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 (or something else?)
  - worst case versus average case

- Practical issues
  - space (in-place or out-of-place?)
  - recursion versus iteration
  - external sort versus internal sort
  - sort keys, not data
  - number of moves
  - memory to memory? disk to disk?
  - locality of reference?
4.1 Bubble Sort

// sort the array ‘a[]’ of length ‘count’
// perform only the first ‘howmany’ sort steps
// keep track of the total number of comparisons
// note that this pulls the smallest element to the i-th position
// in the list with each iteration
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;
            }
        }
    }
}
4.2 Insertion Sort

// sort the array ‘a[*]’ of length ‘count’
// perform only the first ‘howmany’ sort steps
// keep track of the total number of comparisons
// note that this inserts the last element into the sorted list
// to create a sorted initial portion of the list that gets
// longer by one with each iteration
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;
        } // end of for (j = length - 1; j >= 1; j--)
    } // end of for (length = 2; length <= howmany; length++)
}
4.3 Insertion Sort (version 2)

// sort the array ‘a[*]’ of length ‘count’
// perform only the first ‘howmany’ sort steps
// keep track of the total number of comparisons
// this differs from the previous version only in that
// we don’t store back with every step; we only push
// the list down and create the vacancy into which we
// will eventually store the element to be inserted

void insert2(long a[], long howmany, long count, long *comparisons)
{
    long j, length, insertvalue;
    if(howmany == count) howmany--;
    for(length = 2; length <= howmany; length++)
    {
        insertvalue = a[length];
        j = length-1;
        while(a[j] > insertvalue)
        {
            comparisons++;
            a[j+1] = a[j];
            j--;
        }
        a[j+1] = insertvalue;
    }
}
4.4 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

\[ 4, 3, 5, 2, 1 \]

has inversions

\[ (4, 3), (4, 2), (4, 1), (3, 2), (3, 1), (5, 2), (5, 1), (2, 1). \]

*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. **Worst case:** 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.)
4.5 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

// merge two lists
// we keep three pointers
// pointer ptr_list_1 points to the next element in list 1
// pointer ptr_list_2 points to the next element in list 2
// pointer ptr_merged_list points to the next element in the merged list
// algorithm:
//   walk down each list with the pointers
//   compare the elements pointed to in lists 1 and 2
//   copy the smaller (from list i) to the merged list
//   and then bump the list i pointer and the merged list pointer
while(not done)
  if list_1[ptr_list_1] < list_2[ptr_list_2]
    merged_list[ptr_merged_list] = list_1[ptr_list_1]
    ptr_list_1++
    ptr_merged_list++
  else
    merged_list[ptr_merged_list] = list_2[ptr_list_2]
    ptr_list_2++
    ptr_merged_list++
endif
endwhile
when we run out of one list, copy the other list to merged_list

Worst case: we never run out of either list because the two lists exactly interleave
Theorem 4.4. 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

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

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
4.6 Mergesort

The mergesort algorithm: To sort \( n \) items:

1. while \( n \geq 2 \), split the list into two lists of \( n/2 \) items

2. sort those two lists by mergesort

3. and then merge the two lists

**Theorem 4.5.** Mergesort has worst case running time of \( \Theta(n \log n) \).

**Proof.** Clearly

\[
T(n) = 2 \cdot T(n/2) + n - 1
\]

\[
= 4 \cdot T(n/4) + n - 1 + 2(n/2 - 1)
\]

\[
= 4 \cdot T(n/4) + 2n - 1 - 2
\]

\[
= 8 \cdot T(n/8) + 2n - 3 + 4(n/4 - 1)
\]

\[
= 8 \cdot T(n/8) + 3n - 7
\]

\[...
\]

\[= n \cdot T(1) + (\log n)n - (n - 1)
\]

\[= \Theta(n \log n)
\]
4.7 Heapsort

Heapsort in theory is an outstanding sort:

1. worst case is $n \lg n$,

2. average case is $n \lg n$,

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 4.6.** A binary tree $T$ is a **max-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. if a leaf at depth $h$ is missing, then all leaves to its right are also missing;

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.

This last property (4) is called the **max-heap property**.
4.7.1 Example

A heap as a binary tree.

```
    72
   / \   \\
37   52
/ \   / \   \\
16  19  25  38
  / \  / \  / \  / \  \\
 9   7  11  23  3   8   32  22  33
```

4.7.2 Example

IMPORTANT!!!!! A heap as a linear array.

```
| 72 | 37 | 52 | 16 | 25 | 19 | 38 | 9 | 11 | 7 | 23 | 3 | 8 | 22 | 33 |
```

Or, if the binary tree is not a complete tree, something like

```
| 72 | 37 | 52 | 16 | 25 | 19 | 38 | 9 | 11 | 7 | 23 | 3 | 8 | 22 | 33 | * | * | * |
```

Another use for a heap is in a priority queue. If the only need is for the highest priority element in a list (simulation, scheduling of “the next event,” etc.), then a max or min heap (depending on taste) permits insertion into a list of \( n \) elements in \( \lg n \) time, removal of the next element in constant time, and reconstruction of the heap data structure as a heap in \( \lg n \) time. (Contrast this with the linear time of a linked list, for example.)
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.

We will use a heap and several functions that manipulate it:

- **Build-Max-Heap** creates a max-heap from an array in linear time.

- **Max-Heapify** maintains a max-heap in $O(\lg n)$ time for each call.

- **Heapsort** sorts $n$ items in place in $O(n \lg n)$ time using a heap.

- **Max-Heap-Insert**, **Heap-Extract-Max**, **Heap-Increase-Key**, and **Heap-Maximum** allow a heap to be used as a priority queue.
4.7.3 Max-Heapify

// assume that subtrees below node i are heaps
// push a[i] into its proper place to restore the full heap
void max-heapify(a[],i)
{
    lll = left(i);
    rrr = right(i);
    // three way test to get the subscript of the largest of three things
    if((lll <= heap-size(a)) && (a[lll] > a[i]))
    {
        largest = lll;
    }
    else
    {
        largest = i;
    }
    if((rrr <= heap-size(a)) && (a[rrr] > a[largest]))
    {
        largest = rrr;
    }
    // if a child value is the largest, then exchange and rebuild
    if(largest != i)
    {
        exchange a[i] and a[largest]
        max-heapify(a[],largest)
    }
}

This runs in worst-case time equal to the max depth of the tree.

The worst case in terms of height is for the left subtree and right subtrees
to have the same number of nodes and be balanced except for the left subtree
to have one extra node hanging off the left edge.
The worst case here is $2^k$ nodes and height $k$ instead of height $k-1$ if all were balanced.

Looked at differently, in terms of number of nodes, the maximum imbalance in left and right subtree is the following:

With height $k$, we have $2^{k-1}-1$ nodes in the right subtree and $2^{k-1}-1+2^{k-1} = 2^k - 1$ in the left subtree. This makes for $n = 3 \cdot 2^{k-1} - 1$ total nodes, of which $O(2n/3)$ are in the left subtree.

Either way, max-heapify runs in time $O(\lg n)$ comparisons.
4.7.4 Build-Max-Heap

// build a heap of an array a
void build-max-heap(a[])
{
    heap-size(a) = length(a);
    for(i = floor(length(a)/2); i >= 1; i--)
    {
        max-heapify(a,i);
    }
}

Loop Invariant: At the start of the $i$-th iteration of the for loop, every node for subscript larger than $i$ is the root of a max-heap.

Initialization: At the start of the first iteration, we have $i = floor(n/2)$. Every node with a larger subscript is a leaf, and thus every such node is the root of a trivial heap of one element.

Maintenance: The children of node $i$ are numbered higher than $i$, since we are iterating down. Therefore, by the invariant property, the children are both roots of heaps. This means that when we call max-heapify, its conditions are satisfied. Max-heapify preserves the heap-osity of children, so when we are finished with iteration $i$ we have created a heap with root at the $i$-th node.

Termination: At the end of the iteration, we have $i = 0$. We have maintained the loop invariant throughout the iteration, and hence node 1 and all other nodes are the root of a heap.

Timing: The build-max-heap procedure requires (worst case) $O(n \lg n)$ comparisons, because it calls the $O(\lg n)$ max-heapify procedure $n/2$ times.
4.7.5 Heapsort

// sort n items in array a
// algorithm:
//   build a max heap of the array
//   if we have a max heap, then the max is the first element,
//   so exchange the first and last elements in the array,
//   then decrease the heap length by one (so as not to touch the
//   max element that we just put at the end),
//   rebuild the heap to put the new a(1) in its proper place,
//   and repeat
void heapsort(a[])
{
    build-max-heap(a);
    for(i = length(a); i >= 2; i--)
    {
        exchange a[1] and a[i];
        heapsize(a)--;
        max-heapify(a,1);
    }
}

Timing: The algorithm takes $O(n \lg n)$ time, since it calls max-heapify $n - 1$ times at cost $O(\lg n)$ each time, and this cost dominates the linear time cost of build-max-heap.
4.8 Priority Queues

There are lots of applications in which one needs repeatedly to pull off the “next” item from a list. This requires a priority queue, a data structure for maintaining a set $S$ of elements, each with an associated key value, so that they can be accessed in the sorted order of their key value.

- Consider simulation (or the real thing) of events. New events are created dynamically and scheduled to be executed at some future time. The simulation process itself wants to pull off the event list only those items scheduled for the next soonest execution time.

- A similar situation exists in scheduling events in an operating system.

- Data clustering algorithms often want to cluster data nodes based on the nearest pairwise distance, recompute distances, and then cluster again,...

We need functions

- **Insert** a new element $x$ into an existing set $S$.

- **Maximum** to return the element with the max key value.

- **Extract-max** to pull the element with the max key value off the list.

- **Increase-key** to raise the priority (scheduled execution time) of the element from some existing key value to some larger key value.
4.8.1 Heap-maximum

// return the largest key value in the set
def keyvalue heap-maximum(a)
{
    return(a[1]);
}

Clearly constant time

4.8.2 Heap-extract-max

// extract the largest key value in the set,
// move the last value to the first location
// decrease the length by 1
// and then rebuild the heap to maintain the data
// structure as a priority queue
def keyvalue heap-extract-max(a)
{
    if(heap-size(a) < 1)
    {
        UNDERFLOW ERROR
    }

    maxvalue = a[1];
a[1] = a[heap-size(a)];
heap-size(a) = heap-size(a) - 1;
max-heapify(a,1);

    return(maxvalue);
}

Clearly O(lg n) time on an array of length n
4.8.3 Heap-increase-key

// increase the value of a specific element, and
// maintain the priority queue
void heap-increase-key(a,i,key)
{
    if(key < a[i])
    {
        ERROR, new key isn’t an increase
    }
    a[i] = key;
    while((i > 1) && (a[parent(i)] < a[i]))
    {
        exchange a[i] and parent(a[i]);
        i = parent(i);
    }
}

Clearly $O(\lg n)$ time on an array of length $n$

Note the importance of random access into the array, and compare this with a linked list.

4.8.4 Max-heap-insert

// insert a new element into the queue
// maintaining the priority queue
void max-heap-insert(a,key)
{
    heap-size(a)++;
    a[heap-size(a)] = -infinity;
    heap-increase-key(a,heapsize(a),key);
}

Clearly $O(\lg n)$ time on an array of length $n$
4.9 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.

### 4.9.1 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.
4.9.2 Average Case

Theorem 4.7. Quicksort has average case running time, for large \( n \), of approximately \( 2 \ n \log n \) 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 4.8.** For \( n \geq 1 \), we have \( A(n) \leq cn \log n \) for some constant \( c \).

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

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

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.)
Specifically: Let $c = 2 - \varepsilon$. Then

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

$$> n - 1 + \frac{2c}{n} \sum_{i=1}^{n-1} i \log i$$

$$= n - 1 + \frac{2c}{n} \sum_{i=2}^{n-1} i \log i$$

$$\geq n - 1 + \frac{2c}{n} \int_{1}^{n-1} x \log x \, dx$$

$$= n - 1 + \frac{2c}{n} \left[ \frac{x^2 \log x}{2} - \frac{x^2}{4} \right]_{1}^{n-1}$$

$$= n - 1 + \frac{2c}{n} \left( \frac{(n-1)^2 \log(n-1)}{2} - \frac{(n-1)^2}{4} + \frac{1}{4} \right)$$

$$= n - 1 + \frac{c}{n} \left( (n-1)^2 \log(n-1) - \frac{(n-1)^2}{2} + \frac{1}{2} \right)$$

We want this last value to be larger than $cn \ln n$. That is, we want to have

$$n - 1 + \frac{c}{n} \left( (n-1)^2 \log(n-1) - \frac{(n-1)^2}{2} + \frac{1}{2} \right) > cn \ln n$$

$$2n^2 - 2n + c \left( 2(n-1)^2 \log(n-1) - (n-1)^2 + 1 \right) > 2cn^2 \ln n$$

When we substitute $c = 2 - \varepsilon$ and then rearrange terms, we get

$$\varepsilon \left( (n-1)^2 + 2n^2 \ln n - 2(n-1)^2 \ln(n-1) - 1 \right) > 4n^2 \ln n - 4(n-1)^2 \ln(n-1) - 2n$$

Now, what we will show is that the

$$n^2 \ln n - (n-1)^2 \ln(n-1)$$
term is actually of smaller order of magnitude than \( n^2 \), which will show that the \( \varepsilon(n-1)^2 \) term dominates and the inequality holds.

So let’s look at

\[
(n-1)^2 \ln(n-1) - n^2 \ln n = n^2 \ln(n-1) - n^2 \ln n - (2n+1) \ln(n-1) \\
= n^2 \ln \left( \frac{n-1}{n} \right) - (2n+1) \ln(n-1) \\
= n \ln \left( 1 - \frac{1}{n} \right)^n - (2n+1) \ln(n-1)
\]

Now, as \( n \) goes to infinity, \( (1 - 1/n)^n \) goes to \( e^{-1} \), and therefore its natural logarithm goes to \(-1\). The first term on the right hand side therefore looks like \( n \) times a constant for large \( n \), and the second term looks like \( n \ln n \).

What looks like an \( n^2 \ln n \) term that would dominate the merely \( \varepsilon n^2 \) on the left hand side isn’t in fact that big, and the necessary expression above in \( \varepsilon \) is true for \( n \) sufficiently large.

So we’re done. \( \square \)

I wrote programs for several different simple sorts and ran them on 10,000 (allegedly) random integer data items. The following was what I got.

<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>
4.9.3 A Somewhat Counterintuitive Theorem

**Theorem 4.9.** If the partitioning always produces two sublists whose sizes have a constant ratio \( r < 1 \), then the worst case running time is still \( O(n \lg n) \).

**Proof.** We have the recursion

\[
T(n) = [T((1 - r)n)] + [T(rn)] + cn
\]

\[
= [T((1 - r)^2n) + T((1 - r)rn) + c(1 - r)n] + [T((1 - r)rn) + T(r^2n) + crn] + cn
\]

Note that the cost at each level is always \( cn \). We will have \( \log_{1-r} n \) or \( \log_r n \) levels, and hence the cost is \( n \lg n \) (number of levels of recursion times cost per level).

This is one reason that quicksort is so popular. You don’t have to be all that great in picking a partition.