CS 5114: Theory of Algorithms

Clifford A. Shaffer

Department of Computer Science
Virginia Tech
Blacksburg, Virginia

Spring 2010

Copyright © 2010 by Clifford A. Shaffer

Title page

Sorting

Each record contains a field called the **key**.
Linear order: comparison.

The Sorting Problem

Given a sequence of records $R_1, R_2, ..., R_n$ with key values $k_1, k_2, ..., k_n$, respectively, arrange the records into any order s such that records $R_{s_1}, R_{s_2}, ..., R_{s_n}$ have keys obeying the property $k_{s_1} \leq k_{s_2} \leq ... \leq k_{s_n}$.

Measures of cost:

- Comparisons
- Swaps

Linear order means: $a < b$ and $b < c \Rightarrow a < c$.

More simply, sorting means to put keys in ascending order.

Insertion Sort

```
void inssort(Elem* A, int n) { // Insertion Sort
  for (int i=1; i<n; i++)   // Insert i'th record
    for (int j=i; (j>0) && (A[j].key<A[j-1].key);
         j--)
      swap(A, j, j-1);
}
```

	i=1	2	3	4	5	6	7
42	20	17	13	13	13	13	13
20	42	20	17	17	14	14	14
17	17	42	20	20	17	17	15
13	13	13	42	28	20	20	17
28	28	28	28	42	28	23	20
14	14	14	14	14	42	28	23
23	23	23	23	23	23	42	28
15	15	15	15	15	15	15	42

Best Case:
Worst Case:
Average Case:

Best case is 0 swaps, $n - 1$ comparisons.
Worst case is $n^2/2$ swaps and compares.
Average case is $n^2/4$ swaps and compares.

Insertion sort has great best-case performance.

Exchange Sorting

- **Theorem**: Any sort restricted to swapping adjacent records must be $\Omega(n^2)$ in the worst and average cases.
- **Proof**:
  - For any permutation P, and any pair of positions i and j, the relative order of i and j must be wrong in either P or the inverse of P.
  - Thus, the total number of swaps required by P and the inverse of P MUST be

$$\sum_{i=1}^{n-1} i = \frac{n(n-1)}{2}.$$

$n^2/4$ is the average distance from a record to its position in the sorted output.

Quicksort

Divide and Conquer: divide list into values less than pivot and values greater than pivot.

```
void qsort(Elem* A, int i, int j) { // Quicksort
  int pivotindex = findpivot(A, i, j);
  swap(A, pivotindex, j);        // Swap to end
  // k will be first position in right subarray
  int k = partition(A, i-1, j, A[j].key);
  swap(A, k, j);             // Put pivot in place
  if ((k-i) > 1) qsort(A, i, k-1);  // Sort left
  if ((j-k) > 1) qsort(A, k+1, j);  // Sort right
}

int findpivot(Elem* A, int i, int j)
  { return (i+j)/2; }
```


Initial call: `qsort(array, 0, n-1);`

Quicksort Partition

```
int partition(Elem* A, int l, int r, int pivot) {
  do {      // Move bounds inward until they meet
    while (A[++l].key < pivot); // Move right
    while (r && (A[--r].key > pivot));// Left
    swap(A, l, r);      // Swap out-of-place vals
  } while (l < r);      // Stop when they cross
  swap(A, l, r);        // Reverse wasted swap
  return l;     // Return first position in right
}
```

The cost for Partition is $\Theta(n)$.

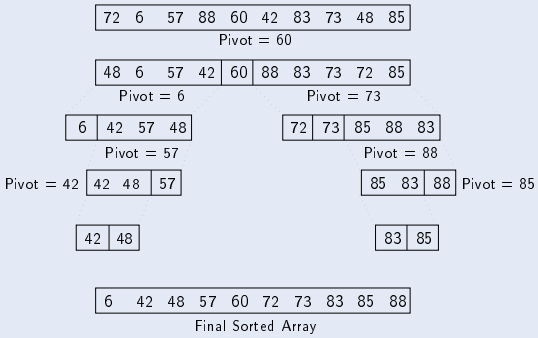
no notes

Partition Example

Initial	72	6	57	88	85	42	83	73	48	60
Pass 1	72	6	57	88	85	42	83	73	48	60
Swap 1	48	6	57	88	85	42	83	73	72	60
Pass 2	48	6	57	88	85	42	83	73	72	60
Swap 2	48	6	57	42	85	88	83	73	72	60
Pass 3	48	6	57	42	85	88	83	73	72	60
Swap 3	48	6	57	85	42	88	83	73	72	60
Reverse Swap	48	6	57	42 |	85	88	83	73	72	60

no notes

Quicksort Example



no notes

Cost for Quicksort

Best Case: Always partition in half.

Worst Case: Bad partition.

Average Case:

$$f(n) = n - 1 + \frac{1}{n}\sum_{i=0}^{n-1}(f(i) + f(n-i-1))$$

Optimizations for Quicksort:

- Better pivot.
- Use better algorithm for small sublists.
- Eliminate recursion.
- Best: Don't sort small lists and just use insertion sort at the end.

Cost for Quicksort
Best Case: Always partition in half.
Worst Case: Bad partition.
Average Case:
$f(n) = n - 1 + \frac{1}{n}\sum_{i=0}^{n-1}(f(i) + f(n-i-1))$
Optimizations for Quicksort:
Better pivot.
Use better algorithm for small sublists.
Eliminate recursion.
Best: Don't sort small lists and just use insertion sort at the end.

Think about when the partition is bad. Note the FindPivot function that we used is pretty good, especially compared to taking the first (or last) value.
Also, think about the distribution of costs: Line up all the permuations from most expensive to cheapest. How many can be expensive? The area under this curve must be low, since the average cost is $\Theta(n \log n)$, but some of the values cost $\Theta(n^2)$. So there can be VERY few of the expensive ones.

This optimization means, for list threshold T, that no element is more than T positions from its destination. Thus, insertion sort's best case is nearly realized. Cost is at worst nT.

Quicksort Average Cost

$$f(n) = \begin{cases} 0 & n \le 1 \\ n - 1 + \frac{1}{n}\sum_{i=0}^{n-1}(f(i) + f(n-i-1)) & n > 1 \end{cases}$$

Since the two halves of the summation are identical,

$$f(n) = \begin{cases} 0 & n \le 1 \\ n - 1 + \frac{2}{n}\sum_{i=0}^{n-1} f(i) & n > 1 \end{cases}$$

Multiplying both sides by n yields

$$nf(n) = n(n-1) + 2\sum_{i=0}^{n-1} f(i).$$

Quicksort Average Cost
$f(n) = \begin{cases} 0 & n \le 1 \\ n - 1 + \frac{1}{n}\sum_{i=0}^{n-1}(f(i) + f(n-i-1)) & n > 1 \end{cases}$
Since the two halves of the summation are identical,
$f(n) = \begin{cases} 0 & n \le 1 \\ n - 1 + \frac{2}{n}\sum_{i=0}^{n-1} f(i) & n > 1 \end{cases}$
Multiplying both sides by n yields
$nf(n) = n(n-1) + 2\sum_{i=0}^{n-1} f(i).$

This is a "recurrence with full history".

Think about what the pieces correspond to.
To do Quicksort on an array of size n, we must:

- Partation: Cost n
- Findpivot: Cost c
- Do the recursion: Cost dependent on the pivot's final position.

These parts are modeled by the equation, including the average over all the cases for position of the pivot.

Average Cost (cont.)

Get rid of the full history by subtracting $nf(n)$ from $(n+1)f(n+1)$

$$\begin{aligned} nf(n) &= n(n-1) + 2\sum_{i=1}^{n-1} f(i) \\ (n+1)f(n+1) &= (n+1)n + 2\sum_{i=1}^{n} f(i) \\ (n+1)f(n+1) - nf(n) &= 2n + 2f(n) \\ (n+1)f(n+1) &= 2n + (n+2)f(n) \\ f(n+1) &= \frac{2n}{n+1} + \frac{n+2}{n+1}f(n). \end{aligned}$$

Average Cost (cont.)
Get rid of the full history by subtracting $nf(n)$ from $(n+1)f(n+1)$
$$\begin{aligned} nf(n) &= n(n-1) + 2\sum_{i=1}^{n-1} f(i) \\ (n+1)f(n+1) &= (n+1)n + 2\sum_{i=1}^{n} f(i) \\ (n+1)f(n+1) - nf(n) &= 2n + 2f(n) \\ (n+1)f(n+1) &= 2n + (n+2)f(n) \\ f(n+1) &= \frac{2n}{n+1} + \frac{n+2}{n+1}f(n). \end{aligned}$$

no notes

Average Cost (cont.)

Note that $\frac{2n}{n+1} \le 2$ for $n \ge 1$.
Expand the recurrence to get:

$$\begin{aligned} f(n+1) &\le 2 + \frac{n+2}{n+1}f(n) \\ &= 2 + \frac{n+2}{n+1}\left(2 + \frac{n+1}{n}f(n-1)\right) \\ &= 2 + \frac{n+2}{n+1}\left(2 + \frac{n+1}{n}\left(2 + \frac{n}{n-1}f(n-2)\right)\right) \\ &= 2 + \frac{n+2}{n+1}\left(2 + \cdots + \frac{4}{3}(2 + \frac{3}{2}f(1))\right) \end{aligned}$$

Average Cost (cont.)
Note that $\frac{2n}{n+1} \le 2$ for $n \ge 1$.
Expand the recurrence to get:
$$\begin{aligned} f(n+1) &\le 2 + \frac{n+2}{n+1}f(n) \\ &= 2 + \frac{n+2}{n+1}\left(2 + \frac{n+1}{n}f(n-1)\right) \\ &= 2 + \frac{n+2}{n+1}\left(2 + \frac{n+1}{n}\left(2 + \frac{n}{n-1}f(n-2)\right)\right) \\ &= 2 + \frac{n+2}{n+1}\left(2 + \cdots + \frac{4}{3}(2 + \frac{3}{2}f(1))\right) \end{aligned}$$

no notes

Average Cost (cont.)

$$
\begin{aligned}
f(n+1) &\le 2\left(1+\frac{n+2}{n+1}+\frac{n+2}{n+1}\frac{n+1}{n}+\cdots \right.\\
&\qquad \left. +\frac{n+2}{n+1}\frac{n+1}{n}\cdots\frac{3}{2}\right)\\
&= 2\left(1+(n+2)\left(\frac{1}{n+1}+\frac{1}{n}+\cdots+\frac{1}{2}\right)\right)\\
&= 2+2(n+2)\left(\mathcal{H}_{n+1}-1\right)\\
&= \Theta(n\log n).
\end{aligned}
$$

Average Cost (cont.)

$$
\begin{aligned}
f(n+1) &\le 2\left(1+\frac{n+2}{n+1}+\frac{n+2}{n+1}\frac{n+1}{n}+\cdots \right.\\
&\qquad \left. +\frac{n+2}{n+1}\frac{n+1}{n}\cdots\frac{3}{2}\right)\\
&= 2\left(1+(n+2)\left(\frac{1}{n+1}+\frac{1}{n}+\cdots+\frac{1}{2}\right)\right)\\
&= 2+2(n+2)\left(\mathcal{H}_{n+1}-1\right)\\
&= \Theta(n\log n).
\end{aligned}
$$

$\mathcal{H}_{n+1} = \Theta(\log n)$

Mergesort

```
List mergesort(List inlist) {
  if (inlist.length() <= 1) return inlist;;
  List l1 = half of the items from inlist;
  List l2 = other half of the items from inlist;
  return merge(mergesort(l1), mergesort(l2));
}
```

36 20 17 13 28 14 23 15

[20 36] [13 17] [14 28] [15 23]

[13 17 20 36] [14 15 23 28]

[13 14 15 17 20 23 28 36]

Mergesort

no notes

Mergesort Implementation (1)

Mergesort is tricky to implement.

```
void mergesort(Elem* A, Elem* temp,
               int left, int right) {
  int mid = (left+right)/2;
  if (left == right) return;       // List of one
  mergesort(A, temp, left, mid);   // Sort half
  mergesort(A, temp, mid+1, right);// Sort half
  for (int i=left; i<=right; i++) // Copy to temp
    temp[i] = A[i];
```


Mergesort Implementation (1)

This implementation requires a second array.

Mergesort Implementation (2)

```
  // Do the merge operation back to array
  int i1 = left; int i2 = mid + 1;
  for (int curr=left; curr<=right; curr++) {
    if (i1 == mid+1)     // Left list exhausted
      A[curr] = temp[i2++];
    else if (i2 > right) // Right list exhausted
      A[curr] = temp[i1++];
    else if (temp[i1].key < temp[i2].key)
      A[curr] = temp[i1++];
    else A[curr] = temp[i2++];
}}
```

Mergesort cost:
Mergesort is good for sorting linked lists.

Mergesort Implementation (2)

Mergesort cost: $\Theta(n \log n)$

Linked lists: Send records to alternating linked lists, mergesort each, then merge.

Heaps

Heap: Complete binary tree with the **Heap Property**:

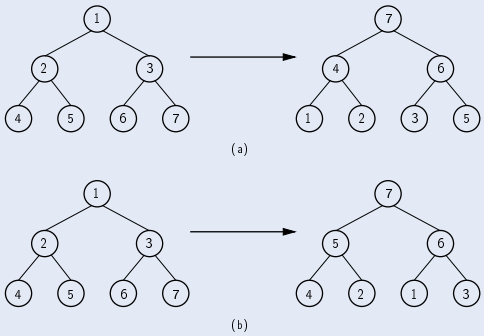
- Min-heap: all values less than child values.
- Max-heap: all values greater than child values.

The values in a heap are **partially ordered**.

Heap representation: normally the array based complete binary tree representation.

no notes

Building the Heap



(a) requires exchanges (4-2), (4-1), (2-1), (5-2), (5-4), (6-3), (6-5), (7-5), (7-6).
(b) requires exchanges (5-2), (7-3), (7-1), (6-1).

This is a Max Heap
How to get a good number of exchanges? By induction.
Heapify the root's subtrees, then push the root to the correct level.

Siftdown

```
void heap::siftdown(int pos) { // Sift ELEM down
  assert((pos >= 0) && (pos < n));
  while (!isLeaf(pos)) {
    int j = leftchild(pos);
    if ((j<(n-1)) &&
        (Heap[j].key < Heap[j+1].key))
      j++; // j now index of child with > value
    if (Heap[pos].key >= Heap[j].key) return;
    swap(Heap, pos, j);
    pos = j;  // Move down
  }
}
```


no notes

BuildHeap

For fast heap construction:

- Work from high end of array to low end.
- Call `siftdown` for each item.
- Don't need to call `siftdown` on leaf nodes.

```
void heap::buildheap()        // Heapify contents
  { for (int i=n/2-1; i>=0; i--) siftdown(i); }
```

Cost for heap construction:

$$\sum_{i=1}^{\log n}(i-1)\frac{n}{2^i} \approx n.$$

$(i-1)$ is number of steps down, $n/2^i$ is number of nodes at that level.

The intuition for why this cost is $\Theta(n)$ is important. Fundamentally, the issue is that nearly all nodes in a tree are close to the bottom, and we are (worst case) pushing all nodes down to the bottom. So most nodes have nowhere to go, leading to low cost.