Announcements

- Reading
  - Chapter 2.1, 2.2
  - Chapter 3 (Mostly review)
  - Start on Chapter 4
- Homework Guidelines
  - Submit homework with Gradescope
  - Describing an algorithm
    - Clarity is most important
    - Pseudocode generally preferable to just English
    - But sometimes both methods combined work best
  - Prove that your algorithm works
    - A proof is a "convincing argument"
    - Give the run time for your algorithm
    - Justify that the algorithm satisfies the runtime bound
  - You may lose points for style
  - Homework assignments will (probably) be worth the same amount

Summary – Five Problems

- Scheduling
- Weighted Scheduling
- Combinatorial Optimization
- Maximum Independent Set
- Competitive Scheduling

What does it mean for an algorithm to be efficient?

Definitions of efficiency

- Fast in practice
- Qualitatively better worst case performance than a brute force algorithm
Polynomial time efficiency

- An algorithm is efficient if it has a polynomial run time
- Run time as a function of problem size
  - Run time: count number of instructions executed on an underlying model of computation
  - $T(n)$: maximum run time for all problems of size at most $n$

Polynomial Time

- Algorithms with polynomial run time have the property that increasing the problem size by a constant factor increases the run time by at most a constant factor (depending on the algorithm)

Why Polynomial Time?

- Generally, polynomial time seems to capture the algorithms which are efficient in practice
- The class of polynomial time algorithms has many good, mathematical properties

Polynomial vs. Exponential Complexity

- Suppose you have an algorithm which takes $n!$ steps on a problem of size $n$
- If the algorithm takes one second for a problem of size 10, estimate the run time for the following problem sizes:

<table>
<thead>
<tr>
<th>Size (n)</th>
<th>12</th>
<th>14</th>
<th>16</th>
<th>18</th>
<th>20</th>
</tr>
</thead>
</table>

Ignoring constant factors

- Express run time as $O(f(n))$
- Emphasize algorithms with slower growth rates
- Fundamental idea in the study of algorithms
- Basis of Tarjan/Hopcroft Turing Award

Why ignore constant factors?

- Constant factors are arbitrary
  - Depend on the implementation
  - Depend on the details of the model
- Determining the constant factors is tedious and provides little insight
Why emphasize growth rates?

• The algorithm with the lower growth rate will be faster for all but a finite number of cases
• Performance is most important for larger problem size
• As memory prices continue to fall, bigger problem sizes become feasible
• Improving growth rate often requires new techniques

Formalizing growth rates

• \( T(n) = O(f(n)) \) \([T : \mathbb{Z}^+ \rightarrow \mathbb{R}^+]\)
  – If \( n \) is sufficiently large, \( T(n) \) is bounded by a constant multiple of \( f(n) \)
  – Exist \( c, n_0 \), such that for \( n > n_0 \), \( T(n) < c f(n) \)

• \( T(n) = O(f(n)) \) will be written as:
  \( T(n) = O(f(n)) \)
  – Be careful with this notation

Prove \( 3n^2 + 5n + 20 \) is \( O(n^2) \)

Let \( c = \)
Let \( n_0 = \)

\( T(n) = O(f(n)) \) if there exist \( c, n_0 \), such that for \( n > n_0 \),
\( T(n) < c f(n) \)

Order the following functions in increasing order by their growth rate

a) \( n \log^4 n \)
b) \( 2n^2 + 10n \)
c) \( 2^n/100 \)
d) \( 1000n + \log^8 n \)
e) \( n^{100} \)
f) \( 3^n \)
g) \( 1000 \log^{10} n \)
h) \( n^{1/2} \)

Lower bounds

• \( T(n) = \Omega(f(n)) \)
  – \( T(n) \) is at least a constant multiple of \( f(n) \)
  – There exists an \( n_0 \), and \( \varepsilon > 0 \) such that
    \( T(n) \geq \varepsilon f(n) \) for all \( n > n_0 \)
• Warning: definitions of \( \Omega \) vary
• \( T(n) = \Theta(f(n)) \) if \( T(n) = O(f(n)) \) and \( T(n) = \Omega(f(n)) \)

Useful Theorems

• If \( \lim_{n \to \infty} \frac{f(n)}{g(n)} = c \) for \( c > 0 \) then
  \( f(n) = \Theta(g(n)) \)

• If \( f(n) = O(g(n)) \) and \( g(n) = O(h(n)) \) then \( f(n) = O(h(n)) \)

• If \( f(n) = O(h(n)) \) and \( g(n) = O(h(n)) \) then
  \( f(n) + g(n) = O(h(n)) \)
Ordering growth rates

• For $b > 1$ and $x > 0$
  – $\log^b n$ is $O(n^x)$

• For $r > 1$ and $d > 0$
  – $n^d$ is $O(r^n)$