Analysis and Design of Algorithms
CSE-306F
CSE 6th SEM
Section A

Brief Review

What is an algorithm? Our text defines an algorithm to be any well-defined computational procedure that takes some values as input and produces some values as output. Like a cooking recipe, an algorithm provides a step-by-step method for solving a computational problem. Unlike programs, algorithms are not dependent on a particular programming language, machine, system, or compiler. They are mathematical entities, which can be thought of as running on some sort of idealized computer with an infinite random access memory and an unlimited word size. Algorithm design is all about the mathematical theory behind the design of good programs.

Need of study algorithm design? Programming is a very complex task, and there are a number of aspects of programming that make it so complex. The first is that most programming projects are very large, requiring the coordinated efforts of many people. (This is the topic a course like software engineering.) The next is that many programming projects involve storing and accessing large quantities of data efficiently. (This is the topic of courses on data structures and databases.) The last is that many programming projects involve solving complex computational problems, for which simplistic or naive solutions may not be efficient enough. The complex problems may involve numerical data (the subject of courses on numerical analysis), but often they involve discrete data. This is where the topic of algorithm design and analysis is important. Although the algorithms discussed in this course will often represent only a tiny fraction of the code that is generated in a large software system, this small fraction may be very important for the success of the overall project. An unfortunately common approach to this problem is to first design an inefficient algorithm and data structure to solve the problem, and then take this poor design and attempt to fine-tune its performance. The problem is that if the underlying design is bad, then often no amount of fine-tuning is going to make a substantial difference. The focus of this course is on how to design good algorithms, and how to analyze their efficiency. This is among the most basic aspects of good programming.

Issues: Algorithms are mathematical objects (in contrast to the much more concrete notion of a computer program implemented in some programming language and executing on some machine). As such, we can reason about the properties of algorithms mathematically. When designing an algorithm there are two fundamental issues to be considered: correctness and efficiency. It is important to justify an algorithm’s correctness mathematically. For very complex algorithms, this typically requires a careful mathematical proof, which may require the proof of many lemmas and properties of the solution, upon which the algorithm relies. For simple algorithms (BubbleSort, for example) a short intuitive explanation of the algorithm’s basic invariants is sufficient. (For example, in BubbleSort, the principal invariant is that on completion of the $i$th iteration, the last $i$ elements are in their proper sorted positions.) Establishing efficiency is a much more complex endeavor. Intuitively, an algorithm’s efficiency is a function of the amount of computational resources it requires, measured typically as...
execution time and the amount of space, or memory, that the algorithm uses. The amount of computational resources can be a complex function of the size and structure of the input set. In order to reduce matters to their simplest form, it is common to consider efficiency as a function of input size. Among all inputs of the same size, we consider the maximum possible running time. This is called worst-case analysis. It is also possible, and often more meaningful, to measure average-case analysis. Average-case analyses tend to be more complex, and may require that some probability distribution be defined on the set of inputs. To keep matters simple, we will usually focus on worst-case analysis in this course.

Mathematical Background

Asymptotics: Asymptotics involves O-notation (“big-Oh”) and its many relatives, \( O, \Theta, o \) (“little-Oh”), /\). Asymptotic notation provides us with a way to simplify the functions that arise in analyzing algorithm running times by ignoring constant factors and concentrating on the trends for large values of \( n \). For example, it allows us to reason that for three algorithms with the respective running times \( n^3 \log n + 4n^2 + 52n \log n \), \( 2 \Theta (n^3 \log n) \), \( 15n^2 + 7 \log^3 n \), \( 2 \Theta (n^2) \), \( 3n + 4 \log 5 + 19n^2 \), \( 2 \Theta (n^2) \):

Thus, the first algorithm is significantly slower for large \( n \), while the other two are comparable, up to a constant factor. Since asymptotics were covered in earlier courses, I will assume that this is familiar to you. Nonetheless, here are a few facts to remember about asymptotic notation:

Ignore constant factors: Multiplicative constant factors are ignored. For example, \( 347n \) is \( \Theta (n) \). Constant factors appearing exponents cannot be ignored. For example, \( 2^n \) is not \( O(2n) \).

Focus on large \( n \): Asymptotic analysis means that we consider trends for large values of \( n \). Thus, the fastest growing function of \( n \) is the only one that needs to be considered. For example, \( 3n^2 \log n + 25n \log n + (\log n)^7 \) is \( \Theta (n^2 \log n) \).

Polylog, polynomial, and exponential: These are the most common functions that arise in analyzing algorithms:

- **Polylogarithmic**: Powers of \( \log n \), such as \( (\log n)^7 \). We will usually write this as \( \log^7 n \).
- **Polynomial**: Powers of \( n \), such as \( n^4 \) and \( n^{1/2} = n^{\frac{1}{2}} \).
- **Exponential**: A constant (not 1) raised to the power \( n \), such as \( 3^n \).

An important fact is that polylogarithmic functions are strictly asymptotically smaller than polynomial function, which are strictly asymptotically smaller than exponential functions (assuming the base of the exponent is bigger than 1). For example, if we let \( _- \) mean “asymptotically smaller” then \( \log^a \) \( n \) \( - \) \( n^b \) \( - \) \( c^a \) for any \( a, b, \) and \( c \), provided that \( b > 0 \) and \( c > 1 \).

Logarithm Simplification: It is a good idea to first simplify terms involving logarithms. For example, the following formulas are useful. Here \( a; b; c \) are constants:

\[
\begin{align*}
\log b &= \log a \frac{n}{\log a} \quad b^{-\log a} = \_ (\log a n) \\
\log a(n^c) &= c \log a n = \_ (\log a n) \\
\log_2(n) &= n^{\log_2 b}
\end{align*}
\]

Avoid using \( \log n \) in exponents. The last rule above can be used to achieve this. For example, rather than saying \( 3 \log^2 n \), express this as \( n^{\log^2 3} \) \( \_ n^{1.585} \).

Following the conventional sloppiness, I will often say \( O(n^2) \), when in fact the stronger statement \( O(n^2) \) holds.

(This is just because it is easier to say “oh” than “theta”.)

Summations: Summations naturally arise in the analysis of iterative algorithms. Also, more complex forms of analysis, such as recurrences, are often solved by reducing them to summations. Solving a summation means reducing it to a **closed form formula**, that is, one having no summations, recurrences, integrals, or other complex operators. In algorithm design it
is often not necessary to solve a summation exactly, since an asymptotic approximation or close upper bound is usually good enough. Here are some common summations and some tips to use in solving summations.

**Constant Series:** For integers $a$ and $b$, $\text{Sum1,}i=a \text{ to } b = \max(b-a+1; 0)$:
Notice that when $b = a - 1$, there are no terms in the summation (since the index is assumed to count upwards only), and the result is 0. Be careful to check that $b \geq a - 1$ before applying this formula blindly.

**Arithmetic Series:** For $n \geq 0$,
$\text{Sum } i, i=0 \text{ to } n = 1+2+\ldots+n = n(n+1)/2$:
This is $O(n^2)$. (The starting bound could have just as easily been set to 1 as 0.)

**Geometric Series:** Let $x \neq 1$ be any constant (independent of $n$), then for $n \geq 0$,
$\text{Sum } x^i, i=0 \text{ to } n = 1+x+x^2+\ldots+x^n = x^{n+1} - 1/x - 1$:
If $0 < x < 1$ then this is $\_1(1)$. If $x > 1$, then this is $O(x^n)$, that is, the entire sum is proportional to the last element of the series.

**Quadratic Series:** For $n \geq 0$,
$\text{Sum } i^2, i=0 \text{ to } n = 1^2+2^2+\ldots+n^2 = 2n^3 + 3n^2 +n/6$:

**Linear-geometric Series:** This arises in some algorithms based on trees and recursion. Let $nxq$ be any constant, then for $n \geq 0$,
$\text{Sum } ix^2 = x + 2x^2 + 3x^3+\ldots+nx^n = (n-1)x^{n+1} - nx^n + x/(x-1)^2$:
As $n$ becomes large, this is asymptotically dominated by the term $(n-1)x^{n+1}/(x-1)^2$. The multiplicative term $n-1$ is very nearly equal to $n$ for large $n$, and, since $x$ is a constant, we may multiply this times the constant $(x-1)/x$ without changing the asymptotics. What remains is $O(nx^n)$.

**Recurrences:** Another useful mathematical tool in algorithm analysis will be recurrences. They arise naturally in the analysis of divide-and-conquer algorithms. Recall that these algorithms have the following general structure.

**Divide:** Divide the problem into two or more subproblems (ideally of roughly equal sizes),

**Conquer:** Solve each subproblem recursively, and

**Combine:** Combine the solutions to the subproblems into a single global solution. How do we analyze recursive procedures like this one? If there is a simple pattern to the sizes of the recursive calls, then the best way is usually by setting up a recurrence, that is, a function which is defined recursively in terms of itself. Here is a typical example. Suppose that we break the problem into two subproblems, each of size roughly $n=2$. (We will assume exactly $n=2$ for simplicity.). The additional overhead of splitting and merging the solutions is $O(n)$. When the subproblems are reduced to size 1, we can solve them in $O(1)$ time. We will ignore constant factors, writing $O(n)$ just as $n$, yielding the following recurrence.
$T(n) = 1$ if $n = 1$,
$T(n) = 2T(n=2) + n$ if $n > 1$.

Note that, since we assume that $n$ is an integer, this recurrence is not well defined unless $n$ is a power of 2 (since otherwise $n=2$ will at some point be a fraction). To be formally correct, I should either write $bn=2c$ or restrict the domain of $n$, but I will often be sloppy in this way.
There are a number of methods for solving the sort of recurrences that show up in divide-and-conquer algorithms. The easiest method is to apply the Master Theorem,

**Theorem:** (Simplified Master Theorem) Let $a \geq 1$, $b > 1$ be constants and let $T(n)$ be the recurrence

$$ T(n) = aT(n/b) + cn^k; $$
defined for $n \geq 0$.

**Case 1:** $a > bk$ then $T(n)$ is $\Theta(n \log_b a)$.

**Case 2:** $a = bk$ then $T(n)$ is $\Theta(nk \log n)$.

**Case 3:** $a < bk$ then $T(n)$ is $\Theta(nk)$.

Using this version of the Master Theorem we can see that in our recurrence $a = 2$, $b = 2$, and $k = 1$, so $a = bk$ and Case 2 applies. Thus $T(n)$ is $\Theta(n \log n)$.

There many recurrences that cannot be put into this form. For example, the following recurrence is quite common: $T(n) = 2T(n/2) + n \log n$. This solves to $T(n) = \Theta(n \log^2 n)$, but the Master Theorem such recurrences, other methods are needed.

**Review of Sorting and Selection**

Review of Sorting: Sorting is among the most basic problems in algorithm design. We are given a sequence of items, each associated with a given key value. The problem is to permute the items so that they are in increasing (or decreasing) order by key. Sorting is important because it is often the first step in more complex algorithms. Sorting algorithms are usually divided into two classes, internal sorting algorithms, which assume that data is stored in an array in main memory, and external sorting algorithm, which assume that data is stored on disk or some other device that is best accessed sequentially. We will only consider internal sorting. You are probably familiar with one or more of the standard simple $\Theta(n^2)$ sorting algorithms, such as Insertion-Sort, SelectionSort and BubbleSort. (By the way, these algorithms are quite acceptable for small lists of, say, fewer than 20 elements.) BubbleSort is the easiest one to remember, but it widely considered to be the worst of the three.

The three canonical efficient comparison-based sorting algorithms are MergeSort, QuickSort, and HeapSort. All run in $\Theta(n \log n)$ time. Sorting algorithms often have additional properties that are of interest, depending on the application. Here are two important properties. In-place: The algorithm uses no additional array storage, and hence (other than perhaps the system’s recursion stack) it is possible to sort very large lists without the need to allocate additional working storage.

Stable: A sorting algorithm is stable if two elements that are equal remain in the same relative position after sorting is completed. This is of interest, since in some sorting applications you sort first on one key and then on another. It is nice to know that two items that are equal on the second key, remain sorted on the first key.
Here is a quick summary of the fast sorting algorithms. If you are not familiar with any of these, check out the descriptions in CLRS. They are shown schematically in Fig.

QuickSort: It works recursively, by first selecting a random “pivot value” from the array. Then it partitions the array into elements that are less than and greater than the pivot. Then it recursively sorts each part. QuickSort is widely regarded as the fastest of the fast sorting algorithms (on modern machines). One explanation is that its inner loop compares elements against a single pivot value, which can be stored in a register for fast access. The other algorithms compare two elements in the array. This is considered an in-place sorting algorithm, since it uses no other array storage. (It does implicitly use the system’s recursion stack, but this is usually not counted.) It is not stable. There is a stable version of QuickSort, but it is not in-place. This algorithm is $\Theta(n \log n)$ in the expected case, and $\Theta(n^2)$ in the worst case. If properly implemented, the probability that the algorithm takes asymptotically longer (assuming that the pivot is chosen randomly) is extremely small for large $n$.

![QuickSort diagram](image)

![MergeSort diagram](image)

![HeapSort diagram](image)

Fig.: Common $O(n \log n)$ comparison-based sorting algorithms.

MergeSort: MergeSort also works recursively. It is a classical divide-and-conquer algorithm. The array is split into two subarrays of roughly equal size. They are sorted recursively. Then the two sorted subarrays are merged together in $\Theta(n)$ time. MergeSort is the only stable sorting algorithm of these three. The downside is the MergeSort is the only algorithm of the three that requires additional array storage (ignoring the recursion stack), and thus it is not in-place. This is because the merging process merges the two arrays into a third array. Although it is possible to merge arrays in-place, it cannot be done in $\Theta(n)$ time.
HeapSort: HeapSort is based on a nice data structure, called a heap, which is an efficient implementation of a priority queue data structure. A priority queue supports the operations of inserting a key, and deleting the element with the smallest key value. A heap can be built for n keys in $\Theta(n)$ time, and the minimum key can be extracted in $\Theta(\log n)$ time. HeapSort is an in-place sorting algorithm, but it is not stable. HeapSort works by building the heap (ordered in reverse order so that the maximum can be extracted efficiently) and then repeatedly extracting the largest element. (Why it extracts the maximum rather than the minimum is an implementation detail, but this is the key to making this work as an in-place sorting algorithm.)

If you only want to extract the k smallest values, a heap can allow you to do this is $\Theta(n+k \log n)$ time. A heap has the additional advantage of being used in contexts where the priority of elements changes. Each change of priority (key value) can be processed in $\Theta(\log n)$ time. Which sorting algorithm should you implement when implementing your programs? The correct answer is probably “none of them”. Unless you know that your input has some special properties that suggest a much faster alternative, it is best to rely on the library sorting procedure supplied on your system. Presumably, it has been engineered to produce the best performance for your system, and saves you from debugging time. Nonetheless, it is important to learn about sorting algorithms, since the fundamental concepts covered there apply to much more complex algorithms.

Selection: A simpler, related problem to sorting is selection. The selection problem is, given an array $A$ of n numbers (not sorted), and an integer $k$, where 1 <= $k$ <= n, return the $k$th smallest value of $A$. Although selection can be solved in $O(n \log n)$ time, by first sorting $A$ and then returning the $k$th element of the sorted list, it is possible to select the $k$th smallest element in $O(n)$ time. The algorithm is a variant of QuickSort.

Lower Bounds for Comparison-Based Sorting: The fact that $O(n \log n)$ sorting algorithms are the fastest around for many years, suggests that this may be the best that we can do. Can we sort faster? The claim is no, provided that the algorithm is comparison-based. A comparison-based sorting algorithm is one in which algorithm permutes the elements based solely on the results of the comparisons that the algorithm makes between pairs of elements.

All of the algorithms we have discussed so far are comparison-based. We will see that exceptions exist in special cases. This does not preclude the possibility of sorting algorithms whose actions are determined by other operations.

Linear Time Sorting: The $(n \log n)$ lower bound implies that if we hope to sort numbers faster than in $O(n \log n)$ time, we cannot do it by making comparisons alone. In some special cases, it is possible to sort without the use of comparisons. This leads to the possibility of sorting in linear (that is, $O(n)$) time. Here are three such algorithms.
Counting Sort: Counting sort assumes that each input is an integer in the range from 1 to \( k \). The algorithm sorts in \( \Theta(n + k) \) time. Thus, if \( k = O(n) \), this implies that the resulting sorting algorithm runs in \( \Theta(n) \) time. The algorithm requires an additional \( \Theta(n + k) \) working storage but has the nice feature that it is stable. The algorithm is remarkably simple, but deceptively clever.

Radix Sort: The main shortcoming of CountingSort is that (due to space requirements) it is only practical for a very small ranges of integers. If the integers are in the range from say, 1 to a million, we may not want to allocate an array of a million elements. RadixSort provides a nice way around this by sorting numbers one digit, or one byte, or generally, some groups of bits, at a time. As the number of bits in each group increases, the algorithm is faster, but the space requirements go up. The idea is very simple. Let’s think of our list as being composed of \( n \) integers, each having \( d \) decimal digits (or digits in any base). To sort these integers we simply sort repeatedly, starting at the lowest order digit, and finishing with the highest order digit. Since the sorting algorithm is stable, we know that if the numbers are already sorted with respect to low order digits, and then later we sort with respect to high order digits, numbers having the same high order digit will remain sorted with respect to their low order digit. An example is shown in Figure.

<table>
<thead>
<tr>
<th>Input</th>
<th>Output</th>
</tr>
</thead>
</table>

Fig.: Example of RadixSort.

The running time is \( \Theta(d(n + k)) \) where \( d \) is the number of digits in each value, \( n \) is the length of the list, and \( k \) is the number of distinct values each digit may have. The space needed is \( \Theta(n + k) \). A common application of this algorithm is for sorting integers over some range that is larger than \( n \), but still polynomial in \( n \). For example, suppose that you wanted to sort a list of integers in the range from 1 to \( n^2 \). First, you could subtract 1 so that they are now in the range from 0 to \( n^2 − 1 \). Observe that any number in this range can be expressed as a 2-digit number, where each digit is over the range from 0 to \( n − 1 \). In particular, given any integer \( L \) in this range, we can write \( L = an + b \), where \( a = \text{LB}(L/n) \) and \( b = L \mod n \). Now, we can think of \( L \) as the 2-digit number \( (a; b) \). So, we can radix sort these numbers in time \( \Theta(2(n + n)) = \Theta(n) \). In general this works to sort any \( n \) numbers over the range from 1 to \( n^d \), in \( \Theta(dn) \) time.
BucketSort: CountingSort and RadixSort are only good for sorting small integers, or at least objects (like characters) that can be encoded as small integers. What if you want to sort a set of floating-point numbers? In the worst-case you are pretty much stuck with using one of the comparison-based sorting algorithms, such as QuickSort, MergeSort, or HeapSort. However, in special cases where you have reason to believe that your numbers are roughly uniformly distributed over some range, then it is possible to do better. (Note that this is a strong assumption. This algorithm should not be applied unless you have good reason to believe that this is the case.) Suppose that the numbers to be sorted range over some interval, say \([0; 1)\). (It is possible in \(O(n)\) time to find the maximum and minimum values, and scale the numbers to fit into this range.) The idea is the subdivide this interval into \(n\) subintervals. For example, if \(n = 100\), the subintervals would be \([0; 0.01); [0.01; 0.02); [0.02; 0.03), and so on. We create \(n\) different buckets, one for each interval. Then we make a pass through the list to be sorted, and using the floor function, we can map each value to its bucket index. (In this case, the index of element \(x\) would be \(\lfloor 100x \rfloor\).) We then sort each bucket in ascending order. The number of points per bucket should be fairly small, so even a quadratic time sorting algorithm (e.g. BubbleSort or InsertionSort) should work. Finally, all the sorted buckets are concatenated together. The analysis relies on the fact that, assuming that the numbers are uniformly distributed, the number of elements lying within each bucket on average is a constant. Thus, the expected time needed to sort each bucket is \(O(1)\). Since there are \(n\) buckets, the total sorting time is \(\Theta(n)\). An example illustrating this idea is given in Fig..

Fig.: BucketSort
Section A

**Divide And Conquer**

Recall that these algorithms have the following general structure.

Divide: Divide the problem into two or more subproblems (ideally of roughly equal sizes),

Conquer: Solve each subproblem recursively, and

Combine: Combine the solutions to the subproblems into a single global solution.

**Binary search**

Let's go back to the original example -- finding matrix multiplication in Baase. I talked about looking it up in the table of contents (by sequential search) but also about looking it up in the index.

The index of Baase and most other books has the useful property that it's alphabetized, so we can be smarter about our search. For instance, we could stop the sequential search whenever we found a y>x, and this would speed up the time for x not in L. But we can be much better, and this is basically what people do in alphabetized lists.

```
binary search(x,L)
{
    let n = length of L, i=n/2.
    if (n = 0) return no match
```
else if (L[i] matches \( x \)) return L[i]

else if (L[i] > x) binary search(x,L[1..i-1])

else binary search(x,L[i+1..n])

}  

Recursion is not really necessary:

alternate search(x,L)

{
  let n = length of L
  let a = 1, b = n
  while (L[i = (a+b)/2] doesn't match)
    if (L[i] > x) b = i-1
    else a = i+1
    if a>b return no match
  return L[i]

}

Analysis: \( T(n) = O(1) + T(n/2) = O(\log n) \)

More precisely in the worst case, \( T(n) = 2 + T(\lceil(n-1)/2\rceil) \) which solves to approximately \( 2 \log n \) (logarithm to base 2).

So binary search is fast, but in order to use it we need to somehow get the list to be in sorted order -- this problem is known as sorting, and we'll see it in much detail next week.

**Merge sort**

Let's look at the combine step first. Suppose you have some data that's close to sorted -- it forms two sorted lists. You want to merge the two sorted lists quickly rather than having to resort to a general purpose sorting algorithm. This is easy enough:

merge(L1,L2)
```c
{ 
  list X = empty 
  while (neither L1 nor L2 empty) 
  { 
    compare first items of L1 & L2 
    remove smaller of the two from its list 
    add to end of X 
  } 
  concatenate remaining list to end of X 
  return X 
} 
```

**Time analysis:** in the worst case both lists empty at about same time, so everything has to be compared. Each comparison adds one item to X so the worst case is $|X| - 1 = |L1| + |L2| - 1$ comparisons. One can do a little better sometimes e.g. if $L1$ is smaller than most of $L2$.

Once we know how to combine two sorted lists, we can construct a divide and conquer sorting algorithm that simply divides the list in two, sorts the two recursively, and merges the results:

```c
merge sort(L) 
{ 
  if (length(L) < 2) return L 
  else { 
    split L into lists $L1$ and $L2$, each of $n/2$ elements 
    $L1$ = merge sort($L1$) 
    $L2$ = merge sort($L2$) 
    return merge($L1$, $L2$) 
  } 
} 
```
This is simpler than heapsort (so easier to program) and works pretty well. How many comparisons does it use? We can use the analysis of the merge step to write down a recurrence:

\[ C(n) \leq n-1 + 2C(n/2) \]

As you know, for \( n = \text{power of 2} \), the solution to this is \( n \log n - n + 1 \). For other \( n \), it's similar but more complicated. To prove this (at least the power of 2 version), you can use the formula above to produce

\[
\log n \\
C(N) \leq \sum_{i=0}^{\log n} 2^i (n/2^i - 1)
\]

= \sum_{i=0}^{\log n} n - 2^i

= n(\log n + 1) - (2n - 1)

= n \log n - n + 1

So the number of comparisons is even less than heapsort.

**Quicksort**

Quicksort, invented by Tony Hoare, follows a very similar divide and conquer idea: partition into two lists and put them back together again. It does more work on the divide side, less on the combine side.

Merge sort worked no matter how you split the lists (one obvious way is to take first \( n/2 \) and last \( n/2 \) elements, another is to take every other element). But if you could perform the splits so that everything in one list was smaller than everything in the other, this information could be used to make merging much easier: you could merge just by concatenating the lists.

How to split so one list smaller than the other? e.g. for alphabetical order, you could split into A-M, N-Z so could use some split depending on what data looks like, but we want a comparison sorting algorithm that works for any data.
Quicksort uses a simple idea: pick one object $x$ from the list, and split the rest into those before $x$ and those after $x$.

quicksort(L)
{
    if (length(L) < 2) return L
    else {
        pick some $x$ in L
        L1 = { $y$ in L : $y < x$ }
        L2 = { $y$ in L : $y > x$ }
        L3 = { $y$ in L : $y = x$ }
        quicksort(L1)
        quicksort(L2)
        return concatenation of L1, L3, and L2
    }
}

(We don't need to sort L3 because everything in it is equal).

**Quicksort analysis**

The partition step of quicksort takes $n-1$ comparisons. So we can write a recurrence for the total number of comparisons done by quicksort:

$$C(n) = n-1 + C(a) + C(b)$$

where $a$ and $b$ are the sizes of L1 and L2, generally satisfying $a+b=n-1$. In the worst case, we might pick $x$ to be the minimum element in L. Then $a=0$, $b=n-1$, and the recurrence simplifies to $C(n)=n-1 + C(n-1) = O(n^2)$. So this seems like a very bad algorithm.

Why do we call it quicksort? How can we make it less bad? Randomization!
Suppose we pick $x=a[k]$ where $k$ is chosen randomly. Then any value of $a$ is equally likely from $0$ to $n-1$. To do average case analysis, we write out the sum over possible random choices of the probability of that choice times the time for that choice. Here the choices are the values of $k$, the probabilities are all $1/n$, and the times can be described by formulas involving the time for the recursive calls to the algorithm. So average case analysis of a randomized algorithm gives a randomized recurrence:

$$C(n) = \sum_{a=0}^{n-1} \frac{1}{n}[n - 1 + C(a) + C(n-a-1)]$$

To simplify the recurrence, note that if $C(a)$ occurs one place in the sum, the same number will occur as $C(n-a-1)$ in another term -- we rearrange the sum to group the two together. We can also take the $(n-1)$ parts out of the sum since the sum of $1/n$ copies of $1/n$ times $n-1$ is just $n-1$.

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

The book gives two proofs that this is $O(n \log n)$. Of these, induction is easier.

One useful idea here: we want to prove $f(n)$ is $O(g(n))$. The $O()$ hides too much information, instead we need to prove $f(n) \leq a \cdot g(n)$ but we don't know what value of $a$ should take. We work it out with a left as a variable then use the analysis to see what values of $a$ work.

We have $C(1) = 0 = a (1 \log 1)$ for all $a$. Suppose $C(i) \leq a \cdot i \log i$ for some $a$, all $i < n$. Then

$$C(n) = n - 1 + \sum_{a=0}^{n-1} \frac{2}{n} a \cdot i \log i$$

$$= n - 1 + 2a/n \sum_{i=2}^{n-1} (i \log i)$$

$$\leq n - 1 + 2a/n \int_{2}^{n-1} (i \log i)$$

$$= n - 1 + 2a/n (n^2 \log n / 2 - n^2/4 - 2 \ln 2 + 1)$$

$$= n - 1 + a \cdot n \log n - an/2 - O(1)$$

and this will work if $n - 1 < an/2$, and in particular if $a = 2$. So we can conclude that $C(n) \leq 2 \cdot n \log n$. 
Note that this is worse than either merge sort or heap sort, and requires random number generator to avoid being really bad. But it's pretty commonly used, and can be tuned in various ways to work better. (For instance, let $x$ be the median of three randomly chosen values rather than just one value).

**Selection sort**

To understand heap sort, let's start with selection sort. An experiment: I write a list of numbers, once I'm done you tell me sorted order.

5, 2, 100, 19, 22, 7

How did you go about finding them? You probably looked through the list for the first number, then looked through it again for the next one, etc. One way of formalizing this process is called selection sort:

```
selection sort(list L)
{
    list X = empty
    while L nonempty
    {
        remove smallest element of L
        and add it to X
    }
}
```

Time analysis: there is one loop, executed $n$ times. But the total time is not $O(n)$. Remember we are counting comparisons. "Remove the smallest element of L" could take many comparisons. We need to look more carefully at this part of the loop. (The other part, adding an element to X, also depends on how we store X, but can be done in constant time for most reasonable implementations and in any case doesn't require any comparisons, which is what we're counting.)

The obvious method of finding (and removing) the smallest element: scan L and keep track of the smallest object. So this produces a nested inner loop, time = $O(\text{length of } L)$ so total time = $O(\text{sum } i) = O(n^2)$. This is one of the slow algorithms. In fact it is as slow as possible: it always makes every possible comparison.

**Strassen's Matrix Multiplication Algorithm**
• The standard method of matrix multiplication of two $n \times n$ matrices takes $O(n^3)$ operations.

• Strassen’s algorithm is a Divide-and-Conquer algorithm that is asymptotically faster, i.e. $O(n^{\log_2 7})$.

• The usual multiplication of two $2 \times 2$ matrices takes 8 multiplications and 4 additions. Strassen showed how two $2 \times 2$ matrices can be multiplied using only 7 multiplications and 18 additions.

For $2 \times 2$ matrices, there is no benefit in using the method.

• To see where this is of help, think about multiplication two $(2k) \times (2k)$ matrices.

• For this problem, the scalar multiplications and additions become matrix multiplications and additions.

• An addition of two matrices requires $O(k^2)$ time, a multiplication requires $O(k^3)$.

• Hence, multiplications are much more expensive and it makes sense to trade one multiplication operation for 18 additions.

Imagine that $A$ and $B$ are each partitioned into four square sub-matrices, each submatrix having dimensions $n/2 \times n/2$.

\[
\begin{bmatrix}
A_{11} & A_{12} \\
A_{21} & A_{22}
\end{bmatrix}
\begin{bmatrix}
B_{11} & B_{12} \\
B_{21} & B_{22}
\end{bmatrix}
= 
\begin{bmatrix}
C_{11} & C_{12} \\
C_{21} & C_{22}
\end{bmatrix}
\]

Where

$C_{11} = A_{11}B_{11} + A_{12}B_{21}$

$C_{12} = A_{11}B_{12} + A_{12}B_{22}$

$C_{21} = A_{21}B_{11} + A_{22}B_{21}$

$C_{22} = A_{21}B_{12} + A_{22}B_{22}$

Strassen ‘observed’ that:
\[
\begin{align*}
\begin{bmatrix}
C_{11} & C_{12} \\
C_{21} & C_{22}
\end{bmatrix}

&= 
\begin{bmatrix}
P_5 & P_4 - P_2 + P_6 \\
P_3 & P_4 \\
P_5 & P_1 - P_3 - P_7
\end{bmatrix}
\end{align*}
\]

where

\[
P_1 = A_{11}(B_{12} - B_{22})
\]

\[
P_2 = (A_{11} + A_{12})B_{22}
\]

\[
P_3 = (A_{21} + A_{22})B_{11}
\]

\[
P_4 = A_{22}(B_{21} - B_{11})
\]

\[
P_5 = (A_{11} + A_{22})(B_{11} + B_{22})
\]

\[
P_6 = (A_{12} - A_{22})(B_{21} + B_{22})
\]

\[
P_7 = (A_{11} - A_{21})(B_{11} + B_{12})
\]

Complexity

- \(T(n) = 7T(n/2) + cn^2\), where \(c\) is a fixed constant. The term \(cn^2\) captures the time for the matrix additions and subtractions needed to compute \(P_1, \ldots, P_7\) and \(C_{11}, \ldots, C_{22}\).

- The solution works out to be: \(T(n) = \Theta(n^{\log_2 7}) = O(n^{2.81})\).

- Currently, the best known algorithm was given by Coppersmith and Winograd and has time complexity \(O(n^{2.376})\).
Section B

Greedy Method

Greedy Algorithms: In many optimization algorithms a series of selections need to be made. In dynamic programming we saw one way to make these selections. Namely, the optimal solution is described in a recursive manner, and then is computed “bottom-up”. Dynamic programming is a powerful technique, but it often leads to algorithms with higher than desired running times. Today we will consider an alternative design technique, called greedy algorithms. This method typically leads to simpler and faster algorithms, but it is not as powerful or as widely applicable as dynamic programming. We will give some examples of problems that can be solved by greedy algorithms. Even when greedy algorithms do not produce the optimal solution, they often provide fast heuristics (nonoptimal solution strategies), are often used in finding good approximations.

The Knapsack Problem

Input: A weight capacity C, and n items of weights W[1:n] and monetary value P[1:n].
Problem: Determine which items to take and how much of each item so that the total weight is $\leq C$, and the total value (profit) is maximized.


Policy 1: Choose the lightest remaining item, and take as much of it as can fit.

Policy 2: Choose the most profitable remaining item, and take as much of it as can fit.

Policy 3: Choose the item with the highest price per unit weight ($P[i]/W[i]$), and take as much of it as can fit.

Exercise: Prove by a counter example that Policy 1 does not guarantee an optimal solution. Same with Policy 2. Policy 3 always gives an optimal solution.

Example

<table>
<thead>
<tr>
<th>i</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>P</td>
<td>5</td>
<td>9</td>
<td>4</td>
<td>8</td>
</tr>
<tr>
<td>W</td>
<td>1</td>
<td>3</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>$C=4$</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$P/W$:</td>
<td>5</td>
<td>3</td>
<td>2</td>
<td>4</td>
</tr>
</tbody>
</table>

Solution:

1st: all of item 1, $x[1]=1$, $x[1]*W[1]=1$


Now the total weight is $4=C$

(x[3]=0)

Minimum Spanning Tree (MST)
Definition of a spanning tree: A spanning tree of a graph G of n nodes is a tree that has all the n nodes of the graph such that every edge of the tree is an edge in the graph.

Definition: if the edges have weights, then the weight of a tree is the sum of the weights of the edges of the tree.

Statement of the MST problem:

Input: a weighted connected graph G=(V,E). The weights are represented by the 2D array (matrix) W[1:n,1:n], where W[i,j] is the weight of edge (i,j).

Problem: Find a minimum-weight spanning tree of G.

The greedy method for this problem works on the basis of this selection policy: choose the minimum-weight remaining edge. If that edge does not create a cycle in the evolving tree, add it to the tree.

The greedy MST algorithm:

Procedure ComputeMST(in:G, W[1:n,1:n]; out:T)

begin
    Put in T the n nodes and no edges;
    while T has less than n-1 edges do
        Choose a remaining edge e of minimum weight;
        Delete e from the graph;
        if (e does not create a cycle in T) then
            Add e to T;
        endif
    endwhile
end ComputeMST

the main implementation questions are:
how to find and delete the min-weight edges

How to tell if an edge creates a cycle in T

For finding and deleting the min-weight edge, use a minheap where its nodes are the labels+weights of the graph edges.

For cycle detection, note that T is a forest at any given time, adding an edge eliminates two trees from the forest and replaces them by a new tree containing the union of the nodes of the two old trees, and an edge e=(x,y) creates a cycle if both x and y belong to the same tree in the forest.

Therefore, a Union-Find data structure is perfect to tell if an edge creates a cycle, and to keep track of what nodes belong to each tree in the forest.

A detailed implementation code of ComputeMST:

Procedure ComputeMST(input:G,W[1:n,1:n];output:T)
begin
integer PARENT[1:n];
initialize PARENT to -1 in each entry;
Build a minheap H[1:|E|] for all the |E| edges;
Put in T the n nodes and no edges;
while T has less than n-1 edges do
   e=delete-min(H); /* assume e=(x,y)*/
   r1 := F(x); r2 := F(y);
   if (r1 != r2) then
      Add e to T;
      U(r1,r2);
   endif
endwhile
end ComputeMST

Time complexity of the MST algorithm:
O(|E|) to build the heap
up to |E| calls to U and F, taking
O(|E|log n) time
up to |E| calls to delete-min, taking
O(|E|log |E|) time.
therefore, the total time is O(|E|log |E|).

Theorem: the ComputeMST algorithms computes a minimum spanning tree.
Proof:
Let T be the tree generated by the algorithm
Let T' be a minimum spanning tree
If T=T', done. So assume that T != T'
Strategy: T' will be transformed to T without a change of weight
Let e be a min-eight edge in T-T'
All edges in T that are < W(e) are thus in T' as well.
Adding e to T' creates a cycle e1 ,e2 ,....ek
This cycle must have an edge ej that is not in T because T has no cycles. ej is in T'.
Claim: W(ej) >= W(e).
We prove the claim by contradiction.
Assume W(ej) < W(e)
ComputeMST would process ej before e
All the edges processed before e and are of weight < W(e), and which are entered into T, are also in T' (by item 6)
Therefore, when ej is processed, it would be found not to create a cycle because ej and all the edges that preceded it are all in T', and T' does not have a cycle.
Thus, ej would have to be added by the algorithm to T, contradicting the fact that ej is not in T.
Replace $e_j$ by $e$ in $T'$, resulting in a new tree $T''$.

$$W(T'') = W(T') + W(e) - W(e_j) \leq W(T').$$

Since $T'$ is a minimum spanning tree, $W(T'')$ can't be $< W(T')$.

Therefore, $W(T'') = W(T')$.

That is, $T''$ is an MST and $T''$ differs less from $T$ that $T'$ did.

This kind of edge replacement operation, which make $T'$ resemble $T$ more and more without a change of weight, can be repeated a finite number of times until $T'$ becomes identical to $T$.

Thus, $T$ and $T'$ have the same weight, making $T$ a minimum spanning tree.

**Single-Source Shortest Path problem**

Input: a weighted connected graph $G=(V,E)$, and a node $s$ designated as a source node. The weights are represented by the 2D array (matrix) $W[1:n,1:n]$, where $W[i,j]$ is the weight of edge $(i,j)$. If $(i,j)$ is not an edge, $W[i,j]=\infty$.

Note: $W[i,i]=0$ for all $i$.

Problem: Find the distance between $s$ and every node in the graph.

The greedy method here will take the definitions of some concept before it can be formulated.

Let $Y$ be a set, initially containing the single source node $s$.

Definition: A path from $s$ to a node $x$ outside $Y$ is called special if every intermediary node on the path belongs to $Y$.

Let $DIST[1:n]$ be a real array where

$DIST[i] = $the length of the shortest special path from $s$ to $i$

Greedy selection policy: choose from all the nodes still outside $Y$ the node of minimum $DIST$ value, and add it to $Y$.

The claim, which will proved later, is that every node in $Y$ has $DIST$ value equal to the distance from it to $s$.

the SSSP algorithm:

Procedure SSSP(in $W[1:n,1:n]$, $s$; out $DIST[1:n]$);

begin
for i = 1 to n do
    \[ \text{DIST}[i] := W[s, i]; \]
endfor

/* implement \( Y \) is Boolean array \( Y[1:n] \) */
/* \( Y[i] = 1 \) if \( i \) belong to set \( Y \), 0 otherwise */

\( \text{Boolean} \ Y[1:n]; \) /* initialized to 0*/
\( Y[s] := 1; \) /* add \( s \) to set \( Y \) */

for num = 2 to n-1 do
    choose a node \( u \) from out of \( Y \) such that
    \[ \text{DIST}[u] = \min\{\text{DIST}[i] \mid Y[i] = 0\}; \]
    \( Y[u] := 1; \) /* Add \( u \) to \( Y \); */
    /* update the \( \text{DIST} \) values of the other nodes*/
    for all node \( w \) where \( Y[w] = 0 \) do
        \[ \text{DIST}[w] = \min(\text{DIST}[w], \text{DIST}[u] + W[u, w]); \]
    endfor
endfor

Time Complexity of the SSSP algorithm:
the 1st for-loop clearly takes \( O(n) \) time
Choosing \( u \) takes \( O(n) \) time, because it involves finding a minimum in an array.
The innermost for-loop for updating \( \text{DIST} \) has a constant-time body, and iterates at most \( n \) times, thus takes \( O(n) \) time.
Therefore, the for-loop iterating over num takes $O(n^2)$ time.

Thus, SSSp takes $O(n^2)$ time.

Theorem: When a node $u$ enters $Y$, we have

$$\text{DIST}[u] = \text{distance}(s, u).$$

Proof:

The proof is by induction on the number $k$ of elements in $Y$.

Basis: $k=1$. That is, $Y$ has only node $s$. Well $\text{DIST}[s] = W[s, s] = 0$; also, $\text{distance}(s, s) = 0$. Thus, $\text{DIST}[s] = \text{distance}(s, s)$.

Induction: assume the theorem holds for every node $v$ that had entered $Y$ before $u$. Prove that the theorem holds for $u$ which is selected by the algorithm to be the next node to enter $Y$. We do so by contradiction.

Assume that $\text{DIST}[u] \neq \text{distance}(s, u)$. That is, $\text{DIST}[u] > \text{distance}(s, u)$.

This means the shortest path from $s$ to $u$ (call that path $P$) is not a special path.

This implies that at some point, $P$ exits $Y$ going through some intermediary node(s) before reaching $u$.

Let $z$ be the first node that $P$ goes through right when $P$ exits $Y$.

Then, the portion of $P$ from $s$ to $z$, which we'll call $Q$, is a special path from $s$ to $z$.

We now have

$$\text{DIST}[z] \leq \text{length}(Q) \leq \text{length}(P) = \text{distance}(s, u) < \text{DIST}[u]$$

That is, $\text{DIST}[z] < \text{DIST}[u]$.

This contradicts the fact that the algorithm choose the min-DIST node $u$ to enter $Y$.

Therefore, $\text{DIST}[u] = \text{distance}(s, u)$. 
Section B

Dynamic Programming

Dynamic Programming: We begin discussion of an important algorithm design technique, called dynamic programming (or DP for short). The technique is among the most powerful for designing algorithms for optimization problems. (This is true for two reasons. Dynamic programming solutions are based on a few common elements. Dynamic programming problems are typically
optimization problems (find the minimum or maximum cost solution, subject to various constraints). The technique is related to divide-and-conquer, in the sense that it breaks problems down into smaller problems that it solves recursively. However, because of the somewhat different nature of dynamic programming problems, standard divide-and-conquer solutions are not usually efficient. The basic elements that characterize a dynamic programming algorithm are:

Substructure: Decompose your problem into smaller (and hopefully simpler) subproblems. Express the solution of the original problem in terms of solutions for smaller problems.

Table-structure: Store the answers to the subproblems in a table. This is done because subproblem solutions are reused many times.

Bottom-up computation: Combine solutions on smaller subproblems to solve larger subproblems. (Our text also discusses a top-down alternative, called memoization.)

Optimal substructure: (Sometimes called the principle of optimality.) It states that for the global problem to be solved optimally, each subproblem should be solved optimally. (Not all optimization problems satisfy this. Sometimes it is better to lose a little on one subproblem in order to make a big gain on another.)

Polynomially many subproblems: An important aspect to the efficiency of DP is that the total number of subproblems to be solved should be at most a polynomial number.

**Optimal Binary Search Trees**

We shall further illustrate the principles of Dynamic Programming with the construction of Optimal Binary Search trees.

A binary search tree is a tree where the key values are stored in the internal nodes, the external nodes (leaves) are null nodes, and the keys are ordered lexicographically. I.e. for each internal node all the keys in the left subtree are less than the keys in the node, and all the keys in the right subtree are greater.

When we know the probabilities of searching each one of the keys, it is quite easy to compute the expected cost of accessing the tree. An OBST is a BST which has minimal expected cost.

Example:

<table>
<thead>
<tr>
<th>Key</th>
</tr>
</thead>
<tbody>
<tr>
<td>-5</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>8</td>
</tr>
<tr>
<td>7</td>
</tr>
<tr>
<td>13</td>
</tr>
<tr>
<td>21</td>
</tr>
</tbody>
</table>
Probabilities 1/8 1/32 1/16 1/32 1/4 1/2

The expectation-value of a search is:
\[ E_{\text{cost}} = 1 \cdot 1/16 + 2 \cdot (1/32 + 1/4) + 3 \cdot (1/8 + 1/32 + 1/2) \]

It's clear that this tree is not optimal. It is easy to see that if the 21 is closer to the root, given its high probability, the tree will have a lower expected cost.

Criterion for an optimal tree: Each optimal binary search tree is composed of a root and (at most) two optimal subtrees, the left and the right.

Method:
The criterion for optimality gives a dynamic programming algorithm. For the root (and each node in turn) we select one value to be stored in the node. (We have n possibilities to do that.)

Once this choice is made, the set of keys which go into the left subtree and right subtree is completely defined, because the tree is lexicographically ordered. The left and right subtrees are now constructed recursively (optimally). This gives the recursive definition of the optim cost:

Let \( p_i \) denote the probability of accessing key \( i \), let \( p_{ij} \) denote the sum of the probabilities from \( p_i \) to \( p_j \).

\[ T_{ij} = \min_{k=i \ldots j} \left( p_{ik} \cdot (1 + T_{ik-1}) + p_{k} \cdot 1 + p_{kj} \cdot (1 + T_{kj+1}) \right) \]

The explanation of the formula is easy once we see that the first term corresponds to the left subtree, which is one level lower than the root, the second term corresponds to the root and the 3rd to the right subtree. Every cost is multiplied by its probability. For simplicity we set \( p_{i,i-1} = 0 \) and \( p_{i+1,i} = 0 \) so \( T_H \) simplifies to \( T_{ij} = \frac{1}{p_{ij}} = 1 \). This procedure is exponential if applied directly. However, the optimal trees are only constructed over contiguous sets of keys, and there are at most different sets of contiguous keys.

In this case we store the optimal cost of a subtree in a matrix. The Matrix-entry \( T_{ij} \) will contain the cost of an optimal subtree constructed with the keys \( i \) to \( j \).

We now fill the matrix diagonal by diagonal. It is customary to fill the matrix with \( p_{ij} \cdot T_{ij,00} \) that we save a lot of multiplications and divisions. Let \( T_{ij} = p_{ij} \cdot T_{ij} \) then
\[ T_{0,0} = \min_{k=0} \left( T_{k,0} + p_k \right) \]

An optimal tree with one node is just the node itself (no other choice), so the diagonal of \( T^* \) is easy to fill: \( T_{ii} = p_i \)

\[
T^* = \begin{bmatrix}
1 & 2 & 3 & 4 & 5 & 6 \\
2 & 0 & 2 & 3 & 4 & 5 \\
3 & 0 & 0 & 1 & 2 & 3 \\
4 & 0 & 0 & 1 & 4 & 5 \\
5 & 0 & 0 & 1 & 0 & 4 \\
6 & 0 & 0 & 1 & 0 & 0 \\
\end{bmatrix} = \frac{1}{62} \times \begin{bmatrix}
4 & 6 & 0 & 15 & 31 & 63 \\
1 & 3 & 7 & 19 & 47 & 0 \\
1 & 4 & 15 & 42 & 0 & 0 \\
2 & 12 & 38 & 0 & 0 & 0 \\
8 & 32 & 16 & 0 & 0 & 0 \\
16 & 0 & 0 & 0 & 0 & 0 \\
\end{bmatrix}
\]

The cost of the \( OBST \) is in \( T_{1,0} \) (\( T_{1,0} \) in our example) And you can see, that it is practical, not to work with the probabilities, but with the frequencies (i.e. the probabilities times the least common multiple of their denominators) to avoid fractions as matrix-entries.

0-1 knapsack problem

A similar dynamic programming solution for the 0-1 knapsack problem also runs in pseudo-polynomial time. As above, assume are strictly positive integers. Define \( m[i,w] \) to be the maximum value that can be attained with weight less than or equal to \( w \) using items up to \( i \).

We can define \( m[i,w] \) recursively as follows:

\( m[0,w] = 0 \)
\( m[i,0] = 0 \)
\( m[i,w] = m[i-1,w] \); if \( w > w \) (the new item is more than the current weight limit)

\[
m[i, w] = \max \left( m[i-1, w], m[i-1, w - w_i] + v_i \right) \text{ if } w_i \leq w.
\]

The solution can then be found by calculating \( m[n,W] \). To do this efficiently we can use a table to store previous computations. This solution will therefore run in \( O(nW) \) time and \( O(nW) \) space.

**Travelling salesman problem**

The Travelling Salesman Problem (TSP) is an NP-complete problem in combinatorial optimization studied in operations research and theoretical computer science. Given a list of
cities and their pairwise distances, the task is to find a shortest possible tour that visits each city exactly once.

The problem was first formulated as a mathematical problem in 1930 and is one of the most intensively studied problems in optimization. It is used as a benchmark for many optimization methods. Even though the problem is computationally difficult, a large number of heuristics and exact methods are known, so that some instances with tens of thousands of cities can be solved.

The TSP has several applications even in its purest formulation, such as planning, logistics, and the manufacture of microchips. Slightly modified, it appears as a sub-problem in many areas, such as DNA sequencing. In these applications, the concept city represents, for example, customers, soldering points, or DNA fragments, and the concept distance represents travelling times or cost, or a similarity measure between DNA fragments. In many applications, additional constraints such as limited resources or time windows make the problem considerably harder.

In the theory of computational complexity, the decision version of the TSP belongs to the class of NP-complete problems. Thus, it is assumed that there is no efficient algorithm for solving TSPs. In other words, it is likely that the worst case running time for any algorithm for the TSP increases exponentially with the number of cities, so even some instances with only hundreds of cities will take many CPU years to solve exactly.

Description

As a graph problem

Symmetric TSP with four cities, such that cities are the graph's vertices, paths are the graph's edges, and a path's distance is the edge's length. A TSP tour becomes a Hamiltonian cycle, and the optimal TSP tour is the shortest Hamiltonian cycle. Often, the model is a complete graph (i.e., an edge connects each pair of vertices). If no path exists between two cities, adding an arbitrarily long edge will complete the graph without affecting the optimal tour.
In the symmetric TSP, the distance between two cities is the same in each opposite direction, forming an undirected graph. This symmetry halves the number of possible solutions. In the asymmetric TSP, paths may not exist in both directions or the distances might be different, forming a directed graph. Traffic collisions, one-way streets, and airfares for cities with different departure and arrival fees are examples of how this symmetry could break down.

With metric distances

In the metric TSP, also known as delta-TSP or $\Delta$-TSP, the intercity distances satisfy the triangle inequality. This can be understood as “no shortcuts”, in the sense that the direct connection from A to B is never longer than the detour via C: $C_{ij} \leq C_{ik} + C_{kj}$

These edge lengths define a metric on the set of vertices. When the cities are viewed as points in the plane, many natural distance functions are metrics.

In the Euclidian TSP the distance between two cities is the Euclidean distance between the corresponding points.

In the Rectilinear TSP the distance between two cities is the sum of the differences of their x- and y-coordinates. This metric is often called the Manhattan distance or city-block metric.

In the maximum metric, the distance between two points is the maximum of the differences of their x- and y-coordinates.

The last two metrics appear for example in routing a machine that drills a given set of holes in a printed circuit board. The Manhattan metric corresponds to a machine that adjusts first one co-ordinate, and then the other, so the time to move to a new point is the sum of both movements. The maximum metric corresponds to a machine that adjusts both co-ordinates simultaneously, so the time to move to a new point is the slower of the two movements.

Despite the simplification over the general case, the metric TSP problem with an arbitrary metric is still NP-complete.

Non-metric distances

Distance measures that do not satisfy the triangle inequality arise in many routing problems. For example, one mode of transportation, such as travel by airplane, may be faster, even though it covers a longer distance.

In its definition, the TSP does not allow cities to be visited twice, but many applications do not need this constraint. In such cases, a symmetric, non-metric instance can be reduced to a metric one. This replaces the original graph with a complete graph in which the inter-city distance $c_{ij}$ is replaced by the shortest path between i and j in the original graph.
Related problems

An equivalent formulation in terms of graph theory is: Given a complete weighted graph (where the vertices would represent the cities, the edges would represent the roads, and the weights would be the cost or distance of that road), find a Hamiltonian cycle with the least weight.

The requirement of returning to the starting city does not change the computational complexity of the problem, see Hamiltonian path problem.

Another related problem is the bottleneck travelling salesman problem (bottleneck TSP): Find a Hamiltonian cycle in a weighted graph with the minimal weight of the weightiest edge. The problem is of considerable practical importance, apart from evident transportation and logistics areas. A classic example is in printed circuit manufacturing: scheduling of a route of the drill machine to drill holes in a PCB. In robotic machining or drilling applications, the "cities" are parts to machine or holes (of different sizes) to drill, and the "cost of travel" includes time for retooling the robot (single machine job sequencing problem).

The generalized travelling salesman problem deals with "states" that have (one or more) "cities" and the salesman has to visit exactly one "city" from each "state". Also known as the "travelling politician problem". One application is encountered in ordering a solution to the cutting stock problem in order to minimise knife changes. Another is concerned with drilling in semiconductor manufacturing, demonstrated that the generalised travelling salesman problem can be transformed into a standard travelling salesman problem with the same number of cities, but a modified distance matrix.

The sequential ordering problem deals with the problem of visiting a set of cities where precedence relations between the cities exist.

The travelling purchaser problem deals with a purchaser who is charged with purchasing a set of products. He can purchase these products in several cities, but at different prices and not all cities offer the same products. The objective is to find a route between a subset of the cities, which minimizes total cost (travel cost + purchasing cost) and which enables the purchase of all required products.

Computing a solution

The traditional lines of attack for the NP-hard problems are the following:

Devising algorithms for finding exact solutions (they will work reasonably fast only for relatively small problem sizes).

Devising "suboptimal" or heuristic algorithms, i.e., algorithms that deliver either seemingly or probably good solutions, but which could not be proved to be optimal.
Finding special cases for the problem ("subproblems") for which either better or exact heuristics are possible.

Computational complexity

The problem has been shown to be NP-hard (more precisely, it is complete for the complexity class FPNP; see function problem), and the decision problem version ("given the costs and a number x, decide whether there is a round-trip route cheaper than x") is NP-complete. The bottleneck travelling salesman problem is also NP-hard. The problem remains NP-hard even for the case when the cities are in the plane with Euclidean distances, as well as in a number of other restrictive cases. Removing the condition of visiting each city "only once" does not remove the NP-hardness, since it is easily seen that in the planar case there is an optimal tour that visits each city only once (otherwise, by the triangle inequality, a shortcut that skips a repeated visit would not increase the tour length).

Complexity of approximation

In the general case, finding a shortest travelling salesman tour is NPO-complete. If the distance measure is a metric and symmetric, the problem becomes APX-complete and Christofides’s algorithm approximates it within $3/2$. If the distances are restricted to 1 and 2 (but still are a metric) the approximation ratio becomes $7/6$. In the asymmetric, metric case, only logarithmic performance guarantees are known, the best current algorithm achieves performance ratio $0.814 \log n$; it is an open question if a constant factor approximation exists.

The corresponding maximization problem of finding the longest travelling salesman tour is approximable within $63/38$. If the distance function is symmetric, the longest tour can be approximated within $4/3$ by a deterministic algorithm and within $(33 + \varepsilon) / 25$ by a randomised algorithm.

Exact algorithms

The most direct solution would be to try all permutations (ordered combinations) and see which one is cheapest (using brute force search). The running time for this approach lies within a polynomial factor of $O(n!)$, the factorial of the number of cities, so this solution becomes impractical even for only 20 cities. One of the earliest applications of dynamic programming is an algorithm that solves the problem in time $O(n^2 2^n)$.

The dynamic programming solution requires exponential space. Using inclusion–exclusion, the problem can be solved in time within a polynomial factor of $2^n$ and polynomial space.

Improving these time bounds seems to be difficult. For example, it is an open problem if there exists an exact algorithm for TSP that runs in time $O(1.9999^n)$.
Other approaches include:

Various branch-and-bound algorithms, which can be used to process TSPs containing 40-60 cities. Progressive improvement algorithms which use techniques reminiscent of linear programming. Works well for up to 200 cities. Implementations of branch-and-bound and problem-specific cut generation; this is the method of choice for solving large instances. This approach holds the current record, solving an instance with 85,900 cities.
Section C

Back Tracking

Concept of Backtracking Method:

In the backtracking method, we need to solve the problem by systematic process until we get the solution. In the 3rd or 4th step if we find difficult to get the solution or there is no way to get the solution means we need to move to the previous step and make changes over there to get a feasible solution in the next step this is called backtracking.

Backtracking is a systematic method for generating all (or subsets of) combinatorial objects.

Examples of combinatorial objects include
- Binary strings of n bits
- Subsets of a given set E of n elements
- Directed graphs of n nodes
- Undirected graphs of n nodes
- Permutations of a given size n
- Hamiltonian cycles of a given graph
- K-cliques of a given graph
- K-colorings of a given graph

The General Backtracking Algorithm

The algorithm will generate all valid arrays X[1:N] whose elements come from the domain S={a1,a2,...,am} of successive integers, such that the constraints C are satisfied.

The algorithm is a depth-first search like traversal (or generation) of the tree that represents the entire solution space.

In the tree, the root designates the starting point, and every path from the root to a leaf is of length N, where the i-th node in level specifies a value for element X[i]. The whole path corresponds to the whole array and represents a single solution, that is, a single object.

During the generation of the tree, when we are to create a new node corresponding to X[i], we try to assign X[i] a new the next domain value (given the current value of X[i] as reference).

If that value does not violate the constraints C, it is assigned.
If, on the other hand, that value violates C, the next value after that is tried, and so on until either a C-compliant value is found or all remaining values are exhausted.

If a C-compliant value is found and assigned to X[i], we move to the next level to find a value for X[i+1].

If no C-compliant remaining value is found for X[i], we backtrack to the previous level to find a new value for X[i-1].

When we backtrack to the root, the whole tree has been fully generated, and the algorithm stops.

Whenever a C-compliant value for X[n] is found, a complete new object has been generated, and the path from the root to that node corresponding to X[n] is printed out as the object.

Note that in the algorithm, when a new node and a new value for X[i] is being generated, the values that are tried are the "next" values from a reference value, which is the current value of X[i].

At the outset, the reference value is initialized be a value a0 = a1 - 1.

That way, the next value is always the reference (current) value + 1.

The full algorithm follows:

Procedure Backtrack()
begin
  Integer r := 1;  -- r is the tree level, or index of X
  Integer X[1:N];
  for i=1 to N do
    X[i] := a0;
  endfor
  while r > 0 do
    getnext(X,r);   -- assigns to X[r] its next
    -- C-compliant value, if available.
  endwhile
end
-- Otherwise, it re-initializes X[r]

-- to a0

if (X[r] = a0) then
    r := r-1; -- backtrack to the previous level
elseif r=n
    print(X[1:N]); -- a new complete solution
else
    r := r+1; -- move to the next level for X[r+1]
endif
endwhile
end

Procedure getnext(input/output: X[1:N]; input: r)
begin
    X[r] := X[r] + 1; -- next tentative value
    while (X[r] is still in the domain) do
        if (Bound(X[1:N],r) is true) then
            return;
        else
            X[r] := X[r] + 1;
        endif
    endwhile
    -- if getnext has not returned,
    -- that mean no C-compliant remaining
    -- value was found. Re-initialize X[r]
    X[r] := a0;
The Bound function checks to see if the constraints C are satisfied. The actual implementation of Bound is problem-dependent.

In every backtracking problem, you need to do the following:

1. Represent the object by an array X[1:N], specifying N, what each X[i] signifies, what the domain of each X[i] is, the value of a0, and the constraints C.
2. Implement the Bound function

**Concept of N-Queens Problem:**

The concept of N queens problem is that we need to place N queens in N x N matrix where in which the following conditions should apply.

- No two queens should be placed in the same row
- No two queens should be placed in the same column
- No two queens should be placed in the same diagonal

Example here we are going to solve the 4 queen's problem:

Here we have Q1, Q2, Q3, and Q4.

The Q1 should be place in row /column 1
The Q2 should be place in row/column 2
The Q3 should be place in row/column 3
The Q4 should be place in row/column 4

Consider the matrix

```
   c1  c2  c3  c4
r1
r2
```
Now place Q1 in row1/column1 so,

<table>
<thead>
<tr>
<th></th>
<th>c1</th>
<th>c2</th>
<th>c3</th>
<th>c4</th>
</tr>
</thead>
<tbody>
<tr>
<td>r1</td>
<td>Q1</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Now Q2 should be placed in row2/column2, if we place in c2xr2 the queens will be crossed because they are in the same diagonal. Therefore, the Q2 should be placed in c3xr2

<table>
<thead>
<tr>
<th></th>
<th>c1</th>
<th>c2</th>
<th>c3</th>
<th>c4</th>
</tr>
</thead>
<tbody>
<tr>
<td>r1</td>
<td>Q1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>r2</td>
<td></td>
<td>Q2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>r3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>r4</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Now place Q3 in row3/column3, if we place in c3r3 the queens meet each other because they are in the same diagonal. Therefore, there is no solution the place the Q3 in the 3rd row/column.

<table>
<thead>
<tr>
<th></th>
<th>c1</th>
<th>c2</th>
<th>c3</th>
<th>c4</th>
</tr>
</thead>
<tbody>
<tr>
<td>r1</td>
<td>Q1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>r2</td>
<td>Q2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>r3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>r4</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

So, back track again, place the Q1 some where in the row, so move the Queen Q1 to c3r1 and also move the position of Q2 so that these two queens should not affect the Q3.

So place Q1 in r1c3.
Place Q2 in r2c1

<table>
<thead>
<tr>
<th></th>
<th>c1</th>
<th>c2</th>
<th>c3</th>
<th>c4</th>
</tr>
</thead>
<tbody>
<tr>
<td>r1</td>
<td>Q1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>r2</td>
<td>Q2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>r3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>r4</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Now place Q3 in row3/column3. We cannot place in either r3c1 or r3c2 or r3c3 because the queens will meet each other. So place the queen in Q3 in r3c4.

<table>
<thead>
<tr>
<th></th>
<th>c1</th>
<th>c2</th>
<th>c3</th>
<th>c4</th>
</tr>
</thead>
<tbody>
<tr>
<td>r1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>r2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>r3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>r4</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Now place Q4 in row4/column4. We cannot place Q4 in c1 or c3 or c4 so only solution is to place the Q4 in c2x4r4.

<table>
<thead>
<tr>
<th>r1</th>
<th>c1</th>
<th>c2</th>
<th>c3</th>
<th>c4</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Q1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>r2</td>
<td></td>
<td>Q2</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Q3</td>
<td></td>
</tr>
<tr>
<td>r4</td>
<td></td>
<td></td>
<td></td>
<td>Q4</td>
</tr>
</tbody>
</table>

Now we got the solution in which we place the entire queen in 4x4 matrixes were in which no two queens meet each other.

**Hamiltonian paths and cycles:**

In the mathematical field of graph theory, a Hamiltonian path (or traceable path) is a path in an undirected graph which visits each vertex exactly once. A Hamiltonian cycle (or Hamiltonian circuit) is a cycle in an undirected graph which visits each vertex exactly once and also returns to the starting vertex. Determining whether such paths and cycles exist in graphs is the Hamiltonian path problem which is NP-complete.
Hamiltonian paths and cycles are named after William Rowan Hamilton who invented the Icosian game, now also known as Hamilton's puzzle, which involves finding a Hamiltonian cycle in the edge graph of the dodecahedron. Hamilton solved this problem using the Icosian Calculus, an algebraic structure based on roots of unity with many similarities to the quaternions (also invented by Hamilton). This solution does not generalize to arbitrary graphs.

A Hamiltonian path or traceable path is a path that visits each vertex exactly once. A graph that contains a Hamiltonian path is called a traceable graph. A graph is Hamilton-connected if for every pair of vertices there is a Hamiltonian path between the two vertices.

A Hamiltonian cycle, Hamiltonian circuit, vertex tour or graph cycle is a cycle that visits each vertex exactly once (except the vertex which is both the start and end, and so is visited twice). A graph that contains a Hamiltonian cycle is called a Hamiltonian graph.

Similar notions may be defined for directed graphs, where each edge (arc) of a path or cycle can only be traced in a single direction (i.e., the vertices are connected with arrows and the edges traced "tail-to-head").

A Hamiltonian decomposition is an edge decomposition of a graph into Hamiltonian circuits.

Hamiltonian Circuit Algorithm using Backtracking Method.

This Article explains about Hamiltonian circuit algorithm in Design and Analysis of Algorithm using Backtracking Method.

This article is mainly useful for the beginners those who need to learn about the hamiltonian circuit algorithm very easily.

Concept of Backtracking Method:

In the backtracking method, we need to solve the problem by step by step process until we get the solution. In the 3rd or 4th step if we find difficult to get the solution or there is no way to get the solution means we need to move to the previous step and make changes over there to get a feasible solution in the next step this is called backtracking.

Hamiltonian Circuit algorithm.
This Algorithm is round trip paths were in which, in the given graph we need to visit/traverse each vertex exactly once and move again to the starting position. The starting and ending vertex should be same.

Example:

Consider the given graph:

```
  a---------------------------b
     |
     c-------------------------f
     |
     d--------------------------e
```

Note: All the Vertex in the above graph have an undirected connectivity among each other.

connectivities of the graph.

a-c-e
b-c-d
e-f
c-e
a-d,b-e

In the above given graph we need to visit every vertex exactly once and we should again return to the starting position a.

Note: this is an undirected graph.

Undirected graph: Undirected graph is a graph in which we can travel in any direction.

Step 1: starting with the vertex a and visit all the vertex and try to end with the starting positing (that is end up with vertex a)

Pass 1: a->b->c->d->e->f-> so dead end comes… back track again.

Step 2: Again start with vertex a and change the path/direction of the graph because this is an undirected graph

Pass 2: a->c->b->e->f so dead end again occurs so back track again.

Step 3: Again start with Vertex a and change the path and try to get the solution.
Section C

Branch and Bound

Branch and bound (BB) is a general algorithm for finding optimal solutions of various optimization problems, especially in discrete and combinatorial optimization. It consists of a systematic enumeration of all candidate solutions, where large subsets of fruitless candidates are discarded en masse, by using upper and lower estimated bounds of the quantity being optimized.

The method was first proposed by A. H. Land and A. G. Doig in 1960 for linear programming.

General description

For definiteness, we assume that the goal is to find the minimum value of a function $f(x)$, where $x$ ranges over some set $S$ of admissible or candidate solutions (the search space or feasible region). Note that one can find the maximum value of $f(x)$ by finding the minimum of $g(x) = -f(x)$. (For example, $S$ could be the set of all possible trip schedules for a bus fleet, and $f(x)$ could be the expected revenue for schedule $x$.)

A branch-and-bound procedure requires two tools. The first one is a splitting procedure that, given a set $S$ of candidates, returns two or more smaller sets whose union covers $S$. Note that the minimum of $f(x)$ over $S$ is $\min_v \min_i f_i(x)$, where each $v_i$ is the minimum of $f(x)$ within $S_i$. This step is called branching, since its recursive application defines a tree structure (the search tree) whose nodes are the subsets of $S$.

Another tool is a procedure that computes upper and lower bounds for the minimum value of $f(x)$ within a given subset $S$. This step is called bounding.

The key idea of the BB algorithm is: if the lower bound for some tree node (set of candidates) $A$ is greater than the upper bound for some other node $B$, then $A$ may be safely discarded from the search. This step is called pruning, and is usually implemented by maintaining a global variable
m (shared among all nodes of the tree) that records the minimum upper bound seen among all subregions examined so far. Any node whose lower bound is greater than m can be discarded.

The recursion stops when the current candidate set S is reduced to a single element; or also when the upper bound for set S matches the lower bound. Either way, any element of S will be a minimum of the function within S.

Effective subdivision

The efficiency of the method depends strongly on the node-splitting procedure and on the upper and lower bound estimators. All other things being equal, it is best to choose a splitting method that provides non-overlapping subsets.

Ideally the procedure stops when all nodes of the search tree are either pruned or solved. At that point, all non-pruned subregions will have their upper and lower bounds equal to the global minimum of the function. In practice the procedure is often terminated after a given time; at that point, the minimum lower bound and the minimum upper bound, among all non-pruned sections, define a range of values that contains the global minimum. Alternatively, within an overriding time constraint, the algorithm may be terminated when some error criterion, such as (max - min)/(min + max), falls below a specified value.

The efficiency of the method depends critically on the effectiveness of the branching and bounding algorithms used; bad choices could lead to repeated branching, without any pruning, until the sub-regions become very small. In that case the method would be reduced to an exhaustive enumeration of the domain, which is often impractically large. There is no universal bounding algorithm that works for all problems, and there is little hope that one will ever be found; therefore the general paradigm needs to be implemented separately for each application, with branching and bounding algorithms that are specially designed for it.

Branch and bound methods may be classified according to the bounding methods and according to the ways of creating/inspecting the search tree nodes.

The branch-and-bound design strategy is very similar to backtracking in that a state space tree is used to solve a problem. The differences are that the branch-and-bound method (1) does not limit us to any particular way of traversing the tree and (2) is used only for optimization problems.

This method naturally lends itself for parallel and distributed implementations, see, e.g., the traveling salesman problem article.

Applications

This approach is used for a number of NP-hard problems, such as

Knapsack problem
Integer programming
Nonlinear programming
Traveling salesman problem (TSP)
Quadratic assignment problem (QAP)
Maximum satisfiability problem (MAX-SAT)
Nearest neighbor search (NNS)
Cutting stock problem
False noise analysis (FNA)

Branch-and-bound may also be a base of various heuristics. For example, one may wish to stop branching when the gap between the upper and lower bounds becomes smaller than a certain threshold. This is used when the solution is "good enough for practical purposes" and can greatly reduce the computations required. This type of solution is particularly applicable when the cost function used is noisy or is the result of statistical estimates and so is not known precisely but rather only known to lie within a range of values with a specific probability. An example of its application here is in biology when performing cladistic analysis to evaluate evolutionary relationships between organisms, where the data sets are often impractically large without heuristics.

For this reason, branch-and-bound techniques are often used in game tree search algorithms, most notably through the use of alpha-beta pruning.

The Travelling Salesman Problem

The Travelling Salesman Problem (TSP) is an NP-complete problem in combinatorial optimization studied in operations research and theoretical computer science. Given a list of cities and their pairwise distances, the task is to find a shortest possible tour that visits each city exactly once.

The problem was first formulated as a mathematical problem in 1930 and is one of the most intensively studied problems in optimization. It is used as a benchmark for many optimization methods. Even though the problem is computationally difficult, a large number of heuristics and exact methods are known, so that some instances with tens of thousands of cities can be solved.

The TSP has several applications even in its purest formulation, such as planning, logistics, and the manufacture of microchips. Slightly modified, it appears as a sub-problem in many areas, such as DNA sequencing. In these applications, the concept city represents, for example,
customers, soldering points, or DNA fragments, and the concept distance represents travelling times or cost, or a similarity measure between DNA fragments. In many applications, additional constraints such as limited resources or time windows make the problem considerably harder.

In the theory of computational complexity, the decision version of the TSP belongs to the class of NP-complete problems. Thus, it is assumed that there is no efficient algorithm for solving TSPs. In other words, it is likely that the worst case running time for any algorithm for the TSP increases exponentially with the number of cities, so even some instances with only hundreds of cities will take many CPU years to solve exactly.

**Travelling salesman problem**

The Travelling Salesman Problem (TSP) is an NP-complete problem in combinatorial optimization studied in operations research and theoretical computer science. Given a list of cities and their pairwise distances, the task is to find a shortest possible tour that visits each city exactly once.

The problem was first formulated as a mathematical problem in 1930 and is one of the most intensively studied problems in optimization. It is used as a benchmark for many optimization methods. Even though the problem is computationally difficult, a large number of heuristics and exact methods are known, so that some instances with tens of thousands of cities can be solved.

The TSP has several applications even in its purest formulation, such as planning, logistics, and the manufacture of microchips. Slightly modified, it appears as a sub-problem in many areas, such as DNA sequencing. In these applications, the concept city represents, for example, customers, soldering points, or DNA fragments, and the concept distance represents travelling times or cost, or a similarity measure between DNA fragments. In many applications, additional constraints such as limited resources or time windows make the problem considerably harder.

In the theory of computational complexity, the decision version of the TSP belongs to the class of NP-complete problems. Thus, it is assumed that there is no efficient algorithm for solving TSPs. In other words, it is likely that the worst case running time for any algorithm for the TSP increases exponentially with the number of cities, so even some instances with only hundreds of cities will take many CPU years to solve exactly.

**Description**

As a graph problem

Symmetric TSP with four cities, such that cities are the graph's vertices, paths are the graph's edges, and a path's distance is the edge's length. A TSP tour becomes a Hamiltonian cycle, and the optimal TSP tour is the shortest Hamiltonian cycle. Often, the model is a complete graph
(i.e., an edge connects each pair of vertices). If no path exists between two cities, adding an arbitrarily long edge will complete the graph without affecting the optimal tour.

Asymmetric and symmetric

In the symmetric TSP, the distance between two cities is the same in each opposite direction, forming an undirected graph. This symmetry halves the number of possible solutions. In the asymmetric TSP, paths may not exist in both directions or the distances might be different, forming a directed graph. Traffic collisions, one-way streets, and airfares for cities with different departure and arrival fees are examples of how this symmetry could break down.

With metric distances

In the metric TSP, also known as delta-TSP or Δ-TSP, the intercity distances satisfy the triangle inequality. This can be understood as “no shortcuts”, in the sense that the direct connection from A to B is never longer than the detour via C: \[ c_{ij} \leq c_{ik} + c_{kj} \]

These edge lengths define a metric on the set of vertices. When the cities are viewed as points in the plane, many natural distance functions are metrics.

In the Euclidian TSP the distance between two cities is the Euclidean distance between the corresponding points.

In the Rectilinear TSP the distance between two cities is the sum of the differences of their x- and y-coordinates. This metric is often called the Manhattan distance or city-block metric.

In the maximum metric, the distance between two points is the maximum of the differences of their x- and y-coordinates.

The last two metrics appear for example in routing a machine that drills a given set of holes in a printed circuit board. The Manhattan metric corresponds to a machine that adjusts first one coordinate, and then the other, so the time to move to a new point is the sum of both movements.
The maximum metric corresponds to a machine that adjusts both co-ordinates simultaneously, so the time to move to a new point is the slower of the two movements.

Despite the simplification over the general case, the metric TSP problem with an arbitrary metric is still NP-complete.

Non-metric distances

Distance measures that do not satisfy the triangle inequality arise in many routing problems. For example, one mode of transportation, such as travel by airplane, may be faster, even though it covers a longer distance.

In its definition, the TSP does not allow cities to be visited twice, but many applications do not need this constraint. In such cases, a symmetric, non-metric instance can be reduced to a metric one. This replaces the original graph with a complete graph in which the inter-city distance \( c_{ij} \) is replaced by the shortest path between \( i \) and \( j \) in the original graph.

Related problems

An equivalent formulation in terms of graph theory is: Given a complete weighted graph (where the vertices would represent the cities, the edges would represent the roads, and the weights would be the cost or distance of that road), find a Hamiltonian cycle with the least weight.

The requirement of returning to the starting city does not change the computational complexity of the problem, see Hamiltonian path problem.

Another related problem is the bottleneck travelling salesman problem (bottleneck TSP): Find a Hamiltonian cycle in a weighted graph with the minimal weight of the weightiest edge. The problem is of considerable practical importance, apart from evident transportation and logistics areas. A classic example is in printed circuit manufacturing: scheduling of a route of the drill machine to drill holes in a PCB. In robotic machining or drilling applications, the "cities" are parts to machine or holes (of different sizes) to drill, and the "cost of travel" includes time for retooling the robot (single machine job sequencing problem).

The generalized travelling salesman problem deals with "states" that have (one or more) "cities" and the salesman has to visit exactly one "city" from each "state". Also known as the "travelling politician problem". One application is encountered in ordering a solution to the cutting stock problem in order to minimise knife changes. Another is concerned with drilling in semiconductor manufacturing, demonstrated that the generalised travelling salesman problem can be transformed into a standard travelling salesman problem with the same number of cities, but a modified distance matrix.
The sequential ordering problem deals with the problem of visiting a set of cities where precedence relations between the cities exist.

The travelling purchaser problem deals with a purchaser who is charged with purchasing a set of products. He can purchase these products in several cities, but at different prices and not all cities offer the same products. The objective is to find a route between a subset of the cities, which minimizes total cost (travel cost + purchasing cost) and which enables the purchase of all required products.

Computing a solution

The traditional lines of attack for the NP-hard problems are the following:

Devising algorithms for finding exact solutions (they will work reasonably fast only for relatively small problem sizes).

Devising "suboptimal" or heuristic algorithms, i.e., algorithms that deliver either seemingly or probably good solutions, but which could not be proved to be optimal.

Finding special cases for the problem ("subproblems") for which either better or exact heuristics are possible.

Computational complexity

The problem has been shown to be NP-hard (more precisely, it is complete for the complexity class FPNP; see function problem), and the decision problem version ("given the costs and a number x, decide whether there is a round-trip route cheaper than x") is NP-complete. The bottleneck travelling salesman problem is also NP-hard. The problem remains NP-hard even for the case when the cities are in the plane with Euclidean distances, as well as in a number of other restrictive cases. Removing the condition of visiting each city "only once" does not remove the NP-hardness, since it is easily seen that in the planar case there is an optimal tour that visits each city only once (otherwise, by the triangle inequality, a shortcut that skips a repeated visit would not increase the tour length).

Complexity of approximation

In the general case, finding a shortest travelling salesman tour is NPO-complete] If the distance measure is a metric and symmetric, the problem becomes APX-complete and Christofides’s algorithm approximates it within 3/2. If the distances are restricted to 1 and 2 (but still are a metric) the approximation ratio becomes 7/6. In the asymmetric, metric case, only logarithmic performance guarantees are known, the best current algorithm achieves performance ratio 0.814 log n; it is an open question if a constant factor approximation exists.
The corresponding maximization problem of finding the longest travelling salesman tour is approximable within 63/38. If the distance function is symmetric, the longest tour can be approximated within 4/3 by a deterministic algorithm and within \((33 + \varepsilon) / 25\) by a randomised algorithm.

Exact algorithms

The most direct solution would be to try all permutations (ordered combinations) and see which one is cheapest (using brute force search). The running time for this approach lies within a polynomial factor of \(O(n!)\), the factorial of the number of cities, so this solution becomes impractical even for only 20 cities. One of the earliest applications of dynamic programming is an algorithm that solves the problem in time \(O(n^2 2^n)\).

The dynamic programming solution requires exponential space. Using inclusion–exclusion, the problem can be solved in time within a polynomial factor of \(2^n\) and polynomial space.

Improving these time bounds seems to be difficult. For example, it is an open problem if there exists an exact algorithm for TSP that runs in time \(O(1.9999^n)\).

Other approaches include:

Various branch-and-bound algorithms, which can be used to process TSPs containing 40-60 cities. Progressive improvement algorithms which use techniques reminiscent of linear programming. Works well for up to 200 cities. Implementations of branch-and-bound and problem-specific cut generation; this is the method of choice for solving large instances. This approach holds the current record, solving an instance with 85,900 cities.

Some Lower Bounds on Parallel computations:

Lower bounds are proven on the parallel-time complexity of several basic functions on the most powerful concurrent-read concurrent-write PRAM with unlimited shared memory and unlimited power of individual processors (denoted by PRIORITY(m)):

1. It is proved that with a number of processors polynomial in \(n \Omega(\log n)\) time is needed for addition, multiplication or bitwise OR of \(n\) numbers, when each number has \(\Pi\) bits. Hence even the bit complexity (i.e., the time complexity as a function of the total number of bits in the input) is logarithmic in this case. This improves a beautiful result of Meyer auf der Heide and Wigderson. They proved a log \(n\) lower bound using Ramsey-type techniques. Using Ramsey
theory, it is possible to get an upper bound on the number of bits in the inputs used. However, for the case of polynomially many processors, this upper bound is more than a polynomial in n.

(2) An $\Omega(\log n)$ lower bound is given for $\text{PRIORIT}(\infty)$ with $n^{o(1)}$ processors on a function with inputs from $\{0, 1\}$, namely for the function $f(x_1, \ldots, x_n) = \sum_{i=1}^{n} x_i a^i$ where $a$ is fixed and $x_i \in (0, 1)$.

(3) Finally, by a new efficient simulation of $\text{PRIORIT}(\infty)$ by unbounded fan-in circuits, that with less than exponential number of processors, it is proven a $\text{PRIORIT}(\infty)$ cannot compute $\text{PARITY}$ in constant time, and with $n^{O(1)}$ processors $\Omega(\log n)^{1/2}$ time is needed. The simulation technique is of independent interest since it can serve as a general tool to translate circuit lower bounds into PRAM lower bounds.

Further, the lower bounds in (1) and (2) remain valid for probabilistic or nondeterministic concurrentread

---

Section D

NP Hard and NP Complete Problems

Decision Problems: Many of the problems that we have discussed involve optimization of one form or another: find the shortest path, find the minimum cost spanning tree, find the minimum weight triangulation. For rather technical reasons, most NP-complete problems that we will discuss will be phrased as decision problems. A problem is called a decision problem if its output is a simple “yes” or “no” (or you may think of this as True/False, 0/1, accept/reject).

We will phrase many optimization problems in terms of decision problems. For example, the minimum spanning tree decision problem might be: Given a weighted graph G and an integer $k$, does G have a spanning tree whose weight is at most $k$?

This may seem like a less interesting formulation of the problem. It does not ask for the weight of the minimum spanning tree, and it does not even ask for the edges of the spanning tree that achieves this weight. However, our job will be to show that certain problems cannot be solved efficiently. If we show that the simple decision problem cannot be solved efficiently, then the more general optimization problem certainly cannot be solved efficiently either.
Definition: Define $P$ to be the set of all languages for which membership can be tested in polynomial time. (Intuitively, this corresponds to the set of all decisions problems that can be solved in polynomial time.)

Note that languages are sets of strings, and $P$ is a set of languages. $P$ is defined in terms of how hard it is computationally to recognized membership in the language. A set of languages that is defined in terms of how hard it is to determine membership is called a complexity class. Since we can compute minimum spanning trees in polynomial time, we have $L \in P$.

Here is a harder one, though. $M = \{ (G; k) \mid G \text{ has a simple path of length at least } k \}$.

Given a graph $G$ and integer $k$ how would you “recognize” whether it is in the language $M$? You might try searching the graph for a simple paths, until finding one of length at least $k$. If you find one then you can accept and terminate. However, if not then you may spend a lot of time searching (especially if $k$ is large, like $n - 1$, and no such path exists). So is $M \in P$? No one knows the answer. In fact, we will show that $M$ is NP-complete. In what follows, we will be introducing a number of classes. We will jump back and forth between the terms “language” and “decision problems”, but for our purposes they mean the same things. Before giving all the technical definitions, let us say a bit about what the general classes look like at an intuitive level.

$P$: This is the set of all decision problems that can be solved in polynomial time. We will generally refer to these problems as being “easy” or “efficiently solvable”. (Although this may be an exaggeration in many cases.)

$NP$: This is the set of all decision problems that can be verified in polynomial time. (We will give a definition of this below.) This class contains $P$ as a subset. Thus, it contains a number of easy problems, but it also contains a number of problems that are believed to be very hard to solve. The term $NP$ does not mean “not polynomial”. Originally the term meant “nondeterministic polynomial time”. But it is bit more intuitive to explain the concept from the perspective of verification.

$NP$-hard: In spite of its name, to say that problem is $NP$-hard does not mean that it is hard to solve. Rather it means that if we could solve this problem in polynomial time, then we could solve all $NP$ problems in polynomial time. Note that for a problem to be $NP$ hard, it does not have to be in the class $NP$. Since it is widely believed that all $NP$ problems are not solvable in polynomial time, it is widely believed that no $NP$-hard problem is solvable in polynomial time.

$NP$-complete: A problem is $NP$-complete if (1) it is in $NP$, and (2) it is $NP$-hard. That is, $NPC = NP \cap NP$-hard. The figure below illustrates one way that the sets $P$, $NP$, $NP$-hard, and $NP$-complete ($NPC$) might look. We say might because we do not know whether all of these complexity classes are distinct or whether they are all solvable in polynomial time. There are some problems in the figure that we will not discuss. One is Graph Isomorphism, which asks whether two graphs...
are identical up to a renaming of their vertices. It is known that this problem is in NP, but it is not known to be in P. The other is QBF, which stands for Quantified Boolean Formulas. In this problem you are given a boolean formula with quantifiers (9 and 8) and you want to know whether the formula is true or false. This problem is beyond the scope of this course, but may be discussed in an advanced course on complexity theory.

![Diagram showing the (possible) structure of P, NP, and related complexity classes.](image)

**One way that things ‘might’ be.**

The (possible) structure of P, NP, and related complexity classes.

**NP decision problems**

The decision problem DL for a formal language L is the computational task:

Given an arbitrary string I 2, to determine whether or not I 2 L.

The input string I is called an instance of the problem DL. It is a positive or “yes” instance if I 2 L, otherwise it is a negative or “no” instance. Any computational decision problem can be represented as the decision problem for some formal language L.

A decision problem is called non-deterministically polynomial (NP) if there is a
relation cert 2 C such that, for each I 2,

1. I 2 L if and only if cert(c; I) for some c 2 C, and

2. for any c 2 C, it can be determined whether or not cert(c; I) in a time that is bounded by a polynomial function of the length of I.

For a positive instance I of DL, any element c 2 C such that cert(c; I) is called a certificate for I. It certifies that I is indeed a positive instance (since only positive instances have certificates), and it can be checked to be a certificate in polynomial time.

For “it can be determined” in the definition above, we may read “there exists a Turing machine which can determine”; this is justified by the Church-Turing thesis. Later we shall be more specific about the format we would like our certificate-checking machine to have.

If we had access to unlimited parallelism, we could build a machine which, given instance I 2 ,

concurrently checks all c 2 C to determine whether cert(c; I). This could be done in polynomial time.

Alternatively, we could use a non-deterministic machine which guesses an element c 2 C and checks whether or not it is a certificate for I. This also can be done in polynomial time, but is not of any practical utility, since the chance of finding a certificate for I at random is remote; but it does explain the origin of the designation NP.

Very many computational problems encountered in practice are NP. You learnt about some of them in the second year.
Stephen Cook showed in 1971 that any NP problem can be converted in polynomial time to the specific

problem SAT, the Satisfiability Problem for Propositional Calculus clauses. We shall define this problem,

explain Cook’s proof, and then discuss its implications: in particular, it leads us to the very important notion

of NP-completeness.

Cook’s Theorem

Cook’s Theorem states that Any NP problem can be converted to SAT in polynomial time. In order to prove this, we require a uniform way of representing NP problems. Remember that what makes a problem NP is the existence of a polynomial-time algorithm—more specifically, a Turing machine—for checking candidate certificates. What Cook did was somewhat analogous to what Turing did when he showed that the Entscheidungsproblem was equivalent to the Halting Problem. He showed how to encode as Propositional Calculus clauses both the relevant facts about the problem instance and the Turing machine which does the certificate-checking, in such a way that the resulting set of clauses is satisfiable if and only if the original problem instance is positive. Thus the problem of determining the latter is reduced to the problem of determining the former.

Assume, then, that we are given an NP decision problem D. By the definition of NP, there is a polynomial function P and a Turing machine M which, when given any instance I of D, together with a candidate certificate c, will check in time no greater than P(n), where n is the length of I, whether or not c is a certificate of I.

Let us assume that M has q states numbered 0, 1, 2…… q-1, and a tape alphabet a1,a2………… as. We shall assume that the operation of the machine is governed by the functions T, U, and D as described in the chapter on the Entscheidungsproblem. We shall further assume that the initial tape is inscribed with the problem instance on the squares 1,2, 3………. n, and the putative certificate on the squares –m……..-2,-1.

Square zero can be assumed to contain a designated separator symbol. We shall also assume that the machine halts scanning square 0, and that the symbol in this square at that stage will be a1 if and only if the candidate certificate is a true certificate. Note that we must have m<= P(n). This is because with a problem instance of length n the computation is completed in at most P(n) steps; during this process, the Turing machine
head cannot move more than \( P(n) \) steps to the left of its starting point. We define some atomic propositions with their intended interpretations as follows:

1. For \( i = 0, 1, \ldots, P(n) \) and \( j = 0, 1, \ldots, Q-1 \), the proposition \( Q_{ij} \) says that after \( i \) computation steps, \( M \) is in state \( j \).

2. For \( i = 0, 1, \ldots, P(n) \), \( j = -P(n), \ldots, P(n) \), and \( k = 1, 2, \ldots, s \), the proposition \( S_{ijk} \) says that after \( i \) computation steps, square \( j \) of the tape contains the symbol \( a_k \).

3. For \( i = 0, 1, \ldots, P(n) \) and \( j = -P(n), \ldots, P(n) \), the proposition \( T_{ij} \) says that after \( i \) computation steps, the machine \( M \) is scanning square \( j \) of the tape. Next, we define some clauses to describe the computation executed by \( M \):

1. At each computation step, \( M \) is in at least one state. For each \( i = 0, \ldots, P(n) \) we have the clause

   \[
   Q_{i0} \lor Q_{i1} \lor \cdots \lor Q_{i(q-1)},
   \]

   giving \( (P(n) + 1)q = O(P(n)) \) literals altogether.

2. At each computation step, \( M \) is in at most one state. For each \( i = 0, \ldots, P(n) \) and for each pair \( j, k \) of distinct states, we have the clause

   \[
   \neg (Q_{ij} \land Q_{ik}),
   \]

   giving a total of \( q(q \cdot 1)(P(n) + 1) = O(P(n)) \) literals altogether.

3. At each step, each tape square contains at least one alphabet symbol. For each \( i = 0, \ldots, P(n) \) and

   \[
   -P(n) \leq j \leq P(n) \text{ we have the clause }
   \]

   \[
   S_{ij1} \lor S_{ij2} \lor \cdots \lor S_{ijq},
   \]

   giving \( (P(n) + 1)(2P(n) + 1)s = O(P(n)^2) \) literals altogether.

4. At each step, each tape square contains at most one alphabet symbol. For each \( i = 0, \ldots, P(n) \) and

   \[
   -P(n) \leq j \leq P(n) \text{ and each distinct pair } a_k; a_l \text{ of symbols we have the clause }
   \]

   \[
   \neg (S_{ijk} \land S_{ijl}),
   \]
giving a total of \((P(n) + 1)(2P(n) + 1)s(s - 1) = O(P(n)^2)\) literals altogether.

5. At each step, the tape is scanning at least one square. For each \(i = 0 \ldots P(n)\), we have the clause

\[
T_{i(-P(n))} \lor T_{i(1-P(n))} \lor \cdots \lor T_{i(P(n)-1)} \lor T_{iP(n)},
\]

giving \((P(n) + 1)(2P(n) + 1) = O(P(n)^2)\) literals altogether.

6. At each step, the tape is scanning at most one square. For each \(i = 0 \ldots P(n)\), and each distinct pair \(j, k\) of tape squares from \(-P(n)\) to \(P(n)\), we have the clause

\[
\neg(T_{ij} \land T_{ik}),
\]

giving a total of \(2P(n)(2P(n) + 1)(P(n) + 1) = O(P(n)^3)\) literals.

7. Initially, the machine is in state 1 scanning square 1. This is expressed by the two clauses

\[
Q_{01}, T_{01},
\]

giving just two literals.

8. The configuration at each step after the first is determined from the configuration at the previous step by the functions \(T, U,\) and \(D\) defining the machine \(M\). For each \(i = 0 \ldots P(n)\), \(-P(n) \leq j \leq P(n)\), \(k = 0 \ldots q - 1\), and \(l = 1 \ldots s\), we have the clauses

\[
T_{ij} \land Q_{ik} \land S_{ijl} \rightarrow Q_{(i+1)T(k,l)}
\]

\[
T_{ij} \land Q_{ik} \land S_{ijl} \rightarrow S_{(i+1)jU(k,l)}
\]

\[
T_{ij} \land Q_{ik} \land S_{ijl} \rightarrow T_{(i+1)(j+D(k,l))}
\]

\[
S_{ijk} \rightarrow T_{ij} \lor S_{(i+1)jk}
\]

The fourth of these clauses ensures that the contents of any tape square other than the currently scanned square remains the same (to see this, note that the given clause is
equivalent to the formula \( S_{ijk} \land \neg T_{ij} \rightarrow S_{(i+1)jk} \). These clauses contribute a total of \((12s + 3)(P(n) + 1)(2P(n) + 1)q = O(P(n)^2)\) literals.

9. Initially, the string \( a_{i1}, a_{i2}, \ldots, a_{in} \) defining the problem instance \( I \) is inscribed on squares 1, 2, …, \( n \) of the tape. This is expressed by the \( n \) clauses

\[
S_{01i1}, S_{02i2}, \ldots, S_{0ni_n},
\]

a total of \( n \) literals.

10. By the \( P(n) \)th step, the machine has reached the halt state, and is then scanning square 0, which contains the symbol \( a_1 \). This is expressed by the three clauses

\[
Q, P(n)0, S_{P(n)01}, T_{P(n)0},
\]

giving another 3 literals.

Altogether the number of literals involved in these clauses is \( O(P(n)^3) \) (in working this out, note that \( q \) and \( s \) are constants, that is, they depend only on the machine and do not vary with the problem instance; thus they do not contribute to the growth of the the number of literals with increasing problem size, which is what the \( O \) notation captures for us). It is thus clear that the procedure for setting up these clauses, given the original machine \( M \) and the instance \( I \) of problem \( D \), can be accomplished in polynomial time.

We must now show that we have succeeded in converting \( D \) into SAT. Suppose first that \( I \) is a positive instance of \( D \). This means that there is a certificate \( c \) such that when \( M \) is run with inputs \( c, I \), it will halt scanning symbol \( a_1 \) on square 0. This means that there is some sequence of symbols that can be placed initially on squares -\( P(n) \)…-1 of the tape so that all the clauses above are satisfied. Hence those clauses constitute a positive instance of SAT. Conversely, suppose \( I \) is a negative instance of \( D \). In that case there is no certificate for \( I \), which means that whatever symbols are placed on squares -\( P(n) \)…-1 of the tape, when the computation halts the machine will not be scanning \( a_1 \) on square 0. This means that the set of clauses above is not satisfiable, and hence constitutes a negative instance of SAT. Thus from the instance \( I \) of problem \( D \) we have constructed, in polynomial time, a set of clauses which constitute a positive instance of SAT if and only if \( I \) is a positive instance of \( D \). In other words, we have converted \( D \) into SAT in polynomial time. And since \( D \) was an arbitrary NP problem it follows that any NP problem can be converted to SAT in polynomial time.
NP-completeness

Cook’s Theorem implies that any NP problem is at most polynomially harder than SAT. This means that if we find a way of solving SAT in polynomial time, we will then be in a position to solve any NP problem in polynomial time. This would have huge practical repercussions, since many frequently encountered problems which are so far believed to be intractable are NP. This special property of SAT is called NP-completeness. A decision problem is NP-complete if it has the property that any NP problem can be converted into it in polynomial time. SAT was the first NP-complete problem to be recognised as such (the theory of NP-completeness having come into existence with the proof of Cook’s Theorem), but it is by no means the only one. There are now literally thousands of problems, cropping up in many different areas of computing, which have been proved to be NP-complete. In order to prove that an NP problem is NP-complete, all that is needed is to show that SAT can be converted into it in polynomial time. The reason for this is that the sequential composition of two polynomial-time algorithms is itself a polynomial-time algorithm, since the sum of two polynomials is itself a polynomial. Suppose SAT can be converted to problem D in polynomial time. Now take any NP problem D0. We know we can convert it into SAT in polynomial time, and we know we can convert SAT into D in polynomial time. The result of these two conversions is a polynomial-time conversion of D0 into D. Since D0 was an arbitrary NP problem, it follows that D is NP-complete. We illustrate this by showing that the problem 3SAT is NP-complete. This problem is similar to SAT, but restricts the clauses to at most three schematic letters each: Given a finite set \{C1,C2,…,Cn\} of clauses, each of which contains at most three schematic letters, determine whether there is an assignment of truth-values to the schematic letters appearing in the clauses which makes all the clauses true.

3SAT is obviously NP (since it is a special case of SAT, which is NP). It turns out to be straightforward to convert an arbitrary instance of SAT into an instance of 3SAT with the same satisfiability property. Take any clause written in disjunctive form as

\[ C \equiv L_1 \lor L_2 \lor \ldots \lor L_n; \]

where \( n > 3 \) and each \( L_i \) is a literal. We replace this by \( n - 2 \) new clauses, using \( n - 3 \) new schematic letters \( X_1 , \ldots , X_{n-3} \), as follows:
\[ L_1 \lor L_2 \lor X_1 \\
X_1 \rightarrow L_3 \lor X_2 \\
X_2 \rightarrow L_4 \lor X_3 \\
\vdots \\
X_{n-4} \rightarrow L_{n-2} \lor X_{n-3} \\
X_{n-3} \rightarrow L_{n-1} \lor L_n \]

Call the new set of clauses \( C \). Any truth-assignment to the schematic letters appearing in the \( Li \) which satisfies \( C \) can be extended to the \( Xi \) so that \( C \) is satisfied, and conversely any truth-assignment which satisfies \( C \) also satisfies \( C \). To prove this, suppose that a certain truth-assignment satisfies \( C \). Then it satisfies at least one of the literals appearing in \( C \), say \( L_k \). Now assign true to \( X_1, X_2, \ldots, X_{k-2} \) and false to \( X_{k-1}, \ldots, X_{n-3} \). Then all the clauses in \( C \) are satisfied: for \( i = 1, 2, \ldots, k-2 \), the \( i \)th clause is satisfied because \( X_i \) is true; the \((k-1)\)th clause is satisfied because \( L_k \) is true; and for \( j = k, k+1, \ldots, n-2 \), the \( j \)th clause is satisfied because \( X_{j-1} \) (appearing in the antecedent) is false. Conversely, suppose we have a truth-assignment satisfying \( C \): each clause in \( C \) is satisfied. Suppose \( L_1, \ldots, L_{n-2} \) are all false. Then it is easy to see that all the \( Xi \) must be true, in particular \( X_{n-3} \) is true, so either \( L_{n-1} \) or \( L_n \) is true. Thus in any event at least one of the \( Li \) is true, and hence \( C \) is true. If we take an instance of SAT and replace all the clauses containing more than three literals by clauses containing exactly three in the way described above, we end up with an instance of 3SAT which is satisfiable if and only if the original instance is satisfiable. Moreover, the conversion can be accomplished in polynomial time. It follows that 3SAT, like SAT, is NP-complete.