Announcements

• Reading
  – Chapter 2.1, 2.2
  – Chapter 3
  – Chapter 4

• Homework Guidelines
  – Prove that your algorithm works
    • A proof is a “convincing argument”
  – Give the run time for you algorithm
    • Justify that the algorithm satisfies the runtime bound
  – You may lose points for style
Announcements

- Monday, January 21 is a holiday
  - No class
- Makeup lecture, Thursday, January 17, 5:00 pm – 6:30 pm
  - UW and Microsoft
  - View off line if you cannot attend
- Homework 2 is due January 21
  - Electronic turn in only
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 problems:

<table>
<thead>
<tr>
<th>Size</th>
<th>Run Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>12</td>
<td>2 minutes</td>
</tr>
<tr>
<td>14</td>
<td>6 hours</td>
</tr>
<tr>
<td>16</td>
<td>2 months</td>
</tr>
<tr>
<td>18</td>
<td>50 years</td>
</tr>
<tr>
<td>20</td>
<td>20,000 years</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

• \( T(n) = O(f(n)) \) \([T : \mathbb{Z}^+ \rightarrow \mathbb{R}^+]\)
  - 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 $c = 6$

Let $n_0 = 5$

$$3n^2 + 5n + 20 < 6n^2$$ for $n > 5$

Assume $n > 5$

$$3n^2 + 5n + 20 \leq 3n^2 + n^2 + n^2$$

$$= 5n^2 \leq 6n^2$$

$T(n)$ is $O(f(n))$ if there exist $c$, $n_0$, such that for $n > n_0$,

$T(n) < c \cdot f(n)$
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

- $T(n)$ is $\Omega(f(n))$
  - $T(n)$ is at least a constant multiple of $f(n)$
  - There exists an $n_0$, and $\varepsilon > 0$ such that $T(n) > \varepsilon f(n)$ 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 $O(r^n)$
Stable Matching
# Reported Results

<table>
<thead>
<tr>
<th>Student</th>
<th>n</th>
<th>M/n</th>
<th>W/n</th>
<th>M/n*W/n</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stanislav</td>
<td>10,000</td>
<td>9.96</td>
<td>1020</td>
<td>10159</td>
</tr>
<tr>
<td>Andy</td>
<td>4,096</td>
<td>8.77</td>
<td>472</td>
<td>4139</td>
</tr>
<tr>
<td>Boris</td>
<td>5,000</td>
<td>10.06</td>
<td>499</td>
<td>5020</td>
</tr>
<tr>
<td>Huy</td>
<td>10,000</td>
<td>10.68</td>
<td>969</td>
<td>10349</td>
</tr>
<tr>
<td>Hans</td>
<td>10,000</td>
<td>9.59</td>
<td>1046</td>
<td>10031</td>
</tr>
<tr>
<td>Vijayanand</td>
<td>1,000</td>
<td>8.60</td>
<td>114</td>
<td>980</td>
</tr>
<tr>
<td>Robert</td>
<td>20,000</td>
<td>12.40</td>
<td>1698</td>
<td>21055</td>
</tr>
<tr>
<td>Zain</td>
<td>2,825</td>
<td>8.61</td>
<td>331</td>
<td>2850</td>
</tr>
<tr>
<td>Uzair</td>
<td>8,192</td>
<td>9.10</td>
<td>883</td>
<td>8035</td>
</tr>
<tr>
<td>Anand</td>
<td>10,000</td>
<td>9.58</td>
<td>1045</td>
<td>10011</td>
</tr>
</tbody>
</table>
Why is $M/n \sim \log n$?

While unmatched $m$, choose $m$ - picks to.

A Coupon Collector Problem

Average number of coupons needed is $n \log n$.
Why is $W/n \sim n / \log n$?

M's propose, $n \log n$ times.

Each $W$ receives $\log n$ proposals.
Graph Theory
Graph Theory

- \( G = (V, E) \)
  - \( V \) – vertices \( |V| = n \)
  - \( E \) – edges \( |E| = m \)

- 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: $v_1, v_2, ..., v_k$, 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.

\[
\text{Graph has an odd cycle} \\
\implies \text{Graph is not bipartite}
\]

\[
\text{Graph has no odd cycle} \\
\implies \text{Graph is bipartite}
\]
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
Connected Components

- Undirected Graphs
Computing Connected Components in $O(n+m)$ time

- A search algorithm from a vertex $v$ can find all vertices in $v$’s component
- While there is an unvisited vertex $v$, search from $v$ to find a new component
Directed Graphs

- A Strongly Connected Component is a subset of the vertices with paths between every pair of vertices.
Identify the Strongly Connected Components
Strongly connected components can be found in O(n+m) time

- But it's tricky!
- Simpler problem: given a vertex v, compute the vertices in v's scc in O(n+m) time

\[ S_1 \text{ All vertex } v \text{ can reach} \]
\[ S_2 \text{ All vertices that can reach } v \]
\[ S_1 \cup S_2 \]
Topological Sort

- Given a set of tasks with precedence constraints, find a linear order of the tasks.
Find a topological order for the following graph

A, H, I, E, C, B
If a graph has a cycle, there is no topological sort

- Consider the first vertex on the cycle in the topological sort
- It must have an incoming edge
Lemma: If a graph is acyclic, it has a vertex with in degree 0

• Proof:
  – Pick a vertex $v_1$, if it has in-degree 0 then done
  – If not, let $(v_2, v_1)$ be an edge, if $v_2$ has in-degree 0 then done
  – If not, let $(v_3, v_2)$ be an edge . . .
  – If this process continues for more than n steps, we have a repeated vertex, so we have a cycle
Topological Sort Algorithm

While there exists a vertex $v$ with in-degree 0

Output vertex $v$

Delete the vertex $v$ and all out going edges
Details for $O(n+m)$ implementation

- Maintain a list of vertices of in-degree 0
- Each vertex keeps track of its in-degree
- Update in-degrees and list when edges are removed
- $m$ edge removals at $O(1)$ cost each
Random Graph models

Edge will exist with probability $p$. 
Random out degree one graph

$n = 100,000,000$

Question:
What is the cycle structure as $N$ gets large?
How many cycles?
What is the cycle length?
Greedy Algorithms