algorithm evaluation

Performance

formal \hspace{1cm} \text{Big O}, \hspace{1cm} \text{for initial design considerations}

"big" \rightarrow \text{high level analysis, } "O" \text{ on order (magnitude)}

experimental \hspace{1cm} \text{measure representative cases (probabilistic variations), for time / space measurements:}

performance \parallel \text{system}

Correctness / understandability

qualitative \hspace{1cm} \text{code review / structured walkthroughs}

for maintenance / modification

software metrics \hspace{1cm} \text{measure, KLOC (thousands lines of code)}

\text{complexity (graphs, ...),}

for maintenance / modification

Size and criticality of problem determines degree of algorithm evaluation required.
Big O simplifications

Big O is an "on order" analysis, efficiency of algorithms estimate, statement of algorithms efficiency growth rate

Time & Memory \( \leftarrow \) \( \text{fn(critical operations, frequency of operations)} \)

\( O(...) \leftarrow \) simplification of \( \text{fn(critical operation, frequency of operation)} \)

- remove constants, assume critical operation(s) are constant
- ignore constants \( O(n) \leftarrow O(n - 1) \)
- ignore low order terms \( O(n^3) \leftarrow O(n^3 + n^2 + n) \)
- ignore multiplicative constants \( O(n^2) \leftarrow O(7 \times n^2) \)
- combine growth rates \( O(n^2 + n) \leftarrow O(n^2) + O(n) \)
- ignore log base \( O(\log n) \leftarrow O(\log_2 n) \)
## Big O values

<table>
<thead>
<tr>
<th>Complexity</th>
<th>Category</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>O(1)</td>
<td>constant</td>
<td>Index into an array, hash functions, pop/push stack, add/remove queue</td>
</tr>
<tr>
<td>O(n)</td>
<td>linear</td>
<td>Search unsorted array or linked list array, list, or BST traversal</td>
</tr>
<tr>
<td>O(n^2)</td>
<td>quadratic</td>
<td>Simple sorts: selection, insertion 2 nested loop through n items: duplicates</td>
</tr>
<tr>
<td>O(log n)</td>
<td>logarithmic</td>
<td>Binary search in a sorted list BST insert, find, Heap insert, remove</td>
</tr>
<tr>
<td>O(n log n)</td>
<td>&quot;n log n&quot;</td>
<td>Mergesort, quicksort recursive divide and conquer algorithms</td>
</tr>
<tr>
<td>O(a^n)</td>
<td>exponential</td>
<td>Tower of Hanoi, recursive fibonacci, permutations a &gt; 1</td>
</tr>
</tbody>
</table>
growth rate

FIGURE 10-3
A comparison of growth-rate functions

from text, Prichard & Carrano, page 512
Sort & Algorithm analysis

Sorting algorithms are good examples for algorithm analysis

number of items compare

number of times items are swapped

Consider the insertion sort

```java
v[]  // assume v is filled with random numbers
for (i = 0; i < v.length - 1; i++) {  // n - 1
    min = i
    for (j = i + 1; j < v.length; j++)  // n/2
        if (v[min] > v[j]) min = j
    if (v[i] > v[min]) swap(v[i], v[min])  // n/2 - 1
}
```

One would expect

<table>
<thead>
<tr>
<th>n = 1,000</th>
<th>expected</th>
<th>simulation</th>
</tr>
</thead>
<tbody>
<tr>
<td>n^2/2 - n/2</td>
<td>499,500</td>
<td>499,500</td>
</tr>
<tr>
<td>n/2 - 1</td>
<td>999</td>
<td>988</td>
</tr>
</tbody>
</table>

\[ n^2/2 - 1 \leq n^2/2 - n/2 + n/2 - 1 \]

O(n^2)
Divide and Conquer

MergeSort, sorts an array by

divide: recursive split into halves until single "sorted" values
conquer: merge the sorted "halves" into larger sorted "wholes",
Mergesort

MergeSort sorts within the array using subscripts requires a temporary array (t[]) for merging. a merge copies from v[] to t[] and then from t[] to v[]

Element v[], t[]; // v's "Element" is comparable

// Pre: v initially unsorted, t is empty
// Post: v is sorted

ms(v, t)
    split(0, v.length-1);

split(int begin, int end) {
    if (begin < end) { // split and merge
        int mid = (begin + end) / 2;
        split(begin, mid);
        split(mid+1, end);
        merge(begin, mid, end);
    }
}
merge(int first, int mid, int last) {

    int first1 = first, last1 = mid;
    int first2 = mid + 1, last2 = last;
    int i = first1;

    while((first1 <= last1) && (first2 <= last2)) {
        if (v[first1].compareTo(v[first2]) < 0) {
            t[i] = v[first1];
            first1++; i++;
        } else {
            t[i] = v[first2];
            first2++; i++;
        }
    }

    while(first1 <= last1) { // copy rest first half
        t[i] = v[first1];
        first1++; i++;
    }

    while(first2 < last2) { // copy rest second half
        t[i] = v[first2];
        first2++; i++;
    }

    for(i = first; i <= last; i++) // update v with t
        v[i] = t[i];
}
Merge Sort v[10]

Before v:  9 1 9 8 7 2 7 3 1 6
split into v[0]..v[4] and v[5]..v[9]
split into v[0]..v[2] and v[3]..v[4]
split into v[0]..v[1] and v[2]..v[2]
split into v[0]..v[0] and v[1]..v[1]
merge(0 to 0 and 1 to 1):  1 9 9 8 7 2 7 3 1 6
merge(0 to 1 and 2 to 2):  1 9 9 8 7 2 7 3 1 6
merge(3 to 3 and 4 to 4):  1 9 9 7 8 2 7 3 1 6
merge(0 to 2 and 3 to 4):  1 7 8 9 9 2 7 3 1 6
split into v[5]..v[7] and v[8]..v[9]
split into v[5]..v[6] and v[7]..v[7]
merge(5 to 5 and 6 to 6):  1 7 8 9 9 2 7 3 1 6
merge(5 to 6 and 7 to 7):  1 7 8 9 9 2 3 7 1 6
split into v[8]..v[8] and v[9]..v[9]
merge(8 to 8 and 9 to 9):  1 7 8 9 9 1 2 3 6 7
merge(5 to 7 and 8 to 9):  1 7 8 9 9 1 2 3 6 7
merge(0 to 4 and 5 to 9):  1 1 2 3 6 7 7 8 9 9
After v:  1 1 2 3 6 7 7 8 9 9
Mergesort time performance

Split performance is recursive division  Each "m" level \( \Rightarrow 2^m \) calls

Merge performance: merge step is algorithms "most effort"

merge (begin, mid, last)  merges n items

1.  n -1 comparisons
2.  move n items to t[]
3.  move n items back to v[]

first call \( \Rightarrow 3 \times n - 1 \)  operations

each call creates 2 m calls with n/2 items

\[
2 (3 \times n/2 - 1) \Rightarrow 3 \times n - 2 \text{ operations}
\]

the \( 2^m \) calls \( \Rightarrow 3 \times (n - 2^m) \Rightarrow O(n) \)

Each recursion level requires O(n) operations

with \( \log_2 n \) levels \( \Rightarrow O(\log n) \)

Mergesort is \( O( n \times \log n) \)    average case = worst case
Quicksort

Consider an array partitioned by a "pivot" point such that all elements (begin .. pivot-1) < pivot <= (pivot + 1 ... last) then sort the smaller subarrays values in sorted array < pivot values closer, smaller problem

\[ v[\] \]

\[ qsort(int begin, int end) \]

\[ if \ (begin < end) \]

\[ pivot = partition(begin, end) \]

\[ qsort(begin, pivot -1) \]

\[ qsort(pivot + 1, end) \]
Consider the array during partitioning

There is the pivot, initially at the beginning
The subarray S1 with values less than the pivot
The subarray S2 with values greater than the pivot
The subarray of values not yet "sorted" into S1 or S2 wrt pivot
Maintain the invariant S1 has values < pivot and S2 has values >= pivot

\[ v[] \]

```c
int partition(int begin, int end) {
    pivot = begin
    low = begin
    hi = end
    while (low < hi) {
        while ( (low < hi) && (v[low] <= v[hi]) ) hi--
        if (low < hi) 
            swap(low, hi)
        pivot = hi
        low++
        while ( (low < hi) && (v[low] < v[hi]) ) low++
        if (low < hi)
            swap(low, hi)
        pivot = low
        hi--
    }
    return pivot
}
```

Partition(0, 4):
\[ s(4, 0) \rightarrow [0 3 0 4 4], \]
\[ 3 < 4, 0 < 4, \]
\[ s(4, 4) \rightarrow [0 3 0 4 4], \]
\[ pivot = 3 \]
QuickSort v[10]

|7| 4 2 4 5 8 2 2 6 5  qsort(0, 9)
Partition(0, 9): s(7, 5), 4 < 7, 2 < 7, 4 < 7, 5 < 7, s(8, 7), s(7, 6), 2 < 7, 2 < 7, pivot = 8
5 4 2 4 5 6 2 2 |7| 8
qsort(0, 7)
Partition(0, 7): s(5, 2), 4 < 5, 2 < 5, 4 < 5, s(5, 5), s(5, 2), s(6, 5), pivot = 5
2 4 2 4 2 |5| 6 5 7 8
qsort(0, 4)
Partition(0, 4): 2 <= 2, 2 <= 4, 2 <= 2, 2 <= 4, pivot = 0
2 4 2 4 2 5 6 5 7 8
qsort(0, -1), halt
qsort(1, 4)
Partition(1, 4): s(4, 2), 2 < 4, s(4, 4), pivot = 3
2 2 2 |4| 4 5 6 5 7 8
qsort(1, 2)
Partition(1, 2): 2 <= 2, pivot = 1
2 |2| 2 4 4 5 6 5 7 8
qsort(1, 0), halt
qsort(2, 2), halt
qsort(4, 4), halt
Quicksort time performance

//  2 2 2 4 4 5 6 5 7 8  from previous page

qsort(6, 7)
Partition(6, 7): s(6, 5), :: pivot = 7
2 2 2 4 4 5 5 |6| 7 8
qsort(6, 6), halt
qsort(8, 7), halt
qsort(9, 9), halt
2 2 2 4 4 5 5 6 7 |8|

Time Performance

QuickSort performance analysis is complex; there are several approaches
to selecting a pivot (partitioning).

average is O( n * log n) and worst is O(n^2)

Worst case the array is sorted and pivot is the first position …. 
Mergesort Vs Quicksort

Quicksort and mergesort take $O(n \log n)$ time and hence time taken to sort the elements remains same.

Quicksort is superior than mergesort in terms of space.

Quicksort is an in-place sorting algorithm; merge sort is not in-place. In-place sorting means, it does not use additional storage space to perform sorting. In mergesort, to merge the sorted arrays it requires a temporary array; it is not in-place.

However time efficiency of the quick sort depends on the choice of the pivot element. Normally the middle or median element is chosen.

Sort choice can be problem domain sensitive ….
exercises

1. What is the O(n) value for computing the standard deviation from an array of n values?

2. How many comparisons would you expect to make to find an element that exists in a sorted array of 1024 values using a (1) linear search and (2) binary search.

3. For the same array in #2 how many comparisons would you make to determine that a value was not in the array?


5. Algorithm analysis is often described in terms of time or space. Explain why?

6. Describe how "big O" analysis is used in selecting among possible algorithms to solve a problem.