we usually mean that the given bound is “tight”
- Note that algorithms that are $O(1)$ are also $\Theta(1)$ and $\Omega(1)$ ... why?

**We’ll see more analysis techniques as we go ...**