9. Sorting

Frank Stephan

March 20, 2014

Sorting

Given a list $(a_i, \ldots, a_n)$ of n elements output a permutation $(a'_i, \ldots, a'_n)$ such that $a'_i \leq a'_j$ whenever $i \leq j$.

Many Algorithms!

Insertion sort, selection sort, bubble sort, quick sort, count sort, bucket sort, etc.

AlgoRythmics

http://www.youtube.com/user/AlgoRythmics

Insertion Sort

for each element in the list move it to the sorted beginning of the list and insert it immediately after the first element encountered that is smaller.

```
1 function sort(A)
2 { var key;
3   for (j=1;j<A.length;j++)
4   { key = A[j];
5     i = j-1
6     while (i>=0 && A[i] > key)
7     { A[i+1] = A[i]; i--; }
8     A[i+1]=key; }
9   return A; }
```

"code/insertion_sort.js"

Insertion Sort

- Worst case performance $O(n^2)$.
- Best case performance $O(n)$.
- Average case performance $O(n^2)$.
- Worst case space complexity $O(n)$.

In Place

Insertion Sort is an in place sorting algorithm. It only needs a constant amount of auxiliary memory.

Stable

Insertion Sort is an stable sorting algorithm. It maintains the relative order of elements with the same key.

Selection Sort

Find the smallest element in the list and put it at the beginning of the not yet sorted elements.

```
1 function sort(A)
2 { var i, j, key, tmp;
3   for (i=0;i<A.length;i++)
4   { key = i;
5     for (j= i+1; j < A.length; j++)
6     { if (A[j] < A[key]) { key=j; } }
7     tmp=A[key]; A[key] = A[i]; A[i] = tmp; }
8   return A; }
```

"code/selection_sort.js"

Selection Sort

- Worst case performance $O(n^2)$.
- Best case performance $O(n^2)$.
- Average case performance $O(n^2)$.
- Worst case space complexity $O(n)$.

Bubble Sort

Move up the largest elements in the list.

```
1 function sort(A)
2 { var i, j, tmp;
3   for (i = 0;  n < A.length; n++)
4   { for (j=0 m < A.length-1; m++)
5     { if (A[y] > A[y+1])
6       { tmp = A[y+1]; A[y+1] = A[y]; A[y]=tmp; } } }
7 return A; }
```

"code/bubble_sort.js"

Bubble Sort Optimization

The sorting can stop if no swapping occurs in the inner loop.

Bubble Sort

- Worst case performance $O(n^2)$.
- Best case performance $O(n)$.
- Average case performance $O(n^2)$.
- Worst case space complexity $O(n)$.

Quick Sort

Choose an element in the list. Let us call it the pivot. Move all elements smaller than the pivot element at the beginning of the list. Repeat the operation for both the list before the pivot and the list after the pivot.

```
1  function sort(A)
2  { var Left = []; var Middle = []; var Right = [];
3    var i; var pivot = Math.floor(A.length/2);
4    if (A.length > 1)
5    { for (i=0; i <A.length; i++)
6      { if (A[i] < A[pivot]) { Left.push(A[i]); }
7        else if (A[i] == A[pivot]) { Middle.push(A[i]); }
8        else if (A[i] > A[pivot]) { Right.push(A[i]); } }
9      Left = sort(Left); Right = sort(Right);
10     A = Left.concat(Middle, Right); }
11   return A; }
```

"code/quick_sort.js"

Quick Sort

- Worst case performance $O(n^2)$.
- Best case performance $O(n \log n)$.
- Average case performance $O(n \log n)$.
- Worst case space complexity $O(n)$.

Merge Sort

Split the list in two halves. Sort the two halves. Merge the two lists (using at most one comparison per element.)

Merge Sort

```
1  function sort(A)
2  { var result;
3    if (A.length >1)
4    { var pivot = Math.floor(A.length / 2);
5      var Left = A.slice(0, pivot);
6      var Right = A.slice(pivot, A.length);
7      Left = sort(Left); Right = sort(Right);
8      result = merge(Left, Right); }
9    else { result = A; }
10   return result; } }
```

"code/merge_sort.js"

Merge Sort

```
1  function merge(L, R)
2  { var result = new Array();
3    while (L.length > 0 && R.length > 0)
4    { if (L[0] <= R[0]) { result.push(L.shift()); }
5      else {result.push(R.shift()); } }
6    while (L.length >0)
7    { result.push(L.shift()); }
8    while (R.length > 0)
9    { result.push(R.shift()); }
10   return result; }
```

"code/merge.js"

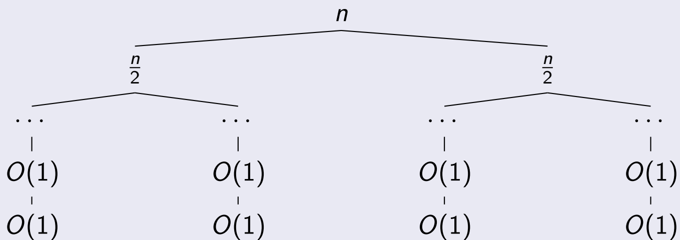
Recurrence Relation

The number of comparisons is $T(n)$ defined by the following recurrence relation (equation).

$$
\begin{aligned}
\text{If } n \leq 2 \quad & \text{then} \quad T(n) = 1 \\
& \text{else} \quad T(n) = 2 \cdot T(\frac{n}{2}) + n
\end{aligned}
$$

How to solve a recurrence equation?

Recursion Tree



Solving Recurrence Equation

We sum all the levels of the recursion tree.

$$T(n) = n \cdot \log(n)$$

Soving Recurrence Equation (Alt.)

We consider the case where n is a power of 2 (i.e. $n = 2^m$).

$$
\begin{aligned}
T(n) &= T(2^m) \\
&= 2^m + 2 \cdot T(2^{m-1}) \\
&= 2^m + 2 \cdot (2^{m-1} + 2 \cdot T(2^{m-2})) \\
&= 2^m + 2^m + 2^2 \cdot T(2^{m-2}) \\
&= k \cdot 2^m + 2^k \cdot T(2^{m-k}) \\
&= (m-1) \cdot 2^m + 2^{m-1} \cdot T(2^1) \\
&\leq m \cdot 2^m = \log(n) \cdot n
\end{aligned}
$$

Master Theorem

$$T(n) = a \cdot T(\frac{n}{b}) + f(n)$$

1. If $f(n) \in O(n^{\log_b(a)-\epsilon})$, then $T(n) \in \Theta(n^{\log_b(a)})$.
2. If $f(n) \in \Theta(n^{\log_b(a)})$, then $T(n) \in \Theta(n^{\log_b(a)} \cdot log_b(n))$
3. If $f(n) \in \Omega(n^{\log_b(a)+\epsilon})$, then $T(n) \in \Theta(f(n))$.

Soving Recurrence Equation (Alt.)

$$T(n) = 2 \cdot T(\frac{n}{2}) + n$$

$a = 2,\ b = 2,\ f(n) = n.$

$$T(n) \in \Theta(n \cdot \log(n))$$

Merge Sort

- Worst case performance $O(n \log n)$.
- Best case performance $O(n \log n)$.
- Average case performance $O(n \log n)$.
- Worst case space complexity $O(n)$.

What can we Expect?

- A sorting algorithm can compare, copy and move elements. No other information is available.
- The running time of a sorting algorithm depends on the number of comparisons.

Sorting Two Elements

There are only two possible cases. One comparison allows sorting two elements.

- If $e_1 \leq e_2$ then (e_1, e_2) is sorted.
- Else (e_2, e_1) is sorted.

Sorting Three Elements

There are only six possible cases. Three comparisons allow sorting three elements.

- If $e_1 \leq e_2$ and $e_2 \leq e_3$ and $e_1 \leq e_3$ then (e_1, e_2, e_3) is sorted.
- If $e_1 \leq e_2$ and $e_2 \leq e_3$ and $e_3 < e_1$ is impossible.
- If $e_1 \leq e_2$ and $e_3 < e_2$ and $e_1 < e_3$ then (e_1, e_3, e_2) is sorted.
- If $e_1 \leq e_2$ and $e_3 < e_2$ and $e_3 < e_1$ then (e_3, e_1, e_2) is sorted.
- If $e_2 < e_1$ and $e_2 < e_3$ and $e_1 < e_3$ then (e_2, e_1, e_3) is sorted.
- If $e_2 < e_1$ and $e_2 \leq e_3$ and $e_3 < e_1$ then (e_2, e_3, e_1) is sorted.
- If $e_2 < e_1$ and $e_3 < e_2$ and $e_1 < e_3$ is impossible
- If $e_2 < e_1$ and $e_3 < e_2$ and $e_3 < e_1$ then (e_3, e_2, e_1) is sorted.

Sorting n Elements

- There are $n!$ possible cases (permutation of n elements).
- With m comparisons we can cater for 2^m cases (some impossible).
- We need m comparisons such that $2^m \geq n!$.
- $n! > (n/2)^{n/2}$; thus $\log(n!) > \log(n/2) \cdot n/2$.
- Therefore $m > n/2 \cdot \log(n/2)$.

Lower Bound

Sorting is $\Omega(n \log(n))$.

Count Sort

If the list contains numbers between 0 and some known maximum, possibly with some duplicates, we count just count the number of occurrences of each number and insert each number as many times as it occurs in the list in increasing order.

```
1  function count_sort(A)
2  { var i,j;
3    var count = new Array();
4    for (i in A)
5    { count[A[i]] = 0; }
6    for (i in A)
7    { count[A[i]]++; }
8    var B = new Array(); var k=0;
9    for (i in count)
10   { for (j=0;j<count[i];j++)
11     { B[k] = i; k++; } }
12   return B; }
13 // The loop "for i in count" lets
14 // i go in ascending numerical order.
```

"code/count_sort.js"

Load the sorting algorithm into a browser

Complexity

- It takes n steps to count the occurrences.
- It takes n insertions to create the sorted list.
- Count Sort is $O(n)$.

Linear

- How is it possible? We were expecting $O(n \log n)$ at best!
- It is not a comparison sort (see also bucket sort).
- The JavaScript interpreter might also cause some runtime overhead by maintaining the dynamical array "count"; this can be avoided by more careful programming when all values in A are from $\{0, 1, \ldots, n\}$.

```
1  function count_sort(A)
2  { var i,j; var max = 0;
3    for (i in A)
4    { if (A[i]>max) { max = A[i]; } }
5    var count = new Array(max+1);
6    for (i=0;i<=max;i++)
7    { count[i] = 0; }
8    for (i in A)
9    { count[A[i]]++; }
10   var B = new Array(); var k=0;
11   for (i=0;i<=max;i++)
12   { for (j=0;j<count[i];j++)
13     { B[k] = i; k++; } }
14   return B; }
15 // Run Time is O(A.length+max);
```

"code/count_sort_static.js"

Load the sorting algorithm into a browser

Attribution

The images and media files used in this presentation are either in the public domain or are licensed under the terms of the GNU Free Documentation License, Version 1.2 or any later version published by the Free Software Foundation, the Creative Commons Attribution-Share Alike 3.0 Unported license or the Creative Commons Attribution-Share Alike 2.5 Generic, 2.0 Generic and 1.0 Generic license.