Big-O and Sorting

April 2–3, 2012
CS 60: Principles of Computer Science

Assignment 8 (Full Unicalc) due Monday, April 2.
**Recall:** $O(\cdots)$ notation

Upper bound

Ignores constant factors

Assumes input gets arbitrarily big

In theory, $O()$ tells us very little.

In practice, the hidden constants are often small, and so we can draw reasonable conclusions about scalability.
# Running Times
(or why people worry about algorithm complexity)

<table>
<thead>
<tr>
<th>problem size complexity</th>
<th>n = 10</th>
<th>100</th>
<th>1,000</th>
<th>10,000</th>
<th>100,000</th>
<th>1,000,000</th>
</tr>
</thead>
<tbody>
<tr>
<td>log^2 n</td>
<td>10.361</td>
<td>44.140</td>
<td>99.317</td>
<td>176.54</td>
<td>275.85</td>
<td>397.24</td>
</tr>
<tr>
<td>sqrt n</td>
<td>3.162</td>
<td>10</td>
<td>31.622</td>
<td>100</td>
<td>316.22</td>
<td>1000</td>
</tr>
<tr>
<td>n</td>
<td>10</td>
<td>100</td>
<td>1000</td>
<td>10000</td>
<td>100000</td>
<td>1000000</td>
</tr>
<tr>
<td>n log n</td>
<td>33.219</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>n^1.5</td>
<td>31.6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>n^2</td>
<td>100</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>n^3</td>
<td>1000</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2^n</td>
<td>1024</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

imagine the time units ~ nanoseconds: \(10^{**(-9)}\)
### Running Times
(or why people worry about algorithm complexity)

<table>
<thead>
<tr>
<th>Problem size complexity</th>
<th>n = 10</th>
<th>100</th>
<th>1,000</th>
<th>10,000</th>
<th>100,000</th>
<th>1,000,000</th>
</tr>
</thead>
<tbody>
<tr>
<td>log² n</td>
<td>10.361</td>
<td>44.140</td>
<td>99.317</td>
<td>176.54</td>
<td>275.85</td>
<td>397.24</td>
</tr>
<tr>
<td>sqrt n</td>
<td>3.162</td>
<td>10</td>
<td>31.622</td>
<td>100</td>
<td>316.22</td>
<td>1000</td>
</tr>
<tr>
<td>n</td>
<td>10</td>
<td>100</td>
<td>1000</td>
<td>10000</td>
<td>100000</td>
<td>1000000</td>
</tr>
<tr>
<td>n log n</td>
<td>33.219</td>
<td>664.38</td>
<td>9965.8</td>
<td>132877</td>
<td>1.66*10⁶</td>
<td>1.99*10⁷</td>
</tr>
<tr>
<td>n¹.⁵</td>
<td>31.6</td>
<td>10³</td>
<td>31.6*10⁴</td>
<td>10⁶</td>
<td>31.6*10⁷</td>
<td>10⁹</td>
</tr>
<tr>
<td>n²</td>
<td>100</td>
<td>10⁴</td>
<td>10⁶</td>
<td>10⁸</td>
<td>10¹⁰</td>
<td>10¹²</td>
</tr>
<tr>
<td>n³</td>
<td>1000</td>
<td>10⁶</td>
<td>10⁹</td>
<td>10¹²</td>
<td>10¹⁵</td>
<td>10¹⁸</td>
</tr>
<tr>
<td>2ⁿ</td>
<td>1024</td>
<td>10³⁰</td>
<td>10³⁰¹</td>
<td>10³⁰¹⁰</td>
<td>10³⁰¹⁰³</td>
<td>10³⁰¹⁰³⁰</td>
</tr>
</tbody>
</table>

factorial? don't even ask!

imagine the time units ~ nanoseconds: \(10^{**(-9)}\)
Common Series in O()-Land

How many terms are here? What’s the O() sum?

\[ 1 + 2 + 4 + 8 + \cdots + 2^{n-2} + 2^{n-1} + 2^n \]

How many terms are here? What’s the O() sum?

\[ 1 + 2 + 4 + 8 + \cdots + \frac{n}{4} + \frac{n}{2} + n \]

How many terms are here? What’s the O() sum?

\[ 1 + 2 + 3 + 4 + \cdots + (n-2) + (n-1) + n \]
**Common Series in O()-Land**

How many terms are here? What's the O() sum?

\[ 1 + 2 + 4 + 8 + \cdots + 2^{n-2} + 2^{n-1} + 2^n = 2^{n+1} - 1 = O(2^n) \]

How many terms are here? What's the O() sum?

\[ 1 + 2 + 4 + 8 + \cdots + n/4 + n/2 + n \]

How many terms are here? What's the O() sum?

\[ 1 + 2 + 3 + 4 + \cdots + (n-2) + (n-1) + n \]
Common Series in $O()$-Land

How many terms are here? What’s the $O()$ sum?

$$1 + 2 + 4 + 8 + \cdots + 2^{n-2} + 2^{n-1} + 2^n = 2^{n+1} - 1 = O(2^n)$$

How many terms are here? What’s the $O()$ sum?

$$1 + 2 + 4 + 8 + \cdots + \frac{n}{4} + \frac{n}{2} + n = 2n - 1 = O(n)$$

How many terms are here? What’s the $O()$ sum?

$$1 + 2 + 3 + 4 + \cdots + (n-2) + (n-1) + n$$
Common Series in $O()$-Land

How many terms are here? What’s the $O()$ sum?

$$1 + 2 + 4 + 8 + \cdots + 2^{n-2} + 2^{n-1} + 2^n = 2^{n+1} - 1 = O(2^n)$$

How many terms are here? What’s the $O()$ sum?

$$1 + 2 + 4 + 8 + \cdots + \frac{n}{4} + \frac{n}{2} + n = 2n - 1 = O(n)$$

How many terms are here? What’s the $O()$ sum?

$$1 + 2 + 3 + 4 + \cdots + (n-2) + (n-1) + n = \frac{n(n+1)}{2} = O(n^2)$$
Loops and $O()$

✓ What are the possible values of the loop index?
✓ For each, how much work is being done inside?
✓ What is the $O()$ running time of each loop?

---

for (int i = 0; i < N; ++i)
  ...1 step of work here...

---

for (int i = N; i > 0; i /= 2)
  ...1 step of work here...
**Loops and \( O() \)**

✓ What are the possible values of the outer loop index?

✓ For each, how much work is being done inside?

✓ What is the \( O() \) running time of each loop?

```c
for ( int i=0; i<N; ++i )
    for ( int j=0; j<N; ++j )
        ...1 time step of work here...
```

```c
for ( int i=1; i<=N; ++i )
    for ( int j=1; j<=i; ++j )
        ...1 time step of work here...
```

```c
for ( int i=1; i<=N; i*=2 )
    for ( int j=0; j<i; j++ )
        ...1 time step of work here ...
```
Compute the asymptotic running time of:

```
for (int i=1; i<=N; i*=2 )
    for (int j=1; j<i; j*=2 )
        ... O(1) work here ...
```
**Sorting Algs: The “Fruit flies” of Complexity**

In the worst-case, how many *comparisons* might the following sorting algorithms perform, given an array of $n$ inputs?

A  **minsort**  repeatedly find the minimum; assemble sorted list

B  **mergesort**  mergesort each half; merge sorted halves.

C  **cs5sort**

```python
def cs5sort(L):
    if len(L) < 2:
        return L
    if L[0] == min(L):
        return [L[0]] + cs5sort(L[1:])
    return cs5sort(L[1:] + L[0])
```
**Sorting Algs: The “Fruit flies” of Complexity**

In the worst-case, how many *comparisons* might the following sorting algorithms perform, given an array of *n* inputs?

A  **minsort**  repeatedly find the minimum; assemble sorted list  \( O(n^2) \)

B  **mergesort**  mergesort each half; merge sorted halves.

C  **cs5sort**

```python
def cs5sort(L):
    if len(L) < 2:
        return L
    if L[0] == min(L):
        return [L[0]] + cs5sort(L[1:])
    return cs5sort(L[1:] + L[0])
```
**Sorting Algs: The “Fruit flies” of Complexity**

In the worst-case, how many *comparisons* might the following sorting algorithms perform, given an array of *n* inputs?

A. **minsort**  
   repeatedly find the minimum; assemble sorted list  
   \[ O(n^2) \]

B. **mergesort**  
   mergesort each half; merge sorted halves  
   \[ O(n \log n) \]

C. **cs5sort**

```python
def cs5sort(L):
    if len(L) < 2:
        return L
    if L[0] == min(L):
        return [L[0]] + cs5sort(L[1:])
    return cs5sort(L[1:] + L[0])
```
**Sorting Algs: The “Fruit flies” of Complexity**

In the worst-case, how many *comparisons* might the following sorting algorithms perform, given an array of *n* inputs?

A. *minsort*: repeatedly find the minimum; assemble sorted list  \( \mathcal{O}(n^2) \)

B. *mergesort*: mergesort each half; merge sorted halves  \( \mathcal{O}(n \log n) \)

C. *cs5sort*

```python
def cs5sort(L):
    if len(L) < 2:
        return L
    if L[0] == min(L):
        return [L[0]] + cs5sort(L[1:])
    return cs5sort(L[1:] + L[0])
```

(n^3)
**Sorting Algs: The “fruit flies” of Complexity**

In the worst-case, how many *comparisons* might the following sorting algorithms perform, given an array of $n$ inputs?

- **D prologsort**: try all permutations in order; check each for being sorted.
- **E bogosort**: repeatedly pick random permutations; check each for being sorted.
- **F stoogesort**: recurse on first 2/3; recurse on last 2/3; recurse on first 2/3.
**Sorting Algs: The “fruit flies” of Complexity**

In the worst-case, how many *comparisons* might the following sorting algorithms perform, given an array of *n* inputs?

- **D prologsort** try all permutations in order; check each for being sorted. \( \mathcal{O}(n!) \)

- **E bogosort** repeatedly pick random permutations; check each for being sorted.

- **F stoogesort** recurse on first 2/3; recurse on last 2/3; recurse on first 2/3.
## Sorting Algs: The “fruit flies” of Complexity

In the worst-case, how many *comparisons* might the following sorting algorithms perform, given an array of \( n \) inputs?

<table>
<thead>
<tr>
<th></th>
<th>Algorithm</th>
<th>Description</th>
<th>Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>D</td>
<td>prologsort</td>
<td>try all permutations in order; check each for being sorted.</td>
<td>( O(n!) )</td>
</tr>
<tr>
<td>E</td>
<td>bogosort</td>
<td>repeatedly pick random permutations; check each for being sorted.</td>
<td>( O(n!) )</td>
</tr>
<tr>
<td>F</td>
<td>stoogesort</td>
<td>recurse on first 2/3; recurse on last 2/3; recurse on first 2/3.</td>
<td>( O(n^{2.7}) )</td>
</tr>
</tbody>
</table>
**Sorting Algs: The “fruit flies” of Complexity**

In the worst-case, how many *comparisons* might the following sorting algorithms perform, given an array of $n$ inputs?

- **D prologsort**  
  try all permutations in order; check each for being sorted. $O(n!)$

- **E bogosort**  
  repeatedly pick random permutations; check each for being sorted. $O(n!)$

- **F stoogesort**  
  recurse on first $2/3$; recurse on last $2/3$; recurse on first $2/3$. $\approx O(n^{2.7})$
Insertion Sort

4 7 5 2 1 3 0 6

Idea:
1. Maintain a "sorted part" at the start.
2. Repeatedly extend this sorted part by moving one more number to its proper location.

Worst-case running time? Best-case running time?