4 Chapter 4 – Sorting

- Reasons for studying
  - sorting is a big deal
  - pedagogically useful
    * the application itself is easy to understand
    * a complete analysis can often be done
    * different sorting algorithms illustrate different ideas

- Theoretical issues
  - count comparisons
  - worst case versus average case

- Practical issues
  - space
  - recursion versus iteration
  - external sort versus internal sort
  - sort keys, not data
  - number of moves
  - memory to memory? disk to disk?
  - locality of reference?
sort the array ‘a[*]’ of length ‘count’
perform only the first ‘howmany’ sort steps
keep track of the total number of comparisons

```c
void bubble(long a[], long howmany, long count, long *comparisons)
{
    long i, j, temp;
    if (howmany == count) howmany--;
    for (i = 1; i <= howmany; i++)
    {
        for (j = i+1; j <= count; j++)
        {
            comparisons++;
            if (a[i] > a[j])
            {
                temp = a[i];
                a[i] = a[j];
                a[j] = temp;
            }
        }
    }
}
```
sort the array ‘a[*]’ of length ‘count’
perform only the first ‘howmany’ sort steps
keep track of the total number of comparisons

```c
void insert(long a[], long howmany, long count, long *comparisons)
{
    long j, length, temp;
    if (howmany == count) howmany--;
    for (length = 2; length <= howmany; length++)
    {
        for (j = length - 1; j >= 1; j--)
        {
            comparisons++;
            if (a[j] > a[j+1])
            {
                temp = a[j];
                a[j] = a[j+1];
                a[j+1] = temp;
            }
            else
                break;
        }
    }
}
```
sort the array ‘a[*]’ of length ‘count’
perform only the first ‘howmany’ sort steps
keep track of the total number of comparisons

void insert2(long a[], long howmany, long count, long *comparisons)
{
    long j, length, insertvalue, temp;
    if(howmany == count) howmany--;
    for(length = 2; length <= howmany; length++)
    {
        insertvalue = a[length];
        for(j = length-1; j >= 1; j--)
        {
            comparisons++;
            if(a[j] > insertvalue)
            {
                a[j+1] = a[j];
            }
            else
            {
                a[j] = insertvalue;
                break;
            }
        }
    }
}
4.1 Lower Bounds

With some sorting algorithms we can prove lower bounds on the complexity of algorithms. We will describe the nature of sorting algorithms by noting first that any list of \( n \) elements is some permutation \( \pi \) of its sorted order.

**Definition 4.1.** Given a permutation \( \pi \) on \( n \) integer keys, an **inversion** is a pair \((i, j)\) such that \( i < j \) but \( \pi(i) > \pi(j) \).

For example, the list

<table>
<thead>
<tr>
<th>list</th>
<th>inversions</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>(4,3) (5,2)</td>
</tr>
<tr>
<td>3</td>
<td>(4,2) (5,1)</td>
</tr>
<tr>
<td>5</td>
<td>(4,1) (2,1)</td>
</tr>
<tr>
<td>2</td>
<td>(3,2)</td>
</tr>
<tr>
<td>1</td>
<td>(3,1)</td>
</tr>
</tbody>
</table>

**FOR THOSE OF YOU WHO KNOW ABOUT PERMUTATION GROUPS,** note that this definition of inversion is not at all the same as that of a transposition.

For every \( n \), there exists a permutation of \( n(n - 1)/2 \) inversions. (This can be proved either directly or by induction. Note that this requires that every pair be out of order.)

**Theorem 4.2.** A sorting algorithm that sorts by comparison of keys and that removes at most one inversion for every comparison made has worst case running time of at least \( n(n - 1)/2 \) comparisons and average case running time of at least \( n(n - 1)/4 \) comparisons.
Proof. The worst case is obvious at this point. For every \( n \) there exists a permutation with \( n(n - 1)/2 \) inversions, so if only one is removed with each comparison, then worst case running time must require \( n(n - 1)/2 \) comparisons.

Average case: Consider permutations in pairs, of a permutation and its reverse. For each of the \( n(n - 1)/2 \) pairs of elements, each pair is an inversion in exactly one of the two permutations, so the \( n(n-1)/2 \) inversions are shared between the two permutations. On average, then, a random permutation has half that many inversions, and the theorem is proved. \( \square \)

**Corollary 4.3.** Insertion sort has lower order worst case running time of at least \( n(n - 1)/2 \) comparisons and average case running time of at least \( n(n - 1)/4 \) comparisons.

Proof. (of the corollary) Insertion sort has the effect of comparing adjacent elements in a list and exchanging them if they are out of order. (“Has the effect” because one version does not compare adjacent elements but instead creates a vacancy. However, the effect is no different from what would happen if the element to be inserted were in fact inserted into the vacancy at every step, and then it would be true that adjacent elements had been compared.)

If one compares two adjacent elements \( a_n \) and \( a_{n+1} \) that are out of order (that is, \( a_n > a_{n+1} \)) and then exchanges them, the only change in the number of inversions is that the inversion \( (a_n, a_{n+1}) \) is eliminated.

So the conditions of the theorem are met and the corollary is true (assuming the theorem is true.) \( \square \)
4.2 Quicksort

- Let's assume that we have \( n \) items to sort.
- (A) Assume that we could (invoke oracle here) pick out the \( n/2 \)-th element \( a_k \), which we can re-index to be \( a_{n/2} \).
- (B) Now, let's assume that we rearrange the array of \( n \) items so that all items subscripted less than \( n/2 \) are less than \( a_{n/2} \) and all items subscripted larger than \( n/2 \) are larger than \( a_{n/2} \). Then call this procedure recursively, invoking the oracle at every stage to be able to choose the midpoint element every time.
- \( n \) items
- \( \lg n \) recursion depth
- Let's assume that with each recursion, we can arrange the data as in (B) in time proportional to \( n \).
- Then we could sort the entire array in time \( O(n \lg n) \).
- We can't, of course, find an oracle. But if we're lucky and careful, we might be able to come close in the average case to choosing the midpoint element as our pivot element.
- As for the rearrangement (B):
  - Assume the pivot element is the first element in the array. (If it isn't, then exchange the pivot element with the first element.) Pull the pivot element aside to create a vacancy in position one.
  - Work from the end toward the front until we find the first element smaller than the pivot. Put that element in the vacancy in position one, thus creating a vacancy in the second half of the array.
  - Now work forward from position 2 until we find an element that belongs in
the second half of the array. Put that in the vacancy, creating a first-half vacancy.

• and so forth.

• At the end, the vacancy will be in the middle of the array, and we put our pivot element there.

• **Bottom Line:** In time $n$ we can cause the array to be split into a first half smaller than the pivot element and a second half larger than the pivot element.

• Assuming the oracle, then, we could guarantee $n \lg n$ running time.

### 4.2.1 Worst Case

Without the oracle, how bad could this be?

If our pivot element in fact was the smallest element, then we would not be doing a divide and conquer splitting the array size in half each time. We would only be shortening the list by one element every time.

In this case, we will do $n$ recursion steps and have to compare the pivot element against $n - k$ elements in the $k$-th step.

• So we have $n^2$ running time in the worst case.
4.2.2 Average Case

**Theorem 4.4.** Quicksort has average case running time, for large \( n \), of approximately \( 2n \log n \) comparisons.

**Proof.**

- Let \( A(n) \) be the average case number of comparisons for sorting \( n \) elements.
- Our choice of pivot element is random, so it is the \( i \)-th element for random \( i \) and splits the array of length \( n \) into subarrays of length \( i \) and \( n - i - 1 \), with each \( i \) having probability \( 1/n \).
- We do \( n - 1 \) comparisons in the initial split.
- We have \( A(0) = A(1) = 0 \).
- So the average case running time is

\[
A(n) = n - 1 + \sum_{i=0}^{n-1} \frac{A(i) + A(n - i - 1)}{n}, \quad n \geq 2.
\]

- This can be collapsed into

\[
A(n) = n - 1 + \frac{2}{n} \sum_{i=1}^{n-1} A(i), \quad n \geq 1. \tag{*}
\]

because the initial conditions happen to fit the formula and because the \( A(i) \) and \( A(n - i - 1) \) terms are symmetric.

- We have already seen that if we had an oracle and could choose exactly the right pivot point, we could get the average running time down to \( n \log n \).
- What if we make a wild leap of faith that average isn’t asymptotically worse than perfect?
Proposition 4.5. For \( n \geq 1 \), we have \( A(n) \leq cn \log n \) for some constant \( c \).

Proof.

• Induct. True for \( n = 1 \). Assume true \( \forall i, 1 \leq i < n \). From (*) we have

\[
A(n) = n - 1 + \frac{2}{n} \sum_{i=1}^{n-1} A(i) \\
= n - 1 + \frac{2}{n} \sum_{i=1}^{n-1} ci \log i \\
\leq n - 1 + \frac{2}{n} \int_1^n c \ x \log x \ dx \\
= n - 1 + \frac{2c}{n} \left( \frac{n^2 \log n}{2} - \frac{n^2}{4} \right) \\
= n - 1 + c \left( n \log n - \frac{n}{2} \right) \\
= c \ n \log n + \left( 1 - \frac{c}{2} \right)n - 1
\]

If we choose \( c \geq 2 \), then clearly the proposition is proved.

• This gives us an upper bound of \( 2n \log n \). To get the lower bound:

1. Choose \( c = 2 - \varepsilon \).

2. Restate the proposition with \( A(n) > c \ n \log n \)

3. Shift the boundaries on the integral to make it go the other way. (This inequality says that the integral of an increasing function is bounded below by rectangles whose height is the left point of intersection with the function. The integral is also bounded from above by rectangles intersecting at the right point.)

• So we’re done. \( \square \)
<table>
<thead>
<tr>
<th>Type</th>
<th>comparisons</th>
<th>comparisons / $n \log n$</th>
</tr>
</thead>
<tbody>
<tr>
<td>bubble</td>
<td>49995000.0</td>
<td>542.814</td>
</tr>
<tr>
<td>insertion</td>
<td>25083810.1</td>
<td>272.344</td>
</tr>
<tr>
<td>quicksort (original)</td>
<td>155983.1</td>
<td>1.694</td>
</tr>
<tr>
<td>quicksort (baase)</td>
<td>153569.2</td>
<td>1.667</td>
</tr>
<tr>
<td>quicksort (median)</td>
<td>134376.2</td>
<td>1.459</td>
</tr>
<tr>
<td>quicksort (median actual)</td>
<td>149383.9</td>
<td>1.622</td>
</tr>
</tbody>
</table>
4.3 Merging Arrays

- If we have two sorted lists each of size $n/2$, we can merge them with no more than $n - 1$ comparisons.
- Requires additional space (i.e., can’t be done in place)
- Do the obvious

keep pointers $\text{ptr\_list\_1}$, $\text{ptr\_list\_2}$, $\text{ptr\_merged\_list}$

while (not done)
  if $\text{list\_1[ptr\_list\_1]} < \text{list\_2[ptr\_list\_2]}$
    $\text{merged\_list[ptr\_merged\_list]} < \text{list\_1[ptr\_list\_1]}$
    $\text{ptr\_list\_1}++$
    $\text{ptr\_merged\_list}++$
  else
    $\text{merged\_list[ptr\_merged\_list]} < \text{list\_2[ptr\_list\_2]}$
    $\text{ptr\_list\_2}++$
    $\text{ptr\_merged\_list}++$
  endif
endwhile

when we run out of one list, copy the other list to $\text{merged\_list}$

keep three pointers

- Worst case: we never run out of either list because the two lists exactly interleave

**Theorem 4.6.** Any algorithm that merges by comparison of keys two sorted arrays of $n/2$ elements into one sorted array of $n$ elements must do at least $n - 1$ comparisons in the worst case.
Proof. Assume arrays $A = [a_i]$ and $B = [b_i]$. We claim that if we have

$$a_1 < b_1 < a_2 < b_2 < ... < a_i < b_i < a_{i+1} < b_{i+1} < ... < a_{n/2} < b_{n/2}$$

then we must do $n - 1$ comparisons. The claim is that we must compare $a_i$ with $b_i$ for $1 \leq i \leq n/2$ and that we must compare $b_i$ with $a_{i+1}$ for $1 \leq i \leq n/2 - 1$, for a total of $n - 1$ comparisons. If for some reason the algorithm is written so as not to compare $a_i$ with $b_i$ for some $i$, then the algorithm cannot correctly sort two arrays in which these elements are reversed in magnitude.

If for some reason the algorithm is written so as not to compare $b_i$ with $a_{i+1}$ for some $i$, then the algorithm cannot correctly sort two arrays in which these elements are reversed in magnitude.

But reversing either pair above does nothing to change the sorted order of the final array except for the order of the pairs in question. \qed
4.4 Mergesort

Theorem 4.7. Mergesort has worst case running time of $\Theta(n \log n)$.

Proof. Clearly

\[ W(n) = 2 \cdot W(n/2) + n - 1 \]
\[ = 4 \cdot W(n/4) + n - 1 + 2(n/2 - 1) \]
\[ = 4 \cdot W(n/4) + 2n - 1 - 2 \]
\[ \vdots \]
\[ = n \cdot W(1) + (\log n)n - (n - 1) \]
\[ = \Theta(n \log n) \]
4.5 Lower Bounds in General

- Consider an algorithm to sort by comparison of keys as a decision tree, as in Figure 4.15.
- There are $n!$ total permutations of $n$ elements, so a decision tree for sorting $n$ elements must have at least $n!$ leaves.
- For a binary tree of $k$ leaves and height $h$ we have $k \leq 2^h$, that is, $h \geq \lg k$.
- So a decision tree, with $n!$ leaves, must have height at least $\lceil \lg(n!) \rceil$
- And the lower bound for number of comparisons is the lower bound for the height of the decision tree.
- So the lower bound for sorting by comparing keys is $\lceil \lg(n!) \rceil$
- Now,

$$n! \geq n(n-1)\ldots([n/2])$$

so

$$\lg(n!) \geq \frac{n}{2} \cdot \lg \frac{n}{2}$$

- Note that

$$\lg(n!) \geq n \lg n - \lg e \cdot n \approx n \lg n - 1.443n$$
4.6 Lower Bounds for the Average Case

- The worst case requires looking for the maximal path down a decision tree.
- The average case requires looking at the average path length.
- Consider the average case, that is,

\[
\frac{\text{sum of all path lengths to leaves}}{n!}
\]

**Proposition 4.8.** The sum of all path lengths to leaves is minimized when the tree is completely balanced.

*Proof.* Disconnecting any subtree from a balanced binary tree, and moving that subtree so as to connect it somewhere else in a binary tree, necessarily increases the sum of the path lengths. \(\square\)

**Theorem 4.9.** The average case of any sort by comparison of keys is at least

\[
\lg(n!)
\]

*Proof.* The minimal sum of all path lengths to trees is

\[
(n!) \cdot (\lg(n!))
\]

so the average path length is bounded below by \(\lg(n!)\) \(\square\)
4.7 Heapsort

- Heapsort in theory is a better sort:
  1. worst case is \( n \log n \),
  2. average case is \( n \log n \),
  3. and it doesn’t require extra space.

- On the other hand:
  1. it’s a fixed cost algorithm—there’s no benefit to having the list sorted to start with,
  2. the constant is worse than other constants,
  3. and it doesn’t cache well.

**Definition 4.10.** A binary tree \( T \) is a **heap** if and only if

1. \( T \) is complete at least through depth \( h - 1 \);
2. all leaves are at depth \( h \) or depth \( h - 1 \);
3. all paths to a leaf at depth \( h \) are to the left of any path to a leaf at depth \( h - 1 \);
4. the key value at any node is greater than or equal to the key value of either of its children, if any children exist.

- Note that Baase carefully defines a **heap structure** as a tree with properties 1-3, a **partial order tree** to be a tree with property 4, and then fails to define a heap itself. Say what?
Examples, page 183, and elsewhere.

Heapsort metaphysics: If we have a heap, then

1. The largest element is the root of the tree.

2. We can exchange the root with the “last” element

3. and then recreate the heap in \( \log n \) steps by pushing the used-to-be-last element into its proper place,

4. resulting in a heap with one element less than before.

5. So if we can create a heap in the first place in \( n \log n \) steps

6. then we can put the elements into sorted order in another \( n \log n \) steps.

7. Further, this would be guaranteed, fixed-cost, running time.

Figure 4.18, page 185, describing the re-heaping process.

Algorithm 4.6, page 186

**Lemma 4.11.** At most \( 2h \) comparisons are needed to recreate the heap after exchange of the root with the currently-last element.

**Proof.** Worst case: We compare the two children of the “current” node to find the larger, and then we compare the larger with the “current” node to get the largest of all three.
4.7.1 Creating the heap

```c
void constructheap(heap)
    if(heap != leaf)
        constructheap(left subtree of heap)
        constructheap(right subtree of heap)
        key = root(heap)
        fixheap(heap,key)
    endif
```

**Theorem 4.12.** Procedure `constructheap(*)` creates a structure with the partial order property (4).
4.7.2 Constructing a Heap: Worst case analysis

- Note that since the two halves of the recursive breakdown of the problem are not necessarily equal in size, a precise analysis could be painful.
- However, the work for a problem of size $n$ will be bounded below and above by problems of size $2^{h-1}$ and $2^h$, respectively, where

$$2^{h-1} \leq n < 2^h$$

- So we deal with complete binary trees, with $N = 2^h - 1$ total nodes.
- Then

$$W(N) = 2W((N - 1)/2) + 2 \log N$$

- Apply the Master Theorem
  - $b = 2$, $c = 2$, $E = 1$, $f(N) = 2 \log N$
  - Choose $\varepsilon$ such that $0 < \varepsilon < 1$, and we get that $W(N) = \Theta(N)$. 
4.7.3 Heapsort: Worst case analysis

• The description of Heapsort on pages 188-191 seems to me to be especially confused...

• But we can at least describe the storage of the array to be sorted.

• If we have a complete binary tree, and we number the nodes top to bottom, left to right, from 1 to \(n - 1\) we find that the two nodes below any particular node \(i\) are those labelled \(2i\) and \(2i + 1\). So we can use this numbering in order to create and sort a heap using no extra storage.

• The array will be a heap under this ordering scheme if \(\forall i, a[i] \geq a[2i]\) and \(a[i] \geq a[2i + 1]\)

• Now, in each step, we remove the largest element, shorten the array by 1, and then fix up the heap. Fixing up a heap of \(k\) nodes takes at most \(2\lceil \lg k \rceil\) comparisons.
• So the total number of comparisons is the $\Theta(n)$ to construct the heap originally plus

$$2 \sum_{k=1}^{n-1} \lfloor \log k \rfloor < 2 \sum_{k=1}^{n-1} \log k$$

$$= 2 \sum_{k=1}^{n-1} \log e \log k$$

$$\leq 2 \int_1^n \log e \log x \, dx$$

$$= 2(\log e)(n \log n - n)$$

• So the running time is asymptotically $n \log n$ worst case.

• Actually, it’s $\Theta(n \log n)$ average case also.
4.7.4 Accelerated Heapsort