ECE365: Data Structures and Algorithms II (DSA 2)

Algorithm Strategies

Algorithm Strategies

- In our two DSA courses, we have discussed algorithms to solve various problems and analyzed their running times
- We have not yet discussed how to solve problems ourselves (i.e., how to design algorithms to compute solutions to problems)
- To a large extent, this generally involves some creativity
- Also keep in mind that some problems that sound simple may not have any efficient solutions (or may not have solutions at all)
- In this topic, we will discuss some general *algorithm strategies* that may help you to determine methods for solving a given problem
- We will also discuss at least one new example of each type of algorithm
- This topic has been largely based on Chapter 10 of the textbook, titled "Algorithm Design Techniques"
- We will cover some of the subtopics in more detail than the textbook

Greedy Algorithms

- The idea behind a **greedy algorithm** is to make a sequence of choices such that each choice seems to be the best choice possible at the time that you make it
- In general, you are making *locally optimal* choices, perhaps with the hope of finding a *globally optimal* solution
- Sometimes, you can then prove that the final solution will be the best solution possible; other times, this will not be the case at all
- If you need an algorithm that is guaranteed to find an optimal solution, then you need to prove that the algorithm is optimal; in some cases, an inductive proof can be used
- Even when greedy algorithms do not find the optimal solution (e.g., for the traveling salesman problem), they might be guaranteed or likely to find a good solution close to the optimal solution
- When you use such a greedy algorithm, you are using it as a *heuristic*
- We have already seen examples of greedy algorithms when discussing graph algorithms
- These include *Dijkstra's algorithm* to solve the *single-source shortest-path problem* and *Prim's algorithm* for determining a *minimum spanning tree*

The Coin-changing Problem

- We will start with a simple example of a greedy algorithm that we might apply to the *coin-changing problem*
- The problem asks you to make change for a specified amount of money using the smallest number of coins possible
- Some versions of the problem accept the values of available coins as part of the input
- For example, if an amount is specified, and we are using U.S. currency, you can use as many quarters as possible, then dimes, then nickels, and then pennies
- We will not prove it, but this greedy algorithm will lead to the best possible solution when applied to U.S. currency
- However, if the types of coins available are 11-cent coins, 5-cent coins, and 1-cent coins, this solution would not lead to an optimal solution
- To see this, consider making change for 15 cents

Simple Scheduling Problem

- A simple example of a *scheduling problem* involves a set of jobs, j_1, j_2, … j_N, all with known running times t_1, t_2, …, t_N
- To start, assume that there is a single processor available
- We are assuming non-preemptive scheduling, meaning that once a job stars, it must run until completion
- We want to schedule the jobs in order to achieve the minimum average completion time
- A solution that is optimal is to sort the jobs in terms of their running time (from shortest to longest) and schedule them in that order
- Even if there are multiple processors, you can cycle through the processors, assigning jobs in sorted order
- This is also guaranteed to be optimal (we won't prove it), although this is not necessarily the only optimal solution
- However, a similar-sounding problem that asks you to minimize the final completion time when multiple processors are involved is NP-complete!

Huffman Codes

- The next greedy algorithm we will discuss is a very popular compression algorithm involving **Huffman code**
- The algorithm is known as *Huffman coding* or *Huffman encoding*
- Consider a file that contains C distinct types of characters
- Then a system that uses the same number of bits per character requires at least bits per character
- ASCII uses 8 bits per character, although only 7 would really be necessary for standard ASCII
- Standard ASCII encodes 128 distinct characters, of which approximately 100 are printable
- According to our textbook, Huffman codes typically save about 25% of space for typically large files, and up to 50% or 60% on many large data files
- According to another source (by Cormen, Leiserson, Rivest, and Stein), anywhere from 20% to 90% savings is typical

Huffman Coding Example

- We will examine an example from the book that assumes the existence of only 7 distinct characters
- Specifically, the distinct characters are: 'a', 'e', 'i', 's', 't', space, and newline
- If the same number of bits were used for each character, 3 bits per character would be required
- We will assume that the frequency of each character in the file has already been computed
- In our example, we will posit that there are 10 a’s 15 e’s, 12 i’s, 3 s’s, 4 t’s, 13 spaces, and 1 newline
- The figure on the next slide shows that for this case, a standard encoding would require 174 bits

Standard Encoding Example

Character	Code	Frequency	Total Bits
a	000	10	30
e	001	15	45
i	010	12	36
s	011	3	9
t	100	4	12
space	101	13	39
newline	110	1	3
Total			174

Figure 10.8 Using a standard coding scheme

Tries

- We can represent such encodings with a special type of binary tree known as a **trie**
- In a trie, left branches represent the bit 0, right branches represent the bit 1, and characters are indicated by the leaves
- Some sources allow the branches of a trie to represent more general characters, or allow internal nodes to represent characters, but that will not be necessary for Huffman coding
- The standard encoding we have seen can be represented by the following trie:

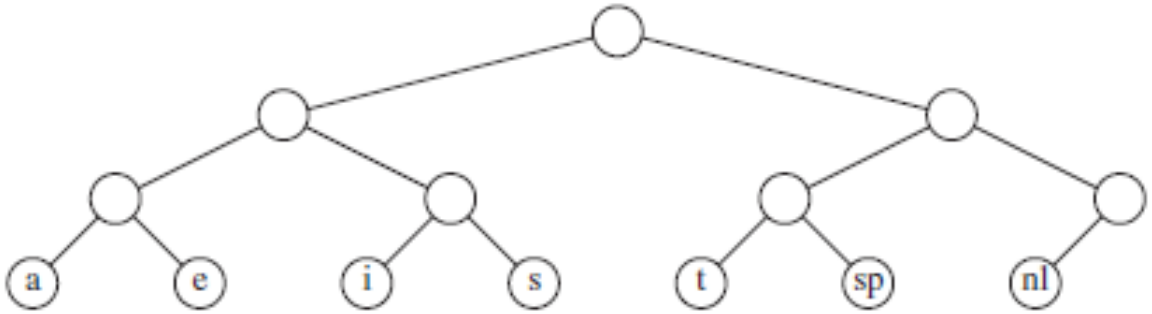


Figure 10.9 Representation of the original code in a tree

-

Slightly Better Trie

- This is clearly not the most efficient encoding
- Note that the only character with a code that starts with 11 is the newline, so the following zero does not provide any useful information
- An optimal trie for encoding will always be a **full binary tree** (i.e., all nodes will either be leaves or have two children)
- A slightly better, but still far-from-optimal, trie is as follows:

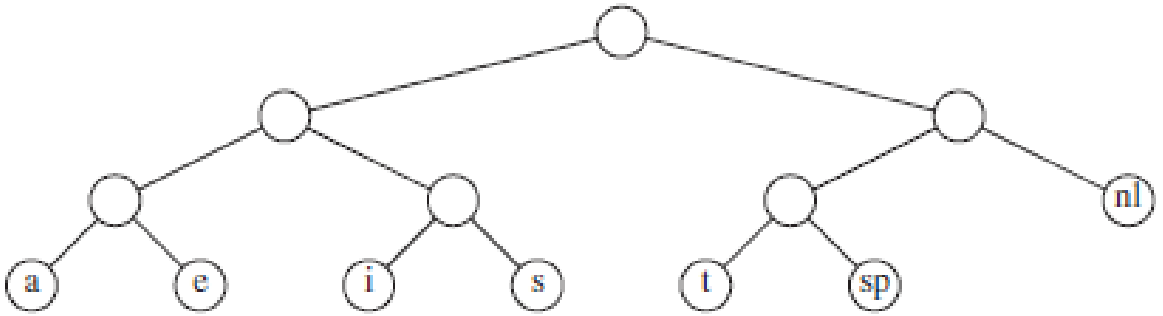


Figure 10.10 A slightly better tree

-

Prefix codes

- Any encoding in which no character code is a prefix of another character code is known as a **prefix code**
- Prefix codes also include that files encoded using them will be unambiguous
- When a file has been encoded using a prefix code, we can easily decode it even though character codes may have a different lengths
- Forcing a trie to indicate each distinct character as a leaf ensures that the trie represents a prefix code
- The goal of Huffman coding is to find the *optimal trie*, i.e., the *optimal prefix code*
- The principal behind this is to use shorter codes for frequent characters and longer codes for rare characters
- The optimal trie for the example we have been dealing with is shown on the next slide
- The slide after that shows that this encoding would require 146 bits to encode the file

Optimal Trie Example

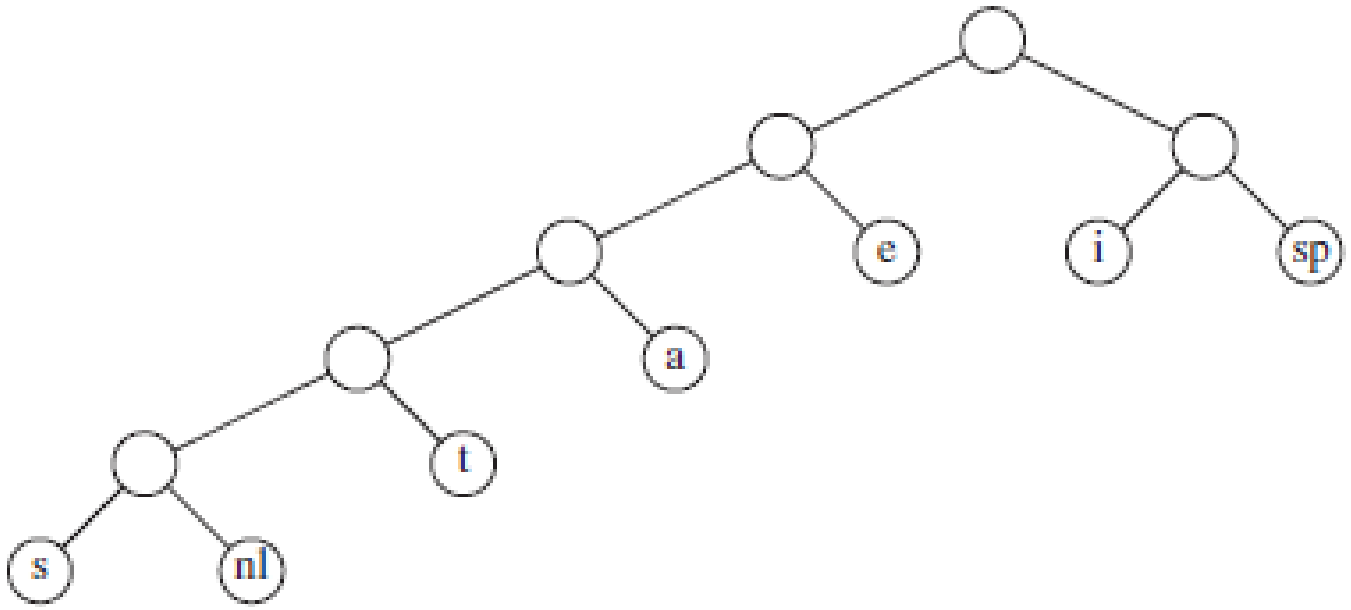


Figure 10.11 Optimal prefix code

Optimal Prefix Code Example

Character	Code	Frequency	Total Bits
a	001	10	30
e	01	15	30
i	10	12	24
s	00000	3	15
t	0001	4	16
space	11	13	26
newline	00001	1	5
Total			146

Figure 10.12 Optimal prefix code

Analyzing the Optimal Trie and Prefix Code

- There are many prefix codes that tie for optimal
- For example, swapping leaves at the same level of the trie or swapping left and right links of internal nodes lead to equally efficient encodings
- In general, if each character c_i occurs f_i times and is at depth d_i in the trie, then the cost of the using code to encode the file is equal to $\sum_i d_i * f_i$
- For this example, a standard encoding uses 174 bits, while the optimal encoding uses 146 bits
- This is only a savings of 16.1%, but that is because we were only dealing with a small file that contains only 7 distinct characters
- The normal ASCII character set has about 100 printable characters, but many occur very rarely, and some occur much more frequently than others

Huffman’s Algorithm

- A greedy algorithm for finding an optimal trie was created by David Huffman in 1952 while he was a graduate student at M.I.T.
- Let's say that the number of distinct characters in a text file to be compressed is C
- Huffman's algorithm maintains a *forest of tries*; the weight of each trie is equal to the sum of the frequencies of its leaves
- At the start, there are C single-node tries representing each of the characters
- Through each of C-1 iterations, the two tries, T1 and T2, of smallest weight are selected (breaking ties arbitrarily)
- A new trie is formed with a new root and T1 and T2 as subtrees
- At the end of the algorithm, there is one trie remaining, and this is an optimal trie representing an optimal prefix encoding
- This algorithm can be implemented efficiently using a *priority queue* (e.g., a *binary heap*)
- The next slide shows my pseudo-code for the algorithm

Huffman Algorithm Pseudo-code

```
Huffman ( Set-of-characters C )
  initialize Q as empty priority queue
  for each character c in C
    create a single-node trie T
    T.data ← c
    T.frequency ← c.frequency
    Insert(Q, T)
  loop |C| - 1 times
    T1 ← DeleteMin(Q)
    T2 ← DeleteMin(Q)
    create a new trie T
    T.left ← T1
    T.right ← T2
    T.frequency ← T1.frequency + T2.frequency
    Insert(Q, T)
  return DeleteMin(Q)
```

Huffman Algorithm Example (first three steps)

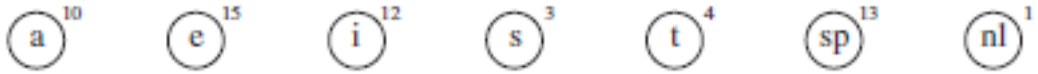


Figure 10.13 Initial stage of Huffman's algorithm

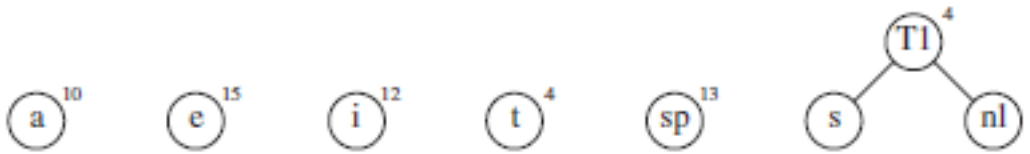


Figure 10.14 Huffman's algorithm after the first merge

Figure 10.15 Huffman's algorithm after the second merge

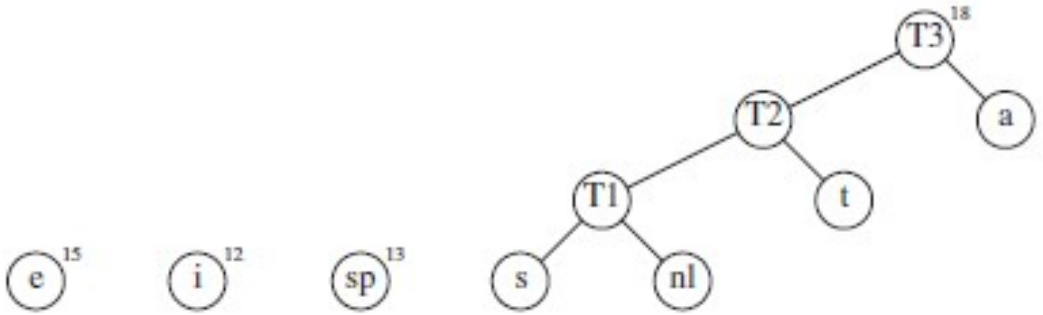


Figure 10.16 Huffman's algorithm after the third merge

Huffman Algorithm Example (last three steps)

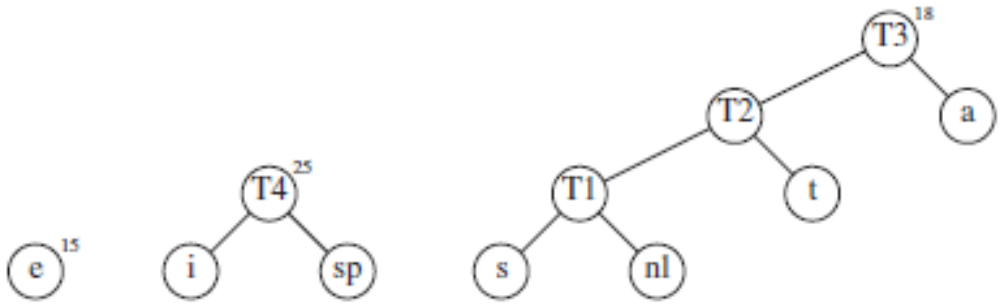


Figure 10.17 Huffman's algorithm after the fourth merge

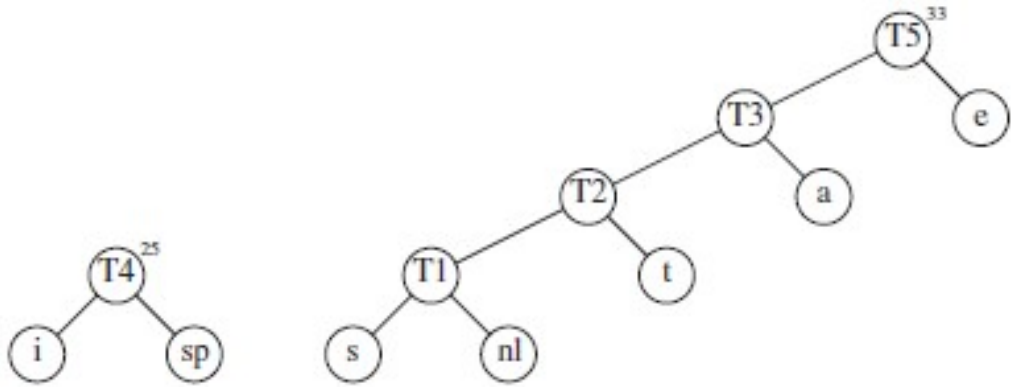


Figure 10.18 Huffman's algorithm after the fifth merge

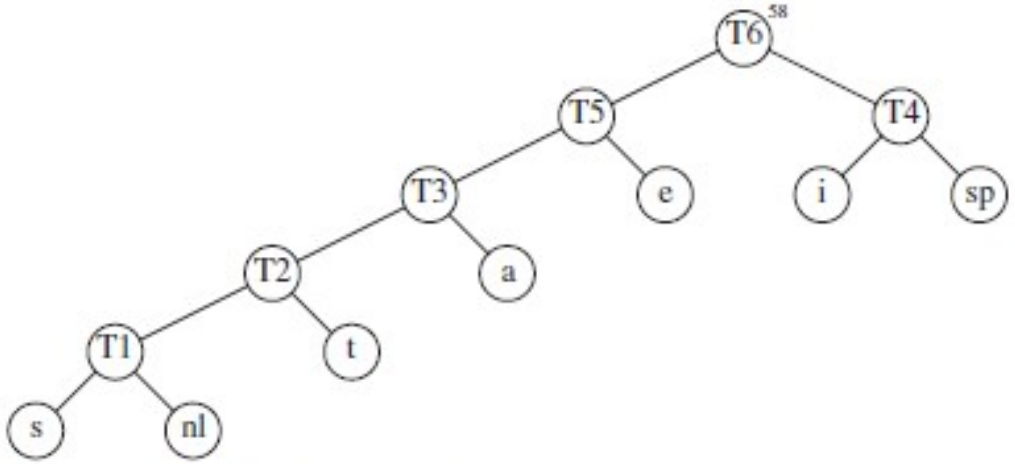


Figure 10.19 Huffman's algorithm after the final merge

Analyzing the Huffman Algorithm

- If C is the number of distinct characters, there will be C-1 iterations
- Each iteration require O(log C) time (using binary heaps), so the running time of this algorithm is O(C log C)
- This assumes that the frequency of each character is known
- Otherwise, an additional loop is necessary at the start of the routine that takes linear time with respect to the size of the file
- Although this does not change the complexity, this linear loop will probably take longer than the rest of the algorithm
- This is because C is generally going to be small, and we probably won't bother compressing a file if it isn't large
- Even if we use a simpler, quadratic implementation, $O(C^2)$, for the main Huffman routine, it will be fast

Why Does This Work?

- We will informally discuss why this algorithm is guaranteed to produce the optimal solution
- Recall that the optimal trie must be a full trie
- Therefore, the deepest leaf must have a sibling (another leaf); clearly, these nodes should represent the two least frequent characters
- The first pass of the algorithm therefore produces a valid part of the trie
- You can then consider the merged tree to represent a *meta-character* with a frequency equal to the sum of the characters represented by its leaves
- Therefore, the later iterations of the algorithm are correct for the same reason as the first; it is always valid to combine the two least-frequent metacharacters
- Whenever we merge two trees, we are adding a bit to the representation of each leaf of both trees
- Each greedy step adds the least number of bits possible to the final trie

Using Huffman Codes

- Once the trie representing the optimal encoding is produced, it can be used to compress the file
- A representation of the encoding must be included with the compressed file (either internal to the file or along with it)
  - Otherwise, the decoding of the file will not be possible
  - This representation could take the form of a chart that maps binary sequences to characters
  - Alternatively, it could take the form of a data structure representing the trie
- In some other contexts, the prefix code is determined based on a large dataset once, and then applied to future files
- This is not optimal for each individual file, but it allows a single shared compression scheme to be applied to many files

Divide and Conquer Algorithms

- The next strategy we will discuss is called **divide-and-conquer**
- This strategy is applicable when a problem can be broken down into parts that can be solved with a similar algorithm to the original problem
- There must be some way to combine the solutions to the parts to form a final solution
- The parts (smaller problems) are often solved with *recursion* (although recursion is never necessary to implement a solution)
- There also must be a base case for which a recursive call does not apply
- We have already seen examples of divide-and-conquer algorithms when discussing sorting in Data Structures and Algorithms I
- These include *mergesort*, *quicksort*, and *quickselect*
- We have seen that the *Master theorem* can sometimes be used to determine the running time complexity of such an algorithm

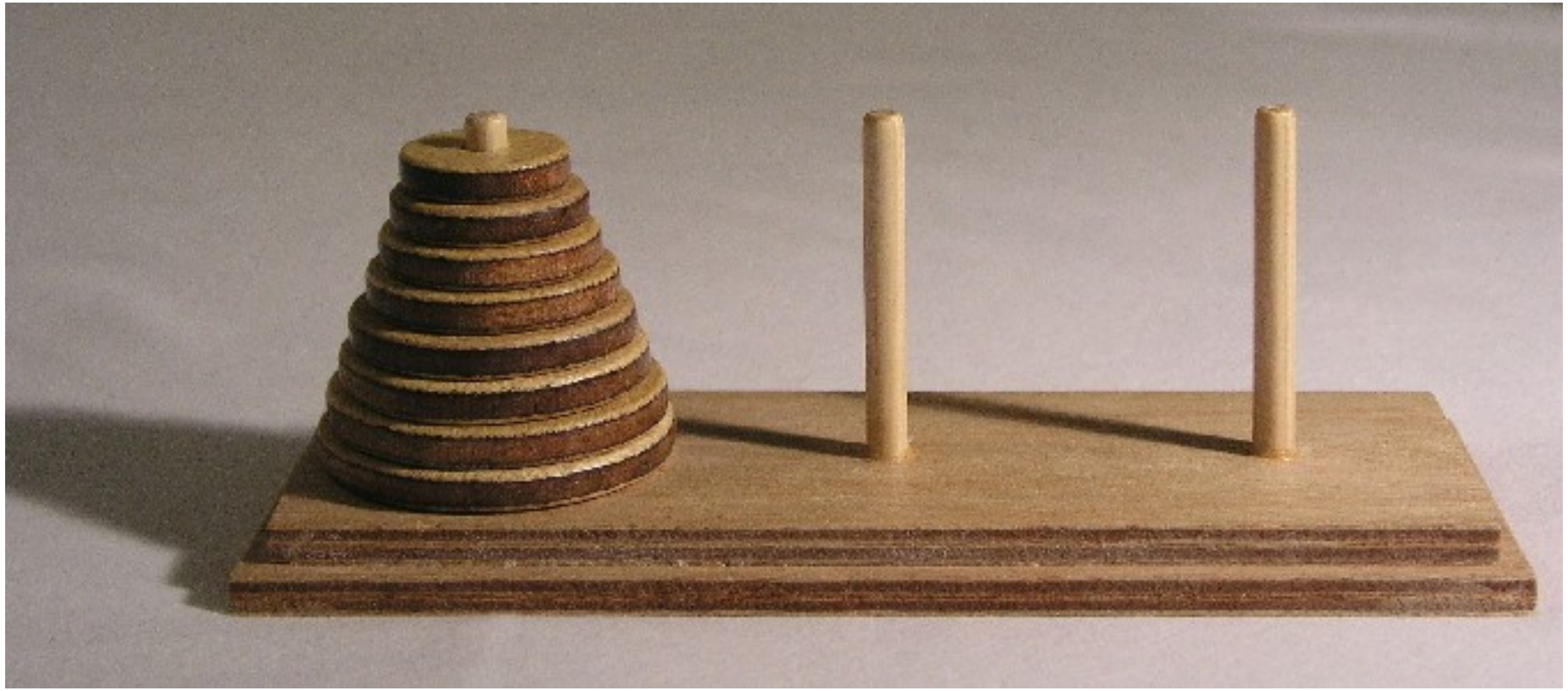
Examples Discussed in Textbook

- The textbook discusses the following divide-and-conquer algorithms:
  - An algorithm for finding the closest pair of points out of a set of N points in O(N log N) time
  - A worst-case linear algorithm for selection involving median-of-median-of-five pivot selection for quickselect
  - An algorithm for multiplying two N-digit numbers in $O(N^{1.59})$ time
  - An algorithm for matrix multiplication that runs in $O(N^{2.81})$ time
- We will go over a different problem that I consider fun, although not of practical significance (and the solution is exponential)

Tower of Hanoi

- The Tower of Hanoi is a puzzle including three sticks and disks that can be placed on them
- At the start, all disks are on one of the stick, arrange from largest (at the bottom) to the smallest (at the top)
- The goal is to move the disks from the starting stick to a designated destination stick, subject to the following rules:
  - Only one disk can be moved at a time
  - You can only lift the top disk off a stick; this disk may then be placed on another stick, as long as it does not violate the next constraint
  - You can never place a larger disk on top of a smaller disk

Tower of Hanoi Image (from Wikipedia)



Solving the Tower of Hanoi

- Assume we have a Tower of Hanoi puzzle with N disks
- At some point, we need to move the largest disk from the source stick to the destination stick
- Before that, we must move the other N-1 disks from the source stick to the auxiliary stick
- The largest disk will not affect any of the moves required to achieve that intermediate goal
- After the largest disk is moved to the destination stick, when then must move the other N-1 disks from the auxiliary stick to the destination stick
- In summary, we can move N disks from the source stick to the destination stick with the following steps:
    1. Move N-1 disks from the source stick to the auxiliary stick (this is a recursive step)
    2. Move the largest disk from the source stick to the destination stick
    3. Move N-1 disks from the auxiliary stick to the destination stick (this is a recursive step)
- Note that all three of the steps listed in this solution are necessary

Code for Solving the Tower of Hanoi

```
void Tower(int n, string src, string dst, string aux)
{
  if (n == 1)
    cout << "Move from " << src << " to " << dst << "\n";
  else {
    Tower(n-1, src, aux, dst);
    cout << "Move from " << src << " to " << dst << "\n";
    Tower(n-1, aux, dst, src);
  }

  return;
}
```

Suppose the disks are labeled "A", "B", and "C", and there are 8 disks that must be moved from "A" to "C"; then we could call the above function as follows:

```
Tower(8, "A", "C", "B");
```

Analyzing the Solution

- Suppose T(N) is the number of moves used to solve the puzzle with our solution when there are N disks
- The equation describing the number of steps used to solve the problem can be described by the following recurrence relation:

  $T(N) = 2 * T(N - 1) + O(1)$

- On the surface, this might look similar to the mergesort recurrence
- However, note that each recursive call here is applied to N-1, not to N/2
- This makes a huge difference; unlike other divide-and-conquer solutions we have seen, this is an exponential solution
- We could clearly see that, as each value of T(N) more than doubles as N increases by 1
- This does not mean it is a bad solution, as far as solutions to this problem go
- An exponential number of moves are required to solve this problem, and it can be shown that this solution uses the fewest moves possible

Dynamic Programming Algorithms

- The next strategy we will discuss is known as **dynamic programming**
- Dynamic programming is related to divide-and-conquer and recursive algorithms
- When dealing with straight-forward implementations of recursive algorithms, algorithms can become much less efficient than necessary if the subproblems are not independent
- We'll start by discussing implementations for computing the k^{th} value in the Fibonacci sequence (1, 1, 2, 3, 5, 8, 13, 21, 34, 55, etc.), which we discussed early in DSA 1
- Really, we are computing the right-most 32-bits of values, since we are not worrying about overflow
- We learned that a typical recursive solution is an exponential time solution
- We also saw that a standard iterative solution is linear, and it only needs to keep track of the two previous values in the Fibonacci sequence (and the current value)
- However, consider a more general recursive mathematical function for which the current value depends not only on the previous two values, but perhaps on all previous values
- This can still be solved iteratively by storing all the values computed so far in an array
- We can also take this approach for Fibonacci, even though it uses more memory than necessary

Fibonacci Implementations (from DSA1)

Recursive Solution (exponential!)

```
int Fibonacci(int n)
{
  if (n <= 2)
    return 1;
  else
    return (Fibonacci(n-1) +
            Fibonacci(n-2));
}
```

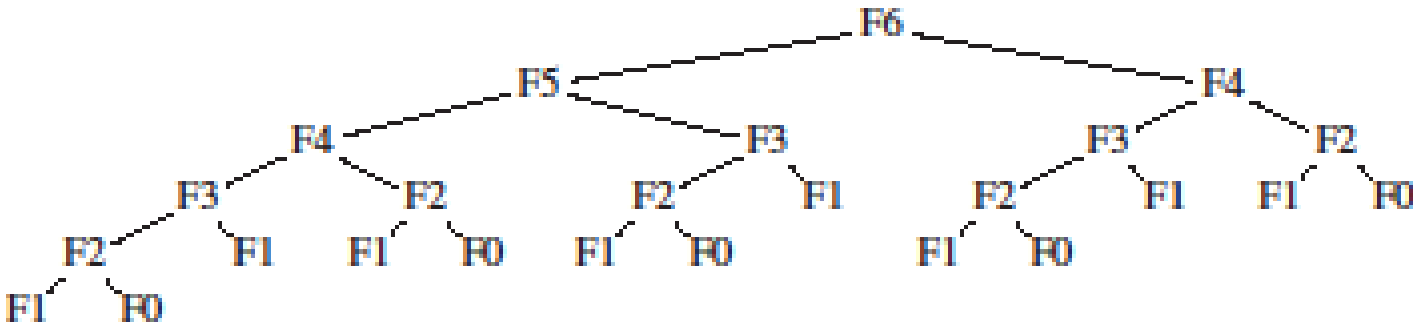
Iterative Solution (linear)

```
int Fibonacci(int n)
{
  int prev1 = 1, prev2 = 1;
  int x, cur;

  if (n <= 2)
    return 1;
  else
  {
    for (x = 3; x <= n; x++,
         prev2 = prev1, prev1 = cur)
      cur = prev2 + prev1;
    return cur;
  }
}
```

Recursive Fibonacci is Exponential

- One way to show that the recursive implementation is an exponential time solution is to examine a tree representing the recursive calls; for example:



- Another way involves analyzing the number of computational steps the routine requires and comparing this to the formula defining the Fibonacci sequence:
  - T(N) = T(N - 1) + T(N - 2) + c
  - Fib(N) = Fib(N - 1) + Fib(N - 2)
- Conclusion: The simplest or most obvious way to compute something is not always the most efficient!

Bottom-Up Dynamic Programming

- To the right, we see pseudo-code for an iterative Fibonacci implementation
- This version uses linear memory, which is more than necessary
- However, there are a couple of advantages:
  - This version is more readable than the previous version
  - This version is easily expandable to functions that rely on many previous values
- This is an example of **bottom-up dynamic programming**, meaning that:
  - We are storing the computed values in an array (or more general matrix)
  - Each newly computed value depends on other values that have already been computed

```
Fibonacci( integer N )
  if N ≤ 2
    return 1

  Fib[1] ← 1
  Fib[2] ← 1
  Loop i from 3 to N
    Fib[i] ← Fib[i-1] +
Fib[i-2]

  return Fib[n]
```

Top-Down Dynamic Programming

- To the right, we see pseudo-code for a recursive Fibonacci implementation
- Although recursive, this function executes in linear time, just like the iterative solution
- Any call asking for a value that has already been computed will execute in quick constant time
- The array "known" could be a global array, a static variable, or a parameter passed by reference
- The array must be large enough to hold N values, all initialized to all "unknown"
- This is an example of **top-down dynamic programming**, meaning that:
  - We store the computed values in an array (or more general matrix) after they are computed via recursive calls
  - The start of each recursive routine checks to see if the desired value has already been computed
  - If so, it just looks retrieves and returns the value
  - The use of the array or matrix is called **memoization**

```
Fibonacci( integer N )
  if known[N] != unknown
    return known[N]

  if N ≤ 2
    v ← 1
  else
    v ← Fibonacci(N-1)+Fibonacci(N-
2)

  known[N] ← v
  return v
```

The Longest Common Subsequence Problem

- We will use dynamic program to solve the **Longest Common Subsequence Problem** (LCSP); note that this problem is not discussed in the textbook
- This involves a different definition of a subsequence than the Maximum Subsequence Sum Problem that we discussed near the start of Data Structures and Algorithms I
- Consider a string: $X = x_1x_2x_3...x_n$
- A subsequence of X is any string $Z = z_1z_2z_3...z_k$ for which there exists a strictly increasing sequence $i_1, i_2, ..., i_k$ such that for all j=1, 2, ..., k, it is true that $z_j =$
- Now consider two strings: X as defined above and also $Y = y_1y_2y_3...y_m$
- The Longest Common Subsequence Problem asks us to determine the longest string, Z, that is a subsequence of both X and Y
- One possible way to do this would be to exhaustively examine every subsequence of X and check if it is also a subsequence of Y
- However, there are 2^n possible subsequences of X, and checking if each is a subsequence of Y would take time linear to m, so this would be an $\theta(2^n * m)$ solution
- Instead, we are going to use an efficient dynamic programming solution

Solving the LCSP

- To solve the problem, we are going to use a matrix M with n+1 rows (numbered 0 through n) and m+1 columns (numbered 0 through m)
- M[i, j] represents the length of the longest common subsequence of the strings $x_1x_2x_3...x_i$ and $y_1y_2y_3...y_j$
- We start by filling in the first column and first row with zeros; this is because the length of the longest common subsequence is 0 if one of the two strings is empty
- We are going to fill in the rest of the matrix one row at a time, from top to bottom, filling each row from left to right
  - Let's say we want to fill in a particular slot M[i, j] and we have already filled in all the slots above this slot and all the slots to the left of this slot in the current row
  - We can then say that if $x_i == y_j$, M[i, j] must equal M[i-1, j-1] +1
  - Otherwise, M[i, j] will be the maximum of M[i-1, j] and M[i, j-1]
- At the end of the algorithm, the length of the longest common subsequence is stored at the bottom right of the matrix, in M[n, m]
- Note that this is clearly a O(n * m) time algorithm
- This is a bottom-up approach; a top-down approach is also possible

LCSP Pseudo-code

```
LongestCommonSubsequence ( String X, String Y )
  n ← length(X)
  m ← length(Y)
  loop i from 0 to m
    M[0,i] ← 0
  Loop i from 0 to n
    M[i,0] ← 0
  Loop i from 1 to n
    Loop j from 1 to m
      if X_i == Y_j
        M[i, j] ← M[i-1, j-1] + 1
      else if M[i-1, j] ≥ M[i, j-1]
        M[i, j] ← M[i-1, j]
      else
        M[i, j] ← M[i, j-1]
```

LCSP Example

- Consider finding the longest common subsequence of X = "wired" and Y = "world"
- The matrix would then be 6 x 6, initialized as follows:

			w	o	r	l	d
		0	1	2	3	4	5
	0	0	0	0	0	0	0
w	1	0					
i	2	0					
r	3	0					
e	4	0					
d	5	0					

LCSP Example (continued)

- We not loop through the rows one at a time from left to right one at a time, starting at position (1, 1)
- Since the two ‘w’s match, this entry becomes M[0, 0] + 1 = 1

			w	o	r	l	d
		0	1	2	3	4	5
	0	0	0	0	0	0	0
w	1	0	1				
i	2	0					
r	3	0					
e	4	0					
d	5	0					

LCSP Example (continued)

- Next, we fill in position (1, 2)
- The ‘w’ and ‘o’ do not match, so this is max(M[0, 2], M[1, 1]) = 1

			w	o	r	l	d
		0	1	2	3	4	5
	0	0	0	0	0	0	0
w	1	0	1	1			
i	2	0					
r	3	0					
e	4	0					
d	5	0					

LCSP Example (continued)

- We continue filling in row 1

			w	o	r	l	d
		0	1	2	3	4	5
	0	0	0	0	0	0	0
w	1	0	1	1	1		
i	2	0					
r	3	0					
e	4	0					
d	5	0					

LCSP Example (continued)

- We continue filling in row 1

			w	o	r	l	d
		0	1	2	3	4	5
	0	0	0	0	0	0	0
w	1	0	1	1	1	1	
i	2	0					
r	3	0					
e	4	0					
d	5	0					

LCSP Example (continued)

- We continue filling in row 1

			w	o	r	l	d
		0	1	2	3	4	5
	0	0	0	0	0	0	0
w	1	0	1	1	1	1	1
i	2	0					
r	3	0					
e	4	0					
d	5	0					

LCSP Example (continued)

- Now, we start the next row at position (2, 1)
- Since the ‘i’ and ‘w’ do not match, we again take the maximum of the value to the left and the value above

			w	o	r	l	d
		0	1	2	3	4	5
	0	0	0	0	0	0	0
w	1	0	1	1	1	1	1
i	2	0	1				
r	3	0					
e	4	0					
d	5	0					

LCSP Example (continued)

- We continue filling in the row 2

			w	o	r	l	d
		0	1	2	3	4	5
	0	0	0	0	0	0	0
w	1	0	1	1	1	1	1
i	2	0	1	1			
r	3	0					
e	4	0					
d	5	0					

LCSP Example (continued)

- We continue filling in the row 2

			w	o	r	l	d
		0	1	2	3	4	5
	0	0	0	0	0	0	0
w	1	0	1	1	1	1	1
i	2	0	1	1	1		
r	3	0					
e	4	0					
d	5	0					

LCSP Example (continued)

- We continue filling in the row 2

			w	o	r	l	d
		0	1	2	3	4	5
	0	0	0	0	0	0	0
w	1	0	1	1	1	1	1
i	2	0	1	1	1	1	
r	3	0					
e	4	0					
d	5	0					

LCSP Example (continued)

- We continue filling in the row 2

			w	o	r	l	d
		0	1	2	3	4	5
	0	0	0	0	0	0	0
w	1	0	1	1	1	1	1
i	2	0	1	1	1	1	1
r	3	0					
e	4	0					
d	5	0					

LCSP Example (continued)

- Now we start filling in row 3 in a similar fashion

			w	o	r	l	d
		0	1	2	3	4	5
	0	0	0	0	0	0	0
w	1	0	1	1	1	1	1
i	2	0	1	1	1	1	1
r	3	0	1				
e	4	0					
d	5	0					

LCSP Example (continued)

- Now we start filling in row 3 in a similar fashion

			w	o	r	l	d
		0	1	2	3	4	5
	0	0	0	0	0	0	0
w	1	0	1	1	1	1	1
i	2	0	1	1	1	1	1
r	3	0	1	1			
e	4	0					
d	5	0					

LCSP Example (continued)

- When we get to cell (3, 3), the ‘r’s match, so this cell takes the value of M[2, 2] + 1 = 2

			w	o	r	l	d
		0	1	2	3	4	5
	0	0	0	0	0	0	0
w	1	0	1	1	1	1	1
i	2	0	1	1	1	1	1
r	3	0	1	1	2		
e	4	0					
d	5	0					

LCSP Example (continued)

- We continue filling in row 3

			w	o	r	l	d
		0	1	2	3	4	5
	0	0	0	0	0	0	0
w	1	0	1	1	1	1	1
i	2	0	1	1	1	1	1
r	3	0	1	1	2	2	
e	4	0					
d	5	0					

LCSP Example (continued)

- We continue filling in row 3

			w	o	r	l	d
		0	1	2	3	4	5
	0	0	0	0	0	0	0
w	1	0	1	1	1	1	1
i	2	0	1	1	1	1	1
r	3	0	1	1	2	2	2
e	4	0					
d	5	0					

LCSP Example (continued)

- Now we fill in row 4

			w	o	r	l	d
		0	1	2	3	4	5
	0	0	0	0	0	0	0
w	1	0	1	1	1	1	1
i	2	0	1	1	1	1	1
r	3	0	1	1	2	2	2
e	4	0	1				
d	5	0					

LCSP Example (continued)

- Now we fill in row 4

			w	o	r	l	d
		0	1	2	3	4	5
	0	0	0	0	0	0	0
w	1	0	1	1	1	1	1
i	2	0	1	1	1	1	1
r	3	0	1	1	2	2	2
e	4	0	1	1			
d	5	0					

LCSP Example (continued)

- Now we fill in row 4

			w	o	r	l	d
		0	1	2	3	4	5
	0	0	0	0	0	0	0
w	1	0	1	1	1	1	1
i	2	0	1	1	1	1	1
r	3	0	1	1	2	2	2
e	4	0	1	1	2		
d	5	0					

LCSP Example (continued)

- Now we fill in row 4

			w	o	r	l	d
		0	1	2	3	4	5
	0	0	0	0	0	0	0
w	1	0	1	1	1	1	1
i	2	0	1	1	1	1	1
r	3	0	1	1	2	2	2
e	4	0	1	1	2	2	
d	5	0					

LCSP Example (continued)

- Now we fill in row 4

			w	o	r	l	d
		0	1	2	3	4	5
	0	0	0	0	0	0	0
w	1	0	1	1	1	1	1
i	2	0	1	1	1	1	1
r	3	0	1	1	2	2	2
e	4	0	1	1	2	2	2
d	5	0					

LCSP Example (continued)

- Now we fill in row 5

			w	o	r	l	d
		0	1	2	3	4	5
	0	0	0	0	0	0	0
w	1	0	1	1	1	1	1
i	2	0	1	1	1	1	1
r	3	0	1	1	2	2	2
e	4	0	1	1	2	2	2
d	5	0	1				

LCSP Example (continued)

- Now we fill in row 5

			w	o	r	l	d
		0	1	2	3	4	5
	0	0	0	0	0	0	0
w	1	0	1	1	1	1	1
i	2	0	1	1	1	1	1
r	3	0	1	1	2	2	2
e	4	0	1	1	2	2	2
d	5	0	1	1			

LCSP Example (continued)

- Now we fill in row 5

			w	o	r	l	d
		0	1	2	3	4	5
	0	0	0	0	0	0	0
w	1	0	1	1	1	1	1
i	2	0	1	1	1	1	1
r	3	0	1	1	2	2	2
e	4	0	1	1	2	2	2
d	5	0	1	1	2		

LCSP Example (continued)

- Now we fill in row 5

			w	o	r	l	d
		0	1	2	3	4	5
	0	0	0	0	0	0	0
w	1	0	1	1	1	1	1
i	2	0	1	1	1	1	1
r	3	0	1	1	2	2	2
e	4	0	1	1	2	2	2
d	5	0	1	1	2	2	

LCSP Example (continued)

- As we fill in the final cell, at position (5, 5), the two ‘d’s match, so this becomes M[4, 4] + 1 = 3

			w	o	r	l	d
		0	1	2	3	4	5
	0	0	0	0	0	0	0
w	1	0	1	1	1	1	1
i	2	0	1	1	1	1	1
r	3	0	1	1	2	2	2
e	4	0	1	1	2	2	2
d	5	0	1	1	2	2	3

LCSP Example (continued)

- We are done
- The length of the longest common subsequence of "world“ and "wired" is 3, the value of M[5, 5]

			w	o	r	l	d
		0	1	2	3	4	5
	0	0	0	0	0	0	0
w	1	0	1	1	1	1	1
i	2	0	1	1	1	1	1
r	3	0	1	1	2	2	2
e	4	0	1	1	2	2	2
d	5	0	1	1	2	2	3

Retrieving the Longest Common Subsequence

- We can also use the matrix to recover the longest common subsequence itself in time that is O(m+n)
- This algorithm applies assuming that M has already been filled by the previous algorithm
- You start at the bottom right of the matrix, M; that is, at position (n, m)
- You will move up or left or both after each iteration and stop when you hit the top row or left column
- If, at position (i, j), it is the case that X_i == Y_j, then this character is part of the subsequence
- Then, we will append the character to the subsequence, and we move up and to the left
- Otherwise, you need to decide if you move up one row or left one column
- We move to the choice that has the larger value, breaking ties by moving up (because that's what the previous algorithm did)
- This can be implemented recursively or iteratively

Retrieving the Subsequence Pseudo-code

```
LongestCommonSubsequence ( String X, String Y, Matrix M )
  i ← length(X)
  j ← length(Y)
  subsequence ← empty string
  while i > 0 and j > 0
    if X_i == Y_j
      subsequence ← X_i + subsequence
      i ← i - 1
      j ← j - 1
    else if M[i-1, j] ≥ M[i, j-1]
      i ← i - 1
    else
      j ← j - 1
```

Retrieving the Subsequence Example

- The colored slots are the ones traversed by the algorithm
- Blue slots represent non-matches (we move up or left)
- Red slots represent matches (the letters are part of the subsequence)

			w	o	r	l	d
		0	1	2	3	4	5
	0	0	0	0	0	0	0
w	1	0	1	1	1	1	1
i	2	0	1	1	1	1	1
r	3	0	1	1	2	2	2
e	4	0	1	1	2	2	2
d	5	0	1	1	2	2	3

The Submarine/Number-line Puzzle

We’ll discuss a solution next week!

Randomized Algorithms

- The next strategy we will discuss is **randomized algorithms**; i.e., algorithms that rely on randomness
- Strictly speaking, though, modern-day computers are incapable of true randomization
- According to Papadimitriou's "Computational Complexity", I am not correct; he writes, "In some very real sense, computation is inherently randomized."
- His reasoning is unusual: "It can be argued that the probability that a computer will be destroyed by a meteorite during any given microsecond of its operation is at least 2^{-100}."
- I disagree with this particular reasoning (it was probably a joke)
- However, perhaps it can be argued that, due to quantum physics, one cannot really be absolutely certain what a computer will do
- However, when dealing with a theoretical computer, true randomness is not possible; this will become clearer during our final topic

Pseudo-random Number Generators

- Of course, many applications must appear to exhibit random behavior (e.g., games)
- Interestingly, randomization can also be useful to implement certain algorithms with efficient expected running times (we will soon see an example)
- All modern programming languages include routines to generate **pseudo-random** numbers
- Often, you will hear these referred to as *random number generators* (I will call them that also), but it would be more accurate to call them *pseudo-random number generators*
- Most modern programming languages offer some built in functionality to produce pseudo-random numbers
- In C and C++, you can use the "rand" function, for example, which returns a pseudo-random integer from 0 to RAND_MAX (inclusive)
- Modern versions of C++ also provide classes that offer more flexible abilities, and that produce pseudo-random numbers with better properties
- Some languages offer a function that returns a pseudo-random fraction from 0 to 1 (typically not inclusive of 1) instead
- Either method can be used to produce values in other ranges
- For example, to produce a pseudo-random integer in the range of MIN to MAX (inclusive), you can use: `rand() % (MAX-MIN+1) + MIN`

Seeds

- We will soon see that random number generators rely on a global variable that changes each time it is used
- The initial value of this global variable is called a **seed** (we can also use this as a verb; i.e., we can seed the random number generator)
- C and C++ provide a function called "srand" which allows us to set the seed; for example: `srand(12345);`
- The entire sequence of pseudo-random numbers that are generated is determined based on this seed in an unpredictable way
- If the seed were actually random, this would be all that was necessary; but this is impossible
- Fortunately, all that matters is that the seed is different every time
- Therefore, a common solution is to seed the random number generator with a value representing the date and time
- For example, C and C++ provide a function called "time", which returns a number representing the date and time down to the nearest second
- We can use this to choose a useful seed; for example: `srand(time(nullptr));`
- Some programming languages automatically seed the random number generator in a similar way
- Note that we should *only seed the pseudo-random number generator once*, *at the start of the program*

Pseudo-random Number Generator Example

- The textbook discusses and shows code for a pseudo-random number generator
- We will look at a simpler one, shown to the right
- This is from "The C Programming Language" by Kernighan and Ritchie (K&R)
- Note that srand simply sets a global variable to the specified seed
- The rand function updates the global and uses it to produce a pseudo-random number
- I have heard that this is not a particularly good pseudo-random number generator
- Even if rand returns the same value twice, this does not mean that the sequence will repeat
- However, if the value of `next` repeats, then the sequence starting at that point will repeat

```
unsigned long int next = 1;

int rand(void)
{
  next = next * 1103515245 + 12345;
  return (unsigned int) (next/65536) % 32768;
}

void srand(unsigned int seed)
{
  next = seed;
}
```

Good Pseudo-random Number Generators

- There are several desired properties for pseudo-random number generators; for example:
  - The period of numbers before repetition should be long
  - Sequences of generated numbers should be uniformly distributed
  - Subsequences of generated numbers should be uncorrelated
  - The generator should be efficient
- If the first three properties are not present, the result of a program relying on the generator might be too predictable

Examples of Randomized Algorithms

- The book discusses two examples of randomized algorithms (we will not cover these in detail)
- The first example involves a data structure called *skip lists*
  - This algorithm involves a linked lists, where some nodes contain more than link (the specific number is chosen pseudo-randomly), each pointing a different number of nodes ahead
  - The data structure can support insertion and search in expected O(log N) time
  - Note that this is a bit different than average-case O(log N) time
  - Expected time depends on the random numbers, where as average-case depends on the input
  - For some algorithms, skip-lists can be more efficient than balanced binary search trees
- The second example a randomized algorithm to test if a number is *prime*
  - If the algorithm says no, then the number is definitely not prime
  - If the algorithm says yes, then the number of prime with high probability
  - By repeating the algorithm multiple times, we can reduce the probability of an error to something negligible
  - Interestingly, there is a known algorithm to definitively test if a number is prime in polynomial time
  - On average, the randomized algorithm is faster
- In DSA 1, we discussed the possibility of using random pivots for quicksort
- Many games rely on pseudo-randomness for various purposes

The 8-Queens Problem

- We are going to discuss an example that comes up fairly early in my Artificial Intelligence (AI) course
- The **8-queens problem** challenges us to place 8 queens on a chessboard such that no two queens are attacking each other
- A brute force approach would have to check $64*63*\ldots*57 \approx 1.8*10^{14}$ possible sequences
- A smarter startegy would recognize that one queen has to be placed in each column, and it can backtrack as soon as two queens attack each other
- Then there are only 2057 sequences (not all lead to solutions)
- This sort of search is simple on a modern computer
- We are going to look at a randomized algorithm that does a better job scaling up to the more general N-queens problem
- The solution that we will cover is also an example of a greedy algorithm!

Hill Climbing

- **Hill climbing** (a.k.a. *greedy local search*) is a search strategy that iteratively replaces the current state with its best neighboring state
- When no neighbor is better than the current state, the search is finished
- We will apply hill-climbing to the 8-queens problem
  - Using our knowledge of the problem, we can start by randomly placing one queen in each column
  - At each step, we will allow movements of any queen to any other square in the same column; therefore, there will always be 56 possible moves or actions
  - The "best neighbor" is the state which has the smallest number of pairs of queens attacking each other
  - Optionally, if no move improves the situation, we can allow *sideways moves*, which may allow us to improve the situation after multiple moves

Hill Climbing for 8-queens Problem Example

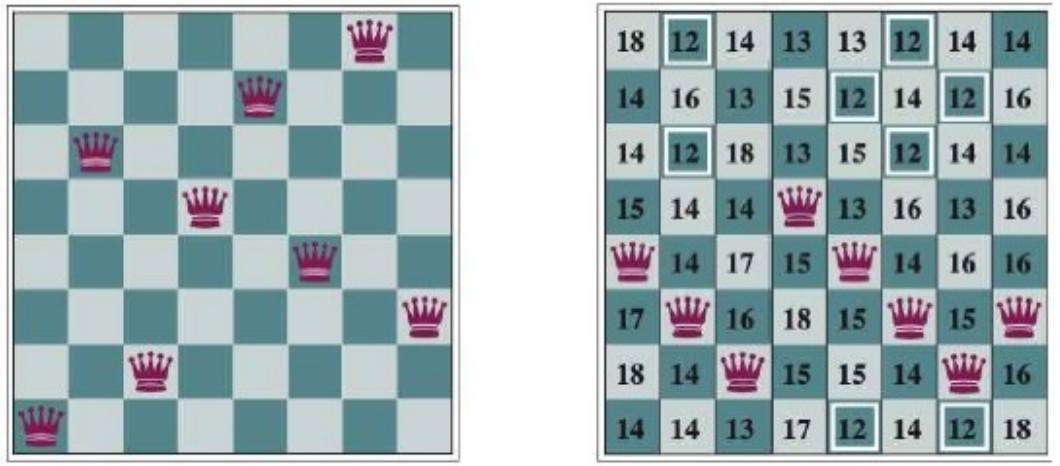


Figure 4.3 (a) The 8-queens problem: place 8 queens on a chess board so that no queen attacks another. (A queen attacks any piece in the same row, column, or diagonal.) This position is almost a solution, except for the two queens in the fourth and seventh columns that attack each other along the diagonal. (b) An 8-queens state with heuristic cost estimate $h = 17$. The board shows the value of h for each possible successor obtained by moving a queen within its column. There are 8 moves that are tied for best, with $h = 12$. The hill-climbing algorithm will pick one of these.

Figure from "Artificial Intelligence: A Modern Approach, 4th Edition" by Russell and Norvig

Hill Climbing for 8-queens Problem Results

- When hill climbing is applied to the 8-queens problem without sideways moves:
  - There is only about a 14% chance of success (i.e., it gets stuck about 86% of the time)
  - When there is a success, it takes an average of about 4 moves
  - When it gets stuck, it takes an average of about 3 moves
- When hill climbing is applied to the 8-queens problem, and up to 100 sideways moves are allowed:
  - There is about a 94% chance of success (i.e., it gets stuck about 6% of the time)
  - When there is a success, it takes an average of about 21 moves
  - When it gets stuck, it takes an average of about 64 moves

Random-restart Hill Climbing

- Of course, we want an algorithm that can solve the 8-queens problem every time
- We can virtually guarantee this with **random-restart hill climbing**
- The approach is simple; apply hill climbing (with or without sideways moves); if it gets stuck, start over!
- If a single hill climbing search has a probability p of success, then the expected number of restarts is 1/p – 1
- The expected number of steps is the cost of a successful iteration plus 1/p – 1 = (1-p)/p times the cost of a failure
- The expected number of steps for the 8-queens problem is approximately 25 if up to 100 sideways moves are allowed and 22 moves if they are not
- According to the Russell and Norvig AI textbook (4^{th} edition), this solution can find a solution for up to three million queens in seconds!

The Submarine/Number-line Puzzle

- Imagine a number-line, infinitely long in both directions, with all integers demarcated
- Somewhere on that number-line, at some finite integer location, there exists a submarine
- The submarine has a finite, integer velocity that will never change
- Therefore, at every time unit, the position of the submarine will be some finite integer
- You are given an infinite supply of bombs!
- At every time unit, you may drop one bomb at one integer location
- In response, you will be told one of two things; either "you hit the submarine" or "you did not hit the submarine"
- The **submarine/number-line puzzle** asks you to come up with a strategy that is guaranteed to eventually hit the submarine

Exhaustive Search

- We are going to solve the submarine/number-line puzzle using an **exhaustive search**, a.k.a. *brute-force search*
- For previous problems we encountered, we may have mentioned an exhaustive search strategy before discussing a more efficient solution
- What makes exhaustive search interesting for the submarine/number-line puzzle is that it may seem like there is no solution at all
- First, we will formalize the problem a bit
  - Let s be the starting position of the submarine; we know s is a finite integer
  - Let v be the constant velocity of the submarine; we know v is a finite integer
  - We need a strategy that helps us choose where to drop a bomb at each time, t
  - We'll arbitrarily start t at 0, and count the number of time units that have passed
- If at any time, t, we drop a bomb at the position s + v*t, we will hit the submarine
- Therefore, we need a strategy that tries out all (s, v) pairs

Submarine/Number-line Puzzle Solution

- There are multiple ways we can iterate through all possible (s, v) pairs, where both s and v are integers
- For example, consider an x-y plane, where the x-axis represents s and the y-axis represent v
- Then start at the middle, and spiral out
- Alternatively, consider an infinitely big grid, where the rows represent values of s, and the columns represent values of v
- Order the rows and columns as follows: 0, 1, -1, 2, -2, 3, -3, etc.
- Then loop through one diagonal at a time, starting with the top left (0, 0); then (0, 1), (1, 0); then (0, -1), (1, 1), (-1, 0); etc.

Backtracking Search

- Our textbook does not specifically cover exhaustive search as an algorithm strategy
- However, they says that **backtracking search** "amounts to a clever implementation of exhaustive search, with generally unfavorable performance"
- The go on to explain that performance is relative, and for some tasks, there may not be more efficient choices
- What backtracking search adds to exhaustive search is pruning, meaning that you can sometimes skip possibilities that cannot lead to a solution
- For example, if we solved the 8-queens problem by placing one queen on the board at a time, without constraints, that would be exhaustive search
- If we stop whenever two queens attack each other, back up, and try another path, that would be backtracking search
- The book talks about two applications of backtracking search
- One is known as the turnpike reconstruction problem, which we will skip

Minimax Search

- The other example of backtracking search discussed in the textbook is **minimax search** with **alpha-beta pruning** for *game playing*
- This will be repetitive for those of you who have taken, or are taking, the AI course (and we will cover it in less depth now, of course)
- All figures from my slides for this subtopic come from the AI textbook (I like those figures better than the ones in the DSA book for this topic)
- Minimax search is a strategy that can be used for *deterministic*, *turn-taking*, *two-player*, *zero-sum games* of *perfect information*
- We will label the two players playing the game is MAX and MIN; only one can win (or the game can be a draw)
- We will start by considering a hypothetical **game tree** for a simple game
- In such a game tree, it is conventional to depict MAX with upward pointing triangles, and MIN with downward pointing triangles
- Then, we will consider at a game tree for tic-tac-toe, which is also the example game used in the DSA book; in such a game, it is possible to explore the entire game tree

Minimax Values Example

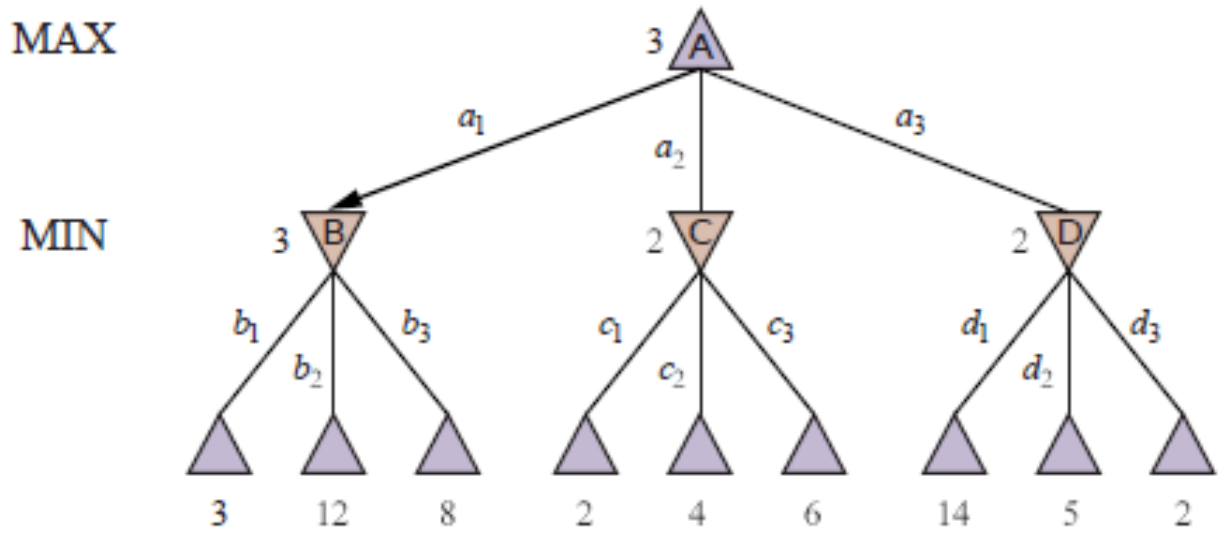


Figure 5.2 A two-ply game tree. The △ nodes are "MAX nodes," in which it is MAX's turn to move, and the ▽ nodes are "MIN nodes." The terminal nodes show the utility values for MAX; the other nodes are labeled with their minimax values. MAX's best move at the root is a_1, because it leads to the state with the highest minimax value, and MIN's best reply is b_1, because it leads to the state with the lowest minimax value.

Example Game Tree: Tic-Tac-Toe

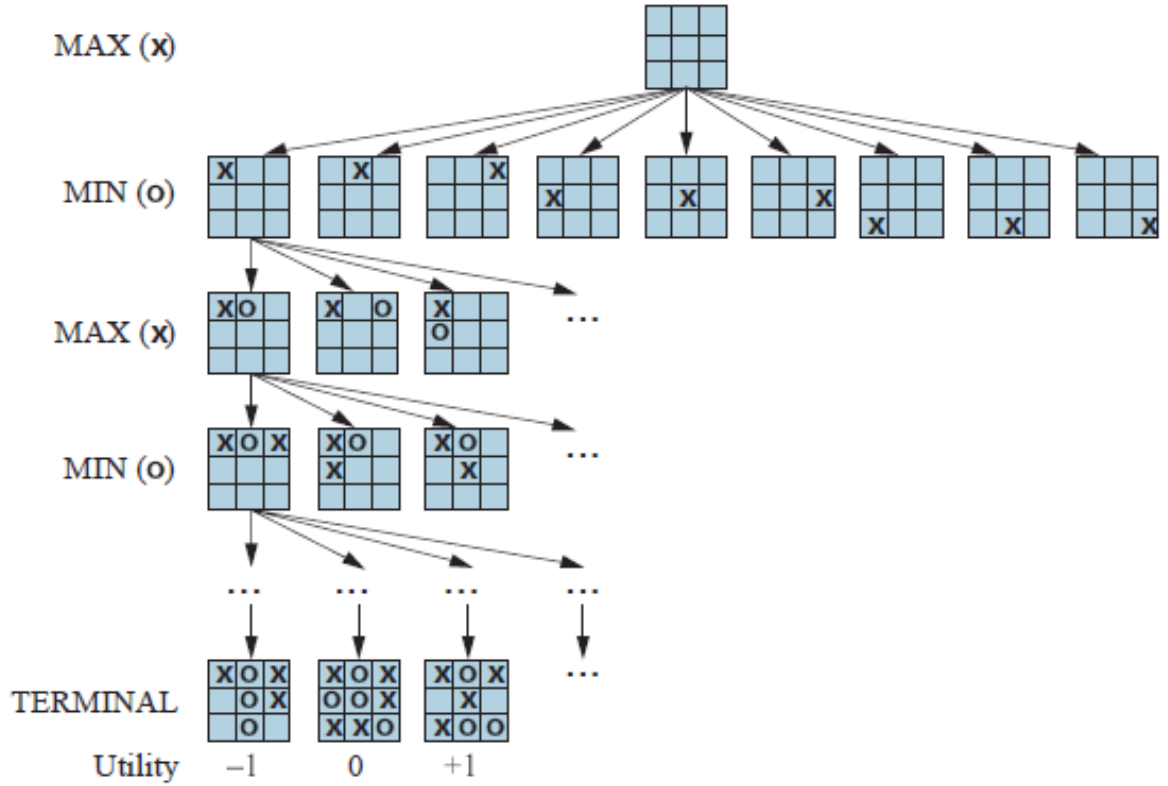


Figure 5.1 A (partial) game tree for the game of tic-tac-toe. The top node is the initial state, and MAX moves first, placing an X in an empty square. We show part of the tree, giving alternating moves by MIN (O) and MAX (X), until we eventually reach terminal states, which can be assigned utilities according to the rules of the game.

Calculating Minimax Values

- A minimax search computes the *minimax value* of each node in the game tree recursively, as follows:

```
MINIMAX (s) =
   UTILITY(s) if IS-TERMINAL(s); i.e., if s is a terminal state (leaf)
   max_{aεACTIONS(s)} MINIMAX(RESULT(s, a)) if TO-MOVE(s) = MAX
   min_{aεACTIONS(s)} MINIMAX(RESULT(s, a)) if TO-MOVE(s) = MIN
```

- Note that *the game tree is never actually stored as a data structure*; the links in the figure represent recursive function calls that are made recursively
- For a game like tic-tac-toe, the algorithm can proceed through entire game trees, all the way to the leaves, and calculate minimax values for every possible position
- The *minimax* decision (the move that is made) using this function maximizes the worst-case scenario
- The opponent may not make the move you expect, but if not, you will do at least as well as you predicted based on the minimax search

Minimax Search Complexity

- This algorithm, as we have covered it so far, performs a complete DFS of the game tree; thus, it is an example of an *exhaustive search*
- Assume the maximum depth of the game tree is m and there are at most b legal moves per state
- Then the time complexity of the search is $O(b^m)$
- The space requirement is O(b * m) if all moves are generated at once
- The space complexity can be reduced to O(m) if one legal move is generated at a time
- We can only search entire game trees for very simple games

Alpha-Beta Pruning

- A major problem with minimax search as described so far is that the number of game states is exponential with respect to the number of moves
- A search that adds **alpha-beta pruning** returns exactly the same move as a full minimax search; with alpha-beta pruning, this is a *backtracking algorithm*
- However, it does not consider, or even generate, the majority of the nodes in the game tree in most cases
- To implement this strategy, two new search parameters need to be added:
  - α = the value of the best choice for MAX (i.e., the highest value) along the current path
  - β = the value of the best choice for MIN (i.e., the lowest value) along the current path
- With these changes, the search is known as **alpha-beta search**, a.k.a. **minimax search with alpha-beta pruning**
- In practice, alpha-beta pruning can allow you to search 50% to 100% deeper in the tree in a given amount of time; this can greatly improve a program's play

Alpha-Beta Explanation

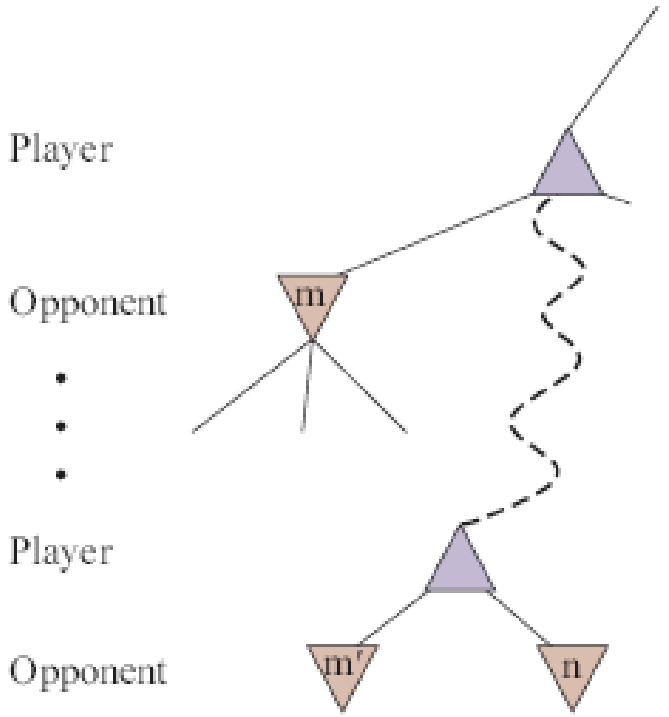
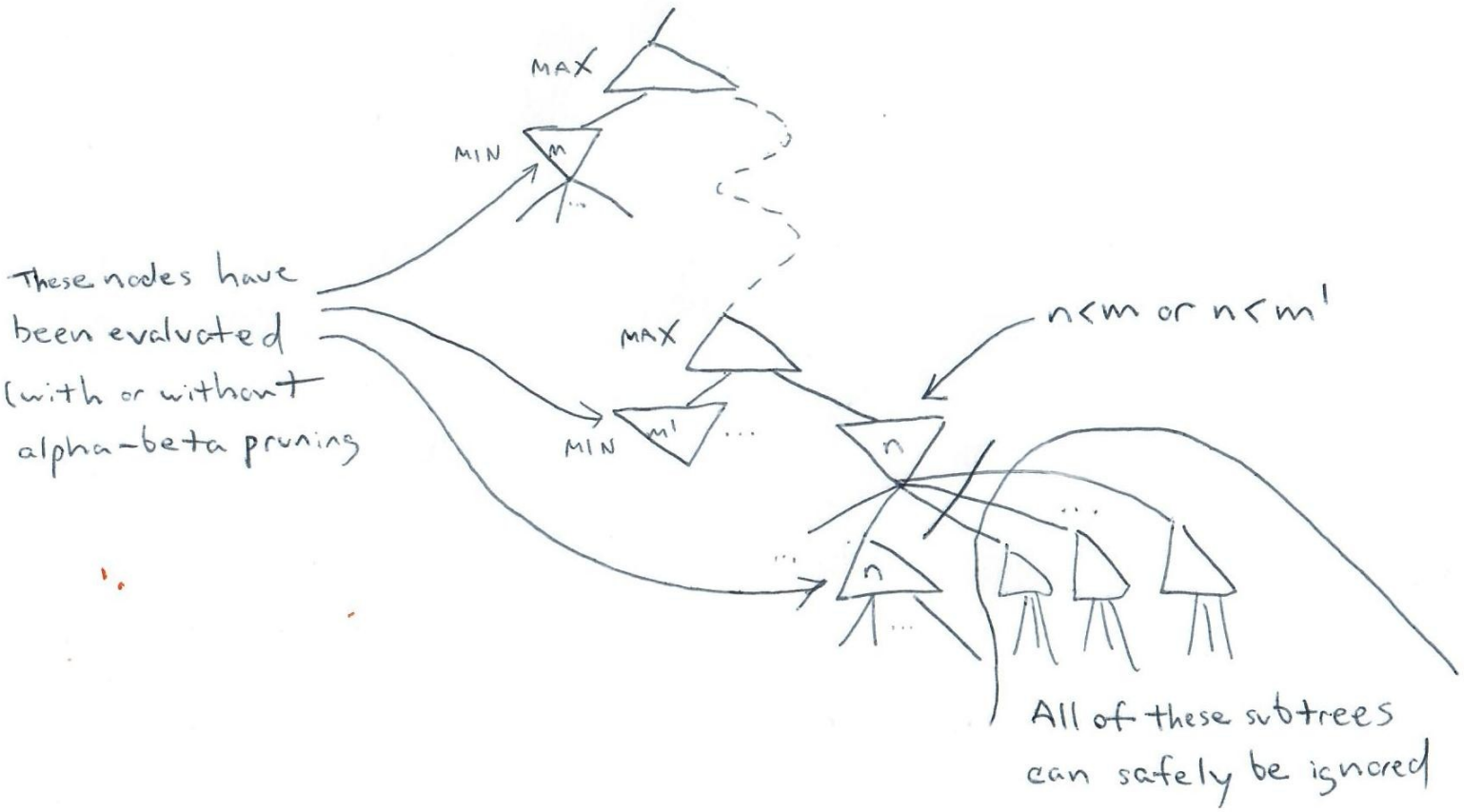


Figure 5.6 The general case for alpha–beta pruning. If m or m' is better than n for Player, we will never get to n in play.

Alpha-Beta Explanation Augmented



And The Rest…

- Even with alpha-beta pruning, it is not feasible to search entire game trees for games like chess, checkers, or Othello
- One possibility is to search to a specified depth limit, at which point an **evaluation function**, a.k.a. **heuristic function**, is applied
  - The evaluation function should order moves based on their goodness, and it must be zero-sum
  - No non-definite win should get a score as high as a definite win, and no non-definite loss should get a score as low as a definite loss
  - Conventionally, evaluation functions have been manually constructed, but in recent years, they have been learned for some games using reinforcement learning
- Alternatively, if each move has a specified time limit, *iterative deepening* can be applied
  - This means that there are sequential depth-limited searches (the previous idea) to depths 1, 2, 3, etc.
  - If the time runs out in the middle of one search, it stops, and the move chosen based on the previous depth-limited search is selected
- A *transposition table* is a hash table that stores evaluations of positions that have been searched up to a particular depth; this can be used to avoid redundant searches
- There are many other potential improvements