Announcements

- Reading
  - Chapter 2.1, 2.2
  - Chapter 3 (Mostly review)
  - Start on Chapter 4
- Homework Guidelines
  - 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

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</th>
<th>Run Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>12</td>
<td></td>
</tr>
<tr>
<td>14</td>
<td></td>
</tr>
<tr>
<td>16</td>
<td></td>
</tr>
<tr>
<td>18</td>
<td></td>
</tr>
<tr>
<td>20</td>
<td></td>
</tr>
</tbody>
</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
- $T(n) = O(f(n))$ means $T : Z^+ \rightarrow R^+$.
  - 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

<table>
<thead>
<tr>
<th>Function</th>
<th>Growth Rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>$n \log^4 n$</td>
<td>a)</td>
</tr>
<tr>
<td>$2n^2 + 10n$</td>
<td>b)</td>
</tr>
<tr>
<td>$2^n/100$</td>
<td>c)</td>
</tr>
<tr>
<td>$1000n + \log^8 n$</td>
<td>d)</td>
</tr>
<tr>
<td>$n^{100}$</td>
<td>e)</td>
</tr>
<tr>
<td>$3^n$</td>
<td>f)</td>
</tr>
<tr>
<td>$1000 \log^{10} n$</td>
<td>g)</td>
</tr>
<tr>
<td>$n^{1/2}$</td>
<td>h)</td>
</tr>
</tbody>
</table>

Lower bounds

- $T(n)$ is $\Omega(f(n))$
  - $T(n)$ is at least a constant multiple of $f(n)$
  - There exists an $n_0$, and $\epsilon > 0$ such that $T(n) > \epsilon f(n)$ for all $n > n_0$
- Warning: definitions of $\Omega$ vary

- $T(n)$ is $\Theta(f(n))$ if $T(n)$ is $O(f(n))$ and $T(n)$ is $\Omega(f(n))$

Useful Theorems

- If $\lim (f(n) / g(n)) = c$ for $c > 0$ then $f(n) = \Theta(g(n))$
- If $f(n)$ is $O(g(n))$ and $g(n)$ is $O(h(n))$ then $f(n)$ is $O(h(n))$
- If $f(n)$ is $O(h(n))$ and $g(n)$ is $O(h(n))$ then $f(n) + g(n)$ is $O(h(n))$

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 $O(r^n)$

Graph Theory

- $G = (V, E)$
  - $V$ – vertices
  - $E$ – edges
- Undirected graphs
  - Edges sets of two vertices $(u, v)$
- Directed graphs
  - Edges ordered pairs $(u, v)$
- Many other flavors
  - Edge / vertices weights
  - Parallel edges
  - Self loops
Definitions

- Path: \( v_1, v_2, \ldots, v_k \), with \((v_i, v_{i+1})\) in \( E \)
  - Simple Path
  - Cycle
  - Simple Cycle
- Distance
- Connectivity
  - Undirected
  - Directed (strong connectivity)
- Trees
  - Rooted
  - Unrooted

Graph search

- Find a path from \( s \) to \( t \)
  
  \[ S = \{ s \} \]
  
  While there exists \((u, v)\) in \( E \) with \( u \) in \( S \) and \( v \) not in \( S \)

  \[ \text{Pred}(v) = u \]
  
  Add \( v \) to \( S \)
  
  if \( v = t \) then path found

Breadth first search

- Explore vertices in layers
  - \( s \) in layer 1
  - Neighbors of \( s \) in layer 2
  - Neighbors of layer 2 in layer 3 . . .

Key observation

- All edges go between vertices on the same layer or adjacent layers

Bipartite Graphs

- A graph \( V \) is bipartite if \( V \) can be partitioned into \( V_1, V_2 \) such that all edges go between \( V_1 \) and \( V_2 \)
- A graph is bipartite if it can be two colored

Can this graph be two colored?
Algorithm

• Run BFS
• Color odd layers red, even layers blue
• If no edges between the same layer, the graph is bipartite
• If edge between two vertices of the same layer, then there is an odd cycle, and the graph is not bipartite

Theorem: A graph is bipartite if and only if it has no odd cycles

Lemma 1

• If a graph contains an odd cycle, it is not bipartite

Lemma 2

• If a BFS tree has an intra-level edge, then the graph has an odd length cycle

Intra-level edge: both end points are in the same level

Lemma 3

• If a graph has no odd length cycles, then it is bipartite