Introduction to Algorithms

M. Kaykobad & Rifat Shahriyar & Atif Hasan Rahman

November 2017

DRAFT – 19 November 2017

to my xxx, yyy (yyy is the people you want to dedicated this book to.)

Contents

List of Figures

List of Tables

Chapter 1

Foundation

1.1 What is Algorithm

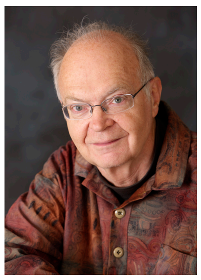
The word ‘Algorithm’, pronounced as AL-go-rith-um, represents a process or set of rules to be followed in calculations or other problem-solving operations, especially by a computer. The word was originated in late 17th century (denoting the Arabic or decimal notation of numbers): variant (influenced by Greek arithmos ’number’) of Middle English algorism, via Old French from medieval Latin algorismus. The Arabic source, al-Kwa-rizmi ‘the man of al-Kwa-rizm’ (now Khiva), was a name given to the 9th-century mathematician Abu Ja’far Muhammad ibn Musa, author of widely translated works on algebra and arithmetic. Professor Donald E. Knuth believes that the word originated from the later source.

Muhammad ibn Musa al-Khwarizmi(c. 780 c. 850)
Muhammad ibn Musa al-Khwarizmi, formerly latinized as Algoritmi, was a Persian mathematician, astronomer, and geographer during the Abbasid Caliphate, a scholar in the House of Wisdom in Baghdad. In the 12th century, Latin translations of his work on the Indian numerals introduced the decimal positional number system to the Western world. Al-Kwa-rizmi’s the Compendious Book on Calculation by Completion and Balancing presented the first systematic solution of linear and quadratic equations in Arabic. He is often considered one of the fathers of algebra. He revised Ptolemy’s Geography and wrote on astronomy and astrology.

An algorithm is any well-defined computational procedure which takes some input and produces output having desirable properties. Steps of algorithms should be definite, effective and finite. If it is devoid of the last property then it is called ’computational procedure’ for which the widely known example is ’operating systems’. If we write a code to verify the now resolved Fermat’s last theorem that there are no positive integers x, y, z and integer $n > 2$ such that $x^n + y^n = z^n$ by creating loops for x, y, z and n then such a computational procedure will not correspond to algorithm since such a procedure will never stop unless forced to do so.

1.2 Problem and Algorithm

Let us formally define a sorting problem as follows:
Input: A sequence of n numbers $(a_1, a_2, ..., a_n)$
Output: A permutation $(a'_1, a'_2, \cdots, a'_n)$ of the input data such that $a'_1 \leq a'_2, \cdots, \leq a'_n$.
If $(12, 7, 31, 25, 24, 25, 54)$ is input then the desirable output is $7, 12, 24, 25, 25, 31, 54$.
Requirement of a good algorithm is the right data structures and algorithm for manipulation of involved data structures. If unary number system is used then there cannot be any *efficient* algorithm for adding or multiplying such numbers. A traditional chess board can be logically replaced by 64 squares in a row in which a computer will be efficiently playing the game whereas for human players such unidimensional representation of a chess board will make it difficult for him to play. So this data structure will not be suitable for human players.
Designing of good algorithms is an art, and Professor Donald Knuth rightly names his encyclopedic series as "The Art of Computer Programming".



Donald Ervin Knuth
Donald Ervin Knuth (/knu/[4] k-NOOTH; born January 10, 1938) is an American computer scientist, mathematician, and professor emeritus at Stanford University.
He is the author of the multi-volume work "The Art of Computer Programming". He contributed to the development of the rigorous analysis of the computational complexity of algorithms and systematized formal mathematical techniques for it. In the process he also popularized the asymptotic notation. In addition to fundamental contributions in several branches of theoretical computer science, Knuth is the creator of the TeX computer typesetting system, the related METAFONT font definition language and rendering system, and the modern family of typefaces.
As a writer and scholar, Knuth created the WEB and CWEB computer programming systems designed to encourage and facilitate literate programming, and designed the MIX/MMIX instruction set architectures. Knuth strongly opposes granting software patents, having expressed his opinion to the United States Patent and Trademark Office and European Patent Organization.

It is not wise to design a different algorithm for each different problem. Rather a set of core algorithms should be used to solve problems by transforming those to problems for which efficient solutions are known. An algorithm is said to be efficient if it can solve a problem in polynomial time of the input size. As Jack Edmond asserts no problem is well solved unless there is a polynomial time algorithm for solving it. In chapter 9 we will show that there are problems for which no efficient algorithm is known, yet there are problems for which there cannot exist any efficient algorithm unless there is a breakthrough in resolving the issue whether $P \neq NP$, where P stands for class of problems that is polynomially solvable, and NP is the class of problems whose solution can be verified in polynomial time.

We shall be using structured English or pseudocode with appropriate indentation for understandability to represent an algorithm. This will help us analyze algorithms at ease, find its bottlenecks and improve those segments. The keywords to be used in pseudocodes will have usual unambiguous meaning.

Different computers take different amount of time in performing various operations. So it is a good idea to give a rough estimation of amount of computational time and other resources that will be involved in execution of a program. In particular we want to have an idea on the upper and lower bounds of computation. We want to know whether the computation time will increase twice or 4 times or even more if the problem size is doubled. What happens when size of the problem increases to infinity? Usually order of computation is measured in terms of functions of the input size of a problem.

1.3 Growth of Functions

Complexity of an algorithm for solving a problem is usually expressed as a function of size of a problem. Size of a problem is the amount of space required in computers to express the problem. If we are adding two $m \times n$ matrices then we need $2mn$ values which can be integers, reals or doubles. So we can say that problem size is proportional to mn. It will require exactly mn additions. In case two matrices $\mathbf{A}(m \times n)$ and $\mathbf{B}(n \times p)$ are multiplied then we need mnp multiplications and equal number of additions. We can say that the algorithm will require operations proportional to mnp. If matrices are of order n, computation will be proportional to $\mathcal{O}(n^3)$. In case of matrix additions if number of rows and columns are doubled then number of operations will increase 4 times whereas in case of multiplication it will increase 8 times. We shall denote worst case running time of an algorithm by $T(n)$ where n is the size of the problem.

$\mathcal{O}(n)$-notation: $f(n)$ is said to be $\mathcal{O}(g(n))$ if there exist positive constants c and n_0 such that

$$0 \leq f(n) \leq cg(n), \ \forall n \geq n_0$$

For sequential search we can choose $g(n) = n$, $c = 1$, $n = 1$, for binary search $g(n) = \log_2 n$, $c = 3$, $n_0 = 2$. It may be noted that a higher value of c will also be fine. So Big-O is an upper bound of the computation requirements. If an algorithm is $\mathcal{O}(n)$ then it is also $\mathcal{O}(n^2)$, $\mathcal{O}(n \log_2 n)$, and so on.

$\Omega(n)$-notation: While Big-O is an asymptotic upper bound for an algorithm Ω-notation provides an asymptotic lower bound. $f(n)$ is said to be $\Omega(g(n))$ if there exist positive constants c and n_0 such that

$$0 \leq cg(n) \leq f(n), \ \forall n \geq n_0$$

For sequential search we can choose $g(n) = n$, $c = \frac{1}{2}$, $n = 1$, for binary search $g(n) = \log_2 n$, $c = 1$, $n_0 = 2$. It may be noted that a lower value of c will also do.
$\theta(n)$**-notation:** $f(n)$ is said to be $\theta(g(n))$ if $f(n)$ is at the same time $\mathcal{O}(g(n))$ and $\Omega(g(n))$. In other words there exist positive constants c_1, c_2 and n_0 such that

$$0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n),\ \forall n \geq n_0$$

$o(n)$**-notation:** $f(n) = o(g(n))$, pronounced as 'little -o' if for any positive constant $c > 0$, there exists a constant $n_0 > 0$ such that for all $n > n_0$

$$0 \leq f(n) < cg(n)$$

When we say $2n^2 = \mathcal{O}(n^2)$ the bound is asymptotically tight but the bound $2n = o(n^2)$ is not. We use o-notation to denote an upper bound that is not asymptotically tight. For example $2n = o(n^2)$, but $2n^2 \neq o(n^2)$.
$\omega(n)$**-notation:** In the same way, $\omega - notation$ is to $\Omega - notation$ as o-notation is to O-notation. We can define $f(n) = \omega(g(n))$ if and only if $g(n) \in o(f(n))$ Formally $f(n)$ is said to be $\mathcal{O}(g(n))$ if there exists positive constants c and n_0 such that

$$0 \leq f(n) \leq cg(n),\ \forall n \geq n_0$$

Some Examples

$$2n^2 + 3n + 1000 = 2n^2 + \Theta(n) = \mathcal{O}(n^2)$$

where $c = 2.1$, $n_0 = 100$ or even we can choose $c = 3$, $n_0 = 33$. We can even say $2n^2 + 3n + 1000 = \mathcal{O}(n^2 \log_2 n) = \mathcal{O}(n^3)$ and so on. However, it is always better to give tighter bounds.

1.4 Counting

Let us show some examples of counting operations within loops.
Assuming every statement requires equal number of operations, total number of operations in algorithm A_1 will be

$$\sum_{i=1}^{n} c = cn$$

Whereas in A_2 it will be $2cn$ and in A_3 it will be

$$\sum_{i=1}^{n}\sum_{j=i}^{n} c = \sum_{i=1}^{n} c(n-i+1) = \sum_{i=1}^{n} ci = \frac{n(n+1)}{2}c = \mathcal{O}(n^2)$$

So roughly speaking number of nested loops determine the order of computation.
Now we shall see some algorithms, analyze them and improve their running time by applying some standard techniques. Let us now see how a program segment can be made more efficient simply by suitably ordering conditions in 'if' or 'case'-like

Algorithm 1 Counting operations within loops

```
Require: Counting in loop in $A_1$
 1: for i=1 to n do
 2:     statement 1
 3:     statement 2
 4:     ...
 5:     statement c
 6: end for
```

```
Require: Counting in sequential loops in $A_2$
 1: for i=1 to n do
 2:     statement 1
 3:     statement 2
 4:     ...
 5:     statement c
 6: end for
 7: for i=1 to n do
 8:     statement 1
 9:     statement 2
10:     ...
11:     statement c
12: end for
```

```
Require: Counting in nested loops in $A_3$
 1: for i=1 to n do
 2:     for j=1 to n do
 3:         statement 1
 4:         statement 2
 5:         ...
 6:         statement c
 7:     end for
 8: end for
```

statements.

Algorithm 2 Counting division operations for determining leap year

```
Require: LpYr(n)
 1: if n is not divisible by 4 then
 2:     n is not a leap year
 3: else if n is not divisible by 100 then
 4:     n is a leap year
 5: else if n is not divisible by 400 then
 6:     n is not a leap year
 7: else
 8:     n is a leap year
 9: end if
```

```
Require: LpYr1(n)
 1: if If n is divisible by 400 then
 2:     n is a leap year
 3: else if n is divisible by 100 then
 4:     n is not a leap year
 5: else if n is divisible by 4 then then
 6:     n is a leap year
 7: else
 8:     n is not a leap year
 9: end if
```

Before we can calculate expected number of operations let us recapitulate notions of random variables. There are two types of random variables- discrete that can assume discrete values like number of comparisons or the value that come up in a dice toss, and continuous random variable that can assume continuous values like the distance traversed by a throwing stone. For any random variable there are two important parameters- measure of central tendency like 'mean', median' and 'mode' and measure of how dispersed are the values of the variable around its central tendency, like 'standard deviation'. The most important among central tendencies is 'mean' also called 'expectation'. Expectation of a random variable is denoted by $E[X]$ and is calculated as follows:

$$E[X] = \sum_{i=1}^{n} p_i.x_i$$

where x_i is the value the discrete random variable X assumes with probability p_i. For example, in dice tossing $x_1 = 1,\ x_2 = 2,\ x_3 = 3, ..., x_6 = 6$, and if the dice is fair then $p_i = \frac{1}{6}, i = 1, ..., 6$.
For a continuous random variable X the formula is

$$E[X] = \int_0^\infty x f(x) dx$$

Expected number of comparisons for Algorithm 2: We shall calculate the expected number of comparisons/divisions for both the algorithms. X- random variable denoting number of comparisons. It assumes values 1, 2 and 3 so is discrete random variable.
For the first algorithm

$$E[X] = \frac{1 \times 300}{400} + \frac{2 \times 96}{400} + \frac{3 \times 4}{400} \approx 1.25$$

For the second algorithm

$$E[X] = \frac{1 \times 1}{400} + \frac{2 \times 3}{400} + \frac{3 \times 396}{400} \approx 3$$

The first algorithm is better. In *if* and *case* statements conditions having higher probability of being satisfied should be placed earlier.

Let us now discuss searching - one of the most important operations done by a computer system.

1.5 Searching

Searching is the most frequent operations performed by a computer as duly acknowledged by Knuth through dedicating the third volume of his celebrated series "The Art of Computer Programming" exclusively to sorting and searching. He remarked that over 25% of computer time in 1960's was used in searching.
Generally we search for solutions/answers in databases or in computation intensive problems. In our daily life shops in markets or a town are arranged to minimize searching time, as chess boards are also represented to make moves efficiently or household goods, admission test results or words in dictionaries are located to minimize searching time. Imagine had there been no shops that would sell all items of monthly shopping how difficult and time consuming would it have been to perform monthly shopping. Had not all shoe shops been located in the same place like that of Elephant Road in the city of Dhaka how difficult would it have been to choose shoes (Fortunately powerful mobiles have changed the situation!). Had a dictionary

contained some 100,000 useful words but in no known order would there have been a knowledge-seeker to find a word from those thousand pages? (Dictionaries available in the Internet has made it easier for us to search for a word without being much worried about whether the words are stored in a sorted order). So we organize data to perform operations efficiently on them, search them out using less effort.
When objects are placed in a list sequentially we cannot do better than searching it sequentially.

Algorithm 3 Sequential search

```
Require: Sequential-Search($A, n, z, index$)
1: $i = 1,\ index = -1$
2: while $i \leq n$ and $A(i) \neq z$ do
3:     $i++$
4: end while
5: if $i \leq n$ then
6:     $index = i$
7: end if
```

```
Require: Sequential-Search1($A, n, z, index$)
1: $A(n+1) = z,\ i = 1,\ index = -1$
2: while $A(i) \neq z$ do
3:     $i++$
4: end while
5: if $i \leq n$ then
6:     $index = i$
7: end if
```

By checking $i \leq n$ we want to ensure that we do not search for z beyond our array. This comparison can be avoided by putting a sentinel at the end like in Algorithm Sequential-search1.
In algorithm Sequential-Search1 index comparison $i \leq n$ is not required since even if the element were not in the original list it has been appended at $(n+1)$st location, and we are not going to run out of the array. In case the array is large even this index comparison can turn out costly, and its trade off with an extra location may be worthwhile. However, in general in analyzing algorithms we will not take cost of index comparisons in consideration since it is far too less costly than data comparison.
Let us do some analysis. When we add two matrices predominant operation is addition, when we multiply two matrices predominant operation is multiplication. In sequential search predominant operation is comparison. So we shall count number of comparisons in the best, average and worst cases, and in case of successful and unsuccessful searches.

Table 1.1: Cost of Sequential Search

columns rows	best	worst	average
successful	1	n	$\frac{n+1}{2}$
unsuccessful	n	n	n

Best case for successful search is when we search for the first element, worst case is the one where the last element is searched, whereas average case is found by counting all the comparisons when we search each element of the array once and then divide by the number of cases. However, when n is very large and we need to search out significant number of data items, sequential search would be too costly to pursue. Imagine had your roll numbers in the successful lists of Dhaka University test for admission into first year classes or that at Bangladesh University of Engineering and Technology appeared in no order, how difficult would it have been for you to go through the whole list to find only yours? On the average half of it, to find your name or after the list is exhausted you come to learn that you are unsuccessful.

To search efficiently we must do much better than sequential search although not without cost. We must preprocess data as in dictionary, maintain some order when stored. One way is to use hashing - keep an element in a location that is a function of the element itself. Another other way is to keep elements in the sorted order.

Hashing is another way of searching efficiently. We can store elements at a location that is a function of the element. Say we want to store 'dam' we compute location as $4.26^2 + 1.26^1 + 13.26^0 = 2743$. If number of location is limited to m (say $m = 1000$) then we take $2743(\mod m)$ and store 'dam' at location 743 if it is empty. If we simply added the numeric position of each letter then 'dam' and 'mad' would have been hashed into the same location causing a collision. So if there is a collision, some hash collision resolution procedure has to be applied. It may be 'linear probe' in which case we need to find the first empty slot to store that data item or double hashing in which another hash function is applied to locate an empty slot for the data item. When we want to retrieve the data we must apply the hash function. If the location generated contains something else we must understand that there was a collision and the data item has been stored using a collision resolution scheme. Hashing is elaborately discussed in a course on 'data structures'.

Another widely used approach of efficient searching is by sorting elements like that in dictionaries. In binary search we compare the middle element of an array to see whether the element we are searching for appears in lower half or upper half of the array. So searching becomes restricted to half the file after each comparison as in the following algorithms.

In case of both the algorithms Binary-search and Binary-search1 dominant operation is comparison. In algorithm Binary-search inside the loop we have on the average 1.5 comparisons. By delaying finding z in Binary-search1 until the loop ends, comparison count within the loop comes down to simply 1 although we have one more comparison outside the loop. So in the worst case Binary-search may require $2\lg n$ comparisons whereas Binary-search1 requires $\lg n + 1$ in the worst case.

Let us look at the average case performance, which we obtain by searching every element of the array A once and then taking average.

For simplicity of analysis let us assume $n = 2^k - 1$. So total number of comparisons is

$$S = 1.2^0 + 2.2^1 + 3.2^2 + ... + k.2^{k-1} 2S = 1.2^1 + 2.2^2 + 3.2^3 + ... + k.2^k$$

Algorithm 4 Two algorithms for binary search

```
Require: Binary-Search($A, n, z, index$)
 1: $index = -1, low = 1, high = n$
 2: while $index = -1$ and $lowhigh$ do
 3:    $mid = (low + high)/2$
 4:    if $A(mid) < z$ then
 5:       $low = mid + 1$
 6:    else if $A(mid) > z$ then
 7:       $high = mid - 1$
 8:    else
 9:       $index = mid$
10:    end if
11: end while
```

```
Require: Binary-Search1($A, n, z, index$)
 1: $A(n+1) = z, index = -1, low = 1, high = n + 1, A(high) = \infty$
 2: while $low < high$ do
 3:    $mid = \frac{low+high}{2}$
 4:    if $A(mid) < z$ then
 5:       $low = mid + 1$
 6:    else
 7:       $high = mid$
 8:    end if
 9: end while
10: if $A(mid) = z$ then
11:    $index = mid$
12: end if
```

Table 1.2

# of elements	# of comparisons
2^0	1
2^1	2
2^2	3
2^{k-1}	k

Subtracting the first equation from the second we get

$$S = k2^k - (2^0 + 2^1 + ... + 2^k) = (k-1)2^k + 1 \approx k - 1 \approx \lg_2 n - 1$$

Hence average number of comparisons is $\approx k - 1 \approx \lg_2 n - 1$. Here we have assumed that each data item is equally likely to be retrieved. If probability distribution is different then we have a different average. In the worst case after the full loop has been executed we may find the element. This will require $\lceil \log_2(n+1) \rceil \times 2$ comparisons per loop in the worst case. But Binary Search1 will take only $\lceil \log_2(n+1) \rceil + 1$ comparison. While for algorithms that are frequently run we shall be interested in average performance, theory of computational complexity generally considers worst case performance of an algorithm.
Dictionary searching is *interpolation search* since if we want to search for a word beginning with 'v' we go near the end whereas for searching a word beginning with 'c' we open a page near the start.

Interval Tree

Given a set of n intervals on the number line, we want to construct a data structure so that we can efficiently retrieve all intervals overlapping another interval or point.
We start by taking the entire range of all the intervals and dividing it in half at x_c (in practice, x_c should be picked to keep the tree relatively balanced). This gives three sets of intervals, those completely to the left of x_c which we'll call S_l, those completely to the right of x_c which we will call S_r, and those overlapping x_c which we will call S_c.
The intervals in S_l and S_r are recursively divided in the same manner until there are no intervals left.
The intervals in S_c that overlap the center point are stored in a separate data structure linked to the node in the interval tree. This data structure consists of two lists, one containing all the intervals sorted by their beginning points, and another containing all the intervals sorted by their ending points.
The result is a ternary tree with each node storing:
A center point
A pointer to another node containing all intervals completely to the left of the center point
A pointer to another node containing all intervals completely to the right of the center point
All intervals overlapping the center point sorted by their beginning point
All intervals overlapping the center point sorted by their ending point

Intersection

Given the data structure constructed above, we answer queries consisting of ranges or points by returning all the ranges in the original set overlapping this input.

With a Point

The task is to find all intervals in the tree that overlap a given point x. The tree is walked with a similar recursive algorithm as would be used to traverse a traditional binary tree, but with extra affordance for the intervals overlapping the "center" point at each node.

For each tree node, x is compared to x_c, the midpoint used in node construction above. If x is less than x_c, the leftmost set of intervals, S_l, is considered. If x is greater than x_c, the rightmost set of intervals, S_r, is considered. All intervals in S_c that begin before x must overlap x if x is less than x_c. Similarly, the same technique also applies in checking a given interval. If a given interval ends at y and y is less than x_c, all intervals in S_c that begin before y must also overlap the given interval.

As each node is processed as we traverse the tree from the root to a leaf, the ranges in its S_c are processed. If x is less than x_c, we know that all intervals in S_c end after x, or they could not also overlap x_c. Therefore, we need only find those intervals in S_c that begin before x. We can consult the list of S_c that have already been constructed. Since we only care about the interval beginnings in this scenario, we can consult the list sorted by beginnings. Suppose we find the closest number no greater than x in this list. All ranges from the beginning of the list to that found point overlap x because they begin before x and end after x (as we know because they overlap x_c which is larger than x). Thus, we can simply start enumerating intervals in the list until the startpoint value exceeds x.

Likewise, if x is greater than x_c, we know that all intervals in S_c must begin before x, so we find those intervals that end after x using the list sorted by interval endings.

If x exactly matches x_c, all intervals in S_c can be added to the results without further processing, and tree traversal can be stopped.

With an Interval

For an interval r to intersect our query interval q one of the following must hold: the start and/or end point of r is in q; or r completely encloses q.

We first find all intervals with start and/or end points inside q using a separately-constructed tree. In the one-dimensional case, we can use a search tree containing all the start and end points in the interval set, each with a pointer to its corresponding interval. A binary search in $\mathcal{O}(logn)$ time for the start and end of q reveals the minimum and maximum points to consider. Each point within this range references an interval that overlaps q and is added to the result list. Care must be taken to avoid duplicates, since an interval might both begin and end within q. This can be done using a binary flag on each interval to mark whether or not it has been added to the result set.

Finally, we must find intervals that enclose q. To find these, we pick any point inside q and use the algorithm above to find all intervals intersecting that point (again, being careful to remove duplicates). Java Example: Searching a point or an interval in the tree

To search for an interval, one walks the tree, using the key (n.getKey()) and high value (n.getValue()) to omit any branches that cannot overlap the query. The simplest case is a point query:

// Search for all intervals containing "p", starting with the // node "n" and adding matching intervals to the list "result" public void search(IntervalNode n, Point p,

List<Interval> result) // Don't search nodes that don't exist if (n == null) return;
// If p is to the right of the rightmost point of any interval // in this node and all children, there won't be any matches. if (p.compareTo(n.getValue()) > 0) return;
// Search left children if (n.getLeft() != null) search(IntervalNode (n.getLeft()), p, result);
// Check this node if (n.getKey().contains(p)) result.add(n.getKey());
// If p is to the left of the start of this interval, // then it can't be in any child to the right. if (p.compareTo(n.getKey().getStart()) < 0) return;
// Otherwise, search right children if (n.getRight() != null) search(IntervalNode (n.getRight()), p, result);
where
a.compareTo(b) returns a negative value if a < b a.compareTo(b) returns zero if a = b a.compareTo(b) returns a positive value if a > b
The code to search for an interval is similar, except for the check in the middle:
// Check this node if (n.getKey().overlapsWith(i)) result.add (n.getKey());
overlapsWith() is defined as:
public boolean overlapsWith(Interval other) return start.compareTo(other.getEnd()) <= 0 and end.compareTo(other.getStart()) >= 0;
We have analyzed two algorithms above and in each case counted number of comparisons. There are algorithms where comparison is not the dominant operation. In fact there may be movement, multiplication and other operations as well.
In the following we show a problem where algorithm has been made efficient by introducing a new data structure. While a naive algorithm may require $\mathcal{O}(n^3)$ computation, adding an extra data structure results in a linear time algorithm.

1.6 A Share Market Problem

In a share market price of commodities increases or decreases everyday. Assume that this increase with respect to previous day's price is denoted by a positive number and decrease by a negative number. So price of the market has been expressed in a series of positive and negative numbers. Assume that by stroke of luck you happen to know this series in advance to decide when to buy and when to sell. Your task is to identify the time when to buy and the time when to sell in order to maximizing profit. In fact we are interested in finding the sum of consecutive subsequence that gives us the maximum value. Naturally we are not going to start the subsequence with negative values since then we will have a higher sum deleting those negative values. In the same way the subsequence cannot end in a sequence of negative values for the same reason. We are given an array A of n numbers, not necessarily positive. Index corresponds to time interval whereas entries of A correspond to profit or loss compared to previous day. The problem is to find its consecutive subsequence whose sum is the maximum. We can have the following two naive algorithms, and later on an efficient algorithm to solve the problem.

Algorithm 5 Two naive algorithms for MSCS

```
Require: MSCS-naive($A, n, maxsum$)
 1: $maxsum = 0$
 2: for i=1 to n do
 3:   for j=i to n do
 4:     $sum = 0$
 5:     for k=i to j do
 6:       $sum = sum + A(k)$
 7:     end for
 8:     if $sum > maxsum$ then
 9:       $maxsum = sum$
10:     end if
11:   end for
12: end for
```

```
Require: MSCS-naive1($A, n, maxsum$)
 1: $maxsum = 0$
 2: for i=1 to n do
 3:   $sum(i) = 0$
 4:   for j=i to n do
 5:     $sum(i) = sum(i) + A(j)$
 6:     if $sum(i) > maxsum$ then
 7:       $maxsum = sum(i)$
 8:     end if
 9:   end for
10: end for
```

Number of operations involved is

$$
\begin{aligned}
\sum_{i=1}^{n}\sum_{j=i}^{n}\sum_{k=i}^{j} 1 &= \sum_{i=1}^{n}\sum_{j=i}^{n}(j-i+1) \\
&= \sum_{i=1}^{n}\sum_{j=1}^{n-i+1} j = \sum_{i=1}^{n}\frac{(n-i+1)(n-i+2)}{2} \\
&= \frac{1}{2}\sum_{i=1}^{n}\{(n-i)^2+3(n-i)+2\} = \frac{1}{2}\sum_{i=1}^{n-1}\{i^2+3i+2\} \\
&= \frac{1}{2}\frac{(n-1)n(2n-1)}{6} + \frac{3n(n-1)}{2} + 2n = \frac{n(n+1)(n+2)}{6} \\
&= O(n^3) \qquad (1.1)
\end{aligned}
$$

If intermediate sums are stored then it can be transformed into an $\mathcal{O}(n^2)$ algorithm. However, if we analyze the problem further we can exploit some properties of successive *maxsum*s we discover. Assume that we have found *maxsum* of the subarray up to position i. How does *maxsum* of subarray up to $(i+1)$st position differ from that of up to ith position? Either it will be the old one, or if it has a new *maxsum* it must be bigger, it must have $A(i+1)$ value and the largest suffix from ith position backward. If this *suffixmax* can be stored and updated with every new element considered then possibly we can do it in a better way requiring lesser computational time. This is how it can be done. Remember naturally *suffixmax* cannot be negative since then it can be ignored to have a larger value. We will assume that *maxsum* will not be negative. In that case *maxsum* will correspond to consecutive subarray having 0 element. Every time we have a new *maxsum* we must have a new *suffixmax* having the same value! If we do not have a new *maxsum* with a new element considered we still must update *suffixmax* since it must contain the last element of the subarray considered. The following linear time algorithm was devised by Joseph Kadane of Carnegie Mellon University in 1984.

Joseph B. Kadane
Joseph B. Kadane (born January 10, 1941) is the Leonard J. Savage University Professor of Statistics, Emeritus in the Department of Statistics and Social and Decision Sciences at Carnegie Mellon University. He is one of the early proponents of Bayesian statistics, particularly the subjective Bayesian philosophy. Born in Washington, DC and raised in Freeport on Long Island, Kadane (known as Jay), prepared at Phillips Exeter Academy, earned an A.B. in mathematics from Harvard College and a Ph.D. in statistics from Stanford in 1966, under the supervision of Professor Herman Chernoff. While in graduate school, Jay worked for the Center for Naval Analyses (CNA). Upon finishing, he accepted a joint appointment at the Yale statistics department and the Cowles Foundation. In 1968 he left Yale and served as an analyst at CNA for three years. In 1971, Jay moved to Pittsburgh to join Morris H. DeGroot at Carnegie Mellon University. He became the second tenured professor in the Department of Statistics. Jay served as department head from 1972-1981 and steered the department to a balance between theoretical and applied work, advocating that statisticians should engage in joint research in substantive areas rather than acting as consultants. Jays contributions span a wide range of fields: econometrics, law, medicine, political science, sociology, computer science (see maximum subarray problem), archaeology, and environmental science, among others. Among many honors, Jay has been elected as a Fellow of the American Academy of Arts and Sciences, a fellow of the American Association for the Advancement of Science, a fellow of the American Statistical Association, and a fellow of the Institute of Mathematical Statistics. He has authored over 250 peer-reviewed publications and has served the statistical community in many capacities, including as editor of the Journal of the American Statistical Association from 1983-85.

Example

Table 1.3: Maximum Sum of Consecutive Subsequence

index-i	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
A(i)	1	-2	3	4	-5	-6	7	8	-9	10	11	-12	13	14	-15	16	-17	18
maxsum	1	1	3	7	7	7	7	15	15	16	27	27	28	42	42	43	43	44
suffixmax	1	0	3	7	2	0	7	15	6	16	27	15	28	42	27	43	26	44

So if we now analyze amount of computation required then we will see that since there is a single loop computation is $\mathcal{O}(n)$! Although comparison is not the only operation, there are assignments as well. However, number of assignments cannot

be more than $2n$. So the algorithm is $\mathcal{O}(n)$. Just by analyzing the solution we have devised a better algorithm faster by an order!
Next we address a problem whose solution will show that sometimes order of computation may be lower than the order of space required to express the problem.

1.7 Celebrity Problem

Celebrity is a person in a group whom everybody knows but who does not know anybody. Strictly speaking, there cannot be any celebrity in any collection if one includes oneself there in 'everybody'. If we ignore this then there can be at most 1 celebrity in any group. Let the relation be expressed by the following matrix:

$$know(i,j) = \begin{cases} 1, & \text{if } i \text{ knows } j \\ 0, & \text{otherwise} \end{cases}$$

While there are n^2 elements computation will be simply $\mathcal{O}(n)$ since comparing an element of a row will either throw a row away or a column away. So we will be left with a row and a column whose elements will have to be checked for compatibility with celebrity conditions. Initially assume that $know(i,i) = 0 \ \forall i$.

Algorithm 6 A smart algorithm for Maximum Sum of Consecutive Subsequence Problem

```
Require: MSCS-smart($A, n, maxsum$)
 1: $maxsum = suffixmax = 0$
 2: for i=1 to n do
 3:   if $suffixmax + A(i) > maxsum$ then
 4:     $maxsum = suffixmax + A(i)$
 5:     $suffixmax = maxsum$
 6:   else if $suffixmax + A(i) > 0$ then
 7:     $suffixmax = suffixmax + A(i)$
 8:   else
 9:     $suffixmax = 0$
10:   end if
11: end for
```

1.8 Efficiency from Recursion

Recursive routines are usually slower in execution. However, this is not always the case as will be evident in the following example. Consider the problem of finding minimum and maximum of an array having n elements. We may have the following program segments.
For the algorithm minmax number of comparisons is $2(n-1)$, number of assignments random, for minmax1 both are random. Assuming each element to be equally likely to be the kth smallest(largest) element of the array, ith element will be the new maximum with probability $\frac{1}{i}$. This observation will lead to an expected number of assignments equalling $2H_n$. For minimax1 number of comparisons will be

Algorithm 7 Two algorithms for MinMax

```
Require: Minmax($A, n, min, max$)
1: $min = max = A(1)$
2: for i=2 to n do
3:     if $A(i) < min$ then
4:         $min = A(i)$
5:     end if
6:     if $A(i) > max$ then
7:         $max = A(i)$
8:     end if
9: end for
```

```
Require: minmax1($A, n, min, max$)
1: $min = max = A(1)$
2: for i=2 to n do
3:     if $A(i) < min$ then
4:         $min = A(i)$
5:     else if $A(i) > max$ then
6:         $max = A(i)$
7:     end if
8: end for
```

$2(n-1) - H_n - 1$ and number of assignments will be $2H_n$. Not an impressive or effective trick though. However, in Chapter 2 we shall show that a recursive algorithm for solving this problem is computationally better.

Having shown some examples where simple tricks can improve running time of an algorithm significantly now we would like to address the issue of designing algorithms. As good essays can be written by reading compositions of good essayists so is the case with the design of algorithms. In the following chapters we shall address some of the design principles that good computer scientists have applied in devising algorithms. These are greedy techniques, dynamic programming, divide and conquer methods, recursions and backtracking.

1.9 Exercises

1. Rank the following functions by order of growth, that is, find an ordering of the following functions such that

$$g_1 = \Omega(g_2) = \Omega(g_3), ..., \Omega(g_{29}) = \Omega(g_{30})$$

Partition the list into equivalence classes such that $f(n)$ and $g(n)$ are in the same class if and only if $f(n) = \Theta(g(n))$.

$\lg\lg^* n$	$2^{\lg^* n}$	$\sqrt{2}^{\lg n}$	n^2	$n!$	$\lg n!$
$\frac{3}{2}^n$	n^3	$\lg^2 n$	$\lg n!$	2^{2^n}	$n^{\frac{1}{\lg n}}$
$\ln\ln n$	$\lg^* n$	$n2^n$	$n^{\lg\lg n}$	$\ln n$	$n\lg^k n$
$2^{\lg n}$	$\lg n^{\lg n}$	e^n	$4^{\lg n}$	$(n+1)!$	$\sqrt{\lg n}$
$\lg^*(\lg n)$	$2^{\sqrt{2\lg n}}$	n	2^n	$n\lg n$	$2^{2^{n+1}}$

Table 1.4: test caption

2. Order the following functions as per complexity classes of $\omega, o, O, \theta, \Omega$
   $\sqrt{n} \lg n!$, 2^{500}, $\lg \lg n$, $\lg n^2$, $2^{\lg n}$, 2^{2^n}, $\lceil \sqrt{n} \rceil$, $n^{0.001}$, $n^4(\frac{5}{3})^n$, $4n^{\frac{3}{2}}$, $3n^{0.5}$, $n^{2.2}(2.2)^n$, $\lfloor 2n \lg n \rfloor$, 2^n, $n \lg_7 n + n^{\frac{5}{6}}$
3. How do you search an element i in an array of numbers such that $A(i) = i$?
4. How do you search for an element in a bitonic sequence? (When a segment of a sorted sequence from the start is brought to the end or vice versa the new sequence is called bitonic).
5. How do you search z equalling sum of an element from a sorted array X and corresponding element from the sorted array Y? Deduce its complexity.
6. Given an $m \times n$ sorted matrix $\mathbf{A}$ with $a_{ij} \leq a_{kl}$ whenever $i \leq k$, and $j \leq l$ design an algorithm for deciding whether an element z is in the matrix or not. Compute its complexity.

** **References** Volume 3: Sorting and Searching. Second Edition (Reading, Massachusetts: Addison-Wesley, 1998), xiv+780pp.+foldout. ISBN 0-201-89685-0

Chapter 1

Throughout a computer program algorithms manipulate values of some locations known as data structures applying valid operations on them. Bits, integers, reals, arrays are all data structures. Some data structures have been found extremely useful for facilitating computation. We shall define and describe some of them.

2.1 Stacks and Queues

One of the simplest and useful data structure is linear/ordered list written as $a = (a_1, a_2, \ldots, a_n)$. Operations that can be done on it are retrieval of data given an index i, done in $\mathcal{O}(1)$ time, insertion of an item requiring $\mathcal{O}(n)$ operations in shifting all elements after downwards and deletion of an element requiring shifting all elements below upwards. A *stack* is an ordered list in which all insertions, called *push*, and deletions called *pop* can be performed only from one end called *top*. A queue is an ordered list in which insertion occurs at the *tail* whereas deletion occurs at the *head*. Pseudocode of these stack operations are as follows assuming that stack can contain a maximum of N elements in the array s. STACKFULL and STACKEMPTY are the two conditions respectively corresponding to the case where allocation of memory is exhausted and stack does not contain any element.

Stack and Queue operations			
Push(s, N, top,z)	Pop(s, N, top,z)	InQ(q, head, tail, z)	DelQ(q, head, tail, z)
if $n = N$ then	If $n = 0$ then	tail=(tail+1)mod N	if head=tail then
STACKFULL	STACKEMPTY	if (head=tail) then	QEMPTY
else	else	QUEUEFULL	else
n++	z=s(top)	if head=0 then	head=(head+1) mod N
s(n)=z	top–	tail=N-1	z=q(head)
endif	endif	endif	endif
			else
			queue(rear)=item
			endif

Operations on a linked list		
Search(L,k)	Insert(L,x)	Delete(L,x)
x=head(L)	next(x)=head(L) , tail=(tail+1)mod N	if head=tail then
while $x \neq .nil$ and $key(x) \neq .k$ do	if $head(L).ne.nil$	if $prev(x) \neq .nil$ then
x=next(x)	prev(head(L))=x	next(prev(x))=next(x)
enddo	endif	else head(L)=next(x)
s(n)=z	head(L)=x	endif
endif	prev(x)=nil	if $next(x) \neq .nil$ then
		prev(next(x))=prev(x)
		endif

So stack is referred to as Last-In-First-Out (LIFO) whereas queue is referred to as a First-In-First-Out (FIFO) system.
Stacks can be implemented using linked lists or arrays. While linked lists require more space but they do provide more flexibility in terms of dynamic allocation of space whereas in array implementation we must assign the memory large enough to meet the worst case requirements. The most important advantage is that insertion and deletion operations require only $\mathcal{O}(1)$ operations irrespective of implementation. Whereas queues can be implemented in circular arrays limiting its size to the maximum elements in the queue only.

2.2 Trees

Trees are important data structures for reducing cost of searches.

2.2.1 Binary Search Trees

Binary search trees (BST) are a data structure for storing "items" (such as numbers, names, etc.) in memory. They allow fast lookup, addition and removal of items, and can be used to implement either dynamic sets of items, or lookup tables that allow finding an item by its key.
Keys are arranged in sorted order, so that lookup and other operations can use the principle of binary search. They traverse the tree from root to leaf, making comparisons with keys stored in the nodes of the tree and deciding, based on the comparison, to continue searching in the left or right subtrees. This means that on the average each comparison allows the operations to skip about half of the tree, so that each lookup, insertion or deletion takes time proportional to the height of the tree or to the logarithm of the number of items stored in the tree. This is much better than the linear time required to find items by key in an (unsorted) array, but slower than the corresponding operations on hash tables. The following recursive routine finds an item stored in a binary search tree.
In the worst case this algorithm must search from the root of the tree to the leaf farthest from the root. So the search operation takes time proportional to the tree's

height. On average, binary search trees with n nodes have $\mathcal{O}(\log n)$ height. However, in the worst case, binary search trees can have $\mathcal{O}(n)$ height, when the unbalanced tree resembles a linked list.
During insertion operation the element is compared with the root, and then we search the left or right subtrees as appropriate. In the process either we reach a vertex that has the same key value or end up in an external node implying that the search has ended unsuccessfully, and if desired we can store that key at that external node location.
Here is how a typical binary search tree insertion might be performed in a binary tree in C++:

```
void insert(Node*& root, int data)
if (!root)
root = new Node(data);
else if (data < root->data)
insert(root->left, data);
else if (data > root->data)
insert(root->right, data);
```

The above destructive procedural variant modifies the tree in place. It uses only constant heap space (and the iterative version uses constant stack space as well), but the prior version of the tree is lost. Alternatively, as in the following pseudocode we can reconstruct all ancestors of the inserted node; any reference to the original tree root remains valid, making the tree a persistent data structure:

```
def binary-tree-insert(node, key, value):
if node is None:
return TreeNode(None, key, value, None)
if key == node.key:
return TreeNode(node.left, key, value, node.right)
if key < node.key:
return TreeNode(binary-tree-insert(node.left, key, value), node.key, node.value, node.right)
else:
return TreeNode(node.left, node.key, node.value, binary-tree-insert(node.right, key, value))
```

The part that is rebuilt uses $\mathcal{O}(\log n)$ space in the average case and $\mathcal{O}(n)$ in the worst case (see big-O notation).
In either version, this operation requires time proportional to the height of the tree in the worst case, which is $\mathcal{O}(\log n)$ time in the average case over all trees, but $\mathcal{O}(n)$ time in the worst case.
As we mentioned a BST may end up in a trivial list with depth proportional to n, and may not prove effective in searching, insertion or deletion operations. So there are variants in which worst case search time is bounded by logarithmic height. One such variation is red-black tree.

A binary search tree is a red-black tree if it satisfies the following red-black properties:

1. Every node is either red or black.
2. The root is black.
3. Every leaf (nil) is black.
4. If a node is red, then both its children are black.
5. For each node, all paths from the node to descendant leaves have the same number of black nodes.

While a binary tree can be degenerate and turn into a linear list thus resulting in linear time for searching, insertion and deletion operations, red-black trees ensure logarithmic time for these operations with height of the tree not exceeding $2\lg(n+1)$.

2.2.2 Interval Trees

A closed interval i is an ordered pair of real numbers $[t_1, t_2]$ with $t_1 \leq t_2$, and is represented by $low(i) = t_1$ and $high(i) = t_2$. The interval $[t_1, t_2]$ represents the set $\{t \in R : t_1 \leq t \leq t_2\}$ Interval trees are an efficient way of representing the events that occupy a continuous period of time. For example, if we like to query a database of time intervals to find out what events occurred during a given interval then interval trees can be used to satisfy such queries efficiently. Any two intervals i and i_1 satisfy the following properties:

1. i and i' overlap.
2. i is to the left of i', that is, $high(i) < low(i')$
3. i is to the right of i', that is, $high(i') < low(i)$

An interval tree is a red-black tree that maintains a dynamic set of elements, with each element x containing an interval $interval(x)$. Interval trees support the following operations.
Interval-insert(T, x) adds the element x, whose *interval* field is assumed to contain an interval x, to the interval tree T. Interval-delete(T, x) removes the element x from the interval tree T. Interval-search(T, i) returns a pointer to an element x in the interval tree such that interval(x) overlaps interval i, or the sentinel nil(T) if no such element is in the set.

```
interval-search($T, i$)
x=root(T)
while $x \neq nil(T)$ and $i$ does not overlap interval(x) do
if $left(x) \neq nil(T)$ and $max(left(x)) \geq low(i)$ then
$x = left(x)$
else
$x = right(x)$
```

endif
enddo

2.3 Hashing

Many applications require dictionary operations like search, insert and delete. For example, a compiler of a language maintains a symbol table, in which keys of elements are arbitrary character strings that correspond to identifiers in the language. A *hash table* is an effective data structure for implementing dictionaries. Although searching in a hash table may take $\mathcal{O}(n)$ time fortunately on the average searching out an element from a hash table requires only $\mathcal{O}(1)$. In a hash table an element is stored in a location that is computed from the value by a function called *hash function*. A hash function is good when it can hash elements uniformly into the allotted space. A hash function h maps any element of the universe into the slots of the hash table $T(0,1,...,m-1)$. We say that an element with key k has been hashed to slot $h(k)$. However, unfortunately different elements can be hashed into the same slot. For example, let us have have a hash table of size $N = 10, k$ be a word and $h(k) = \sum_{i=1}^{m} k(i)(\mod N)$ where $k(i)$ is the position of the ith letter in the alphabet. Then $h(if) = (6+9)(\mod 32) = 15, h(while) = (23+8+9+12+5)(\mod 32) = 15$. If we have already stored 'if' we cannot store 'while' in the same place. So there are collision resolution schemes like linear probing in which next free slot is searched for this key or double hashing in which another hash function is used to find a different slot once collision occurs. Yet another scheme for collision resolution is to maintain a linked list for all elements hashed into the same slot. Let us discuss the following collision resolution techniques *separate chaining*, *linear probing* and *quadratic probing*. (pls draw from http://www.cs.cmu.edu/ ab/15-121N11/lectures/lecture16.pdf)

Separate Chaining:

Use an array of linked lists LinkedList[ ] Table;

Table = new LinkedList(N), where N is the table size

Define Load Factor of Table as λ = number of keys/size of the table (λ can be more than 1) Now we need a good hash function to distribute keys evenly in the Table.

Linear Probing: In this case Table remains a simple array of size N. To carry out the operation Insert(x), first computer $f(x) \mod N$. If the cell is occupied, find another by sequentially searching for the next available slot, that is, go to $f(x)+1, f(x)+2....$ In performing find(x), compute $f(x) \mod N$, if the cell does not match, look elsewhere.

Linear probing function can be given by

$h(x,i) = (f(x)+i) \mod N(i = 1,2,....)$

quadratic probing: Here instead of searching in contiguous positions probing is done at positions i^2 cells away from the hashed location. Naturally success is not guaranteed since all those cells could be occupied. But linear probing is always successful in locating an empty slot so long as load factor $\lambda < 1$.

Chapter 2

Divide and Conquer Algorithms

3.1 Introduction

Divide and conquer (D&C) is an algorithm design paradigm based on multi-branched recursion. This algorithmic technique works by recursively breaking down a problem into several sub-problems of the same (or related) type, until these become simple enough to be solved directly. The solutions to the sub-problems are then combined to form a solution to the original problem.
This technique is the basis of efficient algorithms like, quicksort, merge sort, multiplying large numbers (e.g. Karatsuba), syntactic analysis (e.g., top-down parsers), and computing the Fast Fourier transform (FFTs).
On the other hand, the ability to understand and design D&C algorithms is a skill that requires time to master. The correctness of a divide and conquer algorithm is usually proved by mathematical induction, and its computational cost is often determined by solving recurrence relations. The name "divide and conquer" is sometimes applied also to algorithms that reduce each problem to only one sub-problem, such as the binary search algorithm for finding a record in a sorted list or its analog in numerical computing, the bisection algorithm for root finding[1]. These algorithms can be implemented more efficiently than general divide-and-conquer algorithms. The name decrease and conquer has been proposed instead for the single-subproblem class[3].
Decrease and conquer strategy can be applied to optimization problems, reduction of the search space by a constant factor ("pruning step") at each step would imply that the overall algorithm has the same asymptotic complexity as the pruning step.

3.2 Complexity

Let us first analyze complexity of a divide-and-conquer based algorithm.
Assume that a problem of size n is divided into m subproblems of size $\frac{n}{m}$ and only $k < m$ of those subproblems should be solved to obtain solution of the larger problem.

We also assume that running time of dividing into subproblems and combining their solutions is $g(n)$, whereas complexity of the algorithm is $f(n)$. Suppose that for problem of size $\leq n_0$ a straight forward method requires f_0 cost. Then

$$T(n) = \begin{cases} f_0 & \text{if } n \leq n_0, \\ kT(\frac{n}{m}) + g(n) & \text{if } n > n_0. \end{cases} \tag{3.1}$$

Establishing the order of complexity of a divide-and-conquer based algorithm is ensured by the so-called *Master Theorem* that goes as follows:
Master Theorem If $T(n) \in \Theta(n^d),\ d \geq 0$ then

$$T_(n) = \begin{cases} \Theta(n^d) & \text{if } k \leq m^d, \\ \Theta(n^d \lg n) & \text{if } k = m^d, \\ \Theta(n^{\log_m k} & \text{if } k > m^d. \end{cases} \tag{3.2}$$

Similar result is also applicable for $\mathcal{O}$ notation.

3.3 Examples

Let us begin with the problem of multiplying large numbers and modular exponentiation to appreciate how a large problem can be solved by subdividing it into smaller problems and then merging solutions of subproblems to obtain solution of the large problem.

3.3.1 Karatsuba Algorithm

In 1960, Kolmogorov organized a seminar on mathematical problems in cybernetics at the Moscow State University, where he stated the $\Omega(n^2)$ conjecture and other problems in the complexity of computation. Within a week, Karatsuba, then a 23-year-old student, found an algorithm that multiplies two n-digit numbers in $\Theta(n^{\log_2 3})$ elementary steps, thus disproving the conjecture. Kolmogorov was excited, and delivered some lectures on the Karatsuba result at the conferences all over the world (see, for example, "Proceedings of the international congress of mathematicians 1962", pp. 351-356, and also "6 Lectures delivered at the International Congress of Mathematicians in Stockholm, 1962") and published the method in 1962, in the Proceedings of the USSR Academy of Sciences. The article had been written by Kolmogorov and contained two results on multiplication, Karatsuba's algorithm and a separate result by Yuri Ofman; it listed "A. Karatsuba and Yu. Ofman" as the authors. Karatsuba only became aware of the paper when he received the reprints from the publisher.[2]

Anatoly Alexeevitch Karatsuba
Anatoly Alexeevitch Karatsuba (Russian: ; Grozny, Soviet Union, January 31, 1937 Moscow, Russia, September 28, 2008) was a Russian mathematician working in the field of analytic number theory, p-adic numbers and Dirichlet series. For most of his student and professional life he was associated with the Faculty of Mechanics and Mathematics of Moscow State University, defending a D.Sc. thesis entitled "The method of trigonometric sums and intermediate value theorems" in 1966. He later held a post at the Steklov Institute of Mathematics of the Academy of Sciences. His textbook Foundations of analytic number theory went to two editions, 1975 and 1983. The Karatsuba algorithm is the earliest known divide and conquer algorithm for multiplication and lives on as a special case of its direct generalization, the ToomCook algorithm.

The basic step of Karatsuba's algorithm is a formula that allows us to compute the product of two large numbers x and y using three multiplications of smaller numbers, each with about half as many digits as x or y, plus some additions and digit shifts. Let x and y be represented as n-digit strings in some base B. For any positive integer m less than n, one can write the two given numbers as

$$x = x_1 B^m + x_0,\ y = y_1 B^m + y_0,$$

where x_0 and y_0 are less than B^m. The product is then

$$xy = (x_1 B^m + x_0)(y_1 B^m + y_0),\ xy = z_2 B^{2m} + z_1 B^m + z_0$$

where

$$z_2 = x_1 y_1,\ z_1 = x_1 y_0 + x_0 y_1,\ z_0 = x_0 y_0$$

These formulae require four multiplications, and were known to Charles Babbage[4]. Karatsuba observed that xy can be computed in only three multiplications, at the cost of a few extra additions. With z_0 and z_2 as before we can calculate

$$z_1 = +z_2 + z_0 - (x_1 - x_0)(y_1 - y_0)$$

which holds since

$$z_1 = x_1 y_0 + x_0 y_1,\ z_2 = x_1 y_1 + x_0 y_0 - (x_1 + x_0)(y_1 + y_0)$$

Now the Karatsuba algorithm can be formalized as follows:

```
procedure karatsuba(num1, num2)
$if(num1 < 10)or(num2 < 10)$
$returnnum1 * num2$
/* calculates the size of the numbers */
$m = max(size - base10(num1), size - base10(num2))$
```

```
$m_2 = \frac{m}{2}$
/* split the digit sequences about the middle */
$high1, low1 = splitat(num1, m_2)$
$high2, low2 = splitat(num2, m_2)$
/* 3 calls made to numbers approximately half the size */
$z_0 = karatsuba(low1, low2)$
$z_1 = karatsuba((low1 + high1), (low2 + high2))$
$z_2 = karatsuba(high1, high2)$
return $z_2 * 10^(2 * m_2) + (z_2 + z_0 - z_1) * 10^(m_2) + (z_0)$
```

The mathematician Carl Friedrich Gauss (1777 -1855) once noticed that although the product of two complex numbers $(a + bi)(c + di) = ac - bd + (bc + ad)i$ seems to involve four real-number multiplications, it can in fact be done with just three: ac, bd, and $(a + b)(c + d)$, since $bc + ad = (a + b)(c + d) - ac - bd$. A. A. Karatsuba (1995). "The Complexity of Computations". Proceedings of the Steklov Institute of Mathematics 211: 169-183. Translation from Trudy Mat. Inst. Steklova, 211, 186-202 (1995) Charles Babbage, Chapter VIII - Of the Analytical Engine, Larger Numbers Treated, Passages from the Life of a Philosopher, Longman Green, London, 1864; page 125.

3.3.2 Modular Exponentiation

The problem of modular exponentiation is to find the residue of x^y when divided by N. In this case all these 3 numbers can be very large like having 200 decimal digits. This algorithm is widely used in cryptography. The idea is to raise the power of an integer by repeated squaring to bring down complexity of computation.

```
function modexp($x, y, N$ )
Input: Two n-bit integers $x$ and $N$, an integer exponent $y$
Output: $x^y modN$

if $y = 0$ then return 1
else $z = modexp(x, \lfloor \frac{y}{2} \rfloor, N)$
endif if $y$ is even then
return $z^2 modN$
else
return $xz^2 modN$
endif
```

3.3.3 Euclid's Algorithm

Here is another one for finding the greatest common divisor of two numbers. This algorithm wasa devised by Euclid who has a profound impact on education of science through the famous book " The Elements".

Euclid
Euclid (/jukld/; Greek: , Eukleids Ancient Greek: [eu.kl.ds]; fl. 300 BCE), sometimes called Euclid of Alexandria to distinguish him from Euclides of Megara, was a Greek mathematician, often referred to as the "father of geometry". He was active in Alexandria during the reign of Ptolemy I (323 – 283 BCE). His Elements is one of the most influential works in the history of mathematics, serving as the main textbook for teaching mathematics (especially geometry) from the time of its publication until the late 19th or early 20th century. In the Elements, Euclid deduced the principles of what is now called Euclidean geometry from a small set of axioms. Euclid also wrote works on perspective, conic sections, spherical geometry, number theory and rigor. Euclid is the anglicized version of the Greek name $EU\kappa\lambda\epsilon\iota\delta\eta\zeta$, which means "renowned, glorious".

Euclid's GCD algorithm

```
$functionEuclid(a,b)$
Input: Two integers $a$ and $b$ with $a \geq b \geq 0$
$Output: gcd(a,b)$

$ifb = 0then\ returna$
$else$
$returnEuclid(b, amodb)$
```

3.3.4 Calculating Fibonacci Numbers

While Fibonacci numbers can be calculated using a simple algorithm all the time adding the preceding two Fibonacci numbers, however calculating say $F_{10^{15}}$ will not be that easy computationally. Applying the recurrence relation

$$F_{n+2} = F_{n+1} + F_n,\ F_0 = 0,\ F_1 = 1$$

will require prohibitively large amount of computation. If we even follow the iterative algorithm

```
for i=2 to n do
$F_i = F_{i-1} + F_{i-2}$
enddo
```

10^{15} additions are to be performed not to speak of adding numbers having hundreds of digits. However, we can apply the trick of repeated squaring as follows to bring down order of computation. We start by expressing the Fibonacci relations $F_1 = F_1,\ F_2 = F_0 + F_1$ by

$$\begin{bmatrix} \mathbf{F_1} \\ \mathbf{F_2} \end{bmatrix} = \begin{bmatrix} \mathbf{0} & \mathbf{1} \\ \mathbf{1} & \mathbf{1} \end{bmatrix} \begin{bmatrix} \mathbf{F_0} \\ \mathbf{F_1} \end{bmatrix}$$

In fact since

$$\begin{bmatrix} \mathbf{F_n} \\ \mathbf{F_{n+1}} \end{bmatrix} = \begin{bmatrix} \mathbf{0} & \mathbf{1} \\ \mathbf{1} & \mathbf{1} \end{bmatrix} \begin{bmatrix} \mathbf{F_{n-1}} \\ \mathbf{F_n} \end{bmatrix}$$

we get

$$\begin{bmatrix} \mathbf{F_n} \\ \mathbf{F_{n+1}} \end{bmatrix} = \begin{bmatrix} \mathbf{0} & \mathbf{1} \\ \mathbf{1} & \mathbf{1} \end{bmatrix}^n \begin{bmatrix} \mathbf{F_0} \\ \mathbf{F_1} \end{bmatrix}$$

When n is large as in our case we can raise the power of the matrix exactly in the same manner as we have done exponentiation of a number. This will result in calculation proportional to $logn$. So in order to finding $F_{10^{15}}$ we need to raise the power of the matrix roughly $2log10^{15} \approx 100$ times.
Since F_n is exponential in n, in fact,

$$F_n = \frac{1}{\sqrt{5}}\Big(\frac{1+\sqrt{5}}{2}\Big)^n - \frac{1}{\sqrt{5}}\Big(\frac{1-\sqrt{5}}{2}\Big)^n$$

the numbers themselves will have digits proportional to n. Squaring such numbers using traditional matrix multiplication algorithm will require $\mathcal{O}(n^2)$ multiplications, application of Karatsuba algorithm will require $\mathcal{O}(n^{log3})$. So application of modular exponentiation strengthened with Karatsuba algorithm will result in $\mathcal{O}(n^{log3}logn)$ computation.

3.3.5 Matrix Multiplication

In initial days of computer technology this technique was used to overcome the deficiency of low primary memory. Say we want to multiply 2 matrices

$$\mathbf{C} = \mathbf{AB} \qquad \mathbf{A}, \mathbf{B}, \mathbf{C} \in R^{2^n \times 2^n}$$

A and **B** may be so large that they cannot be stored in primary memory. So matrices **A** and **B** are divided as

$$\mathbf{A} = \begin{bmatrix} \mathbf{A}_{11} & \mathbf{A}_{12} \\ \mathbf{A}_{21} & \mathbf{A}_{22} \end{bmatrix}, \mathbf{B} = \begin{bmatrix} \mathbf{B}_{11} & \mathbf{B}_{12} \\ \mathbf{B}_{21} & \mathbf{B}_{22} \end{bmatrix}, \mathbf{C} = \begin{bmatrix} \mathbf{C}_{11} & \mathbf{C}_{12} \\ \mathbf{C}_{21} & \mathbf{C}_{22} \end{bmatrix}$$

Now for multiplying or adding these two matrices it is sufficient to bring a pair of matrices of half the orders to primary memory. Not only that recursion is being used to devise more efficient algorithms. For example, sorting using bubble sort requires $\mathcal{O}(n^2)$ comparisons, whereas recursively dividing the array results in better algorithms like mergesort, quicksort etc.
Matrix multiplication, by division as above, comes out to an unimpressive $\mathcal{O}(n^3)$, the same as for the default algorithm. But the efciency can be further improved, and as with integer multiplication, the key is some clever algebra. It turns out XY can be computed from just seven $\frac{n}{2} \times \frac{n}{2}$ subproblems, via a decomposition so tricky and

intricate that one wonders how Strassen was ever able to discover it!
Let A, B be two square matrices. We want to calculate the matrix product C as

$$\mathbf{C} = \mathbf{AB} \qquad \mathbf{A}, \mathbf{B}, \mathbf{C} \in R^{2^n \times 2^n}$$

If the matrices $\mathbf{A}, \mathbf{B}$ are not of type $2n \times 2n$ we fill the missing rows and columns with zeros.
We partition $\mathbf{A}, \mathbf{B}$ and $\mathbf{C}$ into equally sized block matrices

$$\mathbf{A} = \begin{bmatrix} \mathbf{A}_{11} & \mathbf{A}_{12} \\ \mathbf{A}_{21} & \mathbf{A}_{22} \end{bmatrix}, \mathbf{B} = \begin{bmatrix} \mathbf{B}_{11} & \mathbf{B}_{12} \\ \mathbf{B}_{21} & \mathbf{B}_{22} \end{bmatrix}, \mathbf{C} = \begin{bmatrix} \mathbf{C}_{11} & \mathbf{C}_{12} \\ \mathbf{C}_{21} & \mathbf{C}_{22} \end{bmatrix}$$

with

$$\mathbf{A}_{ij}, \mathbf{B}_{ij}, \mathbf{C}_{ij} \in R^{2^{n-1} \times 2^{n-1}}$$

then

$$\begin{aligned} \mathbf{C}_{11} &= \mathbf{A}_{11}\mathbf{B}_{11} + \mathbf{A}_{12}\mathbf{B}_{21} \\ \mathbf{C}_{12} &= \mathbf{A}_{11}\mathbf{B}_{12} + \mathbf{A}_{12}\mathbf{B}_{22} \\ \mathbf{C}_{21} &= \mathbf{A}_{21}\mathbf{B}_{11} + \mathbf{A}_{22}\mathbf{B}_{21} \\ \mathbf{C}_{22} &= \mathbf{A}_{21}\mathbf{B}_{12} + \mathbf{A}_{22}\mathbf{B}_{22} \end{aligned}$$

With this construction we have not reduced the number of multiplications. We still need 8 multiplications to calculate the C_{ij} matrices, the same number of multiplications we need when using standard matrix multiplication. However, Strassen had a different idea. If we could compute 2 by 2 matrices in less than 8 non-commutative multiplications then recursive application of the technique would have reduced complexity significantly. In fact we achieve this by the following:

$$\begin{aligned} \mathbf{M}_1 &:= (\mathbf{A}_{11} + \mathbf{A}_{22})(\mathbf{B}_{11} + \mathbf{B}_{22}) \\ \mathbf{M}_2 &:= (\mathbf{A}_{21} + \mathbf{A}_{22})\mathbf{B}_{11} \\ \mathbf{M}_3 &:= \mathbf{A}_{11}(\mathbf{B}_{12} - \mathbf{B}_{22}) \\ \mathbf{M}_4 &:= \mathbf{A}_{22}(\mathbf{B}_{21} - \mathbf{B}_{11}) \\ \mathbf{M}_5 &:= (\mathbf{A}_{11} + \mathbf{A}_{12})\mathbf{B}_{22} \\ \mathbf{M}_6 &:= (\mathbf{A}_{21} - \mathbf{A}_{11})(\mathbf{B}_{11} + \mathbf{B}_{12}) \\ \mathbf{M}_7 &:= (\mathbf{A}_{12} - \mathbf{A}_{22})(\mathbf{B}_{21} + \mathbf{B}_{22}) \end{aligned}$$

Only 7 multiplications (one for each M_k) suffice. We may now express the C_{ij} in terms of M_k, like this:

$$\mathbf{C}_{1,1} = \mathbf{M}_1 + \mathbf{M}_4 - \mathbf{M}_5 + \mathbf{M}_7$$

$$\mathbf{C}_{1,2} = \mathbf{M}_3 + \mathbf{M}_5$$
$$\mathbf{C}_{2,1} = \mathbf{M}_2 + \mathbf{M}_4$$
$$\mathbf{C}_{2,2} = \mathbf{M}_1 - \mathbf{M}_2 + \mathbf{M}_3 + \mathbf{M}_6$$

We iterate this division process n times (recursively) until the submatrices degenerate into numbers (elements of the ring R). The resulting product will be padded with zeroes just like A and B, and should be stripped of the corresponding rows and columns.

Practical implementations of Strassen's algorithm apply standard methods of matrix multiplication for small enough submatrices, for which those algorithms are more efficient. The particular crossover point for which Strassen's algorithm is more efficient depends on the specific implementation and hardware. With the current day architecture it was observed that Strassen outperforms traditional algorithm for sizes exceeding 1000. Running time for Strassen's algorithm can be estimated as $T(n) = 7T(n/2) + \mathcal{O}(n^2)$, which by the Master Theorem works out to $\mathcal{O}(n^{\log_2 7}) \approx O(n^{2.81})$.

However, there have been a lot of research in finding matrix multiplication algorithms of better complexity, and improvements have been made not only in Strassen-like multiplication schemes but also in applying other base matrices recursively.

Table 3.1: Cost of Strassen-like multiplication schemes

Algorithms	Addition	Arithmetic computation	I/O complexity
Strassen	18	$7n^{log 7} - 6n^2$	$6\left(\frac{\sqrt{3}n}{\sqrt{M})^l og7}\right).M - 18n^2 + 3M$
Strassen-Winograd	15	$6n^{log 7} - 5n^2$	$5\left(\frac{\sqrt{3}n}{\sqrt{M}^l og7}\right).M - 15n^2 + 3M$
Karstadt-Schwartz	12	$5n^{log 7} - 4n^2 + 3n^2 log n$	$4\left(\frac{\sqrt{3}n}{\sqrt{M}^l og7}\right).M - 12n^2 + 3n^2.log\left(\frac{\sqrt{2}n}{\sqrt{M}}\right) + 5M$

cited from Karstadt Schwartz paper at SPAA'17

Table 3.2: Cost of Strassen-like multiplication schemes

Algorithms	LO	ILO	LAC	ILAC	CS
$(2,2,2;7)$	15	12	6	5	16.6%
$(3,2,3;15)$	64	52	15.06	7.94	47.3%
$(2,3,4;20)$	78	58	9.96	7.46	25.6%
$(3,3,3;23)$	87	75	8.91	6.57	26.3%
$(6,3,3;40)$	1246	202	55.63	9.39	85.2%

cited from Karstadt Schwartz paper at SPAA'17
LO- linear operations, ILO- improved linear operations, LAC-leading arithmetic coefficient, ILAC- improved linear arithmetic coefficient, CS- computation saved

Volker Strassen
Strassen was born on April 29, 1936, in Düsseldorf-Gerresheim. After studying music, philosophy, physics, and mathematics at several German universities, he received his Ph.D. in mathematics in 1962 from the University of Göttingen. He then took a position in the department of statistics at the University of California, Berkeley. He retired in 1998. Strassen began his researches as a probabilist and his paper on invariance principle has been highly cited leading to a 1966 presentation at the International Congress of Mathematicians. In 1969, Strassen shifted towards the analysis of algorithms with a paper on Gaussian elimination, introducing Strassen's algorithm, the first algorithm for performing matrix multiplication faster than the $\mathcal{O}(n^3)$ time bound for traditional algorithm. In 1971 Strassen published another paper together with Arnold Schönhage on asymptotically fast integer multiplication based on the fast Fourier transform. Strassen is also known for his 1977 work with Robert M. Solovay on the Solovay-Strassen primality test, that verifies whether a number is prime or not using randomized polynomial time.

Let us now see how D&C techniques can be used to compute the product of two degree-d polynomials. For example:
$(1+2x+3x^2)(2+x+4x^2) = 2+5x+12x^2+11x^3+12x^4$.
More generally, if $A(x) = a_0 + a_1x + ... + a_dx^d$ and $B(x) = b_0 + b_1x + ... + b_dx^d$, their product $C(x) = A(x)B(x) = c_0 + c_1x + ... + c_{2d}x^{2d}$ has coefficients ${}^{C}_{k} = \sum_{i=1}^{k} a_ib_{k-1}$ (for $i > d$, take a_i and b_i to be zero). Computing c_k from this formula takes $\mathcal{O}(k)$ steps, and finding all $2d+1$ coefficients would therefore seem to require $\theta(d^2)$ time. Is there anyway for multiplying polynomials faster than this? FFT exactly does this.
An alternative representation of polynomials To arrive at a fast algorithm for polynomial multiplication we take inspiration from an important property of polynomi-

als.
Fact A degree-d polynomial is uniquely characterized by its values at any $d+1$ distinct points. A familiar instance of this is that "any two points determine a line." A linear polynomial is uniquely determined by two points on it, in the same way a polynomial of degree d is uniquely determined by $d+1$ distinct points. Fix any distinct points $x_0, ..., x_d$. We can specify a degree-d polynomial $A(x) = a_0 + a_1x + ... + a_dx^d$ by either one of the following:

1. Its coefcients $a_0, a_1, ..., a_d$

2. The values $A(x_0), A(x_1), ..., A(x_d)$

Of these two representations, the second is the more attractive for polynomial multiplication. Since the product $C(x)$ has degree $2d$. it is completely determined by its value at any $2d+1$ points. And its value at any given point z is easy enough to figure out, just $A(z) \times B(z)$. Thus polynomial multiplication takes linear time in the value representation. The problem is that we expect the input polynomials, and their product, to be specified by coefficients. So we need to translate from coefficients to values first- which is just a matter of evaluating the polynomial at the chosen points - then multiply in the value representation, and finally translate back to coefficients, which is called interpolation.
Polynomial multiplication
Input: Coefficients of two polynomials, $A(x)$ and $B(x)$, of degree d
Output: Their product $C = AB$
Selection
Pick some points $x_0,\ x_1, \cdots,\ x_{n-1}$, where $n \geq 2d+1$
Evaluation
Compute $A(x_0),\ A(x_1), \cdots, A(x_{n-1})$ and $B(x_0),\ B(x_1), \cdots, B(x_{n-1})$
Multiplication
Compute $C(x_k) = A(x_k)B(x_k)\ \forall k = 0, \cdots,\ n-1$
Interpolation
Recover $C(x) = c_0 + c_1x + \ + c_{2d}x^{2d}$
Evaluation by divide-and-conquer: We can suitably choose n points at which to evaluate a polynomial $A(x)$ of degree $n-1$ so that positive-negative pairs, that is, $\pm x_0, \pm x_1, ..., \pm x_{\frac{n}{2}-1}$, will help us compute each $A(x_i)$ and $A(-x_i)$ with a lot of overlaps, since the even powers of x_i coincide with those of $-x_i$. To investigate this, we need to split $A(x)$ into its odd and even powers, for instance

$$3 + 4x + 6x^2 + 2x^3 + x^4 + 10x^5 = (3 + 6x^2 + x^4) + x(4 + 2x^2 + 10x^4)$$

Note that the terms in parentheses are polynomials in x^2. More generally, $A(x) = A_e(x^2) + xA_o(x^2)$, where $A_e()$, with the even-numbered coefficients, and $A_o()$, with the odd-numbered coefficients, are polynomials of degree $\leq \frac{n}{2} - 1$ (assume for convenience that n is even). Given paired points $\pm x_i$, the calculations needed for $A(x_i)$

can be used in computing $A(-x_i)$:

$$A(x_i) = A_e(x_i^2) + x_i A_o(x_i^2)$$

$$A(-x_i) = A_e(x_i^2) - x_i A_o(x_i^2)$$

In other words, evaluating $A(x)$ at n paired points $\pm x_0, \cdots, \pm x_{\frac{n}{2}-1}$ reduces to evaluating $A_e(x)$ and $A_o(x)$ (which each has half the degree of $A(x)$) at just $\frac{n}{2}$ points, $x_0, \cdots, x_{\frac{n}{2}-1}$.
The original problem of size n is in this way has been reduced to two subproblems of size $\frac{n}{2}$, followed by some linear-time arithmetic. If we could recurse, we would get a divide-and-conquer procedure with running time

$$T(n) = 2T(n/2) + \mathcal{O}(n)$$

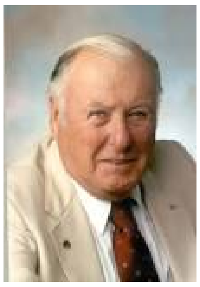
which is $\mathcal{O}(nlogn)$ exactly what we want.
Unfortunately in the next level of recursions we need $\frac{n}{2}$ evaluation points $x_0^2,\ x_1^2,\ x_2^2, \cdots, x_{\frac{n}{2}-1}^2$ which will be plus-minus pairs, where a square will be negative. The task is possible only if we use complex numbers. Here comes nth roots of unity with all the nice properties. Repeated square rooting of x_i's will result in complex numbers, and there will be no problem in evaluation. So complexity of evaluation can be expressed as

$$T(n) = 2T(\frac{n}{2}) + \mathcal{O}(n),$$

which is $\mathcal{O}(nlogn)$, exactly what we want. The next level up then has $\pm$ as well as the complex numbers $\pm\sqrt{-1} = \pm i$, where i is the imaginary unit. By continuing in this manner, we eventually reach the initial set of n points. Perhaps you have already guessed what they are: the complex nth roots of unity, that is, the n complex solutions to the equation $z^n = 1$. The nth roots of unity: the complex numbers $1, \omega^1, \omega^2, \cdots, \omega^{n-1}$, where $\omega = e^{\frac{2\pi}{n}}$. If n is even,
1. The nth roots are plus-minus paired, $\omega^{\frac{n}{2}+j} = \omega^j$.
2. Squaring them produces the $(\frac{n}{2})$ roots of unity. Therefore, if we start with these numbers for some n that is a power of 2, then at successive levels of recursion we will have the $\frac{n}{2^k}$th roots of unity, for $k = 0, 1, 2, 3, \cdots,$. All these sets of numbers are plus-minus paired, and so our divide-and-conquer, as shown in the last panel, works perfectly. The resulting algorithm is the fast Fourier transform (Figure 2.7).
The Fast Fourier Transform (polynomial formulation) function $FFT(A, \omega)$
Input: Coefficient representation of a polynomial $A(x)$ of degree $\leq n-1$, where n is a power of $2, \omega$, an nth root of unity.
Output: Value representation $A(\omega^0), ..., A(\omega^{n-1})$
if $\omega = 1$ then return $A(1)$ else express $A(x)$ in the form $A_e(x^2) + xA_0(x^2)$
call $FFT(A_e, \omega^2)$ to evaluate A_e at even powers of ω
call $FFT(A_o, \omega^2)$ to evaluate A_o at even powers of ω
for $j = 0$ to $n-1$ do
compute $A(\omega^j) = A_e(\omega^{2j}) + \omega^j A_0(\omega^{2j})$

return $A(\omega^0), \cdots, A(\omega^{n-1})$

History of Fast Fourier Transform In 1963, during a meeting of President Kennedy's scientific advisors, John Tukey, a mathematician from Princeton, explained to IBM's Dick Garwin a fast method for computing Fourier transforms. John Cooley was requested to implement Tukey's algorithm. They decided that a paper should be published so that the idea could not be patented. And this is how one of the most famous and most cited scientific papers was published in 1965, co-authored by Cooley and Tukey.



James William Cooley
James William Cooley (born 1926, died June 29, 2016)[1] was an American mathematician. Cooley received a B.A. degree in 1949 from Manhattan College, Bronx, NY, an M.A. degree in 1951 from Columbia University, New York, NY, and a Ph.D. degree in 1961 in applied mathematics from Columbia University. He was a programmer on John von Neumann's computer at the Institute for Advanced Study, Princeton, NJ, from 1953 to 1956. He worked on quantum mechanical computations at the Courant Institute, New York University, from 1956 to 1962, when he joined the Research Staff at the IBM Watson Research Center, Yorktown Heights, NY. Upon retirement from IBM in 1991, he joined the Department of Electrical Engineering, University of Rhode Island, Kingston, where he served on the faculty of the computer engineering program. His most significant contribution to the world of mathematics and digital signal processing is the Fast Fourier transform, which he co-developed with John Tukey (see CooleyTukey FFT algorithm) while working for the research division of IBM in 1965. The motivation for it was provided by Dr. Richard L. Garwin at IBM Watson Research who was concerned about verifying a Nuclear arms treaty with the Soviet Union for the SALT talks. Garwin thought that if he had a very much faster Fourier Transform he could plant sensors in the ground in countries surrounding the Soviet Union. He suggested the idea of how Fourier transforms could be programmed to be much faster to both Cooley and Tukey. They did the work, the sensors were planted, and he was able to locate nuclear explosions to within 15 kilometers of where they were occurring. J.W. Cooley was a member of the Digital Signal Processing Committee of the IEEE, and was later awarded a fellowship of IEEE for his work on FFT. In 2002 he received the IEEE Jack S. Kilby Signal Processing Medal.[2][3] He considerably contributed to the establishing of terminology in digital signal processing.

John Wilder Tukey
John Wilder Tukey (/tuki/;[2] June 16, 1915 July 26, 2000) was an American mathematician best known for development of the FFT algorithm and box plot.[3] The Tukey range test, the Tukey lambda distribution, the Tukey test of additivity, and the TeichmüllerTukey lemma all bear his name. Among many contributions to civil society, Tukey served on a committee of the American Statistical Association that produced a report challenging the conclusions of the Kinsey Report, Statistical Problems of the Kinsey Report on Sexual Behavior in the Human Male. He was awarded the National Medal of Science by President Nixon in 1973.[5] He was awarded the IEEE Medal of Honor in 1982 "For his contributions to the spectral analysis of random processes and the fast Fourier transform (FFT) algorithm."

***(Picture of Cooley and Tukey)
The reason Tukey was reluctant to publish the FFT was not secretiveness or pursuit of profit via patents. He just felt that this was a simple observation that was probably already known. This was typical of the period: back then (and for some time later) algorithms were considered second-class mathematical objects, devoid of depth and elegance, and unworthy of serious attention. But Tukey was right about one thing: it was later discovered that British engineers had used the FFT for hand calculations during the late 1930s. It may be mentioned here that a paper of Gauss in the early 1800s on (what else?) interpolation contained essentially the same idea in it! Gauss' s paper had remained a secret for so long because it was protected by an old-fashioned cryptographic technique: like most scientific papers of its era, it was written in Latin.

3.4 Exercises

1. We studied Euclid's algorithm for computing the greatest common divisor (gcd) of two positive integers: the largest integer which divides them both. Here we will look at an alternative algorithm based on divide-and-conquer.

   (a) Show that the following rule is true.

$$gcd(a,b) = \begin{cases} 2gcd(\frac{a}{2},\frac{b}{2}) & \text{if } a,b \text{ are even,} \\ gcd(a,\frac{b}{2}) & \text{if } a \text{ is odd and } b \text{ is even,} \\ gcd(\frac{(a-b)}{2},b), & \text{if } a,b \text{ are odd.} \end{cases} \tag{3.3}$$

   (b) Give an efficient divide-and-conquer algorithm for greatest common divisor.

   (c) How does the efficiency of your algorithm compare to Euclid's algorithm if a and b are n-bit integers? (In particular, since n might be large, you cannot assume that basic arithmetic operations like addition take constant time.)

2. Given an array A a pair of numbers $A[i], A[j]$ is said to be inversion if for $i < j$, $A[i] > A[j]$. Design a subquadratic algorithm for counting the number of inversions in an array.

3. Given a bitonic sequence, the task is to find Bitonic Point in it. A Bitonic Sequence is concatenation of two subsequences one of which is increasing and the other decreasing. A Bitonic Point is the minimum or maximum element of the array.

4. Given an array of distinct positive integers, design a subquadratic algorithm to find the maximum product of increasing subsequence of size 3, i.e., we need to find i, j, k $i < j < k$, $A[i] < A[j] < A[k]$ such that $A[i].A[j].A[k]$ is the maximum for all possible values of $i < j < k$.

5. Given an array of size n such that array elements are in range from 1 to n. Design an algorithm to count the number of move-to-front operations to arrange items as $\{1, 2, 3, n\}$. The move-to-front operation is to pick any item and place it at first position. Your algorithm should solve the problem in sub-quadratic time.

   This problem can also be seen as a stack of items with only move available is to pull an item from the stack and placing it on top of the stack.

6. In this problem we consider a monotonously decreasing function $f(N) \to Z$ taking integer values, that is $f(i) > f(j)$ whenever $i < j$. Assuming that the value of the function at any point can be evaluated in constant time find

$$\min\{i | f(i) \leq 0\}$$

   The following problem is MSCS of chapter 1.

7. **The maximum partial sum problem** (*MPS*)

   Given an array $A[1, \cdots, n]$ of integers, find values of i and j with $1 \leq i \leq j \leq n$ such that $\sum_{k=i}^{k=j} A[k]$ is maximized. Example:

   For the array $[4, -5, 6, 7, 8, -10, 5]$, the solution to MPS is $i = 3$ and $j = 5$ (sum 21). To help us design an efficient algorithm for the maximum partial sum problem, we consider the left position l maximal partial sum problem ($LMPS_l$). This problem consists of finding value j with $l \leq j \leq n$ such that $\sum_{k=l}^{k=j} A[k]$ is maximized, and similarly, the right position r such that $\sum_{k=j}^{k=r} A[k]$ is maximized. Example: For the array $[5, -4, 7, 8, 9, -9, 6]$ the solution to e.g. $LMPS_4$ is $j = 5$(sum 17) and the solution to $RMPS_7$ is $i = 3(sum 21)$. (a) Describe $\mathcal{O}(n)$ time algorithms for solving $LMPS_l$ and $RMPS_r$ for given l and r.

   (b) Using an $\mathcal{O}(n)$ time algorithm for $LMPS_l$, describe a simple $\mathcal{O}(n^2)$ algorithm for solving MPS.

   (c) Using $\mathcal{O}(n)$ time algorithms for $LMPS_l$ and $RMPS_r$, describe an $\mathcal{O}(n \log n)$ divide- and-conquer algorithm for solving MPS.

8. Let us have an array $A[1..n]$ of sorted integers circularly shifted k positions to the right. For example, $[25, 42, 5, 15, 17, 19]$ is a sorted array that has been circularly shifted $k = 2$ positions, while $[17, 19, 25, 42, 5, 15]$ has been shifted $k = 4$ positions. We can obviously find the largest element in A in $\mathcal{O}(n)$ time. Describe an $\mathcal{O}(logn)$ algorithm.

Problems to include in different chapters

1. Given an array of integers, find the pair which has minimum XOR value. An Efficient solution can solve the above problem in $\mathcal{O}(\backslash)$ time under the assumption that integers take fixed number of bits to store. The idea is to use Trie Data Structure.

2. Given a sorted array of n elements containing elements in range from 1 to $n - 1$ and exactly one element occurs twice, the task is to find the repeating element of the array.

3. Given an array of integers, the task is to find count of elements before which all the elements are smaller. First element is always counted as there is no other element before it.

4. Given an array with repeated elements, the task is to find the maximum distance between two occurrences of an element.

   hint: Hashing

5. Given a sequence of positive numbers, find the maximum sum that can be formed which has no three consecutive elements present.

edited this far.

1. Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest, Introduction to Algorithms (MIT Press, 2000).
2. Brassard, G. and Bratley, P. Fundamental of Algorithmics, Prentice-Hall, 1996.
3. Anany V. Levitin, Introduction to the Design and Analysis of Algorithms (Addison Wesley, 2002).
4. Donald E. Knuth, The Art of Computer Programming: Volume 3, Sorting and Searching, second edition (Addison-Wesley, 1998).
5. Heideman, M. T., D. H. Johnson, and C. S. Burrus, "Gauss and the history of the fast Fourier transform," IEEE ASSP Magazine, 1, (4), 1421 (1984)
6. Knuth, Donald (1998). The Art of Computer Programming: Volume 3 Sorting and Searching. p. 159. ISBN 0-201-89685-0.
7. Karatsuba, Anatolii A.; Yuri P. Ofman (1962). " ". Doklady Akademii Nauk SSSR 146: 293 -294. Translated in Physics-Doklady 7: 595 -596. 1963. Missing or empty |title= (help)
8. M. Frigo; C. E. Leiserson; H. Prokop (1999). "Cache-oblivious algorithms". Proc. 40th Symp. on the Foundations of Computer Science.
9. Nicholas J. Higham, "The accuracy of floating point summation", SIAM J. Scientific Computing 14 (4), 783 - 799 (1993).

10. Frigo, M.; Johnson, S. G. (February 2005). "The design and implementation of FFTW3". Proceedings of the IEEE 93 (2): 216 - 231. doi:10.1109/JPROC.2004.840301.
11. Radu Rugina and Martin Rinard, "Recursion unrolling for divide and conquer programs," in Languages and Compilers for Parallel Computing, chapter 3, pp. 34 - 48. Lecture Notes in Computer Science vol. 2017 (Berlin: Springer, 2001). Skiena, Steven S. (1998), "ğ8.2.3 Matrix multiplication", The Algorithm Design Manual, Berlin, New York: Springer-Verlag, ISBN 978-0-387-94860-7. D'Alberto, Paolo; Nicolau, Alexandru (2005). Using Recursion to Boost ATLAS's Performance (PDF). Sixth Int'l Symp. on High Performance Computing. Webb, Miller (1975). "Computational complexity and numerical stability". SIAM J. Comput: 97107. Burgisser, Clausen, and Shokrollahi. Algebraic Complexity Theory. Springer-Verlag 1997. Frigo, M.; Leiserson, C. E.; Prokop, H.; Ramachandran, S. (1999). Cache-oblivious algorithms (PDF). Proc. IEEE Symp. on Foundations of Computer Science (FOCS). pp. 285297.
Higham, Nicholas J. (1990). "Exploiting fast matrix multiplication within the level 3 BLAS" (PDF). ACM Transactions on Mathematical Software. 16 (4): 352368. doi:10.1145/98267.98290. Strassen, Volker, Gaussian Elimination is not Optimal, Numer. Math. 13, p. 354-356, 1969 Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. Introduction to Algorithms, Second Edition. MIT Press and McGraw-Hill, 2001. ISBN 0-262-03293-7. Chapter 28: Section 28.2: Strassen's algorithm for matrix multiplication, pp. 735741.

Chapter 4

Greedy Techniques

4.1 Introduction

If we want to find a solution that optimizes certain criterion then greedy techniques appear very appealing and intuitive, where decision is taken at every step that looks for the best. While such simple an approach may not lead to the best solution there are problems where this simple strategy indeed leads to finding the best solution. Let us have the following coin-changing problem: There are denominations of 1.00 dollar, 50, 20, 10, 5 and 1 cents. We need to make changes using minimum number of denominations.

To make any amount we just try to satisfy it with the highest denominations. Say, if we have to make 3.88 we give 3 1 dollar notes, 1 $50-$cent, 1 $20-$cent, 1 $10-$cent 1 $5-$cent and 3 $1-$coin. Every time we have chosen the largest denomination for the exchange. This is indeed the minimum number of denominations for the change. Nobody can have a better solution although this solution picked up notes very greedily only. However, there are examples where this greedy pick up may not end up in an optimal solution. To be convinced that greedy choice may not lead to the optimal solution even in examples similar to the one mentioned above.

Now we have a $9-$cent coin in stead of $5-$cent. Greedy strategy will make 3.80 greedily only to be forced to take 8 more coins to make 3.88. So number of coins will be 14 coins. whereas we have a better solution by stopping greedy selection at 3.70 and then pick up 2 $8-$cents amounting to only 7 coins one less than the greedy strategy.

In the notorious Travelling Salesman Problem(TSP) in which a salesman wants to visit a given set of cities and returns to home city traversing minimum distance or in minimum time or at minimum cost a greedy criterion could be : visit the nearest (requiring minimum time or cost) unvisited city. However good the strategy may look like there are examples in which it will not find the best solution. Look at the following example in which one has to visit all the points

$\{(0,\ 0), (0,\ -1), (0,\ 2), (0,\ -4),\ (0,\ 7), (0,\ -12), (0,\ 27)\}$

and come back to the origin.

draw the points on number line

In this example, visiting the nearest unvisited city will force you to visit cities located

in different sides of the point of origin like
$\{(0,\ 0),(0,\ -1),(0,\ 2),(0,\ -4),\ (0,\ 7),(0,\ -12),(0,\ 27)\}$ with a cost of $1+3+6+11+19+39+27=106$. Whereas visiting cities on one side of the point of origin $\{(0,\ 0),(0,\ -1),(0,\ -4),\ (0,\ -12)\}$ and then visiting the cities on the otherside $\{(0,\ 2),\ (0,\ 7),(0,\ 27)\}$ and then back to $(0,0)$ will cost $12+27+27=66$, and will give us a tour of smaller length! However, there are many problems in which greedy techniques will give the best possible solution.

4.2 Examples

In this section we list some of the problems that are amenable to greedy solutions, that is, greedy techniques lead to optimal solution. Some examples are given below:

4.2.1 Fractional Knapsack

A bush walker is preparing his knapsack to fill up with available items to optimize his benefits given weight he can carry. Given a set of items with profits and weights $(p_i, w_i), i = 1, \cdots, n$, and a weight bound W what are the items to be selected to maximize profit? An item can be chosen fractionally as well, like amount of food and so on. Choosing large p_is will not do for which counterexample is $\{(10,8),(1,3),(5,7),(1,2),W = 10\}$ since they may exhaust weight bound quickly. In this case we must remain satisfied with a profit of $11\frac{3}{7}$. Neither should choosing small w_is will do since corresponding profits may also be unnecessarily small. In this case profit is $5\frac{4}{7}$. The winning criterion is choosing items based on large values of $\frac{p_i}{w_i}$. In this case profit will be $11\frac{3}{7}$. So there may be more than one appealing greedy criteria. However, a detailed investigation can only isolate the optimal one. This can be shown that in fractional knapsack problem there is an optimal solution in which at most one object will be chosen fractionally.

4.2.2 Storing Files in Tapes

Let us have n files of length $l_i,\ i = 1, \cdots, n$. Store them in a tape so that total retrieval time is as small as possible. It may be noted that tape is a sequential access storage device, and retrieving ith file will necessitate passing through all the i files. One greedy criterion may be storing the smallest file first. Let file of length l_{i_j} be stored in jth order, Then total retrieval time will be given by

$$\sum_{j=1}^{n}\sum_{k=1}^{j} l_{i_j} = \sum_{j=1}^{n}(n-j+1)l_{i_j}$$

So we are minimizing sum of products of n pairs of nonnegative numbers. The first n-tuple is $l_i,\ i = 1, \cdots, n$, and the other n-tuple is $n, n-1, n-2, \cdots, 1$. This minimum occurs when larger numbers are multiplied by smaller numbers. So length of the smallest file is multiplied by n, second smallest by $n-1$ and in this way the

largest file length simply by 1. This can be seen immediately by observing that if a larger file of length l_{i_j} appears earlier than a smaller file of length l_{i_k} with $j < k$ then swapping their position will result in the reduction of total retrieval time by appears This means that files should be stored in ascending order of their length. So the problem transforms into a sorting one. In Chapter 3 we will see that sorting problem can be solved in $\mathcal{O}(n \log_2 n)$ computation.
Optimality of the solution can be proved by contradiction. Assume that in actual optimal solution file of length l_{i_j} is stored in kth sequence whereas file of length l_{i_k} is stored in jth sequence. Now cost associated with these two files in the fictitous optimal solution is

$$l_{i_j}(n-k+1) + l_{i_k}(n-j+1),\ j < k.$$

Now if these files are swapped then the cost will be

$$l_{i_j}(n-j+1) + l_{i_k}(n-k+1)$$

Difference of the first cost from the second is

$$l_{i_j}(n-k+1) + l_{i_k}(n-j+1) - l_{i_j}(n-j+1) - l_{i_k}(n-k+1) = (l_{i_j} - l_{i_k})(j-k) > 0$$

This indicates that by storing smaller file earlier retrieval time can be reduced. So long as larger files are stored earlier we can continue to improve the solution.
On the other hand, it is known that given two arrays $a_i,\ b_i,\ i = 1, \cdots, n$ of positive numbers

$$\sum_{i=1}^{n} a_i b_{j_i}$$

is maximum when both the arrays are sorted in ascending order, whereas minimum occurs when one is sorted in ascending order and the other one in descending order. In our case given two arrays $(n-j+1)$, and $l_{i_j}, j = 1, , n$ of positive numbers sum of products is minimum when one is in ascending order and the other is in descending order. Greedy criterion does exactly do it. So the solution must be optimal. One can also prove it by assuming contradiction that a larger file of length l_i is placed before a smaller file l_j.

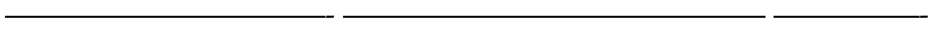
4.2.3 Two-way Merging of Files

Given a set of files of length $l_i, i = 1, \cdots, n$ file the order in which they should be merged into a single file where cost of merging is proportional to the sum total of length of files to be merged. A greedy criterion could be to merge the smallest two files so that increase of cost is as small as possible. Let us take one example to demonstrate the algorithm. Let there be files of length $\{2, 3, 7, 7, 10, 21, 34, 11, 17\}$. The greedy criterion requires that we must find the two smallest files, delete them from the list and add the length of the merged file. For doing that efficiently we need heap as a data structure in which finding requires $\mathcal{O}(1)$ while deleting and adding requires $\mathcal{O}(logn)$ computation. The solution has been depicted in the following graph, where

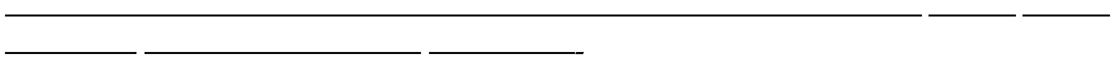
original files correspond to external nodes, whereas merged files correspond to internal nodes. Cost of merging has been shown in the internal nodes.
**** A binary tree *****

4.2.4 Scheduling Jobs in a Processor

Given a set of jobs with start and finish times $(s_i, f_i), i = 1, \cdots, n$, our job is to schedule as many jobs as possible. One greedy criterion may be choosing jobs requiring lesser time. Counter example is the following:



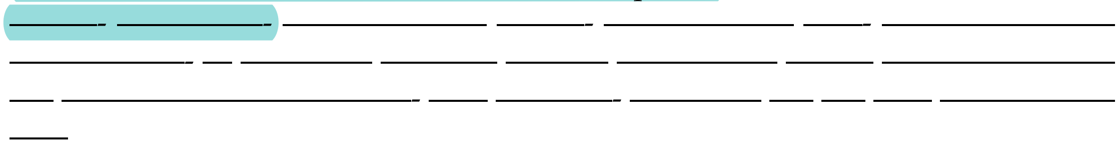
OK, then let us schedule jobs with the earliest start to keep processor busy as early as possible. Alas! Again there is a counter example.



But the winning criterion is the earliest finish time! It helps us process a job and make the process free at the earliest. It stays ahead of other strategies. Whatever other strategies may be we can process other jobs if job with the earliest process time is processed.

4.2.5 Scheduling Lectures in Minimum no. of Class Rooms

Given a set of lectures l_i with start and finish times (s_i, f_i) schedule them using minimum number of class rooms. Lectures are not necessarily of the same duration. Starting with lecture with the earliest start we should start scheduling it in a class. Now disregarding lectures that cannot be scheduled in that class room again select the lecture having earliest start among the remaining ones. Once classroom 1 has been scheduled completely start with class room 2 in the same manner with the lectures not yet scheduled. This will result in a schedule that will require as many class rooms as many lectures will have to be scheduled at a particular time. It can be easily shown that no schedule can do better since if at a particular moment there are m lectures then at least m classrooms will be required.



4.2.6 Job Sequencing with Deadlines

There is a set of n jobs with deadlines $(d_i \geq 0)$ and profits $(p_i > 0$. For any job i profit p_i will only be earned if it can be executed by its deadline. To complete a job it has to be processed on a machine for one unit of time. Only one machine is available for processing jobs. A feasible solution to this problem is a set J of jobs that can be processed by their deadlines. The profit from these jobs is $\sum_{i \in J} p_i$. An optimal solution is the feasible solution with maximum profit. We consider the time

slots $[i-1;i], 1 \leq i \leq b$, such that $b = min\{n, max_i\{d[i]\}\}$. We use i to represent the time slot $[i-1;i]$.
Again as before we can think of many greedy criteria. The one that will lead to the optimal solution is the following. First sort the jobs in descending order of profit. Then try to process each job at its deadline. If that slot of time unit is already occupied by some other job, then find an empty slot to its left. If it cannot be found then the job cannot be processed. Continue doing so with other jobs so long as some time slot is still unoccupied. For each job in worst case we may have to search for an empty slot sequentially to the left. Since there are n jobs and for each job sequential search may lead to $\mathcal{O}(n)$ computation the algorithm becomes $\mathcal{O}(n^2)$. We can improve this algorithm if an empty slot before an occupied slot can be found faster. We have an wonderful data structure for this job. This data structure can be used in other places as well to improve order of computation! This is called *sets and disjoint set union*. Initially each slot is represented by a subset, and its root contains the address of the empty slot before it. Each subset contains a number of consecutive occupied slots, and as usual root of each set contains the address of the immediate left empty slot. When two subsets merge together, because the only empty slot between these two subsets become occupied, its root will contain information as to the empty slot before the merged subset. To implement this we need to FIND the immediate slot no later than the deadline of the job i we wish to process. Once this job is processed at a time slot j we would like to update the empty slot appearing before the slot d_i. Let $S(d_i)$ be the subset d_i is in, then we must join subsets $S(d_i)$ and $S(j)$. Since subsets are represented by father pointers so that starting at any slot i we can quickly find the name of the subset $S(i)$ it belongs to, we can come to know the rightmost empty slot before the deadline d_i. It can be shown that when joining two subsets it is better to make root of the larger subset father of the root of the smaller subset so that height of the tree grows slowly. On the other hand during FIND operation if we need to traverse too many father links we can attach all those nodes directly to the root so that at a later time we are not penalized for searching the root of the nodes on the path. This is called FIND with path compression.
Algorithm $FJS(d, n, b, j)$
// Find an optimal solution $J[1:k]$. Let us assume that $p[1] \leq p[2] \leq ... \leq p[n]$ and that $b = min\{n, max_i\{d[i]\}$ //Initially there are $b+1$ single node trees. Let $p[i]$ be the empty slot appearing on or before slot i for $i = 0$ to b do $p[i] = i$ $k = 0$ for $i = 1$ to n do $q = CollapsingFind(min(n, d[i]))$ if $p[q] \neq 0$ then $k = k+1, J[k] = i$ //select job i $m = CompressingFind(p[q]-1)$ $BalancedUnion(m, q)$ $p[q] = p[m]$ q may be the new root. endif enddo
Here are the algorithms:
Algorithm BalancedUnion(i,j)
// Union sets with roots i and j, $i \neq j$, using weighting rule. $p(i) = -count(i)$ and $p(j) = -count(j)$ temp=p(i)+p(j) If p(i)>p(j) then //i has fewer nodes p(i)=j, p(j)=temp else // j has fewer nodes p(j)=i, p(i)=temp endif
Algorithm CompressingFind(i)
//Find the root of the tree containing element i. Attach all nodes on the path to root

directly to root by redirecting father pointer r=i while (p(r)>0) do r=p(r) enddo While (i $\neq$ r) do s=p(i), p(i)=r, i=s enddo

In 1975 Robert Tarjan was the first to show through inverse Ackermann function that even when number of slots is as high as

$$2^{2^{2^{2^{16}}}}$$

the number of father links to reach the root will not exceed 4, meaning that FIND operation is practically constant for all sizes of sets we can think of.

Robert E Tarjan

Robert Endre Tarjan (born April 30, 1948) is an American computer scientist and mathematician. He is the discoverer of several graph algorithms, including Tarjan's off-line lowest common ancestors algorithm, and co-inventor of both splay trees and Fibonacci heaps. Tarjan is currently the James S. McDonnell Distinguished University Professor of Computer Science at Princeton University, and the Chief Scientist at Intertrust Technologies. While he was in high school, Tarjan got a job, where he worked IBM card punch collators. He first worked with real computers while studying astronomy at the Summer Science Program in 1964.

Tarjan obtained a Bachelor's degree in mathematics from the California Institute of Technology in 1969. At Stanford University, he received his master's degree in computer science in 1971 and a Ph.D. in computer science (with a minor in mathematics) in 1972. At Stanford, he was supervised by Robert Floyd and Donald Knuth, both highly prominent computer scientists, and his Ph.D. dissertation was An Efficient Planarity Algorithm. Tarjan selected computer science as his area of interest because he believed that CS was a way of doing mathematics that could have a practical impact. Tarjan is known for his pioneering work on graph theory algorithms and data structures. Some of his well-known algorithms include Tarjan's off-line least common ancestors algorithm, and Tarjan's strongly connected components algorithm, and he was one of five co-authors of the median of medians linear time selection algorithm. The Hopcroft-Tarjan planarity testing algorithm was the first linear-time algorithm for planarity-testing.

Tarjan has also developed important data structures such as the Fibonacci heap (a heap data structure consisting of a forest of trees), and the splay tree (a self-adjusting binary search tree; co-invented by Tarjan and Daniel Sleator). Another significant contribution was the analysis of the disjoint-set data structure; he was the first to prove the optimal runtime involving the inverse Ackermann function. Tarjan received the Turing Award jointly with John Hopcroft in 1986.

4.2.7 Huffman Coding

There are two types of coding- Fixed or block codes and variable length codes. Block codes are easy to decode since every k bits represent a codeword whereas variable lengths are a little cumbersome to decode. In order to be able to uniquely decode variable length codes it is easier when no codeword appears as a prefix to another codeword. If codewords appear only as external nodes of a binary tree then this property is achieved since all prefixes of any codeword correspond to internal nodes that do not correspond to any codeword. Now the task is to devise a variable length coding scheme that will result in minimum total code length.

David Albert Huffman
David Albert Huffman (August 9, 1925 October 7, 1999) was a pioneer in computer science, known for his Huffman coding. Huffman, a graduate student of MIT, was to solve this as a term paper. After studying for months together when he was about to leave it undone and concentrate on term final he got the idea and devised the algorithm that is being used in high-definition television, modems and a popular electronic device that takes the brain work out of programming a video cassette recorder. He was also one of the pioneers in the field of mathematical origami. David Huffman died at the age of 74, ten months after being diagnosed with cancer.

This ingenuine algorithm is also a greedy one. Let there be word $i, i = 1, ..., n$ with frequency f_i. Our task is to minimize

$$\sum_{i=1}^{n} f_i length(code(i))$$

As we have remarked earlier such a weighted sum will be minimized if words of higher frequency have code lengths not higher than the ones with lower frequency. Huffman represented each word by an external node and joined two external nodes of the smallest frequencies f_i and f_j to an internal node k with frequency $f_k = f_i + f_j$. we must at the same time delete the two frequencies and insert their sum. These operations can be efficiently done in heaps. So whatever code is assigned to k will be appended by 0 to obtain code for word i and 1 to obtain code for j. This will be done recursively until we are left with a single node, the root, with frequency equal to sum of frequencies of all external nodes. This tree is called **Huffman tree**.
Let us use the same frequencies as used in two-way merging of files. $(b, d, e, f, h, i, a, g) = (2, 3, 34, 7, 10, 21, 11, 17)$. The Huffman tree has been constructed below, and codes corresponding to letters have been shown in the following table.
Note that the same tree was generated in case of two-way merging of files- same algorithm for solving two different problems!!! This is a two-pass algorithm since for constructing the tree we need to collect frequencies of each symbol in the text to be coded. Once the tree has been constructed we can write down the codes of each letter by appending 0s if we go left from the root to reach the external node corresponding to the letter and 1 if we move to the right. Unfortunately in many cases like sending news from distant places to a daily newspaper office we may not have the whole text at a time, and therefore cannot send it as it comes whereas news value decreases with the time elapsed from having the news and getting it published. So there is a one-pass algorithm for the problem. This algorithm is based on sending

ht

Table 4.1: Huffman tree calculation

letters	frequencies	codes	codelengths
b	2	001	3
d	2	001	3
e	2	001	3
f	2	001	3
h	2	001	3
i	2	001	3
a	2	001	3
g	2	001	3

$i + 1$st letter on the basis of optimal Huffman tree on the first i letters already sent. After sending the $i + 1$st letter the tree is modified to be optimum for the first $i + 1$ letters. This modification of the tree is done both by the sender and receiver using the same algorithm so that at all times the trees at both ends are synchronized. The trees are grown with the conditions that nodes with higher frequencies appear not in lower levels and not to the right of nodes having smaller frequencies. We must also note that since at any time Huffman tree corresponds to the first i letters. So there must be a node that will correspond to all unsent letters. All such letters will be represented by that node, and to recognize a particular letter we must append the code corresponding to that node by a known code like ASCII, EBCDIC or Unicode. Once a new letter appears that particular node is attached to two nodes one of which represents the new letter and the other for all yet unsent letters. In the following diagrams we show how a tree for the message 'adabcabcbdbcdcdcdd' is sequentially constructed.

4.2.8 Minimum Spanning Tree

Given a connected graph $G = (V, E)$ with weight w_i on edge e_i finding the spanning tree of minimum weight. Spanning tree is an acyclic subgraph. A greedy criterion could be choosing smallest edge so that it does not form cycle with already selected edges. This strategy indeed gives optimal solution and the algorithm is named after Kruskal. Kruskal's algorithm is a greedy algorithm in graph theory that finds a minimum spanning tree for a connected weighted graph. This means it finds a subset of the edges that forms a tree that includes every vertex, where the total weight of all the edges in the tree is minimized. If the graph is not connected, then it finds a minimum spanning forest (a minimum spanning tree for each connected component). This algorithm first appeared in Proceedings of the American Mathematical Society, pp. $48 - 50$ in 1956, and was written by Joseph Kruskal. Other algorithms for this problem include Prim's algorithm, Reverse-Delete algorithm, and Borúvka's algorithm. Description

create a forest F (a set of trees), where each vertex in the graph is a separate tree

create a set S containing all the edges in the graph

while S is nonempty and F is not yet spanning remove an edge with minimum weight from S if that edge connects two different trees, then add it to the forest, combining two trees into a single tree otherwise discard that edge. At the termination of the algorithm, the forest has only one component and forms a minimum spanning tree of the graph. $G = (V, E)$, $T = \emptyset, F = E$

While F is not empty

choose edge e with minimum weight

if $T \cup \{e\}$ does not form a cycle

$T \cup \{e\}$

endif

$F = F \setminus \{e\}$

Enddo Here main challenge is to test for cycle which has to be done very efficiently. Sets and disjoint set union algorithm does this job in a great way. The algorithm was developed in 1930 by Czech mathematician Vojtěch Jarn'ik and later independently by computer scientist Robert C. Prim in 1957 and rediscovered by Edsger Dijkstra in 1959. Therefore, it is also sometimes called the DJP algorithm, the Jarník algorithm, or the Prim-Jarník algorithm.

Robert Clay Prim
Robert Clay Prim (born September 25, 1921 in Sweetwater, Texas) is an American mathematician and computer scientist.
In 1941, Prim received his B.S. in Electrical Engineering from The University of Texas at Austin, where he also met his wife Alice (Hutter) Prim (1921 – 2009), whom he married in 1942. Later in 1949, he received his Ph.D. in Mathematics from Princeton University, where he also worked as a research associate from 1948 until 1949.
During the climax of World War II (19411944), Prim worked as an engineer for General Electric. From 1944 until 1949, he was hired by the United States Naval Ordnance Lab as an engineer and later a mathematician. At Bell Laboratories, he served as director of mathematics research from 1958 to 1961. There, Prim developed Prim's algorithm. Also during his tenure at Bell Labs, Robert Prim assisted the Weapons Reliability Committee at Sandia National Laboratory chaired by Walter McNair in 1951. After Bell Laboratories, Prim became vice president of research at Sandia National Laboratories.
During his career at Bell Laboratories, Robert Prim along with coworker Joseph Kruskal developed two different algorithms (see greedy algorithm) for finding a minimum spanning tree in a weighted graph, a basic stumbling block in computer network design. His self-named algorithm, Prim's algorithm, was originally discovered in 1930 by mathematician Vojtěch Jarn'ik and later independently by Prim in 1957. It was later rediscovered by Edsger Dijkstra in 1959. It is sometimes referred to as the DJP algorithm or the Jarn'ik algorithm.

In this algorithm starting from an arbitrary vertex(termed connected and remaining disconnected) vertex nearest to the connected set is sequentially added updating information on the nearest vertex in the connected set from each member of the disconnected set. The only spanning tree of the empty graph (with an empty vertex set) is again the empty graph. The following description assumes that this special case is handled separately. The algorithm continuously increases the size of a tree, one edge at a time, starting with a tree consisting of a single vertex, until it spans all vertices.
Input: A non-empty connected weighted graph with vertices V and edges E (the weights can be negative). Initialize: $V_{new} = \{x\}$, where x is an arbitrary node (starting point) from V, $E_{new} = \emptyset$
Repeat until $V_{new} = V$:
For each vertex not in V_{new} find nearest vertex in V_{new}
Choose an edge (u, v) with minimal weight such that u is in V_{new} and v is not (if

there are multiple edges with the same weight, any of them may be picked) Add v to V_{new}, and (u, v) to E_{new} Update nearest array Output: V_{new} and E_{new} describe a minimal spanning tree

Prim's algorithm

Time complexity

Minimum edge weight data structure Time complexity (total) adjacency matrix, searching $\mathcal{O}(V^2)$ binary heap and adjacency list $\mathcal{O}((V + E)logV) = \mathcal{O}(ElogV)$ Fibonacci heap and adjacency list $\mathcal{O}(E + VlogV)$ A simple implementation using an adjacency matrix graph representation and searching an array of weights to find the minimum weight edge to add requires $\mathcal{O}(V^2)$ running time. Using a simple binary heap data structure and an adjacency list representation, Prim's algorithm can be shown to run in time $\mathcal{O}(ElogV)$ where E is the number of edges and V is the number of vertices. Using a more sophisticated Fibonacci heap, this can be brought down to $\mathcal{O}(E + VlogV)$, which is asymptotically faster when the graph is dense enough that E is $\Sigma(V)$.

Proof of correctness

The proof consists of two parts. First, it is proved that the algorithm produces a spanning tree. Second, it is proved that the constructed spanning tree is of minimal weight.

Spanning Tree

Let $G = (V, E)$ be a connected, weighted graph and let T be the subgraph of G produced by the algorithm. We cannot have a cycle, since the last edge added to that cycle would have been within one subtree and not between two different trees. cannot be disconnected, since the first encountered edge that joins two components of would have been added by the algorithm. Thus, T is a spanning tree of G.

Minimality

We show that the following proposition **P** is true by induction: If F is the set of edges chosen at any stage of the algorithm, then there is some minimum spanning tree that contains F.

Clearly **P** is true at the beginning, when F is empty: any minimum spanning tree will do, and there exists one because a weighted connected graph always has a minimum spanning tree.

Now assume **P** is true for some non-final edge set F and let T, be a minimum spanning tree that contains F. If the next chosen edge e is also in T, then **P** is true for $F + e$. Otherwise, T+ e has a cycle C and there is another edge f that is in C but not F. (If there were no such edge f, then e could not have been added to F, since doing so would have created the cycle C.) Then $T - f + e$ is a tree, and it has the same weight as T, since T, has minimum weight and the weight of f cannot be less than the weight of e, otherwise the algorithm would have chosen f instead of e. So $T - f + e$ is a minimum spanning tree containing $F + e$ and again P holds.

Therefore, by the principle of induction, **P** holds when F has become a spanning tree, which is only possible if F is a minimum spanning tree itself.

4.2.9 Mincost Arborescence Problem

Given a digraph $G = (V, A)$ and a root vertex $r \in V$, an r -arborescence (or an arborescence with root r) is a subset of arcs $B \in A$ such that for each vertex $v \in V \setminus \{r\}$, there is a directed path from r to v. Let $\delta_{in}(v)$ be set of edges entering into vertex v. There is an equivalent way to define arborescences:
Lemma 1. The set $B \in A$ is an r -arborescence in a digraph $G = (V, A)$ if and only if (V, B) has no cycles, and $|B \cap in(v)| = 1$ for each $v \in V \setminus \{r\}$.
Consider the directed graph with directed weighted edges as

$$w_{ra} = 10,\ w_{rb} = 2,\ w_{rc} = 10,\ w_{ab} = 1,\ w_{bc} = 4,\ w_{ad} = 2,\ w_{cd} = 2,\ w_{af} = 8,\ w_{cf} = 4$$

***** The digraph should be drawn *********
We would like to remind that constructing a minimum spanning tree on underlying graph will not serve the purpose since

1. a minimum spanning tree will never include the largest edge of any cycle but mincost arborescence may contain. For example, the solution to the above problem consists of edges rb, bc, cd, da, ef and bc is the largest edge in the cycle $abcd$.

2. A minimum spanning tree always contains the smallest edge of any cutset but in case of mincost arborescence this might not be true. For example, for the cutset with vertices $\{r, b, c\}$ in one partition and the remaining in the other ab is the smallest edge but arborescence does not contain it, rather it contains all other larger edges.

Lemma 2. A digraph G contains an r -arborescence if and only if each vertex in G is reachable from r.
Proof. If each vertex in G is reachable from r, the breadth-first search (started at r) will find an r -arborescence. The converse is trivial. We consider the following minimum cost r-arborescence problem:
Given a digraph $G = (V, A)$, a root vertex $r \in V$, and a cost function $c : f(A) \to R$, find an r -arborescence of minimum cost in G.
The relationship between arborescences and spanning trees might give an impression that the same techniques can be applied here, too. The following algorithm is a complete analogue of the Prim's method for constructing minimum spanning trees:

1. Set $U = \{r\}, B = \emptyset$;

2. While $U \neq V$;

3. find $a = (u, v)|u \in U, v \in \delta_{out}(U)$ with $c(a) = min\{c(a')|a' \in \delta_{out}(U)\}$

4. set $U := U \cup \{v\}, B = B \cup \{a\}$;

5. output B.

Unfortunately, this algorithm fails on a rather trivial digraph (draw the graph with weights): $c(r,v) = 2$, $c(r,u) = 3$, $c(u,v) = 1$, since execution of the algorithm will result in selecting edges (r,v), (r,u) in this order with the cost 5. But mincost arborescence will correspond to (r,u), (u,v) costing only 4. Nonetheless, it is still possible to design a "greedy-type" algorithm to find a minimum cost r -arborescence. ***

We can also follow: https://www.cs.princeton.edu/ wayne/kleinberg-tardos/pdf/ 04GreedyAlgorithmsII.pdf *****

First, for each $v \in V \setminus \{r\}$, select a cheapest arc $a \in \delta_{in}(v)$ (with ties broken arbitrarily); let B^* be the set of those arcs. Then B^* is a cheapest set of arcs satisfying $|\delta_{in}(v) \cap B^*| = 1$ for all $v \in V \setminus \{r\}$. In particular, $c(B^*) \leq c(B)$ for any r -arborescence B, since B must also have $|\delta_{in}(v) \cap B^*| = 1$ (see Lemma 1). Now, if (V, B^*) does not contain a cycle, then B^* is an r -arborescence itself (again, see Lemma 1), and hence a minimum cost r-arborescence. Some problems arise when B^* does contain a cycle C. Since B^* contains cycle, and there are only $n - 1$ arcs in it, (V, B^*) must be disconnected. Let us consider the component of B^* not containing r. Since r cannot be included in the arborescence, this component must have a directed cycle C, and we must have an arc a entering into this component, in particular, into the vertex v. Since we do not need more than one arc into a vertex, the arc in B^* entering into v can be deleted. In fact wwe need to insert that arc for which increase in weight is the minimum. In order to select the arc that will minimize increase of weight of arborescence we do the following: Let $\alpha(v) = min\{c(a) | a \in \delta_{in}(v)\}$. Do the following:

1. $c'(a) = c(a) - \alpha(v) | a \in \delta_{in}(v) \ \forall v \in C$
2. Construct G' by merging all vertices of C into a single vertex. Update weights of arcs as above.
3. Solve mincost arborescence problem for G'.

This procedure should continue until the arborescence becomes cycleless. Then cycles should be exploded in order, and suitable arcs should be included in the arborescence.

This is an example of a problem in which at each step decision taken does not necessarily constitute a part of optimal soution, which can only be known at the end.

4.3 Exercise

1. **k-way merge** Let us allow k-way merge and assume that cost of merging is proportional to sum total of lengths of files to be merged. How are we going to merge the files to minimize our cost? If we are allowed to vary k what is the best value of k for k-way merging?
2. **Minimum Number of Platforms Required for a Railway/Bus Station:** Given arrival and departure times of all trains that reach a railway station, devise a

greedy algorithm for finding the minimum number of platforms required for the railway station so that no train waits.

3. **Rearrange a string so that all same characters become d distance away:** Given a string, with possible repetation of characters, and a positive integer d rearrange characters of the string such that the same characters become d distance away from each other. The solution may not be unique.

4. **Bin Packing Problem:** Given n items of different weights and bins each of capacity c being not lesser than weights of any item, assign each item to a bin such that number of total used bins is minimized.

5. **Connect n Ropes with Minimum Cost:** Given n ropes of different lengths, we need to connect these ropes into one rope using minimum cost. The cost to connect two ropes is equal to sum of their lengths.

6. **Minimize Cash Flow among Friends:**

   Given a number of friends with lending borrowing relationship design an algorithm by which the total cash flow among all the friends is minimized.

7. **Greedy Algorithm for Egyptian Fraction:**

   Every positive fraction can be represented as sum of unique unit fractions. A fraction is unit fraction if numerator is 1 and denominator is a positive integer. For example $\frac{1}{3}$ is a unit fraction. Design a greedy algorithm for representing any fraction as sum of minimum number of unit fractions.

8. **K Centers Problem:** Given n cities and distances between every pair of cities, devise an algorithm for selecting k cities to place warehouses (or ATMs or Cloud Server) such that the maximum distance of a city to a warehouse (or ATM or Cloud Server) is minimized.

9. **Shortest Superstring Problem:** Given a set of n strings $A[]$, deduce an algorithm for finding the smallest string that contains each string in the given set as substring. We may assume that no string in $A[]$ is a substring of another string.

10. **Completing Jobs with Constraints:** Given an array of jobs with different time requirements for completion. There are n identical assignees available with time requirment for solving each job. Find the minimum time to finish all jobs with the following constraints. An assignee can be assigned only adjacent jobs. For example, an assignee cannot be assigned jobs 1 and 3, but not 2. Two assignees cannot be allocated the same job.

11. **Set Cover Problem:** Given a universe U of n elements and a collection of subsets of U say $S = \{S_1, S_2, ..., S_m\}$ where a cost is associated with every subset S_i. Deduce an approximation algorithm for finding a minimum cost subcollection of S that covers all elements of U.

Chapter 5

Sorting Algorithms

Sorting is a problem from ancient times. We sort a file for finding an element faster, for selecting a set of numbers satisfying certain conditions like the largest or the smallest k numbers as is required in any admission test or selection. Most often sorting of data improves running time of a variety of algorithms. Sorting is so widely used in solving problems efficiently that many sorting algorithms have been devised. And still now efforts are being made to imprve sorting algorithms.

5.1 Important Sorting Algorithms

5.1.1 Bubble Sort

One of the early algorithms is *bubble sort* in which in the ith iteration ith largest element is found always comparing a pair of adjacent elements and pushing the larger element down all the time.

Table 5.1: Bubble Sort

On the ith column largest of the first i rows is placed on the ith row						
11	11	11	11	11	11	11
15	**15**	13	13	13	13	13
13	13	**15**	15	15	15	15
17	17	17	**17**	14	14	14
14	14	14	14	**17**	12	12
12	12	12	12	12	**17**	17

Bubble sort was analyzed as early as in 1956.

5.1.2 Mergesort

An early two-subproblem Divide and Conquer based (D&C) algorithm that was specifically developed for computers and properly analyzed is the merge sort algorithm, invented by John von Neumann in 1945. The array is divided into 2 equal parts and sorted individually. Then both of them are merged to obtain a single sorted array. The routine goes as follows

```
Mergesort(A,1,n)
Low=1, high=n
If low<high then
    merge(A,low,mid,high)
    mergesort(A,low,mid)
    mergesort(A, mid+1,high)
    merge(A,low,mid,high)
endif
merge(A,low,mid,high)
i=low, j=mid+1, k=low
while i<=mid and j<=high do
    mid=(low+high)/2
    if A(i)<A(j) then
        B(k)=A(i), i++
    else
        B(k)=A(j), j++
    endif
    k++
enddo
If i<mid then
    for k=i to mid do
        B(k)=A(k)
    enddo
Else
    for k=j to high do
        B(k)=A(k)
    enddo
endif
for i=low to high do
    A(i)=B(i)
enddo
```

Complexity: Let $T(n)$ be number of comparisons $n = 2^k$

$$T(n) = 2T(\frac{n}{2}) + an = an + 2a\frac{n}{2} + 2^2T(\frac{n}{2^2}) = (an + an + \cdots + an)k = akn = an\log_2 n$$

Mergesort requires too much of overhead in terms of I/O operations and is not an in-place sorting algorithm. An array of 1 million elements will require an an extra of the same size to get sorted using mergesort algoithm. However, mergesort using linked list implementation will not require as much of memory. In the following section we have an in-place D&C based algorithm for sorting.

5.1.3 Quicksort

Quicksort is an in-place sorting algorthm that sorts data in place and that is based on D&C strategy. Pseudocode of the algorithm goes as follows with partition routine playing the crucial role.

```
Quicksort(A,1,n)
partition(A,low,high,j)
Low=1, high=n z=A(low), i=low+1, j=n+1, A(n+1)=infinity
If low<high then
    partition(A,low,high,j)
    quicksort(A,low,j-1)
    quicksort(A,j+1,high)
Endif
partition(A,low,high,j)
while i<j do
    while A(i)<z do
        i++
    enddo
    while A(j)>z do
        j- -
    enddo
    if i<j then
        A(i)A(j)
    endif
enddo
```

If partitioning element is eccentric, too small or too large then one subarray will be almost empty. In each call number of comparisons will be equal to number of elements in the array. In particular if one subarray is empty total number of comparisons will be $n + (n-1) + ... + 1 = \mathcal{O}(n^2)$. But this algorithm of Hoare works much faster on average than the above complexity suggests. Let us do average case analysis assuming that partitioning element can occupy any position with equal probability. Let $C_A(n)$ be average number of comparisons. Let partitioning element be the kth smallest element. Then there will be $k-1$ elements smaller and $n-k$ elements larger.

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

$$nC_A(n) = n^2 + 2\sum_{k=1}^{n} C_A(k-1) \tag{5.1}$$

Putting $n+1$ in place of n we get

$$(n+1)C_A(n+1) = (n+1)^2 + 2\sum_{k=1}^{n+1} C_A(k-1) \tag{5.2}$$

Subtracting the former from the later we get

$$(n+1)C_A(n+1) - nC_A(n) = 2n + 1 + 2C_A(n)$$

Or

$$(n+1)C_A(n+1) - nC_A(n) = 2n + 1 + 2C_A(n)$$
$$(n+1)C_A(n+1) - (n+2)C_A(n) = 2n + 1$$
$$\frac{C_A(n+1)}{n+2} - \frac{C_A(n)}{n+1} = \frac{2n+1}{(n+1)(n+2)}$$

$$C_A(n+1) = \frac{C_A(n)}{n+1} + \frac{2}{n+2} \tag{5.3}$$

Recursively we get

$$\frac{C_A(n+1)}{n+2} = 2(\frac{1}{n+2} + \frac{1}{n+1} + ... + \frac{1}{3} + \frac{C_A(1)}{2}) = 2H_{n+2}$$

Hence $C_A(n) = 2(n+1)H_{n+1}$

Sir Charles Anthony Richard Hoare
Sir Charles Antony Richard Hoare FRS FREng, commonly known as Tony Hoare or C. A. R. Hoare (Born: January 11, 1934, Colombo, Sri Lanka), is a British computer scientist. Tony Hoare's interest in computing was awakened in the early fifties, when he studied philosophy (together with Latin and Greek) at Oxford University, under the tutelage of John Lucas. In 1959, as a graduate student at Moscow State University, he studied the machine translation of languages (together with probability theory, in the school of Kolmogorov). To assist in efficient look-up of words in a dictionary, he discovered the well-known sorting algorithm Quicksort in 1959/1960.

For quicksort to be more efficient we need to have a good partitioning element that can partition the array into nontrivial subarrays. Median of medians is a good example that exactly does that. If we have 49 elements that are divided into 7 subarrays of 7 elements, and all these subarrays are arranged in such a way that their medians are placed in ascending order then median of medians will be greater/less than at least 15 elements. Now this median of medians can be used as a partitioning element. This partitioning algorithm can be used to select any number of elements of an array in probable linear time.

Multi-pivot quicksort

In Multi-pivot quicksort instead of partitioning into two subarrays using a single pivot, partition into some s number of subarrays using $s-1$ pivots is performed.

The dual-pivot case ($s = 3$) considered by Sedgewick and others already in the mid-1970s, could not be found faster in practice than the "classical" quicksort. However, Kaykobad et al mentions of some experiments where dual-pivot quicksort outperformed the traditional one. (Computers & Mathematics with Applications Volume 36, Issue 6, September 1998, Pages 19-24

3 is a more promising algorithmic parameter than 2 MKaykobad Md.MIslam M.EAmyeen M.MMurshed)

Yaroslavskiy also reports in 2009 a version of dual-pivot quicksort that turned out to be fast enough when implemented in Java 7.

5.1.4 Heapsort

Heap is a data structure efficient for deletion of the maximum(minimum) element in $\mathcal{O}(\log n)$ computation, insertion of an element also requires operations of similar order. Heap is implemented in a complete binary tree in which nodes are located top to bottom and left to right. If we number nodes from 1 to n with 1 being the root, then number of a node is twice that of its father if it happens to be left son or $+1$ if right. Similarly if number of a node is divided by 2 then we get number of the father. Heap is a data structure implemented in a complete binary tree. Values at father node is not smaller than that of son nodes. We can construct heap in two different ways- by insertion and by adjustment.

```
Insertintoheap(A,n,z)//inserting z into a heap having n-1 elements
A(n)=z, A(0)=infinity
While z>A(n/2) do
A(n)A(n/2), n=n/2
enddo
Heapbyadjustment(A,n)
For i=n/2 downto 1 do
adjust(A,i)
Enddo
Adjust(A,i)
```

Construct chain of elder sons Insert A(i) in this sorted chain of elder sons using binary search in a file of length

Cost of heap construction

What is the least number of comparisons in the worst case to be required by any comparison-based sorting algorithm?

There are $n!$ possibilities for sorted sequence. So a binary decision tree must have n! leaves. Average height of the tree, using Stirling's approximation, must be

$$\log_2 n! \approx \log_2 \sqrt{2\pi n}\left(\frac{n}{e}\right)^n \approx n(\log_2 n - \log_2 e)$$

5.1.5 Pancake Sorting

Pancake sorting is to sort a disordered stack of pancakes in order of size when a spatula can be inserted at any point in the stack and used to flip all pancakes above it. A *pancake number* is the maximum number of flips required to sort them in order f size for a given number of pancakes. In this form, the problem was first discussed by American geometer Jacob E. Goodman[1]. It is a variation of the sorting problem in which the only allowed operation is to reverse the elements of some prefix of the sequence. Unlike a traditional sorting algorithm, which attempts to sort with the fewest number of comparisons possible, the goal is to sort the sequence in as few reversals as possible. A variant of the problem is concerned with burnt pancakes, where each pancake has a burnt side and all pancakes must, in addition, end up with the burnt side on bottom. This is equivalent to sorting an unsorted array using the following operation.

flip(arr, i): Reverse array from 0 to i

Unlike a traditional sorting algorithm, which attempts to sort with the fewest comparisons possible, the goal is to sort the sequence in as few reversals as possible.

The idea is to do something similar to Selection Sort. We one by one place maximum element at the end and reduce the size of current array by one.

Following are the detailed steps. Let given array be arr[] and size of array be n. 1) Start from current size equal to n. and reduce current size by one while it is greater than 1. Let the current size be currentsize. Do following for every currentsize a) Find index of the maximum element in arr[0..currentsize-1]. Let the index be mi b) Call flip(arr, mi) c) Call flip(arr, currentsize-1)

brief historical note

On September 17, 2008, a team of researchers at the University of Texas at Dallas led by Founders Professor Hal Sudborough announced the acceptance by the journal Theoretical Computer Science of a more efficient algorithm for pancake sorting than the one proposed by Bill Gates and Christos Papadimitriou. This establishes a new upper bound of $\frac{18}{11}n$, improving upon the existing bound of $\frac{5}{3}n$ from 1979.

On November 2, 2011, a paper was submitted to the arXiv announcing a proof that the problem is NP-hard, thereby answering a question open for over three decades. It is worth noting, however, that the NP-hard problem consists of computing the minimum number of flips required to sort n pancakes, and not the actual sorting of the pancakes. As discussed above, the sorting can be trivially computed in $\mathcal{O}(n)$ (see Big O notation) time, which places it in the polynomial class of problems.

5.2 Performance of different sorting algorithms

P-partitioning, M-merging, S-selection, I-insertion, E-exchanging

Table 5.2: Performance of different comparison-based sorting algorithms

Name	Best	Average	Worst	Memory	Stable	Method
Quicksort	$n \log n$	$n \log n$	n^2	$\log n$	N	P
Merge Sort	$n \log n$	$n \log n$	$n \log n$	n worst case	Y	M
In-place Merge Sort	-	-	$n \log^2 n$	1	Y	M
Heapsort	$n \log n$	$n \log n$	$n \log n$	1	No	S
Insertion Sort	n	n^2	n^2	1	Yes	I
Introsort	$n \log n$	$n \log n$	$n \log n$	$\log n$	N	PS
Selection sort	n^2	n^2	n^2	1	N	Se
Timsort	n	$n \log n$	$n \log n$	n	Y	M
Cubesort	n	$n \log n$	$n \log n$	n	Y	I
Shellsort	n	$n \log^2 n$ or $n^{\frac{3}{2}}$	1	No	I	
Bubble sort	n	n^2	n^2	1	Y	E
Binary tree sort	n	$n \log n$	$n \log n$ (balanced)	n	Y	I
Cycle sort	-	n^2	n^2	1	N	I
Library sort	-	$n \log n$	n^2	n	Y	I
Smoothsort	n	$n \log n$	$n \log n$	1	N	S
Strand sort	n	n^2	n^2	n	Y	S
Tournament sort	—	$n \log n$	$n \log n$	n		S
Cocktail sort	n	n^2	n^2	1	Y	E
Comb sort	n	$n \log n$	n^2	1	N	E
Selection						

5.3 Recent Advancements

5.3.1 Recent progress

The significance of sorting and searching has been very duly recognized by Don Knuth by dedicating one of the volumes of "The Art of Computer programming" to it. While sorting is a very widely studied problem still a lot of research is going on and newer and more efficient algorithms are being discovered. Theoretical computer scientists have detailed other sorting algorithms that provide better than $\mathcal{O}(n \log n)$ time complexity assuming additional constraints. These include

Han's algorithm, a deterministic algorithm for sorting keys from a domain of finite size, takes $\mathcal{O}(n \log\ logn)$ time and $\mathcal{O}(n)$ space. Thorup's algorithm, a randomized algorithm for sorting keys from a domain of finite size, takes $\mathcal{O}(n \log \log n)$ time and $\mathcal{O}(n)$ space. A randomized integer sorting algorithm taking $\mathcal{O}(n\sqrt{\log \log n})$ expected time and $\mathcal{O}(n)$ space. Some of the sorting algorithms are as follows: Quicksort, mergesort, in-place-mergesort, heapsort, insertion sort, introsort, bubble sort, selection sort, timsort, cube sort, etc. In addition to it there are some non-comparison based sort like bucket sort, radix sort, pigeonhole sort etc. Some of the areas of research are listed below:

Table 5.3: Performance of different noncomparison-based sorting algorithms

Name	Best	Average	Worst	Memory	Stable	Method
Pigeonhole sort	-	$n + 2^k$	$n + 2^k$	2^k Yes	Yes	
Bucket Sort	-	$n + k$	n^2k	$n.k$	Yes	No
Counting Sort	-	$n + r$	$n + r$	$n + r$	Yes	Yes
MSD Radix sort	-	$n.\frac{k}{d}$	$n.\frac{k}{d}$	$n + 2^d$	Yes	No

5.3.2 Generalized Sorting

Given a set of elements for sorting, and a subset of pairs that can be compared to sort all the elements the goal is to determine the sorted order using minimum number of comparisons. This is called generalized sorting[7]. The problem can also be described in a directed graph to preserve a Hamiltonian path by deleting as many directed edges as possible.

5.3.3 Optimal Sorting

This studies the minimum number of comparisons required to sort a sequence of n elements in the worst case. Lower bound theory suggests that any comparison based sorting algorithm will require at least $\log_2 n!$ comparisons in the worst case. In the *optimal algorithm* as Knuth names, elements are divided into roughly equal parts and then optimal algorithm is run on each of them. Then elements of one sorted sequence are inserted into the other sorted sequence ensuring that the element is inserted in $2^k - 1$ elements so that k comparisons will suffice to identify the position.

5.3.4 Sorting by rounds

In this variation a number of pairs can be compared in a single round. Based upon the results of comparisons another set of pairs can be compared. This can be continued until position of each element in the sorted array is known. For example, if we want to sort in a single round then $\binom{n}{2}$ comparisons are necessary and sufficient. But if we want to sort in more rounds lesser number of comaprisons will suffice.

5.3.5 Lower Bound Theory

Process of Comparison based sorting algorithms can be depicted as trees. The one in the following figure sorts an array of three elements, a_1, a_2, a_3. It starts by comparing a_1 to a_2 and, if the first is larger, compares it with a_3; otherwise it compares a_2 and a_3. And so on. Eventually we end up at a leaf, and this leaf is labeled with the true order of the three elements as a permutation of $1, 2, 3$. For example, if $a_2 < a_1 < a_3$, we get to the leaf labeled "2 1 3".
****(Draw the figure of DPV page 63) ****

The depth of the tree is the number of comparisons on the longest path from root to leaf, in this case 3 is exactly the worst-case time complexity of the algorithm.
This way of looking at sorting algorithms is useful because it allows one to argue that mergesort is optimal, in the sense that $\Omega(nlogn)$ comparisons are necessary for sorting n elements.
Here is the argument: Consider any such tree that sorts an array of n elements. Each of its leaves is labeled by a permutation of $\{1, 2, ..., n\}$. In fact, every permutation must appear as the label of a leaf. The reason is simple: if a particular permutation is missing, what happens if we feed the algorithm an input ordered according to this same permutation? And since there are $n!$ permutations of n elements, it follows that the tree has at least $n!$ leaves.
We are almost done: This is a binary tree, and we argued that it has at least $n!$ leaves. Recall now that a binary tree of depth d has at most 2^d leaves. So, the depth of our tree and the complexity of our algorithm must be at least $log(n!)$. And it is well known that $\log(n!) \geq cnlogn$ for some $c > 0$. There are many ways to see this. The easiest is to notice that $n! \geq (n/2)^{n/2}$ because $n! = 1 \times 2 \times \cdots \times n$ contains at least $n/2$ factors larger than $n/2$, and then take logs of both sides. Another is to recall Stirlings formula

$$n! \approx \sqrt{2\pi n}\left(\frac{n}{e}\right)^n \Rightarrow \log n! \approx n(\log_2 n - \log_2 e) = \mathcal{O}(n \log n)$$

Either way, we have established that any comparison tree that sorts n elements must make, in the worst case, $\Omega(n \log n)$ comparisons, and hence mergesort is optimal! The argument is valid for only comparison based algorithms. Note that there are algorithms that use more sophisticated operations that can solve the problem in linear time. One example is that of sorting integers lying in a small range.
****(Exercise 2.20).****

5.3.6 References

5.3.7 Exercise

1. What happens when Quicksort sorts an array having all equal elements?

2. (a) Prove that the following algorithm actually sorts its input.

```
STOOGESORT($A[0..n1]$)
if $n = 2$ and $A[0] > A[1]$ then
swap $A[0] \leftrightarrow A[1]$
else if n > 2 then
$m = \frac{2n}{3}$
endif
STOOGESORT($A[0..m1]$)
STOOGESORT($A[nm..n1]$)
STOOGESORT($A[0..m1]$)
```

   (b) Would STOOGESORT still sort correctly if we replaced $m = \frac{2n}{3}$ with $m = \frac{n}{3}$? Justify your answer.

   (c) State a recurrence (including the base case(s)) for the number of comparisons executed by STOOGESORT.

   (d) Solve the recurrence, and prove that your solution is correct. [Hint: Ignore the ceiling.]

   (e) Prove that the number of swaps executed by STOOGESORT is at most $\binom{n}{2}$.

3. Describe an algorithm to compute the median of an array $A[1..5]$ of distinct numbers using at most 6 comparisons. Instead of writing pseudocode, describe your algorithm using a decision tree: A binary tree where each internal node is a comparison of the form $A[i] < A[j]$? and each leaf is an index into the array.

4. An inversion in an array $A[1..n]$ is a pair of indices (i, j) such that $i < j$ and $A[i] > A[j]$. The number of inversions in an n-element array is between 0 (if the array is sorted) and $n - 1$ (if the array is sorted backward). Describe and analyze an algorithm to count the number of inversions in an n-element array in $\mathcal{O}(nlogn)$ time. [Hint: Modify mergesort.]

5. (a) Suppose we are given two sorted arrays $A[1..n]$ and $B[1..n]$ and an integer k. Describe an algorithm to find the kth smallest element in the union of A and B in (log n) time. For example, if $k = 1$, your algorithm should return the smallest element of $A \cup B$; if $k = n$, your algorithm should return the median of $A \cup B$.) You can assume that the arrays contain no duplicate elements. [Hint: First solve the special case $k = n$.]

   (b) Now suppose we are given three sorted arrays $A[1..n]$, $B[1..n]$, and $C[1..n]$, and an integer k. Describe an algorithm to find the kth smallest element in A B C in $\mathcal{O}(logn)$ time.

   (c) Finally, suppose we are given a two dimensional array $A[1..m][1..n]$ in which every row $A[i][]$ is sorted, and an integer k. Describe an algorithm to find the kth smallest element in A as quickly as possible. How does the running time of your algorithm depend on m? [Hint: Use the linear-time SELECT algorithm as a subroutine.]

6. You are at a political convention with n delegates, each one a member of exactly one political party. It is impossible to tell which political party any delegate belongs to; in particular, you will be summarily ejected from the convention if you ask. However, you can determine whether any two delegates belong to the same party or not by introducing them to each othermembers of the same party always greet each other with smiles and friendly handshakes; members of different parties always greet each other with angry stares and insults.

   (a) Suppose a majority (more than half) of the delegates are from the same political party. Describe an efficient algorithm that identifies a member (any member) of the majority party.

(b) Now suppose exactly k political parties are represented at the convention and one party has a plurality: more delegates belong to that party than to any other. Present a practical procedure to pick a person from the plurality political party as parsimoniously as possible. (Please.)

[1]Simon Singh (November 14, 2013). "Flipping pancakes with mathematics". The Guardian.
[2]Gates; W.; Papadimitriou; C. (1979). "Bounds for Sorting by Prefix Reversal" (PDF). Discrete Mathematics 27: 4757. doi:10.1016/0012-365X(79)90068-2. [3]"Team Bests Young Bill Gates With Improved Answer to So-Called Pancake Problem in Mathematics". University of Texas at Dallas News [4]Center. September 17, 2008. Retrieved November 10, 2008. "A team of UT Dallas computer science students and their faculty adviser have improved upon a longstanding solution to a mathematical conundrum known as the pancake problem. The previous best solution, which stood for almost 30 years, was devised by Bill Gates and one of his Harvard instructors, Christos Papadimitriou, several years before Microsoft was established." [5]Chitturi, B.; Fahle, W.; Meng, Z.; Morales, L.; Shields, C.O.; Sudborough, I.H.; Voit, W. (2009). "An upper bound for sorting by prefix reversals". Theoretical Computer Science 410 (36): 3372. doi:10.1016/j.tcs.2008.04.045. [6]Bulteau, Laurent; Fertin, Guillaume; Rusu, Irena (2012). "Pancake Flipping Is Hard". Mathematical Foundations of Computer Science 2012. Lecture Notes in Computer Science 7464 : 247.$arXiv$: 1111.0434.doi : $10.1007/978-3-642-32589-2_24.ISBN978-3-642-32588-5$. [7] Zhiyi Huang and Sampath Kannan and Sanjeev Khanna, Algorithms for the Generalized Sorting Problem, 52nd Annual IEEE Symposium on Foundations of Computer Science, (FOCS11), 2011. [8] Demuth, H. Electronic Data Sorting. PhD thesis, Stanford University, 1956. [11]Han, Y. (January 2004). "Deterministic sorting in O(n log log n) time and linear space". Journal of Algorithms 50: 96105. doi:10.1016/j.jalgor.2003.09.001. edit [12]Thorup, M. (February 2002). "Randomized Sorting in O(n log log n) Time and Linear Space Using Addition, Shift, and Bit-wise Boolean Operations". Journal of Algorithms 42 (2): 205230. doi:10.1006/jagm.2002.1211. edit [13] Yijie Han; Thorup, M. (2002). Integer sorting in $O(n\sqrt(\log\log n))$ expected time and linear space. FOCS 2002. The 43rd Annual IEEE Symposium on Foundations of Computer Science, 2002. Proceedings: 135144.doi : 10.1109/$SFCS$.2002.1181890.$ISBN0-7695-1822-2$.

Chapter 6

Graph Algorithms

$G = (V, E)$ is a graph on set of vertices V and set of edges E. Each member of E consists of a pair of members of V, and we write $e = (u, v)$. If this pair is ordered then the graph is said to be *directed graph* or in short *digraph*.Otherwise it is called an undirected graph or simply a graph. If G does not have a cycle then it is called acyclic. Occasionally, we write $|V| = n,\ |E| = m$. A graph is connected if for every pair u, v of its vertices there is a sequence of edges that start at u and end at v with start vertex of an edge is th same as end vertex of the preceding edge if any. $H = (V, E)$ is a subgraph of G if V is a subset of V and E is the subset of E, endpoints of each member of which are in V'. A path from vertex u_1 to u_p is a subgraph $P = (V_P, E_P)$ on vertices $V_P = \{u_1,\ u_2, ..., u_p\}$, and edges $E_P = \{(u_i, u_{i+1})|i = 1, ..., p-1$. An undirected graph is connected if every pair of vertices is joined by a path. If G is connected and does not have a cycle then it is called a tree. Number of edges incident to a vertex u is said to be degree of u and is denoted by $d(u)$. A directed graph is strongly connected if every pair of vertices are reachable from each other. An undirected connected graph is called *simple*, if between every pair of vertices there is at most one edge, and no vertex contains a self loop (i.e. a vertex connected to itself by an edge).

A graph is called *bipartite* if the vertex set V can be partitioned into two subsets $S \cup T = V,\ S \cap T = \emptyset$, such that for any edge $(u, v) \in E\ u \in S$ and $v \in T$. A bipartite graph is *complete* if every vertex in S is adjacent to each vertex in T. For example, K_{mn} refers to a complete bipartite graph consisting of vertices $V = S \cup T$, where $|S| = m$ and $|T| = n$. A graph $G = (V, E)$ is called *planar* if it can be drawn in a plane in such a way that edges do not intersect. Interestingly $K_{3,3}$ is the smallest graph (in terms of edges) which is non-planar.

Kuratowski's Theorem: A graph is planar if and only if it has no subgraph homeomorphic to K_5 or $K_{3,3}$. Two graphs are homeomorphic if both can be obtained from the same graph by a sequence of subdivisions of edges (insertion of a vertex on an edge). For example, any two fundamental cycles are homeomorphic.

A connected acyclic subgraph of G is called a *spanning tree*. A maximally connected subgraph of a graph is said to be a *component*. Occasionally a weight is attached to each edge of a graph. In an unweighted graph, the cost of a path is just the number of edges on the shortest path, which can be found in $\mathcal{O}(n + m)$ time via breadth first search(BFS), whereas cost of a path in a weighted graph is sum total of the weights

of edges in the path. BFS will not necessarily result in constructing minimum paths in a weighted graph. Sometimes visiting more edges can lead to a shorter distance. Note that there can be an exponential number of shortest paths between two nodes - so we cannot report all shortest paths in polynomial time.
Note that existence of negative cost cycles in a path renders the problem of finding the shortest path meaningless, since you can always keep on looping around the negative cost cycle to reduce the cost of the path as much as you wish. Thus in our discussions, we will assume that all edge weights are positive. Other algorithms deal correctly with negative cost edges. Minimum spanning trees are unaffected by negative cost edges.
We first start with an algorithm for traversing the vertices of a graph using breadth-first search(BFS). BFS starts from a vertex and continues to explore vertices as they are discovered.
BFS can be used for both directed or undirected graphs. Here we focus on BFS on directed graphs.
bf BFS: Given $G = (V, E)$ and a distinguished vertex $s \in V$, discover all vertices reachable from s, and compute their distances from s. Let

$$\rho(v) = \begin{cases} \text{the minimum number of edges (arcs) on a shortest path from } s \text{ to } v \\ \infty, \text{ if there is no path from } s \text{ to } v \end{cases} \tag{6.1}$$

Observe, this is a special case of the shortest path problem for unit arc lengths, where length $l(uv) = 1$ for each edge $uv \in E$
Breadth first search (BFS)

```
BFS(G,s)
for each vertex $v \in V$ do
{$d(v) \leftarrow \infty; p(v) \leftarrow NIL$} // parent of $v$ //
$d(s) \leftarrow 0$
enqueue($Q, s$) // $Q$=FIFO queue //
while $Q$ is not empty do
$u \leftarrow head(Q)$ // at the start, $u = s$ //
for each vertex $v$ in $ADJ(u)$ do
if $d(v) = \infty$ // untouched vertex //
{$d(v) \leftarrow d(u) + 1; p(v) \leftarrow u; enqueue(Q, v)$}
$dequeue(Q)$ // delete head of $Q$ //
enddo
```

picture from lecture 13 page 7
It is easy to see that this algorithm runs in $\mathcal{O}(|V| + |E|)$ time since each vertex is inserted (enqueued) once in the queue Q and each edge (x, y) is explored twice, once when the vertex x is dequeued from Q and once when the vertex y is dequeued. Insertion and Deletion of a vertex in Q can be achieved in constant time since Q is a FIFO Queue, and can be maintained as a doubly-connected list. Also adjacency list

representation will suffice and the correctness is left as an assignment problem. We can summarize this as follows:
Theorem. Let $G = (V, E)$ be a simple graph. A breadth-first search traversal of G, and its corresponding tree, can be computed in $\mathcal{O}(|V| + |E|)$ time.
Depth first search (DFS) on a directed graph
- Global variable: $time = 1, 2, ..., 2|V|$- After termination, for each $v \in V, d(v) =$ discovery time of v, and $f(v) =$ finishing time of v
- During the algorithm:
$d(v) = \infty$ means v has not been yet discovered

```
DFS(G)
for each vertex $v \in V$ do
{$d(v) \leftarrow \infty; p(v) \leftarrow NIL$} // $p(v)$=parent of $v$ //
$time \leftarrow 0$
for each vertex $u \in V$ do
if $d(u) = \infty$ //untouched vertex // then
$VISIT(u)$
endif
enddo
enddo
```

```
VISIT(u)
// recursive procedure here //
1. $time \leftarrow time + 1$
2. $d(u) \leftarrow time$ // $u$ has just been discovered //
3. for each vertex $v$ in $ADJ(u)$ do
4. if $d(v) = \infty$ then // untouched vertex //
5. {$p(v) \leftarrow u; VISIT(v)$}
6. $time \leftarrow time + 1$
7. $f(u) \leftarrow time$
```

(example from l13 p15)
DFS: discovery and finishing times We will use DFS to compute the Strongly Connected Components(SCC) of a directed graph. For every $v \in V$, consider the interval $I_v = [d(v), f(v)]$ (recall that $d(v) < f(v)$). Example: $[1, 6], [2, 5], [3, 4], [7, 8], [9, 12], [10, 11]$.
Question: What relations exist among all these intervals?
Classifying arcs with DFS G: directed graph, G_π: DFS forest for G(π denote the parent pointers). Categories of arcs:
1. Tree arcs (or edges): uv is a tree arc if uvG_π.
2. Forward arcs(or edges): uv is a forward arc if v is a descendant of u in G_π, but not a direct descendant (not a tree edge), i.e., $I_vI_u, d(v)d(u) + 2$.
3. Back arcs(or edges): uv is a back arc if u is a descendant of v in a DFS tree, i.e., $I_u \in I_v$.
4. Cross arcs(or edges): all other edges, either between different trees, or in the same

tree but none of u, v is an ancestor of the other, i.e., $I_u \cup I_v = \emptyset$
(picture from l14 p6)
A linear-time algorithm for computing SCCs
1. Call DFS(G) to compute the finishing times $f(v)$ for all $v \in V$.
2. Renumber the vertices v in order of decreasing $f(v)$ (max. finishing time first).
3. Compute G_r, reversed graph of G. Run $DFS(G_r)$ (consider vertices in decreasing order of $f(v)$).
4. Output each tree in $DFS(G_r)$ as a separate strongly connected component.

(picture from l14 p9)
Topological Sort
Topological Sort A directed acyclic graph (or DAG) is a directed graph with no cycles. Theorem Let $G = (V, E)$ be an acyclic graph. Then there exists a linear ordering $\phi : V \to \{1, 2, ..., |V|\}$ such that $\phi(u) \leq \phi(v)$ for any arc $uv \in E$. Such an ordering is called topological sort.
Topological Sort
Algorithm for topological sort:

1. Call DFS(G) to compute finishing times $f(v)$, $v \in V$.
2. Order V in order of decreasing $f(v)$

(picture from l14 p18) Application.
A student needs to take a certain number of courses to graduate, and these courses have prerequisites that must be followed. Assume that all courses are offered every semester and that the student can take an unlimited number of courses. Describe an efficient algorithm, which given a list of courses and their prerequisites, computes a schedule that requires the minimum number of semesters. Show how the algorithm works on an example based on your own list of courses. Question. Which courses can you take in the first semester?
Biconnectivity
Let $G = (V, E)$ be an undirected graph. An articulation point is a vertex $\in V$, such that, if we delete v with all its incident edges from G, we break G into at least two connected components. A connected graph G is biconnected if it has no articulation points (you cannot disconnect it by removing one vertex).
Fact. DFS can be modified so that it finds all articulation points in linear time.
Example: **Depth-First Search Algorithm**
Strongly connected component Let $G(V, E)$ be a digraph. A strongly connected component $G_1 = (V_1, E_1)$ is a subdigraph of G every vertex of which is reachable from every other. In the following is an algorithm for determining a strongly connected component of a digraph $G = (V, E)$.
A linear-time algorithm for computing SCCs 1. Call DFS(G) to compute the finishing times $f(v)$ for all $v \in V$. 2. Renumber the vertices v in order of decreasing $f(v)$ (max. finishing time first). 3. Compute G_r = reversed graph of G. Run DFS (G_r) (consider vertices in decreasing order of $f(v)$). 4. Output each tree in DFS(G_r) as a separate

strongly connected component.
Here in Figure the strongly connected components are induced by the set of vertices $V_1 = \{\}, V_2 = \{\}, \text{and} V_3 = \{\},$

Biconnectivity

We now consider an application of depth-first search to determining biconnected components of a graph. Let $G = (V, E)$ be a connected graph. A vertex w is said to be an *articulation point* of G if there exist vertices u, v such that u, v, w are distinct, and every path between u and v must contain the vertex w. The graph G is biconnected if for every distinct triple u, v, w there exists a path between u, v not containing w. Thus a connected graph is biconnected if and only if it has no articulation point. We can partition the edges of a graph into equivalences in the sense that any two edges will be in the same class if there is a cycle passing through both of them. Let $E_i, i = 1, ..., k$ be the equivalence classes into which edges in E have been partitioned, and let V_i be the corresponding set of vertices. Then $G_i = (V_i, E_i), i = 1, ..., k$ are biconnected components of G. Note that $\forall i \neq j,\ V_i \cap V_j$ contains at most one vertex. dfnumber(v) of a vertex v is the order in which the vertex is reached in depth-first search. $low(v)$ is the smaller of of order in which v has been reached or the vertex reached by back edge from one of the descendants x of v to an ancestor w of v.

```
procedure SEARCHB(v)
mark $v$ labelled
Dfnumber(v)=count
count=count+1
low(v)=dfnumber(v)
for each vertex w on L(v) do
if w is labelled then
add (v,w) to T
father(w)=v
SEARCHB(w)
if low(w) $\geq dfnumber(v)$ then
a biconnected component has been found
endif
$low(v) = min(low(v), low(w))$
elseif w is not a father(v) then
$low(v) = min(low(v), dfnumber(w))$
endif
enddo
```

Cartesian tree

A cartesian tree corresponding to an array is a tree in which in order traversal will result in the array, and for every subarray the smallest element will be in its root. A cartesian tree can be constructed incrementally in $\mathcal{O}(n)$ computation.

Dijkstra's algorithm

Edsger W. Dijkstra conceived the idea of all pairs shortest path algorithm in 1956 and published it in 1959. The principle behind Dijkstras algorithm is that if $s, .., x, , t$ is the

shortest path from s to t, then $s, , x$ had better be the shortest path from s to x. This suggests a dynamic programming-like strategy, where we store the distance from s to all nearby nodes, and use them to nd the shortest path to more distant nodes.
The shortest path from s to s, $d(s, s) = 0$. If all edge weights are positive, the smallest edge incident to s, say (s, x), denes $d(s, x)$. We can use an array to store the length of the shortest path to each node. Initialize each to infinity to start. Soon as we establish the shortest path from s to a new node x, we go through each of its incident edges to see if there is a better way from s to other nodes through x.

```
known[s]=1
for v = 1 to n, dist[v] = infinity
for each edge (s, v), dist[v] = d(s, v)
last=s
while ($last \neq t$) do
select v such that dist(v) = min{unknown dist(i)}
for each (v, x), dist[x] = min{dist[x], dist[v] + d(v; x)}
last=v
known[v] = 1
enddo
```

Complexity $O(n^2)$. This is essentially the same as Prims algorithm. Notice that finding the shortest path between a pair of vertices (s, t) in worst case requires first finding the shortest path from s to all other vertices in the graph. Many applications, such as finding the center or diameter of a graph, require finding the shortest path between all pairs of vertices. We can run Dijkstra's algorithm n times (once from each possible start vertex) to solve all-pairs shortest path problem is $O(n^3)$.

Minimum spanning tree

Given a connected graph $G = (V, E)$ with weight w_i on edge e_i the problem is to find a spanning tree of minimum weight. Spanning tree is an acyclic subgraph. A greedy criterion could be choosing smallest edge so that it does not form cycle with already selected edges. This strategy indeed gives optimal solution and the algorithm is named after Kruskal. Kruskal's algorithm is a greedy algorithm in graph theory that finds a minimum spanning tree for a connected weighted graph. This means it finds a subset of the edges that forms a tree that includes every vertex, where the total weight of all the edges in the tree is minimized. If the graph is not connected, then it finds a minimum spanning forest (a minimum spanning tree for each connected component). This algorithm first appeared in Proceedings of the American Mathematical Society, pp. 4850 in 1956, and was written by Joseph Kruskal. Other algorithms for this problem include Prim's algorithm, Reverse-Delete algorithm, and Borůvka's algorithm.

Kruskal's algorithm create a forest F (a set of trees), where each vertex in the graph is a separate tree

```
create a set S containing all the edges in the graph
while S is nonempty and F is not yet spanning
remove an edge with minimum weight from S
```

if that edge connects two different trees, then add it to the forest, combining two trees into a single tree otherwise discard that edge.
At the termination of the algorithm, the forest has only one component and forms a minimum spanning tree of the graph.

```
$G = (V, E), T = \emptyset, F = E$
While $F$ is not empty
choose edge e with minimum weight
if $T \cup e$ does not form a cycle
$T = T \cup e$
endif
$F = F \setminus e$
Enddo
```

Here main challenge is to test for cycle which has to be done very efficiently. Sets and disjoint set union algorithm does this job in a great way.

The algorithm was developed in 1930 by Czech mathematician Vojtěch Jarník and later independently by computer scientist Robert C. Prim in 1957 and rediscovered by Edsger Wybe Dijkstra in 1959. Therefore it is also sometimes called the DJP algorithm, the Jarník algorithm, or the Prim - Jarník algorithm.

In this algorithm starting from an arbitrary vertex(termed connected and remaining disconnected) vertex nearest to the connected set is sequentially added updating information on the nearest vertex in the connected set from each member of the disconnected set. The only spanning tree of the empty graph (with an empty vertex set) is again the empty graph. The following description assumes that this special case is handled separately. The algorithm continuously increases the size of a tree, one edge at a time, starting with a tree consisting of a single vertex, until it spans all vertices. Input: A non-empty connected weighted graph with vertices V and edges E (the weights can be negative).

```
Initialize:$V_{new} = \{x\}$, where x is an arbitrary node (starting point) from $V, E_{new} =$
Repeat until $V_{new} = V$:
For each vertex not in $V_{new}$ find nearest vertex in $V_{new}$
Choose an edge $(u, v)$ with minimal weight such that $u$ is in $V_{new}$ and $v$ is not (if there are multiple edges with the same weight, any of them may be picked)
Add $v$ to $V_{new}$, and $(u, v)$ to $E_{new}$, Update nearest array
Output: $V_{new}$ and $E_{new}$ describe a minimal spanning tree
```

Prim's algorithm

Time complexity

Minimum edge weight data structure Time complexity (total) adjacency matrix, searching $O(V^2)$ binary heap and adjacency list $O((V+E)logV) = O(ElogV)$ Fibonacci heap and adjacency list $O(E + VlogV)$ A simple implementation using an adjacency matrix graph representation and searching an array of weights to find the minimum weight edge to add requires $O(V^2)$ running time. Using a simple binary heap data structure and an adjacency list representation, Prim's algorithm can be shown to run in time $O(ElogV)$ where E is the number of edges and V is the number of vertices. Us-

ing a more sophisticated Fibonacci heap, this can be brought down to $O(E + VlogV)$, which is asymptotically faster when the graph is dense enough that E is $\Omega(V)$
Proof of correctness The proof consists of two parts. First, it is proved that the algorithm produces a spanning tree. Second, it is proved that the constructed spanning tree is of minimal weight.
Spanning Tree
Let $G = (V, E)$ be a connected, weighted graph and let T be the subgraph of produced by the algorithm. T cannot have a cycle, since the last edge added to that cycle would have been within one subtree and not between two different trees. T cannot be disconnected, since the first encountered edge that joins two components of would have been added by the algorithm. Thus, T is a spanning tree of G.
Minimality
We show that the following proposition P is true by induction: If F is the set of edges chosen at any stage of the algorithm, then there is some minimum spanning tree that contains F. Clearly P is true at the beginning, when F is empty: any minimum spanning tree will do, and there exists one because a weighted connected graph always has a minimum spanning tree. Now assume P is true for some non-final edge set F and let T be a minimum spanning tree that contains F. If the next chosen edge e is also in T, then P is true for $F + e$. Otherwise, $T + e$ has a cycle C and there is another edge f that is in C but not F. (If there were no such edge f, then e could not have been added to F, since doing so would have created the cycle C.) Then $Tf + e$ is a tree, and it has the same weight as T, since T, has minimum weight and the weight of f cannot be less than the weight of e, otherwise the algorithm would have chosen f instead of e. So $Tf + e$ is a minimum spanning tree containing $F + e$ and again P holds. Therefore, by the principle of induction, P holds when F has become a spanning tree, which is only possible if F is a minimum spanning tree itself.
Mincost arborescence problem
Given a digraph D = (V, A) and a root vertex r in V , an r -arborescence (or an arborescence with root r) is a subset of arcs B in A such that for each vertex v in V r, there is a unique.$In(v)$ is the set of edges incoming to v path from r to v in (V, B). There is an equivalent way to define arborescences:
Lemma 1. The set $B \in A$ is anr -arborescence in a digraph $D = (V, A)$ if and only if (V, B) has no cycles, and $|B \cup In(v)| = 1 \forall v \in V\ r$.
Lemma 2. A digraph D contains an r -arborescence if and only if each vertex in D is reachable from r. Proof. If each vertex in D is reachable from r , the breadth-first search (started at r) will find an r -arborescence. The converse is trivial. We consider the following minimum cost r -arborescence problem: Given a digraph D = (V, A), a root vertex r V , and a cost function c : f(A)R, find an r -arborescence of minimum cost in D. The relationship between arborescences and spanning trees might give an impression that the same techniques can be applied here, too. The following algorithm is a complete analogue of the Prims method for spanning trees: (1) Set $U = \{r\}, B$
$emptyset$ (2) While $U \neq V$; (3) find $a = (u, v) \in {}_{out}(U)$ with $c(a) = min\{c(a)|a \in {}_{out}(U)$ (4) set U :=U Uv, B := B U e; (5) output B. Unfortunately, this algorithm fails

on a rather trivial digraph: c(r,v)=2, c(r,u)=3, c(u,v)=1 Nonetheless, it is still possible to design a greedy-type algorithm to find a minimum cost r -arborescence. First, for each v V r , select a cheapest arc a in(v) (with ties broken arbitrarily); let B* be the set of those arcs. Then B* is a cheapest set of arcs satisfying |in(v)B| = 1 for all v V r . In particular, c(B*)c(B) for any r -arborescence B, since B must also have |in(v)B| = 1(see Lemma 1). Now, if (V,B*) does not contain a cycle, then B* is an r -arborescence itself (again, see Lemma 1), and hence a minimum cost r -arborescence. Some problems arise when B* does contain a cycle. (v) := minc(a): a in(v). Following properties hold for new costs: (1)c(a)0 for all a A; (2) (v) =min c(a) : a in(v) for all v V r ; (3)c(a)=0 for all a B* Following lemma asserts asserts equivalence of problems after redefinition of costs Lemma 3. An r -arborescence in a digraph D has the minimum cost with respect to c if and only if it has the minimum cost with respect to c Proof: Let B be an r -arborescence in D. Then
The main idea is now to shrink a cycle in (V,B*), if there is one, into one super-vertex and continue searching for a minimum cost r -arborescence on the smaller digraph. Lemma 4. Let D = (V, A) be a digraph, r V a root vertex and c : A R+ a cost function. Let C be a cycle of zero cost, not containing r . If D contains an r -arborescence, then there is one of minimum cost that enters C exactly once. Proof. =): The only arc in B that enters v is the last arc on a directed path from r to v. If (V,B) contained a cycle, there would be arbitrary many paths from r to any vertex on that cycle. (=: Let v 2 V r . Since |B intersection in(v)| = 1, there is an arc (u, v) in B. If u r , there must be an arc (w,u) in B. Continuing this way we eventually get to r, as (V,B) contains no cycles and therefore we cannot return to a vertex we have previously considered. Remark. The above proof also implies that |in(r)| = 0, as otherwise we could follow arcs in backward direction arbitrarily many times, eventually obtaining a cycle. Thus, an r -arborescence can be viewed as a directed spanning tree rooted at r . In fact, if we ignore the directions of the arcs in D, it becomes just a spanning tree.

Chapter 6

Dynamic Programming
The term dynamic programming was originally used in the 1940s by Richard Bellman to describe the process of solving problems where one needs to find the best decisions one after another. By 1953, he refined this to the modern meaning, referring specifically to nesting smaller decision problems inside larger decisions,[16] and the field was thereafter recognized by the IEEE as a systems analysis and engineering topic. Bellman's contribution is remembered in the name of the Bellman equation, a central result of dynamic programming which restates an optimization problem in recursive form. Dynamic programming is a design technique similar to divide and conquer. Divide-and-conquer algorithms partition the problem into independent subproblems, solve the subproblems recursively, and then combine their solutions to obtain solution of the original problem. Dynamic programming is applicable when the subproblems are not independent, that is, when subproblems share subsubproblems. A dynamic-programming algorithm solves every subsubproblem just once and then saves its answer in a table, thereby avoiding the work of recomputing the answer every time the subsubproblem is encountered.
Dynamic programming was systematized by Richard E. Bellman. He began the systematic study of dynamic programming in 1955. The word "programming," both here and in linear programming, refers to the use of a tabular solution method and not to writing computer code.
Dynamic programming is typically applied to optimization problems. In such problems there can be many possible solutions. **The Principle of Optimality:** To use dynamic programming, the solution must satisfy the principle of optimality, that whatever the initial state is, remaining decisions must be optimal with regard to the state following from the rst decision. However, solutions of many combinatorial problems may satisfy the principle of optimality but may use too much memory/time to be efcient.
Jeep Problem:
A jeep is supposed to reach a spot S inside a desert. Unfortunately there is no fuel depot on the way. However, the jeep is allowed to store fuel on the way anywhere. Jeep can run 1km/litre fuel. Its capacity is C litres. The spot is D kms away from locality. How can it be planned to reach the spot using minimum fuel?

If the jeep reaches C km off S with C liters stored in a depot there, then the problem is solved optimally. Let this state be denoted by (1,C,C), where 1 stands for the first stoppage near S, C stands for distance (d_1) from S and last C stands for the amount of fuel it should carry at that point. To deposit C liters it must ply at least twice in the previous segment. So $3d_2 + C = 2C.Sod_2 = C/3$. The state is (2,C/3,2C). Arguing the same manner we can say $5d_3 + C = 3C.Sod_3 = C/5$ and so on. We continue doing so making k depots so that $\sum_{i=1}^{k-1} d_i \leq S \leq \sum_{i=1}^{k} d_i$. Cost of the last segment would be $(2k-1)(S - \sum_{i=1}^{k-1} d_i)$.
What about if the Jeep is to return to the point it started from? $0-1$ **Knapsack Problem**
Consider the knapsack problem with object i having profit and weight $(p_i, w_i), i = 1,,n$, and weight bound W. Now the condition is that we must either choose an object in full or avoid choosing it. Choose the items for which profit will be the maximum subject to knapsack capacity.
Let $f_i(y)$ be the maximum profit from items $1,2,,i$ with weight bound y. Then $f_1(y) = p_1 \text{if} y \geq w_1, 0 \text{otherwise} \forall y$. $f_{i+1}(y) = \min\{p_{i+1} + f_1(y - w_{i+1}), f_i(y)\}$
This is the dynamic programming recurrence. Once we can compute $f_n(W)$ we have found the solution. Let us have the following example.
Flight Problem
An aircraft is destined to fly from south west corner (s) to north east corner (t) of a mesh each segment of which has a fuel cost to traverse. The aircraft flies to the east or north and never in other directions. How should it undertake this journey minimizing fuel cost? Let us back calculate minimum cost of reaching t from immediate neighbouring vertices at a minimum cost. So every vertex gets a label representing minimum cost of reaching t from that point. At every vertex we have at most 2 decisions to choose from. We always choose smaller of the labels. A vertex can be reached either from east or from north. If $c(i,j)$ is the label of the point in the intersection of ith row and jth column. Then $c(i,j) = \min\{c(i,j-1) + f(j,j-1), c(i+1,j) + f(i,i+1)\}$ where $f(i,i+1)$ is the fuel cost of horizontal segment $(i,i+1)$, and $f(j,j-1)$ is the fuel cost of vertical segment $(j,j-1)$. Once $s = (0,0)$ has been labelled we have solved the problem and the minimum fuel cost equals the label $c(0,0)$. Here is a numerical example.
Chain Matrix Problem
Consider the problem of multiplying $M_1 \times M_2 \times M_3$ where dimensions are respectively $(m_1 \times m_2), (m_2 \times m_3), (m_3 \times m_4)$. We can do this in the following ways $((M_1 \times M_2) \times M_3)$ or $(M_1 \times (M_2 \times M_3))$. The corresponding number of multiplications will be $m_1 \times m_2 \times m_3 + m_1 \times m_3 \times m_4$ or $m_1 \times m_2 \times m_4 + m_2 \times m_3 \times m_4$ not necessarily equal. Then which order of multiplication will be the best?
Let us have matrices $M_i, i = 1,4$ of dimensions $(4X3), (3X7), (7X5), (5X8)$. Let $C(I,j)$ be the minimum cost of multiplying matrices from I to j. Then we have $C(1,2) = 4 \times 3 \times 7 = 84, c(2,3) = 3 \times 7 \times 5 = 105, c(3,4) = 7 \times 5 \times 8 = 280$ Now there are two ways of multiplying M_i to $M_(i+2)$: $((M_i \times M_(i+1)) \times M_(i+2))$ and $(M_i \times (M_(i+1) \times M_(i+2)))$ and the corresponding costs are $C(1,3) = min\{84 + 4.7.5, 4.3.5 + 105\} = 165, C(2,4) = min\{105 + 3.5.8, 280 + 3.7.8\} = 225C(1,4) =$

Table 7.1: Flight Problem Solution

1	2	3	4	5	6
2	9 5	2 8	3 1	7 3	6 9
9	5 3	7 7	3 8	8 5	9 1
3	9 9	2 4	3 6	7 8	6 7
8	6 7	8 1	5 8	9 4	4 9
6	2 1	5 7	1 2	9 8	8 3
	9	2	8	7	3

$min\{225 + 3.7.8, 84 + 280 + 4.7.8, 165 + 4.5.8\} = 325$ In general for multiplying matrices from I to j can be done in $(j - i - 1)$ ways. Starting from trivial multiplication of 2 matrices we build up optimal way of multiplying k matrices. function OptimalMatrixChainParenthesis(chain) n = length(chain) for i = 1, n m[i,i] = 0 //since it takes no calculations to multiply one matrix for len = 2, n for i = 1, n - len + 1 for j = i, len -1 m[i,j] = infinity //so that the first calculation updates for k = i, j-1 q = m[i, k] + m[k+1, j] + $p_{i-1} * p_k * p_j$ if q < m[i, j] // the new order of parenthesis is better than what we had m[i, j] = q //update s[i, j] = k //record which k to split on, i.e. where to place the parenthesis

Longest Increasing Sequence Given an array $A(1..n)$, the length of the longest increasing subsequences is computed by the function call $LISBIGGER(, A(1..n))$, where LISBIGGER is the following recursive algorithm:

LISBIGGER(prev, A[1 .. n], max): if n = 0 then max=0 else max LISBIGGER(prev, A[2 .. n]) endif if A[1] > prev then L 1 + LISBIGGER(A[1], A[2 .. n]) endif if L > max then max L endif

We can simplify notation with 2 simple observations. Prev is infinity or an element of the input array and second argument of LISBIGGER is always a suffix of the original input array. Assuming A[0]=-infinity our goal is to compute LIS(0,1). For all i<j we have LIS(i,j)=0, if j>n, LIS(i,j)=LIS(i,j+1) if A[i]>=A[j] or LIS(i,j)=maxLIS(i,j+1), 1+LIS(j,j+1)

Finally DP algorithm looks like

LIS(A[1 .. n]): A[0] Add a sentinel Base cases for i 0 to n L IS[i, n + 1] 0 enddo for j n downto 1 for i 0 to j 1 if A[i] A[j] L IS[i, j] LIS[i, j + 1] else LIS[i, j] maxLIS[i, j + 1], 1 + LIS[j, j + 1] endif enddo

Optimal Binary Search Tree(OBST)

If insertion, deletion and search operations are quite frequent unsorted or sorted arrays are costly data structures, whereas binary search trees are good that will require

$O(logn)$ computation for each operation. Random binary search trees are good on the average but may turn out to perform poorly in pathological cases. OBST is a good data structure for all cases. Let us search $b_1, b_2, , b_n$ with respective frequencies $p_1, p_2, , p_n$ and let unsuccessful searches be denoted by $a_0, a_1, , a_n$ with frequencies $q_0, q_1, , q_n$ we want to construct a BST for which cost of all those searches are as small as possible. That is

$$\text{minimize} \sum_{i=1}^{n} p_i.(l_i + 1) + \sum_{i=0}^{n} q_i.l_i$$

where l_i is the level of vertex $i \in T$. For successful searches, corresponding to internal nodes number of probes/comparisons equals one more than level whereas for unsuccessful searches, corresponding to external nodes, number of probes is just equal to number of levels since in the external node there is no data comparison.
If we know the optimal root then left and right subtrees are defined and they must also be optimal with respect to nodes they contain. Let us think of the OBST T_{ij} consisting of vertices $\{a_i, b_i, a_{i+1}, b_{i+1}, , b_j, a_j\}$. Let b_k be the root of T_{ij}. Then since both left and right subtrees must be optimal with respect to the nodes there left subtree must be T_{ik-1} and the right one is T_{kj}. Let us further denote the total frequency in T_{ij} by $W_{ij} = \sum_{m=1}^{k} p_l + \sum_{m=0}^{k} q_m$. Then

$$C_{ij} = \min_{k=1}^{n}\{C_{ik-1} + C_{kj}\} + W_{ij}, C_{ii} = 0, C_{ii+1} = p_i$$

since path length (number of comparisons to reach a node) is 1 more in T_{ij} than in its immediate subtrees T_{ik-1} and T_{kj}.
Let us consider yet another problem where dynamic programming techniques are being used.

Multi-Peg Tower of Hanoi Problem

Tower of Hanoi problem is a widely known example of recursion. Now consider that we have p pegs and n disks to shift following regularity condition. That is we can shift only one disk at a time from top of a tower to top of another tower, and no disk should ever be placed on top of a smaller disk. What is the minimum number of moves required to solve the problem, and how it can be implemented. We denoted this problem by $\Pi(n, p)$ and minimum number of moves by $M(n, p)$. While for traditional Tower of Hanoi , that is $\Pi(n, 3)$ optimality of the algorithm is known this is not so for $\Pi(n, p)$. However, it is widely believed that the strategy where certain optimal number of disks is shifted to an intermediate peg, and then the remaining disks reach destination (without visiting that particular intermediate peg since they have been loaded with smaller disks, and then disks of intermediate peg reaches the destination must be optimal, there is no formal proof of this statement. So this strategy is termed *presumed optimal solution(POS)*. So the solution looks like in the following picture.

```
MPTOH(n,p,s,d) /solve $\Pi(np)$ with $n$ disks, $p$ pegs and source disk $s$ and destination disk $d$
if n>1 then
MPTOH($n_1, p, s, i$)
MPTOH($n - n_1, p - p_1, s, d$)
MPTOH($n_1, p, i, d$)
else
move from peg s to peg d
endif
```

We have some properties for the POS strategy. 1. In a POS strategy each disk requires 2^k moves to reach destination.

Proof This is true for $\Pi(n, p)$ with $n = 1$ and for any value of p. Assume this to be true for $< n$ disks. Now we have a tower of n disks. Since each of the subtasks mentioned in the pseudocode consists of tower having lesser than n disks by induction hypothesis for any disk d number of moves it is involved in must be a power of 2. Since moving the disk d to the intermediate and moving it from intermediate to the destination is symmetrical excepting the location of larger disks in both the cases number of moves should be n^k for some integer k. So $n^k + n^k = n^{k+1}$.

Let $N(k, p)$ be maximum number of disks each of which requires 2^k moves to reach destination given infinite supply of disks. Then

2. In a POS strategy $N(k, p) = N(k-1, p) + N(k, p-1)$

Proof The set of disks each of which requires 2^k moves to reach destination can be partitioned into two subsets- those that visit the particular intermediate peg in 2^{k-1} disks, and those that reach destination without visiting the particular intermediate disk loaded with larger disks. So the equation follows.

3.
$$N(k, p) = \binom{p-3+k}{k}$$

Let number of disks each of which requires no more than 2^k moves to reach destination be denoted by $\overline{N}(k, p)$. Then actual number of disks each of which requires exactly $2^{k_{max}}$ moves is $N_a(k_{max}, p) = n - \overline{N}(k-1, p)$

Let n_1 be number of disks to be shifted to an intermediate disk in a POS strategy. Then

$$\overline{N}(k_{max} - 2), p) \leq n_1 \leq \overline{N}(k_{max} - 2, p) + min\{N(k_{max} - 1, p), N_a(k_{max}, p)\}$$

0-1 Knapsack Problem

While fractional Knapsack problem is amenable to a very simply greedy algorithm $0 - 1$ Knapsack problem is as hard as integer programming problem, and hence NP-hard. In the following we propose a dynamic programming-based algorithm for solution.

Let p_i, w_i, $i = 1, ..., n$ be respectively profit and weight of ith item and C being capacity of the knapsack. Our problem is to select a set of items in its totality so that total profit is maximized.

Let $f_i(x)$ be maximum profit that can be earned from selecting items i onwards to n with the remaining capacity x. Then we are interested in maximizing $f_1(C)$, where C is the capacity of the knapsack.

With respect to item i we have only two possible decisions to make- either select item i or do not select it. So

$$f_i(x) = \min \begin{cases} f_{i+1}(x) & \text{if } i\text{th item is not selected ,} \\ f_{i+1}(x - w_i) + p_i & \text{otherwise ,} \end{cases} \tag{7.1}$$

Consider the following example with

$$(p_i,\ w_i) = (3,1),\ (4,2),\ (2,3),\ (5,7),\ (7,17),\ (2,7),\ (6,15)\text{knapsackcapacity} = 10$$

Table 7.2: 0-1 Fractional Knapsack Solution

x/i	$f_6(x)$	$f_6(x)$	$f_6(x)$	$f_6(x)$	$f_6(x)$	$f_6(x)$
0	0	0	0	0	0	0
1	0	0	0	0	0	0
2	0	0	0	3	3	3
3	0	0	0	3	3	8
4	0	0	0	3	3	8
5	0	0	7	7	7	11
6	15	15	15	15	15	15
7	15	17	17	17	17	17
8	15	17	17	18	18	18
9	15	17	17	20	20	23
10	15	17	17	20	20	25

7.1 Exercise Problems:

1. **Maximum Value Contiguous Subsequence.** Given a sequence of n real numbers $A(1)...A(n)$, determine a contiguous subsequence $A(i)...A(j)$ for which the sum of elements in the subsequence is maximized.

2. **Making Change.** You are given n types of coin denominations of values $v(1) < v(2) < ... < v(n)$ (all integers). Assume $v(1) = 1$, so you can always make change for any amount of money C. Give an algorithm which makes change for an amount of money C with as few coins as possible. [on problem set 4]

3. **Longest Increasing Subsequence.** Given a sequence of n real numbers $A(1)...A(n)$, determine a subsequence (not necessarily contiguous) of maximum length in which the values in the subsequence form a strictly increasing sequence. [on problem set 4]

4. **Box Stacking.** You are given a set of n types of rectangular 3-D boxes, where the ith box has height $h(i)$, width $w(i)$ and depth $d(i)$ (all real numbers). You want to create a stack of boxes which is as tall as possible, but you can only stack a box on top of another box if the dimensions of the 2-D base of the lower box are each strictly larger than those of the 2-D base of the higher box. Of course, you can rotate a box so that any side functions as its base. It is also allowable to use multiple instances of the same type of box.

5. **Building Bridges.** Consider a 2-D map with a horizontal river passing through its center. There are n cities on the southern bank with x-coordinates a(1) ... a(n) and n cities on the northern bank with x-coordinates b(1) ... b(n). You want to connect as many north-south pairs of cities as possible with bridges such that no two bridges cross. When connecting cities, you can only connect city i on the northern bank to city i on the southern bank. (Note: this problem was incorrectly stated on the paper copies of the handout given in recitation.)

6. **Integer Knapsack Problem (Duplicate Items Forbidden).** This is the same problem as the example above, except here it is forbidden to use more than one instance of each type of item.

7. **Balanced Partition.** You have a set of n integers each in the range 0 ... K. Partition these integers into two subsets such that you minimize |S1 - S2|, where S1 and S2 denote the sums of the elements in each of the two subsets.

8. **Edit Distance.** Given two text strings A of length n and B of length m, you want to transform A into B with a minimum number of operations of the following types: delete a character from A, insert a character into A, or change some character in A into a new character. The minimal number of such operations required to transform A into B is called the edit distance between A and B.

9. **Counting Boolean Parenthesizations.** You are given a boolean expression consisting of a string of the symbols 'true', 'false', 'and', 'or', and 'xor'. Count the number of ways to parenthesize the expression such that it will evaluate to true. For example, there are 2 ways to parenthesize 'true and false xor true' such that it evaluates to true.

10. **Optimal Strategy for a Game.** Consider a row of n coins of values v(1) ... v(n), where n is even. We play a game against an opponent by alternating turns. In each turn, a player selects either the first or last coin from the row, removes it from the row permanently, and receives the value of the coin. Determine the maximum possible amount of money we can definitely win if we move first.

11. **Delivering optimally.** Decatron Mills has contracted to deliver 20 tons of a special coarsely ground wheat flour at the end of the current month, and 140 tons at the end of the next month. The production cost, based on which the Sales Department has bargained with prospective customers, is $c_1(x_1) = 7500 + (x_1 - 50)^2$ per ton for the first month, and $c_2(x_2) = 7500 + (x_2 - 40)^2$ per ton for

the second month; x_1 and x_2 are the number of tons of the flour produced in the first and second months, respectively. If the company chooses to produce more than 20 tons in the first month, any excess production can be carried to the second month at a storage cost of \$3 per ton. Assuming that there is no initial inventory and that the contracted demands must be satisfied in each month (that is, no back-ordering is allowed), derive the production plan that minimizes total cost. Solve by both backward and forward induction. Consider x_1 and x_2 as continuous variables, since any fraction of a ton may be produced in either month.

12. **Longest Common Subsequence:** Given two strings S1 and S2. Find the longest common subsequence between S1 and S2.

13. **Shortest Palindrome:** Given a string s, form a shortest palindrome by appending characters at the start of the string.

14. **Palindrome Min Cut:** Given a string S, find the minimum number of cuts required to separate the string into a set of palindromes.

15. **Longest Palindromic Substring:** Given a string S, find the longest palindromic substring.

16. **Longest Palindromic Subsequence:** Given a string S, find the longest palindromic subsequence.

17. **Longest Increasing Subsequence $\mathcal{O}(nlogn)$:** Given an array of integers, find the longest increasing subsequence.

Chapter 8

Backtracking

8.1 The Principle of Optimality

To use dynamic programming, the problem must observe the principle of optimality, that whatever the initial state is, remaining decisions must be optimal with regard the state following from the rst decision. Combinatorial problems may have this property but may use too much memory/time to be efcient. A jeep is supposed to reach a spot S inside a desert. Unfortunately there is no fuel depot on the way. However, the jeep is allowed to store fuel on the way anywhere. Jeep can run 1km/litre fuel. Its capacity is C litres. The spot is D kms away from locality. How can it be planned to reach the spot using minimum fuel?
If the jeep reaches C km off S with C liters remaining then the problem is solved optimally. Let this state be denoted by (1,C,C), where 1 stands for the first stoppage near S, C stands for distance (d_1) from S and last C stands for the amount of fuel it should carry at that point. To deposit C liters it must ply at least twice in the previous segment. So $3d_2 + C = 2C$. So $d_2 = C/3$. The state is $(2, C/3, 2C)$. Arguing the same manner we can say $5d_3 + C = 3C$. So $d_3 = C/5$ and so on. What about if the Jeep is to return to the point it started from? Consider the knapsack problem with objects (pi,wi), i=1,,n, weight bound W. Now the condition is that we must either choose an object in full or avoid choosing it. Let fi (y) be the maximum profit from items 1,2,,i with weight bound y. Then f1 (y)=p1 if y>=w1, 0 otherwise for all y. fi+1 (y)=minpi+1 + f1 (y-wi+1), fi (y) This is the dynamic programming recurrence. Once we can compute fn (W) we have found the solution.

8.2 Flight Problem

An aircraft is destined to fly from south west corner (s) to north east corner (t) of a mesh each segment of which has fuel cost to traverse. How should it undertake this journey minimizing fuel cost. Let us back calculate minimum cost of reaching t from immediate neighbouring vertices at a minimum cost. So every vertex gets a label representing minimum cost of reaching t from that point. At every vertex we have at most 2 decisions to choose from. The aircraft flies to the east or north and never in other directions. We always choose smaller of the labels. A vertex can be reached either from east or from north. If c(i,j) is the label of the point in the intersection of ith row and jth column. Then c(i,j)=minc(i,j-1)+f(j,j-1), c(i+1,j)+f(i,i+1) where f(i,i+1) is the fuel cost of horizontal segment (i,i+1), and f(j,j-1) is the fuel cost of vertical segment (j,j-1). Once s=(0,0) has been labelled we have solved the problem and the minimum fuel cost equals the label c(0,0)

8.3 Chain Matrix Problem

Consider the problem of multiplying M1XM2XM3 where dimensions are respectively (m1Xm2), (m2Xm3), (m3Xm4). We can do this in the following ways ((M1XM2)XM3)=(M1X(M2XM3))
The corresponding number of multiplications will be m1Xm2Xm3+m1Xm3Xm4 or m1Xm2Xm4+m2Xm3Xm4 not necessarily equal. Then which order of multiplication will be the best?
Let us have matrices $M_i, i = 1, 4$ of dimensions (4X3), (3X7), (7X5), (5X8). Let C(I,j) be the minimum cost of multiplying matrices from I to j. Then we have C(1,2)=4X3X7=84, c(2,3)=3X7X5=105, c(3,4)=7X5X8=280 Now there are two ways of multiplying M_i to $M_(i+2) : ((M_iXM_(i+1))XM_(i+2))and(M_iX(M_(i+1)XM_(i+2)))$ and the corresponding costs are C(1,3)=min84+4.7.5, 4.3.5+105=165 C(2,4)=min105+3.5.8, 280+3.7.8=225 C(1,4)=min225+3.7.8, 84+280+4.7.8, 165+4.5.8=325 In general for multiplying matrices from I to j can be done in (j-i-1) ways. Starting from trivial multiplication of 2 matrices we build up optimal way of multiplying k matrices.
Given an array A[1 .. n], the length of the longest increasing subsequences is computed by the function call LISBIGGER(, A[1 .. n]), where LISBIGGER is the following recursive algorithm:
LISBIGGER(prev, A[1 .. n]): if n = 0 return 0 else max LISBIGGER(prev, A[2 .. n]) if A[1] > pr ev L 1 + LISBIGGER(A[1], A[2 .. n]) if L > max max L return max
We can simplify notation with 2 simple observations. Prev is infinity or an element of the input array and second argument of LISBIGGER is always a suffix of the original input array. Assuming A[0]=-infinity our goal is to compute LIS(0,1). For all i<j we have LIS(i,j)=0, if j>n, LIS(i,j)=LIS(i,j+1) if A[i]>=A[j] or LIS(i,j)=maxLIS(i,j+1), 1+LIS(j,j+1) Finally DP algorithm looks like LIS(A[1 .. n]): A[0] Add a sentinel Base cases for i 0 to n L IS[i, n + 1] 0
for j n downto 1 for i 0 to j 1 if A[i] A[j] L IS[i, j] LIS[i, j + 1] else LIS[i, j] maxLIS[i, j + 1], 1 + LIS[j, j + 1]
return L IS[0, 1]

8.4 Optimal Binary Search Tree

If insertion, deletion and search operations are quite frequent unsorted or sorted arrays are costly data structures, where is binary search trees are good that will require O(log n) computation for each operation. Random binary search trees are good on the average but may turn out to perform poorly in pathological cases. OBST is a good data structure for all cases. Let us search b1, b2,,bn with respective frequencies

p1, p2,,pn and let unsuccessful searches be denoted by a0, a1,,an with frequencies q0, q1,,qn we want to construct a BST for which cost of all those searches are as small as possible. That is minimize($l_i(T)$ level of vertex I in T, level of root is 0)
For successful search number of comparisons is one more than the level since we start at level 0, and for unsuccessful search in the leaf node there is no element comparison. If we know the optimal root then left and right subtrees are defined and they must also be optimal wrt nodes they contain. Let us think of the OBST Tij consisting of vertices ai,bi,ai+1,bi+1,,bj,aj
For successful search number of comparisons is one more than the level since we start at level 0, and for unsuccessful search in the leaf node there is no element comparison. If we know the optimal root then left and right subtrees are defined and they must also be optimal wrt nodes they contain. Let us think of the OBST Tij consisting of vertices ai,bi,ai+1,bi+1,,bj,aj
Let Cij be cost of the OBST Tij. Then

8.5 Flow Algorithms

Lecture Overview Flows: Problem Denition Ford-Fulkerson Algorithm Edmonds-Karp Algorithm Blocking Flow Algorithms Dinic Algorithm Three Indians Algorithm Flows: Problem Denition A ow network or shortly network is a directed graph G = (V, E) together with the function c : E R+ that gives capacities of the edges. In a ow network we distinguish two vertices: source denoted as s, sink denoted as t. A ow in G is a function f : V Œ V R. Flows: Problem Denition A ow in G is a function f : V Œ V R fullling the following conditions: for every u, v V we have f(u, v) c(u, v) Capacity condition, for every u, v V we have f(u, v) = f(v, u) skew symmetry condition, for every u V s, t we require vVf(u, v) = 0. ow conservation condition
We say that the value f(u, v) is the ow from u do v. The value of the ow f is denoted | f | and dened as the total ow leaving s over all edges, i.e.: | f | = uVf(s, u). In the maximum ow problem for a given network we want to nd the ow with the highest value. We use the following summing notation, where the argument for the function f : X Y can be a subset A of X. In such a case we assume that the value of f is the sum of its values for set A, i.e.: f(X,Y) = xX yY f(x, y). Using this notation we can write the ow conservation condition for the vertex v as f(v, V) = 0. Lemma 1 Let G = (V, E) be the ow network and let f be a ow in it then: for every X V we have f(X, X) = 0, for every X,Y V we have f(X,Y) = f(Y, X), for every X,Y, Z V we have: f(X Y, Z) = f(X, Z) + f(Y, Z) f(X Y, Z) f(Z, X Y) = f(Z, X) + f(Z,Y) f(Z, X Y). | f | = f(s, V) = f(V, t). We prove this lemma during exercises. Intuitively, for given G and the ow f the residual network contains edges that can take more ow. More formally, for a given pair of vertices u, v V the additional ow c f(u, v) that can be send from u to v till we reach the capacity c(u, v) is c f(u, v) = c(u, v) f(u, v). For example, if c(u, v) = 16 and f(u, v) = 11 then we can increase f(u, v) by c f(u, v) = 5 till we violate the capacity of the edge (u, v). For the given network G = (V, E) and ow f , the residual network Gf induced by f is the graph Gf = (V, Ef) such that: Ef = (u, v) V Œ V : c f(u, v) > 0.

In other words for each edge in the residual network we can still increase the ow by more then 0. The edges in Ef can be directed as edges in E or have reversed direction. f f(u, v) < c(u, v) for (u, v) E then c f (u, v) = c(u, v) f(u, v) > 0 and (u, v) Ef . Whereas if f(u, v) > 0 for (u, v) E then f(v, u) < 0. In such case c f (v, u) = c(v, u) f(v, u) > 0, so (v, u) Ef . Of course if (u, v) nor (v, u) are in G then c(u, v) = c(v, u) = 0 and f(u, v) = f(v, u) = 0, so c f (u, v) = c f (v, u) = 0 as well. Hence, |Ef| 2|E|. Note that the residual network Gf is a ow network with capacities given by c f . The following lemma shows the connection between the residual network and the original network. Lemma 2 Let G = (V, E) be the ow network with source s and sing t, and let f be a ow in G. Moreover, let f be a ow in the residual network Gf . Then ow sum f + f is a valid ow in G with value | f + f| = | f | + | f|. We need to show that the ow conditions hold for the sum, i.e.: skew symmetry condition, capacity condition and the ow conservation condition. For the skew symmetry condition note that, for all u, v V, we have: (f + f)(u, v)=f(u, v) + f (u, v) = =f(v, u) f(v, u) ==(f(v, u) + f(v, u)) = (f + f)(v, u). For the capacity condition note that f(u, v) cf(u, v), for all u, v V, and so: (f + f)(u, v)=f(u, v) + f(u, v) f(u, v) + (c(u, v) f(u, v)) = c(u, v). For the ow conservation condition, for all u V s, t, we have: vV(f + f)(u, v) = vV(f(u, v) + f(u, v) = vV(f(u, v) + vVf(u, v) =0+0=0 . Finally, the value of f + f is: | f + f|=vV(f + f)(s, v) = =vV(f(s, v) + f(s, v)) = =vV f(s, v) + vV f(s, v) = | f | + | f| For the ow network G = (V, E) and the ow f , the augmenting path p is a simple path from s to t in the residual network Gf. By the denition of the residual network for every edge (u, v) on the augmenting we can increase the ow from u to v without violating the capacity condition. The highest increase of the value of the ow f that can achieved using the path p is the residual capacity of p: cf(p) = mincf (u, v) : (u, v) is on p. The increase of the ow for p is a function fp : V Œ V R:

fNote that fp is a ow in Gf with value | fp| = c f(p) > 0. Corollary 1 Let f = f + fp then fis a ow in G with value | f| = | f | + | fp| > | f |. The maximum ow and minimum cut theorem, we will show, says that the ow is maximum if and only if the residual network does not contain augmenting path. Cut (S, T) in the ow network G = (V, E) is division of V into S and T = V S such that s S and t T. If f is a ow then netto ow through the cut (S, T) is dened as f(S, T). Capacity of the cut (S, T) is dened as c(S, T). Minimum cut is the one with minimum capacity. The cut (s, v1, v2, v3, v4, t) with netto ow f(v1, v3) + f(v2, v3) + f(v2, v4) = 12 + (4) + 11 = 19, and capacity c(v1, v3) + c(v2, v4) = 12 + 14 = 26. One should note that netto ow through the cut can contain negative ow, but the capacity of the cut contains only nonnegative values. In other words netto ow through the cut (S, T) is obtained by: adding the positive ow from S to T, subtracting the positive ow from T to S. On the other hand the capacity of the cut (S, T) is composed only out of the edges going from S to T. The edges going from T to S are not taken into account. The following lemma shows that the netto ow through any cut is exactly equal to the value of the ow. Lemma 3 Let f be a ow in the network G and let (S, T) be the cut in G then netto ow through (S, T) is equal to f(S, T) = | f |. Note that f(S s, V) = 0 by ow conservation condition, so: f(S, T)=f(S, V) f(S, S) = =f(s, V) + f(S s, V) = f(s, V) = | f |.

A a result we get that the capacity of the cut is an upper bound on the ow value.

Corollary 2 The value of any ow f in G is bounded from above by the capacity of any cut in G. From Lemma 3 and the capacity condition we get: | f |=f(S, T) = uS vT f(u, v) uS vT c(u, v) = c(S, T). The maximum ow is upper bounded by the minimum cut. The maximum ow and minimum cut theorem says that these values are actually equal. Theorem 1 For a ow f in a network G = (V, E) the following conditions are equivalent: 1. f is the maximum ow in G, 2. residual network Gf does not contain augmenting paths, 3. | f | = c(S, T) for some cut (S, T) in G. (1)(2): Assume by contradiction that f is the maximum ow but Gf contains an augmenting path p.Consider the ow f given as the sum f = f + fp where fp is given by (1). By Corollary 1 fis a ow in G and is strictly larger then f . This contradicts the assumption that f is maximum. (2)(3): Let us assume that Gf does not contain any augmenting path, i.e., there is no path going from s to t in Gf. Dene: S := v V : there exists a path from s to v in Gf T := V S There is not path form s to t in so, t T and trivially s S. Hence (S, T) is a cut. For every pair u S and v T, we have f(u, v) = c(u, v), as otherwise (u, v) Ef, and v would be in S. By Lemma 3 we get | f | = f(S, T) = c(S, T). (3)(1): By Corollary 2 we know that | f | c(S, T) for all cuts (S, T). Hence the condition | f | = c(S, T) means that f is the maximum ow. In each iteration of FF we nd an augmenting path. for every edge (u, v) E do f(u, v) = 0 f(v, u) = 0 while there exists a path p from s to t in the residual network Gf do cf (p) = mincf (u, v) : (u, v) p for every edge (u, v) p do f(u, v) = f(u, v) + cf (p) f(v, u) = f(u, v) return f The running time of the Ford-Fulkerson algorithm depends on the choice of the augmenting path. If we do it wrongly the algorithm might even not stop. In the case when the edge capacities are integral, we can easily show that the running time of the algorithm is O(E| f|), where f is the maximum ow. If f is small the algorithm nishes fast, but even in easy cases it might need | f| iterations. Edmonds-Karp algorithm is the Ford-Fulkerson algorithm in which instead of any path we use the shortest augmenting path. We assume that the edges have unit lengths. This modication allows to improve the running time of the algorithm to polynomial time. We will show that Edminds-Karp algorithm works in O(nm2) time. We denote by df(u, v) the distance from u to v in Gf assuming that each edge has unit length. Lemma 4 If Edmonds-Karp algorithm works in ow network G = (V, E) with the source s and sink t then for all vertices v V s, t the distance df(s, v) in the residual network Gf does not decrease. Assume that for some vertex v V s, t there exists augmenting path q that decreases the distance from s to v. Let f be the ow before we apply the path p and let f be the ow just after this. Assume that v was the vertex minimizing df(s, v). We have df (s, v) < df(s, v). Let p = s u v be the shortest path from s to v in Gf such that (u, v) Ef and df (s, u) = df (s, v) 1. Due to the choice of v we know that the distance to u was not decreased: df (s, u) df(s, u). We will prove that (u, v) / Ef. Assume otherwise (u, v) Ef, then using triangle inequality for s, v and u we get: df(s, v) df (s, u) + 1 df (s, u) + 1 df (s, v), This contradicts the assumption that df (s, v) < df (s, v). We have (u, v) / Ef and (u, v) Ef . Hence, the augmentation of the ow f to f Increased the ow v to u. Edmonds-Karp uses shortest paths, so the shortest path from s to u in Gf ends with the edge (v, u) and: df (s, v)=df (s, u) 1 =df(s, u) 1 = df(s, v) 2, this contradict the assumption that df(s, v) < df (s, v). The following theorem bounds the number of iterations of Edmonds-Karp algorithm. Theorem 2 For a network G = (V, E) Edmonds-Karp algorithm is executed at most O(|V||E|) times. Every iteration of the algorithm can be implemented in O(|E| + |V| log |V|) time, so the total running time of the algorithm is O(|E|2|V| + |E||V|2log |V|) time. We say that an edge (u, v) in residual network Gf is critical on an augmenting path p if residual capacity of p equals to the residual capacity of (u, v), i.e.: c f (p) = c f (u, v). After we increase the ow using the augmenting path every critical edge disappears from the residual network. We will show that every edge from E can become critical at most |V|/2 1 times. Let u and v be connected by a critical edge edge. The augmenting paths are the shortest paths so for the critical edge (u, v) we get df (s, v) = df (s, u) + 1. After the augmentation the edge (u, v) disappears and cannot appear on any other augmenting path till the ow from u to v is not decreased. This can happen only if when (v, u) appears on an augmenting path. f f is the ow in G and this happens then we have: df(s, u) = df (s, v) + 1. We have df(s, v) df (s, v) by Lemma 4 we get: df (s, u)=dfa(s, v) + 1 df (s, v) + 1 = df (s, u) + 2. Hence, between the time when (u, v) was once critical and the becomes critical again the distance from the source to u has to increase by at least 2. he distance to u at the beginning is bigger then 0. The path from s to u cannot contain t, because (u, v) is critical and u t. The distance to u is at most |V| 2. Hence, (u, v) can become critical at most (|V| 2)/2 = |V|/2 1 times. Each augmenting path contains at least one critical edge, so the total number of such paths is O(|V||E|). A blocking ow in residual network Gf is a ow b in Gf, such that: 1. every path from s to t in b is a shortest path in Gf, 2. every shortest path in Gf contains a saturated edge in Gf+b . Saturated edge is an edge with residual capacity equal to zero. Note that this denition corresponds to the notion of the maximal set of shortest paths in Hopcroft-Karp algorithm. Dinic algorithm uses blocking ows to nd the maximum ow in the following way. f = 0 while there exists a path from s to t in Gf do b w Gf f = f + b return f The correctness of the algorithm results directly from Maximum Flow and Minimum Cut theorem because when the algorithm stops there are no augmenting paths in G. Let us now see how many times the while loop can be executed. Lemma 5 Let b be the blocking ow in Gf then the shortest augmenting path in Gf+bis longer then the shortest augmenting path in Gf. Assume that the length of the shortest path p in Gf+b is not larger then the length of the shortest path in Gf. Then p shares a saturated edges with the blocking ow b. Let u v be the last such edge on p. This means that v u belongs to b. Otherwise in Gf+b the edge u v would still be saturated. The ow b can be decomposed into the sum of shortest paths in Gf , so by Lemma 4 the distance from s to u did not decrease, i.e., df (s, u) df+b(s, u). However, p is the shortest path from s to t so the distance to v could increase by at most 2, df (s, v) + 2 df+b(s, v). The part of p from v to t is the shortest path as well, so df(s, t) + 2 df+b(s, t), i.e., the length of the shortest path in residual graph has increased. Corollary 3 The length of the shortest path is at most n 1 so the maximum number of phases in Dinic algorithm is at most n. In order to nd blocking we need the layered network. Layered network Gf for the residual network Gf = (V, Ef) is a directed graph Gf= (V, Ef) with edges: Ef = (u, v) : (u, v) Ef and df (s, u) + 1 = df (s, v). Note that all paths from s to t in Gf are shortest paths. Hence, if we nd a blocking ow in layered graph we will automatically full the rst condition. The three Indians algorithm is called as well MKM algorithm (Malhotra, Kumar, Maheshwari). In each execution of the main loop we will saturate one vertex by sending the ow backward and forward. During the algorithm the function f will not fulll the ow conditions. However, at the end of the algorithm these conditions will be restored. Vertex capacity for v V is dened as: c(v) = minuVc(u, v), uVc(v, u)

We need the following two auxiliary procedures: FORWARD(v) if more ow enters v then leaves it, this procedure sends the surplus ow forward in Gf saturating one by one the edges leaving v, BACKWARD(v) if more ow leaves v then enters it, this procedure compensates this insufciency by sending ow on to v in Gf saturating one by one edges edges entering v. b = 0 and construct the graph Gf while Ef do nd a vertex with the smallest c(v) send c(v) units of ow using edges leaving v send c(v) units of ow using edges entering v for i = d(s, v) + 1 to n 1 do for w w V : d(s, w) = i do FORWARD(w) for i = d(s, v) 1 downto 1 do for w w V : d(s, w) = i do BACKWARD(w) remove v from the network and correct capacities of the neighboring vertices return b Note that we have chosen the vertex that has the smallest capacity, in the procedures FORWARD and BACKWARD we can always send the surplus or compensate the insufciency. The main loop will be executed at most n 2 times, because each time we saturate one vertex in the graph. Let us now count how many times we increase ow on edges in the procedures FORWARD and BACKWARD. Edges will be saturated at most m times. We will increase the ow without saturating edges at most O(n), because each time the the procedures FORWARD and BACKWARD are executed we send ow without saturating an edge at most once. there are at most O(n2) calls to these procedures. Hence we need O(n2) time to nd a blocking ow. Together with with the blocking ow algorithm we get an algorithm nding maximum ow in O(n3) time.

8.6 Theory of NP Completeness

There are problems for which it had been impossible to design efficient algorithms. By efficient algorithms we mean algorithms that solve the problem in polynomial time of input size. There are problems for exponential time lower bounds could not be proved. By the early 1970s, literally hundreds of problems were stuck in this limbo. The theory of NP-Completeness, developed by Stephen Cook and Richard Karp, provided the tools to show that all of these problems were really the same problem. Suppose I gave you the following algorithm to solve the bandersnatch problem: Bandersnatch(G) Convert G to an instance of the Bo-billy problem Y . Call the subroutine Bo-billy on Y to solve this instance. Return the answer of Bo-billy(Y) as the answer to G. Such a translation from instances of one type of problem to instances of another type such that answers are preserved is called a reduction. Now suppose my reduction translates G to Y in O(P(n)): 1. If my Bo-billy subroutine ran in O(P(n)) I can solve the Bandersnatch problem in O(P(n) + P(n)) 2. If I know that (P(n)) is a lower-bound to compute Bandersnatch, then (P(n) P(n)) must be a lowerbound

to compute Bo-billy. The second argument is the idea we use to prove problems hard! problem is a general question, with parameters for the input and conditions on what is a satisfactory answer or solution. An instance is a problem with the input parameters specied. Example: The Traveling Salesman Problem: Given a weighted graph G, what tour v1, v2, ..., vn Minimizes sumi=1,n-11d[vi , vi+1] + d[vn, v1]. Instance: d[v1, d2] = 10, d[v1, d3] = 5, d[v1, d4] = 9,
Solution: v1, v2, v3, v4 cost= 27 A problem with answers restricted to yes and no is called a decision problem. Most interesting optimization problems can be phrased as decision problems which capture the essence of the computation. Example: The Traveling Salesman Decision Problem. Given a weighted graph G and integer k, does there exist[v2, d3] = 6, d[v2, d4] = 9, d[v3, d4] = 3
Using binary search and the decision version of the problem we can nd the optimal TSP solution. For convenience, from now on we will talk only about decision problems. Note that there are many possible ways to encode the input graph: adjacency matrices, edge lists, etc. All reasonable encodings will be within polynomial size of each other.
What is a problem? The fact that we can ignore minor differences in encoding is important. We are concerned with the difference between algorithms which are polynomial and exponential in the size of the input. Consider the following logic problem: Instance: A set V of variables and a set of clauses C over V . Question: Does there exist a satisfying truth assignment for C?
Satisfiability Example 1: V = v1, v2 and C = v1, v2, v1, v2 A clause is satised when at least one literal in it is TRUE. C is satised when v1 = v2 =TRUE. Example 2: V = v1, v2, C = v1, v2, v1, v2, v1 Although you try, and you try, and you try and you try, you can get no satisfaction. There is no satisfying assigment since v1 must be FALSE (third clause), so v2 must be FALSE (second clause), but then the rst clause is unsatisable! For various reasons, it is known that satisability is a hard problem. Every top-notch algorithm expert in the world (and countless other, lesser lights) have tried to come up with a fast algorithm to test whether a given set of clauses is satisable, but all have failed. Further, many strange and impossible to-believe things have been shown to be true if someone in fact did nd a fast satisability algorithm. Clearly, Satisability is in NP, since we can guess an assignment of TRUE, FALSE to the literals and check it in polynomial time.
The precise distinction between whether a problem is in P or NP is somewhat technical, requiring formal language theory and Turing machines to state correctly. However, intuitively a problem is in P, (ie. polynomial) if it can be solved in time polynomial in the size of the input. A problem is in NP if, given the answer, it is possible to verify that the answer is correct within time polynomial in the size of the input. Example P Is there a path from s to t in G of length less than k. Example NP Is there a TSP tour in G of length less than k. Given the tour, it is easy to add up the costs and convince that it is correct Example not in NP How many TSP tours are there in G of length less than k. Since there can be an exponential number of them, we cannot count them all in polynomial time. Dont let this issue confuse you the important idea here is of reductions as a way of proving hardness.3-Satisability Instance: A

collection of clause C where each c clause contains exactly 3 literals, boolean variable v. Question: Is there a truth assignment to v so that each clause is satised? Note that this is a more restricted problem than SAT. If 3-SAT is NP-complete, it implies SAT is NP-complete but not visaversa, perhaps long clauses are what makes SAT difcult?! After all, 1-Sat is trivial! Theorem: 3-SAT is NP-Complete Proof: 3-SAT is NP given an assignment, just check that each clause is covered. To prove it is complete, a reduction from Sat 3 Sat must be provided. We will transform each clause independently based on its length. Suppose the clause Ci contains k literals. If k = 1, meaning Ci = z1, create two new variables v1, v2 and four new 3-literal clauses: v1, v2, z1, v1, v2, z1, v1, v2, z1, v1, v2, z1. Note that the only way all four of these can be satised is if z is TRUE. If k = 2, meaning z1, z2, create one new variable v1 and two new clauses: v1, z1, z2, v1, z1, z2 If k = 3, meaning z1, z2, z3, copy into the 3-SAT instance as it is. If k > 3, meaning z1, z2, ..., zn, create n 3 new variables and n 2 new clauses in a chain: vi , zi , vi, . . . If none of the original variables in a clause are TRUE, there is no way to satisfy all of them using the additional variable: (F, F, T),(F, F, T), . . . ,(F, F, F) But if any literal is TRUE, we have n 3 free variables and n 3 remaining 3-clauses, so we can satisfy each of them. (F, F, T),(F, F, T), . . . ,(F, T, F), . . . ,(T, F, F),(T, F, F) Since any SAT solution will also satisfy the 3-SAT instance and any 3-SAT solution sets variables giving a SAT solution the problems are equivalent. If there were n clauses and m total literals in the SAT instance, this transform takes O(m) time, so SAT and 3-SAT. Note that a slight modication to this construction would prove 4-SAT, or 5-SAT,... also NP-complete. However, it breaks down when we try to use it for 2-SAT, since there is no way to stuff anything into the chain of clauses. It turns out that resolution gives a polynomial time algorithm for 2-SAT. Having at least 3-literals per clause is what makes the problem difcult. Now that we have shown 3-SAT is NP-complete, we may use it for further reductions. Since the set of 3-SAT instances is smaller and more regular than the SAT instances, it will be easier to use 3-SAT for future reductions. Remember the direction to reduction! Sat 3 Sat XA Note carefully the direction of the reduction. We must transform every instance of a known NP-complete problem to an instance of the problem we are interested in. If we do the reduction the other way, all we get is a slow way to solve x, by using a subroutine which probably will take exponential time. This always is confusing at rst - it seems bass-ackwards. Make sure you understand the direction of reduction now - and think back to this when you get confused. Backtracking In this lecture, I want to describe another recursive algorithm strategy called backtracking. A backtracking algorithm tries to build a solution to a computational problem incrementally. Whenever the algorithm needs to decide between multiple alternatives to the next component of the solution, it simply tries all possible options recursively. n Queens

The prototypical backtracking problem is the classical n Queens Problem, rst proposed by German chess enthusiast Max Bezzel in 1848 for the standard 8 Œ 8 board, and both solved and generalized to larger boards by Franz Nauck in 1850. The problem is to place n queens on an n Œ n chessboard, so that no two queens can attack each other. For readers not familiar with the rules of chess, this means that no two

queens are in the same row, column, or diagonal. Obviously, in any solution to the n-Queens problem, there is exactly one queen in each row. So we will represent our possible solutions using an array Q[1 .. n], where Q[i] indicates which square in row i contains a queen, or 0 if no queen has yet been placed in row i. To nd a solution, we put queens on the board row by row, starting at the top. A partial solution is an array Q[1 .. n] whose rst r 1 entries are positive and whose last n r + 1 entries are all zeros, for some integer r. The following recursive algorithm recursively enumerates all complete n-queens solutions that are consistent with a given partial solution. The input parameter r is the rst empty row. Thus, to compute all n-queens solutions with no restrictions, we would call RECURSIVENQUEENS(Q[1 .. n], 1).
RECURSIVENQUEENS(Q[1 .. n], r): if r = n + 1 print Q else for j 1 to n legal TRUE for i 1 to r 1 if (Q[i] = j) or (Q[i] = j + r i) or (Q[i] = j r + i) legal FALSE if legal Q[r] j RECURSIVENQUEENS(Q[1 .. n], r + 1)
Picture of an8-queen problem solution.

Bibliography