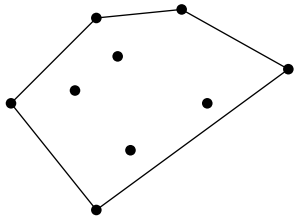
# *Introduction to Algorithms*

# *Part 1: Divide and Conquer Sorting and Searching*

# *Chapter 1: Convex Hulls: An Example*

A polygon is ***convex*** if any line segment joining two points on the boundary stays within the polygon. Equivalently, if you walk around the boundary of the polygon in counterclockwise direction you always take left turns.

The ***convex hull*** of a set of points in the plane is the smallest convex polygon for which each point is either on the boundary or in the interior of the polygon. One might think of the points as being nails sticking out of a wooden board: then the convex hull is the shape formed by a tight rubber band that surrounds all the nails. A ***vertex*** is a corner of a polygon. For example, the highest, lowest, leftmost and rightmost points are all vertices of the convex hull. Some other characterizations are given in the exercises.



We discuss three algorithms: Graham Scan, Jarvis March and Divide & Conquer. We present the algorithms under the ***assumption*** that:

- *no 3 points are collinear (on a straight line)*

## 1.1 Graham Scan

The idea is to identify one vertex of the convex hull and sort the other points as viewed from that vertex. Then the points are scanned in order.

Let $x_0$ be the leftmost point (which is guaranteed to be in the convex hull) and number the remaining points by angle from $x_0$ going counterclockwise: $x_1, x_2, \ldots, x_{n-1}$. Let $x_n = x_0$, the chosen point. Assume that no two points have the same angle from $x_0$.

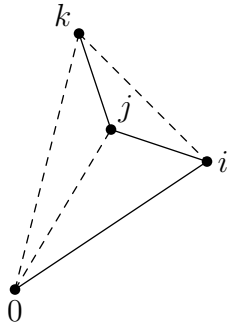
The algorithm is simple to state with a single stack:

**Graham Scan**

1. Sort points by angle from $x_0$
2. Push $x_0$ and $x_1$. Set $i$=2
3. While $i \leq n$ do:
   If $x_i$ makes left turn w.r.t. top 2 items on stack
   then { push $x_i$; $i$++ }
   else { pop and discard }

To prove that the algorithm works, it suffices to argue that:

- *A discarded point is not in the convex hull.* If $x_j$ is discarded, then for some $i < j < k$ the points $x_i \to x_j \to x_k$ form a right turn. So, $x_j$ is inside the triangle $x_0, x_i, x_k$ and hence is not on the convex hull.




- *What remains is convex.* This is immediate as every turn is a left turn.

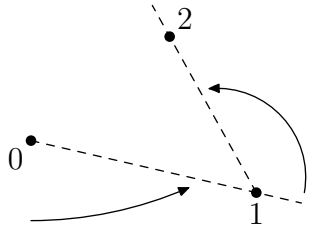
The running time: Each time the while loop is executed, a point is either stacked or discarded. Since a point is looked at only once, the loop is executed at most $2n$ times. There is a constant-time subroutine for checking, given three points in order, whether the angle is a left or a right turn (Exercise). This gives an $O(n)$ time algorithm, apart from the initial sort which takes time $O(n \log n)$. (Recall that the notation $O(f(n))$, pronounced "order $f(n)$", means "asymptotically at most a constant times $f(n)$".)

## 1.2 Jarvis March

This is also called the ***wrapping algorithm***. This algorithm finds the points on the convex hull ***in the order*** in which they appear. It is quick if there are only a few points on the convex hull, but slow if there are many.

Let $x_0$ be the leftmost point. Let $x_1$ be the first point counterclockwise when viewed from $x_0$. Then $x_2$ is the first point counterclockwise when viewed from $x_1$, and so on.

**Jarvis March**
$i = 0$
while not done do
    $x_{i+1}$ = first point counterclockwise from $x_i$




Finding $x_{i+1}$ takes linear time. The while loop is executed at most $n$ times. More specifically, the while loop is executed $h$ times where $h$ is the number of vertices on the convex hull. So Jarvis March takes time $O(nh)$.

The best case is $h = 3$. The worst case is $h = n$, when the points are, for example, arranged on the circumference of a circle.

## 1.3 Divide and Conquer

Divide and Conquer is a popular technique for algorithm design. We use it here to find the convex hull. The first step is a Divide step, the second step is a Conquer step, and the third step is a Combine step.

The idea is to:

**Divide and conquer**

1. Divide the $n$ points into two halves.
2. Find convex hull of each subset.
3. Combine the two hulls into overall convex hull.

Part 2 is simply two ***recursive*** calls. The first point to notice is that, if a point is in the overall convex hull, then it is in the convex hull of any subset of points that contain it. (Use characterization in exercise.) So the task is: given two convex hulls find the convex hull of their union.

### ◇ *Combining two hulls*

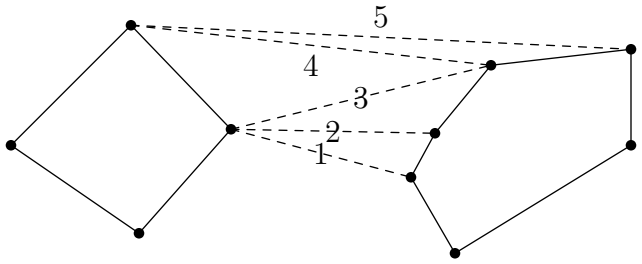
It helps to work with convex hulls that do not overlap. To ensure this, all the points are ***presorted*** from left to right. So we have a left and right half and hence a left and right convex hull.

Define a ***bridge*** as any line segment joining a vertex on the left and a vertex on the right that does not cross the side of either polygon. What we need are the ***upper*** and ***lower*** bridges. The following produces the upper bridge.

1. Start with any bridge. For example, a bridge is guaranteed if you join the rightmost vertex on the left to the leftmost vertex on the right.

2. Keeping the left end of the bridge fixed, see if the right end can be raised. That is, look at the next vertex on the right polygon going clockwise, and see whether that would be a (better) bridge. Otherwise, see if the left end can be raised while the right end remains fixed.

3. If made no progress in (2) (cannot raise either side), then stop else repeat (2).



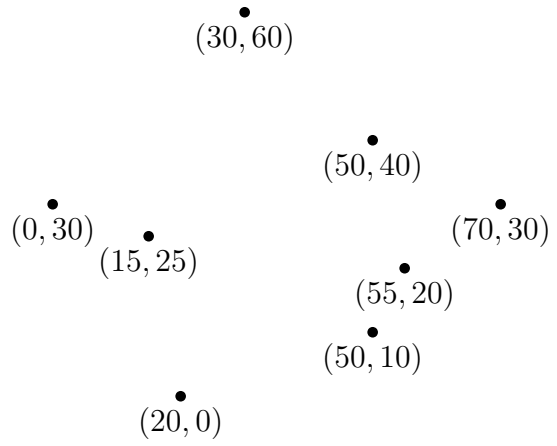

We need to be sure that one will eventually stop. Is this obvious?

Now, we need to determine the running time of the algorithm. The key is to perform step (2) in constant time. For this it is sufficient that each vertex has a pointer to the next vertex going clockwise and going counterclockwise. Hence the choice of data structure: we store each hull using a ***doubly linked circular linked list***.

It follows that the total work done in a merge is proportional to the number of vertices. And as we shall see in the next chapter, this means that the overall algorithm takes time $O(n \log n)$.

## Exercises

1. Find the convex hulls for the following list of points using the three algorithms presented.



2. Give a quick calculation which tells one whether three points make a left or a right turn.

3. Discuss how one might deal with collinear points in the algorithms.

4. Show that a point $D$ is on the convex hull if and only if there do not exist points $A, B, C$ such that $D$ is inside the triangle formed by $A, B, C$.

5. ⓔ Assume one has a fast convex hull subroutine that returns the convex hull ***in order***. Show how to use the subroutine to sort.

# *Chapter 2: Divide and Conquer*

In this chapter we consider divide and conquer: this is essentially a special type of recursion. In divide and conquer, one:

> *divides the problem into pieces,*
> *then conquers the pieces,*
> *and re-assembles.*

An example of this approach is the convex hull algorithm. We divide the problem into two pieces (left and right), conquer each piece (by finding their hulls), and re-assemble (using an efficient merge procedure).

## 2.1 Master Theorem for Recurrences

One useful result is the "Master Theorem" for a certain family of recurrence relations:

Consider the recurrence

$$T(n) = aT(n/b) + f(n).$$

Then

- if $f(n) \ll n^{\log_b a}$ then $T(n) = O(n^{\log_b a})$.
- if $f(n) \approx n^{\log_b a}$ then $T(n) = O(n^{\log_b a} \log n)$.
- if $f(n) \gg n^{\log_b a}$ and $\lim_{n\to\infty} af(n/b)/f(n) < 1$ then $T(n) = O(f(n))$.

Example. If $T(n)$ denotes the time taken for the divide-and-conquer convex hull algorithm (ignoring the initial sort), then we obtain the recurrence

$$T(n) = 2T(n/2) + O(n).$$

This solves to $O(n \log n)$.

## 2.2 Multiplying Long Integers

Another example of divide-and-conquer is the problem of multiplying long integers. The following is based on the discussion by Aho, Hopcroft and Ullman.

Consider the problem of multiplying two $n$-bit integers $X$ and $Y$. At school we learnt long multiplication: if we take two $n$-digit numbers then long multiplication takes $O(n^2)$ (quadratic) time. (Why?)

We can apply divide and conquer. For example, suppose we split $X$ and $Y$ into two halves:

$$X := \boxed{A}\boxed{B} \qquad Y := \boxed{C}\boxed{D}$$

Then $X = A2^{n/2} + B$ and $Y = C2^{n/2} + D$. The product of $X$ and $Y$ can now be written as:

$$XY = AC2^n + (AD + BC)2^{n/2} + BD$$

This product requires four multiplications and three additions. To add two integers takes linear time. (Why?) So we obtain the recurrence:

$$T(n) = 4T(n/2) + O(n)$$

Alas, the solution to this is again quadratic, and in practice this algorithm is worse than normal long multiplication.

However, consider the following formula:

$$XY = AC\,2^n + \Big[(A - B)(D - C) + AC + BD\Big]\,2^{n/2} + BD$$

At first glance this looks more complicated. But we only need three multiplications to do this: $AC$, $BD$ and $(A - B)(D - C)$. Therefore we obtain the recurrence:

$$T(n) = 3T(n/2) + cn$$

whose solution is $T(n) = O(n^{\log_2 3}) = O(n^{1.59})$.

There is some more work needed to write this down in a program. We need to consider shifts and also deal with negative integers. Left as an exercise.

## 2.3 Matrix Multiplication

Consider the problem of multiplying two $n \times n$ matrices. Computing each entry in the product takes $n$ multiplications and there are $n^2$ entries for a total of $O(n^3)$ work. Can one do better?

Strassen devised a better method which has the same basic flavor as the multiplication of long integers. The key idea is to save one multiplication on a small problem and then use recursion.

We look first at multiplying two $2 \times 2$ matrices. This would normally take 8 multiplications, but it turns out we can do it with 7. You can check the following. If

$$A = \begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix} \qquad \text{and} \qquad B = \begin{pmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{pmatrix}$$

and one calculates the following products:

$$\begin{aligned}
m_1 &= (a_{21} + a_{22} - a_{11})(b_{22} - b_{12} + b_{11}) \\
m_2 &= a_{11}b_{11} \\
m_3 &= a_{12}b_{21} \\
m_4 &= (a_{11} - a_{21})(b_{22} - b_{12}) \\
m_5 &= (a_{21} + a_{22})(b_{12} - b_{11}) \\
m_6 &= (a_{12} - a_{21} + a_{11} - a_{22})b_{22} \\
m_7 &= a_{22}(b_{12} + b_{21} - b_{11} - b_{22})
\end{aligned}$$

then the product of $A$ and $B$ is given by

$$AB = \begin{pmatrix} m_2 + m_3 & m_1 + m_2 + m_5 + m_6 \\ m_1 + m_2 + m_4 + m_7 & m_1 + m_2 + m_4 + m_5 \end{pmatrix}$$

Wow! But what use is this?

What we do is to use this idea recursively. If we have a $2n \times 2n$ matrix, we can split it into four $n \times n$ submatrices:

$$A = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \qquad \text{and} \qquad B = \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix}$$

and form seven products $M_1$ up to $M_7$, and the overall product $AB$ can thus be calculated with seven multiplications of $n \times n$ matrices. Since adding matrices is clearly proportional to their size, we obtain a recurrence relation:

$$f(n) = 7f(n/2) + O(n^2)$$

whose solution is $O(n^{\log_2 7})$, an improvement on $O(n^3)$.

## 2.4 Modular Exponentiation

Another example of divide-and-conquer is the modular exponentiation problem. The following is based on the discussion by Brassard and Bratley.

Suppose Alice and Bob wish to establish a secret but do not initially share any common secret information. Their problem is that the only way they can communicate is by using a telephone, which is being tapped by Eve, a malevolent third party. They do not want Eve to know their secret. To simplify the problem, we assume that, although Eve can overhear conversations, she can neither add nor modify messages.

The goal is to find a ***protocol*** by which Alice and Bob can attain their ends.

The first solution to this problem was given in 1976 by Diffie and Hellman. The idea is the following.

1. Alice and Bob agree openly on some integer $p$ with a few hundred decimal digits, and on some other integer $g$ between 2 and $p - 1$. (The security of the secret they intend to establish is not compromised should Eve learn these two numbers.)

2. Alice and Bob choose randomly and independently of each other two positive integers $A$ and $B$ less than $p$.

3. Alice computes $a = g^A \bmod p$ (the mod function is the % of Java/C), and transmits this result to Bob. Similarly, Bob sends Alice the value $b = g^B \bmod p$.

4. Alice computes $x = b^A \bmod p$ and Bob calculates $y = a^B \bmod p$.

Now $x = y$ since both are equal to $g^{AB} \bmod p$. This value is therefore a piece of information shared by Alice and Bob. It can now be used as the key in a conventional cryptographic system.

At the end of the exchange, Eve has the values of $p$, $g$, $a$, and $b$ only. One way for her to deduce $x$ would be to find an integer $A'$ such that $a = g^{A'} \bmod p$, and then to proceed like Alice to calculate $x' = b^{A'} \bmod p$. Under some simple algebraic conditions, such $A'$ is necessarily equal to $A$, and the secret is correctly computed by Eve in this case.

Calculating $A'$ from $p$, $g$ and $a$ is called the problem of the ***discrete logarithm***. There exists an obvious algorithm to solve it. (If the logarithm does not exist, the algorithm returns the value $p$. For instance, there is no integer $A$ such that $3 = 2^A \bmod 7$.)

```
dlog (base g,answer a,modulus p)
    A ← 0; x ← 1
    repeat
        A ← A + 1
        x ← xg
    until (a = x mod p) or (A = p)
    return A
```

This algorithm takes an unacceptable amount of time, since it makes $p/2$ trips round the loop on the average. If each trip round the loop takes 1 microsecond, this average time is more than the age of Earth even if $p$ only has 24 decimal digits. Although there are more efficient algorithms for calculating discrete logarithms, none is able to solve a randomly chosen instance in a reasonable amount of time when $p$ is a prime with several hundred decimal digits. Furthermore, there is no known way of recovering $x$ from $p$, $g$, $a$, and $b$ that does not involve calculating a discrete logarithm. For the time being, it seems therefore that this method of providing Alice and Bob with a shared secret is sound, although no-one has yet been able to prove this.

But is this a joke? If Eve needs to be able to calculate discrete logarithms efficiently to discover the secret shared by Alice and Bob, it is equally true that Alice and Bob must be able to calculate efficiently exponentiations of the form $a = g^A \bmod p$. The obvious algorithm for this is no more subtle or efficient than the one for discrete logarithms.

The fact that

$$xyz \bmod p = ((xy \bmod p) \times z) \bmod p$$

for every $x$, $y$, $z$, and $p$ allows us to avoid accumulation of extremely large integers in the loop. (The same improvement can be made in dlog, which is necessary if we hope to execute each trip round the loop in 1 microsecond.)

Happily for Alice and Bob, there is a more efficient algorithm for computing the exponentiation. An example will reveal the basic idea.

$$x^{25} = (((x^2x)^2)^2)^2x$$

Thus $x^{25}$ can be obtained with just two multiplications and four squarings. This formula arises because $x^{25} = x^{24}x$, $x^{24} = (x^{12})^2$, and so on.

This idea can be generalized to obtain a divide-and-conquer algorithm.

```
dexpo (g,A,p)      % calculates a = g^A mod p
    if A = 0 then return 1
    if A odd then {
        a ← dexpo(g, A − 1, p)
        return(ag mod p)
        }
    else {
        a ← dexpo (g, A/2, p)
        return(a^2 mod p)
        }
```

This requires $O(\log g)$ multiplications.

## 2.5 Good Algorithms

Recursion is often easy to think of but does not always work well. Consider the problem of calculating the $n$th Fibonacci number, which is defined by

$$a_n = a_{n-1} + a_{n-2}$$

with initial values $a_1 = 1$ and $a_2 = 1$. Recursion is terrible! Rather do iteration: calculate $a_3$ then $a_4$ etc.

The key to good performance in divide and conquer is to partition the problem as evenly as possible, and to save something over the naïve implementation.

## Exercises

1. Illustrate the multiplication of 1234 and 5678 using the algorithm described above.
2. Create a class for long integers. Then provide the operations shift, addition and multiplication. Then calculate the square of the 50th Fibonacci number.

3. Use Strassen's algorithm to multiply

$$\begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix} \qquad \text{and} \qquad \begin{pmatrix} 5 & 6 \\ 7 & 8 \end{pmatrix}$$

4. Consider the recurrence:

$$f(n) = 2f(n/2) + cn \qquad f(1) = 0$$

Prove that this has the solution $f(n) = cn \log_2 n$ for $n$ a power of 2.

5. Calculate $7^{2004} \bmod 13$

6. Consider the following program:

```
function Fibonacci(n)
  if n<2 then return n
  else return Fibonacci(n-1) + Fibonacci(n-2)
```

Analyze the time used for this algorithm.

7. ⓔ a) Give a fast algorithm to computer $F^n$ where $F$ is a 2-by-2 matrix and $n$ a positive integer. It should run in time $O(\log n)$.

b) Consider the matrix

$$F = \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix}$$

Show that

$$F^n = \begin{pmatrix} a_{n-1} & a_n \\ a_n & a_{n+1} \end{pmatrix}$$

where $a_n$ denotes the $n$th Fibonacci number.

c) Hence give a fast algorithm to determine the $n$th Fibonacci number.

8. ⓔ Consider an $n$-bit binary counter. This is used to count from 0 up to $2^n - 1$. Each time the counter is incremented, the natural add-one algorithm is follows: one works in from the right-most bit changing bits until there is no carry. Determine the ***average*** number of bit operations needed per increment.

# *Chapter 3: Sorting Revisited*

In this chapter we revisit the sorts Quicksort and Merge Sort which you have probably seen before.

## 3.1 Quicksort

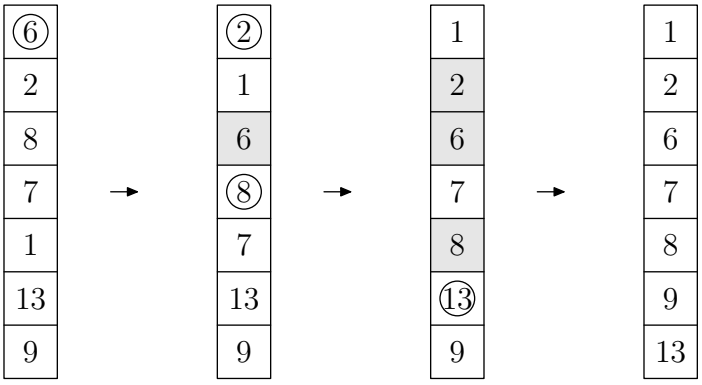
***Quicksort*** is a procedure for sorting that was invented by Hoare in 1962. It uses divide and conquer. Say one starts with $n$ distinct numbers in a list. Call the list $A$. Then Quicksort does:

**Quicksort (A:valuelist)**

1: Choose a Key (often just the first element in the list).

2: Split the list into two sublists $A_<$ and $A_>$ (called buckets). The bucket $A_<$ contains those elements smaller than the Key and the bucket $A_>$ contains those elements larger than the Key.

3: Use Quicksort to sort both buckets recursively.




There remain questions about:

1) Implementation
2) Speed
3) Storage required
4) Is this the best we can do?
5) Problems with the method.

The beauty of Quicksort lies in the storage requirement: the sorting takes place "in situ" and very little extra memory is required.

## 3.2 How Fast is Quicksort?

To analyze the speed, we focus on the number of comparisons between data items. We count only the comparisons. We will come back to whether this is valid or not. But even this counting is hard to do.

### ◇ *Worst case*

What is the worst case scenario? A very uneven split. For example, our Key might be the minimum value. Then we compare it with every element in the list only to find that the bucket $A_<$ is empty. Then when we sort $A_>$ we might again be unlucky and have the minimum value of that bucket as the Key. In fact, if the list was already sorted we would end up comparing every element with every other element for a total of $\binom{n}{2}$ comparisons.

We can analyze the ***worst case*** (when the list is in fact already sorted) another way: The first step breaks the list up into $A_<$ which is empty and $A_>$ which contains $n-1$ items. This takes $n-1$ comparisons. Then $A_>$ is split using $n-2$ comparisons, and leaves a basket of $n-2$ items. So number of comparisons is:

$$(n-1)+(n-2)+(n-3)+\ldots+2+1 = n(n-1)/2 \approx n^2/2.$$

### ◇ *Best case*

The best case is when the list is split evenly each time. (Why?) In this case the size of the largest bucket goes down by a factor of 2 each time.

At the top level we use $n-1$ comparisons and then have to sort the buckets $A_<$ and $A_>$ which have approximately $(n-1)/2$ elements each. To make the arithmetic simpler, let's say that we use $n$ comparisons and end up with two buckets of size $n/2$.

Let $f(n)$ denote the number of comparisons needed by Quicksort in the best case. We then have the recurrence relation:

$$f(n) = n + 2f(n/2)$$

with the boundary condition that $f(1) = 0$. One can then check that the solution to this, at least in the case that $n$ is a power of 2, is

$$f(n) = n \log_2 n$$

### ◇ *Average case*

Of course, we are actually interested in what happens in real life. Fortunately, the typical behavior of Quicksort is much more like $n \log_2 n$ than $n^2$. We do not explore this here—but see Exercise 6.

## 3.3 Binary Sort

An iterative method for sorting is also available. At each stage one inserts the new element in the already sorted list.

Finding the place for the new element is known as ***binary search***. Recall that one compares the new element with the middle element of the list. If the new element is smaller than the middle element, one recursively searches in the top half of the list. If the new element is larger than the middle element, one searches in the bottom half.

**BinarySort (A:valuelist)**

For each element of A in turn:
    Insert the new element into current sorted list using binary search.

A binary search takes $\log_2 m$ comparisons (rounded up) where $m$ is the number of elements in the list. In binary sort, at each stage one does a binary search to find the place where the new element goes. So each stage takes $\log_2 m$ comparisons where $m$ is the number of elements in the list. This provides a guarantee of at most $n \log_2 n$ comparisons.

But this sort is seldom used. Why?

## 3.4 Merge Sort

Another example of recursion is the following.

**MergeSort (A:valuelist)**

1: Arbitrarily split the list into two halves.

2: Use MergeSort to sort each half.

3: Merge the two sorted halves.

One divides the list into two pieces just by slicing in the middle. Then one sorts each piece using recursion. Finally one is left with two sorted lists. And must now combine them. The process of combining is known as ***merging***.

How quickly can one merge? Well, think of the two sorted lists as stacks of exam papers sitting on the desk with the worst grade on top of each pile. The worst grade in the entire

list is either the worst grade in the first pile or the worst grade in the second pile. So compare the two top elements and set the worst aside. The second-worst grade is now found by comparing the top grade on both piles and setting it aside. Etc.

Why does this work?

How long does merging take? Answer: One comparison for every element placed in the sorted pile. So, roughly $n$ comparisons where $n$ is the total number of elements in the combined list. (It could take less. When?)

Merge Sort therefore obeys the following recurrence relation

$$M(n) = n + 2M(n/2).$$

(Or rather the number of comparisons is like this.)

We've seen before that the solution to this equation is $n \log_2 n$. Thus Merge Sort is an $n \log n$ algorithm in the worst case.

What is the drawback of this method?

## 3.5 Optimality of Sorting

So we have sorting algorithms that take time proportional to $n \log_2 n$. Is this the best we can do? Yes, in some sense, as we will show later.

## Exercises

1. Illustrate the behavior of Quicksort, Binary Sort and Merge Sort on the following data: 2, 4, 19, 8, 9, 17, 5, 3, 7, 11, 13, 16

2. Discuss the advantages and disadvantages of the three sorts introduced here.

3. In your favorite programming language, code up the Quicksort algorithm. Test it on random lists of length 10, 100, 1000 and 10 000, and comment on the results.

4. Suppose we have a list of $n$ numbers. The list is guaranteed to have a number which appears more than $n/2$ times on it. Devise a good algorithm to find the Majority element.

5. ⓔ Repeat the previous question but now assume one can only test two numbers for equality or inequality.

6. ⓔ Let $q(n)$ be the average number of data-comparisons required for Quicksort on a randomly generated list.

   a) Explain why $q(1) = 0$, $q(2) = 1$ and $q(3) = 2\frac{2}{3}$

b) Explain why the following recurrence holds:

$$q(n) = n - 1 + \frac{2}{n} \sum_{i=0}^{n-1} q(i)$$

c) Show that the solution to the above recurrence is:

$$q(n) = 2(n+1) \sum_{k=1}^{n} \frac{1}{k} - 4n$$

d) Use a calculator, computer or mathematical analysis to look at the asymptotics of $q(n)$.

7. a) Suppose we have a collection of records with a 1-bit key $K$. Devise an efficient algorithm to separate the records with $K = 0$ from those with $K = 1$.

   b) What about a 2-bit key?

   c) What has all this to do with sorting? Discuss.

# *Chapter 4: Parallel Sorting*

We consider parallel algorithms where the processors have limited capabilities and where there is a definite topology: a processor can communicate with only a few processors. Also there is a global clock (SIMD). We consider the sorting problem where each processor maintains one data value, and there is a specific target order which must be established. At each step, a processor can do a bit of computation, and swap data value with one of its neighbors.

This chapter is based on the discussion by Leighton.

## 4.1 Odd-Even Transposition Sort on a Linear Array

A ***linear array*** has the processors in a row: each processor can communicate with only the one before it and the one after it. We give a linear array sorting algorithm that takes precisely $N$ steps, where $N$ is the number of processors and the number of data values. $N-1$ is a lower bound, because one might need to get a data value from the last cell to the first cell.

**Odd-even transposition sort.**
At odd steps, compare contents of cell 1 and 2, cells 3 and 4, etc, and swap values if necessary so that the smaller value ends up in the leftmost cell. At even steps do the same but for the pairs 2 and 3, 4 and 5, etc.

Example. 46278135.
46-27-81-35 → 4-62-71-83-5 → 42-61-73-85 → 2-41-63-75-8 → 21-43-65-78 → 1-23-45-67-8

To prove that this works, focus on the smallest element. It is clear that after a while the smallest element ends up in the leftmost cell. At that stage we can look at the second smallest element and so on. With a bit more work one can show that the whole process runs in $N$ steps.

## 4.2 Odd-Even Merge Sort

This sort is a famous parallel algorithm. It can be implemented on the topology called a hypercube, but we will consider only the PRAM version. PRAM is ***parallel random access machine***: in this model there is global memory accessible to all processors. In particular all the data is accessible to all processors (though each can only look at one piece in one clock-step). The algorithm runs in $O(\log^2 N)$ steps.

**Merge Sort.**

Given list of length $N$, split into $N$ lists of length 1. Then in parallel merge pairs of length-1 lists to form $N/2$ sorted lists of length 2. These lists are then merged into $N/4$ sorted lists of length 4, and so on. At the end, we merge two sorted lists of length $N/2$ into the final sorted list.

But the merge step we described in Merge Sort looks inherently sequential. The new idea is as follows:

**Odd-even merge.**

To merge two sorted lists $A$ and $B$, split $A$ into two sublists $A_{even}$ and $A_{odd}$ containing the even-numbered entries and the odd-numbered entries respectively; similarly split $B$ into $B_{even}$ and $B_{odd}$. Since $A$ and $B$ are sorted, the four sublists are sorted. Then use recursion to merge $A_{even}$ and $B_{odd}$ to form $C$, and merge $A_{odd}$ and $B_{even}$ to form $D$. Finally, merge $C$ and $D$.

At first glance, this appear ridiculous: we have made no progress. But it turns out that merging $C$ and $D$ is trivial. The fact is, the first two elements of the sorted list are the top elements in $C$ and $D$ in some order, the third and forth elements in the sorted list are the second elements of $C$ and $D$ in some order, and so on. (See below.)

EXAMPLE. To merge A=1467 and B=2358, C is merge of 47 and 25, D is merge of 16 and 38, so C is 2457 and D is 1368.

## 4.3 Proof and Analysis of Odd-Even Merge Sort

The proof is made simpler by considering only small and big elements. For any value $T$ (threshold), we define a data element to be $T$-***small*** if it less than $T$ and $T$-***big*** if it is $T$ or greater. Then:

> OBSERVATION. A list is sorted if and only if for all choices of $T$ all $T$-small elements come before all $T$-big elements.

Now, notice that $A_{even}$ and $A_{odd}$ have almost the same number of $T$-small elements, as do $B_{even}$ and $B_{odd}$. So $C$ and $D$ have almost the same number of $T$-small elements. Indeed, the pairing of $A_{even}$ with $B_{odd}$ ensures that if $A_{even}$ and $A_{odd}$ have different amounts of

small elements, and $B_{odd}$ and $B_{even}$ different amounts too, then $C$ and $D$ have the same number of smalls. Thus in the final list, all the smalls will occur before all the bigs.

It remains to determine how long this process takes. The final merge of $C$ and $D$ takes $O(1)$ time, and the two submerges occur in parallel. So the overall merge takes $O(\log N)$ steps. There are $O(\log N)$ merge-phases, and so the overall algorithm runs in time $O(\log^2 N)$. Provided there are $O(N)$ processors.

## Exercises

1. For the supplied data, illustrate the stages of the given sort and count the actual number of steps. (i) odd-even transposition sort (ii) odd-even merge sort.

   9 3 6 1 7 8 2 4

2. An ***oblivious*** sorting algorithm is one which consists entirely of a prescribed sequence of compare-and-swap operations: each step is to compare two elements and swap them if necessary so that they are in a particular order.

   (a) Explain why the odd-even transposition sort is oblivious, but neither Quicksort nor Merge Sort is oblivious.

   (b) ⓔ Prove that if an oblivious algorithm sorts a list consisting entirely of 0s and 1s, then it sorts a general list.

# *Chapter 5: Finding the Median*

The ***median*** of a collection is the middle value (in sorted order). For example, the collection $\{3, 1, 6, 2, 8, 4, 9\}$ has a median of 4: There are three values smaller than 4 and three values larger. The answer to a median question when there is an even number of values is the smaller of the two middle values. (There are other possible definitions but this is convenient.)

We want an algorithm for finding the median of a collection of $n$ values. One possibility would be to sort the list. But one hopes for a faster method. One can try recursion or iteration, but at first glance they don't seem to work.

Blum, Floyd, Pratt, Rivest and Tarjan provided a linear-time algorithm: it uses $O(n)$ comparisons. It is a bit hairy at places.

## 5.1 A Generalization

One approach to the median problem is to define a generalization: the ***rank selection*** problem. The input is a collection of $n$ numbers and a rank $k$. The output must be the value that is the $k$th smallest value. For example, for the above collection, 4 has rank 4 and 8 has rank 6. The minimum has rank 1.

Thus the median of a collection with $n$ values has rank $\lceil n/2 \rceil$ (round up): it's $n/2$ if $n$ is even and $(n+1)/2$ if $n$ is odd.

## 5.2 Using Median for Rank Selection

Okay. We do not know how to find the median. But I now argue that if (and that's a big if) we have an algorithm for median, then we can derive an algorithm for rank-selection:

> *We use the median algorithm to break the list in half. And keep only the relevant half.*

How much work does this take?

**RankSelect** (A:valuelist, k:integer)

R1: Calculate the median $m$ of the list $A$.

R2: Determine the set $A_<$ of elements smaller than the median.
Determine the set $A_>$ of elements larger than the median.

R3: If $k < \lceil n/2 \rceil$ then return RankSelect($A_<$, $k$)
If $k > \lceil n/2 \rceil$ then return RankSelect($A_>$, $k - \lceil n/2 \rceil$)
Else return $m$

Suppose that the median procedure takes $cn$ comparisons where $c$ is some constant. (That is, suppose we have a linear-time median algorithm.) The number of comparisons for Step R1 the first time is $cn$. The number for Step R2 the first time is $n$. The total for the first two steps is therefore $n(1+c)$ comparisons.

The second time we have a list half the size. So our work is half that of the original. The number of comparisons for Step R1 the second time is $cn/2$. The number for Step R2 the second time is $n/2$. The total for the first two steps is $n(1+c)/2$ comparisons.

Hence the total number of comparisons is

$$n(1+c) + \frac{n(1+c)}{2} + \frac{n(1+c)}{4} + \frac{n(1+c)}{8} + \frac{n(1+c)}{16} + \frac{n(1+c)}{32} + \ldots$$

If this sum continues forever it would add up to

$$2n(1+c).$$

So the total work is at most $2(1+c)n$. This is linear! (The beauty of the geometric series.)

## 5.3 The Pseudo-Median

We still don't have a median algorithm. There are two ideas needed:
(1) that the above algorithm is still efficient if one uses only an ***approximation*** to the median; and
(2) one can actually find an approximation to the median.

We define a ***pseudo-median*** as any value such that at least 30% of the values lie above the pseudo-median and at least 30% of the values lie below the pseudo-median. (The value 30% is chosen to make the presentation easier.)

Suppose we have an algorithm that finds a pseudo-median. Then we can use it for rank selection in exactly the same way as above. (And hence we can find the median!)

**RankSelect** (A:valuelist, k:integer)

R1: Calculate a pseudo-median $m$ of the list $A$.

R2: Determine the set $A_<$ of elements smaller than $m$.
Determine the set $A_>$ of elements larger than $m$.
Calculate the rank $r$ of $m$ in $A$.

R3: If $k < r$ then return RankSelect($A_<$, $k$)
If $k > r$ then return RankSelect($A_>$, $k - r$)
Else return $m$

Hence we have a function RankSelect which uses as a subroutine the function Pseudo-Median.

## 5.4 Finding a Pseudo-Median

How do we find a pseudo-median? What we do is:

*take a **sample** of the data and find the median of the sample.*

This algorithm is bizarre but beautiful: we now have a function Pseudo-Median which uses as a subroutine the function RankSelect.

**Pseudo-Median** (B:valuelist)

P1: Break up the list B into quintets (groups of 5).

P2: Calculate the median in each quintet (by brute-force). Call the result the **representative** of the quintet. There are n/5 representatives.

P3: Return the median of the set of representatives:
RankSelect(reps,n/10)

Does this produce a pseudo-median?

Well, consider the resultant value $m_r$ which is the median of the representatives. This value is bigger than $n/10$ representatives. And for each representative that $m_r$ is bigger

than, $m_r$ is bigger than two other members of the quintet. Thus there are at least $3n/10$ values which $m_r$ is guaranteed to be bigger than. And similarly there are $3n/10$ values which $m_r$ is guaranteed to be smaller than. Hence $m_r$ is guaranteed to be a pseudo-median.

Note that the intricate recursion: RankSelect calls PseudoMedian which calls RankSelect and so on. The reason why this works is that the problem size is getting smaller at each stage. (When implementing this algorithm one must be very careful not to get an infinite loop...)

## 5.5 Analysis

The calculation of the median in a quintet takes at most 10 comparisons. (Since sorting the quintet takes this many comparisons!) So the total number of comparisons for Step P2 is $2n$.

Now, let us return to the RankSelect algorithm modified to use PseudoMedian. Let $g(n)$ be the number of comparisons needed by RankSelect on a list of size $n$ in the worst case. Then the number of comparisons for the three steps are:

Step R1: $2n + g(n/5)$
Step R2: $n$
Step R3: $g(7n/10)$

This gives us the recurrence:

$$g(n) = g(n/5) + g(7n/10) + 3n$$

which has solution $g(n) = 30n$. (Plug it in and check!)

## Exercises

1. Implement the median-finding algorithm.
2. Run your program on random lists of length 10, 100, 1000 and 10 000. Comment on the results.
3. Why quintets? Use either theory or a modification of your program to compare triplets with quintets.

# *Chapter 6: Books*

I have made much use of:

• *Data Structures and Algorithms*,
A. Aho, J. Hopcroft and J. Ullman

• *Fundamentals of Algorithmics*,
G. Brassard and P. Bratley

• *Introduction to Algorithms*,
T. Cormen, C. Leiserson and R. Rivest

• *Introduction to Parellel Algorithms and Architectures: Arrays, Trees, Hypercubes*,
F.T. Leighton

• *Data Structures and Algorithm Analysis,*
M.A. Weiss