Sorting Concluded; Dynamic Programming

April 4–5, 2012
CS 60: Principles of Computer Science

Assignment 9 (Big-O) due Monday, April 9.
## Sorting Algorithms so far

Given an array of $n$ values:

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Complexity</th>
<th>Case</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mergesort</td>
<td>$O(n \log n)$</td>
<td>worst-case</td>
</tr>
<tr>
<td>Insertion Sort</td>
<td>$O(n^2)$</td>
<td>worst-case</td>
</tr>
<tr>
<td></td>
<td>$O(n)$</td>
<td>best-case</td>
</tr>
<tr>
<td>Minsort/Selection Sort</td>
<td>$O(n^2)$</td>
<td>worst-case</td>
</tr>
<tr>
<td>Stoogesort</td>
<td>$O(n^{2.7095})$</td>
<td>worst-case</td>
</tr>
<tr>
<td>cs5sort</td>
<td>$O(n^3)$</td>
<td>worst-case</td>
</tr>
<tr>
<td>Prologsort</td>
<td>$O(n!)$</td>
<td>worst-case</td>
</tr>
<tr>
<td>Bogosort</td>
<td>$O(n!)$</td>
<td>expected</td>
</tr>
</tbody>
</table>
Quicksort

Invented by Tony Hoare. (He also invented null, and has apologized for it).

1. Pick a “pivot” element \( p \).
2. Partition input into two parts \( (< p \text{ and } \geq p) \)
3. Recursively quicksort the two parts.

3 4 6 2 0 1 3 7
**Analysis**

1. What is the recurrence for Quicksort in the *best* case? Solution?

\[
T(1) = 1 \\
T(n) = n + 2T(n/2) \quad (n > 1)
\]

\[
T(n) = O(n \log n)
\]

2. What is the recurrence in the *worst* case? Solution?

\[
T(1) = 1 \\
T(n) = n + T(1) + T(n - 1) \quad (n > 1)
\]

\[
T(n) = O(n^2)
\]
**Common Sorting Algorithms**

Given an array of $n$ values:

- **Mergesort** $O(n \log n)$ worst-case
- **Quicksort**
  - $O(n \log n)$ best-case
  - $O(n^2)$ worst-case
  - $O(n \log n)$ expected case (randomly-chosen pivot)
- **Insertion Sort**
  - $O(n^2)$ worst-case
  - $O(n)$ best-case

*CS 70 discusses Heapsort, which is $O(n \log n)$ worst-case.*
Which $O(n \log n)$ Algorithm is Best?

It depends. Here are some measurements from 2004:

**Measurements**

Now given all this information its a bit hard to see which of these should really be fastest in real life. That leaves us with one final possibility -- actually try it out in practice. Below are the results from just such a test:

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Athlon XP 1.620Ghz</th>
<th>Power4 1Ghz</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Intel C/C++ /O2 /G6 /Qaxi /Qxi /Qip</td>
<td>WATCOM C/C++ /ctexan /6r</td>
</tr>
<tr>
<td>Heapsort</td>
<td>2.09</td>
<td>4.06</td>
</tr>
<tr>
<td>Quicksort</td>
<td>2.58</td>
<td>3.24</td>
</tr>
<tr>
<td>Mergesort</td>
<td>3.51</td>
<td>4.28</td>
</tr>
</tbody>
</table>

Data is time in seconds taken to sort 10000 lists of varying size of about 3000 integers each. [Download test here](http://www.azillionmonkeys.com/qed/sort.html)
Can we do better?

Theorem

No algorithm based on comparisons can sort \( n \) items in better than \( O(n \log n) \) worst-case.

Key insights:

✓ Given \( n \) distinct values, there are \( n! \) possible permutations that might be our input.

✓ Sorting is equivalent to determining which permutation we started with.
Determining Permutations

Each comparison lets us keep some possible permutations and rule out others.
If we’re unlucky (or the input is chosen evilly), each comparison will rule out no more than half of the possibilities.

How many comparisons, to eliminate all but one possibility?
Stirling’s Formula

\[ \ln(n!) = n \ln n - n + O(\log n) \]

Hence the number of comparisons required is \( O(n \log n) \).
Can we do better?

Yes... if you make extra assumptions about the data.

**Bucket Sort:** Suppose we know the inputs are all in the range 1..k.

1. Allocate \( k \) buckets;
2. Put each item in the appropriate bucket;
3. Scan through the buckets in order.

Running time: \( \mathcal{O}(n + k) \) or \( \mathcal{O}(n) \) if \( k \) is constant
Radix Sort

Suppose the inputs are \( n \) sequences with \( k \) items each:

✓ Bucket-sort based the last item
✓ Bucket-sort on the next-to-last item
✓ ...
✓ Bucket-sort on the first item

<table>
<thead>
<tr>
<th>FOR</th>
<th>BAR</th>
<th>BAR</th>
</tr>
</thead>
<tbody>
<tr>
<td>BOX</td>
<td>FAR</td>
<td>BOX</td>
</tr>
<tr>
<td>FOX</td>
<td>FOR</td>
<td>FAR</td>
</tr>
<tr>
<td>BAR</td>
<td>BOX</td>
<td>FOR</td>
</tr>
<tr>
<td>FAR</td>
<td>FOX</td>
<td>FOX</td>
</tr>
</tbody>
</table>

worst-case: \( O(nk) \).
Fibonacci Numbers

The Fibonacci numbers are easy to compute!

```python
def fib(n):
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        return fib(n-1) + fib(n-2)
```

Unfortunately, this code has running time of about $O(1.6^n)$. Why so bad? We repeatedly call the function with the same inputs.
Idea 1: “Memoizing”

Keep track of all questions and answers. If we see a repeat, immediately return the answer.

```python
memopad = {0 : 0, 1 : 1}

def fib(n):
    if memopad.has_key(n):
        return memopad[n]
    else:
        answer = fib(n-1) + fib(n-2)
        memopad[n] = answer
    return answer
```
Idea 2: “Dynamic Programming”

Figure out ahead of time which calls will be needed. Do them in a clever order to avoid checking

def fib(n):
    table = {0 : 0, 1 : 1}
    for i in range(2, n+1):
        table[i] = table[i-1] + table[i-2]
    return table[n]
The Knapsack Problem

Suppose we have \( n \) kinds of items.

✓ They have value (or utility) \( v_1, \ldots, v_n \)

✓ Each has weight (or size or cost) \( w_1, \ldots, w_n \).

What should we choose, if the maximum total weight is \( W \)?
Knapsack Example

Suppose “a friend” can consume up to 13 units of candy.

✓ What’s the maximum possible “value”?
✓ Does “greed” work?
✓ Does “use it or lose it” work?

<table>
<thead>
<tr>
<th>i</th>
<th>w₁ = 2</th>
<th>v₁ = 100</th>
</tr>
</thead>
<tbody>
<tr>
<td>i = 1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>i</th>
<th>w₂ = 3</th>
<th>v₂ = 120</th>
</tr>
</thead>
<tbody>
<tr>
<td>i = 2</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>i</th>
<th>w₃ = 5</th>
<th>v₃ = 230</th>
</tr>
</thead>
<tbody>
<tr>
<td>i = 3</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>i</th>
<th>w₄ = 7</th>
<th>v₄ = 560</th>
</tr>
</thead>
<tbody>
<tr>
<td>i = 4</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>i</th>
<th>w₅ = 9</th>
<th>v₅ = 675</th>
</tr>
</thead>
<tbody>
<tr>
<td>i = 5</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
**Dynamic Programming Solution**

Create a table of maximum value, indexed by total weight

Fill it in “bottom up”

The last entry is your answer!

<table>
<thead>
<tr>
<th>i</th>
<th>v_i</th>
<th>w_i</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>100</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>120</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>230</td>
<td>5</td>
</tr>
<tr>
<td>4</td>
<td>560</td>
<td>7</td>
</tr>
<tr>
<td>5</td>
<td>675</td>
<td>9</td>
</tr>
</tbody>
</table>