Lecture 13: Divide and Conquer (1997)

Steven Skiena

Department of Computer Science
State University of New York
Stony Brook, NY 11794–4400

http://www.cs.sunysb.edu/∼skiena

Problem Solving Techniques

Most important: make sure you understand exactly what the question is asking – if not, you have no hope of answer it!!

Never be afraid to ask for another explanation of a problem until it is clear.

Play around with the problem by constructing examples to get insight into it.

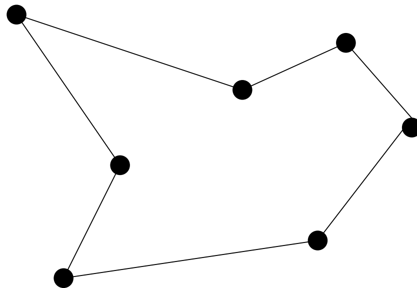
Ask yourself questions. Does the first idea which comes into my head work? If not, why not?

Am I using all information that I am given about the problem?

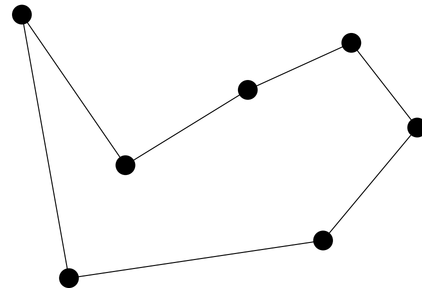
Read Polya's book *How to Solve it*.

The Euclidean traveling-salesman problem is the problem of determining the shortest closed tour that connects a given set of n points in the plane.

Bentley suggested simplifying the problem by restricting attention to bitonic tours, *that is tours which start at the leftmost point, go strictly left to right to the rightmost point, and then go strictly right back to the starting point.*



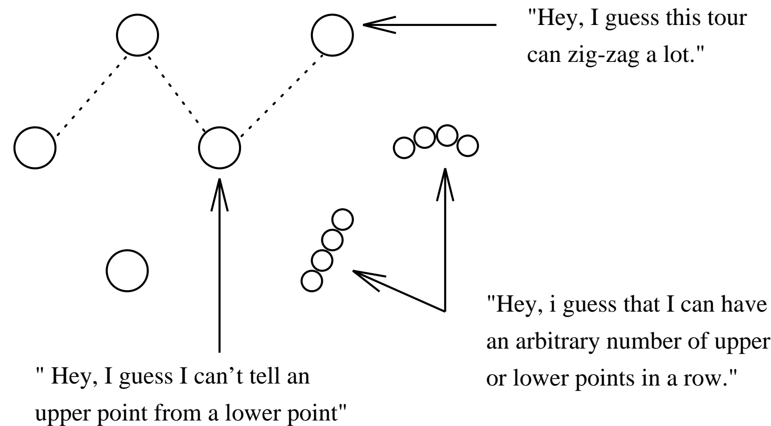
non-bitonic



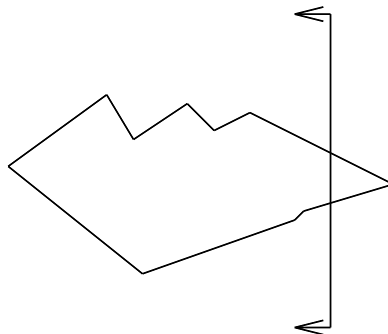
bitonic

Describe an $O(n^2)$ algorithm for finding the optimal bitonic tour. You may assume that no two points have the same x-coordinate. (Hint: scan left to right, maintaining optimal possibilities for the two parts of the tour.)

Make sure you understand what a bitonic tour is, or else it is hopeless.
First of all, play with the problem. Why isn't it trivial?



Am I using all the information?
Why will they let us assume that no two x-coordinates are the same? What does the hint mean? What happens if I scan from left to right?
If we scan from left to right, we get an open tour which uses all points to the left of our scan line.



In the optimal tour, the kth point is connected to exactly one point to the left of k. $(k \neq n)$ Once I decide which point that is, say x. I need the optimal partial tour where the two endpoints are x and $k-1$, because if it isn't optimal I could come up with a better one. Hey, I have got a recurrence! And look, the two parameters which describe my optimal tour are the two endpoints.

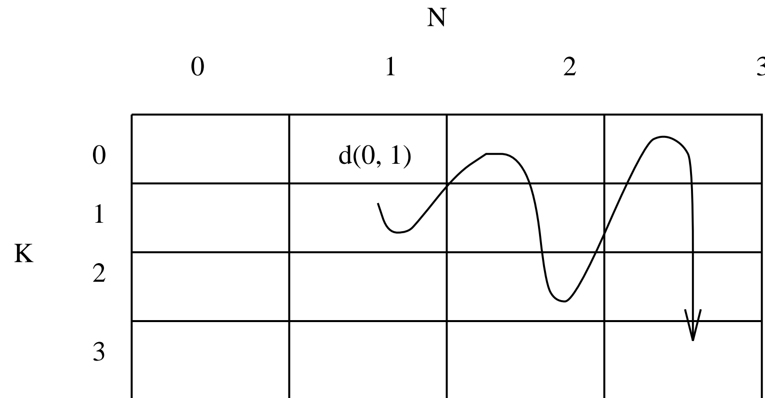
Let $c[k,n]$ be the optimal cost partial tour where the two

endpoints are $k < n$.

$c[k, n] \leq c[k, n-1] + d[n, n-1]$ (when $k < n-1$)

$c[n-1, n] \leq c[k, n-1] + d[k, n]$

$c[0, 1] = d[0, 1]$



Filling the entities in from N=1 to N', k=1 to N, means we always have what we need waiting for us.

$c[n-1, n]$ takes $O(n)$ to update, $c[k, n]$ $k < n-1$ takes $O(1)$

to update. Total time is $O(n^2)$.
But this doesn't quite give the tour, but just an open tour. We simply must figure where the last edge to n must go.

$$Tourcost = \min_{k=1}^{n} C[k, n] + d_{kn}$$

Divide and Conquer

Divide and conquer was a successful military strategy long before it became an algorithm design paradigm. The wise general would attack so as to divide the enemy army into two forces and then mop up one after the other.

To use divide and conquer as an algorithm design technique, we must divide the problem into two smaller subproblems, solve each of them recursively, and then meld the two partial solutions into one solution to the full problem. Whenever the merging takes less time than solving the two subproblems, we get an efficient algorithm.

Mergesort is the classic example of a divide-and-conquer algorithm. It takes only linear time to merge two sorted lists

of $n/2$ elements each of which was obtained in $O(n \lg n)$ time.
Divide and conquer is a design technique with many important algorithms to its credit, including mergesort, the fast Fourier transform, and Strassen's matrix multiplication algorithm.

Fast Exponentiation

Suppose that we need to compute the value of a^n for some reasonably large n. Such problems occur in primality testing for cryptography.

The simplest algorithm performs $n - 1$ multiplications, by computing $a \times a \times \ldots \times a$.

However, we can do better by observing that $n = \lfloor n/2 \rfloor + \lceil n/2 \rceil$. If n is even, then $a^n = (a^{n/2})^2$. If n is odd, then $a^n = a(a^{\lfloor n/2 \rfloor})^2$. In either case, we have halved the size of our exponent at the cost of at most two multiplications, so $O(\lg n)$ multiplications suffice to compute the final value.

function power(a, n)

    if $(n = 0)$ return(1)

$x = \text{power}(a, \lfloor n/2 \rfloor)$
if (n is even) then return(x^2)
else return($a \times x^2$)

This simple algorithm illustrates an important principle of divide and conquer. It always pays to divide a job as evenly as possible.

Twenty Questions

In *Twenty questions* one player selects a word, and the other repeatedly asks true/false questions in an attempt to identify the word. If the word remains unidentified after 20 questions, the first party wins; otherwise, the second player wins.

In fact, the second player always has a winning strategy, based on binary search. Given a printed dictionary, the player opens it in the middle, selects a word (say "move"), and asks whether the unknown word is before "move" in alphabetical order.

Since standard dictionaries contain 50,000 to 200,000 words, we can be certain that the process will always terminate within twenty questions.

Finding a Transition

Other interesting algorithms follow from simple variants of binary search.

Suppose we have an array A consisting of a run of 0's, followed by an unbounded run of 1's, and would like to identify the exact point of transition between them:

$$00000000000000000000000 11111111111$$

Binary search on the array would provide the transition point in $\lceil \lg n \rceil$ tests.

Clearly there is no way to solve this problem any faster.

One-Sided Binary Search

Suppose that we want to search in a sorted array, but we do not know how large the array is. All we know is the starting point.

$$\{2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, \ldots\}$$

How can we use binary search without both boundaries?
In the absence of such a bound, we can test repeatedly at larger intervals ($A[1]$, $A[2]$, $A[4]$, $A[8]$, $A[16]$, $\ldots$) until we find a first nonzero value. Now we have a window containing the target and can proceed with binary search.
This *one-sided binary search* finds the transition point p using at most $2\lceil \lg p \rceil$ comparisons, regardless of how large the array actally is.

One-sided binary search is most useful whenever we are looking for a key that probably lies close to our current position.

Square and Other Roots

The square root of n is the number r such that $r^2 = n$. Square root computations are performed inside every pocket calculator – but how?

Observe that the square root of $n \geq 1$ must be at least 1 and at most n. Let $l = 1$ and $r = n$. Consider the midpoint of this interval, $m = (l + r)/2$. How does m^2 compare to n?

If $n \geq m^2$, then the square root must be greater than m, so the algorithm repeats with $l = m$. If $n < m^2$, then the square root must be less than m, so the algorithm repeats with $r = m$.

Either way, we have halved the interval with only one comparison. Therefore, after only $\lg n$ rounds we will have identified the square root to within ± 1.

This bisection method, as it is called in numerical analysis, can also be applied to the more general problem of finding the roots of an equation. We say that x is a *root* of the function f if $f(x) = 0$.